Red Hat Quay 3.7

Deploy Red Hat Quay on OpenShift with the Quay Operator

Deploy Red Hat Quay on OpenShift with Quay Operator
Deploy Red Hat Quay on OpenShift with Quay Operator
Abstract

Deploy Red Hat Quay on an OpenShift Cluster with the Red Hat Quay Operator
# Table of Contents

PREFAcE ................................................................. 6

CHAPTER 1. INTRODUCTION TO THE RED HAT QUAY OPERATOR .................................................. 7
  1.1. QUAYREGISTRY API ................................................ 7
  1.2. QUAY OPERATOR COMPONENTS .................................. 7
  1.3. USING MANAGED COMPONENTS ................................. 8
  1.4. USING UNMANAGED COMPONENTS FOR DEPENDENCIES ... 9
  1.5. CONFIG BUNDLE SECRET ........................................ 9
  1.6. PREREQUISITES FOR RED HAT QUAY ON OPENSIFT ....... 9
     1.6.1. OpenShift cluster ........................................ 9
     1.6.2. Resource Requirements .................................. 9
     1.6.3. Object Storage ........................................... 9

CHAPTER 2. INSTALLING THE QUAY OPERATOR FROM OPERATORHUB ........................................... 11

CHAPTER 3. CONFIGURING QUAY BEFORE DEPLOYMENT ................................................................. 14
  3.1. PRE-CONFIGURING QUAY FOR AUTOMATION .................. 14
     3.1.1. Allowing the API to create the first user ................ 14
     3.1.2. Enabling general API access ............................. 14
     3.1.3. Adding a super user ...................................... 15
     3.1.4. Restricting user creation .................................. 15
     3.1.5. Enabling new functionality .............................. 15
     3.1.6. Suggested configuration for automation ................. 15
     3.1.7. Deploying the Operator using the initial configuration 15
  3.2. CONFIGURING OBJECT STORAGE .............................. 16
     3.2.1. Unmanaged storage ....................................... 16
          3.2.1.1. AWS S3 storage ..................................... 16
          3.2.1.2. Google Cloud storage .............................. 16
          3.2.1.3. Azure storage ....................................... 17
          3.2.1.4. Ceph / RadosGW Storage / Hitachi HCP storage 17
          3.2.1.5. Swift storage ....................................... 17
          3.2.1.6. NooBaa unmanaged storage ......................... 18
     3.2.2. Managed storage .......................................... 18
          3.2.2.1. About The Standalone Object Gateway .......... 19
               3.2.2.1.1. Create A Standalone Object Gateway .. 19
  3.3. CONFIGURING THE DATABASE ................................. 21
     3.3.1. Using an existing Postgres database ................... 21
     3.3.2. Database configuration .................................. 22
          3.3.2.1. Database URI ....................................... 22
          3.3.2.2. Database connection arguments ................... 22
               3.3.2.2.1. PostgreSQL SSL connection arguments .... 22
               3.3.2.2.2. MySQL SSL connection arguments ........ 23
     3.3.3. Using the managed PostgreSQL .......................... 23
  3.4. CONFIGURING TLS AND ROUTES ............................. 24
     3.4.1. Creating the config bundle secret with TLS cert, key pair: 24
  3.5. CONFIGURING OTHER COMPONENTS .......................... 24
     3.5.1. Using external Redis ..................................... 24
          3.5.1.1. Redis configuration fields ........................ 25
               3.5.1.1.1. Build logs ................................... 25
               3.5.1.1.2. User events .................................. 26
          3.5.1.3. Example redis configuration ....................... 26
     3.5.2. Disabling the Horizontal Pod Autoscaler ............... 27
CHAPTER 4. DEPLOYING QUAY USING THE QUAY OPERATOR .............................................. 30
4.1. DEPLOYING RED HAT QUAY FROM THE COMMAND LINE ........................................... 30
   4.1.1. Viewing created components using the command line ............................................... 32
   4.1.2. Horizontal Pod Autoscaling (HPA) .............................................................................. 32
   4.1.3. Using the API to create the first user ............................................................................ 33
      4.1.3.1. Invoking the API .................................................................................................... 33
      4.1.3.2. Using the OAuth token ......................................................................................... 34
      4.1.3.2.1. Create organization ......................................................................................... 34
      4.1.3.2.2. Get organization details .................................................................................... 34
   4.1.4. Monitoring and debugging the deployment process ....................................................... 35
4.2. DEPLOYING RED HAT QUAY FROM THE OPENSHIFT CONSOLE ......................... 38
   4.2.1. Using the Quay UI to create the first user .................................................................... 38

CHAPTER 5. CONFIGURING QUAY ON OPENSHIFT ......................................................... 41
5.1. EDITING THE CONFIG BUNDLE SECRET IN THE OPENSHIFT CONSOLE ................. 41
5.2. DETERMINING QUAYREGISTRY ENDPOINTS AND SECRETS .............................. 42
5.3. DOWNLOADING THE EXISTING CONFIGURATION ..................................................... 43
5.4. USING THE CONFIG BUNDLE TO CONFIGURE CUSTOM SSL CERTS .................. 44
   5.4.1. Set TLS to unmanaged ............................................................................................... 44
   5.4.2. Add certs to config bundle ....................................................................................... 45

CHAPTER 6. USING THE CONFIG TOOL TO RECONFIGURE QUAY ON OPENSHIFT ...... 47
6.1. ACCESSING THE CONFIG EDITOR ...................................................................................... 47
   6.1.1. Retrieving the config editor credentials ...................................................................... 47
   6.1.2. Logging in to the config editor ................................................................................... 48
   6.1.3. Changing configuration ............................................................................................. 49
6.2. MONITORING RECONFIGURATION IN THE UI .............................................................. 50
   6.2.1. QuayRegistry resource ............................................................................................... 50
   6.2.2. Events ........................................................................................................................ 52
6.3. ACCESSING UPDATED INFORMATION AFTER RECONFIGURATION .................... 53
   6.3.1. Accessing the updated config tool credentials in the UI ............................................ 53
   6.3.2. Accessing the updated config.yaml in the UI .............................................................. 53
6.4. CUSTOM SSL CERTIFICATES UI ...................................................................................... 54
6.5. EXTERNAL ACCESS TO THE REGISTRY ....................................................................... 54

CHAPTER 7. QUAY OPERATOR FEATURES ........................................................................... 55
7.1. CONSOLE MONITORING AND ALERTING ................................................................. 55
   7.1.1. Dashboard ................................................................................................................... 55
   7.1.2. Metrics ....................................................................................................................... 56
   7.1.3. Alerting ....................................................................................................................... 58
7.2. MANUALLY UPDATING THE VULNERABILITY DATABASES FOR CLAIR IN AN AIR-GAPPED OPENSHIFT CLUSTER ............................................................. 58
   7.2.1. Obtaining clairctl ......................................................................................................... 59
   7.2.2. Retrieving the Clair config ........................................................................................ 59
      7.2.2.1. Clair on OpenShift config .................................................................................. 59
      7.2.2.2. Standalone Clair config ...................................................................................... 60
   7.2.3. Exporting the updaters bundle .................................................................................... 60
   7.2.4. Configuring access to the Clair database in the air-gapped OpenShift cluster .......... 60
   7.2.5. Importing the updaters bundle into the air-gapped environment ................................ 61
7.3. FIPS READINESS AND COMPLIANCE ........................................................................ 61
CHAPTER 8. ADVANCED CONCEPTS
8.1. DEPLOYING QUAY ON INFRASTRUCTURE NODES
8.1.1. Label and taint nodes for infrastructure use
8.1.2. Create a Project with node selector and toleration
8.1.3. Install the Quay Operator in the namespace
8.1.4. Create the registry
8.2. ENABLING MONITORING WHEN OPERATOR IS INSTALLED IN A SINGLE NAMESPACE
8.2.1. Creating a cluster monitoring config map
8.2.2. Creating a user-defined workload monitoring config map
8.2.3. Enable monitoring for user-defined projects
8.2.4. Create a Service object to expose Quay metrics
8.2.5. Create a ServiceMonitor object
8.2.6. View the metrics in OpenShift
8.3. RESIZING MANAGED STORAGE
8.3.1. Resize Noobaa PVC
8.3.2. Add Another Storage Pool
8.4. CUSTOMIZING DEFAULT OPERATOR IMAGES
8.4.1. Environment Variables
8.4.2. Applying Overrides to a Running Operator
8.5. AWS S3 CLOUDFRONT
8.5.1. Advanced Clair configuration
8.5.1.1. Unmanaged Clair configuration
8.5.1.1.1. Unmanaging a Clair database
8.5.1.1.2. Configuring a custom Clair database
8.5.1.2. Running a custom Clair configuration with a managed database

CHAPTER 9. RED HAT QUAY BUILD ENHANCEMENTS
9.1. RED HAT QUAY ENHANCED BUILD ARCHITECTURE
9.2. RED HAT QUAY BUILD LIMITATIONS
9.3. CREATING A RED HAT QUAY BUILDERS ENVIRONMENT WITH OPENSHIFT
9.3.1. OpenShift TLS component
9.3.2. Using OpenShift Container Platform for Red Hat Quay builders
9.3.2.1. Preparing OpenShift Container Platform for virtual builders
9.3.2.2. Manually adding SSL certificates.
9.3.2.2.1. Create and sign certs
9.3.2.2.2. Set TLS to unmanaged
9.3.2.2.3. Create temporary secrets
9.3.2.2.4. Copy secret data to config.yaml
9.3.2.3. Using the UI to create a build trigger
9.3.2.4. Modifying your AWS S3 storage bucket

CHAPTER 10. GEO-REPLICATION
10.1. GEO-REPLICATION FEATURES
10.2. GEO-REPLICATION REQUIREMENTS AND CONSTRAINTS
10.3. GEO-REPLICATION USING THE RED HAT QUAY OPERATOR
10.3.1. Setting up geo-replication on Openshift
10.3.1.1. Configuration
10.3.2. Mixed storage for geo-replication

CHAPTER 11. UPGRADING THE QUAY OPERATOR OVERVIEW
11.1. OPERATOR LIFECYCLE MANAGER
11.2. UPGRADING THE QUAY OPERATOR
11.2.1. Upgrading Quay
11.2.2. Notes on upgrading directly from 3.3.z or 3.4.z to 3.6
11.2.2.1. Upgrading with edge routing enabled
11.2.2.2. Upgrading with custom TLS certificate/key pairs without Subject Alternative Names
11.2.2.3. Configuring Clair v4 when upgrading from 3.3.z or 3.4.z to 3.6 using the Quay Operator
11.2.3. Swift configuration when upgrading from 3.3.z to 3.6
11.2.4. Changing the update channel for an Operator
11.2.5. Manually approving a pending Operator upgrade

11.3. UPGRADING A QUAYREGISTRY

11.4. ENABLING FEATURES IN QUAY 3.7
11.4.1. Quota management configuration
11.4.2. Using Red Hat Quay to proxy a remote organization configuration
11.4.3. Red Hat Quay build enhancements
11.4.4. Geo-replication using the Red Hat Quay Operator

11.5. ENABLING FEATURES IN QUAY 3.6
11.5.1. Console monitoring and alerting
11.5.2. OCI and Helm support

11.6. UPGRADING A QUAYECOSYSTEM
11.6.1. Reverting QuayEcosystem Upgrade
11.6.2. Supported QuayEcosystem Configurations for Upgrades

ADDITIONAL RESOURCES
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 Red Hat Quay on an OpenShift cluster.

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, caches, object storage, etc) supported for production use (some components may not be highly available)
- A new robust validation library for Quay’s configuration which is shared by the Quay application and config tool for consistency
- Object storage can now be managed by the Operator using the ObjectBucketClaim Kubernetes API (Red Hat OpenShift Data Foundation can be used to provide a supported implementation of this API on OpenShift)
- Customization of the container images used by deployed pods for testing and development scenarios
CHAPTER 1. INTRODUCTION TO THE RED HAT QUAY OPERATOR

This document outlines the steps for configuring, deploying, managing and upgrading Red Hat Quay on OpenShift using the Red Hat Quay Operator.

It shows you how to:

- Install the Red Hat Quay Operator
- Configure object storage, either managed or unmanaged
- Configure other unmanaged components, if required, including database, Redis, routes, TLS, etc.
- Deploy the Red Hat Quay registry on OpenShift using the Operator
- Use advanced features supported by the Operator
- Upgrade the registry by upgrading the Operator

1.1. 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.

1.2. QUAY OPERATOR COMPONENTS

Quay is a powerful container registry platform and as a result, has a significant 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:

```json
spec:
  components:
  - kind: quay
    managed: true
  - kind: postgres
```
1.3. USING MANAGED COMPONENTS

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

- **quay**: Holds overrides for the Quay deployment, for example, environment variables and number of replicas. This component is new in Red Hat Quay 3.7 and cannot be set to unmanaged.

- **postgres**: For storing the registry metadata, uses a version of Postgres 10 from the Software Collections

- **clair**: Provides image vulnerability scanning

- **redis**: Handles Quay builder coordination and some internal logging

- **horizontalpodautoscaler**: Adjusts the number of Quay pods depending on memory/cpu consumption

- **objectstorage**: For storing image layer blobs, utilizes the ObjectBucketClaim Kubernetes API which is provided by Noobaa/RHOCS

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

- **mirror**: Configures repository mirror workers (to support optional repository mirroring)

- **monitoring**: Features include a Grafana dashboard, access to individual metrics, and alerting to notify for frequently restarting Quay pods

- **tls**: Configures whether Red Hat Quay or OpenShift handles TLS

- **clairpostgres**: Configures a managed Clair database

The Operator will handle any required configuration and installation work needed for Red Hat Quay to use the managed components. If the opinionated deployment performed by the Quay Operator is unsuitable for your environment, you can provide the Operator with unmanaged resources (overrides).
as described in the following sections.

1.4. USING UNMANAGED COMPONENTS FOR DEPENDENCIES

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.

NOTE

The Quay config editor can also be used to create or modify an existing config bundle and simplifies 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.

1.5. 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.

1.6. PREREQUISITES FOR RED HAT QUAY ON OPENSShift

Before you begin the deployment of Red Hat Quay Operator on OpenShift, you should consider the following.

1.6.1. 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.

1.6.2. 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.

1.6.3. 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. Red Hat OpenShift Data Foundation provides this API via its NooBaa component, which will be used in this example.
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 FROM OPERATORHUB

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

2. The Installation page outlines the features and prerequisites:
3. Select Install. The Operator Installation page appears.
4. The following choices are available for customizing the installation:

- **Update Channel**: Choose the update channel, for example, **stable-3.7** for the latest release.

- **Installation Mode**: Choose **All namespaces on the cluster** if you want the Operator to be available cluster-wide. Choose **A specific namespace on the cluster** if you want it deployed only within a single namespace. It is recommended that you install the Operator cluster-wide. If you choose a single namespace, the monitoring component will not be available by default.

- **Approval Strategy**: Choose to approve either automatic or manual updates. Automatic update strategy is recommended.

5. Select Install.

6. After a short time, you will see the Operator installed successfully in the Installed Operators page.
CHAPTER 3. CONFIGURING QUAY BEFORE DEPLOYMENT

The Operator can manage all the Red Hat Quay components when deploying on OpenShift, and this is the default configuration. Alternatively, you can manage one or more components externally yourself, where you want more control over the set up, and then allow the Operator to manage the remaining components.

The standard pattern for configuring unmanaged components is:

1. Create a `config.yaml` configuration file with the appropriate settings

2. Create a Secret using the configuration file

   $ oc create secret generic --from-file config.yaml=./config.yaml config-bundle-secret

3. Create a QuayRegistry YAML file `quayregistry.yaml`, identifying the unmanaged components and also referencing the created Secret, for example:

   ```
   quayregistry.yaml
   ```

   ```yaml
   apiVersion: quay.redhat.com/v1
   kind: QuayRegistry
   metadata:
     name: example-registry
     namespace: quay-enterprise
   spec:
     configBundleSecret: config-bundle-secret
     components:
       - kind: objectstorage
         managed: false
   ```

4. Deploy the registry using the YAML file:

   $ oc create -n quay-enterprise -f quayregistry.yaml

3.1. PRE-CONFIGURING QUAY FOR AUTOMATION

Quay has a number of configuration options that support automation. These options can be set before deployment, to minimize the need to interact with the user interface.

3.1.1. Allowing the API to create the first user

Set the config option `FEATURE_USER_INITIALIZE` to `true`, so that you can use the API `/api/v1/userinitialize` to create the first user. This API endpoint does not require authentication, unlike all other registry API calls which require an OAuth token which is generated by an OAuth application in an existing organization.

Once you have deployed Quay, you can use the API to create a user, for example, `quayadmin`, provided no other users have already been created. For more information, see the section on Creating the first user using the API.

3.1.2. Enabling general API access
Set the config option `BROWSER_API_CALLS_XHR_ONLY` to `false`, to allow general access to the Quay registry API.

### 3.1.3. Adding a super user

While you cannot create a user until after deployment, it is convenient to ensure that first user is an administrator with full permissions. It is easier to configure this in advance, using the `SUPER_USER` configuration object.

### 3.1.4. Restricting user creation

Once you have configured a super user, you can restrict the ability to create new users to the super user group. Set the `FEATURE_USER_CREATION` to `false` to restrict user creation.

### 3.1.5. Enabling new functionality

If you want to take advantage of the new functionality in Red Hat Quay 3.7, enable some or all of the following features:

```yaml
FEATURE_QUOTA_MANAGEMENT: true
FEATURE_BUILD_SUPPORT: true
FEATURE_PROXY_CACHE: true
FEATURE_STORAGE_REPLICATION: true
DEFAULT_SYSTEM_REJECT_QUOTA_BYTES: 10240000
```

### 3.1.6. Suggested configuration for automation

Create a `config.yaml` configuration file that includes the appropriate settings:

```
...,
FEATURE_USER_INITIALIZE: true
BROWSER_API_CALLS_XHR_ONLY: false
SUPER_USERS:
- quayadmin
FEATURE_USER_CREATION: false
...,
```

### 3.1.7. Deploying the Operator using the initial configuration

1. Create a Secret using the configuration file

   ```bash
   $ oc create secret generic -n quay-enterprise --from-file config.yaml=./config.yaml init-config-bundle-secret
   ```

2. Create a QuayRegistry YAML file `quayregistry.yaml`, identifying the unmanaged components and also referencing the created Secret, for example:

   ```yaml
   quayregistry.yaml
   ```
3. Deploy the registry:

```bash
$ oc create -n quay-enterprise -f quayregistry.yaml
```

4. Create the first user, **quayadmin**, using the API

### 3.2. Configuring Object Storage

You need to configure object storage before installing Red Hat Quay, irrespective of whether you are allowing the Operator to manage the storage or managing it yourself.

If you want the Operator to be responsible for managing storage, see the section on Managed storage for information on installing and configuring the NooBaa / RHOCs Operator.

If you are using a separate storage solution, set **objectstorage** as **unmanaged** when configuring the Operator. See the following section. **Unmanaged storage**, for details of configuring existing storage.

#### 3.2.1. Unmanaged storage

Some configuration examples for unmanaged storage are provided in this section for convenience. See the Red Hat Quay configuration guide for full details for setting up object storage.

##### 3.2.1.1. AWS S3 Storage

```yaml
DISTRIBUTED_STORAGE_CONFIG:
  s3Storage:
    - S3Storage
      - host: s3.us-east-2.amazonaws.com
        s3_access_key: ABCDEFGHIJKLMNOP
        s3_secret_key: OL3ABCDEFGHIJKLMNOP
        s3_bucket: quay_bucket
        storage_path: /datastorage/registry
DISTRIBUTED_STORAGE_DEFAULT_LOCATIONS: []
DISTRIBUTED_STORAGE_PREFERENCE:
  - s3Storage
```

##### 3.2.1.2. Google Cloud Storage

```yaml
DISTRIBUTED_STORAGE_CONFIG:
  googleCloudStorage:
    - GoogleCloudStorage
      - access_key: GOOGQIMFB3ABCDEFGHIJKLMNOP
        bucket_name: quay-bucket
        secret_key: FhDAYe2HeuAKfvZCAGyOioNaaRABCDEFGHIJKLMNOP
        storage_path: /datastorage/registry
```

Red Hat Quay 3.7 Deploy Red Hat Quay on OpenShift with the Quay Operator
3.2.1.3. Azure storage

DISTRIBUTED_STORAGE_CONFIG:
azureStorage:
  - AzureStorage
    azure_container: azure_container_here
    storage_path: /datastorage/registry
    azure_account_name: azure_account_name_here
    azure_account_key: azure_account_key_here
    sas_token: some/path/
    endpoint_url: https://[account-name].blob.core.usgovcloudapi.net

As of Red Hat Quay 3.7, you must use the Primary endpoint of your MAG Blob service. Using the Secondary endpoint of your MAG Blob service will result in the following error:

AuthenticationErrorDetail:Cannot find the claimed account when trying to GetProperties for the account whusc8-secondary.

3.2.1.4. Ceph / RadosGW Storage / Hitachi HCP storage

DISTRIBUTED_STORAGE_CONFIG:
radosGWStorage:
  - RadosGWStorage
    access_key: access_key_here
    secret_key: secret_key_here
    bucket_name: bucket_name_here
    hostname: hostname_here
    is_secure: 'true'
    port: '443'
    storage_path: /datastorage/registry

3.2.1.5. Swift storage

DISTRIBUTED_STORAGE_CONFIG:
swiftStorage:
  - SwiftStorage
    swift_user: swift_user_here
    swift_password: swift_password_here
    swift_container: swift_container_here
3.2.1.6. NooBaa unmanaged storage

Use the following procedure to deploy NooBaa as your unmanaged storage configuration.

Procedure

1. Create a NooBaa Object Bucket Claim in the {product-title} console by navigating to Storage → Object Bucket Claims.

2. Retrieve the Object Bucket Claim Data details, including the Access Key, Bucket Name, Endpoint (hostname), and Secret Key.

3. Create a config.yaml configuration file using the information for the Object Bucket Claim:

```yaml
DISTRIBUTED_STORAGE_CONFIG:
  default:
    - RHOCSSStorage
    - access_key: WmrXtSGk8B3nABCDEFGHIJKLMNOPQRSTUVWXYZ
      bucket_name: my-noobaa-bucket-claim-8b844191-dc6c-444e-9ea4-87ece0abcdef
      hostname: s3.openshift-storage.svc.cluster.local
      is_secure: true
      port: "443"
      secret_key: X9P5SDGJtmSuHFCMSLMbdNCMIUABCDEFGH+C5QD
      storage_path: /datastorage/registry
DISTRIBUTED_STORAGE_DEFAULT_LOCATIONS: []
DISTRIBUTED_STORAGE_PREFERENCE:
  - swiftStorage
```

For more information about configuring an Object Bucket Claim, see Object Bucket Claim.

3.2.2. Managed storage

If you want the Operator to manage object storage for Quay, your cluster needs to be capable of providing object storage via the ObjectBucketClaim API. Using the Red Hat OpenShift Data Foundation (ODF) Operator, there are two supported options available:

- A standalone instance of the Multi-Cloud Object Gateway backed by a local Kubernetes PersistentVolume storage
  - Not highly available
  - Included in the Quay subscription
  - Does not require a separate subscription for ODF
- A production deployment of ODF with scale-out Object Service and Ceph
Highly available

Requires a separate subscription for ODF

To use the standalone instance option, continue reading below. For production deployment of ODF, please refer to the official documentation.

NOTE

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.

3.2.2.1. About The Standalone Object Gateway

As part of a Red Hat Quay subscription, users are entitled to use the Multi-Cloud Object Gateway (MCG) component of the Red Hat OpenShift Data Foundation Operator (formerly known as OpenShift Container Storage Operator). This gateway component allows you to provide an S3-compatible object storage interface to Quay backed by Kubernetes PersistentVolume-based block storage. The usage is limited to a Quay deployment managed by the Operator and to the exact specifications of the MCG instance as documented below.

Since Red Hat Quay does not support local filesystem storage, users can leverage the gateway in combination with Kubernetes PersistentVolume storage instead, to provide a supported deployment. A PersistentVolume is directly mounted on the gateway instance as a backing store for object storage and any block-based StorageClass is supported.

By the nature of PersistentVolume, this is not a scale-out, highly available solution and does not replace a scale-out storage system like Red Hat OpenShift Data Foundation (ODF). Only a single instance of the gateway is running. If the pod running the gateway becomes unavailable due to rescheduling, updates or unplanned downtime, this will cause temporary degradation of the connected Quay instances.

3.2.2.1.1. Create A Standalone Object Gateway

To install the ODF (formerly known as OpenShift Container Storage) Operator and configure a single instance Multi-Cloud Gateway service, follow these steps:

1. Open the OpenShift console and select Operators → OperatorHub, then select the OpenShift Data Foundation Operator.

2. Select Install. Accept all default options and select Install again.

3. Within a minute, the Operator will install and create a namespace openshift-storage. You can confirm it has completed when the Status column is marked Succeeded.

When the installation of the ODF Operator is complete, you are prompted to create a storage system. Do not follow this instruction. Instead, create NooBaa object storage as outlined the following steps.

4. Create NooBaa object storage. Save the following YAML to a file called noobaa.yaml.

```yaml
apiVersion: noobaa.io/v1alpha1
```
kind: NooBaa
metadata:
  name: noobaa
  namespace: openshift-storage
spec:
  dbResources:
    requests:
      cpu: '0.1'
      memory: 1Gi
  dbType: postgres
  coreResources:
    requests:
      cpu: '0.1'
      memory: 1Gi

This will create a single instance deployment of the *Multi-cloud Object Gateway*.

5. Apply the configuration with the following command:

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

6. After a couple of minutes, you should see that the MCG instance has finished provisioning (*PHASE* column will be set to *Ready*):

```bash
$ 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
```

7. Next, configure a backing store for the gateway. Save the following YAML to a file called *noobaa-pv-backing-store.yaml*.

```yaml
noobaa-pv-backing-store.yaml

apiVersion: noobaa.io/v1alpha1
kind: BackingStore
metadata:
  finalizers:
    - noobaa.io/finalizer
  labels:
    app: noobaa
    name: noobaa-pv-backing-store
    namespace: openshift-storage
spec:
  pvPool:
    numVolumes: 1
    resources:
      requests:
        storage: 50Gi
    storageClass: STORAGE-CLASS-NABLE
    type: pv-pool
```
The overall capacity of the object storage service, adjust as needed

The **StorageClass** to use for the **PersistentVolumes** requested, delete this property to use the cluster default

8. Apply the configuration with the following command:

```bash
$ oc create -f noobaa-pv-backing-store.yaml
backingstore.noobaa.io/noobaa-pv-backing-store created
```

This creates the backing store configuration for the gateway. All images in Quay will be stored as objects through the gateway in a **PersistentVolume** created by the above configuration.

9. Finally, run the following command to make the **PersistentVolume** backing store the default for all **ObjectBucketClaims** issued by the Operator.

```bash
$ oc patch bucketclass noobaa-default-bucket-class --patch 
"{"spec":{"placementPolicy":
{"tiers": [{"backingStores": ["noobaa-pv-backing-store"]}]}
}}" --type merge -n openshift-storage
```

This concludes the setup of the **Multi-Cloud Object Gateway** instance for Red Hat Quay. Note that this configuration cannot be run in parallel on a cluster with Red Hat OpenShift Data Foundation installed.

### 3.3. CONFIGURING THE DATABASE

#### 3.3.1. Using an existing Postgres database

1. Create a configuration file `config.yaml` with the necessary database fields:

   ```yaml
   config.yaml:
   ```

   ```yaml
   DB_URI: postgs://test-quay-database:postgres@test-quay-database:5432/test-quay-database
   ```

2. Create a Secret using the configuration file:

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

3. Create a QuayRegistry YAML file `quayregistry.yaml` which marks the **postgres** component as unmanaged and references the created Secret:

   ```yaml
   quayregistry.yaml
   ```

   ```yaml
   spec:
   ```

   ```yaml
   configBundleSecret: config-bundle-secret
   ```
4. Deploy the registry as detailed in the following sections.

3.3.2. Database configuration

You configure the connection to the database using the required `DB_URI` field and optional connection arguments in the `DB_CONNECTION_ARGS` structure. Some key-value pairs defined under `DB_CONNECTION_ARGS` are generic while others are database-specific. In particular, SSL configuration depends on the database you are deploying, and examples for PostgreSQL and MySQL are given below.

3.3.2.1. Database URI

Table 3.1. Database URI

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB_URI (Required)</td>
<td>String</td>
<td>The URI for accessing the database, including any credentials</td>
</tr>
</tbody>
</table>

Example:

```
postgresql://quayuser:quaypass@quay-server.example.com:5432/quay
```

3.3.2.2. Database connection arguments

Table 3.2. Database connection arguments

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB_CONNECTIONARGS</td>
<td>Object</td>
<td>Optional connection arguments for the database, such as timeouts and SSL</td>
</tr>
<tr>
<td>.autorollback</td>
<td>Boolean</td>
<td>Whether to use thread-local connections</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Should <strong>ALWAYS</strong> be <strong>true</strong></td>
</tr>
<tr>
<td>.threadlocals</td>
<td>Boolean</td>
<td>Whether to use auto-rollback connections</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Should <strong>ALWAYS</strong> be <strong>true</strong></td>
</tr>
</tbody>
</table>

3.3.2.2.1. PostgreSQL SSL connection arguments
A sample PostgreSQL SSL configuration is given below:

```
DB_CONNECTION_ARGS:
  sslmode: verify-ca
  sslrootcert: /path/to/cacert
```

The `sslmode` option determines whether or with what priority a secure SSL TCP/IP connection will be negotiated with the server. There are six modes:

- **disable**: only try a non-SSL connection
- **allow**: first try a non-SSL connection; if that fails, try an SSL connection
- **prefer** (default): first try an SSL connection; if that fails, try a non-SSL connection
- **require**: only try an SSL connection. If a root CA file is present, verify the certificate in the same way as if `verify-ca` was specified
- **verify-ca**: only try an SSL connection, and verify that the server certificate is issued by a trusted certificate authority (CA)
- **verify-full**: only try an SSL connection, verify that the server certificate is issued by a trusted CA and that the requested server host name matches that in the certificate

More information on the valid arguments for PostgreSQL is available at [https://www.postgresql.org/docs/current/libpq-connect.html](https://www.postgresql.org/docs/current/libpq-connect.html).

### 3.3.2.2. MySQL SSL connection arguments

A sample MySQL SSL configuration follows:

```
DB_CONNECTION_ARGS:
  ssl:
    ca: /path/to/cacert
```


### 3.3.3. Using the managed PostgreSQL

Recommendations:

- 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.
3.4. CONFIGURING TLS AND ROUTES

Support for OpenShift Container Platform Edge-Termination Routes has been added by way of a new managed component, tls. This separates the route component from TLS and allows users to configure both separately. EXTERNAL_TLS_TERMINATION: true is the opinionated setting. Managed tls means that the default cluster wildcard cert is used. Unmanaged tls means that the user provided cert/key pair will be injected into the Route.

ssl.cert and ssl.key are now moved to a separate, persistent Secret, which ensures that the cert/key pair is not re-generated upon every reconcile. These are now formatted as edge routes and mounted to the same directory in the Quay container.

Multiple permutations are possible when configuring TLS and Routes, but the following rules apply:

- If TLS is **managed**, then route must also be **managed**
- If TLS is **unmanaged** then you must supply certs, either with the config tool or directly in the config bundle

The following table outlines the valid options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Route</th>
<th>TLS</th>
<th>Certs provided</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>My own load balancer handles TLS</td>
<td>Managed</td>
<td>Managed</td>
<td>No</td>
<td>Edge Route with default wildcard cert</td>
</tr>
<tr>
<td>Red Hat Quay handles TLS</td>
<td>Managed</td>
<td>Unmanaged</td>
<td>Yes</td>
<td>Passthrough route with certs mounted inside the pod</td>
</tr>
<tr>
<td>Red Hat Quay handles TLS</td>
<td>Unmanaged</td>
<td>Unmanaged</td>
<td>Yes</td>
<td>Certificates are set inside the quay pod but route must be created manually</td>
</tr>
</tbody>
</table>

**NOTE**

Red Hat Quay 3.7 does not support builders when TLS is managed by the Operator.

3.4.1. Creating the config bundle secret with TLS cert, key pair:

To add your own TLS cert and key, include them in the config bundle secret as follows:

```
$ oc create secret generic --from-file config.yaml=./config.yaml --from-file ssl.cert=./ssl.cert --from-file ssl.key=./ssl.key config-bundle-secret
```

3.5. CONFIGURING OTHER COMPONENTS

3.5.1. Using external Redis
If you wish to use an external Redis database, set the component as unmanaged in the `QuayRegistry` instance:

1. Create a configuration file `config.yaml` with the necessary redis fields:

   ```yaml
   BUILDLOGS_REDIS:
     host: quay-server.example.com
     password: strongpassword
     port: 6379

   USER_EVENTS_REDIS:
     host: quay-server.example.com
     password: strongpassword
     port: 6379
   ```

2. Create a Secret using the configuration file

   ```bash
   $ oc create secret generic --from-file config.yaml=./config.yaml config-bundle-secret
   ```

3. Create a QuayRegistry YAML file `quayregistry.yaml` which marks redis component as unmanaged and references the created Secret:

   ```yaml
   apiVersion: quay.redhat.com/v1
   kind: QuayRegistry
   metadata:
     name: example-registry
     namespace: quay-enterprise
   spec:
     configBundleSecret: config-bundle-secret
     components:
     - kind: redis
       managed: false
   ```

4. Deploy the registry

### 3.5.1.1. Redis configuration fields

#### 3.5.1.1.1. Build logs

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUILDLOGS_REDIS (Required)</td>
<td>Object</td>
<td>Redis connection details for build logs caching</td>
</tr>
<tr>
<td>.host (Required)</td>
<td>String</td>
<td>The hostname at which Redis is accessible</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Example: <code>quay-server.example.com</code></td>
</tr>
</tbody>
</table>
### 3.5.1.1.2. User events

#### Table 3.5. User events config

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER_EVENTS_REDIS</td>
<td>Object</td>
<td>Redis connection details for user event handling</td>
</tr>
<tr>
<td>.host</td>
<td>String</td>
<td>The hostname at which Redis is accessible</td>
</tr>
<tr>
<td>Example:</td>
<td></td>
<td>quay-server.example.com</td>
</tr>
<tr>
<td>.port</td>
<td>Number</td>
<td>The port at which Redis is accessible</td>
</tr>
<tr>
<td>Example: 6379</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.password</td>
<td>String</td>
<td>The port at which Redis is accessible</td>
</tr>
<tr>
<td>Example: strongpassword</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 3.5.1.1.3. Example redis configuration

```
BUILDLOGS_REDIS:
    host: quay-server.example.com
    password: strongpassword
    port: 6379
```
### 3.5.2. Disabling the Horizontal Pod Autoscaler

*HorizontalPodAutoscalers* have been added to the Clair, Quay, and Mirror pods, so that they now automatically scale during load spikes.

As HPA is configured by default to be managed, the number of pods for Quay, Clair and repository mirroring is set to two. This facilitates the avoidance of downtime when updating / reconfiguring Quay via the Operator or during rescheduling events.

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

```yaml
apiVersion: quay.redhat.com/v1
kind: QuayRegistry
metadata:
  name: example-registry
  namespace: quay-enterprise
spec:
  components:
    - kind: horizontalpodautoscaler
      managed: false
```

### 3.5.3. Disabling Route Component

To prevent the Operator from creating a *Route*:

1. Mark the component as unmanaged in the *QuayRegistry*:

```yaml
apiVersion: quay.redhat.com/v1
kind: QuayRegistry
metadata:
  name: example-registry
  namespace: quay-enterprise
spec:
  components:
    - kind: route
      managed: false
```

2. Specify that you want Quay to handle TLS in the configuration, by editing the *config.yaml* file:

```yaml
EXTERNAL_TLS_TERMINATION: false
```

```yaml
SERVER_HOSTNAME: example-registry-quay-quay-enterprise.apps.user1.example.com
```

```yaml
PREFERRED_URL_SCHEME: https
```
If you do not configure the unmanaged Route correctly, you will see an error similar to the following:

```json
{
  "kind":"QuayRegistry",
  "namespace":"quay-enterprise",
  "name":"example-registry",
  "uid":"d5879ba5-cc92-406c-ba62-8b19cf56d4aa",
  "apiVersion":"quay.redhat.com/v1",
  "resourceVersion":"2418527"
},
  "reason":"ConfigInvalid",
  "message":"required component `route` marked as unmanaged, but `configBundleSecret` is missing necessary fields"
}
```

**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.

### 3.5.4. Unmanaged monitoring

If you install the Quay Operator in a single namespace, the monitoring component is automatically set to ‘unmanaged’. To enable monitoring in this scenario, see the section **Section 8.2, “Enabling monitoring when Operator is installed in a single namespace”**.

To disable monitoring explicitly:

```yaml
apiVersion: quay.redhat.com/v1
kind: QuayRegistry
metadata:
  name: example-registry
  namespace: quay-enterprise
spec:
  components:
    - kind: monitoring
      managed: false
```

### 3.5.5. Unmanaged mirroring

To disable mirroring explicitly:

```yaml
apiVersion: quay.redhat.com/v1
kind: QuayRegistry
metadata:
  name: example-registry
  namespace: quay-enterprise
spec:
  components:
    - kind: mirroring
      managed: false
```
CHAPTER 4. DEPLOYING QUAY USING THE QUAY OPERATOR

The Operator can be deployed from the command line or from the OpenShift console, but the fundamental steps are the same.

4.1. DEPLOYING RED HAT QUAY FROM THE COMMAND LINE

1. Create a namespace, for example, `quay-enterprise`.

2. Create a secret for the config bundle, if you want to pre-configure any aspects of the deployment.

3. Create a `QuayRegistry` custom resource in a file called `quayregistry.yaml`
   a. For a minimal deployment, using all the defaults:

   ```yaml
   quayregistry.yaml:
   apiVersion: quay.redhat.com/v1
   kind: QuayRegistry
   metadata:
     name: example-registry
     namespace: quay-enterprise
   ```

   b. If you want to have some components unmanaged, add this information in the `spec` field. For example, a minimal deployment might look like:

   ```yaml
   quayregistry.yaml:
   apiVersion: quay.redhat.com/v1
   kind: QuayRegistry
   metadata:
     name: example-registry
     namespace: quay-enterprise
   spec:
     components:
     - kind: clair
       managed: false
     - kind: horizontalpodautoscaler
       managed: false
     - kind: mirror
       managed: false
     - kind: monitoring
       managed: false
   ```

   c. If you have created a config bundle, for example, `init-config-bundle-secret`, reference it in the `quayregistry.yaml` file:

   ```yaml
   quayregistry.yaml:
   apiVersion: quay.redhat.com/v1
   kind: QuayRegistry
   metadata:
   ```
d. If you have a proxy configured, you can add the information using overrides for Quay, Clair, and mirroring:

**quayregistry.yaml:**

```yaml
kind: QuayRegistry
metadata:
  name: quay37
spec:
  configBundleSecret: config-bundle-secret
components:
  - kind: objectstorage
    managed: false
  - kind: route
    managed: true
  - kind: mirror
    managed: true
overrides:
  env:
    - name: DEBUGLOG
      value: "true"
    - name: HTTP_PROXY
      value: quayproxy.qe.devcluster.openshift.com:3128
    - name: HTTPS_PROXY
      value: quayproxy.qe.devcluster.openshift.com:3128
    - name: NO_PROXY
      value: svc.cluster.local,localhost,quay370.apps.quayperf370.perfscale.devcluster.openshift.com
  - kind: tls
    managed: false
  - kind: clair
    managed: true
overrides:
  env:
    - name: HTTP_PROXY
      value: quayproxy.qe.devcluster.openshift.com:3128
    - name: HTTPS_PROXY
      value: quayproxy.qe.devcluster.openshift.com:3128
    - name: NO_PROXY
      value: svc.cluster.local,localhost,quay370.apps.quayperf370.perfscale.devcluster.openshift.com
  - kind: quay
    managed: true
overrides:
  env:
    - name: DEBUGLOG
      value: "true"
    - name: NO_PROXY
      value: svc.cluster.local,localhost,quay370.apps.quayperf370.perfscale.devcluster.openshift.com
    - name: HTTP_PROXY
      value: quayproxy.qe.devcluster.openshift.com:3128
```
4. Create the **QuayRegistry** in specified namespace:

   ```
   $ oc create -n quay-enterprise -f quayregistry.yaml
   ```

5. See the section **Monitoring and debugging the deployment process** for information on how to track the progress of the deployment.

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

   ```
   $ oc get quayregistry -n quay-enterprise example-registry -o jsonpath="{.status.registryEndpoint}" -w
   ```

### 4.1.1. Viewing created components using the command line

Use the **oc get pods** command to view the deployed components:

   ```
   $ oc get pods -n quay-enterprise
   ```

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>example-registry-clair-app-5ffc9f77d6-jwr9s</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>3m42s</td>
</tr>
<tr>
<td>example-registry-clair-app-5ffc9f77d6-wgp7d</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>3m41s</td>
</tr>
<tr>
<td>example-registry-clair-postgres-54956d6d9c-rgs8l</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>3m5s</td>
</tr>
<tr>
<td>example-registry-quay-app-79c6b86c7b-8qnr2</td>
<td>1/1</td>
<td>Running</td>
<td>4</td>
<td>3m42s</td>
</tr>
<tr>
<td>example-registry-quay-app-79c6b86c7b-xk85f</td>
<td>1/1</td>
<td>Running</td>
<td>4</td>
<td>3m41s</td>
</tr>
<tr>
<td>example-registry-quay-app-upgrade-5kl5r</td>
<td>0/1</td>
<td>Completed</td>
<td>4</td>
<td>3m50s</td>
</tr>
<tr>
<td>example-registry-quay-config-editor-597b47c995-svqrl</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>3m42s</td>
</tr>
<tr>
<td>example-registry-quay-database-b466fc4d7-tfmx</td>
<td>1/1</td>
<td>Running</td>
<td>2</td>
<td>3m42s</td>
</tr>
<tr>
<td>example-registry-quay-mirror-6d9bd78756-6lj6p</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>2m58s</td>
</tr>
<tr>
<td>example-registry-quay-mirror-6d9bd78756-bv6gq</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>2m58s</td>
</tr>
<tr>
<td>example-registry-quay-postgres-init-dzbxmx</td>
<td>0/1</td>
<td>Completed</td>
<td>0</td>
<td>3m43s</td>
</tr>
<tr>
<td>example-registry-quay-redis-8bd67b647-sksgx</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>3m42s</td>
</tr>
</tbody>
</table>

### 4.1.2. Horizontal Pod Autoscaling (HPA)

A default deployment shows the following running pods:

- Two pods for the Quay application itself (**example-registry-quay-app-***)  
- One Redis pod for Quay logging (**example-registry-quay-redis-***)  
- One database pod for PostgreSQL used by Quay for metadata storage (**example-registry-quay-database-***)  
- One pod for the Quay config editor (**example-registry-quay-config-editor-***)  
- Two Quay mirroring pods (**example-registry-quay-mirror-***)  
- Two pods for the Clair application (**example-registry-clair-app-***)  
- One PostgreSQL pod for Clair (**example-registry-clair-postgres-***)
As HPA is configured by default to be managed, the number of pods for Quay, Clair and repository mirroring is set to two. This facilitates the avoidance of downtime when updating / reconfiguring Quay via the Operator or during rescheduling events.

4.1.3. Using the API to create the first user

When using the API to create the first user, the following conditions must be met:

- The config option FEATURE_USER_INITIALIZE must be set to true

- No users can already exist in the database

For more information on pre-configuring the deployment, see the section Pre-configuring Quay for automation

4.1.3.1. Invoking the API

Using the status.registryEndpoint URL, invoke the /api/v1/user/initialize API, passing in the username, password and email address. You can also request an OAuth token by specifying "access_token": true.

```bash
$ curl -X POST -k https://example-registry-quay-quay-enterprise.apps.docs.quayteam.org/api/v1/user/initialize --header 'Content-Type: application/json' --data '{ "username": "quayadmin", "password": "quaypass123", "email": "quayadmin@example.com", "access_token": true}'

{ "access_token": "6B4QTRSTSD1HMIG915VPX7BMEZBVB9GPNY2FC2ED", "email": "quayadmin@example.com", "encrypted_password": "1nZMLH57RIE5UGdL/yYpDOHLqiNCgi mb6W9kfF8MJZ1xfDpRyRs9NUNuUuAItW", "username": "quayadmin"}
```

If successful, the method returns an object with the username, email and encrypted password. If a user already exists in the database, an error is returned:

```bash
$ curl -X POST -k https://example-registry-quay-quay-enterprise.apps.docs.quayteam.org/api/v1/user/initialize --header 'Content-Type: application/json' --data '{ "username": "quayuser2", "password": "quaypass123", "email": "quayuser2@example.com"}'

{"message": "Cannot initialize user in a non-empty database"}
```

The password must be at least 8 characters and contain no whitespace:

```bash
$ curl -X POST -k https://example-registry-quay-quay-enterprise.apps.docs.quayteam.org/api/v1/user/initialize --header 'Content-Type: application/json' --data '{ "username": "quayadmin", "password": "pass123", "email": "quayadmin@example.com"}'
```
4.1.3.2. Using the OAuth token

You can now invoke the rest of the Quay API specifying the returned OAuth code. For example, to get a list of the current users:

```bash
```

In this instance, the details for the quayadmin user are returned as it is the only user that has been created so far.

4.1.3.2.1. Create organization

To create an organization, use a POST call to api/v1/organization/ endpoint:

```bash
$ curl -X POST -k --header 'Content-Type: application/json' -H "Authorization: Bearer 6B4QTRSTSD1HMIG915VPX7BMEZBV9GPNY2FC2ED" https://example-registry-quay-quay-enterprise.apps.docs.quayteam.org/api/v1/organization/ --data '{"name": "testorg", "email": "testorg@example.com"}'
```

"Created"

4.1.3.2.2. Get organization details

To retrieve the details of the organization you created:
4.1.4. Monitoring and debugging the deployment process

Users can now troubleshoot problems during the deployment phase. The status in the QuayRegistry object can help you monitor the health of the components during the deployment and help you debug any problems that may arise:

```bash
$ curl -X GET -k --header 'Content-Type: application/json' -H "Authorization: Bearer 6B4QTRSTSD1HMIG915VPX7BMEZBV9GPNY2FC2ED" https://min-registry-quay-quay-enterprise.apps.docs.quayteam.org/api/v1/organization/testorg
```

```

```json
{
    "name": "testorg",
    "email": "testorg@example.com",
    "avatar": {
        "name": "testorg",
        "hash": "5f113632ad532fc78215c9258a4fb60606d1fa386c91b141116a1317bf9c53c8",
        "color": "#a55194",
        "kind": "user"
    },
    "is_admin": true,
    "is_member": true,
    "teams": {
        "owners": {
            "name": "owners",
            "description": "",
            "role": "admin",
            "avatar": {
                "name": "owners",
                "hash": "6f0e3a8c0eb46e8834b43b03374ece43a030621d92a7437beb48f871e90f8d90",
                "color": "#c7c7c7",
                "kind": "team"
            },
            "can_view": true,
            "repo_count": 0,
            "member_count": 1,
            "is_synced": false
        }
    },
    "ordered_teams": ["owners"],
    "invoice_email": false,
    "invoice_email_address": null,
    "tag_expiration_s": 1209600,
    "is_free_account": true
}
```

4.1.4. Monitoring and debugging the deployment process

Users can now troubleshoot problems during the deployment phase. The status in the QuayRegistry object can help you monitor the health of the components during the deployment and help you debug any problems that may arise:

```bash
$ oc get quayregistry -n quay-enterprise -o yaml
```

Immediately after deployment, the QuayRegistry object will show the basic configuration:

```
apiVersion: v1
items:
- apiVersion: quay.redhat.com/v1
```
Use the `oc get pods` command to view the current state of the deployed components:

$ oc get pods -n quay-enterprise

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>example-registry-clair-app-86554c6b49-ds7bl</td>
<td>0/1</td>
<td>ContainerCreating</td>
<td>0</td>
<td>2s</td>
</tr>
<tr>
<td>example-registry-clair-app-86554c6b49-hxp5s</td>
<td>0/1</td>
<td>Running</td>
<td>1</td>
<td>17s</td>
</tr>
<tr>
<td>example-registry-clair-postgres-68d8857899-lbc5n</td>
<td>0/1</td>
<td>ContainerCreating</td>
<td>0</td>
<td>17s</td>
</tr>
<tr>
<td>example-registry-quay-app-upgrade-h2v7h</td>
<td>0/1</td>
<td>ContainerCreating</td>
<td>0</td>
<td>9s</td>
</tr>
<tr>
<td>example-registry-quay-config-editor-5f646cbb7-lbnc2</td>
<td>0/1</td>
<td>ContainerCreating</td>
<td>0</td>
<td>17s</td>
</tr>
<tr>
<td>example-registry-quay-database-66f495c9bc-wqsjf</td>
<td>0/1</td>
<td>ContainerCreating</td>
<td>0</td>
<td>17s</td>
</tr>
<tr>
<td>example-registry-quay-mirror-854c88457b-d845g</td>
<td>0/1</td>
<td>Init:0/1</td>
<td>0</td>
<td>2s</td>
</tr>
<tr>
<td>example-registry-quay-mirror-854c88457b-fghxv</td>
<td>0/1</td>
<td>Init:0/1</td>
<td>0</td>
<td>17s</td>
</tr>
<tr>
<td>example-registry-quay-postgres-init-bktdt</td>
<td>0/1</td>
<td>Terminating</td>
<td>0</td>
<td>17s</td>
</tr>
<tr>
<td>example-registry-quay-redis-f9b9d44bf-4htpz</td>
<td>0/1</td>
<td>ContainerCreating</td>
<td>0</td>
<td>17s</td>
</tr>
</tbody>
</table>

While the deployment is in progress, the QuayRegistry object will show the current status. In this instance, database migrations are taking place, and other components are waiting until this completes.

status:
When the deployment process finishes successfully, the status in the QuayRegistry object shows no unhealthy components:

```
status:
  conditions:
  - lastTransitionTime: "2021-09-14T10:52:36Z"
    lastUpdateTime: "2021-09-14T10:52:36Z"
    message: all registry component healthchecks passing
    reason: HealthChecksPassing
    status: "True"
    type: Available
  - lastTransitionTime: "2021-09-14T10:52:46Z"
    lastUpdateTime: "2021-09-14T10:52:46Z"
    message: all objects created/updated successfully
    reason: ComponentsCreationSuccess
    status: "False"
    type: RolloutBlocked
```

When the deployment process finishes successfully, the status in the QuayRegistry object shows no unhealthy components:
4.2. DEPLOYING RED HAT QUAY FROM THE OPENSHIFT CONSOLE

1. Create a namespace, for example, `quay-enterprise`.

2. Select Operators → Installed Operators, then select the Quay Operator to navigate to the Operator detail view.

3. Click 'Create Instance' on the 'Quay Registry' tile under 'Provided APIs'.

4. Optionally change the 'Name' of the `QuayRegistry`. This will affect the hostname of the registry. All other fields have been populated with defaults.

5. Click 'Create' to submit the `QuayRegistry` to be deployed by the Quay Operator.

6. You should be redirected to the `QuayRegistry` list view. Click on the `QuayRegistry` you just created to see the details view.

7. 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.

4.2.1. Using the Quay UI to create the first user

NOTE

This procedure assumes that the `FEATURE_USER_CREATION` config option has not been set to `false`. If it is `false`, then the `Create Account` functionality on the UI will be disabled, and you will have to use the API to create the first user.

1. In the OpenShift console, navigate to Operators → Installed Operators, with the appropriate namespace / project.

2. Click on the newly installed QuayRegistry, to view the details:
3. Once the **Registry Endpoint** has a value, navigate to this URL in your browser

4. Select ‘Create Account’ in the Quay registry UI to create a user

5. Enter details for username, password, email and click **Create Account**
6. You are automatically logged in to the Quay registry
CHAPTER 5. CONFIGURING QUAY ON OPENSHIFT

Once deployed, you can configure the Quay application by editing the Quay configuration bundle secret `spec.configBundleSecret` and you can also change the managed status of components in the `spec.components` object of the QuayRegistry resource.

Alternatively, you can use the config editor UI to configure the Quay application, as described in the section Chapter 6, Using the config tool to reconfigure Quay on OpenShift.

5.1. EDITING THE CONFIG BUNDLE SECRET IN THE OPENSHIFT CONSOLE

Procedure

1. On the Quay Registry overview screen, click the link for the Config Bundle Secret:

2. To edit the secret, click **Actions → Edit Secret**

3. Modify the configuration and save the changes
4. Monitor the deployment to ensure successful completion and that the configuration changes have taken effect

5.2. DETERMINING QUAYREGISTRY ENDPOINTS AND SECRETS

You can examine the QuayRegistry resource, using `oc describe quayregistry` or `oc get quayregistry -o yaml`, to determine the current endpoints and secrets:

```
$ oc get quayregistry example-registry -n quay-enterprise -o yaml

apiVersion: quay.redhat.com/v1
kind: QuayRegistry
metadata:
  ...  
  name: example-registry
  namespace: quay-enterprise
...
spec:
  components:
  - kind: quay
    managed: true
  ...
  - kind: clairpostgres
    managed: true
    configBundleSecret: init-config-bundle-secret
    status:
      configEditorCredentialsSecret: example-registry-quay-config-editor-credentials-fg2gdgtm24
      configEditorEndpoint: https://example-registry-quay-config-editor-quay-enterprise.apps.docs.gcp.quaydev.org
```
currentVersion: 3.7.0  
lastUpdated: 2022-05-11 13:28:38.199476938 +0000 UTC  
registryEndpoint: https://example-registry-quay-quay-enterprise.apps.docs.gcp.quaydev.org

The relevant fields are:

- **registryEndpoint**: The URL for your registry, for browser access to the registry UI, and for the registry API endpoint

- **configBundleSecret**: The config bundle secret, containing the **config.yaml** file and any SSL certs

- **configEditorEndpoint**: The URL for the config editor tool, for browser access to the config tool, and for the configuration API

- **configEditorCredentialsSecret**: The secret containing the username (typically **quayconfig**) and the password for the config editor tool

To determine the username and password for the config editor tool:

1. Retrieve the secret:

   ```bash
   $ oc get secret -n quay-enterprise example-registry-quay-config-editor-credentials-g2gdgm24 -o yaml
   ```

   ```yaml
   apiVersion: v1  
data:  
password: SkZwQkVKTUN0a1BUZmp4dA==  
username: cXVheWNvbzmZpZw==  
kind: Secret
   ```

2. Decode the username:

   ```bash
   $ echo 'cXVheWNvbzmZpZw==' | base64 --decode  
quayconfig
   ```

3. Decode the password:

   ```bash
   $ echo 'SkZwQkVKTUN0a1BUZmp4dA==' | base64 --decode  
JFpBEJMCtkPTfjxt
   ```

### 5.3. DOWNLOADING THE EXISTING CONFIGURATION

There are a number of methods for accessing the current configuration:

1. Using the config editor endpoint, specifying the username and password for the config editor:

   ```bash
   $ curl -k -u quayconfig:JFpBEJMCtkPTfjxt https://example-registry-quay-config-editor-quay-enterprise.apps.docs.quayteam.org/api/v1/config
   ```

   ```json
   {
   "config.yaml": {
   ```
5.4. USING THE CONFIG BUNDLE TO CONFIGURE CUSTOM SSL CERTS

You can configure custom SSL certs either before initial deployment or after Red Hat Quay is deployed on OpenShift, by creating or updating the config bundle secret. If you are adding the cert(s) to an existing deployment, you must include the existing `config.yaml` in the new config bundle secret, even if you are not making any configuration changes.

5.4.1. Set TLS to unmanaged

In your Quay Registry yaml, set `kind: tls` to `managed: false:`
- kind:_tls
  managed: false

In the events, you should see that the change is blocked until you set up the appropriate config:

- lastTransitionTime: '2022-03-28T12:56:49Z'
- lastUpdateTime: '2022-03-28T12:56:49Z'
- message: >-
  required component `tls` marked as unmanaged, but `configBundleSecret` is missing necessary fields
- reason: ConfigInvalid
- status: 'True'

### 5.4.2. Add certs to config bundle

**Procedure**

1. Create the secret using embedded data or using files:

   a. Embed the configuration details directly in the Secret resource YAML file, for example:

      ```yaml
      apiVersion: v1
      kind: Secret
      metadata:
        name: custom-ssl-config-bundle-secret
        namespace: quay-enterprise
      data:
        config.yaml: |
          FEATURE_USER_INITIALIZE: true
          BROWSER_API_CALLS_XHR_ONLY: false
          SUPER_USERS:
            - quayadmin
          FEATURE_USER_CREATION: false
          FEATURE_QUOTA_MANAGEMENT: true
          FEATURE_PROXY_CACHE: true
          FEATURE_BUILD_SUPPORT: true
          DEFAULT_SYSTEM_REJECT_QUOTA_BYTES: 102400000
          extra_ca_cert_my-custom-ssl.crt: |
          -----BEGIN CERTIFICATE-----
          MIIDsDCCApigAwIBAgIUCqlzkHjF5i5TXLFy+sepFrZr/UswDQYJKoZIhvcNAQEL
          BQAwbzELMAkGA1UEBhMC
          -----END CERTIFICATE-----

      Next, create the secret from the YAML file:
      ```
      
      $ oc create -f custom-ssl-config-bundle.yaml
      ```
b. Alternatively, you can create files containing the desired information, and then create the secret from those files:

```bash
$ oc create secret generic custom-ssl-config-bundle-secret \
   --from-file=config.yaml \
   --from-file=extra_ca_cert_my-custom-ssl.crt=my-custom-ssl.crt
```

2. Create or update the QuayRegistry YAML file `quayregistry.yaml`, referencing the created Secret, for example:

`quayregistry.yaml`

```yaml
apiVersion: quay.redhat.com/v1
kind: QuayRegistry
metadata:
  name: example-registry
  namespace: quay-enterprise
spec:
  configBundleSecret: custom-ssl-config-bundle-secret
```

3. Deploy or update the registry using the YAML file:

```bash
oc apply -f quayregistry.yaml
```
6.1. ACCESSING THE CONFIG EDITOR

In the Details section of the QuayRegistry screen, the endpoint for the config editor is available, along with a link to the secret containing the credentials for logging into the config editor:

6.1.1. Retrieving the config editor credentials

1. Click on the link for the config editor secret:
In the Data section of the Secret details screen, click **Reveal values** to see the credentials for logging in to the config editor:

```
2. In the Data section of the Secret details screen, click **Reveal values** to see the credentials for logging in to the config editor:
```

```
6.1.2. Logging in to the config editor
```

```
Browse to the config editor endpoint and then enter the username, typically **quayconfig**, and the corresponding password to access the config tool:
```

```
6.1.2. Logging in to the config editor
```

```
Browse to the config editor endpoint and then enter the username, typically **quayconfig**, and the corresponding password to access the config tool:
```
6.1.3. Changing configuration

In this example of updating the configuration, a superuser is added via the config editor tool:

1. Add an expiration period, for example `4w`, for the time machine functionality:

2. Select **Validate Configuration Changes** to ensure that the changes are valid

3. Apply the changes by pressing the **Reconfigure Quay** button:
Validating configuration

Configuration Validated

Configuration Validated

4. The config tool notifies you that the change has been submitted to Quay:

Validating configuration

Configuration Validated

Configuration Validated

NOTE

Reconfiguring Red Hat Quay using the config tool UI can lead to the registry being unavailable for a short time, while the updated configuration is applied.

6.2. MONITORING RECONFIGURATION IN THE UI

6.2.1. QuayRegistry resource

After reconfiguring the Operator, you can track the progress of the redeployment in the YAML tab for the specific instance of QuayRegistry, in this case, example-registry.
Each time the status changes, you will be prompted to reload the data to see the updated version. Eventually, the Operator will reconcile the changes, and there will be no unhealthy components reported.
6.2.2. Events

The Events tab for the QuayRegistry shows some events related to the redeployment:
Streaming events, for all resources in the namespace that are affected by the reconfiguration, are available in the OpenShift console under Home → Events:

### 6.3. ACCESSING UPDATED INFORMATION AFTER RECONFIGURATION

#### 6.3.1. Accessing the updated config tool credentials in the UI

With Red Hat Quay 3.7, reconfiguring Quay through the UI no longer generates a new login password. The password now generates only once, and remains the same after reconciling `QuayRegistry` objects.

#### 6.3.2. Accessing the updated config.yaml in the UI

Use the config bundle to access the updated `config.yaml` file.

1. On the QuayRegistry details screen, click on the Config Bundle Secret
2. In the Data section of the Secret details screen, click Reveal values to see the `config.yaml` file
3. Check that the change has been applied. In this case, `4w` should be in the list of `TAG_EXPIRATION_OPTIONS`:

```
...  
SERVER_HOSTNAME: example-quay-openshift-operators.apps.docs.quayteam.org
SETUP_COMPLETE: true
SUPER_USERS:
   - quayadmin
TAG_EXPIRATION_OPTIONS:
   - 2w
   - 4w
... 
```
6.4. CUSTOM SSL CERTIFICATES UI

The config tool can be used to load custom certificates to facilitate access to resources such as external databases. Select the custom certs to be uploaded, ensuring that they are in PEM format, with an extension `.crt`.

The config tool also displays a list of any uploaded certificates. Once you upload your custom SSL cert, it will appear in the list:

6.5. 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
```
CHAPTER 7. QUAY OPERATOR FEATURES

7.1. CONSOLE MONITORING AND ALERTING

Red Hat Quay provides support for monitoring Quay instances that were deployed using the Operator, from inside the OpenShift console. The new monitoring features include a Grafana dashboard, access to individual metrics, and alerting to notify for frequently restarting Quay pods.

NOTE
To enable the monitoring features, the Operator must be installed in "all namespaces" mode.

7.1.1. Dashboard

In the OpenShift console, navigate to Monitoring → Dashboards and search for the dashboard of your desired Quay registry instance:
The dashboard shows various statistics including:

- The number of Organizations, Repositories, Users and Robot accounts
- CPU Usage and Max Memory Usage
- Rates of Image Pulls and Pushes, and Authentication requests
- API request rate
- Latencies

7.1.2. Metrics

You can see the underlying metrics behind the Quay dashboard, by accessing Monitoring → Metrics in the UI. In the Expression field, enter the text `quay_` to see the list of metrics available:
Select a sample metric, for example, `quay_org_rows`:
This metric shows the number of organizations in the registry, and it is directly surfaced in the dashboard as well.

### 7.1.3. Alerting

An alert is raised if the Quay pods restart too often. The alert can be configured by accessing the Alerting rules tab from Monitoring → Alerting in the consol UI and searching for the Quay-specific alert:

Select the QuayPodFrequentlyRestarting rule detail to configure the alert:

---

**7.2. MANUALLY UPDATING THE VULNERABILITY DATABASES FOR CLAIR IN AN AIR-GAPPED OPENSHIFT CLUSTER**

Clair utilizes packages called **updaters** that encapsulate the logic of fetching and parsing different vulnerability databases. Clair supports running updaters in a different environment and importing the results. This is aimed at supporting installations that disallow the Clair cluster from talking to the Internet directly.
To manually update the vulnerability databases for Clair in an air-gapped OpenShift cluster, use the following steps:

- Obtain the `clairctl` program
- Retrieve the Clair config
- Use `clairctl` to export the updaters bundle from a Clair instance that has access to the internet
- Update the Clair config in the air-gapped OpenShift cluster to allow access to the Clair database
- Transfer the updaters bundle from the system with internet access, to make it available inside the air-gapped environment
- Use `clairctl` to import the updaters bundle into the Clair instance for the air-gapped OpenShift cluster

### 7.2.1. Obtaining clairctl

To obtain the `clairctl` program from a Clair deployment in an OpenShift cluster, use the `oc cp` command, for example:

```bash
$ oc -n quay-enterprise cp example-registry-clair-app-64dd48f866-6ptgw:/usr/bin/clairctl ./clairctl
$ chmod u+x ./clairctl
```

For a standalone Clair deployment, use the `podman cp` command, for example:

```bash
$ sudo podman cp clairv4:/usr/bin/clairctl ./clairctl
$ chmod u+x ./clairctl
```

### 7.2.2. Retrieving the Clair config

#### 7.2.2.1. Clair on OpenShift config

To retrieve the configuration file for a Clair instance deployed using the OpenShift Operator, retrieve and decode the config secret using the appropriate namespace, and save it to file, for example:

```bash
$ kubectl get secret -n quay-enterprise example-registry-clair-config-secret -o "jsonpath=\${.data['config\.yaml']}"> clair-config.yaml
```

An excerpt from a Clair configuration file is shown below:

```
clair-config.yaml

http_listen_addr: :8080
introspection_addr: ""
log_level: info
indexer:
  connstring: host=example-registry-clair-postgres port=5432 dbname=postgres user=postgres
  password=postgres sslmode=disable
  scanlock_retry: 10
  layer_scan_concurrency: 5
  migrations: true
```

---

CHAPTER 7. QUAY OPERATOR FEATURES

59
7.2.2.2. Standalone Clair config

For standalone Clair deployments, the config file is the one specified in CLAIR_CONF environment variable in the `podman run` command, for example:

```bash
sudo podman run -d --rm --name clairv4 \
-p 8081:8081 -p 8089:8089 \
-e CLAIR_CONF=/clair/config.yaml -e CLAIR_MODE=combo \
-v /etc/clairv4/config:/clair:Z \
registry.redhat.io/quay/clair-rhel8:v3.7.3
```

7.2.3. Exporting the updaters bundle

From a Clair instance that has access to the internet, use `clairctl` with the appropriate configuration file to export the updaters bundle:

```bash
$ ./clairctl --config ./config.yaml export-updaters updates.gz
```

7.2.4. Configuring access to the Clair database in the air-gapped OpenShift cluster

- Use `kubectl` to determine the Clair database service:

```bash
$ kubectl get svc -n quay-enterprise
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CLUSTER-IP</th>
<th>EXTERNAL-IP</th>
<th>PORT(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>example-registry-clair-app</td>
<td>ClusterIP</td>
<td>172.30.224.93</td>
<td>&lt;none&gt;</td>
<td>80/TCP,8089/TCP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4d21h</td>
</tr>
</tbody>
</table>
- Forward the Clair database port so that it is accessible from the local machine, for example:

  ```
  $ kubectl port-forward -n quay-enterprise service/example-registry-clair-postgres 5432:5432
  ```

- Update the Clair configuration file, replacing the value of the `host` in the multiple `connstring` fields with `localhost`, for example:

  ```yaml
  clair-config.yaml
  ...
  connstring: host=localhost port=5432 dbname=postgres user=postgres password=postgres sslmode=disable
  ...
  ```

**NOTE**

As an alternative to using `kubectl port-forward`, you can use `kubefwd` instead. With this method, there is no need to modify the `connstring` field in the Clair configuration file to use `localhost`.

### 7.2.5. Importing the updaters bundle into the air-gapped environment

After transferring the updaters bundle to the air-gapped environment, use `clairctl` to import the bundle into the Clair database deployed by the OpenShift Operator:

```
$ ./clairctl --config ./clair-config.yaml import-updaters updates.gz
```

### 7.3. FIPS READINESS AND COMPLIANCE

FIPS (the Federal Information Processing Standard developed by the National Institute of Standards and Technology, NIST) is regarded as the gold standard for securing and encrypting sensitive data, particularly in heavily regulated areas such as banking, healthcare and the public sector. Red Hat Enterprise Linux and Red Hat OpenShift Container Platform support this standard by providing a FIPS mode in which the system would only allow usage of certain, FIPS-validated cryptographic modules, like `openssl`. This ensures FIPS compliance.

Red Hat Quay supports running on FIPS-enabled RHEL and Red Hat OpenShift Container Platform from version 3.5.
CHAPTER 8. ADVANCED CONCEPTS

8.1. 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 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.

8.1.1. 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    worker         4h6m    v1.20.0+ba45583
user1-jcnp6-worker-d-h5tv2.c.quay-devel.internal   Ready    worker         4h6m    v1.20.0+ba45583
user1-jcnp6-worker-d-m9gg4.c.quay-devel.internal   Ready    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
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
```

### 8.1.2. 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.

### 8.1.3. 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
```

### 8.1.4. Create the registry

Create the registry as explained earlier, and then wait for the deployment to be ready.
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:

```bash
$ 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-gblf7             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
10.131.0.23 user1-jcnp6-worker-d-h5tv2.c.quay-devel.internal
example-registry-quay-database-8dc7cf69-dr2cc         1/1     Running     0          5m43s   10.129.2.18
user1-jcnp6-worker-d-h5tv2.c.quay-devel.internal
example-registry-quay-mirror-78df886b5cc-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-5688ddc6bd-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
```

8.2. ENABLING MONITORING WHEN OPERATOR IS INSTALLED IN A SINGLE NAMESPACE

When Red Hat Quay Operator is installed in a single namespace, the monitoring component is unmanaged. To configure monitoring, you need to enable it for user-defined namespaces in OpenShift Container Platform. For more information, see the OCP documentation for Configuring the monitoring stack and Enabling monitoring for user-defined projects.

The following steps show you how to configure monitoring for Quay, based on the OCP documentation.

8.2.1. Creating a cluster monitoring config map

1. Check whether the `cluster-monitoring-config` ConfigMap object exists:

   ```bash
   $ oc -n openshift-monitoring get configmap cluster-monitoring-config
   Error from server (NotFound): configmaps "cluster-monitoring-config" not found
   ```

2. If the ConfigMap object does not exist:
   a. Create the following YAML manifest. In this example, the file is called `cluster-monitoring-config.yaml`:

   ```yaml
   $ cat cluster-monitoring-config.yaml
   apiVersion: v1
   kind: ConfigMap
   metadata:
   ```
name: cluster-monitoring-config
namespace: openshift-monitoring
data:
  config.yaml: |

b. Create the ConfigMap object:

$ oc apply -f cluster-monitoring-config.yaml configmap/cluster-monitoring-config created

$ oc -n openshift-monitoring get configmap cluster-monitoring-config

<table>
<thead>
<tr>
<th>NAME</th>
<th>DATA</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>cluster-monitoring-config</td>
<td>1</td>
<td>12s</td>
</tr>
</tbody>
</table>

8.2.2. Creating a user-defined workload monitoring config map

1. Check whether the user-workload-monitoring-config ConfigMap object exists:

$ oc -n openshift-user-workload-monitoring get configmap user-workload-monitoring-config

Error from server (NotFound): configmaps "user-workload-monitoring-config" not found

2. If the ConfigMap object does not exist:

a. Create the following YAML manifest. In this example, the file is called user-workload-monitoring-config.yaml:

$ cat user-workload-monitoring-config.yaml

apiVersion: v1
kind: ConfigMap
metadata:
  name: user-workload-monitoring-config
  namespace: openshift-user-workload-monitoring
data:
  config.yaml: |

b. Create the ConfigMap object:

$ oc apply -f user-workload-monitoring-config.yaml

configmap/user-workload-monitoring-config created

8.2.3. Enable monitoring for user-defined projects

1. Check whether monitoring for user-defined projects is running:

$ oc get pods -n openshift-user-workload-monitoring

No resources found in openshift-user-workload-monitoring namespace.

2. Edit the cluster-monitoring-config ConfigMap:
3. Set `enableUserWorkload: true` to enable monitoring for user-defined projects on the cluster:

```yaml
apiVersion: v1
data:
  config.yaml:
    enableUserWorkload: true
kind: ConfigMap
metadata:
  annotations:
```

4. Save the file to apply the changes and then check that the appropriate pods are running:

```bash
$ oc get pods -n openshift-user-workload-monitoring
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>prometheus-operator-6f96b4b8f8-gq6rl</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>15s</td>
</tr>
<tr>
<td>prometheus-user-workload-0</td>
<td>5/5</td>
<td>Running</td>
<td>1</td>
<td>12s</td>
</tr>
<tr>
<td>prometheus-user-workload-1</td>
<td>5/5</td>
<td>Running</td>
<td>1</td>
<td>12s</td>
</tr>
<tr>
<td>thanos-ruler-user-workload-0</td>
<td>3/3</td>
<td>Running</td>
<td>0</td>
<td>8s</td>
</tr>
<tr>
<td>thanos-ruler-user-workload-1</td>
<td>3/3</td>
<td>Running</td>
<td>0</td>
<td>8s</td>
</tr>
</tbody>
</table>

### 8.2.4. Create a Service object to expose Quay metrics

1. Create a YAML file for the Service object:

```bash
$ cat quay-service.yaml
```

```yaml
apiVersion: v1
kind: Service
metadata:
  annotations:
    labels:
      quay-component: monitoring
      quay-operator/quayregistry: example-registry
name: example-registry-quay-metrics
namespace: quay-enterprise
spec:
  ports:
  - name: quay-metrics
    port: 9091
    protocol: TCP
    targetPort: 9091
  selector:
    quay-component: quay-app
    quay-operator/quayregistry: example-registry
    type: ClusterIP
```

2. Create the Service object:

```
$ oc apply -f quay-service.yaml
service/example-registry-quay-metrics created
```

### 8.2.5. Create a ServiceMonitor object

Configure OpenShift Monitoring to scrape the metrics by creating a ServiceMonitor resource.

1. Create a YAML file for the ServiceMonitor resource:

```
$ cat quay-service-monitor.yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    quay-operator/quayregistry: example-registry
  name: example-registry-quay-metrics-monitor
  namespace: quay-enterprise
spec:
  endpoints:
  - port: quay-metrics
    namespaceSelector:
      any: true
    selector:
      matchLabels:
        quay-component: monitoring
```

2. Create the ServiceMonitor:

```
$ oc apply -f quay-service-monitor.yaml
servicemonitor.monitoring.coreos.com/example-registry-quay-metrics-monitor created
```

### 8.2.6. View the metrics in OpenShift

You can access the metrics in the OpenShift console under Monitoring → Metrics. In the Expression field, enter the text `quay_` to see the list of metrics available:
For example, if you have added users to your registry, select the `quay-users_rows` metric:

8.3. 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.

8.3.1. 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.

### 8.3.2. 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.

### 8.4. 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`.

### 8.4.1. Environment Variables

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

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>RELATED_IMAGE_COMPONENT_QUAY</td>
<td>base</td>
</tr>
<tr>
<td>RELATED_IMAGE_COMPONENT_CLAIR</td>
<td>clair</td>
</tr>
</tbody>
</table>
8.4.2. 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:

```json
- name: RELATED_IMAGE_COMPONENT_QUAY
  value: quay.io/projectquay/quay@sha256:c35f5af964431673f4ff5c9e90bdf45f19e38b8742b5903d41c10cc7f6339a6d
- name: RELATED_IMAGE_COMPONENT_CLAIR
  value: quay.io/projectquay/clair@sha256:70c99feceb4c0973540d22e740659cd8d616775d3ad1c1698ddf71d0221f3ce6
- name: RELATED_IMAGE_COMPONENT_POSTGRES
  value: centos/postgresql-10-centos7@sha256:de1560cb35e5ec643e7b3a772ebaac8e3a7a2a8e8271d9e91ff023539b4dfb33
- name: RELATED_IMAGE_COMPONENT_REDIS
  value: centos/redis-32-centos7@sha256:06dbb609484330ec6be6090109f1fa16e936afcf975d1cbc5ff3e6c7cae7542
```

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

8.5. 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.pem test-config-bundle
```
8.5.1. Advanced Clair configuration

8.5.1.1. Unmanaged Clair configuration

With Red Hat Quay 3.7, users can run an unmanaged Clair configuration on the Red Hat Quay OpenShift Container Platform Operator. This feature allows users to create an unmanaged Clair database, or run their custom Clair configuration without an unmanaged database.

8.5.1.1.1. Unmanaging a Clair database

An unmanaged Clair database allows the Red Hat Quay Operator to work in a geo-replicated environment, where multiple instances of the Operator must communicate with the same database. An unmanaged Clair database can also be used when a user requires a highly-available (HA) Clair database that exists outside of a cluster.

Procedure

- In the Quay Operator, set the `clairpostgres` component of the QuayRegistry custom resource to unmanaged:

```yaml
apiVersion: quay.redhat.com/v1
kind: QuayRegistry
metadata:
  name: quay370
spec:
  configBundleSecret: config-bundle-secret
  components:
    - kind: objectstorage
      managed: false
    - kind: route
      managed: true
    - kind: tls
      managed: false
    - kind: clairpostgres
      managed: false
```

8.5.1.1.2. Configuring a custom Clair database

The Red Hat Quay Operator for OpenShift Container Platform allows users to provide their own Clair configuration by editing the `configBundleSecret` parameter.

Procedure

1. Create a Quay config bundle secret that includes the `clair-config.yaml`:

   ```bash
   $ oc create secret generic --from-file config.yaml=./config.yaml --from-file extra_ca_cert_rds-ca-2019-root.pem=./rds-ca-2019-root.pem --from-file clair-config.yaml=./clair-config.yaml --from-file ssl.cert=./ssl.cert --from-file ssl.key=./ssl.key config-bundle-secret
   ```

   Example `clair-config.yaml` configuration:

   ```yaml
   indexer:
     connstring: host=quay-server.example.com port=5432 dbname=quay user=quayrdsdb
     password=quayrdsdb sslrootcert=/run/certs/rds-ca-2019-root.pem sslmode=verify-ca
   ```
The database certificate is mounted under `/run/certs/rds-ca-2019-root.pem` on the Clair application pod in the `clair-config.yaml`. It must be specified when configuring your `clair-config.yaml`. An example `clair-config.yaml` can be found at [Clair on OpenShift config](https://github.com/redhat-appservices/clair-cluster-operator/blob/master/clair-template.yaml).

2. Add the `clair-config.yaml` to your bundle secret, named `configBundleSecret`:

```yaml
apiVersion: v1
group: quay-enterprise
kind: Secret
data:
  config.yaml: <base64 encoded Quay config>
  clair-config.yaml: <base64 encoded Clair config>
  extra_ca_cert_<name>: <base64 encoded ca cert>
  clair-ssl.crt: >-
  clair-ssl.key: >-
```

**NOTE**
When updated, the provided `clair-config.yaml` is mounted into the Clair pod. Any fields not provided are automatically populated with defaults using the Clair configuration module.

After proper configuration, the Clair application pod should return to a **Ready** state.

### 8.5.1.2. Running a custom Clair configuration with a managed database

In some cases, users might want to run a custom Clair configuration with a **managed** database. This is useful in the following scenarios:

- When a user wants to disable an updater.
- When a user is running in an air-gapped environment.
NOTE

- If you are running Quay in an air-gapped environment, the `airgap` parameter of your `clair-config.yaml` must be set to `true`.

- If you are running Quay in an air-gapped environment, you should disable all updaters.

Use the steps in "Configuring a custom Clair database" to configure your database when `clairpostgres` is set to `managed`.

For more information about running Clair in an air-gapped environment, see Configuring access to the Clair database in the air-gapped OpenShift cluster.
Prior to Red Hat Quay 3.7, Quay ran `podman` commands in virtual machines launched by pods. Running builds on virtual platforms requires enabling nested virtualization, which is not featured in Red Hat Enterprise Linux or OpenShift Container Platform. As a result, builds had to run on bare-metal clusters, which is an inefficient use of resources.

With Red Hat Quay 3.7, the bare-metal constraint required to run builds has been removed by adding an additional build option which does not contain the virtual machine layer. As a result, builds can be run on virtualized platforms. Backwards compatibility to run previous build configurations are also available.

### 9.1. RED HAT QUAY ENHANCED BUILD ARCHITECTURE

The preceding image shows the expected design flow and architecture of the enhanced build features:

With this enhancement, the build manager first creates the **Job Object**. Then, the **Job Object** then creates a pod using the `quay-builder-image`. The `quay-builder-image` will contain the **quay-builder binary** and the **Podman** service. The created pod runs as **unprivileged**. The **quay-builder binary** then builds the image while communicating status and retrieving build information from the Build Manager.

### 9.2. RED HAT QUAY BUILD LIMITATIONS

Running builds in Red Hat Quay in an unprivileged context might cause some commands that were working under the previous build strategy to fail. Attempts to change the build strategy could potentially cause performance issues and reliability with the build.

Running builds directly in a container will not have the same isolation as using virtual machines. Changing the build environment might also caused builds that were previously working to fail.

### 9.3. CREATING A RED HAT QUAY BUILDERS ENVIRONMENT WITH OPENSHIFT

#### 9.3.1. OpenShift TLS component
The **tls** component allows you to control TLS configuration.

**NOTE**

Red Hat Quay 3.7 does not support builders when the TLS component is managed by the Operator.

If you set **tls** to **unmanaged**, you supply your own **ssl.cert** and **ssl.key** files. In this instance, if you want your cluster to support builders, you must add both the Quay route and the builder route name to the SAN list in the cert, or alternatively use a wildcard. To add the builder route, use the following format:

```sh
[quayregistry-cr-name]-quay-builder-[ocp-namespace].[ocp-domain-name]:443
```

### 9.3.2. Using OpenShift Container Platform for Red Hat Quay builders

The following procedure describes how you can implement the builders feature in Red Hat Quay.

**Prerequisites**

- Builders require SSL certificates. For more information, see Adding TLS certificates to the Red Hat Quay container.
- If you are using AWS S3 storage, you must modify your storage bucket in the AWS console, prior to running builders. See "Modifying your AWS S3 storage bucket" in the following section for the required parameters.

**PROCEDURE**

- This procedure assumes you already have a cluster provisioned and a Quay Operator running.
- This procedure is for setting up a virtual namespace on OpenShift Container Platform.

#### 9.3.2.1. Preparing OpenShift Container Platform for virtual builders

1. Log in to your Red Hat Quay cluster using a cluster admin account.
2. Create a new project where your virtual builders will be run (e.g., **virtual-builders**).

```sh
$ oc new-project virtual-builders
$ oc create sa -n virtual-builders quay-builder
$ oc adm policy -n virtual-builders add-role-to-user edit system:serviceaccount:virtual-builders:quay-builder
```

3. Create a **ServiceAccount** in this **Project** that will be used to run builds.

```sh
$ oc create sa -n virtual-builders quay-builder
```

4. Provide the created service account with editing permissions so that it can run the build:

```sh
$ oc adm policy -n virtual-builders add-role-to-user edit system:serviceaccount:virtual-builders:quay-builder
```

5. Grant the Quay builder **anyuid scc** permissions:
$ oc adm policy -n virtual-builders add-scc-to-user anyuid -z quay-builder

NOTE

This action requires cluster admin privileges. This is required because builders must run as the Podman user for unprivileged or rootless builds to work.

6. Obtain the token for the Quay builder service account.
   a. If using OpenShift Container Platform 4.10 or an earlier version, enter the following command:
      
      oc sa get-token -n virtual-builders quay-builder
   
   b. If using OpenShift Container Platform 4.11 or later, enter the following command:
      
      $ oc create token quay-builder -n virtual-builders

Sample output

eyJhbGciOiJSUzI1NiIsImtpZCI6IldfQUJkaDVmb3ltTHZ0dGZMYjhIWhNyZTQzN2dJVEJxcDJscldSdEUtYWtsifQ...

7. Determine the builder route:

   $ oc get route -n quay-enterprise

Sample output

<table>
<thead>
<tr>
<th>NAME</th>
<th>HOST/PORT</th>
<th>PATH</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERVICES</td>
<td>PORT</td>
<td>TERMINATION</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>example-registry-quay-builder</td>
<td>example-registry-quay-builder-quay-</td>
<td>example-registry-quay-app grpc edge/Redirect None</td>
</tr>
<tr>
<td>enterprise.apps.docs.quayteam.org</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8. Generate a self-signed SSL certificate with the .crt extension:

   $ oc extract cm/kube-root-ca.crt -n openshift-apiserver ca.crt
   
   $ mv ca.crt extra_ca_cert_build_cluster.crt

9. Locate the secret for you config bundle in the Console, and choose Actions → Edit Secret and add the appropriate builder configuration:

   FEATURE_USER_INITIALIZE: true
   BROWSER_API_CALLS_XHR_ONLY: false
   SUPER_USERS:
   - <superusername>
   FEATURE_USER_CREATION: false
The build route is obtained by running `oc get route -n` with the name of your OpenShift Operators namespace. A port must be provided at the end of the route, for example, and it should follow the following format: `[quayregistry-cr-name]-quay-builder-[ocp-namespace].[ocp-domain-name]:443`.

If the `JOB_REGISTRATION_TIMEOUT` parameter is set too low, you might receive the following error: `failed to register job to build manager: rpc error: code = Unauthenticated desc = Invalid build token: Signature has expired`. It is suggested that this parameter be set to at least 240.

If your Redis host has a password or SSL certificates, you must update accordingly.

Set to match the name of your virtual builders namespace, for example, `virtual-builders`.

For early access, the `BUILDER_CONTAINER_IMAGE` is currently `quay.io/projectquay/quay-builder:3.7.0-rc.2`. Note that this might change during the early access window. In the event this happens, customers will be alerted.

Obtained by running `oc cluster-info`.

---

1. The build route is obtained by running `oc get route -n` with the name of your OpenShift Operators namespace. A port must be provided at the end of the route, for example, and it should follow the following format: `[quayregistry-cr-name]-quay-builder-[ocp-namespace].[ocp-domain-name]:443`.
2. If the `JOB_REGISTRATION_TIMEOUT` parameter is set too low, you might receive the following error: `failed to register job to build manager: rpc error: code = Unauthenticated desc = Invalid build token: Signature has expired`. It is suggested that this parameter be set to at least 240.
3. If your Redis host has a password or SSL certificates, you must update accordingly.
4. Set to match the name of your virtual builders namespace, for example, `virtual-builders`.
5. For early access, the `BUILDER_CONTAINER_IMAGE` is currently `quay.io/projectquay/quay-builder:3.7.0-rc.2`. Note that this might change during the early access window. In the event this happens, customers will be alerted.
6. Obtained by running `oc cluster-info`.
You must manually create and add your custom CA cert, for example, `K8S_API_TLS_CA`: 
`/conf/stack/extra_ca_certs/build_cluster.crt`.

8. Defaults to 5120Mi if left unspecified.

9. For virtual builds, you must ensure that there are enough resources in your cluster. Defaults to 1000m if left unspecified.

10. Defaults to 3968Mi if left unspecified.

11. Defaults to 500m if left unspecified.

12. Obtained when running `oc create sa`.

Sample config

```yaml
FEATURE_USER_INITIALIZE: true
BROWSER_API_CALLS_XHR_ONLY: false
SUPER_USERS:
  - quayadmin
FEATURE_USER_CREATION: false
FEATURE_QUOTA_MANAGEMENT: true
FEATURE_BUILD_SUPPORT: True
BUILDMAN_HOSTNAME: example-registry-quay-builder-quay-enterprise.apps.docs.quayteam.org:443
BUILD_MANAGER:
  - ephemeral
  - ALLOWED_WORKER_COUNT: 1
ORCHESTRATOR_PREFIX: buildman/production/
JOB_REGISTRATION_TIMEOUT: 3600
ORCHESTRATOR:
  REDIS_HOST: example-registry-quay-redis
  REDIS_PASSWORD: ""
  REDIS_SSL: false
  REDIS_SKIP_KEYSPACE_EVENT_SETUP: false
EXECUTORS:
  - EXECUTOR: kubernetesPodman
    NAME: openshift
    BUILDER_NAMESPACE: virtual-builders
    SETUP_TIME: 180
    MINIMUM_RETRY_THRESHOLD:
    BUILDER_CONTAINER_IMAGE: quay.io/projectquay/quay-builder:3.7.0-rc.2
    # Kubernetes resource options
    K8S_API_SERVER: api.docs.quayteam.org:6443
    K8S_API_TLS_CA: /conf/stack/extra_ca_certs/build_cluster.crt
    VOLUME_SIZE: 8G
    KUBERNETES_DISTRIBUTION: openshift
    CONTAINER_MEMORY_LIMITS: 1G
    CONTAINER_CPU_LIMITS: 1080m
    CONTAINER_MEMORY_REQUEST: 1G
    CONTAINER_CPU_REQUEST: 580m
    NODE_SELECTOR_LABEL_KEY: ""
    NODE_SELECTOR_LABEL_VALUE: ""
    SERVICE_ACCOUNT_NAME: quay-builder
```
9.3.2.2. Manually adding SSL certificates.

IMPORTANT

- Due to a known issue with the configuration tool, you must manually add your custom SSL certificates to properly run builders. Use the following procedure to manually add custom SSL certificates. For more information creating SSL certificates, see Adding TLS certificates to the Red Hat Quay container.

9.3.2.2.1. Create and sign certs

1. Create a certificate authority and sign a certificate. For more information, see Create a Certificate Authority and sign a certificate.

NOTE

- Add an alt_name for the URL of your Quay registry.
- Add an alt_name for the BUILDMAN_HOSTNAME that is specified in your config.yaml.

openssl.cnf

```plaintext
[req]
req_extensions = v3_req
distinguished_name = req_distinguished_name
[req_distinguished_name]
  [ v3_req ]
basicConstraints = CA:FALSE
keyUsage = nonRepudiation, digitalSignature, keyEncipherment
subjectAltName = @alt_names
[alt_names]
DNS.1 = example-registry-quay-quay-enterprise.apps.docs.quayteam.org
DNS.2 = example-registry-quay-builder-quay-enterprise.apps.docs.quayteam.org
```

Sample commands

```plaintext
$ openssl genrsa -out rootCA.key 2048
$ openssl req -x509 -new -nodes -key rootCA.key -sha256 -days 1024 -out rootCA.pem
$ openssl genrsa -out ssl.key 2048
$ openssl req -new -key ssl.key -out ssl.csr
$ openssl x509 -req -in ssl.csr -CA rootCA.pem -CAkey rootCA.key -CAcreateserial -out ssl.cert -days 356 -extensions v3_req -extfile openssl.cnf
```

9.3.2.2.2. Set TLS to unmanaged

In your Quay Registry yaml, set kind: tls to managed: false:
9.3.2.2.3. Create temporary secrets

1. Create a secret in your default namespace for the CA cert:

   ```
   $ oc create secret generic -n quay-enterprise temp-crt --from-file extra_ca_cert_build_cluster.crt
   ```

2. Create a secret in your default namespace for the ssl.key and ssl.cert files:

   ```
   $ oc create secret generic -n quay-enterprise quay-config-ssl --from-file ssl.cert --from-file ssl.key
   ```

9.3.2.2.4. Copy secret data to config.yaml

1. Locate the new secrets in the console UI at Workloads → Secrets.

2. For each secret, locate the YAML view:

   ```yaml
   kind: Secret
   apiVersion: v1
   metadata:
     name: temp-crt
     namespace: quay-enterprise
     uid: a4818adb-8e21-443a-a8db-f334ace9f6d0
     resourceVersion: '9087855'
     creationTimestamp: '2022-03-28T13:05:30Z'
     ...  
   data:
     extra_ca_cert_build_cluster.crt: >-
       LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURNakNDQWhxZ0F3SUJBZ0l....
     type: Opaque
   ```

   ```yaml
   kind: Secret
   apiVersion: v1
   metadata:
     name: quay-config-ssl
     namespace: quay-enterprise
     uid: 4f5ae352-17d8-4e2d-89a2-143a3280783c
     resourceVersion: '9090567'
     creationTimestamp: '2022-03-28T13:10:34Z'
   ```
3. Locate the secret for your Quay Registry configuration bundle in the UI, or via the command line by running a command such as:

```bash
$ oc get quayregistries.quay.redhat.com -o jsonpath="{.items[0].spec.configBundleSecret}{\"n\"}" -n quay-enterprise
```

4. In the OpenShift console, select the YAML tab for your config bundle secret, and add the data from the two secrets you created:

```yaml
kind: Secret
apiVersion: v1
metadata:
  name: init-config-bundle-secret
  namespace: quay-enterprise
  uid: 4724aca5-bff0-406a-9162-ccb1972a27c1
  resourceVersion: '4383160'
  creationTimestamp: '2022-03-22T12:35:59Z'

... data:
  config.yaml: >-
    RkVBVFVSRV9VU0VSX0lOSVRJQUxJWkU6IHRydWUKQIJ...
  extra_ca_cert_build_cluster.crt: >-
    LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUVAakNDQTA2Z0F3SUJBZ0lVT...
  ssl.cert: >-
    LS0tLS1CRUdJTiBSU0EgUFJJVkBkFURSBLRVTkLS0tLQpNSUIFfJQkFBS0NBUUVBc...
  ssl.key: >-
    LS0tLS1CRUdJTiBSU0EgUFJJVkBkFURSBLRVTkLS0tLQpNSUIFfJQkFBS0NBUUVBc...
  type: Opaque
```

5. Click **Save**. You should see the pods being re-started:

```bash
$ oc get pods -n quay-enterprise
```

### Sample output

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>example-registry-quay-app-6786987b99-vgg2v</td>
<td>0/1</td>
<td>ContainerCreating</td>
<td>0</td>
<td>2s</td>
</tr>
<tr>
<td>example-registry-quay-app-7975d4889f-qltv</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>5d21h</td>
</tr>
<tr>
<td>example-registry-quay-app-7975d4889f-zn8bb</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>5d21h</td>
</tr>
<tr>
<td>example-registry-quay-app-upgrade-lswsn</td>
<td>0/1</td>
<td>Completed</td>
<td>0</td>
<td>6d1h</td>
</tr>
<tr>
<td>example-registry-quay-config-editor-77847fo4f5-nsbbv</td>
<td>0/1</td>
<td>ContainerCreating</td>
<td>0</td>
<td>2s</td>
</tr>
<tr>
<td>example-registry-quay-config-editor-c6c4d93cd-2mwg2</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>5d21h</td>
</tr>
<tr>
<td>example-registry-quay-database-66969cd859-n2ssm</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>6d1h</td>
</tr>
</tbody>
</table>
6. After your Quay registry has reconfigured, check that your Quay app pods are running:

```bash
$ oc get pods -n quay-enterprise
```

Sample output

<table>
<thead>
<tr>
<th>Pod Name</th>
<th>Phase</th>
<th>Reason</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>example-registry-quay-app-6786987b99-sz6kb</td>
<td>Running</td>
<td>7m45s</td>
<td></td>
</tr>
<tr>
<td>example-registry-quay-app-6786987b99-vgg2v</td>
<td>Running</td>
<td>9m1s</td>
<td></td>
</tr>
<tr>
<td>example-registry-quay-app-upgrade-lswsn</td>
<td>Completed</td>
<td>6d1h</td>
<td></td>
</tr>
<tr>
<td>example-registry-quay-config-editor-77847fc4f5-nsbbv</td>
<td>Running</td>
<td>9m1s</td>
<td></td>
</tr>
<tr>
<td>example-registry-quay-database-66969cd859-n2ssm</td>
<td>Running</td>
<td>6d1h</td>
<td></td>
</tr>
<tr>
<td>example-registry-quay-mirror-758fc68f7-5wxlp</td>
<td>Running</td>
<td>8m29s</td>
<td></td>
</tr>
<tr>
<td>example-registry-quay-mirror-758fc68f7-lb82</td>
<td>Running</td>
<td>8m29s</td>
<td></td>
</tr>
<tr>
<td>example-registry-quay-redis-7cc5f6c977-956g8</td>
<td>Running</td>
<td>5d21h</td>
<td></td>
</tr>
<tr>
<td>example-registry-quay-redis-7cc5f6c977-956g8</td>
<td>Running</td>
<td>5d21h</td>
<td></td>
</tr>
</tbody>
</table>

7. In your browser, access the registry endpoint and validate that the certificate has been updated appropriately:

Common Name (CN) example-registry-quay-quay-enterprise.apps.docs.quayteam.org
Organisation (O) DOCS
Organisational Unit (OU) QUAY

9.3.2.3. Using the UI to create a build trigger

1. Log in to your Quay repository.

2. Click **Create New Repository** and create a new registry, for example, testrepo.

3. On the **Repositories** page, click **Builds** tab on the left hand pane. Alternatively, use the corresponding URL directly, for example:

   https://example-registry-quay-quay-enterprise.apps.docs.quayteam.org/repository/quayadmin/testrepo?tab=builds

**IMPORTANT**

In some cases, the builder might have issues resolving hostnames. This issue might be related to the **dnsPolicy** being set to **default** on the job object. Currently, there is no workaround for this issue. It will be resolved in a future version of Red Hat Quay.

4. Click **Create Build Trigger → Custom Git Repository Push**

5. Enter the HTTPS or SSH style URL used to clone your Git repository, then click **Continue**. For example:

   https://github.com/gabriel-rh/actions_test.git

6. Check **Tag manifest with the branch or tag name** and then click **Continue**.
7. Enter the location of the Dockerfile to build when the trigger is invoked, for example, /Dockerfile and click **Continue**.

8. Enter the location of the context for the Docker build, for example, /, and click **Continue**.

9. If warranted, create a Robot Account. Otherwise, click **Continue**.

10. Click **Continue** to verify the parameters.

11. On the **Builds** page, click **Options** icon of your Trigger Name, and then click **Run Trigger Now**.

12. Enter a commit SHA from the Git repository and click **Start Build**.

13. You can check the status of your build by clicking the commit in the **Build History** page, or by running `oc get pods -n virtual-builders`.

```
$ oc get pods -n virtual-builders
NAME                                               READY   STATUS    RESTARTS   AGE
f192fe4a-c802-4275-bcce-d2031e635126-9l2b5-25lg2   1/1     Running   0          7s

$ oc get pods -n virtual-builders
NAME                                               READY   STATUS        RESTARTS   AGE
f192fe4a-c802-4275-bcce-d2031e635126-9l2b5-25lg2   1/1     Terminating   0          9s

$ oc get pods -n virtual-builders
No resources found in virtual-builders namespace.
```

14. When the build is finished, you can check the status of the tag under **Tags** on the left hand pane.

**NOTE**

With early access, full build logs and timestamps of builds are currently unavailable.

### 9.3.2.4. Modifying your AWS S3 storage bucket

If you are using AWS S3 storage, you must modify your storage bucket in the AWS console, prior to running builders.

1. Log in to your AWS console at [s3.console.aws.com](http://s3.console.aws.com).

2. In the search bar, search for **S3** and then click **S3**.

3. Click the name of your bucket, for example, **myawsbucket**.

4. Click the **Permissions** tab.

5. Under **Cross-origin resource sharing (CORS)** include the following parameters:

```json
{
   "AllowedHeaders": [
      "Authorization"
   ],
   "AllowedMethods": [
```
"GET",
"AllowedOrigins": [ "*"
],
"ExposeHeaders": [],
"MaxAgeSeconds": 3000
},
{
 "AllowedHeaders": [ "Content-Type", "x-amz-acl", "origin"
],
"AllowedMethods": [ "PUT"
],
"AllowedOrigins": [ "*"
],
"ExposeHeaders": [],
"MaxAgeSeconds": 3000
}
CHAPTER 10. GEO-REPLICATION

Geo-replication allows multiple, geographically distributed Red Hat Quay deployments to work as a single registry from the perspective of a client or user. It significantly improves push and pull performance in a globally-distributed Red Hat Quay setup. Image data is asynchronously replicated in the background with transparent failover / redirect for clients.

With Red Hat Quay 3.7, deployments of Red Hat Quay with geo-replication is supported by standalone and Operator deployments.

10.1. GEO-REPLICATION FEATURES

- When geo-replication is configured, container image pushes will be written to the preferred storage engine for that Red Hat Quay instance (typically the nearest storage backend within the region).
- After the initial push, image data will be replicated in the background to other storage engines.
- The list of replication locations is configurable and those can be different storage backends.
- An image pull will always use the closest available storage engine, to maximize pull performance.
- If replication hasn’t been completed yet, the pull will use the source storage backend instead.

10.2. GEO-REPLICATION REQUIREMENTS AND CONSTRAINTS

- A single database, and therefore all metadata and Quay configuration, is shared across all regions.
- A single Redis cache is shared across the entire Quay setup and needs to accessible by all Quay pods.
- The exact same configuration should be used across all regions, with exception of the storage backend, which can be configured explicitly using the QUAY_DISTRIBUTED_STORAGE_PREFERENCE environment variable.
- Geo-Replication requires object storage in each region. It does not work with local storage or NFS.
- Each region must be able to access every storage engine in each region (requires a network path).
- Alternatively, the storage proxy option can be used.
- The entire storage backend (all blobs) is replicated. This is in contrast to repository mirroring, which can be limited to an organization or repository or image.
- All Quay instances must share the same entrypoint, typically via load balancer.
- All Quay instances must have the same set of superusers, as they are defined inside the common configuration file.
- Geo-replication requires your Clair configuration to be set to unmanaged. An unmanaged Clair database allows the Red Hat Quay Operator to work in a geo-replicated environment, where multiple instances of the Operator must communicate with the same database. For more information, see Advanced Clair configuration.
- Geo-Replication requires SSL/TSL certificates and keys. For more information, see Using SSL to protect connections to Red Hat Quay.

If the above requirements cannot be met, you should instead use two or more distinct Quay deployments and take advantage of repository mirroring functionality.

10.3. GEO-REPLICATION USING THE RED HAT QUAY OPERATOR

In the example shown above, the Red Hat Quay Operator is deployed in two separate regions, with a common database and a common Redis instance. Localized image storage is provided in each region and image pulls are served from the closest available storage engine. Container image pushes are written to the preferred storage engine for the Quay instance, and will then be replicated, in the background, to the other storage engines.

Because the Operator now manages the Clair security scanner and its database separately, geo-replication setups can be leveraged so that they do not manage the Clair database. Instead, an external shared database would be used. Red Hat Quay and Clair support several providers and vendors of PostgreSQL, which can be found in the Red Hat Quay 3.x test matrix. Additionally, the Operator also supports custom Clair configurations that can be injected into the deployment, which allows users to configure Clair with the connection credentials for the external database.

10.3.1. Setting up geo-replication on Openshift
Procedure

1. Deploy Quay postgres instance:
   a. Login to the database
   b. Create a database for Quay
      ```sql
      CREATE DATABASE quay;
      \c quay;
      CREATE EXTENSION IF NOT EXISTS pg_trgm;
      ```
   c. Enable pg_trm extension inside the database

2. Deploy a Redis instance:
   a. Deploy a VM for Redis
   b. Make sure that it is accessible from the clusters where Quay is running
   c. Port 6379/TCP must be open
   d. Run Redis inside the instance
      ```shell
      sudo dnf install -y podman
      podman run -d --name redis -p 6379:6379 redis
      ```
   NOTE
   - Deploying a Redis instance might be unnecessary if your cloud provider has its own service.
   - Deploying a Redis instance is required if you are leveraging Builders.

3. Create two object storage backends, one for each cluster
   Ideally one object storage bucket will be close to the 1st cluster (primary) while the other will run closer to the 2nd cluster (secondary).

4. Deploy the clusters with the same config bundle, using environment variable overrides to select the appropriate storage backend for an individual cluster

5. Configure a load balancer, to provide a single entry point to the clusters

10.3.1.1. Configuration

The `config.yaml` file is shared between clusters, and will contain the details for the common PostgreSQL, Redis and storage backends:

```
SERVER_HOSTNAME: <georep.quayteam.org or any other name>
DB_CONNECTION_ARGS:
  autorollback: true
```
A proper `SERVER_HOSTNAME` must be used for the route and must match the hostname of the global load balancer.

To retrieve the configuration file for a Clair instance deployed using the OpenShift Operator, see Retrieving the Clair config.

Create the `configBundleSecret`:

```
$ oc create secret generic --from-file config.yaml=./config.yaml georep-config-bundle
```

In each of the clusters, set the `configBundleSecret` and use the `QUAY_DISTRIBUTED_STORAGE_PREFERENCE` environmental variable override to configure the appropriate storage for that cluster:

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

Because TLS is unmanaged, and the route is managed, you must supply the certificates with either with the config tool or directly in the config bundle. For more information, see Configuring TLS and routes.

European cluster
10.3.2. Mixed storage for geo-replication

Red Hat Quay geo-replication supports the use of different and multiple replication targets, for example, using AWS S3 storage on public cloud and using Ceph storage on-prem. This complicates the key requirement of granting access to all storage backends from all Red Hat Quay pods and cluster nodes. As a result, it is recommended that you:

- Use a VPN to prevent visibility of the internal storage or
- Use a token pair that only allows access to the specified bucket used by Quay

This will result in the public cloud instance of Red Hat Quay having access to on-prem storage but the network will be encrypted, protected, and will use ACLs, thereby meeting security requirements.

If you cannot implement these security measures, it may be preferable to deploy two distinct Red Hat Quay registries and to use repository mirroring as an alternative to geo-replication.
CHAPTER 11. UPGRADING THE QUAY OPERATOR OVERVIEW

The Quay Operator follows a synchronized versioning scheme, which means that each version of the Operator is tied to the version of Quay and the components that 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.

11.1. 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 desirable, then the Manual approval strategy should be selected.

11.2. UPGRADING THE QUAY OPERATOR

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

In general, Red Hat Quay supports upgrades from a prior (N-1) minor version only. For example, upgrading directly from Red Hat Quay 3.0.5 to the latest version of 3.5 is not supported. Instead, users would have to upgrade as follows:

1. 3.0.5 → 3.1.3
2. 3.1.3 → 3.2.2
3. 3.2.2 → 3.3.4
4. 3.3.4 → 3.4.z
5. 3.4.z → 3.5.z

This is required to ensure that any necessary database migrations are done correctly and in the right order during the upgrade.

In some cases, Red Hat Quay supports direct, single-step upgrades from prior (N-2, N-3) minor versions. This exception to the normal, prior minor version-only, upgrade simplifies the upgrade procedure for customers on older releases. The following upgrade paths are supported:
1. 3.3.z → 3.6.z
2. 3.4.z → 3.6.z
3. 3.4.z → 3.7.z
4. 3.5.z → 3.7.z

For users on standalone deployments of Quay wanting to upgrade to 3.7, see the Standalone upgrade guide.

### 11.2.1. Upgrading Quay

To update Quay from one minor version to the next, for example, 3.4 → 3.5, you need to 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 Quay Operator will upgrade automatically to the newest z stream. This results 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.

### 11.2.2. Notes on upgrading directly from 3.3.z or 3.4.z to 3.6

#### 11.2.2.1. Upgrading with edge routing enabled

- Previously, when running a 3.3.z version of Red Hat Quay with edge routing enabled, users were unable to upgrade to 3.4.z versions of Red Hat Quay. This has been resolved with the release of Red Hat Quay 3.6.

- When upgrading from 3.3.z to 3.6, if tls.termination is set to none in your Red Hat Quay 3.3.z deployment, it will change to HTTPS with TLS edge termination and use the default cluster wildcard certificate. For example:

```yaml
apiVersion: redhatcop.redhat.io/v1alpha1
class: QuayEcosystem
metadata:
  name: quay33
spec:
quay:
  imagePullSecretName: redhat-pull-secret
  enableRepoMirroring: true
  image: quay.io/quay/quay:v3.3.4-2
...
  externalAccess:
    hostname: quayv33.apps.devcluster.openshift.com
    tls:
      termination: none
    database:
      ...
```

#### 11.2.2.2. Upgrading with custom TLS certificate/key pairs without Subject Alternative Names

Red Hat Quay 3.7 Deploy Red Hat Quay on OpenShift with the Quay Operator
There is an issue for customers using their own TLS certificate/key pairs without Subject Alternative Names (SANs) when upgrading from Red Hat Quay 3.3.4 to Red Hat Quay 3.6 directly. During the upgrade to Red Hat Quay 3.6, the deployment is blocked, with the error message from the Quay Operator pod logs indicating that the Quay TLS certificate must have SANs.

If possible, you should regenerate your TLS certificates with the correct hostname in the SANs. A possible workaround involves defining an environment variable in the `quay-app`, `quay-upgrade` and `quay-config-editor` pods after upgrade to enable CommonName matching:

```bash
GODEBUG=x509ignoreCN=0
```

The `GODEBUG=x509ignoreCN=0` flag enables the legacy behavior of treating the CommonName field on X.509 certificates as a host name when no SANs are present. However, this workaround is not recommended, as it will not persist across a redeployment.

11.2.2.3. Configuring Clair v4 when upgrading from 3.3.z or 3.4.z to 3.6 using the Quay Operator

To set up Clair v4 on a new Red Hat Quay deployment on OpenShift, it is highly recommended to use the Quay Operator. By default, the Quay Operator will install or upgrade a Clair deployment along with your Red Hat Quay deployment and configure Clair security scanning automatically.

For instructions on setting up Clair v4 on OpenShift, see Setting Up Clair on a Red Hat Quay OpenShift deployment.

11.2.3. Swift configuration when upgrading from 3.3.z to 3.6

When upgrading from Red Hat Quay 3.3.z to 3.6.z, some users might receive the following error: 

```
Switch auth v3 requires tenant_id (string) in os_options.
```

As a workaround, you can manually update your `DISTRIBUTED_STORAGE_CONFIG` to add the `os_options` and `tenant_id` parameters:

```bash
DISTRIBUTED_STORAGE_CONFIG:
  brscale:
  - SwiftStorage
    auth_url: http://****/v3
    auth_version: "3"
    os_options:
      tenant_id: ****
      project_name: ocp-base
      user_domain_name: Default
    storage_path: /datastorage/registry
    swift_container: ocp-svc-quay-ha
    swift_password: *****
    swift_user: *****
```

11.2.4. 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.

11.2.5. 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:

**11.3. UPGRADING A QUAYREGISTRY**

When the Quay Operator starts, 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.
11.4. ENABLING FEATURES IN QUAY 3.7

11.4.1. Quota management configuration
Quota management is now supported under the `FEATURE_QUOTA_MANAGEMENT` property and is turned off by default. To enable quota management, set the feature flag in your `config.yaml` to `true`:

```
FEATURE_QUOTA_MANAGEMENT: true
```

11.4.2. Using Red Hat Quay to proxy a remote organization configuration
Using Red Hat Quay to proxy a remote organization is now supported under the `FEATURE_PROXY_CACHE` property. To enable proxy cache, set the feature flag in your `config.yaml` to `true`:

```
FEATURE_PROXY_CACHE: true
```

11.4.3. Red Hat Quay build enhancements
Builds can be run on virtualized platforms. Backwards compatibility to run previous build configurations are also available. To enable virtual builds, set the feature flag in your `config.yaml` to `true`:

```
FEATURE_BUILD_SUPPORT: true
```

11.4.4. Geo-replication using the Red Hat Quay Operator
Deployments of Red Hat Quay with geo-replication is now supported by Operator deployments. To enable geo-replication, set the feature flag in your `config.yaml` to `true`:

```
FEATURE_STORAGE_REPLICATION: true
```

11.5. ENABLING FEATURES IN QUAY 3.6

11.5.1. Console monitoring and alerting
The support for monitoring Quay 3.6 in the OpenShift console requires that the Operator is installed in all namespaces. If you previously installed the Operator in a specific namespace, delete the Operator itself and reinstall it for all namespaces once the upgrade has taken place.

11.5.2. OCI and Helm support
Support for Helm and some OCI artifacts is now enabled by default in Red Hat Quay 3.6. If you want to explicitly enable the feature, for example, if you are upgrading from a version where it is not enabled by default, you need to reconfigure your Quay deployment to enable the use of OCI artifacts using the following properties:

```
FEATURE_GENERAL_OCI_SUPPORT: true
```

11.6. UPGRADING A QUAY ECOSYSTEM
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.

11.6.1. 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:

1. Delete the QuayRegistry using either the UI or kubectl:

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

2. 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.

11.6.2. 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 is 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.

**ADDITIONAL RESOURCES**

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