OpenShift Container Platform 4.10

Building applications

Creating and managing applications on OpenShift Container Platform
Creating and managing applications on OpenShift Container Platform
Abstract

This document provides instructions for the various ways to create and manage instances of user-provisioned applications running on OpenShift Container Platform. This includes working with projects and provisioning applications using the Open Service Broker API.
Table of Contents

CHAPTER 1. PROJECTS .......................................................... 8
  1.1. WORKING WITH PROJECTS ........................................... 8
     1.1.1. Creating a project using the web console .................. 8
     1.1.2. Creating a project using the Developer perspective in the web console .................. 8
     1.1.3. Creating a project using the CLI .............................. 10
     1.1.4. Viewing a project using the web console .................. 10
     1.1.5. Viewing a project using the CLI .............................. 10
     1.1.6. Providing access permissions to your project using the Developer perspective .......... 11
     1.1.7. Customizing the available cluster roles using the Developer perspective .......... 12
     1.1.8. Adding to a project ........................................... 13
     1.1.9. Checking project status using the web console .......... 13
     1.1.10. Checking project status using the CLI .................... 13
     1.1.11. Deleting a project using the web console ............... 13
     1.1.12. Deleting a project using the CLI .......................... 14
  1.2. CREATING A PROJECT AS ANOTHER USER ......................... 14
     1.2.1. API impersonation ............................................ 14
     1.2.2. Impersonating a user when you create a project ......... 14
  1.3. CONFIGURING PROJECT CREATION .................................. 15
     1.3.1. About project creation ..................................... 15
     1.3.2. Modifying the template for new projects .................. 15
     1.3.3. Disabling project self-provisioning ....................... 16
     1.3.4. Customizing the project request message ................. 18

CHAPTER 2. CREATING APPLICATIONS ..................................... 20
  2.1. CREATING APPLICATIONS USING THE DEVELOPER PERSPECTIVE ............ 20
     2.1.1. Prerequisites .............................................. 21
     2.1.2. Creating Sample applications ............................. 21
     2.1.3. Importing a codebase from Git to create an application .............................. 22
     2.1.4. Upload JAR files for easy deployment of Java applications .............................. 25
     2.1.5. Using the Devfile registry to access devfiles .................. 26
     2.1.6. Using the Developer Catalog to add services or components to your application .......... 26
     2.1.7. Additional resources ..................................... 27
  2.2. CREATING APPLICATIONS FROM INSTALLED OPERATORS ............. 27
     2.2.1. Creating an etcd cluster using an Operator ............... 28
  2.3. CREATING APPLICATIONS USING THE CLI .......................... 29
     2.3.1. Creating an application from source code ................ 29
        2.3.1.1. Local ................................................ 29
        2.3.1.2. Remote ............................................. 29
        2.3.1.3. Build strategy detection ............................ 30
        2.3.1.4. Language detection ................................ 30
     2.3.2. Creating an application from an image .................... 31
        2.3.2.1. Docker Hub MySQL image ........................... 32
        2.3.2.2. Image in a private registry ....................... 32
        2.3.2.3. Existing image stream and optional image stream tag .............................. 32
     2.3.3. Creating an application from a template .................. 32
        2.3.3.1. Template parameters ................................ 33
     2.3.4. Modifying application creation ............................ 33
        2.3.4.1. Specifying environment variables ..................... 34
        2.3.4.2. Specifying build environment variables ............... 34
        2.3.4.3. Specifying labels ................................... 35
        2.3.4.4. Viewing the output without creation ................. 35
4.4.4. Additional Resources

4.5. GETTING STARTED WITH SERVICE BINDING ON IBM POWER SYSTEMS, IBM Z, AND LINUXONE
Prerequisites

4.5.1. Deploying a PostgreSQL Operator
4.5.2. Creating a PostgreSQL database instance
4.5.3. Deploying the Spring PetClinic sample application
4.5.4. Connecting the Spring PetClinic sample application to the PostgreSQL database service
4.5.5. Additional resources

4.6. EXPOSING BINDING DATA FROM A SERVICE
4.6.1. Methods of exposing binding data
  4.6.1.1. Provisioned service
  4.6.1.2. Direct secret reference
  4.6.1.3. Declaring binding data through CRD or CR annotations
  4.6.1.4. Declaring binding data through OLM descriptors
  4.6.1.5. Detection of binding data through owned resources

4.6.2. Data model
4.6.3. RBAC requirements
4.6.4. Categories of exposable binding data
  4.6.4.1. Exposing a string from a resource
  4.6.4.2. Exposing a constant value as the binding item
  4.6.4.3. Exposing an entire config map or secret that is referenced from a resource
  4.6.4.4. Exposing a specific entry from a config map or secret that is referenced from a resource
  4.6.4.5. Exposing a resource definition value
  4.6.4.6. Exposing entries of a collection with the key and value from each entry
  4.6.4.7. Exposing items of a collection with one key per item
  4.6.4.8. Exposing values of collection entries with one key per entry value

4.6.5. Additional resources

4.7. PROJECTING BINDING DATA
4.7.1. Consumption of binding data
4.7.2. Configuration of the directory path to project the binding data inside workload container
  4.7.2.1. Computation of the final path for projecting the binding data as files
4.7.3. Projecting the binding data
4.7.4. Additional resources

4.8. BINDING WORKLOADS USING SERVICE BINDING OPERATOR
4.8.1. Naming strategies
4.8.2. Advanced binding options
  4.8.2.1. Changing the binding names before projecting them into the workload
  4.8.2.2. Composing custom binding data
  4.8.2.3. Binding workloads using a label selector
4.8.3. Binding secondary workloads that are not compliant with PodSpec
  4.8.3.1. Configuring the custom location of the container path
  4.8.3.2. Configuring the custom location of the secret path
  4.8.3.3. Workload resource mapping
4.8.4. Unbinding workloads from a backing service
4.8.5. Additional resources

4.9. CONNECTING AN APPLICATION TO A SERVICE USING THE DEVELOPER PERSPECTIVE
4.9.1. Creating a visual connection between components
4.9.2. Creating a binding connection between components
4.9.3. Additional resources

CHAPTER 5. WORKING WITH HELM CHARTS
5.1. UNDERSTANDING HELM
  5.1.1. Key features
5.1.2. Red Hat Certification of Helm charts for OpenShift

5.1.3. Additional resources

5.2. INSTALLING HELM

5.2.1. On Linux

5.2.2. On Windows 7/8

5.2.3. On Windows 10

5.2.4. On MacOS

5.3. CONFIGURING CUSTOM HELM CHART REPOSITORIES

5.3.1. Installing a Helm chart on an OpenShift Container Platform cluster

5.3.2. Installing Helm charts using the Developer perspective

5.3.3. Using Helm in the web terminal

5.3.4. Creating a custom Helm chart on OpenShift Container Platform

5.3.5. Adding custom Helm chart repositories

5.3.6. Adding namespace-scoped custom Helm chart repositories

5.3.7. Creating credentials and CA certificates to add Helm chart repositories

5.3.8. Filtering Helm Charts by their certification level

5.3.9. Disabling Helm Chart repositories

5.4. WORKING WITH HELM RELEASES

5.4.1. Prerequisites

5.4.2. Upgrading a Helm release

5.4.3. Rolling back a Helm release

5.4.4. Uninstalling a Helm release

CHAPTER 6. DEPLOYMENTS

6.1. UNDERSTANDING DEPLOYMENT AND DEPLOYMENTCONFIG OBJECTS

6.1.1. Building blocks of a deployment

6.1.1.1. Replication controllers

6.1.1.2. Replica sets

6.1.2. DeploymentConfig objects

6.1.3. Deployments

6.1.4. Comparing Deployment and DeploymentConfig objects

6.1.4.1. Design

6.1.4.2. DeploymentConfig object-specific features

Automatic rollbacks

Triggers

Lifecycle hooks

Custom strategies

6.1.4.3. Deployment-specific features

Rollover

Proportional scaling

Pausing mid-rollout

6.2. MANAGING DEPLOYMENT PROCESSES

6.2.1. Managing DeploymentConfig objects

6.2.1.1. Starting a deployment

6.2.1.2. Viewing a deployment

6.2.1.3. Retrying a deployment

6.2.1.4. Rolling back a deployment

6.2.1.5. Executing commands inside a container

6.2.1.6. Viewing deployment logs

6.2.1.7. Deployment triggers

Config change deployment triggers

Image change deployment triggers

6.2.1.7.1. Setting deployment triggers
6.2.1.8. Setting deployment resources
6.2.1.9. Scaling manually
6.2.1.10. Accessing private repositories from DeploymentConfig objects
6.2.1.11. Assigning pods to specific nodes
6.2.1.12. Running a pod with a different service account

6.3. USING DEPLOYMENT STRATEGIES
6.3.1. Rolling strategy
   6.3.1.1. Canary deployments
   6.3.1.2. Creating a rolling deployment
   6.3.1.3. Starting a rolling deployment using the Developer perspective
6.3.2. Recreate strategy
6.3.3. Starting a recreate deployment using the Developer perspective
6.3.4. Custom strategy
6.3.5. Lifecycle hooks
   6.3.5.1. Setting lifecycle hooks

6.4. USING ROUTE-BASED DEPLOYMENT STRATEGIES
6.4.1. Proxy shards and traffic splitting
6.4.2. N-1 compatibility
6.4.3. Graceful termination
6.4.4. Blue-green deployments
   6.4.4.1. Setting up a blue-green deployment
6.4.5. A/B deployments
   6.4.5.1. Load balancing for A/B testing
      6.4.5.1.1. Managing weights of an existing route using the web console
      6.4.5.1.2. Managing weights of a new route using the web console
   6.4.5.1.3. Managing weights using the CLI
   6.4.5.1.4. One service, multiple Deployment objects

CHAPTER 7. QUOTAS
7.1. RESOURCE QUOTAS PER PROJECT
   7.1.1. Resources managed by quotas
   7.1.2. Quota scopes
   7.1.3. Quota enforcement
   7.1.4. Requests versus limits
   7.1.5. Sample resource quota definitions
   7.1.6. Creating a quota
      7.1.6.1. Creating object count quotas
      7.1.6.2. Setting resource quota for extended resources
   7.1.7. Viewing a quota
   7.1.8. Configuring explicit resource quotas
7.2. RESOURCE QUOTAS ACROSS MULTIPLE PROJECTS
   7.2.1. Selecting multiple projects during quota creation
   7.2.2. Viewing applicable cluster resource quotas
   7.2.3. Selection granularity

CHAPTER 8. USING CONFIG MAPS WITH APPLICATIONS
8.1. UNDERSTANDING CONFIG MAPS
   Config map restrictions
8.2. USE CASES: CONSUMING CONFIG MAPS IN PODS
   8.2.1. Populating environment variables in containers by using config maps
   8.2.2. Setting command-line arguments for container commands with config maps
   8.2.3. Injecting content into a volume by using config maps
CHAPTER 1. PROJECTS

1.1. WORKING WITH PROJECTS

A project allows a community of users to organize and manage their content in isolation from other communities.

NOTE

Projects starting with openshift- and kube- are default projects. These projects host cluster components that run as pods and other infrastructure components. As such, OpenShift Container Platform does not allow you to create projects starting with openshift- or kube- using the oc new-project command. Cluster administrators can create these projects using the oc adm new-project command.

NOTE

You cannot assign an SCC to pods created in one of the default namespaces: default, kube-system, kube-public, openshift-node, openshift-infra, and openshift. You cannot use these namespaces for running pods or services.

1.1.1. Creating a project using the web console

If allowed by your cluster administrator, you can create a new project.

NOTE

Projects starting with openshift- and kube- are considered critical by OpenShift Container Platform. As such, OpenShift Container Platform does not allow you to create Projects starting with openshift- using the web console.

NOTE

You cannot assign an SCC to pods created in one of the default namespaces: default, kube-system, kube-public, openshift-node, openshift-infra, and openshift. You cannot use these namespaces for running pods or services.

Procedure

1. Navigate to Home → Projects.
2. Click Create Project.
3. Enter your project details.
4. Click Create.

1.1.2. Creating a project using the Developer perspective in the web console

You can use the Developer perspective in the OpenShift Container Platform web console to create a project in your cluster.
NOTE
Projects starting with `openshift-` and `kube-` are considered critical by OpenShift Container Platform. As such, OpenShift Container Platform does not allow you to create projects starting with `openshift-` or `kube-` using the Developer perspective. Cluster administrators can create these projects using the `oc adm new-project` command.

NOTE
You cannot assign an SCC to pods created in one of the default namespaces: `default`, `kube-system`, `kube-public`, `openshift-node`, `openshift-infra`, and `openshift`. You cannot use these namespaces for running pods or services.

Prerequisites

- Ensure that you have the appropriate roles and permissions to create projects, applications, and other workloads in OpenShift Container Platform.

Procedure

You can create a project using the Developer perspective, as follows:

1. Click the Project drop-down menu to see a list of all available projects. Select Create Project.

   ![Create Project](image)

2. In the Create Project dialog box, enter a unique name, such as `myproject`, in the Name field.

3. Optional: Add the Display Name and Description details for the project.

4. Click Create.

5. Use the left navigation panel to navigate to the Project view and see the dashboard for your project.

6. Optional:
   - Use the Project drop-down menu at the top of the screen and select all projects to list all of the projects in your cluster.
   - Use the Details tab to see the project details.
If you have adequate permissions for a project, you can use the Project Access tab to provide or revoke administrator, edit, and view privileges for the project.

1.1.3. Creating a project using the CLI

If allowed by your cluster administrator, you can create a new project.

**NOTE**

Projects starting with `openshift-` and `kube-` are considered critical by OpenShift Container Platform. As such, OpenShift Container Platform does not allow you to create Projects starting with `openshift-` or `kube-` using the `oc new-project` command. Cluster administrators can create these Projects using the `oc adm new-project` command.

**NOTE**

You cannot assign an SCC to pods created in one of the default namespaces: default, kube-system, kube-public, openshift-node, openshift-infra, and openshift. You cannot use these namespaces for running pods or services.

**Procedure**

- Run:

  ```bash
  $ oc new-project <project_name> \
  --description="<description>" --display-name="<display_name>"
  ```

  For example:

  ```bash
  $ oc new-project hello-openshift \
  --description="This is an example project" \
  --display-name="Hello OpenShift"
  ```

  **NOTE**

  The number of projects you are allowed to create might be limited by the system administrator. After your limit is reached, you might have to delete an existing project in order to create a new one.

1.1.4. Viewing a project using the web console

**Procedure**

1. Navigate to Home → Projects.

2. Select a project to view.
   
   On this page, click Workloads to see workloads in the project.

1.1.5. Viewing a project using the CLI

When viewing projects, you are restricted to seeing only the projects you have access to view based on the authorization policy.
Procedure

1. To view a list of projects, run:

   $ oc get projects

2. You can change from the current project to a different project for CLI operations. The specified project is then used in all subsequent operations that manipulate project-scoped content:

   $ oc project <project_name>

1.1.6. Providing access permissions to your project using the Developer perspective

You can use the Project view in the Developer perspective to grant or revoke access permissions to your project.

Procedure

To add users to your project and provide Admin, Edit, or View access to them:

1. In the Developer perspective, navigate to the Project view.

2. In the Project page, select the Project Access tab.

3. Click Add Access to add a new row of permissions to the default ones.

   Figure 1.2. Project permissions

4. Enter the user name, click the Select a role drop-down list, and select an appropriate role.

5. Click Save to add the new permissions.

You can also use:

$ oc get projects
$ oc project <project_name>
• The **Select a role** drop-down list, to modify the access permissions of an existing user.

• The **Remove Access** icon, to completely remove the access permissions of an existing user to the project.

**NOTE**

Advanced role-based access control is managed in the **Roles** and **Roles Binding** views in the **Administrator** perspective.

1.1.7. Customizing the available cluster roles using the Developer perspective

The users of a project are assigned to a cluster role based on their access control. You can access these cluster roles by navigating to the **Project → Project access → Role**. By default, these roles are **Admin**, **Edit**, and **View**.

To add or edit the cluster roles for a project, you can customize the YAML code of the cluster.

**Procedure**

To customize the different cluster roles of a project:

1. In the **Search** view, use the **Resources** drop-down list to search for **Console**.

2. From the available options, select the **Console operator.openshift.io/v1**.

**Figure 1.3. Searching Console resource**

```markdown
<table>
<thead>
<tr>
<th>Resources</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td>app=frontend</td>
</tr>
</tbody>
</table>
```

| □ | Console config.openshift.io/v1 |
| □ | Console operator.openshift.io/v1 |
| □ | ConsoleCLIDownload |
| □ | ConsoleExternalLogLink |
| □ | ConsoleLink |
| □ | ConsoleNotification |
| □ | ConsolePlugin |
| □ | ConsoleQuickStart |
| □ | ConsoleYAMLSample |
3. Select **cluster** under the **Name** list.

4. Navigate to the **YAML** tab to view and edit the YAML code.

5. In the YAML code under **spec**, add or edit the list of **availableClusterRoles** and save your changes:

```yaml
spec:
  customization:
    projectAccess:
      availableClusterRoles:
      - admin
      - edit
      - view
```

### 1.1.8. Adding to a project

**Procedure**

1. Select **Developer** from the context selector at the top of the web console navigation menu.

2. Click **+Add**

3. At the top of the page, select the name of the project that you want to add to.

4. Click a method for adding to your project, and then follow the workflow.

**NOTE**

You can also add components to the topology using quick search.

### 1.1.9. Checking project status using the web console

**Procedure**

1. Navigate to **Home → Projects**.

2. Select a project to see its status.

### 1.1.10. Checking project status using the CLI

**Procedure**

1. Run:

   ```bash
   $ oc status
   ```

   This command provides a high-level overview of the current project, with its components and their relationships.

### 1.1.11. Deleting a project using the web console

You can delete a project by using the OpenShift Container Platform web console.
NOTE

If you do not have permissions to delete the project, the Delete Project option is not available.

Procedure

1. Navigate to Home → Projects.

2. Locate the project that you want to delete from the list of projects.

3. On the far right side of the project listing, select Delete Project from the Options menu.

4. When the Delete Project pane opens, enter the name of the project that you want to delete in the field.

5. Click Delete.

1.1.12. Deleting a project using the CLI

When you delete a project, the server updates the project status to Terminating from Active. Then, the server clears all content from a project that is in the Terminating state before finally removing the project. While a project is in Terminating status, you cannot add new content to the project. Projects can be deleted from the CLI or the web console.

Procedure

1. Run:

   $ oc delete project <project_name>

1.2. CREATING A PROJECT AS ANOTHER USER

Impersonation allows you to create a project as a different user.

1.2.1. API impersonation

You can configure a request to the OpenShift Container Platform API to act as though it originated from another user. For more information, see User impersonation in the Kubernetes documentation.

1.2.2. Impersonating a different user when you create a project

You can impersonate a different user when you create a project request. Because system:authenticated:oauth is the only bootstrap group that can create project requests, you must impersonate that group.

Procedure

- To create a project request on behalf of a different user:

  $ oc new-project <project> --as=<user> \
  --as-group=system:authenticated --as-group=system:authenticated:oauth
1.3. CONFIGURING PROJECT CREATION

In OpenShift Container Platform, projects are used to group and isolate related objects. When a request is made to create a new project using the web console or `oc new-project` command, an endpoint in OpenShift Container Platform is used to provision the project according to a template, which can be customized.

As a cluster administrator, you can allow and configure how developers and service accounts can create, or self-provision, their own projects.

1.3.1. About project creation

The OpenShift Container Platform API server automatically provisions new projects based on the project template that is identified by the `projectRequestTemplate` parameter in the cluster’s project configuration resource. If the parameter is not defined, the API server creates a default template that creates a project with the requested name, and assigns the requesting user to the `admin` role for that project.

When a project request is submitted, the API substitutes the following parameters into the template:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROJECT_NAME</td>
<td>The name of the project. Required.</td>
</tr>
<tr>
<td>PROJECT_DISPLAYNAME</td>
<td>The display name of the project. May be empty.</td>
</tr>
<tr>
<td>PROJECT_DESCRIPTION</td>
<td>The description of the project. May be empty.</td>
</tr>
<tr>
<td>PROJECT_ADMIN_USER</td>
<td>The user name of the administrating user.</td>
</tr>
<tr>
<td>PROJECT_REQUESTING_USER</td>
<td>The user name of the requesting user.</td>
</tr>
</tbody>
</table>

Access to the API is granted to developers with the self-provisioner role and the self-provisioners cluster role binding. This role is available to all authenticated developers by default.

1.3.2. Modifying the template for new projects

As a cluster administrator, you can modify the default project template so that new projects are created using your custom requirements.

To create your own custom project template:

**Procedure**

1. Log in as a user with `cluster-admin` privileges.
2. Generate the default project template:

```bash
$ oc adm create-bootstrap-project-template -o yaml > template.yaml
```
3. Use a text editor to modify the generated `template.yaml` file by adding objects or modifying existing objects.

4. The project template must be created in the `openshift-config` namespace. Load your modified template:

   ```bash
   $ oc create -f template.yaml -n openshift-config
   ```

5. Edit the project configuration resource using the web console or CLI.
   - Using the web console:
     i. Navigate to the **Administration → Cluster Settings** page.
     ii. Click **Configuration** to view all configuration resources.
     iii. Find the entry for **Project** and click **Edit YAML**.
   - Using the CLI:
     i. Edit the `project.config.openshift.io/cluster` resource:

   ```bash
   $ oc edit project.config.openshift.io/cluster
   ```

6. Update the `spec` section to include the **projectRequestTemplate** and **name** parameters, and set the name of your uploaded project template. The default name is **project-request**.

   **Project configuration resource with custom project template**

   ```yaml
   apiVersion: config.openshift.io/v1
   kind: Project
   metadata:
     ...
   spec:
     projectRequestTemplate:
       name: <template_name>
   ```

7. After you save your changes, create a new project to verify that your changes were successfully applied.

### 1.3.3. Disabling project self-provisioning

You can prevent an authenticated user group from self-provisioning new projects.

**Procedure**

1. Log in as a user with **cluster-admin** privileges.

2. View the **self-provisioners** cluster role binding usage by running the following command:

   ```bash
   $ oc describe clusterrolebinding.rbac self-provisioners
   ```

   **Example output**
Review the subjects in the **self-provisioners** section.

3. Remove the **self-provisioner** cluster role from the group **system:authenticated:oauth**.
   
   - If the **self-provisioners** cluster role binding binds only the **self-provisioner** role to the **system:authenticated:oauth** group, run the following command:
     
     ```
     $ oc patch clusterrolebinding.rbac self-provisioners -p '{"subjects": null}'
     ```
     
   - If the **self-provisioners** cluster role binding binds the **self-provisioner** role to more users, groups, or service accounts than the **system:authenticated:oauth** group, run the following command:
     
     ```
     $ oc adm policy \
        remove-cluster-role-from-group self-provisioner \
        system:authenticated:oauth
     ```

4. Edit the **self-provisioners** cluster role binding to prevent automatic updates to the role.
   Automatic updates reset the cluster roles to the default state.
   
   - To update the role binding using the CLI:
     
     i. Run the following command:
        
        ```
        $ oc edit clusterrolebinding.rbac self-provisioners
        ```
      
     ii. In the displayed role binding, set the `rbac.authorization.kubernetes.io/autoupdate` parameter value to **false**, as shown in the following example:
        
        ```yaml
        apiVersion: authorization.openshift.io/v1
        kind: ClusterRoleBinding
        metadata:
         annotations:
          rbac.authorization.kubernetes.io/autoupdate: "false"
         ...
        ```
     
   - To update the role binding by using a single command:
     
     ```
     $ oc patch clusterrolebinding.rbac self-provisioners -p '{ "metadata": { "annotations": { "rbac.authorization.kubernetes.io/autoupdate": "false" } } }'
     ```

5. Log in as an authenticated user and verify that it can no longer self-provision a project:
Consider customizing this project request message to provide more helpful instructions specific to your organization.

1.3.4. Customizing the project request message

When a developer or a service account that is unable to self-provision projects makes a project creation request using the web console or CLI, the following error message is returned by default:

You may not request a new project via this API.

Cluster administrators can customize this message. Consider updating it to provide further instructions on how to request a new project specific to your organization. For example:

- To request a project, contact your system administrator at projectname@example.com.
- To request a new project, fill out the project request form located at https://internal.example.com/openshift-project-request.

To customize the project request message:

Procedure

1. Edit the project configuration resource using the web console or CLI.
   - Using the web console:
     i. Navigate to the Administration → Cluster Settings page.
     ii. Click Configuration to view all configuration resources.
     iii. Find the entry for Project and click Edit YAML.
   - Using the CLI:
     i. Log in as a user with cluster-admin privileges.
     ii. Edit the project.config.openshift.io/cluster resource:

   $ oc edit project.config.openshift.io/cluster

2. Update the spec section to include the projectRequestMessage parameter and set the value to your custom message:

Project configuration resource with custom project request message

```yaml
apiVersion: config.openshift.io/v1
kind: Project
metadata:
```
For example:

```yaml
apiVersion: config.openshift.io/v1
kind: Project
metadata:
  ...
  spec:
    projectRequestMessage: To request a project, contact your system administrator at projectname@example.com.
```

3. After you save your changes, attempt to create a new project as a developer or service account that is unable to self-provision projects to verify that your changes were successfully applied.
CHAPTER 2. CREATING APPLICATIONS

2.1. CREATING APPLICATIONS USING THE DEVELOPER PERSPECTIVE

The Developer perspective in the web console provides you the following options from the +Add view to create applications and associated services and deploy them on OpenShift Container Platform:

- **Getting started resources**: Use these resources to help you get started with Developer Console. You can choose to hide the header using the Options menu.

  - **Creating applications using samples**: Use existing code samples to get started with creating applications on the OpenShift Container Platform.

  - **Build with guided documentation**: Follow the guided documentation to build applications and familiarize yourself with key concepts and terminologies.

  - **Explore new developer features**: Explore the new features and resources within the Developer perspective.

- **Developer catalog**: Explore the Developer Catalog to select the required applications, services, or source to image builders, and then add it to your project.

  - **All Services**: Browse the catalog to discover services across OpenShift Container Platform.

  - **Database**: Select the required database service and add it to your application.

  - **Operator Backed**: Select and deploy the required Operator-managed service.

  - **Helm chart**: Select the required Helm chart to simplify deployment of applications and services.

  - **Devfile**: Select a devfile from the Devfile registry to declaratively define a development environment.

  - **Event Source**: Select an event source to register interest in a class of events from a particular system.

- **NOTE**: The Managed services option is also available if the RHOAS Operator is installed.

- **Git repository**: Import an existing codebase, Devfile, or Dockerfile from your Git repository using the From Git, From Devfile, or From Dockerfile options respectively, to build and deploy an application on OpenShift Container Platform.

- **Container images**: Use existing images from an image stream or registry to deploy it on to the OpenShift Container Platform.

- **Pipelines**: Use Tekton pipeline to create CI/CD pipelines for your software delivery process on the OpenShift Container Platform.

- **Serverless**: Explore the Serverless options to create, build, and deploy stateless and serverless applications on the OpenShift Container Platform.
- **Channel**: Create a Knative channel to create an event forwarding and persistence layer with in-memory and reliable implementations.

- **Samples**: Explore the available sample applications to create, build, and deploy an application quickly.

- **From Local Machine**: Explore the From Local Machine tile to import or upload files on your local machine for building and deploying applications easily.

- **Import YAML**: Upload a YAML file to create and define resources for building and deploying applications.

- **Upload JAR file**: Upload a JAR file to build and deploy Java applications.

Note that certain options, such as **Pipelines**, **Event Source**, and **Import Virtual Machines**, are displayed only when the OpenShift Pipelines Operator, OpenShift Serverless Operator, and OpenShift Virtualization Operator are installed, respectively.

### 2.1.1. Prerequisites

To create applications using the Developer perspective ensure that:

- You have logged in to the web console.

- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.

To create serverless applications, in addition to the preceding prerequisites, ensure that:

- You have installed the OpenShift Serverless Operator.

- You have created a KnativeServing resource in the knative-serving namespace.

### 2.1.2. Creating Sample applications

You can use the basic sample applications in the +Add flow of the Developer perspective to create, build, and deploy applications quickly.

The following procedure explains the Samples option in the Developer perspective to create a sample application.

**Procedure**

1. In the +Add view, click on the Samples tile to see the Samples page.

2. On the Samples page, select one of the available sample applications to see the Create Sample Application form.

3. In the Create Sample Application Form
   
   - In the Name field, the deployment name is displayed by default. You can modify this name as required.

   - In the Builder Image Version, a builder image is selected by default. You can modify this image version by using the Builder Image Version drop-down list.

   - A sample Git repository URL is added by default.
2.1.3. Importing a codebase from Git to create an application

You can use the Developer perspective to create, build, and deploy an application on OpenShift Container Platform using an existing codebase in GitHub.

The following procedure walks you through the From Git option in the Developer perspective to create an application.

Procedure

1. In the +Add view, click From Git in the Git Repository tile to see the Import from git form.

2. In the Git section, enter the Git repository URL for the codebase you want to use to create an application. For example, enter the URL of this sample Node.js application https://github.com/sclorg/nodejs-ex. The URL is then validated.

3. Optional: You can click Show Advanced Git Options to add details such as:
   - **Git Reference** to point to code in a specific branch, tag, or commit to be used to build the application.
   - **Context Dir** to specify the subdirectory for the application source code you want to use to build the application.
   - **Source Secret** to create a Secret Name with credentials for pulling your source code from a private repository.

4. Optional: You can import a devfile, a Dockerfile, or a builder image through your Git repository to further customize your deployment.
   - If your Git repository contains a devfile, a Dockerfile, or a builder image, it is automatically detected and populated on the respective path fields. If a devfile, a Dockerfile, and a builder image are detected in the same repository, the devfile is selected by default.
   - To edit the file import type and select a different strategy, click Edit import strategy option.
   - If multiple devfiles, Dockerfiles, or builder images are detected, to import a specific devfile, Dockerfile, or a builder image, specify the respective paths relative to the context directory.

5. After the Git URL is validated, the recommended builder image is selected and marked with a star. If the builder image is not auto-detected, select a builder image. For the https://github.com/sclorg/nodejs-ex Git URL, by default the Node.js builder image is selected.
   - Optional: Use the Builder Image Version drop-down to specify a version.
   - Optional: Use the Edit import strategy to select a different strategy.
   - Optional: For the Node.js builder image, use the Run command field to override the command to run the application.

6. In the General section:
a. In the **Application** field, enter a unique name for the application grouping, for example, `myapp`. Ensure that the application name is unique in a namespace.

b. The **Name** field to identify the resources created for this application is automatically populated based on the Git repository URL if there are no existing applications. If there are existing applications, you can choose to deploy the component within an existing application, create a new application, or keep the component unassigned.

**NOTE**

The resource name must be unique in a namespace. Modify the resource name if you get an error.

7. In the **Resources** section, select:
   - **Deployment**, to create an application in plain Kubernetes style.
   - **Deployment Config**, to create an OpenShift Container Platform style application.
   - **Serverless Deployment**, to create a Knative service.

**NOTE**

The **Serverless Deployment** option is displayed in the **Import from git** form only if the OpenShift Serverless Operator is installed in your cluster. For further details, refer to the OpenShift Serverless documentation.

8. In the **Pipelines** section, select **Add Pipeline**, and then click **Show Pipeline Visualization** to see the pipeline for the application. A default pipeline is selected, but you can choose the pipeline you want from the list of available pipelines for the application.

9. Optional: In the **Advanced Options** section, the **Target port** and the **Create a route to the application** is selected by default so that you can access your application using a publicly available URL.
   - If your application does not expose its data on the default public port, 80, clear the check box, and set the target port number you want to expose.

10. Optional: You can use the following advanced options to further customize your application:

### Routing

By clicking the **Routing** link, you can perform the following actions:

- Customize the hostname for the route.
- Specify the path the router watches.
- Select the target port for the traffic from the drop-down list.
- Secure your route by selecting the **Secure Route** check box. Select the required TLS termination type and set a policy for insecure traffic from the respective drop-down lists.
NOTE

For serverless applications, the Knative service manages all the routing options above. However, you can customize the target port for traffic, if required. If the target port is not specified, the default port of 8080 is used.

Domain mapping

If you are creating a Serverless Deployment, you can add a custom domain mapping to the Knative service during creation.

- In the Advanced options section, click Show advanced Routing options.
  - If the domain mapping CR that you want to map to the service already exists, you can select it from the Domain mapping drop-down menu.
  - If you want to create a new domain mapping CR, type the domain name into the box, and select the Create option. For example, if you type in example.com, the Create option is Create "example.com".

Health Checks

Click the Health Checks link to add Readiness, Liveness, and Startup probes to your application. All the probes have prepopulated default data; you can add the probes with the default data or customize it as required.

To customize the health probes:

- Click Add Readiness Probe if required, modify the parameters to check if the container is ready to handle requests, and select the check mark to add the probe.
- Click Add Liveness Probe if required, modify the parameters to check if a container is still running, and select the check mark to add the probe.
- Click Add Startup Probe, if required, modify the parameters to check if the application within the container has started, and select the check mark to add the probe.

For each of the probes, you can specify the request type - HTTP GET, Container Command, or TCP Socket, from the drop-down list. The form changes as per the selected request type. You can then modify the default values for the other parameters, such as the success and failure thresholds for the probe, number of seconds before performing the first probe after the container starts, frequency of the probe, and the timeout value.

Build Configuration and Deployment

Click the Build Configuration and Deployment links to see the respective configuration options. Some options are selected by default; you can customize them further by adding the necessary triggers and environment variables.

For serverless applications, the Deployment option is not displayed as the Knative configuration resource maintains the desired state for your deployment instead of a DeploymentConfig resource.

Scaling

Click the Scaling link to define the number of pods or instances of the application you want to deploy initially.

If you are creating a serverless deployment, you can also configure the following settings:

- Min Pods determines the lower limit for the number of pods that must be running at any given time for a Knative service. This is also known as the minScale setting.
Max Pods determines the upper limit for the number of pods that can be running at any given time for a Knative service. This is also known as the maxScale setting.

Concurrency target determines the number of concurrent requests desired for each instance of the application at a given time.

Concurrency limit determines the limit for the number of concurrent requests allowed for each instance of the application at a given time.

Concurrency utilization determines the percentage of the concurrent requests limit that must be met before Knative scales up additional pods to handle additional traffic.

Autoscale window defines the time window over which metrics are averaged to provide input for scaling decisions when the autoscaler is not in panic mode. A service is scaled-to-zero if no requests are received during this window. The default duration for the autoscale window is 60s. This is also known as the stable window.

Resource Limit
Click the Resource Limit link to set the amount of CPU and Memory resources a container is guaranteed or allowed to use when running.

Labels
Click the Labels link to add custom labels to your application.

1. Click Create to create the application and a success notification is displayed. You can see the build status of the application in the Topology view.

2.1.4. Upload JAR files for easy deployment of Java applications
You can use the JAR files in the Topology view of the Developer perspective to deploy your Java applications. You can upload a JAR file using the following options:

- Navigate to the +Add view of the Developer perspective, and click Upload JAR file in the From Local Machine tile. Browse and select your JAR file, or drag and drop a JAR file to deploy your application.

- Navigate to the Topology view and use the Upload JAR file option, or drag and drop a JAR file to deploy your application.

- Use the in-context menu in the Topology view, and then use the Upload JAR file option to upload your JAR file to deploy your application.

Use the following instructions to upload a JAR file in the Topology view to deploy a Java application:

Procedure
1. In the Topology view, right-click anywhere in the Topology view to see the Add to Project menu.

2. Hover over the Add to Project menu to see the menu options, and then select the Upload JAR file option to see the Upload JAR file form. Alternatively, you can drag and drop the JAR file in the Topology view.

3. In the JAR file field, browse for the required JAR file on your local machine and upload it. Alternatively, you can drag and drop the JAR file on the field. A toast alert is displayed at the
top right if an incompatible file type is dragged and dropped on the Topology view. A field error
is displayed if an incompatible file type is dropped on the field in the upload form.

4. You can further specify optional Java commands to customize your deployed application. The
Runtime Icon and Builder Image is selected by default. If a Builder Image is not auto-detected,
select a Builder Image. If required, you can change the version using the Builder Image Version
drop-down list.

5. In the optional Application Name field, enter a unique name for your application for the
resource labelling.

6. In the Name field, enter a unique component name to name the associated resources.

7. In the Resources field, choose the resource type for your application.

8. In the Advanced options, click on Create a Route to the Application to configure a public URL
for your deployed application.

9. Click Create to deploy the application. The user sees a toast notification notifying that the JAR
file is being uploaded and takes a while. The toast notification also includes a link to view the
build logs.

NOTE
If the user attempts to close the browser tab while the build is running, a web
alert would be displayed asking the user if they actually want to leave the page.

After the JAR file is uploaded and the application is deployed, you can see the deployment in the
Topology view.

2.1.5. Using the Devfile registry to access devfiles

You can use the devfiles in the +Add flow of the Developer perspective to create an application. The
+Add flow provides a complete integration with the devfile community registry. A devfile is a portable
YAML file that describes your development environment without needing to configure it from scratch.
Using the Devfile registry, you can use a pre-configured devfile to create an application.

Procedure

1. Navigate to Developer Perspective → +Add → Developer Catalog → All Services. A list of all
the available services in the Developer Catalog is displayed.

2. Under All Services, select Devfiles to browse for devfiles that support a particular language or
framework. Alternatively, you can use the keyword filter to search for a particular devfile using
their name, tag, or description.

3. Click the devfile you want to use to create an application. The devfile tile displays the details of
the devfile, including the name, description, provider, and the documentation of the devfile.

4. Click Create to create an application and view the application in the Topology view.

2.1.6. Using the Developer Catalog to add services or components to your application

You use the Developer Catalog to deploy applications and services based on Operator backed services
such as Databases, Builder Images, and Helm Charts. The Developer Catalog contains a collection of application components, services, event sources, or source-to-image builders that you can add to your project. Cluster administrators can customize the content made available in the catalog.

### Procedure

1. In the Developer perspective, navigate to the +Add view and from the Developer Catalog tile, click All Services to view all the available services in the Developer Catalog.

2. Under All Services, select the kind of service or the component you need to add to your project. For this example, select Databases to list all the database services and then click MariaDB to see the details for the service.

3. Click Instantiate Template to see an automatically populated template with details for the MariaDB service, and then click Create to create and view the MariaDB service in the Topology view.

![Figure 2.1. MariaDB in Topology](image)

### 2.1.7. Additional resources

- For more information about Knative routing settings for OpenShift Serverless, see Routing.

- For more information about domain mapping settings for OpenShift Serverless, see Configuring a custom domain for a Knative service.

- For more information about Knative autoscaling settings for OpenShift Serverless, see Autoscaling.

### 2.2. CREATING APPLICATIONS FROM INSTALLED OPERATORS
Operators are a method of packaging, deploying, and managing a Kubernetes application. You can create applications on OpenShift Container Platform using Operators that have been installed by a cluster administrator.

This guide walks developers through an example of creating applications from an installed Operator using the OpenShift Container Platform web console.

**Additional resources**

- See the [Operators](#) guide for more on how Operators work and how the Operator Lifecycle Manager is integrated in OpenShift Container Platform.

### 2.2.1. Creating an etcd cluster using an Operator

This procedure walks through creating a new etcd cluster using the etcd Operator, managed by Operator Lifecycle Manager (OLM).

**Prerequisites**

- Access to an OpenShift Container Platform 4.10 cluster.
- The etcd Operator already installed cluster-wide by an administrator.

**Procedure**

1. Create a new project in the OpenShift Container Platform web console for this procedure. This example uses a project called `my-etcd`.

2. Navigate to the Operators → Installed Operators page. The Operators that have been installed to the cluster by the cluster administrator and are available for use are shown here as a list of cluster service versions (CSVs). CSVs are used to launch and manage the software provided by the Operator.

   **TIP**

   You can get this list from the CLI using:

   ```
   $ oc get csv
   ```

3. On the Installed Operators page, click the etcd Operator to view more details and available actions.
   
   As shown under [Provided APIs](#), this Operator makes available three new resource types, including one for an etcd Cluster (the `EtcdCluster` resource). These objects work similar to the built-in native Kubernetes ones, such as `Deployment` or `ReplicaSet`, but contain logic specific to managing etcd.

4. Create a new etcd cluster:

   a. In the etcd Cluster API box, click Create instance.

   b. The next screen allows you to make any modifications to the minimal starting template of an EtcdCluster object, such as the size of the cluster. For now, click Create to finalize. This triggers the Operator to start up the pods, services, and other components of the new etcd cluster.
5. Click on the example etcd cluster, then click the Resources tab to see that your project now contains a number of resources created and configured automatically by the Operator. Verify that a Kubernetes service has been created that allows you to access the database from other pods in your project.

6. All users with the edit role in a given project can create, manage, and delete application instances (an etcd cluster, in this example) managed by Operators that have already been created in the project, in a self-service manner, just like a cloud service. If you want to enable additional users with this ability, project administrators can add the role using the following command:

```bash
$ oc policy add-role-to-user edit <user> -n <target_project>
```

You now have an etcd cluster that will react to failures and rebalance data as pods become unhealthy or are migrated between nodes in the cluster. Most importantly, cluster administrators or developers with proper access can now easily use the database with their applications.

2.3. CREATING APPLICATIONS USING THE CLI

You can create an OpenShift Container Platform application from components that include source or binary code, images, and templates by using the OpenShift Container Platform CLI.

The set of objects created by `new-app` depends on the artifacts passed as input: source repositories, images, or templates.

2.3.1. Creating an application from source code

With the `new-app` command you can create applications from source code in a local or remote Git repository.

The `new-app` command creates a build configuration, which itself creates a new application image from your source code. The `new-app` command typically also creates a Deployment object to deploy the new image, and a service to provide load-balanced access to the deployment running your image.

OpenShift Container Platform automatically detects whether the pipeline or source build strategy should be used, and in the case of source builds, detects an appropriate language builder image.

2.3.1.1. Local

To create an application from a Git repository in a local directory:

```bash
$ oc new-app /<path to source code>
```

**NOTE**

If you use a local Git repository, the repository must have a remote named `origin` that points to a URL that is accessible by the OpenShift Container Platform cluster. If there is no recognized remote, running the `new-app` command will create a binary build.

2.3.1.2. Remote

To create an application from a remote Git repository:
To create an application from a private remote Git repository:

```bash
$ oc new-app https://github.com/sclorg/cakephp-ex
```

To create an application from a private remote Git repository:

```bash
$ oc new-app https://github.com/youruser/yourprivaterepo --source-secret=yoursecret
```

**NOTE**

If you use a private remote Git repository, you can use the `--source-secret` flag to specify an existing source clone secret that will get injected into your build config to access the repository.

You can use a subdirectory of your source code repository by specifying a `--context-dir` flag. To create an application from a remote Git repository and a context subdirectory:

```bash
$ oc new-app https://github.com/sclorg/s2i-ruby-container.git
   --context-dir=2.0/test/puma-test-app
```

Also, when specifying a remote URL, you can specify a Git branch to use by appending `#` `<branch_name>` to the end of the URL:

```bash
$ oc new-app https://github.com/openshift/ruby-hello-world.git#beta4
```

### 2.3.1.3. Build strategy detection

If a Jenkins file exists in the root or specified context directory of the source repository when creating a new application, OpenShift Container Platform generates a pipeline build strategy. Otherwise, it generates a source build strategy.

Override the build strategy by setting the `--strategy` flag to either `pipeline` or `source`.

```bash
$ oc new-app /home/user/code/myapp --strategy=docker
```

**NOTE**

The `oc` command requires that files containing build sources are available in a remote Git repository. For all source builds, you must use `git remote -v`.

### 2.3.1.4. Language detection

If you use the source build strategy, `new-app` attempts to determine the language builder to use by the presence of certain files in the root or specified context directory of the repository:

#### Table 2.1. Languages detected by `new-app`

<table>
<thead>
<tr>
<th>Language</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>dotnet</td>
<td><code>project.json</code>, <code>*.csproj</code></td>
</tr>
<tr>
<td>jee</td>
<td><code>pom.xml</code></td>
</tr>
</tbody>
</table>
After a language is detected, `new-app` searches the OpenShift Container Platform server for image stream tags that have a `supports` annotation matching the detected language, or an image stream that matches the name of the detected language. If a match is not found, `new-app` searches the Docker Hub registry for an image that matches the detected language based on name.

You can override the image the builder uses for a particular source repository by specifying the image, either an image stream or container specification, and the repository with a ~ as a separator. Note that if this is done, build strategy detection and language detection are not carried out.

For example, to use the `myproject/my-ruby` imagestream with the source in a remote repository:

```
$ oc new-app myproject/my-ruby~https://github.com/openshift/ruby-hello-world.git
```

To use the `openshift/ruby-20-centos7:latest` container image stream with the source in a local repository:

```
$ oc new-app openshift/ruby-20-centos7:latest~/home/user/code/my-ruby-app
```

### NOTE

Language detection requires the Git client to be locally installed so that your repository can be cloned and inspected. If Git is not available, you can avoid the language detection step by specifying the builder image to use with your repository with the `<image>~<repository>` syntax.

The `-i <image> <repository>` invocation requires that `new-app` attempt to clone `repository` to determine what type of artifact it is, so this will fail if Git is not available.

The `-i <image> --code <repository>` invocation requires `new-app` clone `repository` to determine whether `image` should be used as a builder for the source code, or deployed separately, as in the case of a database image.

### 2.3.2. Creating an application from an image

<table>
<thead>
<tr>
<th>Language</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodejs</td>
<td>app.json, package.json</td>
</tr>
<tr>
<td>perl</td>
<td>cpanfile, index.pl</td>
</tr>
<tr>
<td>php</td>
<td>composer.json, index.php</td>
</tr>
<tr>
<td>python</td>
<td>requirements.txt, setup.py</td>
</tr>
<tr>
<td>ruby</td>
<td>Gemfile, Rakefile, config.ru</td>
</tr>
<tr>
<td>scala</td>
<td>build.sbt</td>
</tr>
<tr>
<td>golang</td>
<td>Godeps.main.go</td>
</tr>
</tbody>
</table>
You can deploy an application from an existing image. Images can come from image streams in the OpenShift Container Platform server, images in a specific registry, or images in the local Docker server.

The `new-app` command attempts to determine the type of image specified in the arguments passed to it. However, you can explicitly tell `new-app` whether the image is a container image using the `--docker-image` argument or an image stream using the `-i|--image-stream` argument.

**NOTE**

If you specify an image from your local Docker repository, you must ensure that the same image is available to the OpenShift Container Platform cluster nodes.

### 2.3.2.1. Docker Hub MySQL image

Create an application from the Docker Hub MySQL image, for example:

```
$ oc new-app mysql
```

### 2.3.2.2. Image in a private registry

Create an application using an image in a private registry, specify the full container image specification:

```
$ oc new-app myregistry:5000/example/myimage
```

### 2.3.2.3. Existing image stream and optional image stream tag

Create an application from an existing image stream and optional image stream tag:

```
$ oc new-app my-stream:v1
```

### 2.3.3. Creating an application from a template

You can create an application from a previously stored template or from a template file, by specifying the name of the template as an argument. For example, you can store a sample application template and use it to create an application.

Upload an application template to your current project's template library. The following example uploads an application template from a file called `examples/sample-app/application-template-stibuild.json`:

```
$ oc create -f examples/sample-app/application-template-stibuild.json
```

Then create a new application by referencing the application template. In this example, the template name is `ruby-helloworld-sample`:

```
$ oc new-app ruby-helloworld-sample
```

To create a new application by referencing a template file in your local file system, without first storing it in OpenShift Container Platform, use the `-f|--file` argument. For example:

```
$ oc new-app -f examples/sample-app/application-template-stibuild.json
```

OpenShift Container Platform 4.10 Building applications
2.3.3.1. Template parameters

When creating an application based on a template, use the `-p|--param` argument to set parameter values that are defined by the template:

```
$ oc new-app ruby-helloworld-sample \
   -p ADMIN_USERNAME=admin -p ADMIN_PASSWORD=mypassword
```

You can store your parameters in a file, then use that file with `--param-file` when instantiating a template. If you want to read the parameters from standard input, use `--param-file=-`. The following is an example file called `helloworld.params`:

```
ADMIN_USERNAME=admin
ADMIN_PASSWORD=mypassword
```

Reference the parameters in the file when instantiating a template:

```
$ oc new-app ruby-helloworld-sample --param-file=helloworld.params
```

2.3.4. Modifying application creation

The `new-app` command generates OpenShift Container Platform objects that build, deploy, and run the application that is created. Normally, these objects are created in the current project and assigned names that are derived from the input source repositories or the input images. However, with `new-app` you can modify this behavior.

<table>
<thead>
<tr>
<th>Table 2.2. new-app output objects</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
</tr>
<tr>
<td><strong>BuildConfig</strong></td>
</tr>
<tr>
<td><strong>ImageStreams</strong></td>
</tr>
<tr>
<td><strong>DeploymentConfig</strong></td>
</tr>
<tr>
<td><strong>Service</strong></td>
</tr>
<tr>
<td><strong>Other</strong></td>
</tr>
</tbody>
</table>
2.3.4.1. Specifying environment variables

When generating applications from a template, source, or an image, you can use the `-e|--env` argument to pass environment variables to the application container at run time:

```
$ oc new-app openshift/postgresql-92-centos7 \
  -e POSTGRESQL_USER=user \
  -e POSTGRESQL_DATABASE=db \
  -e POSTGRESQL_PASSWORD=password
```

The variables can also be read from file using the `--env-file` argument. The following is an example file called `postgresql.env`:

```
POSTGRESQL_USER=user
POSTGRESQL_DATABASE=db
POSTGRESQL_PASSWORD=password
```

Read the variables from the file:

```
$ oc new-app openshift/postgresql-92-centos7 --env-file=postgresql.env
```

Additionally, environment variables can be given on standard input by using `--env-file=`:

```
$ oc new-app openshift/postgresql-92-centos7 --env-file=-
```

**NOTE**

Any `BuildConfig` objects created as part of `new-app` processing are not updated with environment variables passed with the `-e|--env` or `--env-file` argument.

2.3.4.2. Specifying build environment variables

When generating applications from a template, source, or an image, you can use the `--build-env` argument to pass environment variables to the build container at run time:

```
$ oc new-app openshift/ruby-23-centos7 \
  --build-env HTTP_PROXY=http://myproxy.net:1337/ \
  --build-env GEM_HOME=~/.gem
```

The variables can also be read from a file using the `--build-env-file` argument. The following is an example file called `ruby.env`:

```
HTTP_PROXY=http://myproxy.net:1337/
GEM_HOME=~/.gem
```

Read the variables from the file:

```
$ oc new-app openshift/ruby-23-centos7 --build-env-file=ruby.env
```

Additionally, environment variables can be given on standard input by using `--build-env-file=`:

```
$ oc new-app openshift/ruby-23-centos7 --build-env-file=-
```
2.3.4.3. Specifying labels

When generating applications from source, images, or templates, you can use the `-l|--label` argument to add labels to the created objects. Labels make it easy to collectively select, configure, and delete objects associated with the application.

```
$ oc new-app https://github.com/openshift/ruby-hello-world -l name=hello-world
```

2.3.4.4. Viewing the output without creation

To see a dry-run of running the `new-app` command, you can use the `-o|--output` argument with a `yaml` or `json` value. You can then use the output to preview the objects that are created or redirect it to a file that you can edit. After you are satisfied, you can use `oc create` to create the OpenShift Container Platform objects.

To output `new-app` artifacts to a file, run the following:

```
$ oc new-app https://github.com/openshift/ruby-hello-world
   -o yaml > myapp.yaml
```

Edit the file:

```
$ vi myapp.yaml
```

Create a new application by referencing the file:

```
$ oc create -f myapp.yaml
```

2.3.4.5. Creating objects with different names

Objects created by `new-app` are normally named after the source repository, or the image used to generate them. You can set the name of the objects produced by adding a `--name` flag to the command:

```
$ oc new-app https://github.com/openshift/ruby-hello-world --name=myapp
```

2.3.4.6. Creating objects in a different project

Normally, `new-app` creates objects in the current project. However, you can create objects in a different project by using the `-n|--namespace` argument:

```
$ oc new-app https://github.com/openshift/ruby-hello-world -n myproject
```

2.3.4.7. Creating multiple objects

The `new-app` command allows creating multiple applications specifying multiple parameters to `new-app`. Labels specified in the command line apply to all objects created by the single command. Environment variables apply to all components created from source or images.

To create an application from a source repository and a Docker Hub image:

```
$ oc new-app https://github.com/openshift/ruby-hello-world mysql
```
NOTE
If a source code repository and a builder image are specified as separate arguments, new-app uses the builder image as the builder for the source code repository. If this is not the intent, specify the required builder image for the source using the ~ separator.

2.3.4.8. Grouping images and source in a single pod

The new-app command allows deploying multiple images together in a single pod. To specify which images to group together, use the + separator. The --group command line argument can also be used to specify the images that should be grouped together. To group the image built from a source repository with other images, specify its builder image in the group:

```
$ oc new-app ruby+mysql
```

To deploy an image built from source and an external image together:

```
$ oc new-app \
  ruby~https://github.com/openshift/ruby-hello-world \
  mysql \
  --group=ruby+mysql
```  

2.3.4.9. Searching for images, templates, and other inputs

To search for images, templates, and other inputs for the oc new-app command, add the --search and --list flags. For example, to find all of the images or templates that include PHP:

```
$ oc new-app --search php
```
CHAPTER 3. VIEWING APPLICATION COMPOSITION USING THE TOPOLOGY VIEW

The **Topology** view in the **Developer** perspective of the web console provides a visual representation of all the applications within a project, their build status, and the components and services associated with them.

### 3.1. PREREQUISITES

To view your applications in the **Topology** view and interact with them, ensure that:

- You have logged in to the web console.
- You have the appropriate roles and permissions in a project to create applications and other workloads in OpenShift Container Platform.
- You have created and deployed an application on OpenShift Container Platform using the Developer perspective.
- You are in the Developer perspective.

### 3.2. VIEWING THE TOPOLOGY OF YOUR APPLICATION

You can navigate to the **Topology** view using the left navigation panel in the Developer perspective. After you deploy an application, you are directed automatically to the **Graph view** where you can see the status of the application pods, quickly access the application on a public URL, access the source code to modify it, and see the status of your last build. You can zoom in and out to see more details for a particular application.

The **Topology** view also provides you the option to monitor your applications using the **List view**. Use the **List view** icon ( ) to see a list of all your applications and use the **Graph view** icon ( ) to switch back to the graph view.

You can customize the views as required using the following:

- Use the **Find by name** field to find the required components. Search results may appear outside of the visible area; click **Fit to Screen** from the lower-left toolbar to resize the **Topology** view to show all components.

- Use the **Display Options** drop-down list to configure the **Topology** view of the various application groupings. The options are available depending on the types of components deployed in the project:
  - **Mode** *(Connectivity or Consumption)*
    - Connectivity: Select to show all the connections between the different nodes in the topology.
    - Consumption: Select to show the resource consumption for all nodes in the topology.
  - **Expand** group:
    - Virtual Machines: Toggle to show or hide the virtual machines.
Application Groupings: Clear to condense the application groups into cards with an overview of an application group and alerts associated with it.

Helm Releases: Clear to condense the components deployed as Helm Release into cards with an overview of a given release.

Knative Services: Clear to condense the Knative Service components into cards with an overview of a given component.

Operator Groupings: Clear to condense the components deployed with an Operator into cards with an overview of the given group.

- **Show** elements based on **Pod Count** or **Labels**
  - Pod Count: Select to show the number of pods of a component in the component icon.
  - Labels: Toggle to show or hide the component labels.

### 3.3. INTERACTING WITH APPLICATIONS AND COMPONENTS

The **Topology** view in the **Developer** perspective of the web console provides the following options to interact with applications and components:

- Click **Open URL** to see your application exposed by the route on a public URL.
- Click **Edit Source code** to access your source code and modify it.

**NOTE**

This feature is available only when you create applications using the **From Git**, **From Catalog**, and the **From Dockerfile** options.

- Hover your cursor over the lower left icon on the pod to see the name of the latest build and its status. The status of the application build is indicated as **New ( )**, **Pending ( )**, **Running ( )**, **Completed ( )**, **Failed ( )**, and **Canceled ( )**.

- The status or phase of the pod is indicated by different colors and tooltips as:
  - **Running ( )**: The pod is bound to a node and all of the containers are created. At least one container is still running or is in the process of starting or restarting.
  - **Not Ready ( )**: The pods which are running multiple containers, not all containers are ready.
  - **Warning ( )**: Containers in pods are being terminated, however termination did not succeed. Some containers may be other states.
  - **Failed ( )**: All containers in the pod terminated but least one container has terminated in failure. That is, the container either exited with non-zero status or was terminated by the system.
  - **Pending ( )**: The pod is accepted by the Kubernetes cluster, but one or more of the containers has not been set up and made ready to run. This includes time a pod spends waiting to be scheduled as well as the time spent downloading container images over the network.
- **Succeeded**: All containers in the pod terminated successfully and will not be restarted.

- **Terminating**: When a pod is being deleted, it is shown as Terminating by some kubectl commands. Terminating status is not one of the pod phases. A pod is granted a graceful termination period, which defaults to 30 seconds.

- **Unknown**: The state of the pod could not be obtained. This phase typically occurs due to an error in communicating with the node where the pod should be running.

- After you create an application and an image is deployed, the status is shown as **Pending**. After the application is built, it is displayed as **Running**.

**Figure 3.1. Application topology**

The application resource name is appended with indicators for the different types of resource objects as follows:

- **CJ**: CronJob
- **D**: Deployment
- **DC**: DeploymentConfig
- **DS**: DaemonSet
- **J**: Job
- **P**: Pod
- **SS**: StatefulSet
- **K** (Knative): A serverless application
Serverless applications take some time to load and display on the Graph view. When you deploy a serverless application, it first creates a service resource and then a revision. After that, it is deployed and displayed on the Graph view. If it is the only workload, you might be redirected to the Add page. After the revision is deployed, the serverless application is displayed on the Graph view.

3.4. SCALING APPLICATION PODS AND CHECKING BUILDS AND ROUTES

The Topology view provides the details of the deployed components in the Overview panel. You can use the Overview and Resources tabs to scale the application pods, check build status, services, and routes as follows:

- Click on the component node to see the Overview panel to the right. Use the Overview tab to:
  - Scale your pods using the up and down arrows to increase or decrease the number of instances of the application manually. For serverless applications, the pods are automatically scaled down to zero when idle and scaled up depending on the channel traffic.
  - Check the Labels, Annotations, and Status of the application.

- Click the Resources tab to:
  - See the list of all the pods, view their status, access logs, and click on the pod to see the pod details.
  - See the builds, their status, access logs, and start a new build if needed.
  - See the services and routes used by the component.

For serverless applications, the Resources tab provides information on the revision, routes, and the configurations used for that component.

3.5. ADDING COMPONENTS TO AN EXISTING PROJECT

Procedure

1. Click Add to Project ( ) next to left navigation pane or press Ctrl+Space

2. Search for the component and select Create or press Enter to add the component to the project and see it in the topology Graph view.
Alternatively, you can also use the Import from Git, Container Image, Database, From Catalog, Operator Backed, Helm Charts, Samples, or Upload JAR file options in the context menu by right-clicking in the topology Graph view to add a component to your project.
3.6. GROUPING MULTIPLE COMPONENTS WITHIN AN APPLICATION

You can use the +Add view to add multiple components or services to your project and use the topology Graph view to group applications and resources within an application group.

Prerequisites

- You have created and deployed minimum two or more components on OpenShift Container Platform using the Developer perspective.

Procedure

- To add a service to the existing application group, press Shift+ drag it to the existing application group. Dragging a component and adding it to an application group adds the required labels to the component.

Figure 3.4. Application grouping

Alternatively, you can also add the component to an application as follows:

1. Click the service pod to see the Overview panel to the right.
2. Click the Actions drop-down menu and select Edit Application Grouping.
3. In the Edit Application Grouping dialog box, click the Application drop-down list, and select an appropriate application group.
4. Click Save to add the service to the application group.

You can remove a component from an application group by selecting the component and using Shift+ drag to drag it out of the application group.
3.7. ADDING SERVICES TO YOUR APPLICATION

To add a service to your application use the \textit{+Add} actions using the context menu in the topology \textit{Graph view}.

\textbf{NOTE}

In addition to the context menu, you can add services by using the sidebar or hovering and dragging the dangling arrow from the application group.

\textbf{Procedure}

1. Right-click an application group in the topology \textit{Graph view} to display the context menu.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig35}
\caption{Add resource context menu}
\end{figure}

2. Use \textit{Add to Application} to select a method for adding a service to the application group, such as \textit{From Git}, \textit{Container Image}, \textit{From Dockerfile}, \textit{From Devfile}, \textit{Upload JAR file}, \textit{Event Source}, \textit{Channel}, or \textit{Broker}.

3. Complete the form for the method you choose and click \textit{Create}. For example, to add a service based on the source code in your Git repository, choose the \textit{From Git} method, fill in the \textit{Import from Git} form, and click \textit{Create}.
3.8. REMOVING SERVICES FROM YOUR APPLICATION

In the topology **Graph view** remove a service from your application using the context menu.

**Procedure**

1. Right-click on a service in an application group in the topology **Graph view** to display the context menu.

2. Select **Delete Deployment** to delete the service.

Figure 3.6. Deleting deployment option
3.9. LABELS AND ANNOTATIONS USED FOR THE TOPOLOGY VIEW

The **Topology** view uses the following labels and annotations:

**Icon displayed in the node**
- Icons in the node are defined by looking for matching icons using the `app.openshift.io/runtime` label, followed by the `app.kubernetes.io/name` label. This matching is done using a predefined set of icons.

**Link to the source code editor or the source**
- The `app.openshift.io/vcs-uri` annotation is used to create links to the source code editor.

**Node Connector**
- The `app.openshift.io/connects-to` annotation is used to connect the nodes.

**App grouping**
- The `app.kubernetes.io/part-of=<appname>` label is used to group the applications, services, and components.

For detailed information on the labels and annotations OpenShift Container Platform applications must use, see Guidelines for labels and annotations for OpenShift applications.

3.10. ADDITIONAL RESOURCES

- See Importing a codebase from Git to create an application for more information on creating an application from Git.
- See Connecting an application to a service using the Developer perspective.
CHAPTER 4. CONNECTING APPLICATIONS TO SERVICES

4.1. RELEASE NOTES FOR SERVICE BINDING OPERATOR

The Service Binding Operator consists of a controller and an accompanying custom resource definition (CRD) for service binding. It manages the data plane for workloads and backing services. The Service Binding Controller reads the data made available by the control plane of backing services. Then, it projects this data to workloads according to the rules specified through the ServiceBinding resource.

With Service Binding Operator, you can:

- Bind your workloads together with Operator-managed backing services.
- Automate configuration of binding data.
- Provide service operators a low-touch administrative experience to provision and manage access to services.
- Enrich development lifecycle with a consistent and declarative service binding method that eliminates discrepancies in cluster environments.

The custom resource definition (CRD) of the Service Binding Operator supports the following APIs:

- Service Binding with the binding.operators.coreos.com API group.
- Service Binding (Spec API Tech Preview) with the servicebinding.io API group.

IMPORTANT

Service Binding (Spec API Tech Preview) with the servicebinding.io API group is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

4.1.1. Support matrix

Some features in this release are currently in Technology Preview. These experimental features are not intended for production use.

In the table, features are marked with the following statuses:

- **TP**: Technology Preview
- **GA**: General Availability

Note the following scope of support on the Red Hat Customer Portal for these features:

Table 4.1. Support matrix
<table>
<thead>
<tr>
<th>Service Binding Operator</th>
<th>API Group and Support Status</th>
<th>OpenShift Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>binding.operators.coreos.com</td>
<td>servicebinding.io</td>
</tr>
<tr>
<td>Version</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1.1</td>
<td>GA</td>
<td>TP</td>
</tr>
<tr>
<td>1.1</td>
<td>GA</td>
<td>TP</td>
</tr>
<tr>
<td>1.0.1</td>
<td>GA</td>
<td>TP</td>
</tr>
<tr>
<td>1.0</td>
<td>GA</td>
<td>TP</td>
</tr>
</tbody>
</table>

### 4.1.2. Making open source more inclusive

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see [Red Hat CTO Chris Wright’s message](https://example.com).

### 4.1.3. Release notes for Service Binding Operator 1.1.1

Service Binding Operator 1.1.1 is now available on OpenShift Container Platform 4.7, 4.8, 4.9, and 4.10.

#### 4.1.3.1. Fixed issues

- Before this update, a security vulnerability [CVE-2021-38561](https://example.com) was noted for Service Binding Operator Helm chart. This update fixes the **CVE-2021-38561** error and updates the `golang.org/x/text` package from v0.3.6 to v0.3.7. [APPSVC-1124](https://example.com).

- Before this update, users of the Developer Sandbox did not have sufficient permissions to read `ClusterWorkloadResourceMapping` resources. As a result, Service Binding Operator prevented all service bindings from being successful. With this update, the Service Binding Operator now includes the appropriate role-based access control (RBAC) rules for any authenticated subject including the Developer Sandbox users. These RBAC rules allow the Service Binding Operator to **get**, **list**, and **watch** the `ClusterWorkloadResourceMapping` resources for the Developer Sandbox users and to process service bindings successfully. [APPSVC-1135](https://example.com).

#### 4.1.3.2. Known issues

- There is currently a known issue with installing Service Binding Operator in a single namespace installation mode. The absence of an appropriate namespace-scoped role-based access control (RBAC) rule prevents the successful binding of an application to a few known Operator-backed services that the Service Binding Operator can automatically detect and bind to. When this happens, it generates an error message similar to the following example:

**Example error message**

```
`postgresclusters.postgres-operator.crunchydata.com "hippo" is forbidden:
```

---

CHAPTER 4. CONNECTING APPLICATIONS TO SERVICES

47
Workaround 1: Install the Service Binding Operator in the all namespaces installation mode. As a result, the appropriate cluster-scoped RBAC rule now exists and the binding succeeds.

Workaround 2: If you cannot install the Service Binding Operator in the all namespaces installation mode, install the following role binding into the namespace where the Service Binding Operator is installed:

**Example: Role binding for Crunchy Postgres Operator**

```yaml
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: service-binding-crunchy-postgres-viewer
subjects:
  - kind: ServiceAccount
    name: service-binding-operator
    roleRef:
      apiGroup: rbac.authorization.k8s.io
      kind: ClusterRole
      name: service-binding-crunchy-postgres-viewer-role
```

**APPSVC-1062**

- Currently, when you modify the `ClusterWorkloadResourceMapping` resources, the Service Binding Operator does not implement correct behavior. As a workaround, perform the following steps:

  1. Delete any `ServiceBinding` resources that use the corresponding `ClusterWorkloadResourceMapping` resource.
  3. Re-apply the `ServiceBinding` resources that you previously removed in step 1.

**APPSVC-1102**

**4.1.4. Release notes for Service Binding Operator 1.1**

Service Binding Operator is now available on OpenShift Container Platform 4.7, 4.8, 4.9, and 4.10.

**4.1.4.1. New features**

This section highlights what is new in Service Binding Operator 1.1:

- Service Binding Options
  - Workload resource mapping: Define exactly where binding data needs to be projected for the secondary workloads.
  - Bind new workloads using a label selector.
4.1.4.2. Fixed issues

- Before this update, service bindings that used label selectors to pick up workloads did not project service binding data into the new workloads that matched the given label selectors. As a result, the Service Binding Operator could not periodically bind such new workloads. With this update, service bindings now project service binding data into the new workloads that match the given label selector. The Service Binding Operator now periodically attempts to find and bind such new workloads. **APPsvc-1083**

4.1.4.3. Known issues

- There is currently a known issue with installing Service Binding Operator in a single namespace installation mode. The absence of an appropriate namespace-scoped role-based access control (RBAC) rule prevents the successful binding of an application to a few known Operator-backed services that the Service Binding Operator can automatically detect and bind to. When this happens, it generates an error message similar to the following example:

**Example error message**

```
`postgresclusters.postgres-operator.crunchydata.com "hippo" is forbidden:
User "system:serviceaccount:my-petclinic:service-binding-operator" cannot
get resource "postgresclusters" in API group "postgres-operator.crunchydata.com"
in the namespace "my-petclinic"
```

Workaround 1: Install the Service Binding Operator in the **all namespaces** installation mode. As a result, the appropriate cluster-scoped RBAC rule now exists and the binding succeeds.

Workaround 2: If you cannot install the Service Binding Operator in the **all namespaces** installation mode, install the following role binding into the namespace where the Service Binding Operator is installed:

**Example: Role binding for Crunchy Postgres Operator**

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: service-binding-crunchy-postgres-viewer
subjects:
  - kind: ServiceAccount
    name: service-binding-operator
    roleRef:
      apiGroup: rbac.authorization.k8s.io
      kind: ClusterRole
      name: service-binding-crunchy-postgres-viewer-role
```

**APPsvc-1062**

- Currently, when you modify the **ClusterWorkloadResourceMapping** resources, the Service Binding Operator does not implement correct behavior. As a workaround, perform the following steps:

  1. Delete any **ServiceBinding** resources that use the corresponding **ClusterWorkloadResourceMapping** resource.

3. Re-apply the **ServiceBinding** resources that you previously removed in step 1.

**APPSVC-1102**

### 4.1.5. Release notes for Service Binding Operator 1.0.1

Service Binding Operator is now available on OpenShift Container Platform 4.7, 4.8 and 4.9.

Service Binding Operator 1.0.1 supports OpenShift Container Platform 4.9 and later running on:

- IBM Power Systems
- IBM Z and LinuxONE

The custom resource definition (CRD) of the Service Binding Operator 1.0.1 supports the following APIs:

- **Service Binding** with the `binding.operators.coreos.com` API group.
- **Service Binding (Spec API Tech Preview)** with the `servicebinding.io` API group.

**IMPORTANT**

**Service Binding (Spec API Tech Preview)** with the `servicebinding.io` API group is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see [https://access.redhat.com/support/offerings/techpreview/](https://access.redhat.com/support/offerings/techpreview/).

#### 4.1.5.1. Support matrix

Some features in this release are currently in Technology Preview. These experimental features are not intended for production use.

**Technology Preview Features Support Scope**

In the table below, features are marked with the following statuses:

- **TP**: Technology Preview
- **GA**: General Availability

Note the following scope of support on the Red Hat Customer Portal for these features:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Service Binding Operator 1.0.1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>binding.operators.coreos.com</code> API group</td>
<td>GA</td>
</tr>
<tr>
<td><code>servicebinding.io</code> API group</td>
<td>TP</td>
</tr>
</tbody>
</table>
4.1.5.2. Fixed issues

- Before this update, binding the data values from a Cluster custom resource (CR) of the `postgresql.k8s.enterprisedb.io/v1` API collected the `host` binding value from the `.metadata.name` field of the CR. The collected binding value is an incorrect hostname and the correct hostname is available at the `.status.writeService` field. With this update, the annotations that the Service Binding Operator uses to expose the binding data values from the backing service CR are now modified to collect the `host` binding value from the `.status.writeService` field. The Service Binding Operator uses these modified annotations to project the correct hostname in the `host` and `provider` bindings. APPSVC-1040

- Before this update, when you would bind a PostgresCluster CR of the `postgres-operator.crunchydata.com/v1beta1` API, the binding data values did not include the values for the database certificates. As a result, the application failed to connect to the database. With this update, modifications to the annotations that the Service Binding Operator uses to expose the binding data from the backing service CR now include the database certificates. The Service Binding Operator uses these modified annotations to project the correct `ca.crt`, `tls.crt`, and `tls.key` certificate files. APPSVC-1045

- Before this update, when you would bind a PerconaXtraDBCluster custom resource (CR) of the `pxc.percona.com` API, the binding data values did not include the `port` and `database` values. These binding values along with the others already projected are necessary for an application to successfully connect to the database service. With this update, the annotations that the Service Binding Operator uses to expose the binding data values from the backing service CR are now modified to project the additional `port` and `database` binding values. The Service Binding Operator uses these modified annotations to project the complete set of binding values that the application can use to successfully connect to the database service. APPSVC-1073

4.1.5.3. Known issues

- Currently, when you install the Service Binding Operator in the single namespace installation mode, the absence of an appropriate namespace-scoped role-based access control (RBAC) rule prevents the successful binding of an application to a few known Operator-backed services that the Service Binding Operator can automatically detect and bind to. In addition, the following error message is generated:

Example error message

```
"postgresclusters.postgres-operator.crunchydata.com "hippo" is forbidden: User "system:serviceaccount:my-petclinic:service-binding-operator" cannot get resource "postgresclusters" in API group "postgres-operator.crunchydata.com" in the namespace "my-petclinic"
```

Workaround 1: Install the Service Binding Operator in the all namespaces installation mode. As a result, the appropriate cluster-scoped RBAC rule now exists and the binding succeeds.

Workaround 2: If you cannot install the Service Binding Operator in the all namespaces installation mode, install the following role binding into the namespace where the Service Binding Operator is installed:

Example: Role binding for Crunchy Postgres Operator

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
```
Service Binding Operator is now available on OpenShift Container Platform 4.7, 4.8 and 4.9.

The custom resource definition (CRD) of the Service Binding Operator 1.0 supports the following APIs:

- **Service Binding** with the `binding.operators.coreos.com` API group.
- **Service Binding (Spec API Tech Preview)** with the `servicebinding.io` API group.

**IMPORTANT**

**Service Binding (Spec API Tech Preview)** with the `servicebinding.io` API group is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see [https://access.redhat.com/support/offerings/techpreview/](https://access.redhat.com/support/offerings/techpreview/).

### 4.1.6.1. Support matrix

Some features in this release are currently in Technology Preview. These experimental features are not intended for production use.

**Technology Preview Features Support Scope**

In the table below, features are marked with the following statuses:

- **TP**: Technology Preview
- **GA**: General Availability

Note the following scope of support on the Red Hat Customer Portal for these features:

**Table 4.3. Support matrix**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Service Binding Operator 1.0</th>
</tr>
</thead>
</table>

APPsvc-1062
4.1.6.2. New features

Service Binding Operator 1.0 supports OpenShift Container Platform 4.9 and later running on:

- IBM Power Systems
- IBM Z and LinuxONE

This section highlights what is new in Service Binding Operator 1.0:

- Exposal of binding data from services
  - Based on annotations present in CRD, custom resources (CRs), or resources.
  - Based on descriptors present in Operator Lifecycle Manager (OLM) descriptors.
  - Support for provisioned services
- Workload projection
  - Projection of binding data as files, with volume mounts.
  - Projection of binding data as environment variables.
- Service Binding Options
  - Bind backing services in a namespace that is different from the workload namespace.
  - Project binding data into the specific container workloads.
  - Auto-detection of the binding data from resources owned by the backing service CR.
  - Compose custom binding data from the exposed binding data.
  - Support for non-[PodSpec](https://kubernetes.io/docs/concepts/containers/pod-security-policy/) compliant workload resources.
- Security
  - Support for role-based access control (RBAC).

4.1.7. Additional resources

- Understanding Service Binding Operator.

4.2. UNDERSTANDING SERVICE BINDING OPERATOR

Application developers need access to backing services to build and connect workloads. Connecting workloads to backing services is always a challenge because each service provider suggests a different way to access their secrets and consume them in a workload. In addition, manual configuration and
maintenance of this binding together of workloads and backing services make the process tedious, inefficient, and error-prone.

The Service Binding Operator enables application developers to easily bind workloads together with Operator-managed backing services, without any manual procedures to configure the binding connection.

4.2.1. Service Binding terminology

This section summarizes the basic terms used in Service Binding.

<table>
<thead>
<tr>
<th>Service binding</th>
<th>The representation of the action of providing information about a service to a workload. Examples include establishing the exchange of credentials between a Java application and a database that it requires.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backing service</td>
<td>Any service or software that the application consumes over the network as part of its normal operation. Examples include a database, a message broker, an application with REST endpoints, an event stream, an Application Performance Monitor (APM), or a Hardware Security Module (HSM).</td>
</tr>
<tr>
<td>Workload (application)</td>
<td>Any process, running within a container. Examples include a Spring Boot application, a NodeJS Express application, or a Ruby on Rails application.</td>
</tr>
<tr>
<td>Binding data</td>
<td>Information about a service that you use to configure the behavior of other resources within the cluster. Examples include credentials, connection details, volume mounts, or secrets.</td>
</tr>
<tr>
<td>Binding connection</td>
<td>Any connection that establishes an interaction between the connected components such as a bindable backing service and an application requiring that backing service.</td>
</tr>
</tbody>
</table>

4.2.2. About Service Binding Operator

The Service Binding Operator consists of a controller and an accompanying custom resource definition (CRD) for service binding. It manages the data plane for workloads and backing services. The Service Binding Controller reads the data made available by the control plane of backing services. Then, it projects this data to workloads according to the rules specified through the ServiceBinding resource.

As a result, the Service Binding Operator enables workloads to use backing services or external services by automatically collecting and sharing binding data with the workloads. The process involves making the backing service bindable and binding the workload and the service together.

4.2.2.1. Making an Operator-managed backing service bindable

To make a service bindable, as an Operator provider you need to expose the binding data required by workloads to bind with the services provided by the Operator. You can provide the binding data either as annotations or as descriptors in the CRD of the Operator that manages the backing service.

4.2.2.2. Binding a workload together with a backing service

By using the Service Binding Operator, as an application developer, you need to declare the intent of establishing a binding connection. You must create a Service Binding CR that references the backing service. This action triggers the Service Binding Operator to project the exposed binding data into the
workload. The Service Binding Operator receives the declared intent and binds the workload together with the backing service.

The CRD of the Service Binding Operator supports the following APIs:

- **Service Binding** with the `binding.operators.coreos.com` API group.
- **Service Binding (Spec API Tech Preview)** with the `servicebinding.io` API group.

**IMPORTANT**

Service Binding (Spec API Tech Preview) with the `servicebinding.io` API group is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process. For more information about the support scope of Red Hat Technology Preview features, see [https://access.redhat.com/support/offerings/techpreview/](https://access.redhat.com/support/offerings/techpreview/).

With Service Binding Operator, you can:

- Bind your workloads to Operator-managed backing services.
- Automate configuration of binding data.
- Provide service operators a low-touch administrative experience to provision and manage access to services.
- Enrich development lifecycle with a consistent and declarative service binding method that eliminates discrepancies in cluster environments.

### 4.2.3. Key features

- **Exposal of binding data from services**
  - Based on annotations present in CRD, custom resources (CRs), or resources.
  - Based on descriptors present in Operator Lifecycle Manager (OLM) descriptors.

- **Workload projection**
  - Projection of binding data as files, with volume mounts.
  - Projection of binding data as environment variables.

- **Service Binding Options**
  - Bind backing services in a namespace that is different from the workload namespace.
  - Project binding data into the specific container workloads.
  - Auto-detection of the binding data from resources owned by the backing service CR.
  - Compose custom binding data from the exposed binding data.
Support for non-PodSpec compliant workload resources.

- Security
  - Support for role-based access control (RBAC).

### 4.2.4. Additional resources

- Getting started with service binding.

### 4.3. INSTALLING SERVICE BINDING OPERATOR

This guide walks cluster administrators through the process of installing the Service Binding Operator to an OpenShift Container Platform cluster.

You can install Service Binding Operator on OpenShift Container Platform 4.7 and later.

#### 4.3.1. Installing the Service Binding Operator using the web console

You can install Service Binding Operator using the OpenShift Container Platform OperatorHub. When you install the Service Binding Operator, the custom resources (CRs) required for the service binding configuration are automatically installed along with the Operator.

**Prerequisites**

You have access to an OpenShift Container Platform cluster using an account with `cluster-admin` permissions.

**Procedure**

1. In the Administrator perspective of the web console, navigate to Operators → OperatorHub.

2. Use the Filter by keyword box to search for Service Binding Operator in the catalog. Click the Service Binding Operator tile.

3. Read the brief description about the Operator on the Service Binding Operator page. Click Install.

4. On the Install Operator page:
   
   a. Select All namespaces on the cluster (default) for the Installation Mode. This mode installs the Operator in the default openshift-operators namespace, which enables the Operator to watch and be made available to all namespaces in the cluster.

   b. Select Automatic for the Approval Strategy. This ensures that the future upgrades to the Operator are handled automatically by the Operator Lifecycle Manager (OLM). If you select the Manual approval strategy, OLM creates an update request. As a cluster administrator, you must then manually approve the OLM update request to update the Operator to the new version.

   c. Select an Update Channel
      
      - By default, the stable channel enables installation of the latest stable and supported release of the Service Binding Operator.

5. Click Install.
NOTE

The Operator is installed automatically into the openshift-operators namespace.


7. Verify that the Status is set to Succeeded to confirm successful installation of Service Binding Operator.

4.3.2. Additional Resources

- Getting started with service binding.

4.4. GETTING STARTED WITH SERVICE BINDING

The Service Binding Operator manages the data plane for workloads and backing services. This guide provides instructions with examples to help you create a database instance, deploy an application, and use the Service Binding Operator to create a binding connection between the application and the database service.

Prerequisites

- You have access to an OpenShift Container Platform cluster using an account with cluster-admin permissions.
- You have installed the oc CLI.
- You have installed Service Binding Operator from OperatorHub.
- You have installed the Crunchy Postgres for Kubernetes Operator from OperatorHub using the v5 Update channel. The installed Operator is available in an appropriate namespace, such as the my-petclinic namespace.

NOTE

You can create the namespace using the oc create namespace my-petclinic command.

4.4.1. Creating a PostgreSQL database instance

To create a PostgreSQL database instance, you must create a PostgresCluster custom resource (CR) and configure the database.

Procedure

1. Create the PostgresCluster CR in the my-petclinic namespace by running the following command in shell:

   ```bash
   $ oc apply -n my-petclinic -f - << EOD
   ---
   apiVersion: postgres-operator.crunchydata.com/v1beta1
   kind: PostgresCluster
   metadata:
   ```
The annotations added in this PostgresCluster CR enable the service binding connection and trigger the Operator reconciliation.

The output verifies that the database instance is created:

**Example output**

```
postgrescluster.postgres-operator.crunchydata.com/hippo created
```

2. After you have created the database instance, ensure that all the pods in the my-petclinic namespace are running:

```
$ oc get pods -n my-petclinic
```
The output, which takes a few minutes to display, verifies that the database is created and configured:

**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>hippo-backup-nqjg-2rq94</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>35s</td>
</tr>
<tr>
<td>hippo-instance1-nw92-0</td>
<td>3/3</td>
<td>Running</td>
<td>0</td>
<td>112s</td>
</tr>
<tr>
<td>hippo-pgbouncer-57b98f4476-znsk5</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>112s</td>
</tr>
<tr>
<td>hippo-repo-host-0</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>112s</td>
</tr>
</tbody>
</table>

After the database is configured, you can deploy the sample application and connect it to the database service.

### 4.4.2. Deploying the Spring PetClinic sample application

To deploy the Spring PetClinic sample application on an OpenShift Container Platform cluster, you must use a deployment configuration and configure your local environment to be able to test the application.

**Procedure**

1. Deploy the `spring-petclinic` application with the `PostgresCluster` custom resource (CR) by running the following command in shell:

   ```
   $ oc apply -n my-petclinic -f - << EOD
   ---
   apiVersion: apps/v1
   kind: Deployment
   metadata:
     name: spring-petclinic
   labels:
     app: spring-petclinic
   spec:
     replicas: 1
     selector:
       matchLabels:
         app: spring-petclinic
     template:
       metadata:
         labels:
           app: spring-petclinic
       spec:
         containers:
           - name: app
             image: quay.io/service-binding/spring-petclinic:latest
             imagePullPolicy: Always
             env:
               - name: SPRING_PROFILES_ACTIVE
                 value: postgres
             ports:
               - name: http
                 containerPort: 8080
   ---
   apiVersion: v1
   kind: Service
   ```
The output verifies that the Spring PetClinic sample application is created and deployed:

**Example output**

```
deployment.apps/spring-petclinic created
service/spring-petclinic created
```

**NOTE**

If you are deploying the application using Container images in the Developer perspective of the web console, you must enter the following environment variables under the Deployment section of the Advanced options:

- Name: SPRING_PROFILES_ACTIVE
- Value: postgres

2. Verify that the application is not yet connected to the database service by running the following command:

```
$ oc get pods -n my-petclinic
```

The output takes a few minutes to display the *CrashLoopBackOff* status:

**Example output**

```
NAME                                READY   STATUS             RESTARTS      AGE
spring-petclinic-5b4c7999d4-wzdtz   0/1     CrashLoopBackOff   4 (13s ago)   2m25s
```

At this stage, the pod fails to start. If you try to interact with the application, it returns errors.

You can now use the Service Binding Operator to connect the application to the database service.

### 4.4.3. Connecting the Spring PetClinic sample application to the PostgreSQL database service

To connect the sample application to the database service, you must create a ServiceBinding custom resource (CR) that triggers the Service Binding Operator to project the binding data into the application.
Procedure

1. Create a **ServiceBinding** CR to project the binding data:

   ```
   $ oc apply -n my-petclinic -f - << EOD
   ---
   apiVersion: binding.operators.coreos.com/v1alpha1
   kind: ServiceBinding
   metadata:
     name: spring-petclinic-pgcluster
   spec:
     services: 1
     - group: postgres-operator.crunchydata.com
       version: v1beta1
       kind: PostgresCluster 2
       name: hippo
     application: 3
     name: spring-petclinic
     group: apps
     version: v1
     resource: deployments
   EOD
   
   1. Specifies a list of service resources.
   2. The CR of the database.
   3. The sample application that points to a Deployment or any other similar resource with an embedded PodSpec.
   
   The output verifies that the **ServiceBinding** CR is created to project the binding data into the sample application.

   **Example output**
   
   servicebinding.binding.operators.coreos.com/spring-petclinic created
   
2. Verify that the request for service binding is successful:

   ```
   $ oc get servicebindings -n my-petclinic
   
   **Example output**
   
<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>REASON</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring-petclinic-pgcluster</td>
<td>True</td>
<td>ApplicationsBound</td>
<td>7s</td>
</tr>
</tbody>
</table>
   
   By default, the values from the binding data of the database service are projected as files into the workload container that runs the sample application. For example, all the values from the Secret resource are projected into the `bindings/spring-petclinic-pgcluster` directory.
NOTE

Optionally, you can also verify that the files in the application contain the projected binding data, by printing out the directory contents:

```bash
$ for i in username password host port type; do oc exec -it deploy/spring-petclinic -n my-petclinic -- /bin/bash -c "cd /tmp; find /bindings/*"$i" -exec echo -n {}:" " \; -exec cat {} \"; echo; done
```

Example output: With all the values from the secret resource

```
/bindings/spring-petclinic-pgcluster/username: hippo
/bindings/spring-petclinic-pgcluster/password: KXKF{nAI,I-J6zLt:W+FKnze
/bindings/spring-petclinic-pgcluster/host: hippo-primary.my-petclinic.svc
/bindings/spring-petclinic-pgcluster/port: 5432
/bindings/spring-petclinic-pgcluster/type: postgresql
```

3. Set up the port forwarding from the application port to access the sample application from your local environment:

```bash
$ oc port-forward --address 0.0.0.0 svc/spring-petclinic 8080:80 -n my-petclinic
```

Example output

```
Forwarding from 0.0.0.0:8080 -> 8080
Handling connection for 8080
```


You can now remotely access the Spring PetClinic sample application at `localhost:8080` and see that the application is now connected to the database service.

### 4.4.4. Additional Resources

- Installing Service Binding Operator.
- Creating applications using the Developer perspective.
- Managing resources from custom resource definitions.
- Known bindable Operators.

### 4.5. GETTING STARTED WITH SERVICE BINDING ON IBM POWER SYSTEMS, IBM Z, AND LINUXONE

The Service Binding Operator manages the data plane for workloads and backing services. This guide provides instructions with examples to help you create a database instance, deploy an application, and use the Service Binding Operator to create a binding connection between the application and the database service.

**Prerequisites**

- You have access to an OpenShift Container Platform cluster using an account with `cluster-admin` permissions.
4.5.1. Deploying a PostgreSQL Operator

**Procedure**

1. To deploy the Dev4Devs PostgreSQL Operator in the `my-petclinic` namespace run the following command in shell:

   ```bash
   $ oc apply -f - << EOD
   ---
   apiVersion: v1
   kind: Namespace
   metadata:
     name: my-petclinic
   ---
   apiVersion: operators.coreos.com/v1
   kind: OperatorGroup
   metadata:
     name: postgres-operator-group
     namespace: my-petclinic
   ---
   apiVersion: operators.coreos.com/v1alpha1
   kind: CatalogSource
   metadata:
     name: ibm-multiarch-catalog
     namespace: openshift-marketplace
   spec:
     sourceType: grpc
     image: quay.io/ibm/operator-registry-<architecture>  
     imagePullPolicy: IfNotPresent
     displayName: ibm-multiarch-catalog
     updateStrategy:
       registryPoll:
         interval: 30m
   ---
   apiVersion: operators.coreos.com/v1alpha1
   kind: Subscription
   metadata:
     name: postgresql-operator-dev4devs-com
     namespace: openshift-operators
   spec:
     channel: alpha
     installPlanApproval: Automatic
     name: postgresql-operator-dev4devs-com
     source: ibm-multiarch-catalog
     sourceNamespace: openshift-marketplace
   ---
   apiVersion: rbac.authorization.k8s.io/v1
   kind: ClusterRole
   metadata:
     name: database-view
   labels:
   ```
Servicebinding.IO/Controller: "true"

rules:
  - apiGroups:
    - postgresql.dev4devs.com
  resources:
    - databases
  verbs:
    - get
    - list

EOD

The Operator image.

- For IBM Power: quay.io/ibm/operator-registry-ppc64le:release-4.9
- For IBM Z and LinuxONE: quay.io/ibm/operator-registry-s390x:release-4.8

Verification

1. After the operator is installed, list the operator subscriptions in the openshift-operators namespace:

   $ oc get subs -n openshift-operators

Example output

<table>
<thead>
<tr>
<th>NAME</th>
<th>PACKAGE</th>
<th>SOURCE</th>
<th>CHANNEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>postgresql-operator-dev4devs-com</td>
<td>postgresql-operator-dev4devs-com</td>
<td>ibm-multiarch-catalog alpha</td>
<td>alpha</td>
</tr>
<tr>
<td>rh-service-binding-operator</td>
<td>rh-service-binding-operator</td>
<td>redhat-operators</td>
<td>stable</td>
</tr>
</tbody>
</table>

4.5.2. Creating a PostgreSQL database instance

To create a PostgreSQL database instance, you must create a Database custom resource (CR) and configure the database.

Procedure

1. Create the Database CR in the my-petclinic namespace by running the following command in shell:

   $ oc apply -f - << EOD
   apiVersion: postgresql.dev4devs.com/v1alpha1
   kind: Database
   metadata:
   name: sampledatabase
   namespace: my-petclinic
   annotations:
   host: sampledatabase
   type: postgresql
   port: "5432"
   service.binding/database: 'path={.spec.databaseName}'
   service.binding/port: 'path={.metadata.annotations.port}'
   EOD
The annotations added in this Database CR enable the service binding connection and trigger the Operator reconciliation.

The output verifies that the database instance is created:

Example output

```
database.postgresql.dev4devs.com/sampledatabase created
```

2. After you have created the database instance, ensure that all the pods in the my-petclinic namespace are running:

```
$ oc get pods -n my-petclinic
```

The output, which takes a few minutes to display, verifies that the database is created and configured:

Example output

```
NAME                                     READY    STATUS      RESTARTS   AGE
sampledatabase-cbc655488-74kss            0/1     Running        0       32s
```

After the database is configured, you can deploy the sample application and connect it to the database service.

4.5.3. Deploying the Spring PetClinic sample application

To deploy the Spring PetClinic sample application on an OpenShift Container Platform cluster, you must use a deployment configuration and configure your local environment to be able to test the application.

Procedure
1. Deploy the **spring-petclinic** application with the **PostgresCluster** custom resource (CR) by running the following command in shell:

```bash
$ oc apply -n my-petclinic -f - << EOD
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-petclinic
  labels:
    app: spring-petclinic
spec:
  replicas: 1
  selector:
    matchLabels:
      app: spring-petclinic
  template:
    metadata:
      labels:
        app: spring-petclinic
    spec:
      containers:
        - name: app
          image: quay.io/service-binding/spring-petclinic:latest
          imagePullPolicy: Always
          env:
            - name: SPRING_PROFILES_ACTIVE
              value: postgres
            - name: org.springframework.cloud.bindings.boot.enable
              value: "true"
          ports:
            - name: http
              containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: spring-petclinic
  name: spring-petclinic
spec:
  type: NodePort
  ports:
    - port: 80
      protocol: TCP
      targetPort: 8080
  selector:
    app: spring-petclinic
EOD
```

The output verifies that the Spring PetClinic sample application is created and deployed:

**Example output**

```
deployment.apps/spring-petclinic created
service/spring-petclinic created
```
NOTE

If you are deploying the application using Container images in the Developer perspective of the web console, you must enter the following environment variables under the Deployment section of the Advanced options:

- Name: SPRING_PROFILES_ACTIVE
- Value: postgres

2. Verify that the application is not yet connected to the database service by running the following command:

```
$ oc get pods -n my-petclinic
```

It takes take a few minutes until the CrashLoopBackOff status is displayed:

**Example output**

```
NAME                                READY STATUS             RESTARTS AGE
spring-petclinic-5b4c7999d4-wzdtz   0/1     CrashLoopBackOff   4 (13s ago) 2m25s
```

At this stage, the pod fails to start. If you try to interact with the application, it returns errors.

You can now use the Service Binding Operator to connect the application to the database service.

### 4.5.4. Connecting the Spring PetClinic sample application to the PostgreSQL database service

To connect the sample application to the database service, you must create a **ServiceBinding** custom resource (CR) that triggers the Service Binding Operator to project the binding data into the application.

**Procedure**

1. Create a **ServiceBinding** CR to project the binding data:

   ```
   $ oc apply -n my-petclinic -f - << EOD
   ---
   apiVersion: binding.operators.coreos.com/v1alpha1
   kind: ServiceBinding
   metadata:
     name: spring-petclinic-pgcluster
   spec:
     services: 1
     - group: postgresql.dev4devs.com
       kind: Database 2
       name: sampledatabase
       version: v1alpha1
     application: 3
     name: spring-petclinic
     group: apps
   ```
1. Specifies a list of service resources.
2. The CR of the database.
3. The sample application that points to a Deployment or any other similar resource with an embedded PodSpec.

The output verifies that the **ServiceBinding** CR is created to project the binding data into the sample application.

**Example output**

```
servicebinding.binding.operators.coreos.com/spring-petclinic created
```

2. Verify that the request for service binding is successful:

```
$ oc get servicebindings -n my-petclinic
```

**Example output**

```
NAME                            READY REASON              AGE
spring-petclinic-postgresql     True   ApplicationsBound   47m
```

By default, the values from the binding data of the database service are projected as files into the workload container that runs the sample application. For example, all the values from the Secret resource are projected into the `bindings/spring-petclinic-pgcluster` directory.

3. Once this is created, you can go to the topology to see the visual connection.

**Figure 4.1. Connecting spring-petclinic to a sample database**
4. Set up the port forwarding from the application port to access the sample application from your local environment:

$ oc port-forward --address 0.0.0.0 svc/spring-petclinic 8080:80 -n my-petclinic

Example output

Forwarding from 0.0.0.0:8080 -> 8080
Handling connection for 8080

You can now remotely access the Spring PetClinic sample application at localhost:8080 and see that the application is now connected to the database service.

4.5.5. Additional resources

- Installing Service Binding Operator
- Creating applications using the Developer perspective
- Managing resources from custom resource definitions

4.6. EXPOSING BINDING DATA FROM A SERVICE

Application developers need access to backing services to build and connect workloads. Connecting workloads to backing services is always a challenge because each service provider requires a different way to access their secrets and consume them in a workload.

The Service Binding Operator enables application developers to easily bind workloads together with operator-managed backing services, without any manual procedures to configure the binding connection. For the Service Binding Operator to provide the binding data, as an Operator provider or user who creates backing services, you must expose the binding data to be automatically detected by the Service Binding Operator. Then, the Service Binding Operator automatically collects the binding data from the backing service and shares it with a workload to provide a consistent and predictable experience.

4.6.1. Methods of exposing binding data

This section describes the methods you can use to expose the binding data.

Ensure that you know and understand your workload requirements and environment, and how it works with the provided services.

Binding data is exposed under the following circumstances:

- Backing service is available as a provisioned service resource.
  The service you intend to connect to is compliant with the Service Binding specification. You must create a Secret resource with all the required binding data values and reference it in the backing service custom resource (CR). The detection of all the binding data values is automatic.

- Backing service is not available as a provisioned service resource.
  You must expose the binding data from the backing service. Depending on your workload requirements and environment, you can choose any of the following methods to expose the binding data:
- Direct secret reference
- Declaring binding data through custom resource definition (CRD) or CR annotations
- Declaring binding data through Operator Lifecycle Manager (OLM) descriptors
- Detection of binding data through owned resources

4.6.1.1. Provisioned service

Provisioned service represents a backing service CR with a reference to a Secret resource placed in the .status.binding.name field of the backing service CR.

As an Operator provider or the user who creates backing services, you can use this method to be compliant with the Service Binding specification, by creating a Secret resource and referencing it in the .status.binding.name section of the backing service CR. This Secret resource must provide all the binding data values required for a workload to connect to the backing service.

The following examples show an AccountService CR that represents a backing service and a Secret resource referenced from the CR.

Example: AccountService CR

```yaml
apiVersion: example.com/v1alpha1
description: AccountService
ame: prod-account-service
spec:
  ...
  status:
    binding:
      name: hippo-pguser-hippo
```

Example: Referenced Secret resource

```yaml
apiVersion: v1
description: Secret
metadata:
  name: hippo-pguser-hippo
data:
  password: "MTBz"
  user: "Z3Vlc3Q="
  ...
```

When creating a service binding resource, you can directly give the details of the AccountService resource in the ServiceBinding specification as follows:

Example: ServiceBinding resource

```yaml
apiVersion: binding.operators.coreos.com/v1alpha1
description: ServiceBinding
metadata:
  name: account-service
spec:
  ...
```
IMPORTANT

Service Binding (Spec API Tech Preview) with the `servicebinding.io` API group is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process. For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

Example: `ServiceBinding` resource in Specification API

```yaml
apiVersion: servicebinding.io/v1alpha3
kind: ServiceBinding
metadata:
  name: account-service
spec:
  ...  
  service:
    apiVersion: example.com/v1alpha1
    kind: AccountService
    name: prod-account-service
  workload:
    apiVersion: apps/v1
    kind: Deployment
    name: spring-petclinic
```

This method exposes all the keys in the `hippo-pguser-hippo` referenced `Secret` resource as binding data that is to be projected into the workload.

4.6.1.2. Direct secret reference

You can use this method, if all the required binding data values are available in a `Secret` resource that you can reference in your Service Binding definition. In this method, a `ServiceBinding` resource directly references a `Secret` resource to connect to a service. All the keys in the `Secret` resource are exposed as binding data.

Example: Specification with the `binding.operators.coreos.com` API

```yaml
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
```
Example: Specification that is compliant with the servicebinding.io API

```yaml
apiVersion: servicebinding.io/v1alpha3
kind: ServiceBinding
metadata:
  name: account-service
spec:
  ...
  service:
    apiVersion: v1
    kind: Secret
    name: hippo-pguser-hippo
```

4.6.1.3. Declaring binding data through CRD or CR annotations

You can use this method to annotate the resources of the backing service to expose the binding data with specific annotations. Adding annotations under the `metadata` section alters the CRs and CRDs of the backing services. Service Binding Operator detects the annotations added to the CRs and CRDs and then creates a `Secret` resource with the values extracted based on the annotations.

The following examples show the annotations that are added under the `metadata` section and a referenced `ConfigMap` object from a resource:

Example: Exposing binding data from a Secret object defined in the CR annotations

```yaml
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
namespace: my-petclinic
annotations:
  service.binding: 'path={.metadata.name}-pguser-{.metadata.name},objectType=Secret'
```

The previous example places the name of the secret name in the `{.metadata.name}-pguser-{.metadata.name}` template that resolves to `hippo-pguser-hippo`. The template can contain multiple JSONPath expressions.

Example: Referenced Secret object from a resource

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: hippo-pguser-hippo
```
Example: Exposing binding data from a ConfigMap object defined in the CR annotations

```yaml
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
namespace: my-petclinic
annotations:
  service.binding: 'path={.metadata.name}-config,objectType=ConfigMap'
...
```

The previous example places the name of the config map in the `{.metadata.name}-config` template that resolves to `hippo-config`. The template can contain multiple JSONPath expressions.

Example: Referenced ConfigMap object from a resource

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: hippo-config
data:
  db_timeout: "10s"
  user: "hippo"
```

### 4.6.1.4. Declaring binding data through OLM descriptors

You can use this method if your backing service is provided by an Operator. If your Operator is distributed as an OLM bundle, you can add OLM descriptors to describe the binding data that is to be exposed. The OLM descriptors are part of Cluster Service Version resources. The Service Binding Operator detects the OLM descriptors and then creates a Secret resource with the values extracted based on the detected OLM descriptors.

You can expose the binding data by using the `specDescriptors` array and `statusDescriptors` array. The `specDescriptors` array specifies a path under the `.spec` section of a CR. The `statusDescriptors` array specifies a path under the `.status` section of a CR.

Following are the only two fields that are used for binding the data:

- **Path**: A dot-delimited path of the field on the object as described by the descriptor.
- **X-Descriptors**: Defines the binding data.

The following examples show how to define an X-Descriptor depending on the resource to which you point the path:

**Example: X-Descriptor definition for exposing a secret**

```yaml
- path: data.dbConfiguration
  x-descriptors:
  - urn:alm:descriptor:io.kubernetes:Secret
```
4.6.1.5. Detection of binding data through owned resources

You can use this method if your backing service owns one or more Kubernetes resources such as route, service, config map, or secret that you can use to detect the binding data. In this method, the Service Binding Operator detects the binding data from resources owned by the backing service CR.

The following examples show the `detectBindingResources` API option set to `true` in the `ServiceBinding` CR:

Example

```yaml
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
  name: spring-petclinic-detect-all
  namespace: my-petclinic
spec:
detectBindingResources: true
services:
  - group: postgres-operator.crunchydata.com
    version: v1beta1
    kind: PostgresCluster
    name: hippo
    application:
      name: spring-petclinic
      group: apps
      version: v1
      resource: deployments
```

In the previous example, `PostgresCluster` custom service resource owns one or more Kubernetes resources such as route, service, config map, or secret.

The Service Binding Operator automatically detects the binding data exposed on each of the owned resources.

4.6.2. Data model
The data model used in the annotations and OLM descriptors follow specific conventions.

Service binding annotations must use the following convention:

\[
\text{service.binding(</<\text{NAME}>)>?:
  \"<\text{VALUE}>|(\text{path=<JSONPATH\_TEMPLATE>}(,\text{objectType=<OBJECT\_TYPE>})?(,\text{elementType=<ELEMENT\_TYPE>})?(,\text{sourceKey=<SOURCE\_KEY>})?(,\text{sourceValue=<SOURCE\_VALUE>})?)\"
\]

where:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;NAME&gt;</td>
<td>Specifies the name under which the binding value is to be exposed. You can exclude it only when the \text{objectType} parameter is set to \text{Secret} or \text{ConfigMap}.</td>
<td></td>
</tr>
<tr>
<td>&lt;VALUE&gt;</td>
<td>Specifies the constant value exposed when no \text{path} is set.</td>
<td></td>
</tr>
</tbody>
</table>

Although, the data model is the same for custom resource definitions (CRD), custom resource (CR) annotations, and Operator Lifecycle Manager (OLM) descriptors, the syntax for each one differs.

The data model provides the details on the allowed values and semantic for the \text{path}, \text{elementType}, \text{objectType}, \text{sourceKey}, and \text{sourceValue} parameters.

### Table 4.4. Parameters and their descriptions

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>JSONPath template that consists JSONPath expressions enclosed by curly braces {}.</td>
<td>N/A</td>
</tr>
<tr>
<td>elementType</td>
<td>Specifies whether the value of the element referenced in the \text{path} parameter complies with any one of the following types:</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>\begin{itemize}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\item string \end{itemize}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\begin{itemize}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\item sliceOfStrings \end{itemize}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\begin{itemize}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\item sliceOfMaps \end{itemize}</td>
<td></td>
</tr>
<tr>
<td>objectType</td>
<td>Specifies whether the value of the element indicated in the \text{path} parameter refers to a \text{ConfigMap}, \text{Secret}, or plain string in the current namespace.</td>
<td>\text{Secret}, if \text{elementType} is non-string.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Default value</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>sourceKey</td>
<td>Specifies the key in the ConfigMap or Secret resource to be added to the binding secret when collecting the binding data. Note:&lt;br&gt;  - When used in conjunction with elementType=sliceOfMaps, the sourceKey parameter specifies the key in the slice of maps whose value is used as a key in the binding secret.&lt;br&gt;  - Use this optional parameter to expose a specific entry in the referenced Secret or ConfigMap resource as binding data.&lt;br&gt;  - When not specified, all keys and values from the Secret or ConfigMap resource are exposed and are added to the binding secret.</td>
<td>N/A</td>
</tr>
<tr>
<td>sourceValue</td>
<td>Specifies the key in the slice of maps. Note:&lt;br&gt;  - The value of this key is used as the base to generate the value of the entry for the key-value pair to be added to the binding secret.&lt;br&gt;  - In addition, the value of the sourceKey is used as the key of the entry for the key-value pair to be added to the binding secret.&lt;br&gt;  - It is mandatory only if elementType=sliceOfMaps.</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**NOTE**

The sourceKey and sourceValue parameters are applicable only if the element indicated in the path parameter refers to a ConfigMap or Secret resource.

### 4.6.3. RBAC requirements

To expose the backing service binding data using the Service Binding Operator, you require certain Role-based access control (RBAC) permissions. Specify certain verbs under the rules field of the ClusterRole resource to grant the RBAC permissions for the backing service resources. When you define these rules, you allow the Service Binding Operator to read the binding data of the backing service resources throughout the cluster. If the users do not have permissions to read binding data or
modify application resource, the Service Binding Operator prevents such users to bind services to application. Adhering to the RBAC requirements avoids unnecessary permission elevation for the user and prevents access to unauthorized services or applications.

The Service Binding Operator performs requests against the Kubernetes API using a dedicated service account. By default, this account has permissions to bind services to workloads, both represented by the following standard Kubernetes or OpenShift objects:

- Deployments
- DaemonSets
- ReplicaSets
- StatefulSets
- DeploymentConfigs

The Operator service account is bound to an aggregated cluster role, allowing Operator providers or cluster administrators to enable binding custom service resources to workloads. To grant the required permissions within a `ClusterRole`, label it with the `servicebinding.io/controller` flag and set the flag value to `true`. The following example shows how to allow the Service Binding Operator to get, watch, and list the custom resources (CRs) of Crunchy PostgreSQL Operator:

**Example: Enable binding to PostgreSQL database instances provisioned by Crunchy PostgreSQL Operator**

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: postgrescluster-reader
  labels:
    servicebinding.io/controller: "true"
rules:
- apiGroups:
  - postgres-operator.crunchydata.com
  resources:
  - postgresclusters
  verbs:
  - get
  - watch
  - list
  ...
```

This cluster role can be deployed during the installation of the backing service Operator.

### 4.6.4. Categories of exposed binding data

The Service Binding Operator enables you to expose the binding data values from the backing service resources and custom resource definitions (CRDs).

This section provides examples to show how you can use the various categories of exposable binding data. You must modify these examples to suit your work environment and requirements.

#### 4.6.4.1. Exposing a string from a resource
The following example shows how to expose the string from the `metadata.name` field of the `PostgresCluster` custom resource (CR) as a username:

Example

```yaml
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    service.binding/username: path={.metadata.name}
...
```

### 4.6.4.2. Exposing a constant value as the binding item

The following examples show how to expose a constant value from the `PostgresCluster` custom resource (CR):

Example: Exposing a constant value

```yaml
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    "service.binding/type": "postgresql"

1 Binding type to be exposed with the `postgresql` value.
```

### 4.6.4.3. Exposing an entire config map or secret that is referenced from a resource

The following examples show how to expose an entire secret through annotations:

Example: Exposing an entire secret through annotations

```yaml
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    service.binding: 'path={.metadata.name}-pguser-{.metadata.name},objectType=Secret'
```

Example: The referenced secret from the backing service resource

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: hippo-pguser-hippo
```
The following example shows how to expose an entire config map through OLM descriptors:

Example: Exposing an entire config map through OLM descriptors

```yaml
- path: data.dbConfiguration
  x-descriptors:
    - urn:alm:descriptor:io.kubernetes:ConfigMap
    - service.binding
```

This example uses the path attribute with a urn:alm:descriptor:io.kubernetes:ConfigMap entry to indicate that the path points to the ConfigMap service resource.

If you intend to project all the values from a ConfigMap service resource, you must specify it as an attribute in the backing service CR. For example, if the attribute is part of the .spec section, you can create and use a specDescriptors array. Or, if the attribute is part of the .status section, you can create and use a statusDescriptors array.

4.6.4.4. Exposing a specific entry from a config map or secret that is referenced from a resource

The following examples show how to expose a specific entry from a config map through annotations:

Example: Exposing an entry from a config map through annotations

```yaml
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
annotations:
  service.binding: "path={.metadata.name}-config,objectType=ConfigMap,sourceKey=user"
```

Example: The referenced config map from the backing service resource

The binding data should have a key with name as db_timeout and value as 10s:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: hippo-config
data:
  db_timeout: "10s"
  user: "hippo"
```

The following example shows how to expose a specific entry from a config map through OLM descriptors:

Example: Exposing an entry from a config map through OLM descriptors

```yaml
- path: data.dbConfiguration
```
4.6.4.5. Exposing a resource definition value

The following example shows how to expose a resource definition value through annotations:

**Example: Exposing a resource definition value through annotations**

```yaml
apiVersion: postgres-operator.crunchydata.com/v1beta1
group: PostgresCluster
name: hippo
namespace: my-petclinic
annotations:
  service.binding/username: path={.metadata.name}
...
```

The following example shows how to expose a resource definition value through OLM descriptors:

**Example: Exposing a resource definition value through OLM descriptors**

```yaml
- path: data.connectionURL
  x-descriptors:
    - service.binding:uri
```

The previous example uses the `connectionURL` attribute that points to the required resource definition value that is to be projected as `uri`.

If required values are available as attributes of backing service resources, annotating these values using **X-Descriptors** identifies them as the binding data.

4.6.4.6. Exposing entries of a collection with the key and value from each entry

The following example shows how to expose the entries of a collection with the key and value from each entry through annotations:

**Example: Exposing the entries of a collection through annotations**

```yaml
apiVersion: postgres-operator.crunchydata.com/v1beta1
group: PostgresCluster
name: hippo
namespace: my-petclinic
metadata:
  name: hippo
  namespace: my-petclinic
annotations:
  "service.binding/uri": "path=
```
The following example shows how the previous entries of a collection in annotations are projected into the bound application.

**Example: Binding data files**

```
{.status.connections},elementType=sliceOfMaps,sourceKey=type,sourceValue=url
spec:
...
status:
  connections:
    - type: primary
      url: primary.example.com
    - type: secondary
      url: secondary.example.com
    - type: '404'
      url: black-hole.example.com
```

The following example shows how to expose the entries of a collection with the key and value from each entry through OLM descriptors:

**Example: Exposing the entries of a collection through OLM descriptors**

```
/bindings/<binding-name>/uri_primary => primary.example.com
/bindings/<binding-name>/uri_secondary => secondary.example.com
/bindings/<binding-name>/uri_404 => black-hole.example.com
```

The previous example shows how to expose the entries of a collection with the key and value from each entry through OLM descriptors:

**Example: Configuration from a backing service resource**

```
status:
  connections:
    - type: primary
      url: primary.example.com
    - type: secondary
      url: secondary.example.com
    - type: '404'
      url: black-hole.example.com
```

The previous example helps you to project all those values with keys such as `primary`, `secondary`, and so on.

**4.6.4.7. Exposing items of a collection with one key per item**

The following example shows how to expose the items of a collection with one key per item through annotations:

**Example: Exposing the items of a collection through annotations**

```
{.status.connections},elementType=sliceOfMaps,sourceKey=type,sourceValue=url
spec:
...
status:
  connections:
    - type: primary
      url: primary.example.com
    - type: secondary
      url: secondary.example.com
    - type: '404'
      url: black-hole.example.com
```
The following example shows how the previous items of a collection in annotations are projected into the bound application.

**Example: Binding data files**

```yaml
/bindings/<binding-name>/tags_0 => knowledge
/bindings/<binding-name>/tags_1 => is
/bindings/<binding-name>/tags_2 => power
```

The following example shows how to expose the items of a collection with one key per item through OLM descriptors:

**Example: Exposing the items of a collection through OLM descriptors**

- path: spec.tags
  x-descriptors:
  - service.binding/tags:elementType=sliceOfStrings

The previous example uses the path attribute with an X-Descriptors update for the required items of a collection.

**Example: Configuration from a backing service resource**

```yaml
spec:
  tags:
  - knowledge
  - is
  - power
```

4.6.4.8. Exposing values of collection entries with one key per entry value

The following example shows how to expose the values of collection entries with one key per entry value through annotations:

**Example: Exposing the values of collection entries through annotations**

```yaml
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
```
The following example shows how the previous values of a collection in annotations are projected into the bound application.

**Example: Binding data files**

```
/bindings/<binding-name>/url_0 => primary.example.com
/bindings/<binding-name>/url_1 => secondary.example.com
/bindings/<binding-name>/url_2 => black-hole.example.com
```

The following example shows how to expose the values of collection entries with one key per entry value through OLM descriptors:

**Example: Exposing the values of collection entries through OLM descriptors**

```
- path: bootstrap
  x-descriptors:
    - service.binding:endpoints:elementType=sliceOfStrings:sourceValue=url
```

4.6.5. Additional resources

- OLM Descriptor Reference.
- Defining cluster service versions (CSVs).
- Projecting binding data.

4.7. PROJECTING BINDING DATA

This section provides information on how you can consume the binding data.

4.7.1. Consumption of binding data

After the backing service exposes the binding data, for a workload to access and consume this data, you must project it into the workload from a backing service. Service Binding Operator automatically projects this set of data into the workload in the following methods:

1. By default, as files.

2. As environment variables, after you configure the `.spec.bindAsFiles` parameter from the `ServiceBinding` resource.
4.7.2. Configuration of the directory path to project the binding data inside workload container

By default, Service Binding Operator mounts the binding data as files at a specific directory in your workload resource. You can configure the directory path using the `SERVICE_BINDING_ROOT` environment variable setup in the container where your workload runs.

Example: Binding data mounted as files

```
$SERVICE_BINDING_ROOT
├── account-database
│   ├── type
│   └── provider
│       ├── uri
│       └── username
│           └── password
└── transaction-event-stream
    ├── type
    ├── connection-count
    ├── uri
    ├── certificates
    └── private-key
```

1. Root directory.
2. Directory that stores the binding data.
3. Mandatory identifier that identifies the type of the binding data projected into the corresponding directory.
4. Optional: Identifier to identify the provider so that the application can identify the type of backing service it can connect to.

To consume the binding data as environment variables, use the built-in language feature of your programming language of choice that can read environment variables.

Example: Python client usage

```python
import os
username = os.getenv("USERNAME")
password = os.getenv("PASSWORD")
```

4.7.2.1. Computation of the final path for projecting the binding data as files

The following table summarizes the configuration of how the final path for the binding data projection is computed when files are mounted at a specific directory:

<table>
<thead>
<tr>
<th><code>SERVICE_BINDING_ROOT</code></th>
<th>Final path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not available</td>
<td><code>/bindings/&lt;ServiceBinding_ResourceName&gt;</code></td>
</tr>
</tbody>
</table>
In the previous table, the `<ServiceBinding_ResourceName>` entry specifies the name of the `ServiceBinding` resource that you configure in the `.metadata.name` section of the custom resource (CR).

To access and consume the binding data within the existing `SERVICE_BINDING_ROOT` environment variable, use the built-in language feature of your programming language of choice that can read environment variables.

**Example: Python client usage**

```python
from pyservicebinding import binding
try:
    sb = binding.ServiceBinding()
except binding.ServiceBindingRootMissingError as msg:
    # log the error message and retry/exit
    print("SERVICE_BINDING_ROOT env var not set")
    sb = binding.ServiceBinding()
    bindings_list = sb.bindings("postgresql")
```

In the previous example, the `bindings_list` variable contains the binding data for the `postgresql` database service type.

### 4.7.3. Projecting the binding data

Depending on your workload requirements and environment, you can choose to project the binding data either as files or environment variables.

**Prerequisites**

- You understand the following concepts:
  - Environment and requirements of your workload, and how it works with the provided services.
  - Consumption of the binding data in your workload resource.
  - Configuration of how the final path for data projection is computed for the default method.
- The binding data is exposed from the backing service.

**Procedure**

1. To project the binding data as files, determine the destination folder by ensuring that the existing `SERVICE_BINDING_ROOT` environment variable is present in the container where your workload runs.

2. To project the binding data as environment variables, set the value for the `.spec.bindAsFiles` parameter to `false` from the `ServiceBinding` resource in the custom resource (CR).
4.7.4. Additional resources

- Exposing binding data from a service.
- Using the projected binding data in the source code of the application.

4.8. BINDING WORKLOADS USING SERVICE BINDING OPERATOR

Application developers must bind a workload to one or more backing services by using a binding secret. This secret is generated for the purpose of storing information to be consumed by the workload.

As an example, consider that the service you want to connect to is already exposing the binding data. In this case, you would also need a workload to be used along with the `ServiceBinding` custom resource (CR). By using this `ServiceBinding` CR, the workload sends a binding request with the details of the services to bind with.

Example of `ServiceBinding` CR

```yaml
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
  name: spring-petclinic-pgcluster
  namespace: my-petclinic
spec:
services:
  - group: postgres-operator.crunchydata.com
    version: v1beta1
    kind: PostgresCluster
    name: hippo
    application:
      name: spring-petclinic
      group: apps
      version: v1
      resource: deployments
```

1. Specifies a list of service resources.
2. The sample application that points to a Deployment or any other similar resource with an embedded PodSpec.

As shown in the previous example, you can also directly use a `ConfigMap` or a `Secret` itself as a service resource to be used as a source of binding data.

4.8.1. Naming strategies

Naming strategies are available only for the `binding.operators.coreos.com` API group.

Naming strategies use Go templates to help you define custom binding names through the service binding request. Naming strategies apply for all attributes including the mappings in the `ServiceBinding` custom resource (CR).

A backing service projects the binding names as files or environment variables into the workload. If a workload expects the projected binding names in a particular format, but the binding names to be projected from the backing service are not available in that format, then you can change the binding
names using naming strategies.

**Predefined post-processing functions**

While using naming strategies, depending on the expectations or requirements of your workload, you can use the following predefined post-processing functions in any combination to convert the character strings:

- **upper**: Converts the character strings into capital or uppercase letters.
- **lower**: Converts the character strings into lowercase letters.
- **title**: Converts the character strings where the first letter of each word is capitalized except for certain minor words.

**Predefined naming strategies**

Binding names declared through annotations or Operator Lifecycle Manager (OLM) descriptors are processed for their name change before their projection into the workload according to the following predefined naming strategies:

- **none**: When applied, there are no changes in the binding names.

  **Example**

  After the template compilation, the binding names take the `{{ .name }}` form.

  ```
  host: hippo-pgbouncer
  port: 5432
  ```

- **upper**: Applied when no **namingStrategy** is defined. When applied, converts all the character strings of the binding name key into capital or uppercase letters.

  **Example**

  After the template compilation, the binding names take the `{{ .service.kind | upper }}_{{ .name | upper }}` form.

  ```
  DATABASE_HOST: hippo-pgbouncer
  DATABASE_PORT: 5432
  ```

If your workload requires a different format, you can define a custom naming strategy and change the binding name using a prefix and a separator, for example, **PORT_DATABASE**.

**NOTE**

- When the binding names are projected as files, by default the predefined **none** naming strategy is applied, and the binding names do not change.

- When the binding names are projected as environment variables and no **namingStrategy** is defined, by default the predefined **uppercase** naming strategy is applied.

- You can override the predefined naming strategies by defining custom naming strategies using different combinations of custom binding names and predefined post-processing functions.
4.8.2. Advanced binding options

You can define the ServiceBinding custom resource (CR) to use the following advanced binding options:

- Changing binding names: This option is available only for the binding.operators.coreos.com API group.
- Composing custom binding data: This option is available only for the binding.operators.coreos.com API group.
- Binding workloads using label selectors: This option is available for both the binding.operators.coreos.com and servicebinding.io API groups.

4.8.2.1. Changing the binding names before projecting them into the workload

You can specify the rules to change the binding names in the .spec.namingStrategy attribute of the ServiceBinding CR. For example, consider a Spring PetClinic sample application that connects to the PostgreSQL database. In this case, the PostgreSQL database service exposes the host and port fields of the database to use for binding. The Spring PetClinic sample application can access this exposed binding data through the binding names.

Example: Spring PetClinic sample application in the ServiceBinding CR

```yaml
... 
  application:
    name: spring-petclinic
    group: apps
    version: v1
    resource: deployments 
...
```

Example: PostgreSQL database service in the ServiceBinding CR

```yaml
... 
  services:
    - group: postgres-operator.crunchydata.com
      version: v1beta1
      kind: PostgresCluster
      name: hippo 
...
```

If namingStrategy is not defined and the binding names are projected as environment variables, then the host: hippo-pgbouncer value in the backing service and the projected environment variable would appear as shown in the following example:

Example

```bash
    DATABASE_HOST: hippo-pgbouncer
```

where:
### DATABASE
Specifies the **kind** backend service.

### HOST
Specifies the binding name.

After applying the `POSTGRESQL_{{ .service.kind | upper }}_{{ .name | upper }}_ENV` naming strategy, the list of custom binding names prepared by the service binding request appears as shown in the following example:

#### Example

```plaintext
POSTGRESQL_DATABASE_HOST_ENV: hippo-pgbouncer
POSTGRESQL_DATABASE_PORT_ENV: 5432
```

The following items describe the expressions defined in the `POSTGRESQL_{{ .service.kind | upper }}_{{ .name | upper }}_ENV` naming strategy:

- **.name**: Refers to the binding name exposed by the backing service. In the previous example, the binding names are `HOST` and `PORT`.
- **.service.kind**: Refers to the kind of service resource whose binding names are changed with the naming strategy.
- **upper**: String function used to post-process the character string while compiling the Go template string.
- **POSTGRESQL**: Prefix of the custom binding name.
- **ENV**: Suffix of the custom binding name.

Similar to the previous example, you can define the string templates in `namingStrategy` to define how each key of the binding names should be prepared by the service binding request.

#### 4.8.2.2. Composing custom binding data

As an application developer, you can compose custom binding data under the following circumstances:

- The backing service does not expose binding data.
- The values exposed are not available in the required format as expected by the workload.

For example, consider a case where the backing service CR exposes the host, port, and database user as binding data, but the workload requires that the binding data be consumed as a connection string. You can compose custom binding data using attributes in the Kubernetes resource representing the backing service.

#### Example

```yaml
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
  name: spring-petclinic-pgcluster
  namespace: my-petclinic
spec:
```
services:
  - group: postgres-operator.crunchydata.com
    version: v1beta1
    kind: PostgresCluster
    name: hippo
    id: postgresDB
    - group: ""
      version: v1
      kind: Secret
      name: hippo-pguser-hippo
      id: postgresSecret
      application:
        name: spring-petclinic
        group: apps
        version: v1
        resource: deployments
      mappings:
        ## From the database service
        - name: JDBC_URL
          value: 'jdbc:postgresql://{{ .postgresDB.metadata.annotations.proxy }}:{{ .postgresDB.spec.port }}/ {{ .postgresDB.metadata.name }}'
        ## From both the services!
        - name: CREDENTIALS
          value: '{{ .postgresDB.metadata.name }}{{ translationService.postgresSecret.data.password }}'
        ## Generate JSON
        - name: DB_JSON
          value: '{{ json .postgresDB.status }}'

1 Name of the backing service resource.
2 Optional identifier.
3 The JSON name that the Service Binding Operator generates. The Service Binding Operator projects this JSON name as the name of a file or environment variable.
4 The JSON value that the Service Binding Operator generates. The Service Binding Operator projects this JSON value as a file or environment variable. The JSON value contains the attributes from your specified field of the backing service custom resource.

4.8.2.3. Binding workloads using a label selector

You can use a label selector to specify the workload to bind. If you declare a service binding using the label selectors to pick up workloads, the Service Binding Operator periodically attempts to find and bind new workloads that match the given label selector.

For example, as a cluster administrator, you can bind a service to every Deployment in a namespace with the environment: production label by setting an appropriate labelSelector field in the ServiceBinding CR. This enables the Service Binding Operator to bind each of these workloads with one ServiceBinding CR.

Example ServiceBinding CR in the binding.operators.coreos.com/v1alpha1 API

```yaml
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
```
metadata:
  name: multi-application-binding
namespace: service-binding-demo
spec:
  application:
    labelSelector: 
      matchLabels:
        environment: production
group: apps
version: v1
resource: deployments
services:
  group: ""
  version: v1
  kind: Secret
  name: super-secret-data

1 Specifies the workload that is being bound.

Example ServiceBinding CR in the servicebinding.io API

apiVersion: servicebindings.io/v1alpha3
crDName: ServiceBinding
metadata:
  name: multi-application-binding
namespace: service-binding-demo
spec:
  workload:
    selector: 
      matchLabels:
        environment: production
apiVersion: app/v1
kind: Deployment
service:
  apiVersion: v1
  kind: Secret
  name: super-secret-data

1 Specifies the workload that is being bound.

**IMPORTANT**

If you define the following pairs of fields, Service Binding Operator refuses the binding operation and generates an error:

- The **name** and **labelSelector** fields in the `binding.operators.coreos.com/v1alpha1` API.
- The **name** and **selector** fields in the `servicebinding.io` API (Spec API).

4.8.3. Binding secondary workloads that are not compliant with PodSpec
A typical scenario in service binding involves configuring the backing service, the workload (Deployment), and Service Binding Operator. Consider a scenario that involves a secondary workload (which can also be an application Operator) that is not compliant with PodSpec and is between the primary workload (Deployment) and Service Binding Operator.

For such secondary workload resources, the location of the container path is arbitrary. For service binding, if the secondary workload in a CR is not compliant with the PodSpec, you must specify the location of the container path. Doing so projects the binding data into the container path specified in the secondary workload of the ServiceBinding custom resource (CR), for example, when you do not want the binding data inside a pod.

In Service Binding Operator, you can configure the path of where containers or secrets reside within a workload and bind these paths at a custom location.

4.8.3.1. Configuring the custom location of the container path

This custom location is available for the binding.operators.coreos.com API group when Service Binding Operator projects the binding data as environment variables.

Consider a secondary workload CR, which is not compliant with the PodSpec and has containers located at the spec.containers path:

Example: Secondary workload CR

```yaml
apiVersion: "operator.sbo.com/v1"
kind: SecondaryWorkload
metadata:
  name: secondary-workload
spec:
  containers:
  - name: hello-world
    image: quay.io/baijum/secondary-workload:latest
    ports:
    - containerPort: 8080
```

Procedure

- Configure the spec.containers path by specifying a value in the ServiceBinding CR and bind this path to a spec.application.bindingPath.containersPath custom location:

Example: ServiceBinding CR with the spec.containers path in a custom location

```yaml
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
  name: spring-petclinic-pgcluster
spec:
  services:
  - group: postgres-operator.crunchydata.com
    version: v1beta1
    kind: PostgresCluster
    name: hippo
    id: postgresDB
  - group: ""
    version: v1
```
The sample application that points to a Deployment or any other similar resource with an embedded PodSpec.

The secondary workload, which is not compliant with the PodSpec.

The custom location of the container path.

After you specify the location of the container path, Service Binding Operator generates the binding data, which becomes available in the container path specified in the secondary workload of the ServiceBinding CR.

The following example shows the `spec.containers` path with the `envFrom` and `secretRef` fields:

**Example: Secondary workload CR with the `envFrom` and `secretRef` fields**

```yaml
apiVersion: "operator.sbo.com/v1"
kind: SecondaryWorkload
metadata:
  name: secondary-workload
spec:
  containers:
  - env:
    - name: ServiceBindingOperatorChangeTriggerEnvVar
      value: "31793"
    envFrom:
      - secretRef:
        name: secret-resource-name
    image: quay.io/baijum/secondary-workload:latest
    name: hello-world
    ports:
    - containerPort: 8080
  resources: {} 
```

1. Unique array of containers with values generated by the Service Binding Operator. These values are based on the backing service CR.

2. Name of the **Secret** resource generated by the Service Binding Operator.
4.8.3.2. Configuring the custom location of the secret path

This custom location is available for the `binding.operators.coreos.com` API group when Service Binding Operator projects the binding data as environment variables.

Consider a secondary workload CR, which is not compliant with the PodSpec, with only the secret at the `spec.secret` path:

**Example: Secondary workload CR**

```yaml
apiVersion: "operator.sbo.com/v1"
kind: SecondaryWorkload
metadata:
  name: secondary-workload
spec:
  secret: ""
```

**Procedure**

- Configure the `spec.secret` path by specifying a value in the `ServiceBinding` CR and bind this path at a `spec.application.bindingPath.secretPath` custom location:

**Example: ServiceBinding CR with the spec.secret path in a custom location**

```yaml
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
  name: spring-petclinic-pgcluster
spec:
  ...  
  application: 1
    name: secondary-workload
    group: operator.sbo.com
    version: v1
    resource: secondaryworkloads
    bindingPath:
      secretPath: spec.secret 2
  ...
```

1. The secondary workload, which is not compliant with the PodSpec.
2. The custom location of the secret path that contains the name of the `Secret` resource.

After you specify the location of the secret path, Service Binding Operator generates the binding data, which becomes available in the secret path specified in the secondary workload of the `ServiceBinding` CR.

The following example shows the `spec.secret` path with the `binding-request` value:

**Example: Secondary workload CR with the binding-request value**

```yaml
...
apiVersion: "operator.sbo.com/v1"
```
4.8.3.3. Workload resource mapping

**NOTE**

- Workload resource mapping is available for the secondary workloads of the `ServiceBinding` custom resource (CR) for both the API groups: `binding.operators.coreos.com` and `servicebinding.io`.

- You must define `ClusterWorkloadResourceMapping` resources only under the `servicebinding.io` API group. However, the `ClusterWorkloadResourceMapping` resources interact with `ServiceBinding` resources under both the `binding.operators.coreos.com` and `servicebinding.io` API groups.

If you cannot configure custom path locations by using the configuration method for container path, you can define exactly where binding data needs to be projected. Specify where to project the binding data for a given workload kind by defining the `ClusterWorkloadResourceMapping` resources in the `servicebinding.io` API group.

The following example shows how to define a mapping for the `CronJob.batch/v1` resources.

**Example: Mapping for CronJob.batch/v1 resources**

```yaml
apiVersion: servicebinding.io/v1alpha3
kind: ClusterWorkloadResourceMapping
metadata:
  name: cronjobs.batch
spec:
  versions:
    - version: "v1"
      annotations:
        .spec.jobTemplate.spec.template.metadata.annotations
      containers:
        - path: .spec.jobTemplate.spec.template.spec.containers[*]
          name: .name
          env:
            .env
          volumeMounts: .volumeMounts
          volumes: .spec.jobTemplate.spec.template.spec.volumes
```

1. Name of the `ClusterWorkloadResourceMapping` resource, which must be qualified as the `plural.group` of the mapped workload resource.

2. Version of the resource that is being mapped. Any version that is not specified can be matched.
Version of the resource that is being mapped. Any version that is not specified can be matched with the "*" wildcard.

3 Optional: Identifier of the .annotations field in a pod, specified with a fixed JSONPath. The default value is .spec.template.spec.annotations.

4 Identifier of the .containers and .initContainers fields in a pod, specified with a JSONPath. If no entries under the containers field are defined, the Service Binding Operator defaults to two paths: .spec.template.spec.containers[*] and .spec.template.spec.initContainers[*], with all other fields set as their default. However, if you specify an entry, then you must define the .path field.

5 Optional: Identifier of the .name field in a container, specified with a fixed JSONPath. The default value is .name.

6 Optional: Identifier of the .env field in a container, specified with a fixed JSONPath. The default value is .env.

7 Optional: Identifier of the .volumeMounts field in a container, specified with a fixed JSONPath. The default value is .volumeMounts.

8 Optional: Identifier of the .volumes field in a pod, specified with a fixed JSONPath. The default value is .spec.template.spec.volumes.

IMPORTANT

- In this context, a fixed JSONPath is a subset of the JSONPath grammar that accepts only the following operations:
  - Field lookup: .spec.template
  - Array indexing: .spec['template']

All other operations are not accepted.

- Most of these fields are optional. When they are not specified, the Service Binding Operator assumes defaults compatible with PodSpec resources.

- The Service Binding Operator requires that each of these fields is structurally equivalent to the corresponding field in a pod deployment. For example, the contents of the .env field in a workload resource must be able to accept the same structure of data that the .env field in a Pod resource would. Otherwise, projecting binding data into such a workload might result in unexpected behavior from the Service Binding Operator.

Behavior specific to the binding.operators.coreos.com API group

You can expect the following behaviors when ClusterWorkloadResourceMapping resources interact with ServiceBinding resources under the binding.operators.coreos.com API group:

- If a ServiceBinding resource with the bindAsFiles: false flag value is created together with one of these mappings, then environment variables are projected into the .envFrom field underneath each .path field specified in the corresponding ClusterWorkloadResourceMapping resource.
As a cluster administrator, you can specify both a `ClusterWorkloadResourceMapping` resource and the `spec.application.bindingPath.containersPath` field in a `ServiceBinding.bindings.coreos.com` resource for binding purposes. The Service Binding Operator attempts to project binding data into the locations specified in both a `ClusterWorkloadResourceMapping` resource and the `spec.application.bindingPath.containersPath` field. This behavior is equivalent to adding a container entry to the corresponding `ClusterWorkloadResourceMapping` resource with the path: `$containersPath` attribute, with all other values taking their default value.

4.8.4. Unbinding workloads from a backing service

You can unbind a workload from a backing service by using the `oc` tool.

- To unbind a workload from a backing service, delete the `ServiceBinding` custom resource (CR) linked to it:

  ```
  $ oc delete ServiceBinding <.metadata.name>
  ```

  **Example**

  ```
  $ oc delete ServiceBinding spring-petclinic-pgcluster
  ```

  where:

  | spring-petclinic-pgcluster | Specifies the name of the `ServiceBinding` CR. |

4.8.5. Additional resources

- Binding a workload together with a backing service.
- Connecting the Spring PetClinic sample application to the PostgreSQL database service.
- Creating custom resources from a file
- Example schema of the `ClusterWorkloadResourceMapping` resource.

4.9. CONNECTING AN APPLICATION TO A SERVICE USING THE DEVELOPER PERSPECTIVE

In addition to grouping multiple components within an application, you can also use the `Topology` view to connect components with each other. You can either use a binding connector or a visual one to connect components.

A binding connection between the components can be established only if the target node is an Operator-backed service. This is indicated by the `Create a binding connector` tool-tip which appears when you drag an arrow to such a target node. When an application is connected to a service using a binding connector a `ServiceBinding` resource is created. Then, the Service Binding Operator controller projects the necessary binding data into the application deployment. After the request is successful, the application is redeployed establishing an interaction between the connected components.

A visual connector establishes only a visual connection between the components, depicting an intent to
connect. No interaction between the components is established. If the target node is not an Operator-
backed service the Create a visual connector tool-tip is displayed when you drag an arrow to a target
node.

4.9.1. Creating a visual connection between components

You can depict an intent to connect application components using the visual connector.

This procedure walks you through an example of creating a visual connection between a PostgreSQL
Database service and a Spring PetClinic sample application.

Prerequisites

- You have created and deployed a Spring PetClinic sample application using the Developer
  perspective.

- You have created and deployed a Crunchy PostgreSQL database instance using the Developer
  perspective. This instance has the following four components: hippo-backup, hippo-instance,
  hippo-repo-host, and hippo-pgbouncer.

Procedure

1. Hover over the Spring PetClinic sample application to see a dangling arrow on the node.

2. Click and drag the arrow towards the hippo-pgbouncer deployment to connect the Spring
   PetClinic sample application with it.

3. Click the spring-petclinic deployment to see the Overview panel. Under the Details tab, click
   the edit icon in the Annotations section to see the Key = app.openshift.io/connects-to and
   Value = ["apiVersion":"apps/v1","kind":"Deployment","name":"hippo-pgbouncer"]
   annotation added to the deployment.

Similarly you can create other applications and components and establish visual connections between
them.
4.9.2. Creating a binding connection between components

You can establish a binding connection with Operator-backed components.

This procedure walks through an example of creating a binding connection between a PostgreSQL Database service and a Spring PetClinic sample application. To create a binding connection with a service that is backed by the PostgreSQL Database Operator, you must first add the Red Hat-provided PostgreSQL Database Operator to the OperatorHub, and then install the Operator. The PostgreSQL Database Operator then creates and manages the Database resource, which exposes the binding information in secrets, config maps, status, and spec attributes.

Prerequisites

- You have created and deployed a Spring PetClinic sample application using the Developer perspective.
- You have installed the Service Binding Operator from the OperatorHub.
- You have installed the Crunchy Postgres for Kubernetes Operator from the OperatorHub using v5 Update channel.
- You have created and deployed a Crunchy PostgreSQL database instance using the Developer perspective. This instance has the following four components: hippo-backup, hippo-instance, hippo-repo-host, and hippo-pgbouncer.
Procedure

1. Switch to the Developer perspective and ensure that you are in the appropriate project, for example, my-petclinic. In the Topology view, hover over the Spring PetClinic sample application to see a dangling arrow on the node.

2. Click and drag the arrow towards the hippo database in the Postgres Cluster to make a binding connection with the Spring PetClinic sample application. A Create Service Binding dialog box appears.

   a. Enter the name and click Create.

   ![Figure 4.4. Service Binding modal](image)

Alternatively, in the +Add view, click the YAML option to see the Import YAML screen. Use the YAML editor and add the ServiceBinding resource:

```yaml
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
  name: spring-petclinic-pgcluster
  namespace: my-petclinic
spec:
  services:
    - group: postgres-operator.crunchydata.com
      version: v1beta1
      kind: PostgresCluster
      name: hippo
      application:
        name: spring-petclinic
        group: apps
        version: v1
        resource: deployments
```

A service binding request is created and the Service Binding Operator controller projects the database service connection information into the application deployment as files using a volume mount. After the request is successful, the application is redeployed and the connection is established.
NOTE

You can also use the context menu by dragging the dangling arrow to add and create a binding connection to an operator-backed service.

Figure 4.6. Context menu to create binding connection

4.9.3. Additional resources
- Getting started with service binding.
- Known bindable Operators.
CHAPTER 5. WORKING WITH HELM CHARTS

5.1. UNDERSTANDING HELM

Helm is a software package manager that simplifies deployment of applications and services to OpenShift Container Platform clusters.

Helm uses a packaging format called charts. A Helm chart is a collection of files that describes the OpenShift Container Platform resources.

A running instance of the chart in a cluster is called a release. A new release is created every time a chart is installed on the cluster.

Each time a chart is installed, or a release is upgraded or rolled back, an incremental revision is created.

5.1.1. Key features

Helm provides the ability to:

- Search through a large collection of charts stored in the chart repository.
- Modify existing charts.
- Create your own charts with OpenShift Container Platform or Kubernetes resources.
- Package and share your applications as charts.

NOTE

In OpenShift Container Platform 4.10, Helm is disabled for the Multicluster Console (Technology Preview).

5.1.2. Red Hat Certification of Helm charts for OpenShift

You can choose to verify and certify your Helm charts by Red Hat for all the components you will be deploying on the Red Hat OpenShift Container Platform. Charts go through an automated Red Hat OpenShift certification workflow that guarantees security compliance as well as best integration and experience with the platform. Certification assures the integrity of the chart and ensures that the Helm chart works seamlessly on Red Hat OpenShift clusters.

5.1.3. Additional resources

- For more information on how to certify your Helm charts as a Red Hat partner, see Red Hat Certification of Helm charts for OpenShift.
- For more information on OpenShift and Container certification guides for Red Hat partners, see Partner Guide for OpenShift and Container Certification.
- For a list of the charts, see the Red Hat Helm index file.
- You can view the available charts at the Red Hat Marketplace. For more information, see Using the Red Hat Marketplace.
5.2. INSTALLING HELM

The following section describes how to install Helm on different platforms using the CLI.

You can also find the URL to the latest binaries from the OpenShift Container Platform web console by clicking the ? icon in the upper-right corner and selecting Command Line Tools.

Prerequisites

- You have installed Go, version 1.13 or higher.

5.2.1. On Linux

1. Download the Helm binary and add it to your path:
   - Linux (x86_64, amd64)
     
     ```bash
     # chmod +x /usr/local/bin/helm
     ```
   - Linux on IBM Z and LinuxONE (s390x)
     
     ```bash
     ```
   - Linux on IBM Power (ppc64le)
     
     ```bash
     # curl -L https://mirror.openshift.com/pub/openshift-v4/clients/helm/latest/helm-linux-ppc64le -o /usr/local/bin/helm
     ```

2. Make the binary file executable:

   ```bash
   # chmod +x /usr/local/bin/helm
   ```

3. Check the installed version:

   ```bash
   $ helm version
   ```

   Example output

   ```json
   version.BuildInfo{Version:"v3.0",
   GitCommit:"b31719aab7963acf4887a1c1e6d5e53378e34d93", GitTreeState:"clean",
   GoVersion:"go1.13.4"}
   ```

5.2.2. On Windows 7/8

1. Download the latest .exe file and put in a directory of your preference.

2. Right click Start and click Control Panel.

3. Select System and Security and then click System.
4. From the menu on the left, select **Advanced systems settings** and click **Environment Variables** at the bottom.

5. Select **Path** from the **Variable** section and click **Edit**.

6. Click **New** and type the path to the folder with the `.exe` file into the field or click **Browse** and select the directory, and click **OK**.

5.2.3. **On Windows 10**

1. Download the latest `.exe` file and put in a directory of your preference.

2. Click **Search** and type `env` or `environment`.

3. Select **Edit environment variables for your account**

4. Select **Path** from the **Variable** section and click **Edit**.

5. Click **New** and type the path to the directory with the `exe` file into the field or click **Browse** and select the directory, and click **OK**.

5.2.4. **On MacOS**

1. Download the Helm binary and add it to your path:

   ```bash
   # chmod +x /usr/local/bin/helm
   ```

2. Make the binary file executable:

   ```bash
   # chmod +x /usr/local/bin/helm
   ```

3. Check the installed version:

   ```bash
   $ helm version
   ```

   **Example output**

   ```bash
   version.BuildInfo{Version:"v3.0", 
   GitCommit:"b31719aab7963acf4887a1c1e6d5e53378e34d93", GitTreeState:"clean", 
   GoVersion:"go1.13.4"}
   ```

5.3. **CONFIGURING CUSTOM HELM CHART REPOSITORIES**

You can install Helm charts on an OpenShift Container Platform cluster using the following methods:

- The CLI.
- The **Developer** perspective of the web console.

The **Developer Catalog**, in the **Developer** perspective of the web console, displays the Helm charts available in the cluster. By default, it lists the Helm charts from the Red Hat OpenShift Helm chart repository. For a list of the charts, see the Red Hat [Helm index file](#).
As a cluster administrator, you can add multiple cluster-scoped and namespace-scoped Helm chart repositories, separate from the default cluster-scoped Helm repository, and display the Helm charts from these repositories in the Developer Catalog.

As a regular user or project member with the appropriate role-based access control (RBAC) permissions, you can add multiple namespace-scoped Helm chart repositories, apart from the default cluster-scoped Helm repository, and display the Helm charts from these repositories in the Developer Catalog.

### 5.3.1. Installing a Helm chart on an OpenShift Container Platform cluster

#### Prerequisites

- You have a running OpenShift Container Platform cluster and you have logged into it.
- You have installed Helm.

#### Procedure

1. Create a new project:
   
   ```bash
   $ oc new-project vault
   ```

2. Add a repository of Helm charts to your local Helm client:
   
   ```bash
   $ helm repo add openshift-helm-charts https://charts.openshift.io/
   ```
   **Example output**
   
   "openshift-helm-charts" has been added to your repositories

3. Update the repository:
   
   ```bash
   $ helm repo update
   ```

4. Install an example HashiCorp Vault:
   
   ```bash
   $ helm install example-vault openshift-helm-charts/hashicorp-vault
   ```
   **Example output**
   
   NAME: example-vault
   LAST DEPLOYED: Fri Mar 11 12:02:12 2022
   NAMESPACE: vault
   STATUS: deployed
   REVISION: 1
   NOTES:
   Thank you for installing HashiCorp Vault!

5. Verify that the chart has installed successfully:
   
   ```bash
   $ helm list
   ```
5.3.2. Installing Helm charts using the Developer perspective

You can use either the Developer perspective in the web console or the CLI to select and install a chart from the Helm charts listed in the Developer Catalog. You can create Helm releases by installing Helm charts and see them in the Developer perspective of the web console.

Prerequisites

- You have logged in to the web console and have switched to the Developer perspective.

Procedure

To create Helm releases from the Helm charts provided in the Developer Catalog:

1. In the Developer perspective, navigate to the +Add view and select a project. Then click Helm Chart option to see all the Helm Charts in the Developer Catalog.

2. Select a chart and read the description, README, and other details about the chart.

3. Click Install Helm Chart

Figure 5.1. Helm charts in developer catalog

4. In the Install Helm Chart page:
   a. Enter a unique name for the release in the Release Name field.
   b. Select the required chart version from the Chart Version drop-down list.
   c. Configure your Helm chart by using the Form View or the YAML View.
NOTE
Where available, you can switch between the **YAML View** and **Form View**. The data is persisted when switching between the views.

d. Click **Install** to create a Helm release. You will be redirected to the **Topology** view where the release is displayed. If the Helm chart has release notes, the chart is pre-selected and the right panel displays the release notes for that release.

You can upgrade, rollback, or uninstall a Helm release by using the **Actions** button on the side panel or by right-clicking a Helm release.

### 5.3.3. Using Helm in the web terminal

You can use Helm by initializing the web terminal in the **Developer** perspective of the web console. For more information, see [Using the web terminal](#).

### 5.3.4. Creating a custom Helm chart on OpenShift Container Platform

**Procedure**

1. Create a new project:

   ```
   $ oc new-project nodejs-ex-k
   ```

2. Download an example Node.js chart that contains OpenShift Container Platform objects:

   ```
   $ git clone https://github.com/redhat-developer/redhat-helm-charts
   ```

3. Go to the directory with the sample chart:

   ```
   $ cd redhat-helm-charts/alpha/nodejs-ex-k/
   ```

4. Edit the **Chart.yaml** file and add a description of your chart:

   ```
   apiVersion: v2
   name: nodejs-ex-k
   description: A Helm chart for OpenShift
   icon: https://static.redhat.com/libs/redhat/brand-assets/latest/corp/logo.svg
   version: 0.2.1
   ```

   - **1** The chart API version. It should be **v2** for Helm charts that require at least Helm 3.
   - **2** The name of your chart.
   - **3** The description of your chart.
   - **4** The URL to an image to be used as an icon.
   - **5** The Version of your chart as per the Semantic Versioning (SemVer) 2.0.0 Specification.

5. Verify that the chart is formatted properly:
$ helm lint

Example output

[INFO] Chart.yaml: icon is recommended
1 chart(s) linted, 0 chart(s) failed

6. Navigate to the previous directory level:

$ cd ..

7. Install the chart:

$ helm install nodejs-chart nodejs-ex-k

8. Verify that the chart has installed successfully:

$ helm list

Example output

<table>
<thead>
<tr>
<th>NAME</th>
<th>NAMESPACE</th>
<th>REVISION</th>
<th>UPDATED</th>
<th>STATUS</th>
<th>CHART</th>
<th>APP</th>
<th>VERSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodejs-chart</td>
<td>nodejs-ex-k</td>
<td>1</td>
<td>2019-12-05</td>
<td>deployed</td>
<td>nodejs-</td>
<td>0.1.0</td>
<td>1.16.0</td>
</tr>
</tbody>
</table>

5.3.5. Adding custom Helm chart repositories

As a cluster administrator, you can add custom Helm chart repositories to your cluster and enable access to the Helm charts from these repositories in the Developer Catalog.

Procedure

1. To add a new Helm Chart Repository, you must add the Helm Chart Repository custom resource (CR) to your cluster.

Sample Helm Chart Repository CR

```
apiVersion: helm.openshift.io/v1beta1
kind: HelmChartRepository
metadata:
  name: <name>
spec:
  # optional name that might be used by console
  # name: <chart-display-name>
  connectionConfig:
    url: <helm-chart-repository-url>
```

For example, to add an Azure sample chart repository, run:

$ cat <<EOF | oc apply -f -

EOF
2. Navigate to the **Developer Catalog** in the web console to verify that the Helm charts from the chart repository are displayed. For example, use the **Chart repositories** filter to search for a Helm chart from the repository.

![Chart repositories filter](image)

**NOTE**

If a cluster administrator removes all of the chart repositories, then you cannot view the Helm option in the +Add view, **Developer Catalog**, and left navigation panel.

### 5.3.6. Adding namespace-scoped custom Helm chart repositories

The cluster-scoped **HelmChartRepository** custom resource definition (CRD) for Helm repository provides the ability for administrators to add Helm repositories as custom resources. The namespace-scoped **ProjectHelmChartRepository** CRD allows project members with the appropriate role-based access control (RBAC) permissions to create Helm repository resources of their choice but scoped to their namespace. Such project members can see charts from both cluster-scoped and namespace-scoped Helm repository resources.

**NOTE**

- Administrators can limit users from creating namespace-scoped Helm repository resources. By limiting users, administrators have the flexibility to control the RBAC through a namespace role instead of a cluster role. This avoids unnecessary permission elevation for the user and prevents access to unauthorized services or applications.

- The addition of the namespace-scoped Helm repository does not impact the behavior of the existing cluster-scoped Helm repository.

---

```yaml
apiVersion: helm.openshift.io/v1beta1
kind: HelmChartRepository
metadata:
  name: azure-sample-repo
spec:
  name: azure-sample-repo
  connectionConfig:
    url: https://raw.githubusercontent.com/Azure-Samples/helm-charts/master/docs
```

---

OpenShift Container Platform 4.10 Building applications
As a regular user or project member with the appropriate RBAC permissions, you can add custom namespace-scoped Helm chart repositories to your cluster and enable access to the Helm charts from these repositories in the Developer Catalog.

Procedure

1. To add a new namespace-scoped Helm Chart Repository, you must add the Helm Chart Repository custom resource (CR) to your namespace.

   **Sample Namespace-scoped Helm Chart Repository CR**

   ```yaml
   apiVersion: helm.openshift.io/v1beta1
   kind: ProjectHelmChartRepository
   metadata:
     name: <name>
   spec:
     url: https://my.chart-repo.org/stable

     # optional name that might be used by console
     name: <chart-repo-display-name>

     # optional and only needed for UI purposes
     description: <My private chart repo>

     # required: chart repository URL
     connectionConfig:
       url: <helm-chart-repository-url>
   
   $ cat <<EOF | oc apply --namespace my-namespace -f -
   apiVersion: helm.openshift.io/v1beta1
   kind: ProjectHelmChartRepository
   metadata:
     name: azure-sample-repo
   spec:
     name: azure-sample-repo
     connectionConfig:
       url: https://raw.githubusercontent.com/Azure-Samples/helm-charts/master/docs
   EOF
   
   The output verifies that the namespace-scoped Helm Chart Repository CR is created:

   **Example output**

   `projecthelmchartrepository.helm.openshift.io/azure-sample-repo created`

2. Navigate to the Developer Catalog in the web console to verify that the Helm charts from the chart repository are displayed in your my-namespace namespace.

   For example, use the Chart repositories filter to search for a Helm chart from the repository.
Figure 5.3. Chart repositories filter in your namespace

Alternatively, run:

```bash
$ oc get projecthelmchartrepositories --namespace my-namespace
```

Example output

<table>
<thead>
<tr>
<th>NAME</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>azure-sample-repo</td>
<td>1m</td>
</tr>
</tbody>
</table>

**NOTE**

If a cluster administrator or a regular user with appropriate RBAC permissions removes all of the chart repositories in a specific namespace, then you cannot view the Helm option in the +Add view, Developer Catalog, and left navigation panel for that specific namespace.

5.3.7. Creating credentials and CA certificates to add Helm chart repositories

Some Helm chart repositories need credentials and custom certificate authority (CA) certificates to connect to it. You can use the web console as well as the CLI to add credentials and certificates.

**Procedure**

To configure the credentials and certificates, and then add a Helm chart repository using the CLI:

1. In the `openshift-config` namespace, create a **ConfigMap** object with a custom CA certificate in PEM encoded format, and store it under the `ca-bundle.crt` key within the config map:

   ```bash
   $ oc create configmap helm-ca-cert \ 
   --from-file=ca-bundle.crt=/path/to/certs/ca.crt \ 
   -n openshift-config
   
   $ oc get projecthelmchartrepositories --namespace my-namespace
   
   $ oc create configmap helm-ca-cert \ 
   --from-file=ca-bundle.crt=/path/to/certs/ca.crt \ 
   -n openshift-config
   
   $ oc get projecthelmchartrepositories --namespace my-namespace
   
   $ oc create secret tls helm-tls-configs \
   ```

2. In the `openshift-config` namespace, create a **Secret** object to add the client TLS configurations:
Note that the client certificate and key must be in PEM encoded format and stored under the keys `tls.crt` and `tls.key`, respectively.

3. Add the Helm repository as follows:

```bash
$ cat <<EOF | oc apply -f -
apiVersion: helm.openshift.io/v1beta1
kind: HelmChartRepository
metadata:
  name: <helm-repository>
spec:
  name: <helm-repository>
  connectionConfig:
    url: <URL for the Helm repository>
    tlsConfig:
      name: helm-tls-configs
      ca:
        name: helm-ca-cert
EOF
```

The `ConfigMap` and `Secret` are consumed in the HelmChartRepository CR using the `tlsConfig` and `ca` fields. These certificates are used to connect to the Helm repository URL.

4. By default, all authenticated users have access to all configured charts. However, for chart repositories where certificates are needed, you must provide users with read access to the `helm-ca-cert` config map and `helm-tls-configs` secret in the `openshift-config` namespace, as follows:

```bash
$ cat <<EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: openshift-config
  name: helm-chartrepos-tls-conf-viewer
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  resourceNames: ["helm-ca-cert"]
  verbs: ["get"]
- apiGroups: [""]
  resources: ["secrets"]
  resourceNames: ["helm-tls-configs"]
  verbs: ["get"]
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: openshift-config
  name: helm-chartrepos-tls-conf-viewer
subjects:
- kind: Group
  apiGroup: rbac.authorization.k8s.io
  name: openshift-authz-admin
 EOF
```
5.3.8. Filtering Helm Charts by their certification level

You can filter Helm charts based on their certification level in the Developer Catalog.

Procedure

1. In the Developer perspective, navigate to the +Add view and select a project.

2. From the Developer Catalog tile, select the Helm Chart option to see all the Helm charts in the Developer Catalog.

3. Use the filters to the left of the list of Helm charts to filter the required charts:
   - Use the Chart Repositories filter to filter charts provided by Red Hat Certification Charts or OpenShift Helm Charts.
   - Use the Source filter to filter charts sourced from Partners, Community, or Red Hat.
   
   Certified charts are indicated with the (شرع) icon.

   **NOTE**
   
   The Source filter will not be visible when there is only one provider type.

You can now select the required chart and install it.

5.3.9. Disabling Helm Chart repositories

You can disable Helm Charts from a particular Helm Chart Repository in the catalog by setting the disabled property in the HelmChartRepository custom resource to true.

Procedure

- To disable a Helm Chart repository by using CLI, add the disabled: true flag to the custom resource. For example, to remove an Azure sample chart repository, run:

```
$ cat <<EOF | oc apply -f -
apiVersion: helm.openshift.io/v1beta1
kind: HelmChartRepository
metadata:
  name: azure-sample-repo
spec:
  connectionConfig:
    url:https://raw.githubusercontent.com/Azure-Samples/helm-charts/master/docs
disabled: true
EOF
```

- To disable a recently added Helm Chart repository by using Web Console:
1. Go to Custom Resource Definitions and search for the HelmChartRepository custom resource.

2. Go to Instances, find the repository you want to disable, and click its name.

3. Go to the YAML tab, add the disabled: true flag in the spec section, and click Save.

Example

```
spec:
  connectionConfig:
    url: <url-of-the-repository-to-be-disabled>
    disabled: true
```

The repository is now disabled and will not appear in the catalog.

### 5.4. WORKING WITH HELM RELEASES

You can use the Developer perspective in the web console to update, rollback, or uninstall a Helm release.

#### 5.4.1. Prerequisites

- You have logged in to the web console and have switched to the Developer perspective.

#### 5.4.2. Upgrading a Helm release

You can upgrade a Helm release to upgrade to a new chart version or update your release configuration.

**Procedure**

1. In the Topology view, select the Helm release to see the side panel.

2. Click Actions → Upgrade Helm Release.

3. In the Upgrade Helm Release page, select the Chart Version you want to upgrade to, and then click Upgrade to create another Helm release. The Helm Releases page displays the two revisions.

#### 5.4.3. Rolling back a Helm release

If a release fails, you can rollback the Helm release to a previous version.

**Procedure**

To rollback a release using the Helm view:

1. In the Developer perspective, navigate to the Helm view to see the Helm Releases in the namespace.

2. Click the Options menu adjoining the listed release, and select Rollback.
3. In the **Rollback Helm Release** page, select the **Revision** you want to rollback to and click **Rollback**.

4. In the **Helm Releases** page, click on the chart to see the details and resources for that release.

5. Go to the **Revision History** tab to see all the revisions for the chart.

![Helm revision history](image)

6. If required, you can further use the **Options** menu adjoining a particular revision and select the revision to rollback to.

### 5.4.4. Uninstalling a Helm release

**Procedure**

1. In the **Topology** view, right-click the Helm release and select **Uninstall Helm Release**.

2. In the confirmation prompt, enter the name of the chart and click **Uninstall**.
CHAPTER 6. DEPLOYMENTS

6.1. UNDERSTANDING DEPLOYMENT AND DEPLOYMENTCONFIG OBJECTS

The Deployment and DeploymentConfig API objects in OpenShift Container Platform provide two similar but different methods for fine-grained management over common user applications. They are composed of the following separate API objects:

- A DeploymentConfig or Deployment object, either of which describes the desired state of a particular component of the application as a pod template.
- DeploymentConfig objects involve one or more replication controllers, which contain a point-in-time record of the state of a deployment as a pod template. Similarly, Deployment objects involve one or more replica sets, a successor of replication controllers.
- One or more pods, which represent an instance of a particular version of an application.

6.1.1. Building blocks of a deployment

Deployments and deployment configs are enabled by the use of native Kubernetes API objects ReplicaSet and ReplicationController, respectively, as their building blocks.

Users do not have to manipulate replication controllers, replica sets, or pods owned by DeploymentConfig objects or deployments. The deployment systems ensure changes are propagated appropriately.

TIP

If the existing deployment strategies are not suited for your use case and you must run manual steps during the lifecycle of your deployment, then you should consider creating a custom deployment strategy.

The following sections provide further details on these objects.

6.1.1.1. Replication controllers

A replication controller ensures that a specified number of replicas of a pod are running at all times. If pods exit or are deleted, the replication controller acts to instantiate more up to the defined number. Likewise, if there are more running than desired, it deletes as many as necessary to match the defined amount.

A replication controller configuration consists of:

- The number of replicas desired, which can be adjusted at run time.
- A Pod definition to use when creating a replicated pod.
- A selector for identifying managed pods.

A selector is a set of labels assigned to the pods that are managed by the replication controller. These labels are included in the Pod definition that the replication controller instantiates. The replication controller uses the selector to determine how many instances of the pod are already running in order to adjust as needed.
The replication controller does not perform auto-scaling based on load or traffic, as it does not track either. Rather, this requires its replica count to be adjusted by an external auto-scaler.

The following is an example definition of a replication controller:

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend-1
spec:
  replicas: 1
  selector:
    name: frontend
  template:
    metadata:
      name: frontend
    spec:
      containers:
        - image: openshift/hello-openshift
          name: helloworld
          ports:
            - containerPort: 8080
              protocol: TCP
          restartPolicy: Always
```

1. The number of copies of the pod to run.
2. The label selector of the pod to run.
3. A template for the pod the controller creates.
4. Labels on the pod should include those from the label selector.
5. The maximum name length after expanding any parameters is 63 characters.

### 6.1.1.2. Replica sets

Similar to a replication controller, a **ReplicaSet** is a native Kubernetes API object that ensures a specified number of pod replicas are running at any given time. The difference between a replica set and a replication controller is that a replica set supports set-based selector requirements whereas a replication controller only supports equality-based selector requirements.

**NOTE**

Only use replica sets if you require custom update orchestration or do not require updates at all. Otherwise, use deployments. Replica sets can be used independently, but are used by deployments to orchestrate pod creation, deletion, and updates. Deployments manage their replica sets automatically, provide declarative updates to pods, and do not have to manually manage the replica sets that they create.

The following is an example **ReplicaSet** definition:
A label query over a set of resources. The result of `matchLabels` and `matchExpressions` are logically conjoined.

Equality-based selector to specify resources with labels that match the selector.

Set-based selector to filter keys. This selects all resources with key equal to `tier` and value equal to `frontend`.

### 6.1.2. DeploymentConfig objects

Building on replication controllers, OpenShift Container Platform adds expanded support for the software development and deployment lifecycle with the concept of `DeploymentConfig` objects. In the simplest case, a `DeploymentConfig` object creates a new replication controller and lets it start up pods.

However, OpenShift Container Platform deployments from `DeploymentConfig` objects also provide the ability to transition from an existing deployment of an image to a new one and also define hooks to be run before or after creating the replication controller.

The `DeploymentConfig` deployment system provides the following capabilities:

- A `DeploymentConfig` object, which is a template for running applications.
- Triggers that drive automated deployments in response to events.
- User-customizable deployment strategies to transition from the previous version to the new version. A strategy runs inside a pod commonly referred as the deployment process.
- A set of hooks (lifecycle hooks) for executing custom behavior in different points during the lifecycle of a deployment.
• Versioning of your application to support rollbacks either manually or automatically in case of deployment failure.

• Manual replication scaling and autoscaling.

When you create a DeploymentConfig object, a replication controller is created representing the DeploymentConfig object’s pod template. If the deployment changes, a new replication controller is created with the latest pod template, and a deployment process runs to scale down the old replication controller and scale up the new one.

Instances of your application are automatically added and removed from both service load balancers and routers as they are created. As long as your application supports graceful shutdown when it receives the TERM signal, you can ensure that running user connections are given a chance to complete normally.

The OpenShift Container Platform DeploymentConfig object defines the following details:

1. The elements of a ReplicationController definition.

2. Triggers for creating a new deployment automatically.

3. The strategy for transitioning between deployments.

4. Lifecycle hooks.

Each time a deployment is triggered, whether manually or automatically, a deployer pod manages the deployment (including scaling down the old replication controller, scaling up the new one, and running hooks). The deployment pod remains for an indefinite amount of time after it completes the deployment to retain its logs of the deployment. When a deployment is superseded by another, the previous replication controller is retained to enable easy rollback if needed.

Example DeploymentConfig definition

```yaml
apiVersion: apps.openshift.io/v1
kind: DeploymentConfig
metadata:
  name: frontend
spec:
  replicas: 5
  selector:
    name: frontend
  template: { ... }
  triggers:
    - type: ConfigChange
    - imageChangeParams:
        automatic: true
        containerNames:
        - helloworld
        from:
          kind: ImageStreamTag
          name: hello-openshift:latest
        type: ImageChange
      strategy:
        type: Rolling
```
A configuration change trigger results in a new replication controller whenever changes are detected in the pod template of the deployment configuration.

An image change trigger causes a new deployment to be created each time a new version of the backing image is available in the named image stream.

The default Rolling strategy makes a downtime-free transition between deployments.

6.1.3. Deployments

Kubernetes provides a first-class, native API object type in OpenShift Container Platform called Deployment. Deployment objects serve as a descendant of the OpenShift Container Platform-specific DeploymentConfig object.

Like DeploymentConfig objects, Deployment objects describe the desired state of a particular component of an application as a pod template. Deployments create replica sets, which orchestrate pod lifecycles.

For example, the following deployment definition creates a replica set to bring up one hello-openshift pod:

Deployment definition

apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-openshift
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello-openshift
template:
  metadata:
    labels:
      app: hello-openshift
spec:
  containers:
  - name: hello-openshift
    image: openshift/hello-openshift:latest
    ports:
    - containerPort: 80

6.1.4. Comparing Deployment and DeploymentConfig objects

Both Kubernetes Deployment objects and OpenShift Container Platform-provided DeploymentConfig objects are supported in OpenShift Container Platform; however, it is recommended to use Deployment objects unless you need a specific feature or behavior provided by DeploymentConfig objects.

The following sections go into more detail on the differences between the two object types to further help you decide which type to use.

6.1.4.1. Design
One important difference between Deployment and DeploymentConfig objects is the properties of the CAP theorem that each design has chosen for the rollout process. DeploymentConfig objects prefer consistency, whereas Deployments objects take availability over consistency.

For DeploymentConfig objects, if a node running a deployer pod goes down, it will not get replaced. The process waits until the node comes back online or is manually deleted. Manually deleting the node also deletes the corresponding pod. This means that you can not delete the pod to unstick the rollout, as the kubelet is responsible for deleting the associated pod.

However, deployment rollouts are driven from a controller manager. The controller manager runs in high availability mode on masters and uses leader election algorithms to value availability over consistency. During a failure it is possible for other masters to act on the same deployment at the same time, but this issue will be reconciled shortly after the failure occurs.

### 6.1.4.2. DeploymentConfig object-specific features

**Automatic rollbacks**
Currently, deployments do not support automatically rolling back to the last successfully deployed replica set in case of a failure.

**Triggers**
Deployments have an implicit config change trigger in that every change in the pod template of a deployment automatically triggers a new rollout. If you do not want new rollouts on pod template changes, pause the deployment:

```
$ oc rollout pause deployments/<name>
```

**Lifecycle hooks**
Deployments do not yet support any lifecycle hooks.

**Custom strategies**
Deployments do not support user-specified custom deployment strategies yet.

### 6.1.4.3. Deployment-specific features

**Rollover**
The deployment process for Deployment objects is driven by a controller loop, in contrast to DeploymentConfig objects which use deployer pods for every new rollout. This means that the Deployment object can have as many active replica sets as possible, and eventually the deployment controller will scale down all old replica sets and scale up the newest one.

DeploymentConfig objects can have at most one deployer pod running, otherwise multiple deployers end up conflicting while trying to scale up what they think should be the newest replication controller. Because of this, only two replication controllers can be active at any point in time. Ultimately, this translates to faster rapid rollouts for Deployment objects.

**Proportional scaling**
Because the deployment controller is the sole source of truth for the sizes of new and old replica sets owned by a Deployment object, it is able to scale ongoing rollouts. Additional replicas are distributed proportionally based on the size of each replica set.

DeploymentConfig objects cannot be scaled when a rollout is ongoing because the controller will end up having issues with the deployer process about the size of the new replication controller.

**Pausing mid-rollout**

Deployments can be paused at any point in time, meaning you can also pause ongoing rollouts. On the other hand, you cannot pause deployer pods currently, so if you try to pause a deployment in the middle of a rollout, the deployer process will not be affected and will continue until it finishes.

CHAPTER 6. DEPLOYMENTS

6.2. MANAGING DEPLOYMENT PROCESSES

6.2.1. Managing DeploymentConfig objects

DeploymentConfig objects can be managed from the OpenShift Container Platform web console’s Workloads page or using the oc CLI. The following procedures show CLI usage unless otherwise stated.

6.2.1.1. Starting a deployment

You can start a rollout to begin the deployment process of your application.

Procedure

1. To start a new deployment process from an existing DeploymentConfig object, run the following command:

   $ oc rollout latest dc/<name>

   **NOTE**

   If a deployment process is already in progress, the command displays a message and a new replication controller will not be deployed.

6.2.1.2. Viewing a deployment

You can view a deployment to get basic information about all the available revisions of your application.

Procedure

1. To show details about all recently created replication controllers for the provided DeploymentConfig object, including any currently running deployment process, run the following command:

   $ oc rollout history dc/<name>

2. To view details specific to a revision, add the --revision flag:

   $ oc rollout history dc/<name> --revision=1

3. For more detailed information about a DeploymentConfig object and its latest revision, use the oc describe command:

   $ oc describe dc <name>

6.2.1.3. Retrying a deployment

If the current revision of your DeploymentConfig object failed to deploy, you can restart the deployment process.
Procedure

1. To restart a failed deployment process:

   `$ oc rollout retry dc/<name>`

   If the latest revision of it was deployed successfully, the command displays a message and the deployment process is not retried.

   **NOTE**

   Retrying a deployment restarts the deployment process and does not create a new deployment revision. The restarted replication controller has the same configuration it had when it failed.

6.2.1.4. Rolling back a deployment

Rollbacks revert an application back to a previous revision and can be performed using the REST API, the CLI, or the web console.

Procedure

1. To rollback to the last successful deployed revision of your configuration:

   `$ oc rollout undo dc/<name>`

   The DeploymentConfig object’s template is reverted to match the deployment revision specified in the undo command, and a new replication controller is started. If no revision is specified with `--to-revision`, then the last successfully deployed revision is used.

2. Image change triggers on the DeploymentConfig object are disabled as part of the rollback to prevent accidentally starting a new deployment process soon after the rollback is complete. To re-enable the image change triggers:

   `$ oc set triggers dc/<name> --auto`

   **NOTE**

   Deployment configs also support automatically rolling back to the last successful revision of the configuration in case the latest deployment process fails. In that case, the latest template that failed to deploy stays intact by the system and it is up to users to fix their configurations.

6.2.1.5. Executing commands inside a container

You can add a command to a container, which modifies the container’s startup behavior by overruling the image’s ENTRYPOINT. This is different from a lifecycle hook, which instead can be run once per deployment at a specified time.

Procedure
1. Add the **command** parameters to the **spec** field of the **DeploymentConfig** object. You can also add an **args** field, which modifies the **command** (or the **ENTRYPOINT** if **command** does not exist).

```yaml
spec:
  containers:
  - name: <container_name>
    image: 'image'
    command:
      - '<command>'
    args:
      - '<argument_1>'
      - '<argument_2>'
      - '<argument_3>'
```

For example, to execute the `java` command with the `-jar` and `/opt/app-root/springboots2idemo.jar` arguments:

```yaml
spec:
  containers:
  - name: example-spring-boot
    image: 'image'
    command:
      - java
    args:
      - '-jar'
      - '/opt/app-root/springboots2idemo.jar'
```

### 6.2.1.6. Viewing deployment logs

**Procedure**

1. To stream the logs of the latest revision for a given **DeploymentConfig** object:

   ```bash
   $ oc logs -f dc/<name>
   ```

   If the latest revision is running or failed, the command returns the logs of the process that is responsible for deploying your pods. If it is successful, it returns the logs from a pod of your application.

2. You can also view logs from older failed deployment processes, if and only if these processes (old replication controllers and their deployer pods) exist and have not been pruned or deleted manually:

   ```bash
   $ oc logs --version=1 dc/<name>
   ```

### 6.2.1.7. Deployment triggers

A **DeploymentConfig** object can contain triggers, which drive the creation of new deployment processes in response to events inside the cluster.
WARNING
If no triggers are defined on a DeploymentConfig object, a config change trigger is added by default. If triggers are defined as an empty field, deployments must be started manually.

Config change deployment triggers
The config change trigger results in a new replication controller whenever configuration changes are detected in the pod template of the DeploymentConfig object.

NOTE
If a config change trigger is defined on a DeploymentConfig object, the first replication controller is automatically created soon after the DeploymentConfig object itself is created and it is not paused.

Config change deployment trigger

```
triggers:
  - type: "ConfigChange"
```

Image change deployment triggers
The image change trigger results in a new replication controller whenever the content of an image stream tag changes (when a new version of the image is pushed).

Image change deployment trigger

```
triggers:
  - type: "ImageChange"
    imageChangeParams:
      automatic: true
      from:
        kind: "ImageStreamTag"
        name: "origin-ruby-sample:latest"
        namespace: "myproject"
        containerNames:
          - "helloworld"
```

1 If the imageChangeParams.automatic field is set to false, the trigger is disabled.

With the above example, when the latest tag value of the origin-ruby-sample image stream changes and the new image value differs from the current image specified in the DeploymentConfig object’s helloworld container, a new replication controller is created using the new image for the helloworld container.
NOTE

If an image change trigger is defined on a DeploymentConfig object (with a config change trigger and `automatic=false`, or with `automatic=true`) and the image stream tag pointed by the image change trigger does not exist yet, the initial deployment process will automatically start as soon as an image is imported or pushed by a build to the image stream tag.

6.2.1.7.1. Setting deployment triggers

Procedure

1. You can set deployment triggers for a DeploymentConfig object using the `oc set triggers` command. For example, to set a image change trigger, use the following command:

   ```
   $ oc set triggers dc/<dc_name> \
   --from-image=<project>/<image>:<tag> -c <container_name>
   ```

6.2.1.8. Setting deployment resources

A deployment is completed by a pod that consumes resources (memory, CPU, and ephemeral storage) on a node. By default, pods consume unbounded node resources. However, if a project specifies default container limits, then pods consume resources up to those limits.

NOTE

The minimum memory limit for a deployment is 12 MB. If a container fails to start due to a Cannot allocate memory pod event, the memory limit is too low. Either increase or remove the memory limit. Removing the limit allows pods to consume unbounded node resources.

You can also limit resource use by specifying resource limits as part of the deployment strategy. Deployment resources can be used with the recreate, rolling, or custom deployment strategies.

Procedure

1. In the following example, each of `resources`, `cpu`, `memory`, and `ephemeral-storage` is optional:

   ```
   type: "Recreate"
   resources:
     limits:
       cpu: "100m" ①
       memory: "256Mi" ②
       ephemeral-storage: "1Gi" ③
   ```

   ① `cpu` is in CPU units: `100m` represents 0.1 CPU units (100 * 1e-3).

   ② `memory` is in bytes: `256Mi` represents 268435456 bytes (256 * 2^20).

   ③ `ephemeral-storage` is in bytes: `1Gi` represents 1073741824 bytes (2^30).

However, if a quota has been defined for your project, one of the following two items is required:
• A **resources** section set with an explicit **requests**:

```
type: "Recreate"
resources:
  requests:
    cpu: "100m"
    memory: "256Mi"
    ephemeral-storage: "1Gi"
```

1 The **requests** object contains the list of resources that correspond to the list of resources in the quota.

• A limit range defined in your project, where the defaults from the **LimitRange** object apply to pods created during the deployment process.

To set deployment resources, choose one of the above options. Otherwise, deploy pod creation fails, citing a failure to satisfy quota.

**Additional resources**

• For more information about resource limits and requests, see [Understanding managing application memory](#).

### 6.2.1.9. Scaling manually

In addition to rollbacks, you can exercise fine-grained control over the number of replicas by manually scaling them.

**NOTE**

Pods can also be auto-scaled using the **oc autoscale** command.

**Procedure**

1. To manually scale a **DeploymentConfig** object, use the **oc scale** command. For example, the following command sets the replicas in the **frontend DeploymentConfig** object to 3.

   ```bash
   $ oc scale dc frontend --replicas=3
   ```

   The number of replicas eventually propagates to the desired and current state of the deployment configured by the **DeploymentConfig** object **frontend**.

### 6.2.1.10. Accessing private repositories from DeploymentConfig objects

You can add a secret to your **DeploymentConfig** object so that it can access images from a private repository. This procedure shows the OpenShift Container Platform web console method.

**Procedure**

1. Create a new project.

2. From the **Workloads** page, create a secret that contains credentials for accessing a private image repository.
3. Create a DeploymentConfig object.

4. On the DeploymentConfig object editor page, set the Pull Secret and save your changes.

### 6.2.1.11. Assigning pods to specific nodes

You can use node selectors in conjunction with labeled nodes to control pod placement.

Cluster administrators can set the default node selector for a project in order to restrict pod placement to specific nodes. As a developer, you can set a node selector on a Pod configuration to restrict nodes even further.

**Procedure**

1. To add a node selector when creating a pod, edit the Pod configuration, and add the `nodeSelector` value. This can be added to a single Pod configuration, or in a Pod template:

   ```yaml
   apiVersion: v1
   kind: Pod
   spec:
     nodeSelector:
       disktype: ssd
   ...
   ```

   Pods created when the node selector is in place are assigned to nodes with the specified labels. The labels specified here are used in conjunction with the labels added by a cluster administrator.

   For example, if a project has the `type=user-node` and `region=east` labels added to a project by the cluster administrator, and you add the above `disktype: ssd` label to a pod, the pod is only ever scheduled on nodes that have all three labels.

   **NOTE**

   Labels can only be set to one value, so setting a node selector of `region=west` in a Pod configuration that has `region=east` as the administrator-set default, results in a pod that will never be scheduled.

### 6.2.1.12. Running a pod with a different service account

You can run a pod with a service account other than the default.

**Procedure**

1. Edit the DeploymentConfig object:

   ```bash
   $ oc edit dc/<deployment_config>
   ```

2. Add the `serviceAccount` and `serviceAccountName` parameters to the `spec` field, and specify the service account you want to use:

   ```yaml
   spec:
     securityContext: {}
     serviceAccount: <service_account>
   ```
6.3. USING DEPLOYMENT STRATEGIES

A deployment strategy is a way to change or upgrade an application. The aim is to make the change without downtime in a way that the user barely notices the improvements.

Because the end user usually accesses the application through a route handled by a router, the deployment strategy can focus on DeploymentConfig object features or routing features. Strategies that focus on the deployment impact all routes that use the application. Strategies that use router features target individual routes.

Many deployment strategies are supported through the DeploymentConfig object, and some additional strategies are supported through router features. Deployment strategies are discussed in this section.

Choosing a deployment strategy

Consider the following when choosing a deployment strategy:

- Long-running connections must be handled gracefully.
- Database conversions can be complex and must be done and rolled back along with the application.
- If the application is a hybrid of microservices and traditional components, downtime might be required to complete the transition.
- You must have the infrastructure to do this.
- If you have a non-isolated test environment, you can break both new and old versions.

A deployment strategy uses readiness checks to determine if a new pod is ready for use. If a readiness check fails, the DeploymentConfig object retries to run the pod until it times out. The default timeout is 10m, a value set in TimeoutSeconds in dc.spec.strategy.*params.

6.3.1. Rolling strategy

A rolling deployment slowly replaces instances of the previous version of an application with instances of the new version of the application. The rolling strategy is the default deployment strategy used if no strategy is specified on a DeploymentConfig object.

A rolling deployment typically waits for new pods to become ready via a readiness check before scaling down the old components. If a significant issue occurs, the rolling deployment can be aborted.

When to use a rolling deployment:

- When you want to take no downtime during an application update.
- When your application supports having old code and new code running at the same time.

A rolling deployment means you have both old and new versions of your code running at the same time. This typically requires that your application handle N-1 compatibility.

Example rolling strategy definition
The time to wait between individual pod updates. If unspecified, this value defaults to 1.

The time to wait between polling the deployment status after update. If unspecified, this value defaults to 1.

The time to wait for a scaling event before giving up. Optional; the default is 600. Here, giving up means automatically rolling back to the previous complete deployment.

maxSurge is optional and defaults to 25% if not specified. See the information below the following procedure.

maxUnavailable is optional and defaults to 25% if not specified. See the information below the following procedure.

pre and post are both lifecycle hooks.

The rolling strategy:

1. Executes any pre lifecycle hook.

2. Scales up the new replication controller based on the surge count.

3. Scales down the old replication controller based on the max unavailable count.

4. Repeats this scaling until the new replication controller has reached the desired replica count and the old replication controller has been scaled to zero.

5. Executes any post lifecycle hook.

IMPORTANT

When scaling down, the rolling strategy waits for pods to become ready so it can decide whether further scaling would affect availability. If scaled up pods never become ready, the deployment process will eventually time out and result in a deployment failure.

The maxUnavailable parameter is the maximum number of pods that can be unavailable during the update. The maxSurge parameter is the maximum number of pods that can be scheduled above the original number of pods. Both parameters can be set to either a percentage (e.g., 10%) or an absolute value (e.g., 2). The default value for both is 25%.

These parameters allow the deployment to be tuned for availability and speed. For example:
- maxUnavailable*=0 and maxSurge*=20% ensures full capacity is maintained during the update and rapid scale up.

- maxUnavailable*=10% and maxSurge*=0 performs an update using no extra capacity (an in-place update).

- maxUnavailable*=10% and maxSurge*=10% scales up and down quickly with some potential for capacity loss.

Generally, if you want fast rollouts, use maxSurge. If you have to take into account resource quota and can accept partial unavailability, use maxUnavailable.

### 6.3.1.1. Canary deployments

All rolling deployments in OpenShift Container Platform are *canary deployments*; a new version (the canary) is tested before all of the old instances are replaced. If the readiness check never succeeds, the canary instance is removed and the DeploymentConfig object will be automatically rolled back.

The readiness check is part of the application code and can be as sophisticated as necessary to ensure the new instance is ready to be used. If you must implement more complex checks of the application (such as sending real user workloads to the new instance), consider implementing a custom deployment or using a blue-green deployment strategy.

### 6.3.1.2. Creating a rolling deployment

Rolling deployments are the default type in OpenShift Container Platform. You can create a rolling deployment using the CLI.

**Procedure**

1. Create an application based on the example deployment images found in Quay.io:

   ```bash
   $ oc new-app quay.io/openshifttest/deployment-example:latest
   ```

2. If you have the router installed, make the application available via a route or use the service IP directly.

   ```bash
   $ oc expose svc/deployment-example
   ```

3. Browse to the application at `deployment-example.<project>.<router_domain>` to verify you see the v1 image.

4. Scale the DeploymentConfig object up to three replicas:

   ```bash
   $ oc scale dc/deployment-example --replicas=3
   ```

5. Trigger a new deployment automatically by tagging a new version of the example as the latest tag:

   ```bash
   $ oc tag deployment-example:v2 deployment-example:latest
   ```

6. In your browser, refresh the page until you see the v2 image.
When using the CLI, the following command shows how many pods are on version 1 and how many are on version 2. In the web console, the pods are progressively added to v2 and removed from v1:

```
$ oc describe dc deployment-example
```

During the deployment process, the new replication controller is incrementally scaled up. After the new pods are marked as ready (by passing their readiness check), the deployment process continues.

If the pods do not become ready, the process aborts, and the deployment rolls back to its previous version.

### 6.3.1.3. Starting a rolling deployment using the Developer perspective

**Prerequisites**

- Ensure that you are in the Developer perspective of the web console.
- Ensure that you have created an application using the Add view and see it deployed in the Topology view.

**Procedure**

To start a rolling deployment to upgrade an application:

1. In the Topology view of the Developer perspective, click on the application node to see the Overview tab in the side panel. Note that the Update Strategy is set to the default Rolling strategy.

2. In the Actions drop-down menu, select Start Rollout to start a rolling update. The rolling deployment spins up the new version of the application and then terminates the old one.
Additional resources

- Creating and deploying applications on OpenShift Container Platform using the Developer perspective
- Viewing the applications in your project, verifying their deployment status, and interacting with them in the Topology view

6.3.2. Recreate strategy

The recreate strategy has basic rollout behavior and supports lifecycle hooks for injecting code into the deployment process.

Example recreate strategy definition

```yaml
strategy:
  type: Recreate
  recreateParams:
    pre: {}
    mid: {}  # 2
    post: {}  # 2

1 recreateParams are optional.
2 pre, mid, and post are lifecycle hooks.
```
The recreate strategy:

1. Executes any **pre** lifecycle hook.
2. Scales down the previous deployment to zero.
3. Executes any **mid** lifecycle hook.
4. Scales up the new deployment.
5. Executes any **post** lifecycle hook.

**IMPORTANT**

During scale up, if the replica count of the deployment is greater than one, the first replica of the deployment will be validated for readiness before fully scaling up the deployment. If the validation of the first replica fails, the deployment will be considered a failure.

When to use a recreate deployment:

- When you must run migrations or other data transformations before your new code starts.
- When you do not support having new and old versions of your application code running at the same time.
- When you want to use a RWO volume, which is not supported being shared between multiple replicas.

A recreate deployment incurs downtime because, for a brief period, no instances of your application are running. However, your old code and new code do not run at the same time.

6.3.3. Starting a recreate deployment using the Developer perspective

You can switch the deployment strategy from the default rolling update to a recreate update using the **Developer** perspective in the web console.

**Prerequisites**

- Ensure that you are in the **Developer** perspective of the web console.
- Ensure that you have created an application using the **Add** view and see it deployed in the **Topology** view.

**Procedure**

To switch to a recreate update strategy and to upgrade an application:

1. In the **Actions** drop-down menu, select **Edit Deployment Config** to see the deployment configuration details of the application.
2. In the YAML editor, change the **spec.strategy.type** to **Recreate** and click **Save**.
3. In the **Topology** view, select the node to see the **Overview** tab in the side panel. The **Update Strategy** is now set to **Recreate**.
4. Use the **Actions** drop-down menu to select **Start Rollout** to start an update using the recreate strategy. The recreate strategy first terminates pods for the older version of the application and then spins up pods for the new version.

**Figure 6.2. Recreate update**

---

Additional resources

- Creating and deploying applications on OpenShift Container Platform using the **Developer** perspective
- Viewing the applications in your project, verifying their deployment status, and interacting with them in the **Topology** view

### 6.3.4. Custom strategy

The custom strategy allows you to provide your own deployment behavior.

**Example custom strategy definition**

```yaml
strategy:
```
In the above example, the organization/strategy container image provides the deployment behavior. The optional command array overrides any CMD directive specified in the image’s Dockerfile. The optional environment variables provided are added to the execution environment of the strategy process.

Additionally, OpenShift Container Platform provides the following environment variables to the deployment process:

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPENSHIFT_DEPLOYMENT_NAME</td>
<td>The name of the new deployment, a replication controller.</td>
</tr>
<tr>
<td>OPENSHIFT_DEPLOYMENT_NAMESPACE</td>
<td>The name space of the new deployment.</td>
</tr>
</tbody>
</table>

The replica count of the new deployment will initially be zero. The responsibility of the strategy is to make the new deployment active using the logic that best serves the needs of the user.

Alternatively, use the customParams object to inject the custom deployment logic into the existing deployment strategies. Provide a custom shell script logic and call the openshift-deploy binary. Users do not have to supply their custom deployer container image; in this case, the default OpenShift Container Platform deployer image is used instead:

```yaml
strategy:
  type: Rolling
  customParams:
    command:
      - /bin/sh
      - -c
      - | set -e
      - openshift-deploy --until=50%
      - echo Halfway there
      - openshift-deploy
      - echo Complete
```

This results in following deployment:

```
Started deployment #2
--> Scaling up custom-deployment-2 from 0 to 2, scaling down custom-deployment-1 from 2 to 0
(keep 2 pods available, don't exceed 3 pods)
  Scaling custom-deployment-2 up to 1
--> Reached 50% (currently 50%)
Halfway there
```
If the custom deployment strategy process requires access to the OpenShift Container Platform API or the Kubernetes API the container that executes the strategy can use the service account token available inside the container for authentication.

6.3.5. Lifecycle hooks

The rolling and recreate strategies support *lifecycle hooks*, or deployment hooks, which allow behavior to be injected into the deployment process at predefined points within the strategy:

**Example pre lifecycle hook**

```yaml
pre:
  failurePolicy: Abort
  execNewPod: {}
```

**execNewPod** is a pod-based lifecycle hook.

Every hook has a *failure policy*, which defines the action the strategy should take when a hook failure is encountered:

<table>
<thead>
<tr>
<th>Failure Policy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abort</td>
<td>The deployment process will be considered a failure if the hook fails.</td>
</tr>
<tr>
<td>Retry</td>
<td>The hook execution should be retried until it succeeds.</td>
</tr>
<tr>
<td>Ignore</td>
<td>Any hook failure should be ignored and the deployment should proceed.</td>
</tr>
</tbody>
</table>

Hooks have a type-specific field that describes how to execute the hook. Currently, pod-based hooks are the only supported hook type, specified by the **execNewPod** field.

**Pod-based lifecycle hook**

Pod-based lifecycle hooks execute hook code in a new pod derived from the template in a DeploymentConfig object.

The following simplified example deployment uses the rolling strategy. Triggers and some other minor details are omitted for brevity:

```yaml
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: frontend
spec:
template:
  metadata:
    labels:
```
name: frontend
spec:
  containers:
  - name: helloworld
    image: openshift/origin-ruby-sample
replicas: 5
selector:
  name: frontend
strategy:
  type: Rolling
rollingParams:
  pre:
    failurePolicy: Abort
    execNewPod:
      containerName: helloworld
      command: ["/usr/bin/command", "arg1", "arg2"]
    env:
      - name: CUSTOM_VAR1
        value: custom_value1
    volumes:
      - data

1. The helloworld name refers to spec.template.spec.containers[0].name.
2. This command overrides any ENTRYPOINT defined by the openshift/origin-ruby-sample image.
3. env is an optional set of environment variables for the hook container.
4. volumes is an optional set of volume references for the hook container.

In this example, the pre hook will be executed in a new pod using the openshift/origin-ruby-sample image from the helloworld container. The hook pod has the following properties:

- The hook command is /usr/bin/command arg1 arg2.
- The hook container has the CUSTOM_VAR1=custom_value1 environment variable.
- The hook failure policy is Abort, meaning the deployment process fails if the hook fails.
- The hook pod inherits the data volume from the DeploymentConfig object pod.

6.3.5.1. Setting lifecycle hooks

You can set lifecycle hooks, or deployment hooks, for a deployment using the CLI.

Procedure

1. Use the oc set deployment-hook command to set the type of hook you want: --pre, --mid, or --post. For example, to set a pre-deployment hook:

```bash
$ oc set deployment-hook dc/frontend
  --pre -c helloworld -e CUSTOM_VAR1=custom_value1
  --volumes data --failure-policy=abort -- /usr/bin/command arg1 arg2
```
6.4. USING ROUTE-BASED DEPLOYMENT STRATEGIES

Deployment strategies provide a way for the application to evolve. Some strategies use Deployment objects to make changes that are seen by users of all routes that resolve to the application. Other advanced strategies, such as the ones described in this section, use router features in conjunction with Deployment objects to impact specific routes.

The most common route-based strategy is to use a blue-green deployment. The new version (the green version) is brought up for testing and evaluation, while the users still use the stable version (the blue version). When ready, the users are switched to the green version. If a problem arises, you can switch back to the blue version.

A common alternative strategy is to use A/B versions that are both active at the same time and some users use one version, and some users use the other version. This can be used for experimenting with user interface changes and other features to get user feedback. It can also be used to verify proper operation in a production context where problems impact a limited number of users.

A canary deployment tests the new version but when a problem is detected it quickly falls back to the previous version. This can be done with both of the above strategies.

The route-based deployment strategies do not scale the number of pods in the services. To maintain desired performance characteristics the deployment configurations might have to be scaled.

6.4.1. Proxy shards and traffic splitting

In production environments, you can precisely control the distribution of traffic that lands on a particular shard. When dealing with large numbers of instances, you can use the relative scale of individual shards to implement percentage based traffic. That combines well with a proxy shard, which forwards or splits the traffic it receives to a separate service or application running elsewhere.

In the simplest configuration, the proxy forwards requests unchanged. In more complex setups, you can duplicate the incoming requests and send to both a separate cluster as well as to a local instance of the application, and compare the result. Other patterns include keeping the caches of a DR installation warm, or sampling incoming traffic for analysis purposes.

Any TCP (or UDP) proxy could be run under the desired shard. Use the oc scale command to alter the relative number of instances serving requests under the proxy shard. For more complex traffic management, consider customizing the OpenShift Container Platform router with proportional balancing capabilities.

6.4.2. N-1 compatibility

Applications that have new code and old code running at the same time must be careful to ensure that data written by the new code can be read and handled (or gracefully ignored) by the old version of the code. This is sometimes called schema evolution and is a complex problem.

This can take many forms: data stored on disk, in a database, in a temporary cache, or that is part of a user’s browser session. While most web applications can support rolling deployments, it is important to test and design your application to handle it.

For some applications, the period of time that old code and new code is running side by side is short, so bugs or some failed user transactions are acceptable. For others, the failure pattern may result in the entire application becoming non-functional.
One way to validate N-1 compatibility is to use an A/B deployment: run the old code and new code at the same time in a controlled way in a test environment, and verify that traffic that flows to the new deployment does not cause failures in the old deployment.

### 6.4.3. Graceful termination

OpenShift Container Platform and Kubernetes give application instances time to shut down before removing them from load balancing rotations. However, applications must ensure they cleanly terminate user connections as well before they exit.

On shutdown, OpenShift Container Platform sends a **TERM** signal to the processes in the container. Application code, on receiving **SIGTERM**, stop accepting new connections. This ensures that load balancers route traffic to other active instances. The application code then waits until all open connections are closed, or gracefully terminate individual connections at the next opportunity, before exiting.

After the graceful termination period expires, a process that has not exited is sent the **KILL** signal, which immediately ends the process. The `terminationGracePeriodSeconds` attribute of a pod or pod template controls the graceful termination period (default 30 seconds) and can be customized per application as necessary.

### 6.4.4. Blue-green deployments

Blue-green deployments involve running two versions of an application at the same time and moving traffic from the in-production version (the blue version) to the newer version (the green version). You can use a rolling strategy or switch services in a route.

Because many applications depend on persistent data, you must have an application that supports **N-1 compatibility**, which means it shares data and implements live migration between the database, store, or disk by creating two copies of the data layer.

Consider the data used in testing the new version. If it is the production data, a bug in the new version can break the production version.

#### 6.4.4.1. Setting up a blue-green deployment

Blue-green deployments use two **Deployment** objects. Both are running, and the one in production depends on the service the route specifies, with each **Deployment** object exposed to a different service.

**NOTE**

Routes are intended for web (HTTP and HTTPS) traffic, so this technique is best suited for web applications.

You can create a new route to the new version and test it. When ready, change the service in the production route to point to the new service and the new (green) version is live.

If necessary, you can roll back to the older (blue) version by switching the service back to the previous version.

**Procedure**

1. Create two independent application components.
Create a copy of the example application running the v1 image under the `example-blue` service:

```
$ oc new-app openshift/deployment-example:v1 --name=example-blue
```

b. Create a second copy that uses the v2 image under the `example-green` service:

```
$ oc new-app openshift/deployment-example:v2 --name=example-green
```

2. Create a route that points to the old service:

```
$ oc expose svc/example-blue --name=bluegreen-example
```

3. Browse to the application at `bluegreen-example-<project>.<router_domain>` to verify you see the v1 image.

4. Edit the route and change the service name to `example-green`:

```
$ oc patch route/bluegreen-example -p '{"spec":{"to":{"name":"example-green"}}}'
```

5. To verify that the route has changed, refresh the browser until you see the v2 image.

### 6.4.5. A/B deployments

The A/B deployment strategy lets you try a new version of the application in a limited way in the production environment. You can specify that the production version gets most of the user requests while a limited fraction of requests go to the new version.

Because you control the portion of requests to each version, as testing progresses you can increase the fraction of requests to the new version and ultimately stop using the previous version. As you adjust the request load on each version, the number of pods in each service might have to be scaled as well to provide the expected performance.

In addition to upgrading software, you can use this feature to experiment with versions of the user interface. Since some users get the old version and some the new, you can evaluate the user’s reaction to the different versions to inform design decisions.

For this to be effective, both the old and new versions must be similar enough that both can run at the same time. This is common with bug fix releases and when new features do not interfere with the old. The versions require N-1 compatibility to properly work together.

OpenShift Container Platform supports N-1 compatibility through the web console as well as the CLI.

#### 6.4.5.1. Load balancing for A/B testing

The user sets up a route with multiple services. Each service handles a version of the application.

Each service is assigned a weight and the portion of requests to each service is the service weight divided by the sum of weights. The weight for each service is distributed to the service’s endpoints so that the sum of the endpoint weights is the service weight.

The route can have up to four services. The weight for the service can be between 0 and 256. When the weight is 0, the service does not participate in load-balancing but continues to serve existing persistent connections. When the service weight is not 0, each endpoint has a minimum weight of 1. Because of
this, a service with a lot of endpoints can end up with higher weight than intended. In this case, reduce the number of pods to get the expected load balance weight.

Procedure

To set up the A/B environment:

1. Create the two applications and give them different names. Each creates a Deployment object. The applications are versions of the same program; one is usually the current production version and the other the proposed new version.
   a. Create the first application. The following example creates an application called ab-example-a:

   ```
   $ oc new-app openshift/deployment-example --name=ab-example-a
   ```
   b. Create the second application:

   ```
   $ oc new-app openshift/deployment-example:v2 --name=ab-example-b
   ```
   Both applications are deployed and services are created.

2. Make the application available externally via a route. At this point, you can expose either. It can be convenient to expose the current production version first and later modify the route to add the new version.

   ```
   $ oc expose svc/ab-example-a
   ```
   Browse to the application at ab-example-a.<project>.<router_domain> to verify that you see the expected version.

3. When you deploy the route, the router balances the traffic according to the weights specified for the services. At this point, there is a single service with default weight=1 so all requests go to it. Adding the other service as an alternateBackends and adjusting the weights brings the A/B setup to life. This can be done by the oc set route-backends command or by editing the route. Setting the oc set route-backend to 0 means the service does not participate in load-balancing, but continues to serve existing persistent connections.

   **NOTE**

   Changes to the route just change the portion of traffic to the various services. You might have to scale the deployment to adjust the number of pods to handle the anticipated loads.

   To edit the route, run:

   ```
   $ oc edit route <route_name>
   ```

Example output

```
... metadata:  name: route-alternate-service annotations:
```
6.4.5.1.1. Managing weights of an existing route using the web console

Procedure

1. Navigate to the Networking → Routes page.

   ![Networking and Routes page]

2. Click the Actions menu next to the route you want to edit and select Edit Route.

3. Edit the YAML file. Update the weight to be an integer between 0 and 256 that specifies the relative weight of the target against other target reference objects. The value 0 suppresses requests to this back end. The default is 100. Run `oc explain routes.spec.alternateBackends` for more information about the options.

4. Click Save.

6.4.5.1.2. Managing weights of a new route using the web console

1. Navigate to the Networking → Routes page.

2. Click Create Route.

3. Enter the route Name.

4. Select the Service.

5. Click Add Alternate Service.

6. Enter a value for Weight and Alternate Service Weight. Enter a number between 0 and 255 that depicts relative weight compared with other targets. The default is 100.

7. Select the Target Port.

8. Click Create.

6.4.5.1.3. Managing weights using the CLI

Procedure

1. To manage the services and corresponding weights load balanced by the route, use the `oc set route-backends` command:

   ```
   haproxy.router.openshift.io/balance: roundrobin
   spec:
   host: ab-example.my-project.my-domain
   to:
   kind: Service
   name: ab-example-a
   weight: 10
   alternateBackends:
   - kind: Service
     name: ab-example-b
     weight: 15
   ...
   ```

   OpenShift Container Platform 4.10 Building applications
For example, the following sets `ab-example-a` as the primary service with `weight=198` and `ab-example-b` as the first alternate service with a `weight=2`:

```
$ oc set route-backends ab-example ab-example-a=198 ab-example-b=2
```

This means 99% of traffic is sent to service `ab-example-a` and 1% to service `ab-example-b`.

This command does not scale the deployment. You might be required to do so to have enough pods to handle the request load.

2. Run the command with no flags to verify the current configuration:

```
$ oc set route-backends ab-example
```

Example output

```
NAME               KIND     TO           WEIGHT
routes/ab-example   Service  ab-example-a 198 (99%)  
routes/ab-example   Service  ab-example-b 2   (1%)   
```

3. To alter the weight of an individual service relative to itself or to the primary service, use the `--adjust` flag. Specifying a percentage adjusts the service relative to either the primary or the first alternate (if you specify the primary). If there are other backends, their weights are kept proportional to the changed.

The following example alters the weight of `ab-example-a` and `ab-example-b` services:

```
$ oc set route-backends ab-example --adjust ab-example-a=200 ab-example-b=10
```

Alternatively, alter the weight of a service by specifying a percentage:

```
$ oc set route-backends ab-example --adjust ab-example-b=5%
```

By specifying `+` before the percentage declaration, you can adjust a weighting relative to the current setting. For example:

```
$ oc set route-backends ab-example --adjust ab-example-b=+15%
```

The `--equal` flag sets the `weight` of all services to 100:

```
$ oc set route-backends ab-example --equal
```

The `--zero` flag sets the `weight` of all services to 0. All requests then return with a 503 error.

**NOTE**

Not all routers may support multiple or weighted backends.

6.4.5.1.4. One service, multiple Deployment objects
Procedure

1. Create a new application, adding a label `ab-example=true` that will be common to all shards:

   ```bash
   $ oc new-app openshift/deployment-example --name=ab-example-a --as-deployment-config=true --labels=ab-example=true --env=SUBTITLE=shardA
   $ oc delete svc/ab-example-a
   
   The application is deployed and a service is created. This is the first shard.
   
2. Make the application available via a route, or use the service IP directly:

   ```bash
   $ oc expose deployment ab-example-a --name=ab-example --selector=ab-example=true
   $ oc expose service ab-example
   
   $ oc scale dc/ab-example-a --replicas=0
   $ oc scale dc/ab-example-a --replicas=1; oc scale dc/ab-example-b --replicas=0
   
   $ oc edit dc/ab-example-a
   $ oc edit dc/ab-example-b
   
   $ oc new-app openshift/deployment-example:v2 --name=ab-example-b --labels=ab-example=true SUBTITLE="shard B" COLOR="red" --as-deployment-config=true
   
   $ oc delete svc/ab-example-b
   $ oc new-app openshift/deployment-example --name=ab-example-a --as-deployment-config=true --labels=ab-example=true --env=SUBTITLE=shardA
   
   $ oc delete svc/ab-example-a
   
3. Browse to the application at `ab-example-<project_name>.<router_domain>` to verify you see the `v1` image.

4. Create a second shard based on the same source image and label as the first shard, but with a different tagged version and unique environment variables:

   ```bash
   $ oc new-app openshift/deployment-example:v2 \
   --name=ab-example-b --labels=ab-example=true \
   SUBTITLE="shard B" COLOR="red" --as-deployment-config=true
   $ oc delete svc/ab-example-b
   
4. Create a second shard based on the same source image and label as the first shard, but with a different tagged version and unique environment variables:

   ```bash
   $ oc new-app openshift/deployment-example --name=ab-example-a --as-deployment-config=true --labels=ab-example=true --env=SUBTITLE=shardA
   
5. At this point, both sets of pods are being served under the route. However, because both browsers (by leaving a connection open) and the router (by default, through a cookie) attempt to preserve your connection to a back-end server, you might not see both shards being returned to you.

   To force your browser to one or the other shard:

   a. Use the `oc scale` command to reduce replicas of `ab-example-a` to 0.

      ```bash
      $ oc scale dc/ab-example-a --replicas=0
      
      Refresh your browser to show v2 and shard B (in red).
      
   b. Scale `ab-example-a` to 1 replica and `ab-example-b` to 0:

      ```bash
      $ oc scale dc/ab-example-a --replicas=1; oc scale dc/ab-example-b --replicas=0
      
      Refresh your browser to show v1 and shard A (in blue).
      
6. If you trigger a deployment on either shard, only the pods in that shard are affected. You can trigger a deployment by changing the `SUBTITLE` environment variable in either Deployment object:

   ```bash
   $ oc edit dc/ab-example-a
   
   or
   
   $ oc edit dc/ab-example-b
   ```
7.1. RESOURCE QUOTAS PER PROJECT

A resource quota, defined by a `ResourceQuota` object, provides constraints that limit aggregate resource consumption per project. It can limit the quantity of objects that can be created in a project by type, as well as the total amount of compute resources and storage that might be consumed by resources in that project.

This guide describes how resource quotas work, how cluster administrators can set and manage resource quotas on a per-project basis, and how developers and cluster administrators can view them.

7.1.1. Resources managed by quotas

The following describes the set of compute resources and object types that can be managed by a quota.

**NOTE**

A pod is in a terminal state if `status.phase in (Failed, Succeeded)` is true.

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu</td>
<td>The sum of CPU requests across all pods in a non-terminal state cannot exceed this value. <code>cpu</code> and <code>requests.cpu</code> are the same value and can be used interchangeably.</td>
</tr>
<tr>
<td>memory</td>
<td>The sum of memory requests across all pods in a non-terminal state cannot exceed this value. <code>memory</code> and <code>requests.memory</code> are the same value and can be used interchangeably.</td>
</tr>
<tr>
<td>ephemeral-storage</td>
<td>The sum of local ephemeral storage requests across all pods in a non-terminal state cannot exceed this value. <code>ephemeral-storage</code> and <code>requests.ephemeral-storage</code> are the same value and can be used interchangeably.</td>
</tr>
<tr>
<td>requests.cpu</td>
<td>The sum of CPU requests across all pods in a non-terminal state cannot exceed this value. <code>cpu</code> and <code>requests.cpu</code> are the same value and can be used interchangeably.</td>
</tr>
<tr>
<td>requests.memory</td>
<td>The sum of memory requests across all pods in a non-terminal state cannot exceed this value. <code>memory</code> and <code>requests.memory</code> are the same value and can be used interchangeably.</td>
</tr>
<tr>
<td>requests.ephemeral-storage</td>
<td>The sum of ephemeral storage requests across all pods in a non-terminal state cannot exceed this value. <code>ephemeral-storage</code> and <code>requests.ephemeral-storage</code> are the same value and can be used interchangeably.</td>
</tr>
<tr>
<td>Resource Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>limits.cpu</td>
<td>The sum of CPU limits across all pods in a non-terminal state cannot exceed this value.</td>
</tr>
<tr>
<td>limits.memory</td>
<td>The sum of memory limits across all pods in a non-terminal state cannot exceed this value.</td>
</tr>
<tr>
<td>limits ephemeral-storage</td>
<td>The sum of ephemeral storage limits across all pods in a non-terminal state cannot exceed this value.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>requests.storage</td>
<td>The sum of storage requests across all persistent volume claims in any state cannot exceed this value.</td>
</tr>
<tr>
<td>persistentvolumeclaims</td>
<td>The total number of persistent volume claims that can exist in the project.</td>
</tr>
<tr>
<td>&lt;storage-class-name&gt;.storageclass.storage.k8s.io/requests.storage</td>
<td>The sum of storage requests across all persistent volume claims in any state that have a matching storage class, cannot exceed this value.</td>
</tr>
<tr>
<td>&lt;storage-class-name&gt;.storageclass.storage.k8s.io/persistentvolumeclaims</td>
<td>The total number of persistent volume claims with a matching storage class that can exist in the project.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pods</td>
<td>The total number of pods in a non-terminal state that can exist in the project.</td>
</tr>
<tr>
<td>replicationcontrollers</td>
<td>The total number of ReplicationControllers that can exist in the project.</td>
</tr>
<tr>
<td>resourcequotas</td>
<td>The total number of resource quotas that can exist in the project.</td>
</tr>
<tr>
<td>services</td>
<td>The total number of services that can exist in the project.</td>
</tr>
<tr>
<td>services.loadbalancers</td>
<td>The total number of services of type <strong>LoadBalancer</strong> that can exist in the project.</td>
</tr>
<tr>
<td>services.nodeports</td>
<td>The total number of services of type <strong>NodePort</strong> that can exist in the project.</td>
</tr>
<tr>
<td>Resource Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>secrets</td>
<td>The total number of secrets that can exist in the project.</td>
</tr>
<tr>
<td>configmaps</td>
<td>The total number of ConfigMap objects that can exist in the project.</td>
</tr>
<tr>
<td>persistentvolumeclaims</td>
<td>The total number of persistent volume claims that can exist in the project.</td>
</tr>
<tr>
<td>openshift.io/imagestreams</td>
<td>The total number of imagestreams that can exist in the project.</td>
</tr>
</tbody>
</table>

### 7.1.2. Quota scopes

Each quota can have an associated set of scopes. A quota only measures usage for a resource if it matches the intersection of enumerated scopes.

Adding a scope to a quota restricts the set of resources to which that quota can apply. Specifying a resource outside of the allowed set results in a validation error.

<table>
<thead>
<tr>
<th>Scope</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminating</td>
<td>Match pods where <code>spec.activeDeadlineSeconds &gt;= 0</code>.</td>
</tr>
<tr>
<td>NotTerminating</td>
<td>Match pods where <code>spec.activeDeadlineSeconds</code> is <code>nil</code>.</td>
</tr>
<tr>
<td>BestEffort</td>
<td>Match pods that have best effort quality of service for either <code>cpu</code> or <code>memory</code>.</td>
</tr>
<tr>
<td>NotBestEffort</td>
<td>Match pods that do not have best effort quality of service for <code>cpu</code> and <code>memory</code>.</td>
</tr>
</tbody>
</table>

A **BestEffort** scope restricts a quota to limiting the following resources:

- pods

A **Terminating**, **NotTerminating**, and **NotBestEffort** scope restricts a quota to tracking the following resources:

- pods
- memory
- requests.memory
- limits.memory
- cpu
7.1.3. Quota enforcement

After a resource quota for a project is first created, the project restricts the ability to create any new resources that may violate a quota constraint until it has calculated updated usage statistics.

After a quota is created and usage statistics are updated, the project accepts the creation of new content. When you create or modify resources, your quota usage is incremented immediately upon the request to create or modify the resource.

When you delete a resource, your quota use is decremented during the next full recalculation of quota statistics for the project. A configurable amount of time determines how long it takes to reduce quota usage statistics to their current observed system value.

If project modifications exceed a quota usage limit, the server denies the action, and an appropriate error message is returned to the user explaining the quota constraint violated, and what their currently observed usage statistics are in the system.

7.1.4. Requests versus limits

When allocating compute resources, each container might specify a request and a limit value each for CPU, memory, and ephemeral storage. Quotas can restrict any of these values.

If the quota has a value specified for requests.cpu or requests.memory, then it requires that every incoming container make an explicit request for those resources. If the quota has a value specified for limits.cpu or limits.memory, then it requires that every incoming container specify an explicit limit for those resources.

7.1.5. Sample resource quota definitions

core-object-counts.yaml

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: core-object-counts
spec:
  hard:
    configmaps: "10" 1
    persistentvolumeclaims: "4" 2
    replicationcontrollers: "20" 3
    secrets: "10" 4
    services: "10" 5
    services.loadbalancers: "2" 6
```
1. The total number of **ConfigMap** objects that can exist in the project.
2. The total number of persistent volume claims (PVCs) that can exist in the project.
3. The total number of replication controllers that can exist in the project.
4. The total number of secrets that can exist in the project.
5. The total number of services that can exist in the project.
6. The total number of services of type **LoadBalancer** that can exist in the project.

**openshift-object-counts.yaml**

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:  
  name: openshift-object-counts
spec:  
  hard:
    openshift.io/imagestreams: "10"
```

1. The total number of image streams that can exist in the project.

**compute-resources.yaml**

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:  
  name: compute-resources
spec:  
  hard:
    pods: "4"
    requests.cpu: "1"
    requests.memory: 1Gi
    requests ephemeral-storage: 2Gi
    limits.cpu: "2"
    limits.memory: 2Gi
    limits ephemeral-storage: 4Gi
```

1. The total number of pods in a non-terminal state that can exist in the project.
2. Across all pods in a non-terminal state, the sum of CPU requests cannot exceed 1 core.
3. Across all pods in a non-terminal state, the sum of memory requests cannot exceed 1Gi.
4. Across all pods in a non-terminal state, the sum of ephemeral storage requests cannot exceed 2Gi.
5. Across all pods in a non-terminal state, the sum of CPU limits cannot exceed 2 cores.
6. Across all pods in a non-terminal state, the sum of memory limits cannot exceed 2Gi.
Across all pods in a non-terminal state, the sum of ephemeral storage limits cannot exceed 4Gi.

**besteffort.yaml**

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: besteffort
spec:
  hard:
    pods: "1"
  scopes:
    - BestEffort
```

1. The total number of pods in a non-terminal state with BestEffort quality of service that can exist in the project.
2. Restricts the quota to only matching pods that have BestEffort quality of service for either memory or CPU.

**compute-resources-long-running.yaml**

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources-long-running
spec:
  hard:
    pods: "4"
    limits.cpu: "4"
    limits.memory: "2Gi"
  scopes:
    - NotTerminating
```

1. The total number of pods in a non-terminal state.
2. Across all pods in a non-terminal state, the sum of CPU limits cannot exceed this value.
3. Across all pods in a non-terminal state, the sum of memory limits cannot exceed this value.
4. Restricts the quota to only matching pods where spec.activeDeadlineSeconds is set to nil. Build pods fall under NotTerminating unless the RestartNever policy is applied.

**compute-resources-time-bound.yaml**

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources-time-bound
spec:
  hard:
```

---

OpenShift Container Platform 4.10 Building applications
The total number of pods in a terminating state.

Across all pods in a terminating state, the sum of CPU limits cannot exceed this value.

Across all pods in a terminating state, the sum of memory limits cannot exceed this value.

Restricts the quota to only matching pods where `spec.activeDeadlineSeconds >= 0`. For example, this quota charges for build or deployer pods, but not long running pods like a web server or database.

```
apiVersion: v1
class: ResourceQuota
metadata:
  name: storage-consumption
spec:
  hard:
    persistentvolumeclaims: "10"
    requests.storage: "50Gi"
    gold.storageclass.storage.k8s.io/requests.storage: "10Gi"
    silver.storageclass.storage.k8s.io/requests.storage: "20Gi"
    silver.storageclass.storage.k8s.io/persistentvolumeclaims: "5"
    bronze.storageclass.storage.k8s.io/requests.storage: "0"
    bronze.storageclass.storage.k8s.io/persistentvolumeclaims: "0"
```

The total number of persistent volume claims in a project

Across all persistent volume claims in a project, the sum of storage requested cannot exceed this value.

Across all persistent volume claims in a project, the sum of storage requested in the gold storage class cannot exceed this value.

Across all persistent volume claims in a project, the sum of storage requested in the silver storage class cannot exceed this value.

Across all persistent volume claims in a project, the total number of claims in the silver storage class cannot exceed this value.

Across all persistent volume claims in a project, the sum of storage requested in the bronze storage class cannot exceed this value. When this is set to 0, it means bronze storage class cannot request storage.

Across all persistent volume claims in a project, the sum of storage requested in the bronze storage class cannot exceed this value. When this is set to 0, it means bronze storage class cannot create claims.
7.1.6. Creating a quota

You can create a quota to constrain resource usage in a given project.

Procedure

1. Define the quota in a file.

2. Use the file to create the quota and apply it to a project:

   ```bash
   $ oc create -f <file> [-n <project_name>]
   
   For example:
   $ oc create -f core-object-counts.yaml -n demoproject
   ```

7.1.6.1. Creating object count quotas

You can create an object count quota for all standard namespaced resource types on OpenShift Container Platform, such as `BuildConfig` and `DeploymentConfig` objects. An object quota count places a defined quota on all standard namespaced resource types.

When using a resource quota, an object is charged against the quota upon creation. These types of quotas are useful to protect against exhaustion of resources. The quota can only be created if there are enough spare resources within the project.

Procedure

To configure an object count quota for a resource:

1. Run the following command:

   ```bash
   $ oc create quota <name> \
   --hard=count/<resource>.<group>=<quota>,count/<resource>.<group>=<quota>
   
   For example:
   
   ```

   ```bash
   $ oc create quota test \
   --hard=count/deployments.extensions=2,count/replicasets.extensions=4,count/pods=3,count/secrets=4
   ```

   Example output

   ```bash
   resourcequota "test" created
   ```

   This example limits the listed resources to the hard limit in each project in the cluster.

2. Verify that the quota was created:
7.1.6.2. Setting resource quota for extended resources

Overcommitment of resources is not allowed for extended resources, so you must specify requests and limits for the same extended resource in a quota. Currently, only quota items with the prefix requests is allowed for extended resources. The following is an example scenario of how to set resource quota for the GPU resource nvidia.com/gpu.

Procedure

1. Determine how many GPUs are available on a node in your cluster. For example:

   ```bash
   # oc describe node ip-172-31-27-209.us-west-2.compute.internal | grep 'Capacity|Allocatable|gpu'
   openshift.com/gpu-accelerator=true
   Capacity:
   nvidia.com/gpu: 2
   Allocatable:
   nvidia.com/gpu: 2
   nvidia.com/gpu: 0
   
   In this example, 2 GPUs are available.
   
2. Set a quota in the namespace nvidia. In this example, the quota is 1:

   ```bash
   # cat gpu-quota.yaml
   
   apiVersion: v1
   kind: ResourceQuota
   metadata:
     name: gpu-quota
   namespace: nvidia
   spec:
     hard:
       requests.nvidia.com/gpu: 1
   ```
3. Create the quota:

```bash
# oc create -f gpu-quota.yaml
```

**Example output**

```
resourcequota/gpu-quota created
```

4. Verify that the namespace has the correct quota set:

```bash
# oc describe quota gpu-quota -n nvidia
```

**Example output**

```
Name:                    gpu-quota
Namespace:               nvidia
Resource                 Used  Hard
--------                 ----  ----
requests.nvidia.com/gpu  0     1
```

5. Define a pod that asks for a single GPU. The following example definition file is called `gpu-pod.yaml`:

```yaml
apiVersion: v1
generateName: gpu-pod-
namespace: nvidia
spec:
  containers:
    - name: rhel7-gpu-pod
      image: rhel7
      env:
        - name: NVIDIA_VISIBLE_DEVICES
          value: all
        - name: NVIDIA_DRIVER_CAPABILITIES
          value: "compute,utility"
        - name: NVIDIA_REQUIRE_CUDA
          value: "cuda>=5.0"
      command: ["sleep"]
      args: ["infinity"]
      resources:
        limits:
          nvidia.com/gpu: 1
```

6. Create the pod:

```bash
# oc create -f gpu-pod.yaml
```

7. Verify that the pod is running:

```bash
# oc get pods
```
8. Verify that the quota Used counter is correct:

```
# oc describe quota gpu-quota -n nvidia
```

**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>gpu-pod-s46h7</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>1m</td>
</tr>
</tbody>
</table>

9. Attempt to create a second GPU pod in the `nvidia` namespace. This is technically available on the node because it has 2 GPUs:

```
# oc create -f gpu-pod.yaml
```

**Example output**

Error from server (Forbidden): error when creating "gpu-pod.yaml": pods "gpu-pod-f7z2w" is forbidden: exceeded quota: gpu-quota, requested: requests.nvidia.com/gpu=1, used: requests.nvidia.com/gpu=1, limited: requests.nvidia.com/gpu=1

This *Forbidden* error message is expected because you have a quota of 1 GPU and this pod tried to allocate a second GPU, which exceeds its quota.

### 7.1.7. Viewing a quota

You can view usage statistics related to any hard limits defined in a project’s quota by navigating in the web console to the project’s Quota page.

You can also use the CLI to view quota details.

**Procedure**

1. Get the list of quotas defined in the project. For example, for a project called `demoproject`:

```
$ oc get quota -n demoproject
```

**Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>besteffort</td>
<td>11m</td>
</tr>
<tr>
<td>compute-resources</td>
<td>2m</td>
</tr>
<tr>
<td>core-object-counts</td>
<td>29m</td>
</tr>
</tbody>
</table>

2. Describe the quota you are interested in, for example the `core-object-counts` quota:
7.1.8. Configuring explicit resource quotas

Configure explicit resource quotas in a project request template to apply specific resource quotas in new projects.

Prerequisites

- Access to the cluster as a user with the cluster-admin role.
- Install the OpenShift CLI (oc).

Procedure

1. Add a resource quota definition to a project request template:
   - If a project request template does not exist in a cluster:
     a. Create a bootstrap project template and output it to a file called template.yaml:

        ```
        $ oc adm create-bootstrap-project-template -o yaml > template.yaml
        ```

     b. Add a resource quota definition to template.yaml. The following example defines a resource quota named 'storage-consumption'. The definition must be added before the parameters: section in the template:

        ```yaml
        - apiVersion: v1
          kind: ResourceQuota
          metadata:
            name: storage-consumption
            namespace: ${PROJECT_NAME}
          spec:
            hard:
              persistentvolumeclaims: "10"  
              requests.storage: "50Gi"  
              gold.storageclass.storage.k8s.io/requests.storage: "10Gi"  
              silver.storageclass.storage.k8s.io/requests.storage: "20Gi"  
              silver.storageclass.storage.k8s.io/persistentvolumeclaims: "5"  
              bronze.storageclass.storage.k8s.io/requests.storage: "0"  
              bronze.storageclass.storage.k8s.io/persistentvolumeclaims: "0"
        ```
1. The total number of persistent volume claims in a project.

2. Across all persistent volume claims in a project, the sum of storage requested cannot exceed this value.

3. Across all persistent volume claims in a project, the sum of storage requested in the gold storage class cannot exceed this value.

4. Across all persistent volume claims in a project, the sum of storage requested in the silver storage class cannot exceed this value.

5. Across all persistent volume claims in a project, the total number of claims in the silver storage class cannot exceed this value.

6. Across all persistent volume claims in a project, the sum of storage requested in the bronze storage class cannot exceed this value. When this value is set to 0, the bronze storage class cannot request storage.

7. Across all persistent volume claims in a project, the sum of storage requested in the bronze storage class cannot exceed this value. When this value is set to 0, the bronze storage class cannot create claims.

c. Create a project request template from the modified `template.yaml` file in the `openshift-config` namespace:

```
$ oc create -f template.yaml -n openshift-config
```

**NOTE**

To include the configuration as a `kubectk.kubernetes.io/last-applied-configuration` annotation, add the `--save-config` option to the `oc create` command.

By default, the template is called `project-request`.

- If a project request template already exists within a cluster:

  **NOTE**

  If you declaratively or imperatively manage objects within your cluster by using configuration files, edit the existing project request template through those files instead.

a. List templates in the `openshift-config` namespace:

```
$ oc get templates -n openshift-config
```

b. Edit an existing project request template:

```
$ oc edit template <project_request_template> -n openshift-config
```
c. Add a resource quota definition, such as the preceding `storage-consumption` example, into the existing template. The definition must be added before the `parameters:` section in the template.

2. If you created a project request template, reference it in the cluster’s project configuration resource:
   a. Access the project configuration resource for editing:
      - By using the web console:
         i. Navigate to the Administration → Cluster Settings page.
         ii. Click Configuration to view all configuration resources.
         iii. Find the entry for Project and click Edit YAML.
      - By using the CLI:
         i. Edit the `project.config.openshift.io/cluster` resource:

            ```
            $ oc edit project.config.openshift.io/cluster
            ```

   b. Update the `spec` section of the project configuration resource to include the `projectRequestTemplate` and `name` parameters. The following example references the default project request template name `project-request`:

   ```
   apiVersion: config.openshift.io/v1
   kind: Project
   metadata:
   ...
   spec:
   projectRequestTemplate:
   name: project-request
   ```

3. Verify that the resource quota is applied when projects are created:
   a. Create a project:

      ```
      $ oc new-project <project_name>
      ```

   b. List the project’s resource quotas:

      ```
      $ oc get resourcequotas
      ```

   c. Describe the resource quota in detail:

      ```
      $ oc describe resourcequotas <resource_quota_name>
      ```

7.2. RESOURCE QUOTAS ACROSS MULTIPLE PROJECTS

A multi-project quota, defined by a `ClusterResourceQuota` object, allows quotas to be shared across multiple projects. Resources used in each selected project are aggregated and that aggregate is used to limit resources across all the selected projects.
This guide describes how cluster administrators can set and manage resource quotas across multiple projects.

7.2.1. Selecting multiple projects during quota creation

When creating quotas, you can select multiple projects based on annotation selection, label selection, or both.

Procedure

1. To select projects based on annotations, run the following command:

   ```bash
   $ oc create clusterquota for-user \
       --project-annotation-selector openshift.io/requester=<user_name> \
       --hard pods=10 \
       --hard secrets=20
   ```

   This creates the following `ClusterResourceQuota` object:

   ```yaml
   apiVersion: quota.openshift.io/v1
   kind: ClusterResourceQuota
   metadata:
     name: for-user
   spec:
     quota: 1
     hard:
       pods: "10"
       secrets: "20"
     selector:
       annotations: 2
         openshift.io/requester: <user_name>
       labels: null 3
   status:
     namespaces: 4
     - namespace: ns-one
       status:
         hard:
           pods: "10"
           secrets: "20"
         used:
           pods: "1"
           secrets: "9"
       total: 5
         hard:
           pods: "10"
           secrets: "20"
         used:
           pods: "1"
           secrets: "9"
   ```

   1. The `ResourceQuotaSpec` object that will be enforced over the selected projects.
   2. A simple key-value selector for annotations.
A label selector that can be used to select projects.

A per-namespace map that describes current quota usage in each selected project.

The aggregate usage across all selected projects.

This multi-project quota document controls all projects requested by `<user_name>` using the default project request endpoint. You are limited to 10 pods and 20 secrets.

2. Similarly, to select projects based on labels, run this command:

```bash
$ oc create clusterresourcequota for-name
   --project-label-selector=name=frontend
   --hard=pods=10 --hard=secrets=20
```

Both `clusterresourcequota` and `clusterquota` are aliases of the same command. `for-name` is the name of the `ClusterResourceQuota` object.

To select projects by label, provide a key-value pair by using the format `--project-label-selector=key=value`.

This creates the following `ClusterResourceQuota` object definition:

```yaml
apiVersion: quota.openshift.io/v1
kind: ClusterResourceQuota
metadata:
  creationTimestamp: null
  name: for-name
spec:
  quota:
    hard:
      pods: "10"
      secrets: "20"
  selector:
    annotations: null
    labels:
      matchLabels:
        name: frontend
```

### 7.2.2. Viewing applicable cluster resource quotas

A project administrator is not allowed to create or modify the multi-project quota that limits his or her project, but the administrator is allowed to view the multi-project quota documents that are applied to his or her project. The project administrator can do this via the `AppliedClusterResourceQuota` resource.

**Procedure**

1. To view quotas applied to a project, run:

```bash
$ oc describe AppliedClusterResourceQuota
```

**Example output**
7.2.3. Selection granularity

Because of the locking consideration when claiming quota allocations, the number of active projects selected by a multi-project quota is an important consideration. Selecting more than 100 projects under a single multi-project quota can have detrimental effects on API server responsiveness in those projects.
CHAPTER 8. USING CONFIG MAPS WITH APPLICATIONS

Config maps allow you to decouple configuration artifacts from image content to keep containerized applications portable.

The following sections define config maps and how to create and use them.

For information on creating config maps, see Creating and using config maps.

8.1. UNDERSTANDING CONFIG MAPS

Many applications require configuration using some combination of configuration files, command line arguments, and environment variables. In OpenShift Container Platform, these configuration artifacts are decoupled from image content to keep containerized applications portable.

The ConfigMap object provides mechanisms to inject containers with configuration data while keeping containers agnostic of OpenShift Container Platform. A config map can be used to store fine-grained information like individual properties or coarse-grained information like entire configuration files or JSON blobs.

The ConfigMap API object holds key-value pairs of configuration data that can be consumed in pods or used to store configuration data for system components such as controllers. For example:

```
ConfigMap Object Definition

kind: ConfigMap
apiVersion: v1
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: example-config
  namespace: default
data:
  example.property.1: hello
  example.property.2: world
  example.property.file: |
    property.1=value-1
    property.2=value-2
    property.3=value-3
binaryData:
  bar: L3Jvb3QvMTAw
```

1. Contains the configuration data.

2. Points to a file that contains non-UTF8 data, for example, a binary Java keystore file. Enter the file data in Base 64.

NOTE

You can use the `binaryData` field when you create a config map from a binary file, such as an image.

Configuration data can be consumed in pods in a variety of ways. A config map can be used to:
• Populate environment variable values in containers
• Set command-line arguments in a container
• Populate configuration files in a volume

Users and system components can store configuration data in a config map.

A config map is similar to a secret, but designed to more conveniently support working with strings that do not contain sensitive information.

**Config map restrictions**

A config map must be created before its contents can be consumed in pods.

Controllers can be written to tolerate missing configuration data. Consult individual components configured by using config maps on a case-by-case basis.

**ConfigMap objects reside in a project.**

They can only be referenced by pods in the same project.

The Kubelet only supports the use of a config map for pods it gets from the API server.

This includes any pods created by using the CLI, or indirectly from a replication controller. It does not include pods created by using the OpenShift Container Platform node’s `--manifest-url` flag, its `--config` flag, or its REST API because these are not common ways to create pods.

**8.2. USE CASES: CONSUMING CONFIG MAPS IN PODS**

The following sections describe some uses cases when consuming ConfigMap objects in pods.

**8.2.1. Populating environment variables in containers by using config maps**

Config maps can be used to populate individual environment variables in containers or to populate environment variables in containers from all keys that form valid environment variable names.

As an example, consider the following config map:

**ConfigMap with two environment variables**

| apiVersion: v1 |
| kind: ConfigMap |
| metadata: |
| name: special-config |
| namespace: default |
| data: |
| special.how: very |
| special.type: charm |

1. Name of the config map.
2. The project in which the config map resides. Config maps can only be referenced by pods in the same project.
3.4. Environment variables to inject.
**ConfigMap with one environment variable**

apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config
  namespace: default
data:
  log_level: INFO

1. Name of the config map.
2. Environment variable to inject.

**Procedure**

- You can consume the keys of this **ConfigMap** in a pod using **configMapKeyRef** sections.

**Sample Pod specification configured to inject specific environment variables**

apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
  - name: test-container
    image: gcr.io/google_containers/busybox
    command: ["/bin/sh", "-c", "env"]
    env:
      - name: SPECIAL_LEVEL_KEY
        valueFrom:
          configMapKeyRef:
            name: special-config
            key: special.how
      - name: SPECIAL_TYPE_KEY
        valueFrom:
          configMapKeyRef:
            name: special-config
            key: special.type
          optional: true
    envFrom:
      - configMapRef:
          name: env-config
    restartPolicy: Never

1. Stanza to pull the specified environment variables from a **ConfigMap**.
2. Name of a pod environment variable that you are injecting a key’s value into.
3-5. Name of the **ConfigMap** to pull specific environment variables from.
4. Environment variable to pull from the ConfigMap.

5. Makes the environment variable optional. As optional, the pod will be started even if the specified ConfigMap and keys do not exist.

6. Stanza to pull all environment variables from a ConfigMap.

7. Name of the ConfigMap to pull all environment variables from.

When this pod is run, the pod logs will include the following output:

```
SPECIAL_LEVEL_KEY=very
log_level=INFO
```

**NOTE**

SPECIAL_TYPE_KEY=charm is not listed in the example output because optional: true is set.

### 8.2.2. Setting command-line arguments for container commands with config maps

A config map can also be used to set the value of the commands or arguments in a container. This is accomplished by using the Kubernetes substitution syntax `$(VAR_NAME)`. Consider the following config map:

```
apiversion: v1
kind: ConfigMap
metadata:
  name: special-config
namespace: default
data:
  special.how: very
  special.type: charm
```

**Procedure**

- To inject values into a command in a container, you must consume the keys you want to use as environment variables, as in the consuming ConfigMaps in environment variables use case. Then you can refer to them in a container’s command using the `$(VAR_NAME)` syntax.

**Sample Pod specification configured to inject specific environment variables**

```
apiversion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: ['/bin/sh', '-c', 'echo $(SPECIAL_LEVEL_KEY) $(SPECIAL_TYPE_KEY)']
env:
```
Inject the values into a command in a container using the keys you want to use as environment variables.

When this pod is run, the output from the echo command run in the test-container container is as follows:

very charm

8.2.3. Injecting content into a volume by using config maps

You can inject content into a volume by using config maps.

Example ConfigMap custom resource (CR)

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

Procedure

You have a couple different options for injecting content into a volume by using config maps.

- The most basic way to inject content into a volume by using a config map is to populate the volume with files where the key is the file name and the content of the file is the value of the key:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: ["/bin/sh", "cat", "/etc/config/special.how"]
      volumeMounts:
        - name: config-volume
```

```yaml
- name: SPECIAL_LEVEL_KEY
  valueFrom:
    configMapKeyRef:
      name: special-config
      key: special.how
- name: SPECIAL_TYPE_KEY
  valueFrom:
    configMapKeyRef:
      name: special-config
      key: special.type
```

RestartPolicy: Never
mountPath: /etc/config
volumes:
- name: config-volume
  configMap:
    name: special-config
  restartPolicy: Never

File containing key.

When this pod is run, the output of the cat command will be:

very

- You can also control the paths within the volume where config map keys are projected:

apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "cat", "/etc/config/path/to/special-key" ]
      volumeMounts:
        - name: config-volume
          mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
      items:
        - key: special.how
          path: path/to/special-key
      restartPolicy: Never

Path to config map key.

When this pod is run, the output of the cat command will be:

very
CHAPTER 9. MONITORING PROJECT AND APPLICATION METRICS USING THE DEVELOPER PERSPECTIVE

The Observe view in the Developer perspective provides options to monitor your project or application metrics, such as CPU, memory, and bandwidth usage, and network related information.

9.1. PREREQUISITES

- You have created and deployed applications on OpenShift Container Platform.
- You have logged in to the web console and have switched to the Developer perspective.

9.2. MONITORING YOUR PROJECT METRICS

After you create applications in your project and deploy them, you can use the Developer perspective in the web console to see the metrics for your project.

Procedure

1. On the left navigation panel of the Developer perspective, click Observe to see the Dashboard, Metrics, Alerts, and Events for your project.

2. Optional: Use the Dashboard tab to see graphs depicting the following application metrics:
   - CPU usage
   - Memory usage
   - Bandwidth consumption
   - Network-related information such as the rate of transmitted and received packets and the rate of dropped packets.

In the Dashboard tab, you can access the Kubernetes compute resources dashboards.
Use the following options to see further details:

- Select a dashboard from the **Dashboard** list to see the filtered metrics. All dashboards produce additional sub-menus when selected, except **Kubernetes / Compute Resources / Namespace (Pods)**.

- Select an option from the **Time Range** list to determine the time frame for the data being captured.

- Set a custom time range by selecting **Custom time range** from the **Time Range** list. You can input or select the **From** and **To** dates and times. Click **Save** to save the custom time range.

- Select an option from the **Refresh Interval** list to determine the time period after which the data is refreshed.

- Hover your cursor over the graphs to see specific details for your pod.

- Click **Inspect** located in the upper-right corner of every graph to see any particular graph details. The graph details appear in the **Metrics** tab.

3. Optional: Use the **Metrics** tab to query for the required project metric.
a. In the **Select Query** list, select an option to filter the required details for your project. The filtered metrics for all the application pods in your project are displayed in the graph. The pods in your project are also listed below.

b. From the list of pods, clear the colored square boxes to remove the metrics for specific pods to further filter your query result.

c. Click **Show PromQL** to see the Prometheus query. You can further modify this query with the help of prompts to customize the query and filter the metrics you want to see for that namespace.

d. Use the drop-down list to set a time range for the data being displayed. You can click **Reset Zoom** to reset it to the default time range.

e. Optional: In the **Select Query** list, select **Custom Query** to create a custom Prometheus query and filter relevant metrics.

4. Optional: Use the **Alerts** tab to do the following tasks:
   - See the rules that trigger alerts for the applications in your project.
   - Identify the alerts firing in the project.
   - Silence such alerts if required.
Use the following options to see further details:

- Use the **Filter** list to filter the alerts by their **Alert State** and **Severity**.
- Click on an alert to go to the details page for that alert. In the **Alerts Details** page, you can click **View Metrics** to see the metrics for the alert.
- Use the **Notifications** toggle adjoining an alert rule to silence all the alerts for that rule, and then select the duration for which the alerts will be silenced from the **Silence for** list. You must have the permissions to edit alerts to see the **Notifications** toggle.
- Use the **Options** menu adjoining an alert rule to see the details of the alerting rule.

5. Optional: Use the **Events** tab to see the events for your project.

**Figure 9.4. Monitoring events**

You can filter the displayed events using the following options:

- In the **Resources** list, select a resource to see events for that resource.
- In the **All Types** list, select a type of event to see events relevant to that type.
- Search for specific events using the **Filter events by names or messages** field.

### 9.3. MONITORING YOUR APPLICATION METRICS

After you create applications in your project and deploy them, you can use the **Topology** view in the **Developer** perspective to see the alerts and metrics for your application. Critical and warning alerts for your application are indicated on the workload node in the **Topology** view.

**Procedure**

To see the alerts for your workload:

1. In the **Topology** view, click the workload to see the workload details in the right panel.
2. Click the **Observe** tab to see the critical and warning alerts for the application; graphs for metrics, such as CPU, memory, and bandwidth usage; and all the events for the application.
NOTE

Only critical and warning alerts in the Firing state are displayed in the Topology view. Alerts in the Silenced, Pending and Not Firing states are not displayed.

Figure 9.5. Monitoring application metrics

- Click the alert listed in the right panel to see the alert details in the Alert Details page.
- Click any of the charts to go to the Metrics tab to see the detailed metrics for the application.
- Click View monitoring dashboard to see the monitoring dashboard for that application.

9.4. ADDITIONAL RESOURCES

- Monitoring overview
CHAPTER 10. MONITORING APPLICATION HEALTH BY USING HEALTH CHECKS

In software systems, components can become unhealthy due to transient issues such as temporary connectivity loss, configuration errors, or problems with external dependencies. OpenShift Container Platform applications have a number of options to detect and handle unhealthy containers.

10.1. UNDERSTANDING HEALTH CHECKS

A health check periodically performs diagnostics on a running container using any combination of the readiness, liveness, and startup health checks.

You can include one or more probes in the specification for the pod that contains the container which you want to perform the health checks.

NOTE

If you want to add or edit health checks in an existing pod, you must edit the pod DeploymentConfig object or use the Developer perspective in the web console. You cannot use the CLI to add or edit health checks for an existing pod.

Readiness probe

A readiness probe determines if a container is ready to accept service requests. If the readiness probe fails for a container, the kubelet removes the pod from the list of available service endpoints. After a failure, the probe continues to examine the pod. If the pod becomes available, the kubelet adds the pod to the list of available service endpoints.

Liveness health check

A liveness probe determines if a container is still running. If the liveness probe fails due to a condition such as a deadlock, the kubelet kills the container. The pod then responds based on its restart policy. For example, a liveness probe on a pod with a restartPolicy of Always or OnFailure kills and restarts the container.

Startup probe

A startup probe indicates whether the application within a container is started. All other probes are disabled until the startup succeeds. If the startup probe does not succeed within a specified time period, the kubelet kills the container, and the container is subject to the pod restartPolicy. Some applications can require additional startup time on their first initialization. You can use a startup probe with a liveness or readiness probe to delay that probe long enough to handle lengthy start-up time using the failureThreshold and periodSeconds parameters.

For example, you can add a startup probe, with a failureThreshold of 30 failures and a periodSeconds of 10 seconds (30 * 10s = 300s) for a maximum of 5 minutes, to a liveness probe. After the startup probe succeeds the first time, the liveness probe takes over.

You can configure liveness, readiness, and startup probes with any of the following types of tests:

- HTTP GET: When using an HTTP GET test, the test determines the healthiness of the container by using a web hook. The test is successful if the HTTP response code is between 200 and 399. You can use an HTTP GET test with applications that return HTTP status codes when completely initialized.
• Container Command: When using a container command test, the probe executes a command inside the container. The probe is successful if the test exits with a 0 status.

• TCP socket: When using a TCP socket test, the probe attempts to open a socket to the container. The container is only considered healthy if the probe can establish a connection. You can use a TCP socket test with applications that do not start listening until initialization is complete.

You can configure several fields to control the behavior of a probe:

• `initialDelaySeconds`: The time, in seconds, after the container starts before the probe can be scheduled. The default is 0.

• `periodSeconds`: The delay, in seconds, between performing probes. The default is 10. This value must be greater than `timeoutSeconds`.

• `timeoutSeconds`: The number of seconds of inactivity after which the probe times out and the container is assumed to have failed. The default is 1. This value must be lower than `periodSeconds`.

• `successThreshold`: The number of times that the probe must report success after a failure to reset the container status to successful. The value must be 1 for a liveness probe. The default is 1.

• `failureThreshold`: The number of times that the probe is allowed to fail. The default is 3. After the specified attempts:
  - for a liveness probe, the container is restarted
  - for a readiness probe, the pod is marked **Unready**
  - for a startup probe, the container is killed and is subject to the pod’s `restartPolicy`.

**Example probes**
The following are samples of different probes as they would appear in an object specification.

**Sample readiness probe with a container command readiness probe in a pod spec**

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: health-check
name: my-application
...
spec:
  containers:
  - name: goproxy-app
    args:
      image: k8s.gcr.io/goproxy:0.1
    readinessProbe:
      exec:
        command: 
        - /tmp/healthy
...
```
The container name.

The container image to deploy.

A readiness probe.

A container command test.

The commands to execute on the container.

Sample container command startup probe and liveness probe with container command tests in a pod spec

```yaml
apiVersion: v1
type: Pod
metadata:
  labels:
    test: health-check
name: my-application
...
spec:
  containers:
    - name: goproxy-app
      args:
        image: k8s.gcr.io/goproxy:0.1
  livenessProbe:
    httpGet:
      scheme: HTTPS
      path: /healthz
      port: 8080
      httpHeaders:
        - name: X-Custom-Header
          value: Awesome
  startupProbe:
    httpGet:
      path: /healthz
      port: 8080
      failureThreshold: 30
      periodSeconds: 10

...  
```

The container name.

Specify the container image to deploy.

A liveness probe.

An HTTP **GET** test.

The internet scheme: **HTTP** or **HTTPS**. The default value is **HTTP**.

The port on which the container is listening.
A startup probe.

An HTTP GET test.

The port on which the container is listening.

The number of times to try the probe after a failure.

The number of seconds to perform the probe.

Sample liveness probe with a container command test that uses a timeout in a pod spec

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: health-check
  name: my-application
spec:
  containers:
    - name: goproxy-app
      image: k8s.gcr.io/goproxy:0.1
      livenessProbe:
        exec:
          command:
            - /bin/bash
            - '-c'
            - timeout 60 /opt/eap/bin/livenessProbe.sh
        periodSeconds: 10
        successThreshold: 1
        failureThreshold: 3
```

1. The container name.
2. Specify the container image to deploy.
3. The liveness probe.
4. The type of probe, here a container command probe.
5. The command line to execute inside the container.
6. How often in seconds to perform the probe.
7. The number of consecutive successes needed to show success after a failure.
8. The number of times to try the probe after a failure.

Sample readiness probe and liveness probe with a TCP socket test in a deployment
10.2. CONFIGURING HEALTH CHECKS USING THE CLI

To configure readiness, liveness, and startup probes, add one or more probes to the specification for the pod that contains the container which you want to perform the health checks.

**NOTE**

If you want to add or edit health checks in an existing pod, you must edit the pod **DeploymentConfig** object or use the **Developer** perspective in the web console. You cannot use the CLI to add or edit health checks for an existing pod.

**Procedure**

To add probes for a container:

1. Create a **Pod** object to add one or more probes:

   ```yaml
   apiVersion: v1
   kind: Pod
   metadata:
   labels:
   ```
Specify the container name.

Specify the container image to deploy.

Optional: Create a Liveness probe.

Specify a test to perform, here a TCP Socket test.

Specify the port on which the container is listening.

Specify the time, in seconds, after the container starts before the probe can be scheduled.

Specify the number of seconds to perform the probe. The default is 10. This value must be greater than `timeoutSeconds`.

Specify the number of seconds of inactivity after which the probe is assumed to have failed. The default is 1. This value must be lower than `periodSeconds`.

Optional: Create a Readiness probe.

Specify the type of test to perform, here an HTTP test.

Specify a host IP address. When `host` is not defined, the `PodIP` is used.

Specify `HTTP` or `HTTPS`. When `scheme` is not defined, the `HTTP` scheme is used.
Specify the port on which the container is listening.

Optional: Create a Startup probe.

Specify the type of test to perform, here an Container Execution probe.

Specify the commands to execute on the container.

Specify the number of times to try the probe after a failure.

Specify the number of seconds to perform the probe. The default is 10. This value must be greater than `timeoutSeconds`.

Specify the number of seconds of inactivity after which the probe is assumed to have failed. The default is 1. This value must be lower than `periodSeconds`.

**NOTE**

If the `initialDelaySeconds` value is lower than the `periodSeconds` value, the first Readiness probe occurs at some point between the two periods due to an issue with timers.

The `timeoutSeconds` value must be lower than the `periodSeconds` value.

2. Create the **Pod** object:

   ```
   $ oc create -f <file-name>.yaml
   ```

3. Verify the state of the health check pod:

   ```
   $ oc describe pod health-check
   ```

**Example output**

Events:

<table>
<thead>
<tr>
<th>Type</th>
<th>Reason</th>
<th>Age</th>
<th>From</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>Scheduled</td>
<td>9s</td>
<td>default-scheduler</td>
<td>Successfully assigned openshift-logging/liveness-exec to ip-10-0-143-40.ec2.internal</td>
</tr>
<tr>
<td>Normal</td>
<td>Pulling</td>
<td>2s</td>
<td>kubelet, ip-10-0-143-40.ec2.internal</td>
<td>pulling image &quot;k8s.gcr.io/liveness&quot;</td>
</tr>
<tr>
<td>Normal</td>
<td>Pulled</td>
<td>1s</td>
<td>kubelet, ip-10-0-143-40.ec2.internal</td>
<td>Successfully pulled image &quot;k8s.gcr.io/liveness&quot;</td>
</tr>
<tr>
<td>Normal</td>
<td>Created</td>
<td>1s</td>
<td>kubelet, ip-10-0-143-40.ec2.internal</td>
<td>Created container</td>
</tr>
<tr>
<td>Normal</td>
<td>Started</td>
<td>1s</td>
<td>kubelet, ip-10-0-143-40.ec2.internal</td>
<td>Started container</td>
</tr>
</tbody>
</table>

The following is the output of a failed probe that restarted a container:

**Sample Liveness check output with unhealthy container**

```
$ oc describe pod pod1
```

**Example output**
10.3. MONITORING APPLICATION HEALTH USING THE DEVELOPER PERSPECTIVE

You can use the Developer perspective to add three types of health probes to your container to ensure that your application is healthy:

- Use the Readiness probe to check if the container is ready to handle requests.
- Use the Liveness probe to check if the container is running.
- Use the Startup probe to check if the application within the container has started.

You can add health checks either while creating and deploying an application, or after you have deployed an application.

10.4. ADDING HEALTH CHECKS USING THE DEVELOPER PERSPECTIVE

You can use the Topology view to add health checks to your deployed application.

Prerequisites:

- You have switched to the Developer perspective in the web console.
- You have created and deployed an application on OpenShift Container Platform using the Developer perspective.
Procedure

1. In the **Topology** view, click on the application node to see the side panel. If the container does not have health checks added to ensure the smooth running of your application, a **Health Checks** notification is displayed with a link to add health checks.

2. In the displayed notification, click the **Add Health Checks** link.

3. Alternatively, you can also click the **Actions** drop-down list and select **Add Health Checks**. Note that if the container already has health checks, you will see the **Edit Health Checks** option instead of the add option.

4. In the **Add Health Checks** form, if you have deployed multiple containers, use the **Container** drop-down list to ensure that the appropriate container is selected.

5. Click the required health probe links to add them to the container. Default data for the health checks is prepopulated. You can add the probes with the default data or further customize the values and then add them. For example, to add a Readiness probe that checks if your container is ready to handle requests:
   a. Click **Add Readiness Probe** to see a form containing the parameters for the probe.
   b. Click the **Type** drop-down list to select the request type you want to add. For example, in this case, select **Container Command** to select the command that will be executed inside the container.
   c. In the **Command** field, add an argument **cat**, similarly, you can add multiple arguments for the check, for example, add another argument **/tmp/healthy**.
   d. Retain or modify the default values for the other parameters as required.

   **NOTE**

   The **Timeout** value must be lower than the **Period** value. The **Timeout** default value is **1**. The **Period** default value is **10**.

   e. Click the check mark at the bottom of the form. The **Readiness Probe Added** message is displayed.

6. Click **Add** to add the health check. You are redirected to the **Topology** view and the container is restarted.

7. In the side panel, verify that the probes have been added by clicking on the deployed pod under the **Pods** section.

8. In the **Pod Details** page, click the listed container in the **Containers** section.

9. In the **Container Details** page, verify that the Readiness probe - **Exec Command** **cat** **/tmp/healthy** has been added to the container.

10.5. EDITING HEALTH CHECKS USING THE DEVELOPER PERSPECTIVE

You can use the **Topology** view to edit health checks added to your application, modify them, or add more health checks.
Prerequisites:

- You have switched to the Developer perspective in the web console.
- You have created and deployed an application on OpenShift Container Platform using the Developer perspective.
- You have added health checks to your application.

Procedure

1. In the Topology view, right-click your application and select Edit Health Checks. Alternatively, in the side panel, click the Actions drop-down list and select Edit Health Checks.

2. In the Edit Health Checks page:
   - To remove a previously added health probe, click the minus sign adjoining it.
   - To edit the parameters of an existing probe:
     a. Click the Edit Probe link next to a previously added probe to see the parameters for the probe.
     b. Modify the parameters as required, and click the check mark to save your changes.
   - To add a new health probe, in addition to existing health checks, click the add probe links. For example, to add a Liveness probe that checks if your container is running:
     a. Click Add Liveness Probe, to see a form containing the parameters for the probe.
     b. Edit the probe parameters as required.

   NOTE
   The Timeout value must be lower than the Period value. The Timeout default value is 1. The Period default value is 10.
   c. Click the check mark at the bottom of the form. The Liveness Probe Added message is displayed.

3. Click Save to save your modifications and add the additional probes to your container. You are redirected to the Topology view.

4. In the side panel, verify that the probes have been added by clicking on the deployed pod under the Pods section.

5. In the Pod Details page, click the listed container in the Containers section.

6. In the Container Details page, verify that the Liveness probe - HTTP Get 10.129.4.65:8080/ has been added to the container, in addition to the earlier existing probes.

10.6. MONITORING HEALTH CHECK FAILURES USING THE DEVELOPER PERSPECTIVE

In case an application health check fails, you can use the Topology view to monitor these health check violations.
Prerequisites:

- You have switched to the Developer perspective in the web console.
- You have created and deployed an application on OpenShift Container Platform using the Developer perspective.
- You have added health checks to your application.

Procedure

1. In the Topology view, click on the application node to see the side panel.

2. Click the Observe tab to see the health check failures in the Events (Warning) section.

3. Click the down arrow adjoining Events (Warning) to see the details of the health check failure.

Additional resources

- For details on switching to the Developer perspective in the web console, see About the Developer perspective.

- For details on adding health checks while creating and deploying an application, see Advanced Options in the Creating applications using the Developer perspective section.
CHAPTER 11. EDITING APPLICATIONS

You can edit the configuration and the source code of the application you create using the Topology view.

11.1. PREREQUISITES

- You have the appropriate roles and permissions in a project to create and modify applications in OpenShift Container Platform.
- You have created and deployed an application on OpenShift Container Platform using the Developer perspective.
- You have logged in to the web console and have switched to the Developer perspective.

11.2. EDITING THE SOURCE CODE OF AN APPLICATION USING THE DEVELOPER PERSPECTIVE

You can use the Topology view in the Developer perspective to edit the source code of your application.

Procedure

- In the Topology view, click the Edit Source code icon, displayed at the bottom-right of the deployed application, to access your source code and modify it.

NOTE

This feature is available only when you create applications using the From Git, From Catalog, and the From Dockerfile options.

If the Eclipse Che Operator is installed in your cluster, a Che workspace is created and you are directed to the workspace to edit your source code. If it is not installed, you will be directed to the Git repository your source code is hosted in.

11.3. EDITING THE APPLICATION CONFIGURATION USING THE DEVELOPER PERSPECTIVE

You can use the Topology view in the Developer perspective to edit the configuration of your application.

NOTE

Currently, only configurations of applications created by using the From Git, Container Image, From Catalog, or From Dockerfile options in the Add workflow of the Developer perspective can be edited. Configurations of applications created by using the CLI or the YAML option from the Add workflow cannot be edited.

Prerequisites
Ensure that you have created an application using the From Git, Container Image, From Catalog, or From Dockerfile options in the Add workflow.

**Procedure**

1. After you have created an application and it is displayed in the Topology view, right-click the application to see the edit options available.

   **Figure 11.1. Edit application**

2. Click Edit application-name to see the Add workflow you used to create the application. The form is pre-populated with the values you had added while creating the application.

3. Edit the necessary values for the application.
NOTE

You cannot edit the Name field in the General section, the CI/CD pipelines, or the Create a route to the application field in the Advanced Options section.

4. Click Save to restart the build and deploy a new image.

Figure 11.2. Edit and redeploy application
CHAPTER 12. PRUNING OBJECTS TO RECLAIM RESOURCES

Over time, API objects created in OpenShift Container Platform can accumulate in the cluster’s etcd data store through normal user operations, such as when building and deploying applications.

Cluster administrators can periodically prune older versions of objects from the cluster that are no longer required. For example, by pruning images you can delete older images and layers that are no longer in use, but are still taking up disk space.

12.1. BASIC PRUNING OPERATIONS

The CLI groups prune operations under a common parent command:

```
$ oc adm prune <object_type> <options>
```

This specifies:

- The `<object_type>` to perform the action on, such as `groups`, `builds`, `deployments`, or `images`.
- The `<options>` supported to prune that object type.

12.2. PRUNING GROUPS

To prune groups records from an external provider, administrators can run the following command:

```
$ oc adm prune groups --sync-config=path/to/sync/config [<options>]
```

Table 12.1. `oc adm prune groups` flags

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--confirm</td>
<td>Indicate that pruning should occur, instead of performing a dry-run.</td>
</tr>
<tr>
<td>--blacklist</td>
<td>Path to the group blacklist file.</td>
</tr>
<tr>
<td>--whitelist</td>
<td>Path to the group whitelist file.</td>
</tr>
<tr>
<td>--sync-config</td>
<td>Path to the synchronization configuration file.</td>
</tr>
</tbody>
</table>

Procedure

1. To see the groups that the prune command deletes, run the following command:

   ```
   $ oc adm prune groups --sync-config=ldap-sync-config.yaml
   ```

2. To perform the prune operation, add the `--confirm` flag:

   ```
   $ oc adm prune groups --sync-config=ldap-sync-config.yaml --confirm
   ```
12.3. PRUNING DEPLOYMENT RESOURCES

You can prune resources associated with deployments that are no longer required by the system, due to age and status.

The following command prunes replication controllers associated with DeploymentConfig objects:

```
$ oc adm prune deployments [options]
```

**NOTE**

To also prune replica sets associated with Deployment objects, use the `--replica-sets` flag. This flag is currently a Technology Preview feature.

**Table 12.2. oc adm prune deployments flags**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--confirm</td>
<td>Indicate that pruning should occur, instead of performing a dry-run.</td>
</tr>
<tr>
<td>--keep-complete=&lt;N&gt;</td>
<td>Per the DeploymentConfig object, keep the last N replication controllers that have a status of Complete and replica count of zero. The default is 5.</td>
</tr>
<tr>
<td>--keep-failed=&lt;N&gt;</td>
<td>Per the DeploymentConfig object, keep the last N replication controllers that have a status of Failed and replica count of zero. The default is 1.</td>
</tr>
<tr>
<td>--keep-younger-than=</td>
<td>Do not prune any replication controller that is younger than &lt;duration&gt; relative to the current time. Valid units of measurement include nanoseconds (ns), microseconds (us), milliseconds (ms), seconds (s), minutes (m), and hours (h). The default is 60m.</td>
</tr>
<tr>
<td>&lt;duration&gt;</td>
<td></td>
</tr>
<tr>
<td>--orphans</td>
<td>Prune all replication controllers that no longer have a DeploymentConfig object, has status of Complete or Failed, and has a replica count of zero.</td>
</tr>
<tr>
<td>--replica-sets=true</td>
<td>false</td>
</tr>
</tbody>
</table>

**IMPORTANT**

This flag is a Technology Preview feature.

**Procedure**

1. To see what a pruning operation would delete, run the following command:

   ```
   $ oc adm prune deployments --orphans --keep-complete=5 --keep-failed=1
   --keep-younger-than=60m
   ```
2. To actually perform the prune operation, add the `--confirm` flag:

```
$ oc adm prune deployments --orphans --keep-complete=5 --keep-failed=1 \
   --keep-younger-than=60m --confirm
```

### 12.4. PRUNING BUILDS

To prune builds that are no longer required by the system due to age and status, administrators can run the following command:

```
$ oc adm prune builds [<options>]
```

#### Table 12.3. `oc adm prune builds` flags

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--confirm</code></td>
<td>Indicate that pruning should occur, instead of performing a dry-run.</td>
</tr>
<tr>
<td><code>--orphans</code></td>
<td>Prune all builds whose build configuration no longer exists, status is</td>
</tr>
<tr>
<td></td>
<td>complete, failed, error, or canceled.</td>
</tr>
<tr>
<td><code>--keep-complete=&lt;N&gt;</code></td>
<td>Per build configuration, keep the last \text{N} builds whose status is</td>
</tr>
<tr>
<td></td>
<td>complete. The default is 5.</td>
</tr>
<tr>
<td><code>--keep-failed=&lt;N&gt;</code></td>
<td>Per build configuration, keep the last \text{N} builds whose status is</td>
</tr>
<tr>
<td></td>
<td>failed, error, or canceled. The default is 1.</td>
</tr>
<tr>
<td><code>--keep-younger-than=&lt;duration&gt;</code></td>
<td>Do not prune any object that is younger than \text{duration} relative to</td>
</tr>
<tr>
<td></td>
<td>the current time. The default is 60m.</td>
</tr>
</tbody>
</table>

#### Procedure

1. To see what a pruning operation would delete, run the following command:

   ```
   $ oc adm prune builds --orphans --keep-complete=5 --keep-failed=1 \
   --keep-younger-than=60m
   ```

2. To actually perform the prune operation, add the `--confirm` flag:

   ```
   $ oc adm prune builds --orphans --keep-complete=5 --keep-failed=1 \
   --keep-younger-than=60m --confirm
   ```

#### NOTE

Developers can enable automatic build pruning by modifying their build configuration.

#### Additional resources

- Performing advanced builds → Pruning builds
12.5. AUTOMATICALLY PRUNING IMAGES

Images that are no longer required by the system due to age, status, or exceed limits are automatically pruned. Cluster administrators can configure the Pruning Custom Resource, or suspend it.

Prerequisites

- Cluster administrator permissions.
- Install the `oc` CLI.

Procedure

- Verify that the object named `imagepruners.imageregistry.operator.openshift.io/cluster` contains the following `spec` and `status` fields:

```
spec:
schedule: 0 0 * * *
suspend: false
keepTagRevisions: 3
keepYoungerThanDuration: 60m
keepYoungerThan: 3600000000000
resources: {}
affinity: {}
nodeSelector: {}
tolerations: []
successfulJobsHistoryLimit: 3
failedJobsHistoryLimit: 3
status:
obsevedGeneration: 2
conditions:
- type: Available
  status: "True"
  lastTransitionTime: 2019-10-09T03:13:45
  reason: Ready
  message: "Periodic image pruner has been created."
- type: Scheduled
  status: "True"
  lastTransitionTime: 2019-10-09T03:13:45
  reason: Scheduled
  message: "Image pruner job has been scheduled."
- type: Failed
  status: "False"
  lastTransitionTime: 2019-10-09T03:13:45
  reason: Succeeded
  message: "Most recent image pruning job succeeded."
```

1. **schedule**: CronJob formatted schedule. This is an optional field, default is daily at midnight.
2. **suspend**: If set to `true`, the CronJob running pruning is suspended. This is an optional field, default is `false`. The initial value on new clusters is `false`.
3. **keepTagRevisions**: The number of revisions per tag to keep. This is an optional field, default is 3. The initial value is 3.
keepYoungerThanDuration: Retain images younger than this duration. This is an optional field. If a value is not specified, either keepYoungerThan or the default value 60m (60 minutes) is used.

keepYoungerThan: Deprecated. The same as keepYoungerThanDuration, but the duration is specified as an integer in nanoseconds. This is an optional field. When keepYoungerThanDuration is set, this field is ignored.

resources: Standard pod resource requests and limits. This is an optional field.

affinity: Standard pod affinity. This is an optional field.

nodeSelector: Standard pod node selector. This is an optional field.

tolerations: Standard pod tolerations. This is an optional field.

successfulJobsHistoryLimit: The maximum number of successful jobs to retain. Must be >= 1 to ensure metrics are reported. This is an optional field, default is 3. The initial value is 3.

failedJobsHistoryLimit: The maximum number of failed jobs to retain. Must be >= 1 to ensure metrics are reported. This is an optional field, default is 3. The initial value is 3.

observedGeneration: The generation observed by the Operator.

conditions: The standard condition objects with the following types:

- Available: Indicates if the pruning job has been created. Reasons can be Ready or Error.
- Scheduled: Indicates if the next pruning job has been scheduled. Reasons can be Scheduled, Suspended, or Error.
- Failed: Indicates if the most recent pruning job failed.

**IMPORTANT**

The Image Registry Operator’s behavior for managing the pruner is orthogonal to the managementState specified on the Image Registry Operator’s ClusterOperator object. If the Image Registry Operator is not in the Managed state, the image pruner can still be configured and managed by the Pruning Custom Resource.

However, the managementState of the Image Registry Operator alters the behavior of the deployed image pruner job:

- Managed: the --prune-registry flag for the image pruner is set to true.
- Removed: the --prune-registry flag for the image pruner is set to false, meaning it only prunes image metadata in etcd.
- Unmanaged: the --prune-registry flag for the image pruner is set to false.

### 12.6. MANUALLY PRUNING IMAGES

The pruning custom resource enables automatic image pruning. However, administrators can manually prune images that are no longer required by the system due to age, status, or exceed limits. There are two methods to manually prune images:
- Running image pruning as a **Job** or **CronJob** on the cluster.
- Running the **oc adm prune images** command.

**Prerequisites**

- To prune images, you must first log in to the CLI as a user with an access token. The user must also have the **system:image-pruner** cluster role or greater (for example, **cluster-admin**).
- Expose the image registry.

**Procedure**

To manually prune images that are no longer required by the system due to age, status, or exceed limits, use one of the following methods:

- Run image pruning as a **Job** or **CronJob** on the cluster by creating a YAML file for the **pruner** service account, for example:

  ```bash
  $ oc create -f <filename>.yaml
  ```

**Example output**

```
kind: List
apiVersion: v1
items:
  - apiVersion: v1
    kind: ServiceAccount
    metadata:
      name: pruner
      namespace: openshift-image-registry
  - apiVersion: rbac.authorization.k8s.io/v1
    kind: ClusterRoleBinding
    metadata:
      name: openshift-image-registry-pruner
      roleRef:
        apiGroup: rbac.authorization.k8s.io
        kind: ClusterRole
        name: system:image-pruner
    subjects:
      - kind: ServiceAccount
        name: pruner
        namespace: openshift-image-registry
  - apiVersion: batch/v1
    kind: CronJob
    metadata:
      name: image-pruner
      namespace: openshift-image-registry
    spec:
      schedule: "0 0 * * *"
      concurrencyPolicy: Forbid
      successfulJobsHistoryLimit: 1
      failedJobsHistoryLimit: 3
      jobTemplate:
        spec:
          template:
```
Run the `oc adm prune images [<options>]` command:

```
$ oc adm prune images [<options>]
```

Pruning images removes data from the integrated registry unless `--prune-registry=false` is used.

Pruning images with the `--namespace` flag does not remove images, only image streams. Images are non-namespaced resources. Therefore, limiting pruning to a particular namespace makes it impossible to calculate its current usage.

By default, the integrated registry caches metadata of blobs to reduce the number of requests to storage, and to increase the request-processing speed. Pruning does not update the integrated registry cache. Images that still contain pruned layers after pruning will be broken because the pruned layers that have metadata in the cache will not be pushed. Therefore, you must redeploy the registry to clear the cache after pruning:

```
$ oc rollout restart deployment/image-registry -n openshift-image-registry
```

If the integrated registry uses a Redis cache, you must clean the database manually.

If redeploying the registry after pruning is not an option, then you must permanently disable the cache.

`oc adm prune images` operations require a route for your registry. Registry routes are not created by default.

The Prune images CLI configuration options table describes the options you can use with the `oc adm prune images <options>` command.

Table 12.4. Prune images CLI configuration options
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--all</td>
<td>Include images that were not pushed to the registry, but have been mirrored by pullthrough. This is on by default. To limit the pruning to images that were pushed to the integrated registry, pass --all=false.</td>
</tr>
<tr>
<td>--certificate-authority</td>
<td>The path to a certificate authority file to use when communicating with the OpenShift Container Platform–managed registries. Defaults to the certificate authority data from the current user’s configuration file. If provided, a secure connection is initiated.</td>
</tr>
<tr>
<td>--confirm</td>
<td>Indicate that pruning should occur, instead of performing a test–run. This requires a valid route to the integrated container image registry. If this command is run outside of the cluster network, the route must be provided using --registry-url.</td>
</tr>
<tr>
<td>--force-insecure</td>
<td>Use caution with this option. Allow an insecure connection to the container registry that is hosted via HTTP or has an invalid HTTPS certificate.</td>
</tr>
<tr>
<td>--keep-tag-revisions=&lt;N&gt;</td>
<td>For each imagestream, keep up to at most N image revisions per tag (default 3).</td>
</tr>
<tr>
<td>--keep-younger-than=&lt;duration&gt;</td>
<td>Do not prune any image that is younger than &lt;duration&gt; relative to the current time. Alternately, do not prune any image that is referenced by any other object that is younger than &lt;duration&gt; relative to the current time (default 60m).</td>
</tr>
<tr>
<td>--prune-over-size-limit</td>
<td>Prune each image that exceeds the smallest limit defined in the same project. This flag cannot be combined with --keep-tag-revisions nor --keep-younger-than.</td>
</tr>
<tr>
<td>--registry-url</td>
<td>The address to use when contacting the registry. The command attempts to use a cluster-internal URL determined from managed images and image streams. In case it fails (the registry cannot be resolved or reached), an alternative route that works needs to be provided using this flag. The registry hostname can be prefixed by https:// or http://, which enforces particular connection protocol.</td>
</tr>
<tr>
<td>--prune-registry</td>
<td>In conjunction with the conditions stipulated by the other options, this option controls whether the data in the registry corresponding to the OpenShift Container Platform image API object is pruned. By default, image pruning processes both the image API objects and corresponding data in the registry. This option is useful when you are only concerned with removing etcd content, to reduce the number of image objects but are not concerned with cleaning up registry storage, or if you intend to do that separately by hard pruning the registry during an appropriate maintenance window for the registry.</td>
</tr>
</tbody>
</table>
12.6.1. Image prune conditions

You can apply conditions to your manually pruned images.

- To remove any image managed by OpenShift Container Platform, or images with the annotation `openshift.io/image.managed`:
  - Created at least `--keep-younger-than` minutes ago and are not currently referenced by any:
    - Pods created less than `--keep-younger-than` minutes ago
    - Image streams created less than `--keep-younger-than` minutes ago
    - Running pods
    - Pending pods
    - Replication controllers
    - Deployments
    - Deployment configs
    - Replica sets
    - Build configurations
    - Builds
    - `--keep-tag-revisions` most recent items in `stream.status.tags[].items`
  - That are exceeding the smallest limit defined in the same project and are not currently referenced by any:
    - Running pods
    - Pending pods
    - Replication controllers
    - Deployments
    - Deployment configs
    - Replica sets
    - Build configurations
    - Builds

- There is no support for pruning from external registries.

- When an image is pruned, all references to the image are removed from all image streams that have a reference to the image in `status.tags`.

- Image layers that are no longer referenced by any images are removed.
NOTE

The --prune-over-size-limit flag cannot be combined with the --keep-tag-revisions flag nor the --keep-younger-than flags. Doing so returns information that this operation is not allowed.

Separating the removal of OpenShift Container Platform image API objects and image data from the registry by using --prune-registry=false, followed by hard pruning the registry, can narrow timing windows and is safer when compared to trying to prune both through one command. However, timing windows are not completely removed.

For example, you can still create a pod referencing an image as pruning identifies that image for pruning. You should still keep track of an API object created during the pruning operations that might reference images so that you can mitigate any references to deleted content.

Re-doing the pruning without the --prune-registry option or with --prune-registry=true does not lead to pruning the associated storage in the image registry for images previously pruned by --prune-registry=false. Any images that were pruned with --prune-registry=false can only be deleted from registry storage by hard pruning the registry.

12.6.2. Running the image prune operation

Procedure

1. To see what a pruning operation would delete:
   a. Keeping up to three tag revisions, and keeping resources (images, image streams, and pods) younger than 60 minutes:

   ```
   $ oc adm prune images --keep-tag-revisions=3 --keep-younger-than=60m
   ```

   b. Pruning every image that exceeds defined limits:

   ```
   $ oc adm prune images --prune-over-size-limit
   ```

2. To perform the prune operation with the options from the previous step:

   ```
   $ oc adm prune images --keep-tag-revisions=3 --keep-younger-than=60m --confirm
   $ oc adm prune images --prune-over-size-limit --confirm
   ```

12.6.3. Using secure or insecure connections

The secure connection is the preferred and recommended approach. It is done over HTTPS protocol with a mandatory certificate verification. The prune command always attempts to use it if possible. If it is not possible, in some cases it can fall-back to insecure connection, which is dangerous. In this case, either certificate verification is skipped or plain HTTP protocol is used.

The fall-back to insecure connection is allowed in the following cases unless --certificate-authority is specified:

1. The prune command is run with the --force-insecure option.
2. The provided registry-url is prefixed with the http:// scheme.

3. The provided registry-url is a local-link address or localhost.

4. The configuration of the current user allows for an insecure connection. This can be caused by the user either logging in using --insecure-skip-tls-verify or choosing the insecure connection when prompted.

**IMPORTANT**

If the registry is secured by a certificate authority different from the one used by OpenShift Container Platform, it must be specified using the --certificate-authority flag. Otherwise, the prune command fails with an error.

12.6.4. Image pruning problems

**Images not being pruned**

If your images keep accumulating and the prune command removes just a small portion of what you expect, ensure that you understand the image prune conditions that must apply for an image to be considered a candidate for pruning.

Ensure that images you want removed occur at higher positions in each tag history than your chosen tag revisions threshold. For example, consider an old and obsolete image named sha:abz. By running the following command in namespace N, where the image is tagged, the image is tagged three times in a single image stream named myapp:

```bash
$ oc get is -n N -o go-template='{{range $isi, $is := .items}}{{range $ti, $tag := $is.status.tags}}{{range $ii, $item := $tag.items}}{{if eq $item.image "sha:abz"}}{{$is.metadata.name}}:{{$tag.tag}} at position {{$ii}} out of {{len $tag.items}}
{{end}}{{end}}{{end}}{{end}}'
```

**Example output**

```text
myapp:v2 at position 4 out of 5
myapp:v2.1 at position 2 out of 2
myapp:v2.1-may-2016 at position 0 out of 1
```

When default options are used, the image is never pruned because it occurs at position 0 in a history of myapp:v2.1-may-2016 tag. For an image to be considered for pruning, the administrator must either:

- Specify --keep-tag-revisions=0 with the oc adm prune images command.

**WARNING**

This action removes all the tags from all the namespaces with underlying images, unless they are younger or they are referenced by objects younger than the specified threshold.
- Delete all the istags where the position is below the revision threshold, which means myapp:v2.1 and myapp:v2.1-may-2016.

- Move the image further in the history, either by running new builds pushing to the same istag, or by tagging other image. This is not always desirable for old release tags.

Tags having a date or time of a particular image’s build in their names should be avoided, unless the image must be preserved for an undefined amount of time. Such tags tend to have just one image in their history, which prevents them from ever being pruned.

**Using a secure connection against insecure registry**

If you see a message similar to the following in the output of the `oc adm prune images` command, then your registry is not secured and the `oc adm prune images` client attempts to use a secure connection:

```
```

- The recommended solution is to secure the registry. Otherwise, you can force the client to use an insecure connection by appending `--force-insecure` to the command; however, this is not recommended.

**Using an insecure connection against a secured registry**

If you see one of the following errors in the output of the `oc adm prune images` command, it means that your registry is secured using a certificate signed by a certificate authority other than the one used by `oc adm prune images` client for connection verification:

```
error: error communicating with registry: Get http://172.30.30.30:5000/healthz: malformed HTTP response "x15\x03\x01\x00\x02\x02"
error: error communicating with registry: [Get https://172.30.30.5000/healthz: x509: certificate signed by unknown authority, Get http://172.30.30.5000/healthz: malformed HTTP response "x15\x03\x01\x00\x02\x02"]
```

By default, the certificate authority data stored in the user’s configuration files is used; the same is true for communication with the master API.

Use the `--certificate-authority` option to provide the right certificate authority for the container image registry server.

**Using the wrong certificate authority**

The following error means that the certificate authority used to sign the certificate of the secured container image registry is different from the authority used by the client:

```
error: error communicating with registry: Get https://172.30.30.5000/: x509: certificate signed by unknown authority
```

Make sure to provide the right one with the flag `--certificate-authority`.

As a workaround, the `--force-insecure` flag can be added instead. However, this is not recommended.

**Additional resources**

- Accessing the registry
- Exposing the registry
12.7. HARD PRUNING THE REGISTRY

The OpenShift Container Registry can accumulate blobs that are not referenced by the OpenShift Container Platform cluster’s etcd. The basic pruning images procedure, therefore, is unable to operate on them. These are called orphaned blobs.

Orphaned blobs can occur from the following scenarios:

- Manually deleting an image with `oc delete image <sha256:image-id>` command, which only removes the image from etcd, but not from the registry’s storage.
- Pushing to the registry initiated by daemon failures, which causes some blobs to get uploaded, but the image manifest (which is uploaded as the very last component) does not. All unique image blobs become orphans.
- OpenShift Container Platform refusing an image because of quota restrictions.
- The standard image pruner deleting an image manifest, but is interrupted before it deletes the related blobs.
- A bug in the registry pruner, which fails to remove the intended blobs, causing the image objects referencing them to be removed and the blobs becoming orphans.

Hard pruning the registry, a separate procedure from basic image pruning, allows cluster administrators to remove orphaned blobs. You should hard prune if you are running out of storage space in your OpenShift Container Registry and believe you have orphaned blobs.

This should be an infrequent operation and is necessary only when you have evidence that significant numbers of new orphans have been created. Otherwise, you can perform standard image pruning at regular intervals, for example, once a day (depending on the number of images being created).

Procedure

To hard prune orphaned blobs from the registry:

1. **Log in.**
   Log in to the cluster with the CLI as `kubeadmin` or another privileged user that has access to the `openshift-image-registry` namespace.

2. **Run a basic image prune**
   Basic image pruning removes additional images that are no longer needed. The hard prune does not remove images on its own. It only removes blobs stored in the registry storage. Therefore, you should run this just before the hard prune.

3. **Switch the registry to read-only mode.**
   If the registry is not running in read-only mode, any pushes happening at the same time as the prune will either:
   - fail and cause new orphans, or
   - succeed although the images cannot be pulled (because some of the referenced blobs were deleted).
Pushes will not succeed until the registry is switched back to read-write mode. Therefore, the hard prune must be carefully scheduled.

To switch the registry to read-only mode:

a. In `configs.imageregistry.operator.openshift.io/cluster`, set `spec.readOnly` to `true`:

```
$ oc patch configs.imageregistry.operator.openshift.io/cluster -p '{"spec":
{"readOnly":true}}' --type=merge
```

4. **Add the `system:image-pruner` role.**
The service account used to run the registry instances requires additional permissions to list some resources.

a. Get the service account name:

```
$ service_account=$(oc get -n openshift-image-registry \
-o jsonpath='{.spec.template.spec.serviceAccountName}' deploy/image-registry)
```

b. Add the `system:image-pruner` cluster role to the service account:

```
$ oc adm policy add-cluster-role-to-user \
 system:image-pruner -z \n ${service_account} -n openshift-image-registry
```

5. **Optional: Run the pruner in dry-run mode.**
To see how many blobs would be removed, run the hard pruner in dry-run mode. No changes are actually made. The following example references an image registry pod called `image-registry-3-vhndw`:

```
$ oc -n openshift-image-registry exec pod/image-registry-3-vhndw -- /bin/sh -c \\
'REGISTRY_LOG_LEVEL=info /usr/bin/dockerregistry -prune=check'
```

Alternatively, to get the exact paths for the prune candidates, increase the logging level:

```
$ oc -n openshift-image-registry exec pod/image-registry-3-vhndw -- /bin/sh -c \\
'REGISTRY_LOG_LEVEL=debug /usr/bin/dockerregistry -prune=check'
```

**Example output**
```
time="2017-06-22T11:50:25.066156047Z" level=info msg="start prune (dry-run mode)"
distribution_version="v2.4.1+unknown" kubernetes_version=v1.6.1+$Format:%h$
openshift_version=unknown
time="2017-06-22T11:50:25.092257421Z" level=info msg="Would delete blob: sha256:00043a2a5e384f6b59ab17e2c3d3a3d0a7de01b2cabebe606243e468acc663fa5"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:25.092492183Z" level=info msg="Would delete blob: sha256:0022d49612807cb348cabc562c072e34d756adfe0100a61952cbbcb87ee6578a"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:25.092492183Z" level=info msg="Would delete blob: sha256:0029dd4228961086707e53b881e25e0a564fa80033fbbbe27847a28d16a37c"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:26.673946639Z" level=info msg="Would delete blob: sha256:ff7664dfc213d6cc60fd5c5f5bb00a7bf4a687e18e1df12d349a1d07b2cf7663"
```
6. Run the hard prune.
Execute the following command inside one running instance of a `image-registry` pod to run the hard prune. The following example references an image registry pod called `image-registry-3-vhndw`:

```
$ oc -n openshift-image-registry exec pod/image-registry-3-vhndw -- /bin/sh -c '
/usr/bin/dockerregistry -prune=delete'
```

Example output

Deleted 13374 blobs
Freed up 2.835 GiB of disk space

7. Switch the registry back to read-write mode.
After the prune is finished, the registry can be switched back to read-write mode. In `configs.imageregistry.operator.openshift.io/cluster`, set `spec.readOnly` to `false`:

```
$ oc patch configs.imageregistry.operator.openshift.io/cluster -p ' nuestras acompaniar de"spec":{"readOnly":false}}' -
-type=merge
```

12.8. PRUNING CRON JOBS

Cron jobs can perform pruning of successful jobs, but might not properly handle failed jobs. Therefore, the cluster administrator should perform regular cleanup of jobs manually. They should also restrict the access to cron jobs to a small group of trusted users and set appropriate quota to prevent the cron job from creating too many jobs and pods.

Additional resources

- Running tasks in pods using jobs
- Resource quotas across multiple projects
- Using RBAC to define and apply permissions
CHAPTER 13. IDLING APPLICATIONS

Cluster administrators can idle applications to reduce resource consumption. This is useful when the cluster is deployed on a public cloud where cost is related to resource consumption.

If any scalable resources are not in use, OpenShift Container Platform discovers and idles them by scaling their replicas to 0. The next time network traffic is directed to the resources, the resources are unidled by scaling up the replicas, and normal operation continues.

Applications are made of services, as well as other scalable resources, such as deployment configs. The action of idling an application involves idling all associated resources.

13.1. IDLING APPLICATIONS

Idling an application involves finding the scalable resources (deployment configurations, replication controllers, and others) associated with a service. Idling an application finds the service and marks it as idled, scaling down the resources to zero replicas.

You can use the `oc idle` command to idle a single service, or use the `--resource-names-file` option to idle multiple services.

13.1.1. Idling a single service

Procedure

1. To idle a single service, run:

   $ oc idle <service>

13.1.2. Idling multiple services

Idling multiple services is helpful if an application spans across a set of services within a project, or when idling multiple services in conjunction with a script to idle multiple applications in bulk within the same project.

Procedure

1. Create a file containing a list of the services, each on their own line.

2. Idle the services using the `--resource-names-file` option:

   $ oc idle --resource-names-file <filename>

   NOTE

   The `idle` command is limited to a single project. For idling applications across a cluster, run the `idle` command for each project individually.

13.2. UNIDLING APPLICATIONS

Application services become active again when they receive network traffic and are scaled back up their previous state. This includes both traffic to the services and traffic passing through routes.
Applications can also be manually unidled by scaling up the resources.

**Procedure**

1. To scale up a DeploymentConfig, run:
   ```bash
   $ oc scale --replicas=1 dc <dc_name>
   ```

**NOTE**

Automatic unidling by a router is currently only supported by the default HAProxy router.

**NOTE**

Services do not support automatic unidling if you configure Kuryr-Kubernetes as an SDN.
CHAPTER 14. DELETING APPLICATIONS

You can delete applications created in your project.

14.1. DELETING APPLICATIONS USING THE DEVELOPER PERSPECTIVE

You can delete an application and all of its associated components using the Topology view in the Developer perspective:

1. Click the application you want to delete to see the side panel with the resource details of the application.

2. Click the Actions drop-down menu displayed on the upper right of the panel, and select Delete Application to see a confirmation dialog box.

3. Enter the name of the application and click Delete to delete it.

You can also right-click the application you want to delete and click Delete Application to delete it.
CHAPTER 15. USING THE RED HAT MARKETPLACE

The Red Hat Marketplace is an open cloud marketplace that makes it easy to discover and access certified software for container-based environments that run on public clouds and on-premises.

15.1. RED HAT MARKETPLACE FEATURES

Cluster administrators can use the Red Hat Marketplace to manage software on OpenShift Container Platform, give developers self-service access to deploy application instances, and correlate application usage against a quota.

15.1.1. Connect OpenShift Container Platform clusters to the Marketplace

Cluster administrators can install a common set of applications on OpenShift Container Platform clusters that connect to the Marketplace. They can also use the Marketplace to track cluster usage against subscriptions or quotas. Users that they add by using the Marketplace have their product usage tracked and billed to their organization.

During the cluster connection process, a Marketplace Operator is installed that updates the image registry secret, manages the catalog, and reports application usage.

15.1.2. Install applications

Cluster administrators can install Marketplace applications from within OperatorHub in OpenShift Container Platform, or from the Marketplace web application.

You can access installed applications from the web console by clicking Operators > Installed Operators.

15.1.3. Deploy applications from different perspectives

You can deploy Marketplace applications from the web console’s Administrator and Developer perspectives.

The Developer perspective
Developers can access newly installed capabilities by using the Developer perspective.

For example, after a database Operator is installed, a developer can create an instance from the catalog within their project. Database usage is aggregated and reported to the cluster administrator.

This perspective does not include Operator installation and application usage tracking.

The Administrator perspective
Cluster administrators can access Operator installation and application usage information from the Administrator perspective.

They can also launch application instances by browsing custom resource definitions (CRDs) in the Installed Operators list.