Red Hat Quay 3.8

Use Red Hat Quay
Use Red Hat Quay
Abstract

Learn to use Red Hat Quay
# Table of Contents

**PREFACE** ................................................................. 5

**CHAPTER 1. USERS AND ORGANIZATIONS IN RED HAT QUAY** .............................................. 6
  1.1. RED HAT QUAY TENANCY MODEL ............................................ 6
  1.2. CREATING USER ACCOUNTS .................................................. 6
  1.3. CREATING ORGANIZATION ACCOUNTS ..................................... 7

**CHAPTER 2. CREATING A REPOSITORY** .............................................................. 9
  2.1. CREATING AN IMAGE REPOSITORY VIA THE UI .................................. 9
  2.2. CREATING AN IMAGE REPOSITORY VIA DOCKER OR PODMAN .................. 10

**CHAPTER 3. MANAGING ACCESS TO REPOSITORIES** .................................................. 11
  3.1. ALLOWING ACCESS TO USER REPOSITORIES .................................. 11
     3.1.1. Allowing user access to a user repository .............................. 11
  3.2. ALLOWING ROBOT ACCESS TO A USER REPOSITORY ......................... 12
  3.3. ALLOWING ACCESS TO ORGANIZATION REPOSITORIES ..................... 13
     3.3.1. Adding a Team to an organization .................................... 13
     3.3.2. Setting a Team role .................................................... 14
     3.3.3. Adding users to a Team .............................................. 15

**CHAPTER 4. WORKING WITH TAGS** ............................................................. 16
  4.1. VIEWING AND MODIFYING TAGS .............................................. 16
     4.1.1. Adding a new tag to a tagged image .................................. 16
     4.1.2. Moving a tag .......................................................... 16
     4.1.3. Deleting a tag ......................................................... 16
     4.1.4. Viewing tag history and going back in time ............................ 16
           4.1.4.1. Viewing tag history .......................................... 16
           4.1.4.2. Going back in time .......................................... 17
     4.1.5. Fetching an image by tag or digest .................................. 17
  4.2. TAG EXPIRATION .............................................................. 17
     4.2.1. Setting tag expiration from a Dockerfile .............................. 18
     4.2.2. Setting tag expiration from the repository ............................ 18
  4.3. SECURITY SCANNING .......................................................... 19

**CHAPTER 5. VIEWING AND EXPORTING LOGS** .................................................... 20
  5.1. VIEWING LOGS .............................................................. 20
  5.2. EXPORTING REPOSITORY LOGS .............................................. 21

**CHAPTER 6. AUTOMATICALLY BUILDING DOCKERFILES WITH BUILD WORKERS** ................. 23
  6.1. ARCHITECTURE OVERVIEW .................................................. 23
     6.1.1. Build manager ......................................................... 23
     6.1.2. Build workers’ control plane ....................................... 23
     6.1.3. Orchestrator ......................................................... 23
  6.2. OPENSHIFT REQUIREMENTS .................................................. 23
  6.3. ORCHESTRATOR REQUIREMENTS ............................................. 24
  6.4. SETTING UP RED HAT QUAY BUILDERS WITH OPENSHIFT .................... 24
     6.4.1. OpenShift TLS component ......................................... 24
     6.4.2. Prepare OpenShift for Red Hat Quay Builds .......................... 24
     6.4.3. Enable Builders and add Build Configuration to Red Hat Quay’s Configuration Bundle 26
  6.5. OPENSHIFT ROUTES LIMITATION ............................................ 28
  6.6. TROUBLESHOOTING BUILDS .................................................. 29
     6.6.1. DEBUG config flag .................................................. 29
  6.7. SETTING UP GITHUB BUILDS (OPTIONAL) ................................... 30
14.4. ESTABLISHING QUOTA WITH THE RED HAT QUAY API

14.4.1. Setting the quota
14.4.2. Viewing the quota
14.4.3. Modifying the quota
14.4.4. Pushing images
  14.4.4.1. Pushing ubuntu:18.04
  14.4.4.2. Using the API to view quota usage
  14.4.4.3. Pushing another image
14.4.5. Rejecting pushes using quota limits
  14.4.5.1. Setting reject and warning limits
  14.4.5.2. Viewing reject and warning limits
  14.4.5.3. Pushing an image when the reject limit is exceeded
  14.4.5.4. Notifications for limits exceeded

CHAPTER 15. RED HAT QUAY AS A PROXY CACHE FOR UPSTREAM REGISTRIES

15.1. PROXY CACHE ARCHITECTURE
15.2. PROXY CACHE LIMITATIONS
15.3. USING RED HAT QUAY TO PROXY A REMOTE REGISTRY
  15.3.1. Leveraging storage quota limits in proxy organizations
    15.3.1.1. Testing the storage quota limits feature in proxy organizations

CHAPTER 16. RED HAT QUAY BUILD ENHANCEMENTS

16.1. RED HAT QUAY ENHANCED BUILD ARCHITECTURE
16.2. RED HAT QUAY BUILD LIMITATIONS
16.3. CREATING A RED HAT QUAY BUILDERS ENVIRONMENT WITH OPENSSHIFT CONTAINER PLATFORM
  16.3.1. OpenShift Container Platform TLS component
  16.3.2. Using OpenShift Container Platform for Red Hat Quay builders
    16.3.2.1. Preparing OpenShift Container Platform for virtual builders
    16.3.2.2. Manually adding SSL/TLS certificates
      16.3.2.2.1. Creating and signing certificates
      16.3.2.2.2. Setting TLS to unmanaged
      16.3.2.2.3. Creating temporary secrets
      16.3.2.2.4. Copying secret data to the configuration YAML
    16.3.2.3. Using the UI to create a build trigger
    16.3.2.4. Modifying your AWS S3 storage bucket
    16.3.2.5. Modifying your Google Cloud Platform object bucket

CHAPTER 17. USING THE RED HAT QUAY API

17.1. ACCESSING THE QUAY API FROM QUAY.IO
17.2. CREATE OAUTH ACCESS TOKEN
17.3. ACCESSING YOUR QUAY API FROM A WEB BROWSER
17.4. ACCESSING THE RED HAT QUAY API FROM THE COMMAND LINE
  17.4.1. Get superuser information
  17.4.2. Creating a superuser using the API
  17.4.3. List usage logs
    17.4.3.1. Example for pagination
  17.4.4. Directory synchronization
  17.4.5. Create a repository build via API
  17.4.6. Create an org robot
  17.4.7. Trigger a build
  17.4.8. Create a private repository
  17.4.9. Create a mirrored repository
PREFACE

Red Hat Quay container image registries let you store container images in a central location. As a regular user of a Red Hat Quay registry, you can create repositories to organize your images and selectively add read (pull) and write (push) access to the repositories you control. A user with administrative privileges can perform a broader set of tasks, such as the ability to add users and control default settings.

This guide assumes you have a Red Hat Quay deployed and are ready to start setting it up and using it.
CHAPTER 1. USERS AND ORGANIZATIONS IN RED HAT QUAY

Before you begin creating repositories to hold your container images in Red Hat Quay, you should consider how you want to organize those repositories. Every repository in a Red Hat Quay instance must be associated with either an Organization or a User.

1.1. RED HAT QUAY TENANCY MODEL

- **Organizations** provide a way of sharing repositories under a common namespace which does not belong to a single user, but rather to many users in a shared setting (such as a company).

- **Teams** provide a way for an organization to delegate permissions (both global and on specific repositories) to sets or groups of users.

- **Users** can log in to a registry through the Red Hat Quay web UI or a client (such as `podman login`). Each user automatically gets a user namespace, for example, `quay-server.example.com/user/<username>`.

- **Super users** have enhanced access and privileges via the Super User Admin Panel in the user interface and through Super User API calls that are not visible or accessible to normal users.

- **Robot accounts** provide automated access to repositories for non-human users such as pipeline tools and are similar in nature to OpenShift service accounts. Permissions can be granted to a robot account in a repository by adding that account like any other user or team.

1.2. CREATING USER ACCOUNTS

To create a new user for your Red Hat Quay instance:

1. Log in to Red Hat Quay as the superuser (quay by default).

2. Select your account name from the upper right corner of the home page and choose Super User Admin Panel.

3. Select the Users icon from the left column.

4. Select the Create User button.

5. Enter the new user’s Username and Email address, then select the Create User button.
6. Back on the Users page, select the Options icon to the right of the new Username. A drop-down menu appears, as shown in the following figure:

![Change Options Menu]

7. Choose Change Password from the menu.

8. Add the new password and verify it, then select the Change User Password button.

The new user can now use that username and password to log in via the web ui or through some container client.

1.3. CREATING ORGANIZATION ACCOUNTS

Any user can create their own organization to share repositories of container images. To create a new organization:

1. While logged in as any user, select the plus sign (+) from the upper right corner of the home page and choose New Organization.

2. Type the name of the organization. The name must be alphanumeric, all lower case, and between 2 and 255 characters long

3. Select Create Organization. The new organization appears, ready for you to begin adding repositories, teams, robot accounts and other features from icons on the left column. The following figure shows an example of the new organization’s page with the settings tab selected.
Organization Settings

Namespace: clairv4-org
Organization names cannot be changed once set.

Avatar: 
Avatar is generated based off the organization's name.

Delete organization:

Time Machine: 14 days
The amount of time, after a tag is deleted, that the tag is accessible in time machine before being garbage collected.

Save Expiration Time
CHAPTER 2. CREATING A REPOSITORY

A repository provides a central location for storing a related set of container images. There are two ways to create a repository in Red Hat Quay: via a push (from `docker` or `podman`) and via the Red Hat Quay UI. These are essentially the same, whether you are using Quay.io or your own instance of Red Hat Quay.

2.1. CREATING AN IMAGE REPOSITORY VIA THE UI

To create a repository in the Red Hat Quay UI under a user account:

1. Log in to the user account through the web UI.
2. Click the + icon in the top right of the header on the home page (or other page related to the user) and choose New Repository, as shown in the following figure:

   ![New Repository button](image)

3. On the Create New Repository page that appears:
   - Add the new repository name to your user name
   - Click Repository Description and type a description of the repository
   - In Repository Visibility, select whether you want the repository to be public or private
   - Click the Create Repository button.

The new repository is created, starting out empty. A `docker pull` command you could use to pull an image from this repository (minus the image name) appears on the screen.

To create a repository in the Red Hat Quay UI under an organization:

1. Log in as a user that has Admin or Write permission to the organization.
2. From the Repositories view, select the organization name from the right column under Users and Organizations. The page for the organization appears, similar to the page shown in Figure 2.x:
3. Click +Create New Repository in the upper-right part of the page.
4. On the Create New Repository page that appears:
   - Add the new repository name to the organization name
   - Click Repository Description and type a description of the repository
   - In Repository Visibility, select whether you want the repository to be public or private
   - Click the Create Repository button.

The new repository is created, starting out empty. A `docker pull` command you could use to pull an image from this repository (minus the image name) appears on the screen.
2.2. CREATING AN IMAGE REPOSITORY VIA DOCKER OR PODMAN

Assuming you have the proper credentials, pushing an image to a repository that does not yet exist in your Red Hat Quay instance will create that repository as it pushes the image to that repository. Either the **docker** or **podman** commands will work for these examples.

1. **Tag the image:** With an image available from **docker** or **podman** on your local system, tag that image with the new repository name and image name. Here are examples for pushing images to Quay.io or your own Red Hat Quay setup (for example, reg.example.com). For the examples, replace namespace with your Red Hat Quay user name or organization and repo_name with the name of the repository you want to create:

   ```bash
   # sudo podman tag myubi-minimal quay.io/namespace/repo_name
   # sudo podman tag myubi-standard reg.example.com/namespace/repo_name
   ```

2. **Push to the appropriate registry.** For example:

   ```bash
   # sudo podman push quay.io/namespace/repo_name
   # sudo podman push reg.example.com/namespace/repo_name
   ```

**NOTE**

To create an application repository, follow the same procedure you did for creating a container image repository.
CHAPTER 3. MANAGING ACCESS TO REPOSITORIES

As a Red Hat Quay user, you can create your own repositories and make them accessible to other users on your Red Hat Quay instance. As an alternative, you can create organizations to allow access to repositories based on teams. In both user and organization repositories, you can allow access to those repositories by creating credentials associated with robot accounts. Robot accounts make it easy for a variety of container clients (such as docker or podman) to access your repos, without requiring that the client have a Red Hat Quay user account.

3.1. ALLOWING ACCESS TO USER REPOSITORIES

When you create a repository in a user namespace, you can add access to that repository to user accounts or through robot accounts.

3.1.1. Allowing user access to a user repository

To allow access to a repository associated with a user account, do the following:

1. Log into your Red Hat Quay user account.
2. Select a repository under your user namespace to which you want to share access.
3. Select the Settings icon from the left column.
4. Type the name of the user to which you want to grant access to your repository. The user name should appear as you type, as shown in the following figure:

   ![Repositories](image)

5. In the permissions box, select one of the following:
   - Read - Allows the user to view the repository and pull from it.
   - Write - Allows the user to view the repository, as well as pull images from or push images to the repository.
   - Admin - Allows all administrative settings to the repository, as well as all Read and Write permissions.

6. Select the Add Permission button. The user now has the assigned permission.

To remove the user permissions to the repository, select the Options icon to the right of the user entry, then select Delete Permission.
3.2. ALLOWING ROBOT ACCESS TO A USER REPOSITORY

Robot accounts are used to set up automated access to the repositories in your Red Hat Quay registry. They are similar to OpenShift service accounts. When you set up a robot account, you:

- Generate credentials that are associated with the robot account
- Identify repositories and images that the robot can push images to or pull images from
- Copy and paste generated credentials to use with different container clients (such as Docker, podman, Kubernetes, Mesos and others) to access each defined repository

Keep in mind that each robot account is limited to a single user namespace or organization. So, for example, the robot could provide access to all repositories accessible to a user jsmith, but not to any that are not in the user’s list of repositories.

The following procedure steps you through setting up a robot account to allow access to your repositories.

1. Select Robot icon: From the Repositories view, select the Robot icon from the left column.

2. Create Robot account: Select the Create Robot Account button.

3. Set Robot name: Enter the name and description, then select the Create robot account button. The robot name becomes a combination of your user name, plus the robot name you set (for example, jsmith+myrobot)

4. Add permission to the robot account: From the Add permissions screen for the robot account, define the repositories you want the robot to access as follows:
   - Put a check mark next to each repository the robot can access
   - For each repository, select one of the following, and click Add permissions:
     - None - Robot has no permission to the repository
     - Read - Robot can view and pull from the repository
     - Write - Robot can read (pull) from and write (push) to the repository
     - Admin - Full access to pull from and push to the repository, plus the ability to do administrative tasks associated with the repository
   - Select the Add permissions button to apply the settings

5. Get credentials to access repositories via the robot: Back on the Robot Accounts page, select the Robot account name to see credential information for that robot.

6. Get the token: Select Robot Token, as shown in the following figure, to see the token that was generated for the robot. If you want to reset the token, select Regenerate Token.

**NOTE**

It is important to understand that regenerating a token makes any previous tokens for this robot invalid.
7. Get credentials: Once you are satisfied with the generated token, get the resulting credentials in the following ways:

- Kubernetes Secret: Select this to download credentials in the form of a Kubernetes pull secret yaml file.
- rkt Configuration: Select this to download credentials for the rkt container runtime in the form of a json file.
- Docker Login: Select this to copy a full `docker login` command line that includes the credentials.
- Docker Configuration: Select this to download a file to use as a Docker config.json file, to permanently store the credentials on your client system.
- Mesos Credentials: Select this to download a tarball that provides the credentials that can be identified in the uris field of a Mesos configuration file.

3.3. ALLOWING ACCESS TO ORGANIZATION REPOSITORIES

Once you have created an organization, you can associate a set of repositories directly to that organization. To add access to the repositories in that organization, you can add Teams (sets of users with the same permissions) and individual users. Essentially, an organization has the same ability to create repositories and robot accounts as a user does, but an organization is intended to set up shared repositories through groups of users (in teams or individually).

Other things to know about organizations:

- You cannot have an organization in another organization. To subdivide an organization, you use teams.
- Organizations can’t contain users directly. You must first add a team, then add one or more users to each team.
- Teams can be set up in organizations as just members who use the repos and associated images or as administrators with special privileges for managing the organization.

3.3.1. Adding a Team to an organization

When you create a team for your organization you can select the team name, choose which repositories to make available to the team, and decide the level of access to the team.
1. From the Organization view, select the Teams and Membership icon from the left column. You will see that an owners Team exists with Admin privilege for the user who created the Organization.

2. Select Create New Team. You are prompted for the new team name to be associated with the organization. Type the team name, which must start with a lowercase letter, with the rest of the team name as any combination of lowercase letters and numbers (no capitals or special characters allowed).

3. Select the Create team button. The Add permissions window appears, displaying a list of repositories in the organization.

4. Check each repository you want the team to be able to access. Then select one of the following permissions for each:
   - Read - Team members are able to view and pull images
   - Write - Team members can view, pull, and push images
   - Admin - Team members have full read/write privilege, plus the ability to do administrative tasks related to the repository

5. Select Add permissions to save the repository permissions for the team.

### 3.3.2. Setting a Team role

After you have added a team, you can set the role of that team within the organization. From the Teams and Membership screen within the organization, select the TEAM ROLE drop-down menu, as shown in the following figure:

For the selected team, choose one of the following roles:

- **Member** - Inherits all permissions set for the team
- **Creator** - All member permissions, plus the ability to create new repositories
- **Admin** - Full administrative access to the organization, including the ability to create teams, add members, and set permissions.
3.3.3. Adding users to a Team

As someone with Admin privilege to an organization, you can add users and robots to a team. When you add a user, it sends an email to that user. The user remains pending until that user accepts the invitation.

To add users or robots to a team, start from the organization’s screen and do the following:

1. Select the team you want to add users or robots to.
2. In the Team Members box, type one of the following:
   - A username from an account on the Red Hat Quay registry
   - The email address for a user account on the registry
   - The name of a robot account. The name must be in the form of orgname+robotname
3. In the case of the robot account, it is immediately added to the team. For a user account, an invitation to join is mailed to the user. Until the user accepts that invitation, the user remains in the INVITED TO JOIN state.

Next, the user accepts the email invitation to join the team. The next time the user logs in to the Red Hat Quay instance, the user moves from the INVITED TO JOIN list to the MEMBERS list for the organization.
CHAPTER 4. WORKING WITH TAGS

Tags provide a way to identify the version of an image, as well as offering a means of naming the same image in different ways. Besides an image’s version, an image tag can identify its uses (such as devel, testing, or prod) or the fact that it is the most recent version (latest).

From the Tags tab of an image repository, you can view, modify, add, move, delete, and see the history of tags. You also can fetch command-lines you can use to download (pull) a specific image (based on its name and tag) using different commands.

4.1. VIEWING AND MODIFYING TAGS

The tags of a repository can be viewed and modified in the tags panel of the repository page, found by clicking on the Tags tab.

4.1.1. Adding a new tag to a tagged image

A new tag can be added to a tagged image by clicking on the gear icon next to the tag and choosing Add New Tag. Red Hat Quay will confirm the addition of the new tag to the image.

4.1.2. Moving a tag

Moving a tag to a different image is accomplished by performing the same operation as adding a new tag, but giving an existing tag name. Red Hat Quay will confirm that you want the tag moved, rather than added.

4.1.3. Deleting a tag

A specific tag and all its images can be deleted by clicking on the tag’s gear icon and choosing Delete Tag. This will delete the tag and any images unique to it. Images will not be deleted until no tag references them either directly or indirectly through a parent child relationship.

4.1.4. Viewing tag history and going back in time

4.1.4.1. Viewing tag history

To view the image history for a tag, click on the View Tags History menu item located under the Actions menu. The page shown will display each image to which the tag pointed in the past and when it pointed to that image.
4.1.4.2. Going back in time

To revert the tag to a previous image, find the history line where your desired image was overwritten, and click on the Restore link.

4.1.5. Fetching an image by tag or digest

From the Tags tab, you can view different ways of pulling images from the clients that are ready to use those images.

1. Select a particular repository/image
2. Select Tags in the left column
3. Select the Fetch Tag icon for a particular image/tag combination
4. When the Fetch Tag pop-up appears, select the Image format box to see a drop-down menu that shows different ways that are available to pull the image. The selections offer full command lines for pulling a specific container image to the local system:

You can select to pull a regular of an image by tag name or by digest name using the docker command. Choose the type of pull you want, then select Copy Command. The full command-line is copied into your clipboard. These two commands show a docker pull by tag and by digest:

```
docker pull quay.io/cnegus/whatever:latest
```
```
docker pull quay.io/cnegus/whatever@sha256:e02231a6aa8ba7f5da3859a359f99d77e371cb47e643ce78e101958782581fb9
```

Paste the command into a command-line shell on a system that has the docker command and service available, and press Enter. At this point, the container image is ready to run on your local system.

On RHEL and Fedora systems, you can substitute podman for docker to pull and run the selected image.

4.2. TAG EXPIRATION

Images can be set to expire from a Red Hat Quay repository at a chosen date and time using a feature called tag expiration. Here are a few things to know about about tag expiration:
When a tag expires, the tag is deleted from the repository. If it is the last tag for a specific image, the image is set to be deleted.

Expiration is set on a per-tag basis, not for a repository on the whole.

When a tag expires or is deleted, it is not immediately removed from the registry. The value of Time Machine (in User settings) defines when the deleted tag is actually removed and garbage collected. By default, that value is 14 days. Up until that time, a tag can be repointed to an expired or deleted image.

The Red Hat Quay superuser has no special privilege related to deleting expired images from user repositories. There is no central mechanism for the superuser to gather information and act on user repositories. It is up to the owners of each repository to manage expiration and ultimate deletion of their images.

Tag expiration can be set in different ways:

- By setting the `quay.expires-after` LABEL in the Dockerfile when the image is created. This sets a time to expire from when the image is built.

- By choosing the expiration date from the EXPIRES column for the repository tag and selecting a specific date and time to expire.

The following figure shows the Options entry for changing tag expiration and the EXPIRES field for when the tag expires. Hover over the EXPIRES field to see the expiration date and time that is currently set.

### 4.2.1. Setting tag expiration from a Dockerfile

Adding a label like `quay.expires-after=20h` via the Dockerfile LABEL command will cause a tag to automatically expire after the time indicated. The time values could be something like `1h`, `2d`, `3w` for hours, days, and weeks, respectively, from the time the image is built.

### 4.2.2. Setting tag expiration from the repository

On the Repository Tag page there is a UI column titled `EXPIRES` that indicates when a tag will expire. Users can set this by clicking on the time that it will expire or by clicking the Settings button (gear icon) on the right and choosing `Change Expiration`.

Choose the date and time when prompted and select `Change Expiration`. The tag will be set to be deleted from the repository when the expiration time is reached.
4.3. SECURITY SCANNING

By clicking the on the vulnerability or fixable count next to a tab you can jump into the security scanning information for that tag. There you can find which CVEs your image is susceptible to, and what remediation options you may have available.

Keep in mind that image scanning only lists vulnerabilities found by the Clair image scanner. What each user does about the vulnerabilities that are uncovered is completely up to that user. The Red Hat Quay superuser does not act on those vulnerabilities found.
CHAPTER 5. VIEWING AND EXPORTING LOGS

Activity logs are gathered for all repositories and namespaces (users and organizations) in Red Hat Quay. There are multiple ways of accessing log files, including:

- Viewing logs through the web UI
- Exporting logs so they can be saved externally.
- Accessing log entries via the API

To access logs, you must have Admin privilege to the selected repository or namespace.

NOTE

A maximum of 100 log results are available at a time via the API. To gather more results that that, you must use the log exporter feature described in this chapter.

5.1. VIEWING LOGS

To view log entries for a repository or namespace from the web UI, do the following:

1. Select a repository or namespace (organization or user) for which you have Admin privileges.

2. Select the Usage Logs icon from the left column. A Usage Logs screen appears, like the one shown in the following figure:

   ![Usage Logs Screen]

3. From the Usage Logs page, you can:
   - Set the date range for viewing log entries by adding dates to the From and to boxes. By default, the most recent one week of log entries is displayed.
   - Type a string into the Filter Logs box to display log entries that contain the given string.
5.2. EXPORTING REPOSITORY LOGS

To be able to grab a larger number of log files and save them outside of the Red Hat Quay database, you can use the Export Logs feature. Here are a few things you should know about using Export Logs:

- You can choose a range of dates for the logs you want to gather from a repository.
- You can request that the logs be sent to you via an email attachment or directed to a callback URL.
- You need Admin privilege to the repository or namespace to export logs
- A maximum of 30 days of log data can be exported at a time
- Export Logs only gathers log data that was previously produced. It does not stream logging data.
- Your Red Hat Quay instance must be configured for external storage for this feature (local storage will not work).
- Once the logs are gathered and available, you should immediately copy that data if you want to save it. By default, the data expires in an hour.

To use the Export Logs feature:

1. Select a repository for which you have Admin privileges.
2. Select the Usage Logs icon from the left column. A Usage Logs screen appears.
3. Choose the From and to date range of the log entries you want to gather.
4. Select the Export Logs button. An Export Usage Logs pop-up appears, as shown

   **Export Usage Logs**

   Enter an e-mail address or callback URL [must start with http:// or https://] at which to receive the exported logs once they have been fully processed:

   johnjones@example.com

   Note: The export process can take up to an hour to process if there are many logs. As well, only a single export process can run at a time for each namespace. Additional export requests will be queued.

   ![Export Usage Logs Pop-Up]

5. Enter the email address or callback URL you want to receive the exported logs. For the callback URL, you could use a URL to a place such as webhook.site.
6. Select Start Logs Export. This causes Red Hat Quay to begin gathering the selected log entries. Depending on the amount of logging data being gathered, this can take anywhere from one minute to an hour to complete.
7. When the log export is completed you will either:
- Receive an email, alerting you to the availability of your requested exported log entries.
- See a successful status of your log export request from the webhook URL. A link to the exported data will be available for you to select to download the logs.

Keep in mind that the URL points to a location in your Red Hat Quay external storage and is set to expire within an hour. So make sure you copy the exported logs before that expiration time if you intend to keep them.
CHAPTER 6. AUTOMATICALLY BUILDING DOCKERFILES WITH BUILD WORKERS

Red Hat Quay supports building Dockerfiles using a set of worker nodes on OpenShift or Kubernetes. Build triggers, such as GitHub webhooks can be configured to automatically build new versions of your repositories when new code is committed. This document will walk you through enabling builds with your Red Hat Quay installation and setting up one or more OpenShift/K8s clusters to accept builds from Red Hat Quay. With Red Hat Quay 3.4, the underlying Build Manager has been completely re-written as part of Red Hat Quay’s migration from Python 2 to Python 3. As a result, builder nodes are now dynamically created as Kubernetes Jobs versus builder nodes that ran continuously in Red Hat Quay 3.3 and earlier. This greatly simplifies how Red Hat Quay manages builds and provides the same mechanism quay.io utilizes to handle thousands of container image builds daily. Customers who are currently running static (“Enterprise” builders under Red Hat Quay 3.3) will be required to migrate to a Kubernetes-based build mechanism.

6.1. ARCHITECTURE OVERVIEW

The Red Hat Quay Build system is designed for scalability (since it is used to host all builds at quay.io). The Build Manager component of Red Hat Quay provides an orchestration layer that tracks build requests and ensures that a Build Executor (OpenShift/K8s cluster) will carry out each request. Each build is handled by a Kubernetes Job which launches a small virtual machine to completely isolate and contain the image build process. This ensures that container builds do not affect each other or the underlying build system. Multiple Executors can be configured to ensure that builds are performed even in the event of infrastructure failures. Red Hat Quay will automatically send builds to a different Executor if it detects that one Executor is having difficulties.

NOTE

The upstream version of Red Hat Quay provides instructions on how to configure an AWS/EC2 based Executor. This configuration is not supported for Red Hat Quay customers.

6.1.1. Build manager

The build manager is responsible for the lifecycle of scheduled build. Operations requiring updating the build queue, build phase and running jobs’ status is handled by the build manager.

6.1.2. Build workers’ control plane

Build jobs are run on separate worker nodes, and are scheduled on separate control planes (executor). Currently, Red Hat Quay supports running jobs on AWS and Kubernetes. Builds are executed using quay.io/quay/quay-builder. On AWS, builds are scheduled on EC2 instances. On k8s, the builds are scheduled as job resources.

6.1.3. Orchestrator

The orchestrator is used to store the state of currently running build jobs, and publish events for the build manager to consume. e.g expiry events. Currently, the supported orchestrator backend is Redis.

6.2. OPENSHIFT REQUIREMENTS

Red Hat Quay builds are supported on Kubernetes and OpenShift 4.5 and higher. A bare metal (non-virtualized) worker node is required since build pods require the ability to run kvm virtualization. Each

23
build is done in an ephemeral virtual machine to ensure complete isolation and security while the build is running. In addition, your OpenShift cluster should permit the ServiceAccount associated with Red Hat Quay builds to run with the necessary SecurityContextConstraint to support privileged containers.

6.3. ORCHESTRATOR REQUIREMENTS

The Red Hat Quay builds need access to a Redis instance to track build status information. It is acceptable to use the same Redis instance already deployed with your Red Hat Quay installation. All build queues are managed in the Red Hat Quay database so there is no need for a highly available Redis instance.

6.4. SETTING UP RED HAT QUAY BUILDERS WITH OPENSHIFT

6.4.1. 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:

[quayregistry-cr-name]-quay-builder-[ocp-namespace].[ocp-domain-name]

6.4.2. Prepare OpenShift for Red Hat Quay Builds

There are several actions that are needed on an OpenShift cluster before it can accept builds from Red Hat Quay.

1. Create a project where builds will be run (e.g. ‘builder’)

   $ oc new-project builder

2. Create a ServiceAccount in this Project that will be used to run builds. Ensure that it has sufficient privileges to create Jobs and Pods. Copy the ServiceAccount’s token for use later.

   $ oc create sa -n builder quay-builder
   $ oc policy add-role-to-user -n builder edit system:serviceaccount:builder:quay-builder
   $ oc sa get-token -n builder quay-builder

3. Identify the URL for the OpenShift cluster’s API server. This can be found from the OpenShift Console.

4. Identify a worker node label to be used when scheduling build Jobs. Because build pods need to run on bare metal worker nodes, typically these are identified with specific labels. Check with your cluster administrator to determine exactly which node label should be used.
5. If the cluster is using a self-signed certificate, get the kube apiserver’s CA to add to Red Hat Quay’s extra certs.
   a. Get the name of the secret containing the CA:

```
$ oc get sa openshift-apiserver-sa --namespace=openshift-apiserver -o json | jq '.secrets[] | select(.name | contains("openshift-apiserver-sa-token"))|.name
```

b. Get the `ca.crt` key value from the secret in the Openshift console. The value should begin with “-----BEGIN CERTIFICATE-----”

c. Import the CA in Red Hat Quay using the ConfigTool. Ensure the name of this file matches `K8S_API_TLS_CA`.

6. Create the necessary security contexts/role bindings for the **ServiceAccount**:

```yaml
apiVersion: security.openshift.io/v1
kind: SecurityContextConstraints
metadata:
  name: quay-builder
priority: null
readOnlyRootFilesystem: false
requiredDropCapabilities: null
runAsUser:
  type: RunAsAny
seLinuxContext:
  type: RunAsAny
seccompProfiles:
  - '*'
supplementalGroups:
  type: RunAsAny
volumes:
  - '*'
allowHostDirVolumePlugin: true
allowHostIPC: true
allowHostNetwork: true
allowHostPID: true
allowHostPorts: true
allowPrivilegeEscalation: true
allowPrivilegedContainer: true
allowedCapabilities:
  - '*'
allowedUnsafeSysctls:
  - '*'
defaultAddCapabilities: null
fsGroup:
  type: RunAsAny
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: quay-builder-scc
  namespace: builder
rules:
  - apiGroups:
    - security.openshift.io
      apiVersion: security.openshift.io/v1
      kind: SecurityContextConstraints
      metadata:
        name: quay-builder
        priority: null
        readOnlyRootFilesystem: false
        requiredDropCapabilities: null
        runAsUser:
          type: RunAsAny
        seLinuxContext:
          type: RunAsAny
        seccompProfiles:
          - '*'
        supplementalGroups:
          type: RunAsAny
        volumes:
          - '*'
      allowHostDirVolumePlugin: true
      allowHostIPC: true
      allowHostNetwork: true
      allowHostPID: true
      allowHostPorts: true
      allowPrivilegeEscalation: true
      allowPrivilegedContainer: true
      allowedCapabilities:
        - '*'
      allowedUnsafeSysctls:
        - '*'
      defaultAddCapabilities: null
    - apiGroups:
      - security.openshift.io
      apiVersion: security.openshift.io/v1
      kind: SecurityContextConstraints
      metadata:
        name: quay-builder
        priority: null
        readOnlyRootFilesystem: false
```

CHAPTER 6. AUTOMATICALLY BUILDING DOCKERFILES WITH BUILD WORKERS
resourceNames:
- quay-builder
resources:
- securitycontextconstraints
verbs:
- use
---

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: quay-builder-scc
  namespace: builder
subjects:
- kind: ServiceAccount
  name: quay-builder
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: quay-builder-scc

6.4.3. Enable Builders and add Build Configuration to Red Hat Quay’s Configuration Bundle

1. Ensure that you’ve got Builds enabled in your Red Hat Quay configuration.

   FEATURE_BUILD_SUPPORT: True

1. Add the following to your Red Hat Quay configuration bundle, replacing each value with a value specific to your installation.

   NOTE

   Currently only the Build feature itself can be enabled via the Red Hat Quay Config Tool. The actual configuration of the Build Manager and Executors must be done manually in the config.yaml file.

   BUILD_MANAGER:
   - ephemeral
   - ALLOWED_WORKER_COUNT: 1
   ORCHESTRATOR_PREFIX: buildman/production/
   ORCHESTRATOR:
     REDIS_HOST: quay-redis-host
     REDIS_PASSWORD: quay-redis-password
     REDIS_SSL: true
     REDIS_SKIP_KEYSPACE_EVENT_SETUP: false
   EXECUTOR:
     kubernetes
     BUILDER_NAMESPACE: builder
     K8S_API_SERVER: api.openshift.somehost.org:6443
     K8S_API_TLS_CA: /conf/stack/extra_ca_certs/build_cluster.crt
     VOLUME_SIZE: 8G
     KUBERNETES_DISTRIBUTION: openshift
     CONTAINER_MEMORY_LIMITS: 5120Mi
     CONTAINER_CPU_LIMITS: 1000m
CONTAINER_MEMORY_REQUEST: 3968Mi
CONTAINER_CPU_REQUEST: 500m
NODE_SELECTOR_LABEL_KEY: beta.kubernetes.io/instance-type
NODE_SELECTOR_LABEL_VALUE: n1-standard-4
CONTAINER_RUNTIME: podman
SERVICE_ACCOUNT_NAME: *****
SERVICE_ACCOUNT_TOKEN: *****
QUAY_USERNAME: quay-username
QUAY_PASSWORD: quay-password
WORKER_IMAGE: <registry>/quay-quay-builder
WORKER_TAG: some_tag
BUILDER_VM_CONTAINER_IMAGE: <registry>/quay-quay-builder-qemu-rhcos:v3.4.0
SETUP_TIME: 180
MINIMUM_RETRY_THRESHOLD: 0
SSH_AUTHORIZED_KEYS:
- ssh-rsa 12345 someuser@email.com
- ssh-rsa 67890 someuser2@email.com

Each configuration field is explained below.

ALLOWED_WORKER_COUNT
Defines how many Build Workers are instantiated per Red Hat Quay Pod. Typically this is ‘1’.

ORCHESTRATOR_PREFIX
Defines a unique prefix to be added to all Redis keys (useful to isolate Orchestrator values from other Redis keys).

REDIS_HOST
Hostname for your Redis service.

REDIS_PASSWORD
Password to authenticate into your Redis service.

REDIS_SSL
Defines whether or not your Redis connection uses SSL.

REDIS_SKIP_KEYSPACE_EVENT_SETUP
By default, Red Hat Quay does not set up the keyspace events required for key events at runtime. To do so, set REDIS_SKIP_KEYSPACE_EVENT_SETUP to false.

EXECUTOR
Starts a definition of an Executor of this type. Valid values are ‘kubernetes’ and ‘ec2’

BUILDER_NAMESPACE
Kubernetes namespace where Red Hat Quay builds will take place

K8S_API_SERVER
Hostname for API Server of OpenShift cluster where builds will take place

K8S_API_TLS_CA
The filepath in the Quay container of the build cluster’s CA certificate for the Quay app to trust when making API calls.

KUBERNETES_DISTRIBUTION
Indicates which type of Kubernetes is being used. Valid values are ‘openshift’ and ‘k8s’.

CONTAINER_*
Define the resource requests and limits for each build pod.

NODE_SELECTOR_*
Defines the node selector label name/value pair where build Pods should be scheduled.

**CONTAINER_RUNTIME**

Specifies whether the builder should run `docker` or `podman`. Customers using Red Hat’s `quay-builder` image should set this to `podman`.

**SERVICE_ACCOUNT_NAME/SERVICE_ACCOUNT_TOKEN**

Defines the Service Account name/token that will be used by build Pods.

**QUAY_USERNAME/QUAY_PASSWORD**

Defines the registry credentials needed to pull the Red Hat Quay build worker image that is specified in the `WORKER_IMAGE` field. Customers should provide a Red Hat Service Account credential as defined in the section “Creating Registry Service Accounts” against registry.redhat.io in the article at https://access.redhat.com/RegistryAuthentication.

**WORKER_IMAGE**

Image reference for the Red Hat Quay builder image. `registry.redhat.io/quay/quay-builder`

**WORKER_TAG**

Tag for the builder image desired. The latest version is v3.4.0.

**BUILDER_VM_CONTAINER_IMAGE**

The full reference to the container image holding the internal VM needed to run each Red Hat Quay build (`registry.redhat.io/quay/quay-builder-qemu-rhcos:v3.4.0`).

**SETUP_TIME**

Specifies the number of seconds at which a build times out if it has not yet registered itself with the Build Manager (default is 500 seconds). Builds that time out are attempted to be restarted three times. If the build does not register itself after three attempts it is considered failed.

**MINIMUM_RETRY_THRESHOLD**

This setting is used with multiple Executors; it indicates how many retries are attempted to start a build before a different Executor is chosen. Setting to 0 means there are no restrictions on how many tries the build job needs to have. This value should be kept intentionally small (three or less) to ensure failovers happen quickly in the event of infrastructure failures. You must specify a value for this setting. E.g Kubernetes is set as the first executor and EC2 as the second executor. If we want the last attempt to run a job to always be executed on EC2 and not Kubernetes, we would set the Kubernetes executor’s `MINIMUM_RETRY_THRESHOLD` to 1 and EC2’s `MINIMUM_RETRY_THRESHOLD` to 0 (defaults to 0 if not set). In this case, kubernetes’ `MINIMUM_RETRY_THRESHOLD` > `retries_remaining(1)` would evaluate to False, thus falling back to the second executor configured.

**SSH_AUTHORIZED_KEYS**

List of ssh keys to bootstrap in the ignition config. This allows other keys to be used to ssh into the EC2 instance or QEMU VM

### 6.5. OPENSHIFT ROUTES LIMITATION

**NOTE**

This section only applies if you are using the Quay Operator on OpenShift with managed `route` component.

Due to a limitation of OpenShift Routes to only be able to serve traffic to a single port, additional steps are required to set up builds. Ensure that your `kubectl` or `oc` CLI tool is configured to work with the cluster where the Quay Operator is installed and that your `QuayRegistry` exists (not necessarily the same as the bare metal cluster where your builders run).
- Ensure that HTTP/2 ingress is enabled on the OpenShift cluster by following these steps.

- The Quay Operator will create a Route which directs gRPC traffic to the build manager server running inside the existing Quay pod(s). If you want to use a custom hostname (such as a subdomain like builder.registry.example.com), ensure that you create a CNAME record with your DNS provider which points to the status.ingress[0].host of the created Route:

  ```
  $ kubectl get -n <namespace> route <quayregistry-name>-quay-builder -o jsonpath=
  {.status.ingress[0].host}
  ```

- Using the OpenShift UI or CLI, update the Secret referenced by spec.configBundleSecret of the QuayRegistry with the build cluster CA certificate (name the key extra_ca_cert_build_cluster.cert), and update the config.yaml entry with the correct values referenced in the builder config above (depending on your build executor) along with the BUILDMAN_HOSTNAME field:

  ```
  BUILDMAN_HOSTNAME: <build-manager-hostname>
  BUILD_MANAGER:
  - ephemeral
  - ALLOWED_WORKER_COUNT: 1
  ORCHESTRATOR_PREFIX: buildman/production/
  JOB_REGISTRATION_TIMEOUT: 600
  ORCHESTRATOR:
  REDIS_HOST: quay-redis-host
  REDIS_PASSWORD: quay-redis-password
  REDIS_SSL: true
  REDIS_SKIP_KEYSPACE_EVENT_SETUP: false
  EXECUTORS:
  - EXECUTOR: kubernetes
  BUILDING_NAMESPACE: builder
  ```

  The extra configuration field is explained below:

  **BUILDMAN_HOSTNAME**

  The externally accessible server hostname which the build jobs use to communicate back to the build manager. Default is the same as SERVER_HOSTNAME. For OpenShift Route, it is either status.ingress[0].host or the CNAME entry if using a custom hostname. BUILDMAN_HOSTNAME needs to include the port number, e.g. somehost:443 for OpenShift Route, as the gRPC client used to communicate with the build manager does not infer any port if omitted.

### 6.6. TROUBLESHOOTING BUILDS

The builder instances started by the build manager are ephemeral. This means that they will either get shut down by Red Hat Quay on timeouts/failure or garbage collected by the control plane (EC2/K8s). This means that in order to get the builder logs, one needs to do so while the builds are running.

#### 6.6.1. DEBUG config flag

A DEBUG flag can be set in order to prevent the builder instances from getting cleaned up after completion/failure. To do so, in the desired executor configuration, set DEBUG to true. For example:

```
EXECUTORS:
- EXECUTOR: ec2
```
When set to true, DEBUG will prevent the build nodes from shutting down after the quay-builder service is done or fails, and will prevent the build manager from cleaning up the instances (terminating EC2 instances or deleting k8s jobs). This will allow debugging builder node issues, and should not be set in a production environment. The lifetime service will still exist. i.e The instance will still shutdown after approximately 2 hours (EC2 instances will terminate, k8s jobs will complete) Setting DEBUG will also affect ALLOWED_WORKER_COUNT, as the unterminated instances/jobs will still count towards the total number of running workers. This means the existing builder workers will need to manually be deleted if ALLOWED_WORKER_COUNT is reached to be able to schedule new builds.

The following steps:

1. The guest VM forwards its SSH port (22) to its host’s (the pod) port 2222. Port forward the builder pod’s port 2222 to a port on localhost. e.g

   ```
   $ kubectl port-forward <builder pod> 9999:2222
   ```

2. SSH into the VM running inside the container using a key set from SSH_AUTHORIZED_KEYS:

   ```
   $ ssh -i /path/to/ssh/key/set/in/ssh_authorized_keys -p 9999 core@localhost
   ```

3. Get the quay-builder service logs:

   ```
   $ systemctl status quay-builder
   $ journalctl -f -u quay-builder
   ```

   Step 2-3 can also be done in a single SSH command:

   ```
   $ ssh -i /path/to/ssh/key/set/in/ssh_authorized_keys -p 9999 core@localhost 'systemctl status quay-builder'
   $ ssh -i /path/to/ssh/key/set/in/ssh_authorized_keys -p 9999 core@localhost 'journalctl -f -u quay-builder'
   ```

6.7. SETTING UP GITHUB BUILDS (OPTIONAL)

If your organization plans to have builds be conducted via pushes to GitHub (or GitHub Enterprise), continue with Creating an OAuth application in GitHub.
CHAPTER 7. BUILDING DOCKERFILES

Red Hat Quay supports the ability to build Dockerfiles on our build fleet and push the resulting image to the repository.

7.1. VIEWING AND MANAGING BUILDS

Repository Builds can be viewed and managed by clicking the Builds tab in the Repository View.

7.2. MANUALLY STARTING A BUILD

To manually start a repository build, click the + icon in the top right of the header on any repository page and choose New Dockerfile Build. An uploaded Dockerfile, .tar.gz, or an HTTP URL to either can be used for the build.

NOTE
You will not be able to specify the Docker build context when manually starting a build.

7.3. BUILD TRIGGERS

Repository builds can also be automatically triggered by events such as a push to an SCM (GitHub, BitBucket or GitLab) or via a call to a webhook.

7.3.1. Creating a new build trigger

To setup a build trigger, click the Create Build Trigger button on the Builds view page and follow the instructions of the dialog. You will need to grant Red Hat Quay access to your repositories in order to setup the trigger and your account requires admin access on the SCM repository.

7.3.2. Manually triggering a build trigger

To trigger a build trigger manually, click the icon next to the build trigger and choose Run Now.

7.3.3. Build Contexts

When building an image with Docker, a directory is specified to become the build context. This holds true for both manual builds and build triggers because the builds conducted by Red Hat Quay are no different from running docker build on your own machine.

Red Hat Quay build contexts are always the specified subdirectory from the build setup and fallback to the root of the build source if none is specified. When a build is triggered, Red Hat Quay build workers clone the git repository to the worker machine and enter the build context before conducting a build.

For builds based on tar archives, build workers extract the archive and enter the build context. For example:

```
example
  ├── .git
  └── Dockerfile
```
Imagine the example above is the directory structure for a GitHub repository called "example". If no subdirectory is specified in the build trigger setup or while manually starting a build, the build will operate in the example directory.

If `subdir` is specified to be the subdirectory in the build trigger setup, only the Dockerfile within it is visible to the build. This means that you cannot use the `ADD` command in the Dockerfile to add `file`, because it is outside of the build context.

Unlike the Docker Hub, the Dockerfile is part of the build context on Red Hat Quay. Thus, it must not appear in the `.dockerignore` file.
CHAPTER 8. SETTING UP A CUSTOM GIT TRIGGER

A Custom Git Trigger is a generic way for any git server to act as a build trigger. It relies solely on SSH keys and webhook endpoints; everything else is left to the user to implement.

8.1. CREATING A TRIGGER

Creating a Custom Git Trigger is similar to the creation of any other trigger with a few subtle differences:

- It is not possible for Red Hat Quay to automatically detect the proper robot account to use with the trigger. This must be done manually in the creation process.
- There are extra steps after the creation of the trigger that must be done in order to use the trigger. These steps are detailed below.

8.2. POST TRIGGER-CREATION SETUP

Once a trigger has been created, there are 2 additional steps required before the trigger can be used:

- Provide read access to the SSH public key generated when creating the trigger.
- Setup a webhook that POSTs to the Red Hat Quay endpoint to trigger a build.

The key and the URL are both available at all times by selecting View Credentials from the gear located in the trigger listing.

Trigger Credentials

In order to use this trigger, the following first requires action:
- You must give the following public key read access to the git repository.
- You must set your repository to POST to the following URL to trigger a build.
For more information, refer to the Custom Git Triggers documentation.

SSH Public Key:

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDv2pbbxUd8il1wCExfL3LMUEwze8xm3CV9

Webhook Endpoint URL:

http://%24token:NJKMIE8A2597KBPV2W2TJ2R6VNX3X2E3ZK5l3T6JEKRHKSSA5VKD64EP

8.2.1. SSH public key access
Depending on the Git server setup, there are various ways to install the SSH public key that Red Hat Quay generates for a custom git trigger. For example, Git documentation describes a small server setup in which simply adding the key to $HOME/.ssh/authorize_keys would provide access for builders to clone the repository. For any git repository management software that isn’t officially supported, there is usually a location to input the key often labeled as Deploy Keys.

8.2.2. Webhook

In order to automatically trigger a build, one must POST a JSON payload to the webhook URL with the following format:

```json
{
  "commit": "1c002dd", // required
  "ref": "refs/heads/master", // required
  "default_branch": "master", // required
  "commit_info": {
    "url": "gitsoftware.com/repository/commits/1234567", // required
    "message": "initial commit", // required
    "date": "timestamp", // required
    "author": { // optional
      "username": "user", // required
      "avatar_url": "gravatar.com/user.png", // required
      "url": "gitsoftware.com/users/user" // required
    },
    "committer": { // optional
      "username": "user", // required
      "avatar_url": "gravatar.com/user.png", // required
      "url": "gitsoftware.com/users/user" // required
    }
  }
}
```

NOTE

This request requires a Content-Type header containing application/json in order to be valid.

Once again, this can be accomplished in various ways depending on the server setup, but for most cases can be done via a post-receive git hook.
CHAPTER 9. SKIPPING A SOURCE CONTROL-TRIGGERED BUILD

To specify that a commit should be ignored by the Red Hat Quay build system, add the text [skip build] or [build skip] anywhere in the commit message.
CHAPTER 10. SET UP GITHUB BUILD TRIGGER TAGS

Red Hat Quay supports using GitHub or GitHub Enterprise as a trigger to building images. If you have not yet done so, go ahead and enable build support in Red Hat Quay.

10.1. UNDERSTANDING TAG NAMING FOR BUILD TRIGGERS

Prior to Red Hat Quay 3.3, how images created from build triggers were named was limited. Images built by build triggers were named:

- With the branch or tag whose change invoked the trigger
- With a latest tag for images that used the default branch

As of Red Hat Quay 3.3 and later, you have more flexibility in how you set image tags. The first thing you can do is enter custom tags, to have any string of characters assigned as a tag for each built image. However, as an alternative, you could use the following tag templates to tag images with information from each commit:

- \${commit_info.short_sha}: The commit’s short SHA
- \${commit_info.date}: The timestamp for the commit
- \${commit_info.author}: The author from the commit
- \${commit_info.committer}: The committer of the commit
- \${parsed_ref.branch}: The branch name

The following procedure describes how you set up tagging for build triggers.

10.2. SETTING TAG NAMES FOR BUILD TRIGGERS

Follow these steps to configure custom tags for build triggers:

1. From the repository view, select the Builds icon from the left navigation.

2. Select the Create Build Trigger menu, and select the type of repository push you want (GitHub, Bitbucket, GitLab, or Custom Git repository push). For this example, GitHub Repository Push is chosen, as illustrated in the following figure.
3. When the Setup Build Trigger page appears, select the repository and namespace in which you want the trigger set up.

4. Under Configure Trigger, select either Trigger for all branches and tags or Trigger only on branches and tags matching a regular expression. Then select Continue. The Configure Tagging section appears, as shown in the following figure:

5. Scroll down to Configure Tagging and select from the following options:

   - **Tag manifest with the branch or tag name** Check this box to use the name of the branch or tag in which the commit occurred as the tag used on the image. This is enabled by default.

   - **Add latest tag if on default branch** Check this box to use the latest tag for the image if it is on the default branch for the repository. This is enabled by default.

   - **Add custom tagging templates** Enter a custom tag or a template into the Enter a tag template box. There are multiple tag templates you can enter here, as described earlier in this section. They include ways of using short SHA, timestamps, author name, committer, and branch name from the commit as tags.

6. Select Continue. You are prompted to select the directory build context for the Docker build. The build context directory identifies the location of the directory containing the Dockerfile, along with other files needed when the build is triggered. Enter "/" if the Dockerfile is in the root
of the git repository.

7. Select Continue. You are prompted to add an optional Robot Account. Do this if you want to pull a private base image during the build process. The robot account would need access to the build.

8. Select Continue to complete the setup of the build trigger.

If you were to return to the Repository Builds page for the repository, the build triggers you set up will be listed under the Build Triggers heading.
CHAPTER 11. CREATING AN OAUTH APPLICATION IN GITHUB

You can authorize your registry to access a GitHub account and its repositories by registering it as a GitHub OAuth application.

11.1. CREATE NEW GITHUB APPLICATION

1. Log into GitHub (Enterprise)

2. Visit the Applications page under your organization’s settings.

3. Click Register New Application. The Register a new OAuth application configuration screen is displayed:

   ![Register a new OAuth application configuration screen](image)

   4. Set Homepage URL: Enter the Quay Enterprise URL as the Homepage URL. **NOTE** If using public GitHub, the Homepage URL entered must be accessible by your users. It can still be an internal URL.

   5. Set Authorization callback URL: Enter $RED_HAT_QUAY_URL/oauth2/github/callback as the Authorization callback URL.

   6. Save your settings by clicking the Register application button. The new application’s summary is shown:

   7. Record the Client ID and Client Secret shown for the new application.
CHAPTER 12. REPOSITORY NOTIFICATIONS

Quay supports adding notifications to a repository for various events that occur in the repository’s lifecycle. To add notifications, click the Settings tab while viewing a repository and select Create Notification. From the When this event occurs field, select the items for which you want to receive notifications:

After selecting an event, further configure it by adding how you will be notified of that event.

**NOTE**

Adding notifications requires repository admin permission.

The following are examples of repository events.

### 12.1. REPOSITORY EVENTS

#### 12.1.1. Repository Push

A successful push of one or more images was made to the repository:

```json
{
   "name": "repository",
   "repository": "dgangaia/test",
   "namespace": "dgangaia",
   "docker_url": "quay.io/dgangaia/test",
   "homepage": "https://quay.io/repository/dgangaia/repository",
   "updated_tags": [
      "latest"
   ]
}
```

#### 12.1.2. Dockerfile Build Queued

Here is a sample response for a Dockerfile build has been queued into the build system. The response can differ based on the use of optional attributes.

```json
{
   "build_id": "296ec063-5f86-4706-a469-f0a400bf9df2",
}
12.1.3. Dockerfile Build Started

Here is an example of a Dockerfile build being started by the build system. The response can differ based on some attributes being optional.
12.1.4. Dockerfile Build Successfully Completed

Here is a sample response of a Dockerfile build that has been successfully completed by the build system.

**NOTE**

This event will occur *simultaneously* with a Repository Push event for the built image(s)
12.1.5. Dockerfile Build Failed

A Dockerfile build has failed

```json
{
  "build_id": "5346a21d-3434-4764-85be-5be1296f293c",
  "trigger_kind": "github", //Optional
  "name": "test",
  "repository": "dgangaia/test",
  "docker_url": "quay.io/dgangaia/test",
  "error_message": "Could not find or parse Dockerfile: unknown instruction: GIT",
  "namespace": "dgangaia",
  "trigger_id": "38b6e180-9521-4ff7-9844-acf371340b9e", //Optional
  "docker_tags": [
    "master",
    "latest"
  ],
  "build_name": "6ae9a86",
  "trigger_metadata": { //Optional
    "commit": "6ae9a86930fc73dd07b02e4c5bf63ee60be180ad",
    "ref": "refs/heads/master",
    "default_branch": "master",
    "git_url": "git@github.com:dgangaia/test.git",
    "commit_info": { //Optional
      "url": "https://github.com/dgangaia/test/commit/6ae9a86930fc73dd07b02e4c5bf63ee60be180ad",
      "date": "2019-03-06T14:18:16+11:00",
      "message": "adding 5",
      "committer": { //Optional
        "username": "web-flow",
        "url": "https://github.com/web-flow",
      }
    }
  }
}
```
"message": "failed build test",
"committer": { //Optional
  "username": "web-flow",
  "url": "https://github.com/web-flow", //Optional
},
"author": { //Optional
  "username": "dgangaia",
  "url": "https://github.com/dgangaia",
  "avatar_url": "https://avatars1.githubusercontent.com/u/43594254?v=4" //Optional
}
},
"homepage": "https://quay.io/repository/dgangaia/test/build/5346a21d-3434-4764-85be-5be1296f293c"
}

12.1.6. Dockerfile Build Cancelled

A Dockerfile build was cancelled

{
  "build_id": "cbd534c5-f1c0-4816-b4e3-55446b851e70",
  "trigger_kind": "github",
  "name": "test",
  "repository": "dgangaia/test",
  "namespace": "dgangaia",
  "docker_url": "quay.io/dgangaia/test",
  "trigger_id": "38b6e180-9521-4ff7-9844-acf371340b9e",
  "docker_tags": [
    "master",
    "latest"
  ],
  "build_name": "cbce83c",
  "trigger_metadata": {
    "commit": "cbce83c04bfbb59734fc42a83aab738704ba7ec41",
    "ref": "refs/heads/master",
    "default_branch": "master",
    "git_url": "git@github.com:dgangaia/test.git",
    "commit_info": {
      "url": "https://github.com/dgangaia/testcommit/cbce83c04bfbb59734fc42a83aab738704ba7ec41",
      "date": "2019-03-06T14:27:53+11:00",
      "message": "testing cancel build",
      "committer": {
        "username": "web-flow",
        "url": "https://github.com/web-flow",
        "avatar_url": "https://avatars3.githubusercontent.com/u/19864447?v=4"
      },
      "author": {
        "username": "dgangaia",
        "url": "https://github.com/dgangaia",
        "avatar_url": "https://avatars1.githubusercontent.com/u/43594254?v=4"
      }
    }
  }
}
12.1.7. Vulnerability Detected

A vulnerability was detected in the repository

```
{
  "repository": "dgangaia/repository",
  "namespace": "dgangaia",
  "name": "repository",
  "docker_url": "quay.io/dgangaia/repository",
  "homepage": "https://quay.io/repository/dgangaia/repository",

  "tags": ["latest", "othertag"],

  "vulnerability": {
    "id": "CVE-1234-5678",
    "description": "This is a bad vulnerability",
    "link": "http://url/to/vuln/info",
    "priority": "Critical",
    "has_\_fix": true
  }
}
```

12.2. NOTIFICATION ACTIONS

12.2.1. Quay Notification

A notification will be added to the Quay.io notification area. The notification area can be found by clicking on the bell icon in the top right of any Quay.io page.

Quay.io notifications can be setup to be sent to a User, Team, or the organization as a whole.

12.2.2. E-mail

An e-mail will be sent to the specified address describing the event that occurred.

**NOTE**

All e-mail addresses will have to be verified on a per-repository basis

12.2.3. Webhook POST

An HTTP POST call will be made to the specified URL with the event’s data (see above for each event’s data format).

When the URL is HTTPS, the call will have an SSL client certificate set from Quay.io. Verification of this certificate will prove the call originated from Quay.io. Responses with status codes in the 2xx range are considered successful. Responses with any other status codes will be considered failures and result in a retry of the webhook notification.
12.2.4. Flowdock Notification
Posts a message to Flowdock.

12.2.5. Hipchat Notification
Posts a message to HipChat.

12.2.6. Slack Notification
Posts a message to Slack.
CHAPTER 13. OCI SUPPORT AND RED HAT QUAY

Container registries such as Red Hat Quay were originally designed to support container images in the Docker image format. To promote the use of additional runtimes apart from Docker, the Open Container Initiative (OCI) was created to provide a standardization surrounding container runtimes and image formats. Most container registries support the OCI standardization as it is based on the Docker image manifest V2, Schema 2 format.

In addition to container images, a variety of artifacts have emerged that support not just individual applications, but the Kubernetes platform as a whole. These range from Open Policy Agent (OPA) policies for security and governance to Helm charts and Operators to aid in application deployment.

Red Hat Quay is a private container registry that not only stores container images, but supports an entire ecosystem of tooling to aid in the management of containers. Support for OCI based artifacts in Red Hat Quay 3.6 has extended from solely Helm to include cosign and ztsd compression schemes by default. As such, **FEATURE_HELM_OCI_SUPPORT** has been deprecated.

When Red Hat Quay 3.6 is deployed using the OpenShift Operator, support for Helm and OCI artifacts is now enabled by default under the **FEATURE_GENERAL_OCI_SUPPORT** configuration. If you need to explicitly enable the feature, for example, if it has previously been disabled or if you have upgraded from a version where it is not enabled by default, see the section **Explicitly enabling OCI and Helm support**.

13.1. HELM AND OCI PREREQUISITES

- **Trusted certificates:** Communication between the Helm client and Quay is facilitated over HTTPS and as of Helm 3.5, support is only available for registries communicating over HTTPS with trusted certificates. In addition, the operating system must trust the certificates exposed by the registry. Support in future Helm releases will allow for communicating with remote registries insecurely. With that in mind, ensure that your operating system has been configured to trust the certificates used by Quay, for example:
  
  ```
  $ sudo cp rootCA.pem /etc/pki/ca-trust/source/anchors/
  $ sudo update-ca-trust extract
  ```

- **Generally available:** As of Helm 3.8, OCI registry support for charts is now generally available.

- **Install Helm client:** Download your desired version from the Helm releases page. Unpack it and move the helm binary to its desired destination:
  
  ```
  $ tar -zxvf helm-v3.8.2-linux-amd64.tar.gz
  $ mv linux-amd64/helm /usr/local/bin/helm
  ```

- **Create organization in Quay:** Create a new organization for storing the Helm charts, using the Quay registry UI. For example, create an organization named **helm**.

13.2. HELM CHARTS WITH RED HAT QUAY

Helm, as a graduated project of the Cloud Native Computing Foundation (CNCF), has become the de facto package manager for Kubernetes as it simplifies how applications are packaged and deployed. Helm uses a packaging format called Charts which contain the Kubernetes resources representing an application. Charts can be made available for general distribution and consumption in repositories. A Helm repository is an HTTP server that serves an **index.yaml** metadata file and optionally a set of packaged charts. Beginning with Helm version 3, support was made available for distributing charts in OCI registries as an alternative to a traditional repository.
13.2.1. Using Helm charts with Red Hat Quay

Use the following example to download and push an etherpad chart from the Red Hat Community of Practice (CoP) repository.

Procedure

1. Add a chart repository:
   
   `$ helm repo add redhat-cop https://redhat-cop.github.io/helm-charts`

2. Update the information of available charts locally from the chart repository:
   
   `$ helm repo update`

3. Download a chart from a repository:
   
   `$ helm pull redhat-cop/etherpad --version=0.0.4 --untar`

4. Package the chart into a chart archive:
   
   `$ helm package ./etherpad`

   Example output

   Successfully packaged chart and saved it to: /home/user/linux-amd64/etherpad-0.0.4.tgz

5. Log in to your Quay repository using `helm registry login`:
   
   `$ helm registry login quay370.apps.quayperf370.perfscale.devcluster.openshift.com`

6. Push the chart to your Quay repository using the `helm push` command:
   
   `$ helm push etherpad-0.0.4.tgz
   
   oci://quay370.apps.quayperf370.perfscale.devcluster.openshift.com`

   Example output:

   Pushed:  quay370.apps.quayperf370.perfscale.devcluster.openshift.com/etherpad:0.0.4
   Digest:  sha256:a6667ff2a0e2bd7aa4813db9ac854b5124ff1c458d170b70c2d2375325f2451b

7. Ensure that the push worked by deleting the local copy, and then pulling the chart from the repository:
   
   `$ rm -rf etherpad-0.0.4.tgz`

   `$ helm pull oci://quay370.apps.quayperf370.perfscale.devcluster.openshift.com/etherpad --version 0.0.4`

   Example output:
13.3. OCI AND HELM CONFIGURATION FIELDS

Support for Helm is now supported under the FEATURE_GENERAL_OCI_SUPPORT property. If you need to explicitly enable the feature, for example, if it has previously been disabled or if you have upgraded from a version where it is not enabled by default, you need to add two properties in the Quay configuration to enable the use of OCI artifacts:

```
FEATURE_GENERAL_OCI_SUPPORT: true
FEATURE_HELM_OCI_SUPPORT: true
```

Table 13.1. OCI and Helm configuration fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEATURE_GENERAL_OCI_SUPPORT</td>
<td>Boolean</td>
<td>Enable support for OCI artifacts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: True</td>
</tr>
<tr>
<td>FEATURE_HELM_OCI_SUPPORT</td>
<td>Boolean</td>
<td>Enable support for Helm artifacts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: True</td>
</tr>
</tbody>
</table>

**IMPORTANT**

As of Red Hat Quay 3.6, FEATURE_HELM_OCI_SUPPORT has been deprecated and will be removed in a future version of Red Hat Quay. In Red Hat Quay 3.6, Helm artifacts are supported by default and included under the FEATURE_GENERAL_OCI_SUPPORT property. Users are no longer required to update their config.yaml files to enable support.

13.4. COSIGN OCI SUPPORT WITH RED HAT QUAY

Cosign is a tool that can be used to sign and verify container images. It uses the ECDSA-P256 signature algorithm and Red Hat’s Simple Signing payload format to create public keys that are stored in PKIX files. Private keys are stored as encrypted PEM files.

Cosign currently supports the following:

- Hardware and KMS Signing
- Bring-your-own PKI
- OIDC PKI
- Built-in binary transparency and timestamping service

13.5. USING COSIGN WITH QUAY

If you have Go 1.16+, you can directly install cosign with the following command:
$ go install github.com/sigstore/cosign/cmd/cosign@v1.0.0
go: downloading github.com/sigstore/cosign v1.0.0
go: downloading github.com/peterbourgon/ff/v3 v3.1.0
...

Next, generate a keypair:

$ cosign generate-key-pair
Enter password for private key:
Enter again:
Private key written to cosign.key
Public key written to cosign.pub

Sign the keypair with the following command:

$ cosign sign -key cosign.key quay-server.example.com/user1/busybox:test
Enter password for private key:
Pushing signature to: quay-server.example.com/user1/busybox:sha256-ff13b8f6f289b92ec2913fa57c5dd0a874c3a7f8f149aabee50e3d01546473e3.sig

Some users may experience the following error:

error: signing quay-server.example.com/user1/busybox:test: getting remote image: GET https://quay-server.example.com/v2/user1/busybox/manifests/test: UNAUTHORIZED: access to the requested resource is not authorized; map[]

Because cosign relies on ~/.docker/config.json for authorization, you might need to execute the following command:

$ podman login --authfile ~/.docker/config.json quay-server.example.com
Username:
Password:
Login Succeeded!

You can see the updated authorization configuration using the following command:

$ cat ~/.docker/config.json
{  
  "auths": {  
    "quay-server.example.com": {  
      "auth": "cXVheWFkbWluOnBhc3N3b3Jk"
    }  
  }  
}

13.6. ADDING OTHER OCI MEDIA TYPES TO QUAY

Helm, cosign, and ztsd compression scheme artifacts are built into Red Hat Quay 3.6 by default. For any other OCI media type that is not supported by default, you can add them to the ALLOWED_OCI_ARTIFACT_TYPES configuration in Quay’s config.yaml using the following format:

ALLOWED_OCI_ARTIFACT_TYPES:
  <oci config type 1>:
    - <oci layer type 1>
For example, you can add Singularity (SIF) support by adding the following to your config.yaml:

```yaml
ALLOWED_OCI_ARTIFACT_TYPES:
  application/vnd.oci.image.config.v1+json:
    - application/vnd.dev.cosign.simplesigning.v1+json
  application/vnd.cncf.helm.config.v1+json:
    - application/tar+gzip
    - application/vnd.sylabs.sif.config.v1+json:
      - application/vnd.sylabs.sif.layer.v1+tar
```

**NOTE**

When adding OCI media types that are not configured by default, users will also need to manually add support for cosign and Helm if desired. The ztsd compression scheme is supported by default, so users will not need to add that OCI media type to their config.yaml to enable support.

### 13.7. DISABLING OCI ARTIFACTS IN QUAY

If you want to disable OCI artifact support, you can set `FEATURE_GENERAL_OCI_SUPPORT` to `False` in your config.yaml:

```yaml
FEATURE_GENERAL_OCI_SUPPORT = False
```
CHAPTER 14. RED HAT QUAY QUOTA MANAGEMENT AND ENFORCEMENT

With Red Hat Quay 3.7, users have the ability to report storage consumption and to contain registry growth by establishing configured storage quota limits. On-premise Quay users are now equipped with the following capabilities to manage the capacity limits of their environment:

- **Quota reporting**: With this feature, a superuser can track the storage consumption of all their organizations. Additionally, users can track the storage consumption of their assigned organization.

- **Quota management**: With this feature, a superuser can define soft and hard checks for Red Hat Quay users. Soft checks tell users if the storage consumption of an organization reaches their configured threshold. Hard checks prevent users from pushing to the registry when storage consumption reaches the configured limit.

Together, these features allow service owners of a Quay registry to define service level agreements and support a healthy resource budget.

14.1. QUOTA MANAGEMENT ARCHITECTURE

The `RepositorySize` database table holds the storage consumption, in bytes, of a Red Hat Quay repository within an organization. The sum of all repository sizes for an organization defines the current storage size of a Red Hat Quay organization. When an image push is initialized, the user’s organization storage is validated to check if it is beyond the configured quota limits. If an image push exceeds defined quota limitations, a soft or hard check occurs:

- For a soft check, users are notified.
- For a hard check, the push is stopped.

If storage consumption is within configured quota limits, the push is allowed to proceed.

Image manifest deletion follows a similar flow, whereby the links between associated image tags and the manifest are deleted. Additionally, after the image manifest is deleted, the repository size is recalculated and updated in the `RepositorySize` table.

14.2. QUOTA MANAGEMENT LIMITATIONS

Quota management helps organizations to maintain resource consumption. One limitation of quota management is that calculating resource consumption on push results in the calculation becoming part of the push’s critical path. Without this, usage data might drift.

The maximum storage quota size is dependent on the selected database:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Postgres</td>
<td>8388608 TB</td>
</tr>
<tr>
<td>MySQL</td>
<td>8388608 TB</td>
</tr>
</tbody>
</table>
14.3. 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
```

**NOTE**

In Red Hat Quay 3.7, superuser privileges are required to create, update and delete quotas. While quotas can be set for users as well as organizations, you cannot reconfigure the user quota using the Red Hat Quay UI and you must use the API instead.

14.3.1. Default quota

To specify a system-wide default storage quota that is applied to every organization and user, use the `DEFAULT_SYSTEM_REJECT_QUOTA_BYTES` configuration flag.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT_SYSTEM_REJECT_QUOTA_BYTES</td>
<td>String</td>
<td>The quota size to apply to all organizations and users.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>By default, no limit is set.</td>
</tr>
</tbody>
</table>

If you configure a specific quota for an organization or user, and then delete that quota, the system-wide default quota will apply if one has been set. Similarly, if you have configured a specific quota for an organization or user, and then modify the system-wide default quota, the updated system-wide default will override any specific settings.

14.4. ESTABLISHING QUOTA WITH THE RED HAT QUAY API

When an organization is first created, it does not have a quota applied. Use the `/api/v1/organization/[organization]/quota` endpoint:

**Sample command**

```
```

**Sample output**

```
[]
```
14.4.1. Setting the quota

To set a quota for an organization, POST data to the /api/v1/organization/{orgname}/quota endpoint:

```
```

Sample output

"Created"

14.4.2. Viewing the quota

To see the applied quota, GET data from the /api/v1/organization/{orgname}/quota endpoint:

```
```

Sample output

```
[
  {
    "id": 1,
    "limit_bytes": 10485760,
    "default_config": false,
    "limits": [],
    "default_config_exists": false
  }
]
```

14.4.3. Modifying the quota

To change the existing quota, in this instance from 10 MB to 100 MB, PUT data to the /api/v1/organization/{orgname}/quota/{quota_id} endpoint:

```
```

Sample output

```
{
  "id": 1,
```
14.4.4. Pushing images

To see the storage consumed, push various images to the organization.

14.4.4.1. Pushing ubuntu:18.04

Push ubuntu:18.04 to the organization from the command line:

Sample commands

```
$ podman pull ubuntu:18.04
$ podman tag docker.io/library/ubuntu:18.04 example-registry-quay-quay-enterprise.apps.docs.gcp.quaydev.org/testorg/ubuntu:18.04
$ podman push --tls-verify=false example-registry-quay-quay-enterprise.apps.docs.gcp.quaydev.org/testorg/ubuntu:18.04
```

14.4.4.2. Using the API to view quota usage

To view the storage consumed, GET data from the /api/v1/repository endpoint:

Sample command

```
```

Sample output

```
{
  "repositories": [
    {
      "namespace": "testorg",
      "name": "ubuntu",
      "description": null,
      "is_public": false,
      "kind": "image",
      "state": "NORMAL",
      "quota_report": {
        "quota_bytes": 27959066,
        "configured_quota": 104857600
      },
      "last_modified": 1651225630,
      "popularity": 0,
      "is_starred": false
    }
  ]
}
```
14.4.4.3. Pushing another image

1. Pull, tag, and push a second image, for example, nginx:

   **Sample commands**

   ```
   $ podman pull nginx
   $ podman tag docker.io/library/nginx example-registry-quay-quay-enterprise.apps.docs.gcp.quaydev.org/testorg/nginx
   $ podman push --tls-verify=false example-registry-quay-quay-enterprise.apps.docs.gcp.quaydev.org/testorg/nginx
   ```

2. To view the quota report for the repositories in the organization, use the `/api/v1/repository` endpoint:

   **Sample command**

   ```
   ```

   **Sample output**

   ```json
   {
   "repositories": [
   {
   "namespace": "testorg",
   "name": "ubuntu",
   "description": null,
   "is_public": false,
   "kind": "image",
   "state": "NORMAL",
   "quota_report": {
   "quota_bytes": 27959066,
   "configured_quota": 104857600
   },
   "last_modified": 1651225630,
   "popularity": 0,
   "is_starred": false
   },
   {
   "namespace": "testorg",
   "name": "nginx",
   "description": null,
   "is_public": false,
   "kind": "image",
   "state": "NORMAL",
   "quota_report": {
   ```
3. To view the quota information in the organization details, use the 
/api/v1/organization/{orgname} endpoint:

Sample command

```
$ curl -k -X GET -H "Authorization: Bearer <token>" -H 'Content-Type: application/json'
'https://example-registry-quay-quay-enterprise.apps.docs.gcp.quaydev.org/api/v1/organization/testorg' | jq
```

Sample output

```
{
  "name": "testorg",
  ...
  "quotas": [
    {
      "id": 1,
      "limit_bytes": 104857600,
      "limits": []
    }
  ],
  "quota_report": {
    "quota_bytes": 87190725,
    "configured_quota": 104857600
  }
}
```

14.4.5. Rejecting pushes using quota limits

If an image push exceeds defined quota limitations, a soft or hard check occurs:

- For a soft check, or `warning`, users are notified.
- For a hard check, or `reject`, the push is terminated.

14.4.5.1. Setting reject and warning limits

To set `reject` and `warning` limits, POST data to the
/api/v1/organization/{orgname}/quota/{quota_id}/limit endpoint:

Sample reject limit command

```
"quota_bytes": 59231659,
"configured_quota": 104857600
},
"last_modified": 1651229507,
"popularity": 0,
"is_starred": false}
```


Sample warning limit command


14.4.5.2. Viewing reject and warning limits

To view the reject and warning limits, use the /api/v1/organization/{orgname}/quota endpoint:

View quota limits


Sample output for quota limits

```json
[
  {
    "id": 1,
    "limit_bytes": 104857600,
    "default_config": false,
    "limits": [
      {
        "id": 2,
        "type": "Warning",
        "limit_percent": 50
      },
      {
        "id": 1,
        "type": "Reject",
        "limit_percent": 80
      }
    ],
    "default_config_exists": false
  }
]
```

14.4.5.3. Pushing an image when the reject limit is exceeded

In this example, the reject limit (80%) has been set to below the current repository size (~83%), so the next push should automatically be rejected.

Push a sample image to the organization from the command line:

Sample image push

$ podman pull ubuntu:20.04
When limits are exceeded, a notification appears:

**Quota notifications**

14.4.5.4. Notifications for limits exceeded

When limits are exceeded, a notification appears:

**Quota notifications**
CHAPTER 15. RED HAT QUAY AS A PROXY CACHE FOR UPSTREAM REGISTRIES

With the growing popularity of container development, customers increasingly rely on container images from upstream registries like Docker or Google Cloud Platform to get services up and running. Today, registries have rate limitations and throttling on the number of times users can pull from these registries.

With this feature, Red Hat Quay will act as a proxy cache to circumvent pull-rate limitations from upstream registries. Adding a cache feature also accelerates pull performance, because images are pulled from the cache rather than upstream dependencies. Cached images are only updated when the upstream image digest differs from the cached image, reducing rate limitations and potential throttling.

With Red Hat Quay cache proxy, the following features are available:

- Specific organizations can be defined as a cache for upstream registries.
- Configuration of a Quay organization that acts as a cache for a specific upstream registry. This repository can be defined by using the Quay UI, and offers the following configurations:
  - Upstream registry credentials for private repositories or increased rate limiting.
  - Expiration timer to avoid surpassing cache organization size.
- Global on/off configurable via the configuration application.
- Caching of entire upstream registries or just a single namespace, for example, all of docker.io or just docker.io/library.
- Logging of all cache pulls.
- Cached images scannability by Clair.

15.1. PROXY CACHE ARCHITECTURE

The following image shows the expected design flow and architecture of the proxy cache feature.
When a user pulls an image, for example, **postgres:14**, from an upstream repository on Red Hat Quay, the repository checks to see if an image is present. If the image does not exist, a fresh pull is initiated. After being pulled, the image layers are saved to cache and server to the user in parallel. The following image depicts an architectural overview of this scenario:

If the image in the cache exists, users can rely on Quay’s cache to stay up-to-date with the upstream source so that newer images from the cache are automatically pulled. This happens when tags of the original image have been overwritten in the upstream registry. The following image depicts an architectural overview of what happens when the upstream image and cached version of the image are different:
If the upstream image and cached version are the same, no layers are pulled and the cached image is delivered to the user.

In some cases, users initiate pulls when the upstream registry is down. If this happens with the configured staleness period, the image stored in cache is delivered. If the pull happens after the configured staleness period, the error is propagated to the user. The following image depicts an architectural overview when a pull happens after the configured staleness period:
Quay administrators can leverage the configurable size limit of an organization to limit cache size so that backend storage consumption remains predictable. This is achieved by discarding images from the cache according to the frequency in which an image is used. The following image depicts an architectural overview of this scenario:

### 15.2. PROXY CACHE LIMITATIONS

Proxy caching with Red Hat Quay has the following limitations:

- Your proxy cache must have a size limit of greater than, or equal to, the image you want to cache. For example, if your proxy cache organization has a maximum size of 500 MB, and the image a user wants to pull is 700 MB, the image will be cached and will overflow beyond the configured limit.

- Cached images must have the same properties that images on a Quay repository must have.
15.3. USING RED HAT QUAY TO PROXY A REMOTE REGISTRY

The following procedure describes how you can use Red Hat Quay to proxy a remote registry. This procedure is set up to proxy quay.io, which allows users to use podman to pull any public image from any namespace on quay.io.

Prerequisites

- FEATURE_PROXY_CACHE in your config.yaml is set to true.
- Assigned the Member team role. For more information about team roles, see Users and organizations in Red Hat Quay.

Procedure

1. In your Quay organization on the UI, for example, cache-quayio, click Organization Settings on the left hand pane.
2. Optional: Click Add Storage Quota to configure quota management for your organization. For more information about quota management, see Quota Management.

   **NOTE**
   In some cases, pulling images with Podman might return the following error when quota limit is reached during a pull: unable to pull image: Error parsing image configuration: Error fetching blob: invalid status code from registry 403 (Forbidden). Error 403 is inaccurate, and occurs because Podman hides the correct API error: Quota has been exceeded on namespace. This known issue will be fixed in a future Podman update.

3. In Remote Registry enter the name of the remote registry to be cached, for example, quay.io, and click Save.

   **NOTE**
   By adding a namespace to the Remote Registry, for example, quay.io/<namespace>, users in your organization will only be able to proxy from that namespace.


   **NOTE**
   If you do not set a Remote Registry Username and Remote Registry Password, you cannot add one without removing the proxy cache and creating a new registry.

5. Optional: Set a time in the Expiration field.
NOTE

- The default tag **Expiration** field for cached images in a proxy organization is set to 86400 seconds. In the proxy organization, the tag expiration is refreshed to the value set in the UI’s **Expiration** field every time the tag is pulled. This feature is different than Quay’s default individual tag expiration feature. In a proxy organization, it is possible to override the individual tag feature. When this happens, the individual tag’s expiration is reset according to the **Expiration** field of the proxy organization.

- Expired images will disappear after the allotted time, but are still stored in Quay. The time in which an image is completely deleted, or collected, depends on the **Time Machine** setting of your organization. The default time for garbage collection is 14 days unless otherwise specified.

6. Click **Save**.

7. On the CLI, pull a public image from the registry, for example, quay.io, acting as a proxy cache:
   ```bash
   $ podman pull <registry_url>/<organization_name>/<quayio_namespace>/<image_name>
   ```

   **IMPORTANT**
   If your organization is set up to pull from a single namespace in the remote registry, the remote registry namespace must be omitted from the URL. For example, `podman pull <registry_url>/<organization_name>/<image_name>`.

15.3.1. Leveraging storage quota limits in proxy organizations

With Red Hat Quay 3.8, the proxy cache feature has been enhanced with an auto-pruning feature for tagged images. The auto-pruning of image tags is only available when a proxied namespace has quota limitations configured. Currently, if an image size is greater than quota for an organization, the image is skipped from being uploaded until an administrator creates the necessary space. Now, when an image is pushed that exceeds the allotted space, the auto-pruning enhancement marks the least recently used tags for deletion. As a result, the new image tag is stored, while the least used image tag is marked for deletion.

**IMPORTANT**

- As part of the auto-pruning feature, the tags that are marked for deletion are eventually garbage collected by the garbage collector (gc) worker process. As a result, the quota size restriction is not fully enforced during this period.

- Currently, the namespace quota size computation does not take into account the size for manifest child. This is a known issue and will be fixed in a future version of Red Hat Quay.

15.3.1.1. Testing the storage quota limits feature in proxy organizations

Use the following procedure to test the auto-pruning feature of an organization with proxy cache and storage quota limitations enabled.

**Prerequisites**
- Your organization is configured to serve as a proxy organization. The following example proxies from quay.io.

- **FEATURE_PROXY_CACHE** is set to true in your `config.yaml` file.

- **FEATURE_QUOTA_MANAGEMENT** is set to true in your `config.yaml` file.

- Your organization is configured with a quota limit, for example, **150 MB**.

Procedure

1. Pull an image to your repository from your proxy organization, for example:

   ```
   $ podman pull quay-server.example.com/proxytest/projectquay/quay:3.7.9
   ```

2. Depending on the space left in your repository, you might need to pull additional images from your proxy organization, for example:

   ```
   $ podman pull quay-server.example.com/proxytest/projectquay/quay:3.6.2
   ```

3. In the Red Hat Quay registry UI, click the name of your repository.
   - Click Tags in the navigation pane and ensure that **quay:3.7.9** and **quay:3.6.2** are tagged.

4. Pull the last image that will result in your repository exceeding the allotted quota, for example:

   ```
   $ podman pull quay-server.example.com/proxytest/projectquay/quay:3.5.1
   ```

5. Refresh the Tags page of your Red Hat Quay registry. The first image that you pushed, for example, **quay:3.7.9** should have been auto-pruned. The Tags page should now show **quay:3.6.2** and **quay:3.5.1**.
CHAPTER 16. RED HAT QUAY BUILD ENHANCEMENTS

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.

16.1. 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.

16.2. 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.

16.3. CREATING A RED HAT QUAY BUILDERS ENVIRONMENT WITH OPENSPLIT CONTAINER PLATFORM

The procedures in this section explain how to create a Red Hat Quay virtual builders environment with OpenShift Container Platform.
16.3.1. OpenShift Container Platform TLS component

The **tls** component allows you to control TLS configuration.

**NOTE**

Red Hat Quay 3.8 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 use a wildcard.

To add the builder route, use the following format:

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

16.3.2. Using OpenShift Container Platform for Red Hat Quay builders

Builders require SSL/TLS certificates. For more information about SSL/TLS certificates, see Adding TLS certificates to the Red Hat Quay container.

If you are using Amazon Web Service (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.

16.3.2.1. Preparing OpenShift Container Platform for virtual builders

Use the following procedure to prepare OpenShift Container Platform for Red Hat Quay virtual builders.

**NOTE**

- 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.

**Procedure**

1. Log in to your Red Hat Quay cluster using a cluster administrator account.

2. Create a new project where your virtual builders will be run, for example, **virtual-builders**, by running the following command:

   ```
   $ oc new-project virtual-builders
   ```

3. Create a **ServiceAccount** in the project that will be used to run builds by entering the following command:

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

4. Provide the created service account with editing permissions so that it can run the build:
5. Grant the Quay builder **anyuid scc** permissions by entering the following command:

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

   ```bash
   oc sa get-token -n virtual-builders quay-builder
   ```

   b. If using OpenShift Container Platform 4.11 or later, enter the following command:

   ```bash
   $ oc create token quay-builder -n virtual-builders
   ```

   **Example output**

   ```bash
   eyJhbGciOiJSUzI1NiIsImtpZCI6IldfQUJkaDVmb3ltTHZ0dGZMYjhIWnYxZTQzN2dJVEJxcDJscldSdEUtYWsifQ...
   ```

7. Determine the builder route by entering the following command:

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

   **Example output**

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

8. Generate a self-signed SSL/TLS certificate with the .crt extension by entering the following command:

   ```bash
   $ oc extract cm/kube-root-ca.crt -n openshift-apiserver
   ```

   **Example output**

   ```bash
   ca.crt
   ```
9. Rename the `ca.crt` file to `extra_ca_cert_build_cluster.crt` by entering the following command:

```
$ mv ca.crt extra_ca_cert_build_cluster.crt
```

10. Locate the secret for your configuration bundle in the **Console**, and select **Actions → Edit Secret** and add the appropriate builder configuration:

```plaintext
FEATURE_USER_INITIALIZE: true
BROWSER_API_CALLS_XHR_ONLY: false
SUPER_USERS:
- <superusername>
FEATURE_USER CREATION: false
FEATURE QUOTA MANAGEMENT: true
FEATURE BUILD SUPPORT: True
BUILDMAN HOSTNAME: <sample_build_route>  
BUILD_MANAGER:
- ephemeral
- ALLOWED WORKER COUNT: 1
ORCHESTRATOR PREFIX: buildman/production/
JOB REGISTRATION TIMEOUT: 3600
ORCHESTRATOR:
  REDIS HOST: <sample_redis_hostname>
  REDIS PASSWORD: ***
  REDIS SSL: false
  REDIS SKIP KEYSERVICE EVENT SETUP: false
EXECUTORS:
- EXECUTOR: kubernetesPodman
  NAME: openshift
  BUILDERNAMESPACE: <sample_builder_namespace>
  SETUP TIME: 180
  MINIMUM RETRY THRESHOLD: 0
  BUILDER CONTAINER IMAGE: <sample_builder_container_image>
  # Kubernetes resource options
  K8S API SERVER: <sample_k8s_api_server>
  K8S_API_TLS_CA: <sample_crt_file>
  VOLUME SIZE: 8G
  KUBERNETES DISTRIBUTION: openshift
  CONTAINER MEMORY LIMITS: 300m
  CONTAINER CPU LIMITS: 1G
  CONTAINER MEMORY REQUEST: 300m
  CONTAINER CPU REQUEST: 1G
  NODE SELECTOR LABEL KEY: ""
  NODE SELECTOR LABEL VALUE: ""
  SERVICE ACCOUNT NAME: <sample_service_account_name>
  SERVICE ACCOUNT TOKEN: <sample_account_token>
```

1. The build route is obtained by running `oc get route -n` with the name of your OpenShift Operator’s namespace. A port must be provided at the end of the route, and it should use 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/TLS 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. If this happens, customers are alerted.
6. The `K8S_API_SERVER` is obtained by running `oc cluster-info`.
7. 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 configuration

```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: 0
    # Kubernetes resource options
    K8S_API_SERVER: api.docs.quayteam.org:6443
    # Red Hat Quay 3.8 Use Red Hat Quay
```

Red Hat Quay 3.8 Use Red Hat Quay
16.3.2.2. Manually adding SSL/TLS certificates

Due to a known issue with the configuration tool, you must manually add your custom SSL/TLS certificates to properly run builders. Use the following procedure to manually add custom SSL/TLS certificates.

For more information creating SSL/TLS certificates, see Adding TLS certificates to the Red Hat Quay container.

16.3.2.2.1. Creating and signing certificates

Use the following procedure to create and sign an SSL/TLS certificate.

Procedure

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

openssl.cnf

```
[req]
req_extensions = v3_req

[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
```

1. An alt_name for the URL of your Red Hat Quay registry must be included.
2. An alt_name for the BUILDMAN_HOSTNAME

Sample commands

```
$ openssl genrsa -out rootCA.key 2048
```
### 16.3.2.2.2. Setting TLS to unmanaged

Use the following procedure to set `kind:tls` to unmanaged.

**Procedure**

1. In your Red Hat Quay Registry YAML, set `kind: tls` to `managed: false`:

   ```yaml
   - kind: tls
     managed: false
   ```

2. On the *Events* page, the change is blocked until you set up the appropriate `config.yaml` file. For example:

   ```yaml
   - 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'
   ```

### 16.3.2.2.3. Creating temporary secrets

Use the following procedure to create temporary secrets for the CA certificate.

**Procedure**

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

   ```bash
   $ 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:

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

### 16.3.2.2.4. Copying secret data to the configuration YAML

Use the following procedure to copy secret data to your `config.yaml` file.

**Procedure**

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

2. For each secret, locate the YAML view:

   ```bash
   $ 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
   ```
Locate the secret for your Red Hat Quay registry configuration bundle in the UI, or through the command line by running a command like the following:

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

3. Locate the secret for your Red Hat Quay registry configuration bundle in the UI, or through the command line by running a command like the following:

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

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

```
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: >-
      LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSOtLS0tCk1JSURNakNDQWhxZ0F3SUJBZ0I....
    type: Opaque
```

```
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'

  data:
    ssl.cert: >-
      LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNSUlFcFFJQkFBs0NBuUVCb...
    ssl.key: >-
      LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNSUlFcFFJQkFBs0NBuUVCb...
    type: Opaque
```

```
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: >-
      LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSOtLS0tCk1JSURNakNDQWhxZ0F3SUJBZ0I....
    type: Opaque
```
5. Click **Save**.

6. Enter the following command to see if your pods are restarting:

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

   **Example 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-q7tv</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-1svs</td>
<td>0/1</td>
<td>Completed</td>
<td>0</td>
<td>6d1h</td>
</tr>
<tr>
<td>example-registry-quay-config-editor-77847fc4f5-nsbbv</td>
<td>0/1</td>
<td>ContainerCreating</td>
<td>0</td>
<td>2s</td>
</tr>
<tr>
<td>example-registry-quay-config-editor-c6c4d9cc-2mg2</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>
<tr>
<td>example-registry-quay-mirror-764d7b68d-jmlkk</td>
<td>1/1</td>
<td>Terminating</td>
<td>0</td>
<td>5d21h</td>
</tr>
<tr>
<td>example-registry-quay-mirror-764d7b68d-jqzwg</td>
<td>1/1</td>
<td>Terminating</td>
<td>0</td>
<td>5d21h</td>
</tr>
<tr>
<td>example-registry-quay-redis-7cc5f6c977-956g8</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>5d21h</td>
</tr>
</tbody>
</table>

7. After your Red Hat Quay registry has reconfigured, enter the following command to check if the Red Hat Quay app pods are running:

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

   **Example 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-sz6kb</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>7m45s</td>
</tr>
<tr>
<td>example-registry-quay-app-6786987b99-vgg2v</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>9m1s</td>
</tr>
<tr>
<td>example-registry-quay-app-upgrade-1svs</td>
<td>0/1</td>
<td>Completed</td>
<td>0</td>
<td>6d1h</td>
</tr>
<tr>
<td>example-registry-quay-config-editor-77847fc4f5-nsbbv</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>9m1s</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>
<tr>
<td>example-registry-quay-mirror-758fc68f5-7wshx</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>8m29s</td>
</tr>
<tr>
<td>example-registry-quay-mirror-758fc68f7-lbll2</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>8m29s</td>
</tr>
<tr>
<td>example-registry-quay-redis-7cc5f6c977-956g8</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>5d21h</td>
</tr>
</tbody>
</table>

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

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

**16.3.2.3. Using the UI to create a build trigger**
Use the following procedure to use the UI to create a build trigger.

**Procedure**

1. Log in to your Red Hat Quay repository.
2. Click **Create New Repository** and create a new registry, for example, testrepo.
3. On the **Repositories** page, click the **Builds** tab on the navigation pane. Alternatively, use the corresponding URL directly:

   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`. For example:

   ```bash
   $ oc get pods -n virtual-builders
   ```

   **Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>f192fe4a-c802-4275-bcfe-d2031e63526-92b5-25lg2</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>7s</td>
</tr>
</tbody>
</table>
$ oc get pods -n virtual-builders

**Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>f192fe4a-c802-4275-bcce-d2031e635126-9l2b5-25lg2</td>
<td>1/1</td>
<td>Terminating</td>
<td>0</td>
<td>9s</td>
</tr>
</tbody>
</table>

$ oc get pods -n virtual-builders

**Example output**

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 navigation pane.

**NOTE**

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

### 16.3.2.4. Modifying your AWS S3 storage bucket

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

**Procedure**

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
}
```
16.3.2.5. Modifying your Google Cloud Platform object bucket

Use the following procedure to configure cross-origin resource sharing (CORS) for virtual builders.

**NOTE**

Without CORS configuration, uploading a build Dockerfile fails.

**Procedure**

1. Use the following reference to create a JSON file for your specific CORS needs. For example:

   ```
   $ cat gcp_cors.json
   ``

   **Example output**

   ```
   [
    {
      "origin": ["*"],
      "method": ["GET"],
      "responseHeader": ["Authorization"],
      "maxAgeSeconds": 3600
    },
    {
      "origin": ["*"],
      "method": ["PUT"],
      "responseHeader": [
        "Content-Type",
        "x-goog-acl",
        "origin"],
      "maxAgeSeconds": 3600
    }
   ]
   ```

2. Enter the following command to update your GCP storage bucket:

   ```
   $ gcloud storage buckets update gs://<bucket_name> --cors-file=./gcp_cors.json
   ```
Example output

Updating
Completed 1

3. You can display the updated CORS configuration of your GCP bucket by running the following command:

```
$ gcloud storage buckets describe gs://<bucket_name> --format="default(cors)"
```

Example output

cors:
- maxAgeSeconds: 3600
  method:
  - GET
  origin:
    - '*'
  responseHeader:
    - Authorization
    - maxAgeSeconds: 3600
    - PUT
    origin:
      - '*'
    responseHeader:
      - Content-Type
      - x-goog-acl
      - origin
CHAPTER 17. USING THE RED HAT QUAY API

Red Hat Quay provides a full OAuth 2, RESTful API that:

- Is available from endpoints of each Red Hat Quay instance from the URL https://<yourquayhost>/api/v1
- Lets you connect to endpoints, via a browser, to get, delete, post, and put Red Hat Quay settings by enabling the Swagger UI
- Can be accessed by applications that make API calls and use OAuth tokens
- Sends and receives data as JSON

The following text describes how to access the Red Hat Quay API and use it to view and modify setting in your Red Hat Quay cluster. The next section lists and describes API endpoints.

17.1. ACCESSING THE QUAY API FROM QUAY.IO

If you don’t have your own Red Hat Quay cluster running yet, you can explore the Red Hat Quay API available from Quay.io from your web browser:

https://docs.quay.io/api/swagger/

The API Explorer that appears shows Quay.io API endpoints. You will not see superuser API endpoints or endpoints for Red Hat Quay features that are not enabled on Quay.io (such as Repository Mirroring).

From API Explorer, you can get, and sometimes change, information on:

- Billing, subscriptions, and plans
- Repository builds and build triggers
- Error messages and global messages
- Repository images, manifests, permissions, notifications, vulnerabilities, and image signing
- Usage logs
- Organizations, members and OAuth applications
- User and robot accounts
- and more...

Select to open an endpoint to view the Model Schema for each part of the endpoint. Open an endpoint, enter any required parameters (such as a repository name or image), then select the Try it out! button to query or change settings associated with a Quay.io endpoint.

17.2. CREATE OAUTH ACCESS TOKEN

To create an OAuth access token so you can access the API for your organization:

1. Log in to Red Hat Quay and select your Organization (or create a new one).
2. Select the Applications icon from the left navigation.

3. Select Create New Application and give the new application a name when prompted.

4. Select the new application.

5. Select Generate Token from the left navigation.

6. Select the checkboxes to set the scope of the token and select Generate Access Token.

7. Review the permissions you are allowing and select Authorize Application to approve it.

8. Copy the newly generated token to use to access the API.

17.3. ACCESSING YOUR QUAY API FROM A WEB BROWSER

By enabling Swagger, you can access the API for your own Red Hat Quay instance through a web browser. This URL exposes the Red Hat Quay API explorer via the Swagger UI and this URL:

```
```

That way of accessing the API does not include superuser endpoints that are available on Red Hat Quay installations. Here is an example of accessing a Red Hat Quay API interface running on the local system by running the swagger-ui container image:

```
# export SERVER_HOSTNAME=<yourhostname>
docker.io/swaggerapi/swagger-ui
```

With the swagger-ui container running, open your web browser to localhost port 8888 to view API endpoints via the swagger-ui container.

To avoid errors in the log such as "API calls must be invoked with an X-Requested-With header if called from a browser," add the following line to the `config.yaml` on all nodes in the cluster and restart Red Hat Quay:

```
BROWSER_API_CALLS_XHR_ONLY: false
```

17.4. ACCESSING THE RED HAT QUAY API FROM THE COMMAND LINE

You can use the `curl` command to GET, PUT, POST, or DELETE settings via the API for your Red Hat Quay cluster. Replace `<token>` with the OAuth access token you created earlier to get or change settings in the following examples.

17.4.1. Get superuser information

```
$ curl -X GET -H "Authorization: Bearer <token_here>" \
   "https://<yourquayhost>/api/v1/superuser/users/"
```

For example:

```
$ curl -X GET -H "Authorization: Bearer mFCdgS7SAIoMcnTsHCGx23vcNsTgziAa4CmmHIs6g" 
   http://quay-server:8080/api/v1/superuser/users/ | jq
```
17.4.2. Creating a superuser using the API

- Configure a superuser name, as described in the Deploy Quay book:
  - Use the configuration editor UI or
  - Edit the `config.yaml` file directly, with the option of using the configuration API to validate (and download) the updated configuration bundle

- Create the user account for the superuser name:
  - Obtain an authorization token as detailed above, and use `curl` to create the user:

```bash
$ curl -H "Content-Type: application/json" -H "Authorization: Bearer Fava2kV9C92p1eXnMawBZx9vTqVnksvwNm0ckFKZ" -X POST --data '{
  "username": "quaysuper",
  "email": "quaysuper@example.com"
}' http://quay-server:8080/api/v1/superuser/users/ | jq
```

  - The returned content includes a generated password for the new user account:

```json
{
  "username": "quaysuper",
  "email": "quaysuper@example.com",
  "password": "EH67NB3Y6PTBED8H0HC6UVHGGGA3ODSE",
  "encrypted_password": "fn37AZAUQH0PTsU+viO9IS0QxPW9A/boXL4ovZjIFlUPrBz9i4j9U0DqMjuxQ/0HTfy38goKEpG8zYXVeQh3lOFzuOjSvKic2Vq7xdtQsU="
}
```

Now, when you request the list of users, it will show `quaysuper` as a superuser:

```bash
$ curl -X GET -H "Authorization: Bearer mFCDqS7SAIoMcnTsHCGx23vcNsTgziAa4CmmHIsg" http://quay-server:8080/api/v1/superuser/users/ | jq
```
17.4.3. List usage logs

An internal API, `/api/v1/superuser/logs`, is available to list the usage logs for the current system. The results are paginated, so in the following example, more than 20 repos were created to show how to use multiple invocations to access the entire result set.

17.4.3.1. Example for pagination

First invocation

```
$ curl -X GET -k -H "Authorization: Bearer qz9NZ2Np1f55CSZ3RVOvxjeUdkszYuCp0pKggABCD"
https://example-registry-quay-quay-enterprise.apps.example.com/api/v1/superuser/logs | jq
```

Initial output

```
{
    "users": [
    {
        "kind": "user",
        "name": "quayadmin",
        "username": "quayadmin",
        "email": "quayadmin@example.com",
        "verified": true,
        "avatar": {
            "name": "quayadmin",
            "hash": "357a20e8c56e69d6f9734d23ef9517e8",
            "color": "#5254a3",
            "kind": "user"
        },
        "super_user": true,
        "enabled": true
    },
    {
        "kind": "user",
        "name": "quaysuper",
        "username": "quaysuper",
        "email": "quaysuper@example.com",
        "verified": true,
        "avatar": {
            "name": "quaysuper",
            "hash": "c0e0f155afcef68e58a42243b153df08",
            "color": "#969696",
            "kind": "user"
        },
        "super_user": true,
        "enabled": true
    }
    ]
}
```
"logs": [
  {
    "kind": "create_repo",
    "metadata": {
      "repo": "t21",
      "namespace": "namespace1"
    },
    "ip": "10.131.0.13",
    "datetime": "Mon, 13 Dec 2021 11:41:16 -0000",
    "performer": {
      "kind": "user",
      "name": "user1",
      "is_robot": false,
      "avatar": {
        "name": "user1",
        "hash": "5d40b245471708144de9760f2f18113d75aa2488ec82e12435b9de34a6565f73",
        "color": "#ad494a",
        "kind": "user"
      }
    },
    "namespace": {
      "kind": "org",
      "name": "namespace1",
      "avatar": {
        "name": "namespace1",
        "hash": "6cf18b5c19217bfc6df0e7d788746ff7e8201a68cba333fca0437e42379b984f",
        "color": "#e377c2",
        "kind": "org"
      }
    }
  },
  {
    "kind": "create_repo",
    "metadata": {
      "repo": "t20",
      "namespace": "namespace1"
    },
    "ip": "10.131.0.13",
    "datetime": "Mon, 13 Dec 2021 11:41:05 -0000",
    "performer": {
      "kind": "user",
      "name": "user1",
      "is_robot": false,
      "avatar": {
        "name": "user1",
        "hash": "5d40b245471708144de9760f2f18113d75aa2488ec82e12435b9de34a6565f73",
        "color": "#ad494a",
        "kind": "user"
      }
    },
    "namespace": {
      "kind": "org",
      "name": "namespace1",
      "avatar": {
        "name": "namespace1",
        "hash": "6cf18b5c19217bfc6df0e7d788746ff7e8201a68cba333fca0437e42379b984f",
        "color": "#e377c2",
        "kind": "org"
      }
    }
  }
],
"namespace": {
  "kind": "org",
  "name": "namespace1",
  "avatar": {
    "name": "namespace1",
    "hash": "6cf18b5c19217bfc6df0e7d788746ff7e8201a68cba333fca0437e42379b984f",
    "color": "#e377c2",
    "kind": "org"
  }
},
"namespace": {
  "kind": "org",
  "name": "namespace1",
  "avatar": {
    "name": "namespace1",
    "hash": "6cf18b5c19217bfc6df0e7d788746ff7e8201a68cba333fca0437e42379b984f",
    "color": "#e377c2",
    "kind": "org"
  }
}
Second invocation using next_page

```bash
$ curl -X GET -k -H "Authorization: Bearer qz9NZ2Np1f55CSZ3RVOvxjeUdkzYuCp0pKggABCD" https://example-registry-quay-quay-enterprise.apps.example.com/api/v1/superuser/logs?next_page=gAAAAABhtzGDsH38x7pjWhD8MJaq1_2FAggUw2X9S2LoCLNPH65QJqB4XAU2qAxYb6QqtIcWj9eI6DJiMN_q3e30aqCvB2VPQ8rY75WeaiUz3rQfMC4i6EIR78t8oUxVfNp1RMPWQRQYYZYXP9h6E8LZZhqTMs0S-SedaQJ3kVFtxxZqJwHVjgt23Ts2DonVoYwtKgl3bCC5 | jq
```

Output from second invocation

```json
{
}
```
17.4.4. Directory synchronization

To enable directory synchronization for the team `newteam` in organization `testadminorg`, where the corresponding group name in LDAP is `ldapgroup`:

```
$ curl -X POST -H "Authorization: Bearer 9rJYBR3v3pXcj5XqI2XX6Thkwk4gld4TCYLLWDF" \
   -H "Content-type: application/json" \
   -d "{"group_dn": "cn=ldapgroup,ou=Users"}" \n   http://quay1-server:8080/api/v1/organization/testadminorg/team/newteam/syncing
```

To disable synchronization for the same team:

```
$ curl -X DELETE -H "Authorization: Bearer 9rJYBR3v3pXcj5XqI2XX6Thkwk4gld4TCYLLWDF" \n   http://quay1-server:8080/api/v1/organization/testadminorg/team/newteam/syncing
```

17.4.5. Create a repository build via API

In order to build a repository from the specified input and tag the build with custom tags, users can use:

```
"start_time": "Sun, 12 Dec 2021 11:42:46 -0000",
"end_time": "Tue, 14 Dec 2021 11:42:46 -0000",
"logs": [
    {
      "kind": "create_repo",
      "metadata": {
        "repo": "t1",
        "namespace": "namespace1"
      },
      "ip": "10.131.0.13",
      "datetime": "Mon, 13 Dec 2021 11:25:07 -0000",
      "performer": {
        "kind": "user",
        "name": "user1",
        "is_robot": false,
        "avatar": {
          "name": "user1",
          "hash": "5d40b245471708144de9760f2f18113d75aa2488ec82e12435b9de34a6566f73",
          "color": "#ad494a",
          "kind": "user"
        }
      },
      "namespace": {
        "kind": "org",
        "name": "namespace1",
        "avatar": {
          "name": "namespace1",
          "hash": "6cf18b5c19217bfc6df0e7d788746ff7e8201a68cba333fca0437e42379b984f",
          "color": "#e377c2",
          "kind": "org"
        }
      }
    },
    ...
  ]
```
In order to build a repository from the specified input and tag the build with custom tags, users can use `requestRepoBuild` endpoint. It takes the following data:

```
{
  "docker_tags": [
    "string"
  ],
  "pull_robot": "string",
  "subdirectory": "string",
  "archive_url": "string"
}
```

The `archive_url` parameter should point to a tar or zip archive that includes the Dockerfile and other required files for the build. The `file_id` parameter was apart of our older build system. It cannot be used anymore. If Dockerfile is in a sub-directory it needs to be specified as well.

The archive should be publicly accessible. OAuth app should have “Administer Organization” scope because only organization admins have access to the robots’ account tokens. Otherwise, someone could get robot permissions by simply granting a build access to a robot (without having access themselves), and use it to grab the image contents. In case of errors, check the json block returned and ensure the archive location, pull robot, and other parameters are being passed correctly. Click “Download logs” on the top-right of the individual build’s page to check the logs for more verbose messaging.

### 17.4.6. Create an org robot

```
$ curl -X PUT https://quay.io/api/v1/organization/{orgname}/robots/{robot shortname} \
   -H 'Authorization: Bearer <token>''
```

### 17.4.7. Trigger a build

```
$ curl -X POST https://quay.io/api/v1/repository/YOURORGNAME/YOURREPONAME/build/ \
   -H 'Authorization: Bearer <token>''
```

Python with requests

```python
import requests
r = requests.post('https://quay.io/api/v1/repository/example/example/image', headers={'content-type': 'application/json', 'Authorization': 'Bearer <redacted>'), data=[[<request-body-contents>])
print(r.text)
```

### 17.4.8. Create a private repository

```
$ curl -X POST https://quay.io/api/v1/repository \
   -H 'Authorization: Bearer {token}' \
   -H 'Content-Type: application/json' \
   -d '{"namespace":"yournamespace", "repository":"yourreponame", "description":"descriptionofyourrepo", "visibility": "private"} | jq
```

### 17.4.9. Create a mirrored repository

Minimal configuration

```
```
curl -X POST
-H "Authorization: Bearer ${bearer_token}"
-H "Content-Type: application/json"
--data "{"external_reference": "quay.io/minio/mc", "external_registry_username": "", "sync_interval": 600, "sync_start_date": "2021-08-06T11:11:39Z", "root_rule": {"rule_kind": "tag_glob_csv", "rule_value": ["latest"]}, "robot_username": "orga+robot"}"
https://${quay_registry}/api/v1/repository/${orga}/${repo}/mirror | jq

Extended configuration

$ curl -X POST
-H "Authorization: Bearer ${bearer_token}"
-H "Content-Type: application/json"
https://${quay_registry}/api/v1/repository/${orga}/${repo}/mirror | jq