Red Hat OpenShift Dev Spaces 3.5

Administration guide

Administering Red Hat OpenShift Dev Spaces 3.5

Last Updated: 2023-05-31

Red Hat OpenShift Dev Spaces 3.5 Administration guide

Administering Red Hat OpenShift Dev Spaces 3.5

Robert Kratky
rkratky@redhat.com

Fabrice Flore-Thébault
ffloreth@redhat.com

Jana Vrbkova
jvrbkova@redhat.com

Max Leonov
mleonov@redhat.com

Legal Notice

Abstract

Information for administrators operating Red Hat OpenShift Dev Spaces.

Table of Contents

CHAPTER 1. PREPARING THE INSTALLATION

To prepare a OpenShift Dev Spaces installation, learn about OpenShift Dev Spaces ecosystem and deployment constraints:

- Section 1.1, “Supported platforms”
- Section 1.2, “Architecture”
- Section 1.3, “Calculating Dev Spaces resource requirements”
- Section 3.1, “Understanding the **CheCluster** Custom Resource”

1.1. SUPPORTED PLATFORMS

OpenShift Dev Spaces runs on OpenShift 4.10–4.12 on the following CPU architectures:

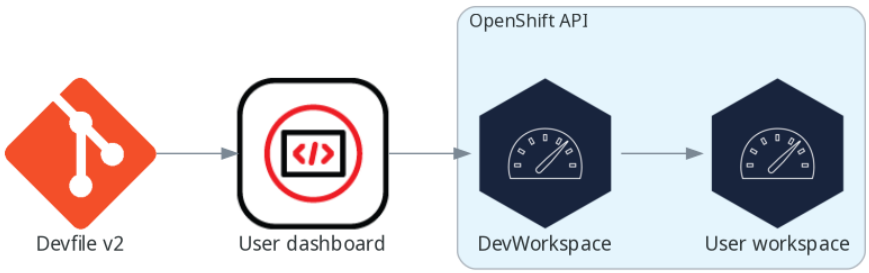
- AMD64 and Intel 64 (**x86_64**)
- IBM Power (**ppc64le**) and IBM Z (**s390x**)

Additional resources

- OpenShift Documentation

1.2. ARCHITECTURE

Figure 1.1. High-level OpenShift Dev Spaces architecture with the Dev Workspace operator



OpenShift Dev Spaces runs on three groups of components:

OpenShift Dev Spaces server components

Manage User project and workspaces. The main component is the User dashboard, from which users control their workspaces.

Dev Workspace operator

Creates and controls the necessary OpenShift objects to run User workspaces. Including **Pods**, **Services**, and **PersistentVolumes**.

User workspaces

Container-based development environments, the IDE included.

The role of these OpenShift features is central:

Dev Workspace Custom Resources

Valid OpenShift objects representing the User workspaces and manipulated by OpenShift Dev Spaces. It is the communication channel for the three groups of components.

OpenShift role-based access control (RBAC)

Controls access to all resources.

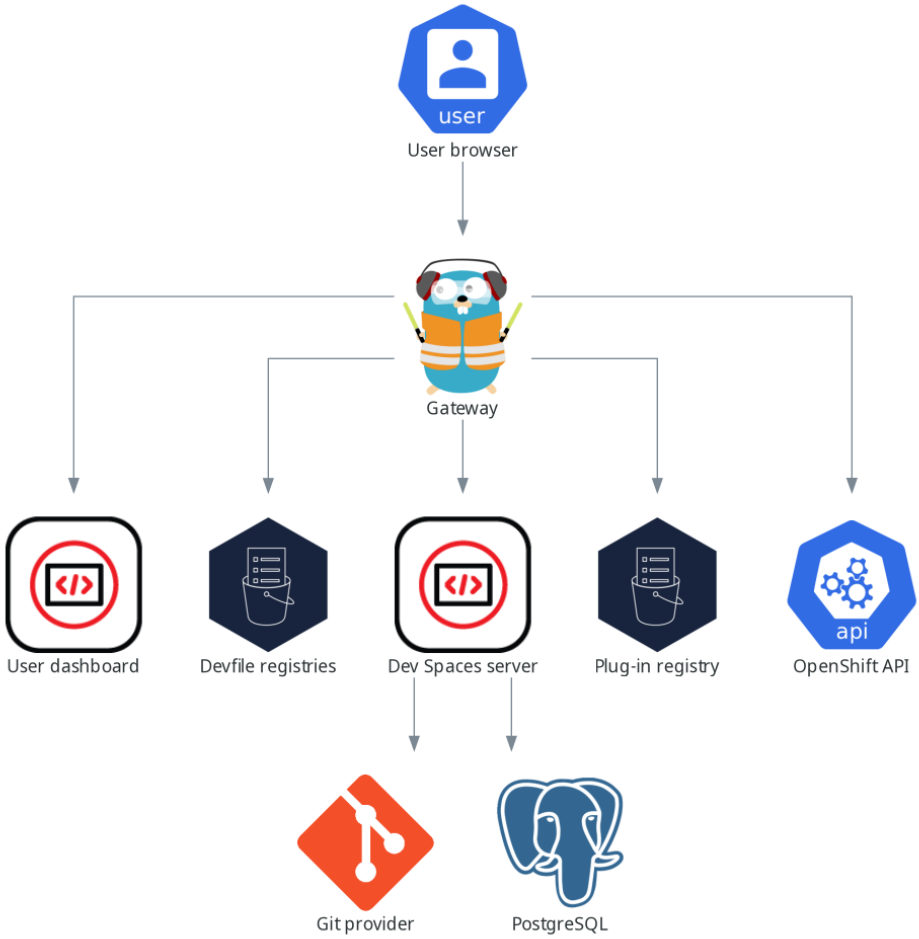
Additional resources

- Section 1.2.1, “Server components”
- Section 1.2.1.2, “Dev Workspace operator”
- Section 1.2.2, “User workspaces”
- Dev Workspace Operator repository
- Kubernetes documentation - Custom Resources

1.2.1. Server components

The OpenShift Dev Spaces server components ensure multi-tenancy and workspaces management.

Figure 1.2. OpenShift Dev Spaces server components interacting with the Dev Workspace operator



Additional resources

- Section 1.2.1.1, “Dev Spaces operator”
- Section 1.2.1.2, “Dev Workspace operator”
- Section 1.2.1.3, “Gateway”
- Section 1.2.1.4, “User dashboard”
- Section 1.2.1.5, “Devfile registries”
- Section 1.2.1.6, “Dev Spaces server”
- Section 1.2.1.7, “PostgreSQL”

- Section 1.2.1.8, “Plug-in registry”

1.2.1.1. Dev Spaces operator

The OpenShift Dev Spaces operator ensure full lifecycle management of the OpenShift Dev Spaces server components. It introduces:

CheCluster custom resource definition (CRD)

Defines the **CheCluster** OpenShift object.

OpenShift Dev Spaces controller

Creates and controls the necessary OpenShift objects to run a OpenShift Dev Spaces instance, such as pods, services, and persistent volumes.

CheCluster custom resource (CR)

On a cluster with the OpenShift Dev Spaces operator, it is possible to create a **CheCluster** custom resource (CR). The OpenShift Dev Spaces operator ensures the full lifecycle management of the OpenShift Dev Spaces server components on this OpenShift Dev Spaces instance:

- Section 1.2.1.2, “Dev Workspace operator”
- Section 1.2.1.3, “Gateway”
- Section 1.2.1.4, “User dashboard”
- Section 1.2.1.5, “Devfile registries”
- Section 1.2.1.6, “Dev Spaces server”
- Section 1.2.1.7, “PostgreSQL”
- Section 1.2.1.8, “Plug-in registry”

Additional resources

- Section 3.1, “Understanding the **CheCluster** Custom Resource”
- Chapter 2, *Installing Dev Spaces*

1.2.1.2. Dev Workspace operator

The Dev Workspace operator extends OpenShift to provide Dev Workspace support. It introduces:

Dev Workspace custom resource definition

Defines the Dev Workspace OpenShift object from the Devfile v2 specification.

Dev Workspace controller

Creates and controls the necessary OpenShift objects to run a Dev Workspace, such as pods, services, and persistent volumes.

Dev Workspace custom resource

On a cluster with the Dev Workspace operator, it is possible to create Dev Workspace custom resources (CR). A Dev Workspace CR is a OpenShift representation of a Devfile. It defines a User workspaces in a OpenShift cluster.

Additional resources

- Devfile API repository

1.2.1.3. Gateway

The OpenShift Dev Spaces gateway has following roles:

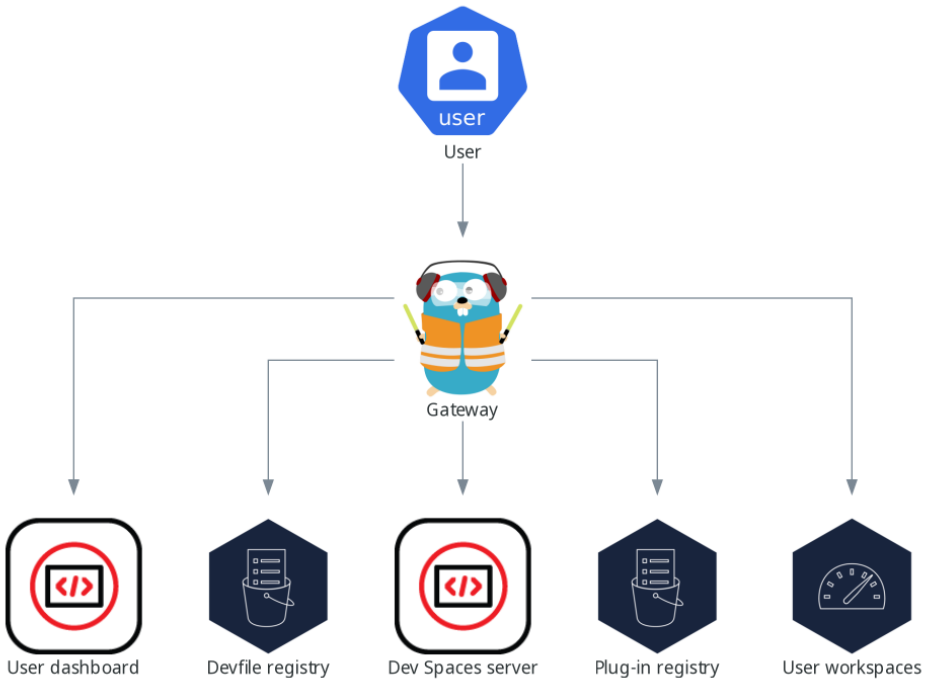
- Routing requests. It uses Traefik.
- Authenticating users with OpenID Connect (OIDC). It uses OpenShift OAuth2 proxy.
- Applying OpenShift Role based access control (RBAC) policies to control access to any OpenShift Dev Spaces resource. It uses `kube-rbac-proxy`.

The OpenShift Dev Spaces operator manages it as the **che-gateway** Deployment.

It controls access to:

- Section 1.2.1.4, “User dashboard”
- Section 1.2.1.5, “Devfile registries”
- Section 1.2.1.6, “Dev Spaces server”
- Section 1.2.1.8, “Plug-in registry”
- Section 1.2.2, “User workspaces”

Figure 1.3. OpenShift Dev Spaces gateway interactions with other components



Additional resources

- Section 3.9, “Managing identities and authorizations”

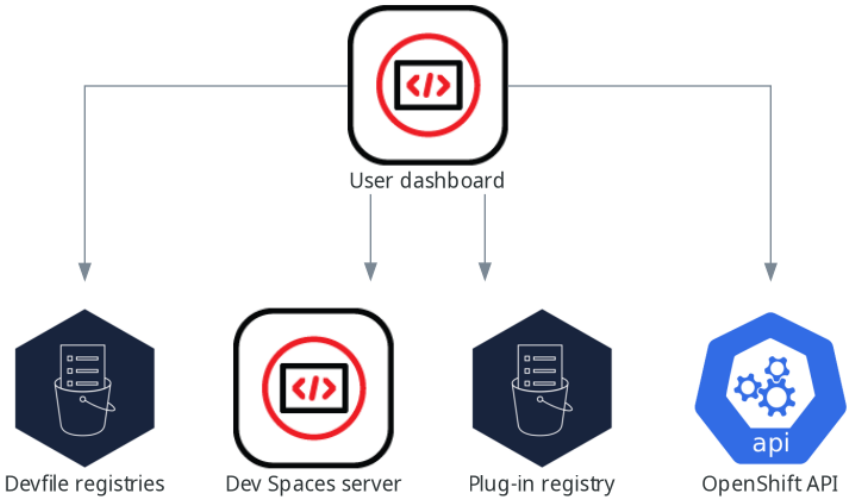
1.2.1.4. User dashboard

The user dashboard is the landing page of Red Hat OpenShift Dev Spaces. OpenShift Dev Spaces users browse the user dashboard to access and manage their workspaces. It is a React application. The OpenShift Dev Spaces deployment starts it in the **devspaces-dashboard** Deployment.

It need access to:

- Section 1.2.1.5, “Devfile registries”
- Section 1.2.1.6, “Dev Spaces server”
- Section 1.2.1.8, “Plug-in registry”
- OpenShift API

Figure 1.4. User dashboard interactions with other components



When the user requests the user dashboard to start a workspace, the user dashboard executes this sequence of actions:

1. Collects the devfile from the Section 1.2.1.5, "Devfile registries", when the user is creating a workspace from a code sample.
2. Sends the repository URL to Section 1.2.1.6, "Dev Spaces server" and expects a devfile in return, when the user is creating a workspace from a remote devfile.
3. Reads the devfile describing the workspace.
4. Collects the additional metadata from the Section 1.2.1.8, "Plug-in registry".
5. Converts the information into a Dev Workspace Custom Resource.
6. Creates the Dev Workspace Custom Resource in the user project using the OpenShift API.
7. Watches the Dev Workspace Custom Resource status.
8. Redirects the user to the running workspace IDE.

1.2.1.5. Devfile registries

Additional resources

The OpenShift Dev Spaces devfile registries are services providing a list of sample devfiles to create ready-to-use workspaces. The Section 1.2.1.4, "User dashboard" displays the samples list on the

Dashboard → Create Workspace page. Each sample includes a Devfile v2. The OpenShift Dev Spaces deployment starts one devfile registry instance in the **devfile-registry** deployment.

Figure 1.5. Devfile registries interactions with other components

Additional resources

- Devfile v2 documentation
- devfile registry latest community version online instance
- OpenShift Dev Spaces devfile registry repository

1.2.1.6. Dev Spaces server

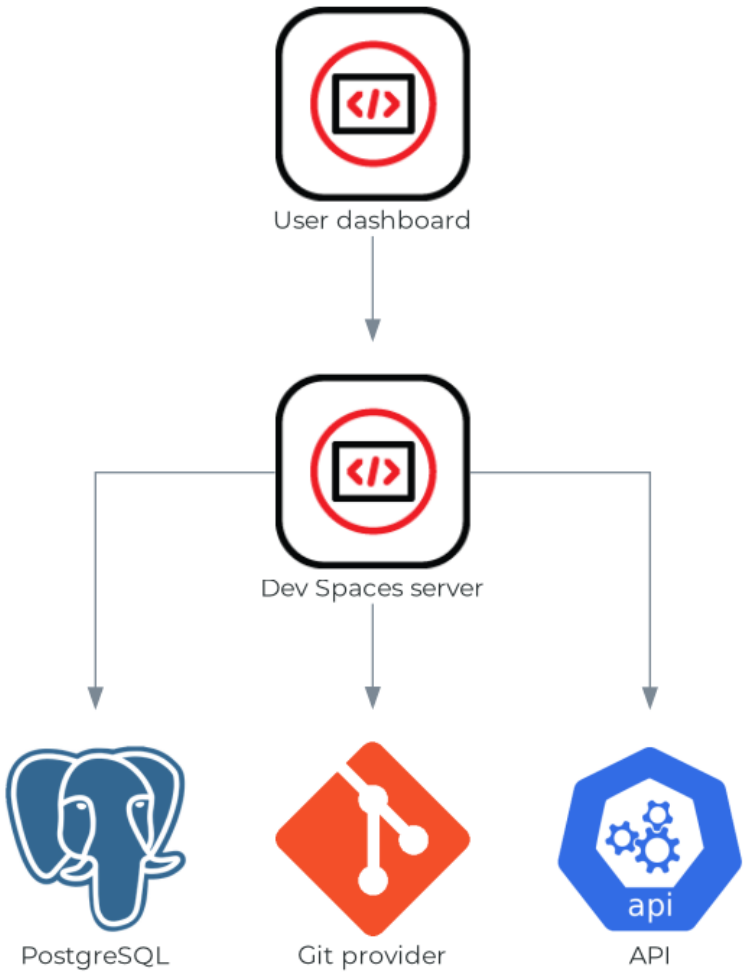
The OpenShift Dev Spaces server main functions are:

- Creating user namespaces.
- Provisioning user namespaces with required secrets and config maps.
- Integrating with Git services providers, to fetch and validate devfiles and authentication.

The OpenShift Dev Spaces server is a Java web service exposing an HTTP REST API and needs access to:

- Section 1.2.1.7, “PostgreSQL”
- Git service providers
- OpenShift API

Figure 1.6. OpenShift Dev Spaces server interactions with other components



Additional resources

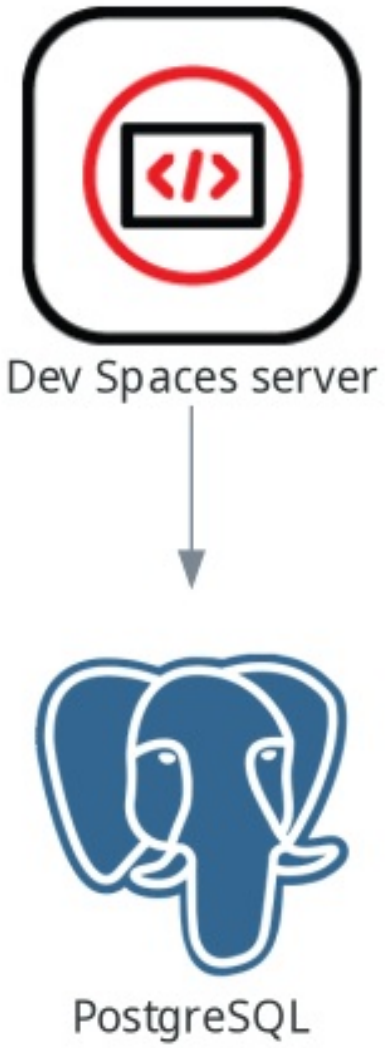
- Section 3.3.2, "Advanced configuration options for Dev Spaces server"

1.2.1.7. PostgreSQL

OpenShift Dev Spaces server uses the PostgreSQL database to persist user configurations such as workspaces metadata.

The OpenShift Dev Spaces deployment starts a dedicated PostgreSQL instance in the **postgres** Deployment. You can use an external database instead.

Figure 1.7. PostgreSQL interactions with other components



Additional resources

- **quay.io/eclipse/che—centos—postgresql-96-centos7** container image
- **quay.io/eclipse/che—centos—postgresql-13-centos7** container image

1.2.1.8. Plug-in registry

Each OpenShift Dev Spaces workspace starts with a specific editor and set of associated extensions. The OpenShift Dev Spaces plugin registry provides the list of available editors and editor extensions. A Devfile v2 describes each editor or extension.

The Section 1.2.1.4, "User dashboard" is reading the content of the registry.

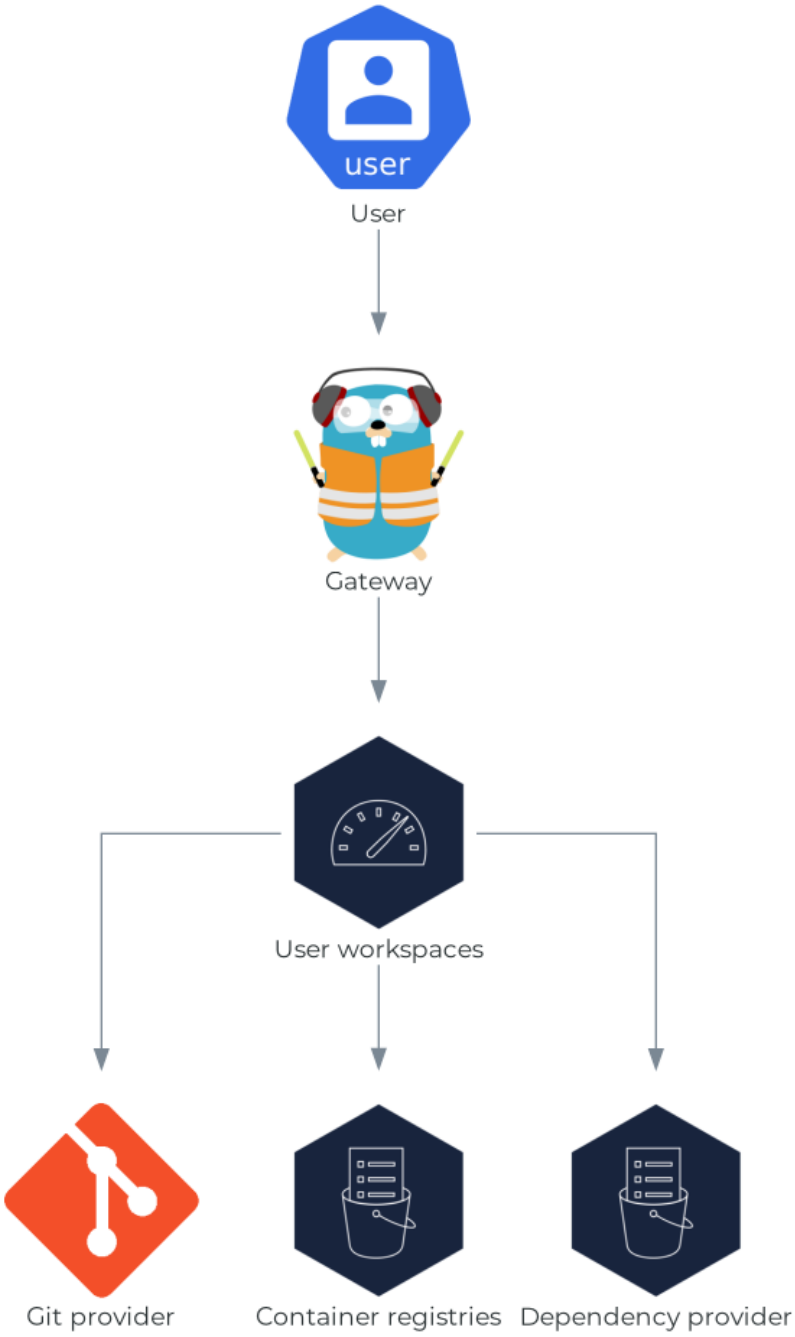
Figure 1.8. Plugin registries interactions with other components

Additional resources

- Editor definitions in the OpenShift Dev Spaces plugin registry repository
- Plugin registry latest community version online instance

1.2.2. User workspaces

Figure 1.9. User workspaces interactions with other components



User workspaces are web IDEs running in containers.

A User workspace is a web application. It consists of microservices running in containers providing all the services of a modern IDE running in your browser:

- Editor
- Language auto-completion
- Language server
- Debugging tools
- Plug-ins
- Application runtimes

A workspace is one OpenShift Deployment containing the workspace containers and enabled plugins, plus related OpenShift components:

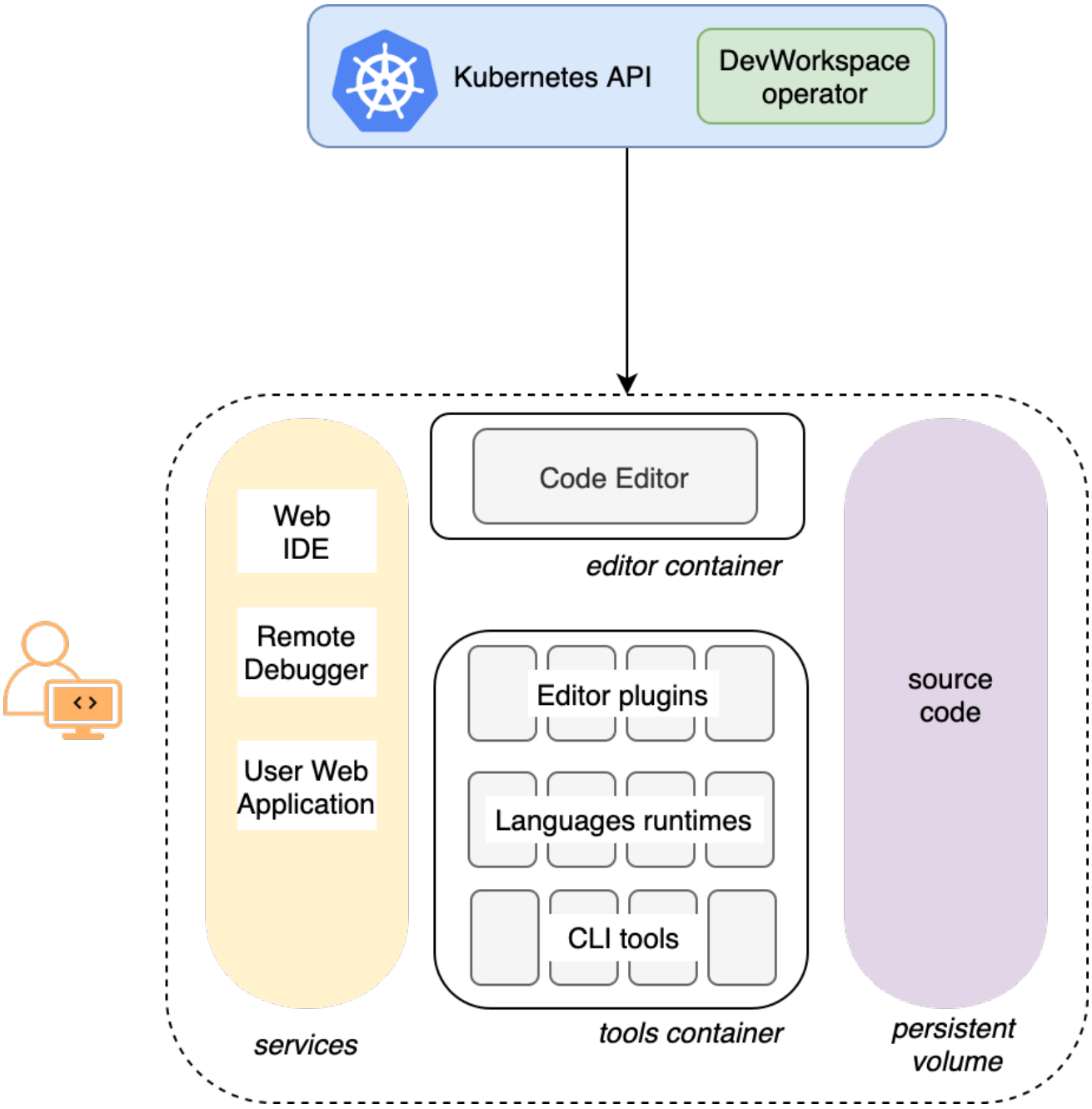
- Containers
- ConfigMaps
- Services
- Endpoints
- Ingresses or Routes
- Secrets
- Persistent Volumes (PV)

A OpenShift Dev Spaces workspace contains the source code of the projects, persisted in a OpenShift Persistent Volume (PV). Microservices have read-write access to this shared directory.

Use the devfile v2 format to specify the tools and runtime applications of a OpenShift Dev Spaces workspace.

The following diagram shows one running OpenShift Dev Spaces workspace and its components.

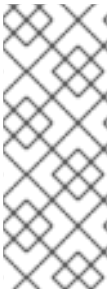
Figure 1.10. OpenShift Dev Spaces workspace components



In the diagram, there is one running workspaces.

1.3. CALCULATING DEV SPACES RESOURCE REQUIREMENTS

The OpenShift Dev Spaces Operator, Dev Workspace Controller, and user workspaces consist of a set of pods. The pods contribute to the resource consumption in CPU and memory limits and requests.



NOTE

The following link to an example devfile is a pointer to material from the upstream community. This material represents the very latest available content and the most recent best practices. These tips have not yet been vetted by Red Hat's QE department, and they have not yet been proven by a wide user group. Please, use this information cautiously. It is best used for educational and 'developmental' purposes rather than 'production' purposes.

Procedure

1. Identify the workspace resource requirements which depend on the devfile that is used for defining the development environment. This includes identifying the workspace components explicitly specified in the **components** section of the devfile.

   - Here is an example devfile with the following components:

     Example 1.1. tools

     The **tools** component of the devfile defines the following requests and limits:

     ```
     memoryLimit: 6G
     memoryRequest: 512M
     cpuRequest: 1000m
     cpuLimit: 4000m
     ```

     Example 1.2. postgresql

     The **postgresql** component does not define any requests and limits and therefore falls back on the defaults for the dedicated container:

     ```
     memoryLimit: 128M
     memoryRequest: 64M
     cpuRequest: 10m
     cpuLimit: 1000m
     ```

   - During the workspace startup, an internal **che-gateway** container is implicitly provisioned with the following requests and limits:

     ```
     memoryLimit: 256M
     memoryRequest: 64M
     cpuRequest: 50m
     cpuLimit: 500m
     ```

2. Calculate the sums of the resources required for each workspace. If you intend to use multiple devfiles, repeat this calculation for every expected devfile.

   Example 1.3. Workspace requirements for theexample devfile in the previous step

Purpose	Pod	Container name	Memory limit	Memory request	CPU limit	CPU request
Developer tools	**workspace**	**tools**	6 GiB	512 MiB	4000 m	1000 m
Database	**workspace**	**postgresql**	128 MiB	64 MiB	1000 m	10 m

Purpose	Pod	Container name	Memory limit	Memory request	CPU limit	CPU request
OpenShift Dev Spaces gateway	**workspace**	**che-gateway**	256 MiB	64 MiB	500 m	50 m
		Total	**6.4 GiB**	**640 MiB**	**5500 m**	**1060 m**

3. Multiply the resources calculated per workspace by the number of workspaces that you expect all of your users to run simultaneously.

4. Calculate the sums of the requirements for the OpenShift Dev Spaces Operator, Operands, and Dev Workspace Controller.

Table 1.1. Default requirements for the OpenShift Dev Spaces Operator, Operands, and Dev Workspace Controller

Purpose	Pod name	Container names	Memory limit	Memory request	CPU limit	CPU request
OpenShift Dev Spaces operator	**devspaces-operator**	**devspaces-operator**	256 MiB	64 MiB	500 m	100 m
OpenShift Dev Spaces Server	**devspaces**	**devspaces-server**	1 GiB	512 MiB	1000 m	100 m
OpenShift Dev Spaces Dashboard	**devspaces-dashboard**	**devspaces-dashboard**	256 MiB	32 MiB	500 m	100 m
OpenShift Dev Spaces Gateway	**devspaces-gateway**	**traefik**	4 GiB	128 MiB	1000 m	100 m
OpenShift Dev Spaces Gateway	**devspaces-gateway**	**configbump**	256 MiB	64 MiB	500 m	50 m

Purpose	Pod name	Container names	Memory limit	Memory request	CPU limit	CPU request
OpenShift Dev Spaces Gateway	**devspaces-gateway**	**oauth-proxy**	512 MiB	64 MiB	500 m	100 m
OpenShift Dev Spaces Gateway	**devspaces-gateway**	**kube-rbac-proxy**	512 MiB	64 MiB	500 m	100 m
Devfile registry	**devfile-registry**	**devfile-registry**	256 MiB	32 MiB	500 m	100 m
Plugin registry	**plugin-registry**	**plugin-registry**	256 MiB	32 MiB	500 m	100 m
PostgreSQL database	**postgres**	**postgres**	1 Gi	512 Mi	500 m	100 m
Dev Workspace Controller Manager	**devworkspace-controller-manager**	**devworkspace-controller**	1 GiB	100 MiB	1000 m	250 m
Dev Workspace Controller Manager	**devworkspace-controller-manager**	**kube-rbac-proxy**	N/A	N/A	N/A	N/A
Dev Workspace webhook server	**devworkspace-webhook-server**	**webhook-server**	300 MiB	20 MiB	200 m	100 m
Dev Workspace Operator Catalog	**devworkspace-operator-catalog**	**registry-server**	N/A	50 MiB	N/A	10 m
Dev Workspace Webhook Server	**devworkspace-webhook-server**	**webhook-server**	300 MiB	20 MiB	200 m	100 m

Purpose	Pod name	Container names	Memory limit	Memory request	CPU limit	CPU request
Dev Workspace Webhook Server	**devworkspace-webhook-server**	**kube-rbac-proxy**	N/A	N/A	N/A	N/A
		Total	**9 GiB**	**1.2 GiB**	**6.9**	**1.3**

Additional resources

- What is a devfile
- Benefits of devfile
- Devfile customization overview

CHAPTER 2. INSTALLING DEV SPACES

This section contains instructions to install Red Hat OpenShift Dev Spaces.

You can deploy only one instance of OpenShift Dev Spaces per cluster.

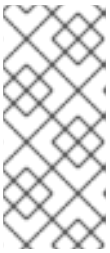
- Section 2.3, “Installing Dev Spaces on OpenShift using the web console”
- Section 2.2, “Installing Dev Spaces on OpenShift using CLI”
- Section 2.4, “Installing Dev Spaces in a restricted environment”

2.1. INSTALLING THE DSC MANAGEMENT TOOL

You can install **dsc**, the Red Hat OpenShift Dev Spaces command-line management tool, on Microsoft Windows, Apple MacOS, and Linux. With **dsc**, you can perform operations the OpenShift Dev Spaces server such as starting, stopping, updating, and deleting the server.

Prerequisites

- Linux or macOS.



NOTE

For installing **dsc** on Windows, see the following pages:

- https://developers.redhat.com/products/openshift-dev-spaces/download
- https://github.com/redhat-developer/devspaces-chectl

Procedure

1. Download the archive from https://developers.redhat.com/products/openshift-dev-spaces/download to a directory such as **$HOME**.
2. Run **tar xvzf** on the archive to extract the **/dsc** directory.
3. Add the extracted **/dsc/bin** subdirectory to **$PATH**.

Verification

- Run **dsc** to view information about it.

```
$ dsc
```

Additional resources

- "dsc reference documentation"

2.2. INSTALLING DEV SPACES ON OPENSHIFT USING CLI

You can install OpenShift Dev Spaces on OpenShift.

Prerequisites

- OpenShift Container Platform
- An active **oc** session with administrative permissions to the OpenShift cluster. See Getting started with the OpenShift CLI.
- **dsc**. See: Section 2.1, "Installing the dsc management tool" .

Procedure

1. Optional: If you previously deployed OpenShift Dev Spaces on this OpenShift cluster, ensure that the previous OpenShift Dev Spaces instance is removed:

```
$ dsc server:delete
```

2. Create the OpenShift Dev Spaces instance:

```
$ dsc server:deploy --platform openshift
```

Verification steps

1. Verify the OpenShift Dev Spaces instance status:

```
$ dsc server:status
```

2. Navigate to the OpenShift Dev Spaces cluster instance:

```
$ dsc dashboard:open
```

2.3. INSTALLING DEV SPACES ON OPENSHIFT USING THE WEB CONSOLE

This section describes how to install OpenShift Dev Spaces using the OpenShift web console. Consider Section 2.2, "Installing Dev Spaces on OpenShift using CLI" instead.

Prerequisites

- An OpenShift web console session by a cluster administrator. See Accessing the web console.

Procedure

1. Optional: If you previously deployed OpenShift Dev Spaces on this OpenShift cluster, ensure that the previous OpenShift Dev Spaces instance is removed:

```
$ dsc server:delete
```

2. Install the Red Hat OpenShift Dev Spaces Operator. See Installing from OperatorHub using the web console.
3. Create the **openshift-devspaces** project in OpenShift as follows:

```
oc create namespace openshift-devspaces
```

4. In the **Administrator** view of the OpenShift web console, go to **Operators → Installed Operators → Red Hat OpenShift Dev Spaces instance Specification→ Create CheCluster → YAML view**.

5. In the **YAML view**, replace **namespace: openshift-operators** with **namespace: openshift-devspaces**.

6. Select **Create**. See Creating applications from installed Operators .

Verification

1. To verify that the OpenShift Dev Spaces instance has installed correctly, navigate to the **Dev Spaces Cluster** tab of the **Operator details** page. The **Red Hat OpenShift Dev Spaces instance Specification** page displays the list of Red Hat OpenShift Dev Spaces instances and their status.

2. Click **devspaces CheCluster** and navigate to the **Details** tab.

3. See the content of the following fields:

   - The **Message** field contains error messages. The expected content is **None**.

   - The **Red Hat OpenShift Dev Spaces URL**field contains the URL of the Red Hat OpenShift Dev Spaces instance. The URL appears when the deployment finishes successfully.

4. Navigate to the **Resources** tab. View the list of resources assigned to the OpenShift Dev Spaces deployment and their status.

2.4. INSTALLING DEV SPACES IN A RESTRICTED ENVIRONMENT

On an OpenShift cluster operating in a restricted network, public resources are not available.

However, deploying OpenShift Dev Spaces and running workspaces requires the following public resources:

- Operator catalog
- Container images
- Sample projects

To make these resources available, you can replace them with their copy in a registry accessible by the OpenShift cluster.

Prerequisites

- The OpenShift cluster has at least 64 GB of disk space.
- The OpenShift cluster is ready to operate on a restricted network, and the OpenShift control plane has access to the public internet. See About disconnected installation mirroring and Using Operator Lifecycle Manager on restricted networks.
- An active **oc** session with administrative permissions to the OpenShift cluster. See Getting started with the OpenShift CLI.

- An active **oc registry** session to the **registry.redhat.io** Red Hat Ecosystem Catalog. See: Red Hat Container Registry authentication.
- **opm**. See Installing the **opm** CLI.
- **jq**. See Downloading **jq**.
- **podman**. See Installing Podman.
- An active **skopeo** session with administrative access to the *<my_registry>* registry. See Installing Skopeo, Authenticating to a registry , and Mirroring images for a disconnected installation .
- **dsc** for OpenShift Dev Spaces version 3.5. See Section 2.1, "Installing the dsc management tool".

Procedure

1. Download and execute the mirroring script to install a custom Operator catalog and mirror the related images: prepare-restricted-environment.sh.

```
$ bash prepare-restricted-environment.sh \
  --ocp_ver "4.12" \
  --devworkspace_operator_index "registry.redhat.io/redhat/redhat-operator-index:v4.12" \
  --devworkspace_operator_version "v0.19.0" \
  --prod_operator_index "registry.redhat.io/redhat/redhat-operator-index:v4.12" \
  --prod_operator_package_name "devspaces-operator" \
  --prod_operator_version "v3.5.0" \
  --my_registry "<my_registry>" \
  --my_catalog "<my_catalog>"
```

2. Install OpenShift Dev Spaces with the configuration set in the **che-operator-cr-patch.yaml** during the previous step:

```
$ dsc server:deploy --platform=openshift \
  --che-operator-cr-patch-yaml=che-operator-cr-patch.yaml
```

3. Allow incoming traffic from the OpenShift Dev Spaces namespace to all Pods in the user projects. See: Section 3.7.1, "Configuring network policies".

Additional resources

- Red Hat-provided Operator catalogs
- Managing custom catalogs

CHAPTER 3. CONFIGURING DEV SPACES

This section describes configuration methods and options for Red Hat OpenShift Dev Spaces.

3.1. UNDERSTANDING THE CHECLUSTER CUSTOM RESOURCE

A default deployment of OpenShift Dev Spaces consists of a **CheCluster** Custom Resource parameterized by the Red Hat OpenShift Dev Spaces Operator.

The **CheCluster** Custom Resource is a Kubernetes object. You can configure it by editing the **CheCluster** Custom Resource YAML file. This file contains sections to configure each component: **devWorkspace**, **cheServer**, **pluginRegistry**, **devfileRegistry**, **database**, **dashboard** and **imagePuller**.

The Red Hat OpenShift Dev Spaces Operator translates the **CheCluster** Custom Resource into a config map usable by each component of the OpenShift Dev Spaces installation.

The OpenShift platform applies the configuration to each component, and creates the necessary Pods. When OpenShift detects changes in the configuration of a component, it restarts the Pods accordingly.

Example 3.1. Configuring the main properties of the OpenShift Dev Spaces server component

1. Apply the **CheCluster** Custom Resource YAML file with suitable modifications in the **cheServer** component section.
2. The Operator generates the **che ConfigMap**.
3. OpenShift detects changes in the **ConfigMap** and triggers a restart of the OpenShift Dev Spaces Pod.

Additional resources

- Understanding Operators
- "Understanding Custom Resources"

3.1.1. Using dsc to configure the CheCluster Custom Resource during installation

To deploy OpenShift Dev Spaces with a suitable configuration, edit the **CheCluster** Custom Resource YAML file during the installation of OpenShift Dev Spaces. Otherwise, the OpenShift Dev Spaces deployment uses the default configuration parameterized by the Operator.

Prerequisites

- An active **oc** session with administrative permissions to the OpenShift cluster. See Getting started with the CLI.
- **dsc**. See: Section 2.1, “Installing the dsc management tool”.

Procedure

- Create a **che-operator-cr-patch.yaml** YAML file that contains the subset of the **CheCluster** Custom Resource to configure:

```
spec:
  <component>:
    <property_to_configure>: <value>
```

- Deploy OpenShift Dev Spaces and apply the changes described in **che-operator-cr-patch.yaml** file:

```
$ dsc server:deploy \
--che-operator-cr-patch-yaml=che-operator-cr-patch.yaml \
--platform <chosen_platform>
```

Verification

1. Verify the value of the configured property:

```
$ oc get configmap che -o jsonpath='{.data.<configured_property>}' \
-n openshift-devspaces
```

Additional resources

- Section 3.1.3, "**CheCluster** Custom Resource fields reference".
- Section 3.3.2, "Advanced configuration options for Dev Spaces server" .

3.1.2. Using the CLI to configure the CheCluster Custom Resource

To configure a running instance of OpenShift Dev Spaces, edit the **CheCluster** Custom Resource YAML file.

Prerequisites

- An instance of OpenShift Dev Spaces on OpenShift.
- An active **oc** session with administrative permissions to the destination OpenShift cluster. See Getting started with the CLI.

Procedure

1. Edit the CheCluster Custom Resource on the cluster:

```
$ oc edit checluster/devspaces -n openshift-devspaces
```

2. Save and close the file to apply the changes.

Verification

1. Verify the value of the configured property:

```
$ oc get configmap che -o jsonpath='{.data.<configured_property>}' \
-n openshift-devspaces
```

Additional resources

- Section 3.1.3, "**CheCluster** Custom Resource fields reference".
- Section 3.3.2, "Advanced configuration options for Dev Spaces server" .

3.1.3. CheCluster Custom Resource fields reference

This section describes all fields available to customize the **CheCluster** Custom Resource.

- Example 3.2, "A minimal **CheCluster** Custom Resource example."
- Table 3.1, "Development environment configuration options."
- Table 3.4, "OpenShift Dev Spaces components configuration."
- Table 3.5, "DevWorkspace operator component configuration."
- Table 3.6, "General configuration settings related to the OpenShift Dev Spaces server component."
- Table 3.7, "Configuration settings related to the Plug-in registry component used by the OpenShift Dev Spaces installation."
- Table 3.8, "Configuration settings related to the Devfile registry component used by the OpenShift Dev Spaces installation."
- Table 3.9, "Configuration settings related to the Database component used by the OpenShift Dev Spaces installation."
- Table 3.10, "Configuration settings related to the Dashboard component used by the OpenShift Dev Spaces installation."
- Table 3.11, "Kubernetes Image Puller component configuration."
- Table 3.12, "OpenShift Dev Spaces server metrics component configuration."
- Table 3.13, "Networking, OpenShift Dev Spaces authentication and TLS configuration."
- Table 3.14, "Configuration of an alternative registry that stores OpenShift Dev Spaces images."
- Table 3.15, "**CheCluster** Custom Resource **status** defines the observed state of OpenShift Dev Spaces installation"

Example 3.2. A minimal CheCluster Custom Resource example.

```
apiVersion: org.eclipse.che/v2
kind: CheCluster
metadata:
  name: devspaces
spec:
  devEnvironments:
    defaultNamespace:
      template: '<username>-che'
    storage:
      pvcStrategy: 'common'
  components:
    database:
```

```
    externalDb: false
  metrics:
    enable: true
```

Table 3.1. Development environment configuration options.

Property	Description
containerBuildConfiguration	Container build configuration.
defaultComponents	Default components applied to DevWorkspaces. These default components are meant to be used when a Devfile, that does not contain any components.
defaultEditor	The default editor to workspace create with. It could be a plugin ID or a URI. The plugin ID must have **publisher/plugin/version** format. The URI must start from **http://** or **https://**.
defaultNamespace	User's default namespace.
defaultPlugins	Default plug-ins applied to DevWorkspaces.
disableContainerBuildCapabilities	Disables the container build capabilities.
maxNumberOfRunningWorkspacesPerUser	The maximum number of running workspaces per user. The value, -1, allows users to run an unlimited number of workspaces.
maxNumberOfWorkspacesPerUser	Total number of workspaces, both stopped and running, that a user can keep. The value, -1, allows users to keep an unlimited number of workspaces.
nodeSelector	The node selector limits the nodes that can run the workspace pods.
podSchedulerName	Pod scheduler for the workspace pods. If not specified, the pod scheduler is set to the default scheduler on the cluster.
secondsOfInactivityBeforeIdling	Idle timeout for workspaces in seconds. This timeout is the duration after which a workspace will be idled if there is no activity. To disable workspace idling due to inactivity, set this value to -1.
secondsOfRunBeforeIdling	Run timeout for workspaces in seconds. This timeout is the maximum duration a workspace runs. To disable workspace run timeout, set this value to -1.
serviceAccount	ServiceAccount to use by the DevWorkspace operator when starting the workspaces.
startTimeoutSeconds	StartTimeoutSeconds determines the maximum duration (in seconds) that a workspace can take to start before it is automatically failed. If not specified, the default value of 300 seconds (5 minutes) is used.

Property	Description
storage	Workspaces persistent storage.
tolerations	The pod tolerations of the workspace pods limit where the workspace pods can run.
trustedCerts	Trusted certificate settings.

Table 3.2. Development environment defaultNamespace options.

Property	Description
autoProvision	Indicates if is allowed to automatically create a user namespace. If it set to false, then user namespace must be pre-created by a cluster administrator.
template	If you don't create the user namespaces in advance, this field defines the Kubernetes namespace created when you start your first workspace. You can use **<username>** and **<userid>** placeholders, such as che-workspace-<username>.

Table 3.3. Development environment storage options.

Property	Description
perUserStrategyPvcConfig	PVC settings when using the **per-user** PVC strategy.
perWorkspaceStrategyPvcConfig	PVC settings when using the **per-workspace** PVC strategy.
pvcStrategy	Persistent volume claim strategy for the OpenShift Dev Spaces server. The supported strategies are: **per-user** (all workspaces PVCs in one volume), **per-workspace** (each workspace is given its own individual PVC) and **ephemeral** (non-persistent storage where local changes will be lost when the workspace is stopped.)

Table 3.4. OpenShift Dev Spaces components configuration.

Property	Description
cheServer	General configuration settings related to the OpenShift Dev Spaces server.
dashboard	Configuration settings related to the dashboard used by the OpenShift Dev Spaces installation.
database	Configuration settings related to the database used by the OpenShift Dev Spaces installation.

Property	Description
devWorkspace	DevWorkspace Operator configuration.
devfileRegistry	Configuration settings related to the devfile registry used by the OpenShift Dev Spaces installation.
imagePuller	Kubernetes Image Puller configuration.
metrics	OpenShift Dev Spaces server metrics configuration.
pluginRegistry	Configuration settings related to the plug-in registry used by the OpenShift Dev Spaces installation.

Table 3.5. DevWorkspace operator component configuration.

Property	Description
runningLimit	Deprecated in favor of **MaxNumberOfRunningWorkspacesPerUser** The maximum number of running workspaces per user.

Table 3.6. General configuration settings related to the OpenShift Dev Spaces server component.

Property	Description
clusterRoles	ClusterRoles assigned to OpenShift Dev Spaces ServiceAccount. The defaults roles are: - **<devspaces-namespace>-cheworkspaces-namespaces-clusterrole** - **<devspaces-namespace>-cheworkspaces-clusterrole** - **<devspaces-namespace>-cheworkspaces-devworkspace-clusterrole** where the <devspaces-namespace> is the namespace where the CheCluster CRD is created. Each role must have a **app.kubernetes.io/part-of=che.eclipse.org** label. The OpenShift Dev Spaces Operator must already have all permissions in these ClusterRoles to grant them.
debug	Enables the debug mode for OpenShift Dev Spaces server.
deployment	Deployment override options.
extraProperties	A map of additional environment variables applied in the generated **che** ConfigMap to be used by the OpenShift Dev Spaces server in addition to the values already generated from other fields of the **CheCluster** custom resource (CR). If the **extraProperties** field contains a property normally generated in **che** ConfigMap from other CR fields, the value defined in the **extraProperties** is used instead.
logLevel	The log level for the OpenShift Dev Spaces server: **INFO** or **DEBUG**.

Property	Description
proxy	Proxy server settings for Kubernetes cluster. No additional configuration is required for OpenShift cluster. By specifying these settings for the OpenShift cluster, you override the OpenShift proxy configuration.

Table 3.7. Configuration settings related to the Plug-in registry component used by the OpenShift Dev Spaces installation.

Property	Description
deployment	Deployment override options.
disableInternalRegistry	Disables internal plug-in registry.
externalPluginRegistries	External plugin registries.
openVSXURL	Open VSX registry URL. If omitted an embedded instance will be used.

Table 3.8. Configuration settings related to the Devfile registry component used by the OpenShift Dev Spaces installation.

Property	Description
deployment	Deployment override options.
disableInternalRegistry	Disables internal devfile registry.
externalDevfileRegistries	External devfile registries serving sample ready-to-use devfiles.

Table 3.9. Configuration settings related to the Database component used by the OpenShift Dev Spaces installation.

Property	Description
credentialsSecretName	The secret that contains PostgreSQL **user** and **password** that the OpenShift Dev Spaces server uses to connect to the database. The secret must have a **app.kubernetes.io/part-of=che.eclipse.org** label.
deployment	Deployment override options.
externalDb	Instructs the Operator to deploy a dedicated database. By default, a dedicated PostgreSQL database is deployed as part of the OpenShift Dev Spaces installation. When **externalDb** is set as **true**, no dedicated database is deployed by the Operator and you need to provide connection details about the external database you want to use.

Property	Description
postgresDb	PostgreSQL database name that the OpenShift Dev Spaces server uses to connect to the database.
postgresHostName	PostgreSQL database hostname that the OpenShift Dev Spaces server connects to. Override this value only when using an external database. See field **externalDb**.
postgresPort	PostgreSQL Database port the OpenShift Dev Spaces server connects to. Override this value only when using an external database. See field **externalDb**.
pvc	PVC settings for PostgreSQL database.

Table 3.10. Configuration settings related to the Dashboard component used by the OpenShift Dev Spaces installation.

Property	Description
deployment	Deployment override options.
headerMessage	Dashboard header message.

Table 3.11. Kubernetes Image Puller component configuration.

Property	Description
enable	Install and configure the community supported Kubernetes Image Puller Operator. When you set the value to **true** without providing any specs, it creates a default Kubernetes Image Puller object managed by the Operator. When you set the value to **false**, the Kubernetes Image Puller object is deleted, and the Operator uninstalled, regardless of whether a spec is provided. If you leave the **spec.images** field empty, a set of recommended workspace-related images is automatically detected and pre-pulled after installation. Note that while this Operator and its behavior is community-supported, its payload may be commercially-supported for pulling commercially-supported images.
spec	A Kubernetes Image Puller spec to configure the image puller in the CheCluster.

Table 3.12. OpenShift Dev Spaces server metrics component configuration.

Property	Description
enable	Enables **metrics** for the OpenShift Dev Spaces server endpoint.

Table 3.13. Networking, OpenShift Dev Spaces authentication and TLS configuration.

Property	Description
annotations	Defines annotations which will be set for an Ingress (a route for OpenShift platform). The defaults for kubernetes platforms are: kubernetes.io/ingress.class: \nginx\ nginx.ingress.kubernetes.io/proxy-read-timeout: \3600\, nginx.ingress.kubernetes.io/proxy-connect-timeout: \3600\, nginx.ingress.kubernetes.io/ssl-redirect: \true\
auth	Authentication settings.
domain	For an OpenShift cluster, the Operator uses the domain to generate a hostname for the route. The generated hostname follows this pattern: che-<devspaces-namespace>.<domain>. The <devspaces-namespace> is the namespace where the CheCluster CRD is created. In conjunction with labels, it creates a route served by a non-default Ingress controller. For a Kubernetes cluster, it contains a global ingress domain. There are no default values: you must specify them.
hostname	The public hostname of the installed OpenShift Dev Spaces server.
labels	Defines labels which will be set for an Ingress (a route for OpenShift platform).
tlsSecretName	The name of the secret used to set up Ingress TLS termination. If the field is an empty string, the default cluster certificate is used. The secret must have a **app.kubernetes.io/part-of=che.eclipse.org** label.

Table 3.14. Configuration of an alternative registry that stores OpenShift Dev Spaces images.

Property	Description
hostname	An optional hostname or URL of an alternative container registry to pull images from. This value overrides the container registry hostname defined in all the default container images involved in a OpenShift Dev Spaces deployment. This is particularly useful for installing OpenShift Dev Spaces in a restricted environment.
organization	An optional repository name of an alternative registry to pull images from. This value overrides the container registry organization defined in all the default container images involved in a OpenShift Dev Spaces deployment. This is particularly useful for installing OpenShift Dev Spaces in a restricted environment.

Table 3.15. `CheCluster` Custom Resource `status` defines the observed state of OpenShift Dev Spaces installation

Property	Description
chePhase	Specifies the current phase of the OpenShift Dev Spaces deployment.
cheURL	Public URL of the OpenShift Dev Spaces server.
cheVersion	Currently installed OpenShift Dev Spaces version.
devfileRegistryURL	The public URL of the internal devfile registry.
gatewayPhase	Specifies the current phase of the gateway deployment.
message	A human readable message indicating details about why the OpenShift Dev Spaces deployment is in the current phase.
pluginRegistryURL	The public URL of the internal plug-in registry.
postgresVersion	The PostgreSQL version of the image in use.
reason	A brief CamelCase message indicating details about why the OpenShift Dev Spaces deployment is in the current phase.
workspaceBaseDomain	The resolved workspace base domain. This is either the copy of the explicitly defined property of the same name in the spec or, if it is undefined in the spec and we're running on OpenShift, the automatically resolved basedomain for routes.

3.2. CONFIGURING PROJECTS

For each user, OpenShift Dev Spaces isolates workspaces in a project. OpenShift Dev Spaces identifies the user project by the presence of labels and annotations. When starting a workspace, if the required project doesn't exist, OpenShift Dev Spaces creates the project using a template name.

You can modify OpenShift Dev Spaces behavior by:

- Section 3.2.1, "Configuring project name"
- Section 3.2.2, "Provisioning projects in advance"

3.2.1. Configuring project name

You can configure the project name template that OpenShift Dev Spaces uses to create the required project when starting a workspace.

A valid project name template follows these conventions:

- The **<username>** or **<userid>** placeholder is mandatory.
- Usernames and IDs cannot contain invalid characters. If the formatting of a username or ID is incompatible with the naming conventions for OpenShift objects, OpenShift Dev Spaces

changes the username or ID to a valid name by replacing incompatible characters with the **-** symbol.

- OpenShift Dev Spaces evaluates the **<userid>** placeholder into a 14 character long string, and adds a random six character long suffix to prevent IDs from colliding. The result is stored in the user preferences for reuse.
- Kubernetes limits the length of a project name to 63 characters.
- OpenShift limits the length further to 49 characters.

Procedure

- Configure the **CheCluster** Custom Resource. See Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource".

```
spec:
  components:
    devEnvironments:
      defaultNamespace:
        template: <workspace_namespace_template_>
```

Example 3.3. User workspaces project name template examples

User workspaces project name template	Resulting project example
<username>-devspaces (default)	user1-devspaces
<userid>-namespace	**cge1egvsb2nhba-namespace-ul1411**
<userid>-aka-<username>-namespace	**cgezegvsb2nhba-aka-user1-namespace-6m2w2b**

Additional resources

- Section 3.1.1, "Using dsc to configure the **CheCluster** Custom Resource during installation"
- Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource"

3.2.2. Provisioning projects in advance

You can provision workspaces projects in advance, rather than relying on automatic provisioning. Repeat the procedure for each user.

Procedure

- Create the *<project_name>* project for *<username>* user with the following labels and annotations:

```
kind: Namespace
apiVersion: v1
```

```
metadata:
  name: <project_name> 1
  labels:
    app.kubernetes.io/part-of: che.eclipse.org
    app.kubernetes.io/component: workspaces-namespace
  annotations:
    che.eclipse.org/username: <username>
```

1 Use a project name of your choosing.

Additional resources

- Section 3.1.1, “Using dsc to configure the **CheCluster** Custom Resource during installation”
- Section 3.1.2, “Using the CLI to configure the CheCluster Custom Resource”

3.3. CONFIGURING SERVER COMPONENTS

- Section 3.3.1, “Mounting a Secret or a ConfigMap as a file or an environment variable into a Red Hat OpenShift Dev Spaces container”

3.3.1. Mounting a Secret or a ConfigMap as a file or an environment variable into a Red Hat OpenShift Dev Spaces container

Secrets are OpenShift objects that store sensitive data such as:

- usernames
- passwords
- authentication tokens

in an encrypted form.

Users can mount a OpenShift Secret that contains sensitive data or a ConfigMap that contains configuration in a OpenShift Dev Spaces managed containers as:

- a file
- an environment variable

The mounting process uses the standard OpenShift mounting mechanism, but it requires additional annotations and labeling.

3.3.1.1. Mounting a Secret or a ConfigMap as a file into a OpenShift Dev Spaces container

Prerequisites

- A running instance of Red Hat OpenShift Dev Spaces.

Procedure

1. Create a new OpenShift Secret or a ConfigMap in the OpenShift project where a OpenShift Dev Spaces is deployed. The labels of the object that is about to be created must match the set of labels:

   - **app.kubernetes.io/part-of: che.eclipse.org**
   - **app.kubernetes.io/component: <DEPLOYMENT_NAME>-<OBJECT_KIND>**
   - The **<DEPLOYMENT_NAME>** corresponds to the one following deployments:
     - **postgres**
     - **keycloak**
     - **devfile-registry**
     - **plugin-registry**
     - **devspaces**
       and
   - **<OBJECT_KIND>** is either:
     - **secret**
       or
     - **configmap**

Example 3.4. Example:

```
apiVersion: v1
kind: Secret
metadata:
  name: custom-settings
  labels:
    app.kubernetes.io/part-of: che.eclipse.org
    app.kubernetes.io/component: devspaces-secret
...
```

or

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: custom-settings
  labels:
    app.kubernetes.io/part-of: che.eclipse.org
    app.kubernetes.io/component: devspaces-configmap
...
```

Annotations must indicate that the given object is mounted as a file.

1. Configure the annotation values:

   - **che.eclipse.org/mount-as: file** - To indicate that a object is mounted as a file.

- **che.eclipse.org/mount-path: *<TARGET_PATH>*** - To provide a required mount path.

Example 3.5. Example:

```
apiVersion: v1
kind: Secret
metadata:
  name: custom-data
  annotations:
    che.eclipse.org/mount-as: file
    che.eclipse.org/mount-path: /data
  labels:
...
```

or

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: custom-data
  annotations:
    che.eclipse.org/mount-as: file
    che.eclipse.org/mount-path: /data
  labels:
...
```

The OpenShift object can contain several items whose names must match the desired file name mounted into the container.

Example 3.6. Example:

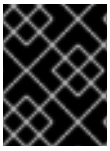
```
apiVersion: v1
kind: Secret
metadata:
  name: custom-data
  labels:
    app.kubernetes.io/part-of: che.eclipse.org
    app.kubernetes.io/component: devspaces-secret
  annotations:
    che.eclipse.org/mount-as: file
    che.eclipse.org/mount-path: /data
data:
  ca.crt: <base64 encoded data content here>
```

or

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: custom-data
  labels:
    app.kubernetes.io/part-of: che.eclipse.org
    app.kubernetes.io/component: devspaces-configmap
```

```
  annotations:
    che.eclipse.org/mount-as: file
    che.eclipse.org/mount-path: /data
data:
  ca.crt: <data content here>
```

This results in a file named **ca.crt** being mounted at the **/data** path of OpenShift Dev Spaces container.



IMPORTANT

To make the changes in a OpenShift Dev Spaces container visible, recreate the object entirely.

Additional resources

- Section 3.1.1, “Using dsc to configure the **CheCluster** Custom Resource during installation”
- Section 3.1.2, “Using the CLI to configure the CheCluster Custom Resource”

3.3.1.2. Mounting a Secret or a ConfigMap as an environment variable into a OpenShift Dev Spaces container

Prerequisites

- A running instance of Red Hat OpenShift Dev Spaces.

Procedure

1. Create a new OpenShift Secret or a ConfigMap in the OpenShift project where a OpenShift Dev Spaces is deployed. The labels of the object that is about to be created must match the set of labels:
   - **app.kubernetes.io/part-of: che.eclipse.org**
   - **app.kubernetes.io/component: <DEPLOYMENT_NAME>-<OBJECT_KIND>**
   - The **<DEPLOYMENT_NAME>** corresponds to the one following deployments:
     - **postgres**
     - **keycloak**
     - **devfile-registry**
     - **plugin-registry**
     - **devspaces**
       and
   - **<OBJECT_KIND>** is either:
     - **secret**
       or

- **configmap**

Example 3.7. Example:

```
apiVersion: v1
kind: Secret
metadata:
  name: custom-settings
  labels:
    app.kubernetes.io/part-of: che.eclipse.org
    app.kubernetes.io/component: devspaces-secret
...
```

or

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: custom-settings
  labels:
    app.kubernetes.io/part-of: che.eclipse.org
    app.kubernetes.io/component: devspaces-configmap
...
```

Annotations must indicate that the given object is mounted as a environment variable.

1. Configure the annotation values:
   - **che.eclipse.org/mount-as: env** - to indicate that a object is mounted as an environment variable
   - **che.eclipse.org/env-name: *<FOO_ENV>*** - to provide an environment variable name, which is required to mount a object key value

Example 3.8. Example:

```
apiVersion: v1
kind: Secret
metadata:
  name: custom-settings
  annotations:
    che.eclipse.org/env-name: FOO_ENV
    che.eclipse.org/mount-as: env
  labels:
    ...
data:
  mykey: myvalue
```

or

```
apiVersion: v1
kind: ConfigMap
metadata:
```

```
  name: custom-settings
  annotations:
    che.eclipse.org/env-name: FOO_ENV
    che.eclipse.org/mount-as: env
  labels:
    ...
data:
  mykey: myvalue
```

This results in two environment variables:

- **FOO_ENV**
- **myvalue**

being provisioned into a OpenShift Dev Spaces container.

If the object provides more than one data item, the environment variable name must be provided for each of the data keys as follows:

Example 3.9. Example:

```
apiVersion: v1
kind: Secret
metadata:
  name: custom-settings
  annotations:
    che.eclipse.org/mount-as: env
    che.eclipse.org/mykey_env-name: FOO_ENV
    che.eclipse.org/otherkey_env-name: OTHER_ENV
  labels:
    ...
data:
  mykey: __<base64 encoded data content here>__
  otherkey: __<base64 encoded data content here>__
```

or

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: custom-settings
  annotations:
    che.eclipse.org/mount-as: env
    che.eclipse.org/mykey_env-name: FOO_ENV
    che.eclipse.org/otherkey_env-name: OTHER_ENV
  labels:
    ...
data:
  mykey: __<data content here>__
  otherkey: __<data content here>__
```

This results in two environment variables:

- **FOO_ENV**
- **OTHER_ENV**

being provisioned into a OpenShift Dev Spaces container.

NOTE

The maximum length of annotation names in a OpenShift object is 63 characters, where 9 characters are reserved for a prefix that ends with /. This acts as a restriction for the maximum length of the key that can be used for the object.

IMPORTANT

To make the changes in a OpenShift Dev Spaces container visible, recreate the object entirely.

Additional resources

- Section 3.1.1, "Using dsc to configure the **CheCluster** Custom Resource during installation"
- Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource"

3.3.2. Advanced configuration options for Dev Spaces server

The following section describes advanced deployment and configuration methods for the OpenShift Dev Spaces server component.

3.3.2.1. Understanding OpenShift Dev Spaces server advanced configuration

The following section describes the OpenShift Dev Spaces server component advanced configuration method for a deployment.

Advanced configuration is necessary to:

- Add environment variables not automatically generated by the Operator from the standard **CheCluster** Custom Resource fields.
- Override the properties automatically generated by the Operator from the standard **CheCluster** Custom Resource fields.

The **customCheProperties** field, part of the **CheCluster** Custom Resource **server** settings, contains a map of additional environment variables to apply to the OpenShift Dev Spaces server component.

Example 3.10. Override the default memory limit for workspaces

- Configure the **CheCluster** Custom Resource. See Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource".

```
apiVersion: org.eclipse.che/v2
kind: CheCluster
spec:
  components:
```

```
cheServer:
  extraProperties:
    CHE_LOGS_APPENDERS_IMPL: json
```

NOTE

Previous versions of the OpenShift Dev Spaces Operator had a ConfigMap named **custom** to fulfill this role. If the OpenShift Dev Spaces Operator finds a **configMap** with the name **custom**, it adds the data it contains into the **customCheProperties** field, redeploys OpenShift Dev Spaces, and deletes the **custom configMap**.

Additional resources

- Section 3.1.3, "**CheCluster** Custom Resource fields reference".

3.4. CONFIGURING WORKSPACES GLOBALLY

This section describes how an administrator can configure workspaces globally.

- Section 3.4.1, "Limiting the number of workspaces that a user can keep"
- Section 3.4.2, "Enabling users to run multiple workspaces simultaneously"
- Section 3.4.3, "Git with self-signed certificates"
- Section 3.4.4, "Configuring workspaces nodeSelector"

3.4.1. Limiting the number of workspaces that a user can keep

By default, users can keep an unlimited number of workspaces in the dashboard, but you can limit this number to reduce demand on the cluster.

This configuration is part of the **CheCluster** Custom Resource:

```
spec:
  devEnvironments:
    maxNumberOfWorkspacesPerUser: <kept_workspaces_limit> 1
```

1 Sets the maximum number of workspaces per user. The default value, **-1**, allows users to keep an unlimited number of workspaces. Use a positive integer to set the maximum number of workspaces per user.

Procedure

1. Get the name of the OpenShift Dev Spaces namespace. The default is **openshift-devspaces**.

```
$ oc get checluster --all-namespaces \
  -o=jsonpath="{.items[*].metadata.namespace}"
```

2. Configure the **maxNumberOfWorkspacesPerUser**:

```
$ oc patch checluster/devspaces -n openshift-devspaces \ (1)
--type='merge' -p \
'{"spec":{"devEnvironments":{"maxNumberOfWorkspacesPerUser":
<kept_workspaces_limit>}}}' (2)
```

(1) The OpenShift Dev Spaces namespace that you got in step 1.

(2) Your choice of the ***<kept_workspaces_limit>*** value.

Additional resources

- Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource"

3.4.2. Enabling users to run multiple workspaces simultaneously

By default, a user can run only one workspace at a time. You can enable users to run multiple workspaces simultaneously.

NOTE

If using the default storage method, users might experience problems when concurrently running workspaces if pods are distributed across nodes in a multi-node cluster. Switching from the per-user **common** storage strategy to the **per-workspace** storage strategy or using the **ephemeral** storage type can avoid or solve those problems.

This configuration is part of the **CheCluster** Custom Resource:

```
spec:
  devEnvironments:
    maxNumberOfRunningWorkspacesPerUser: <running_workspaces_limit> (1)
```

(1) Sets the maximum number of simultaneously running workspaces per user. The **-1** value enables users to run an unlimited number of workspaces. The default value is **1**.

Procedure

1. Get the name of the OpenShift Dev Spaces namespace. The default is **openshift-devspaces**.

```
$ oc get checluster --all-namespaces \
  -o=jsonpath="{.items[*].metadata.namespace}"
```

2. Configure the **maxNumberOfRunningWorkspacesPerUser**:

```
$ oc patch checluster/devspaces -n openshift-devspaces \ (1)
--type='merge' -p \
'{"spec":{"devEnvironments":{"maxNumberOfRunningWorkspacesPerUser":
<running_workspaces_limit>}}}' (2)
```

(1) The OpenShift Dev Spaces namespace that you got in step 1.

2 Your choice of the ***<running_workspaces_limit>*** value.

Additional resources

- Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource"

3.4.3. Git with self-signed certificates

You can configure OpenShift Dev Spaces to support operations on Git providers that use self-signed certificates.

Prerequisites

- An active **oc** session with administrative permissions to the OpenShift cluster. See Getting started with the OpenShift CLI.
- Git version 2 or later

Procedure

1. Create a new **ConfigMap** with details about the Git server:

```
$ oc create configmap che-git-self-signed-cert \
  --from-file=ca.crt=<path_to_certificate> \ 1
  --from-literal=githost=<host:port> -n openshift-devspaces 2
```

1 Path to self-signed certificate

2 The host and port of the HTTPS connection on the Git server (optional).

NOTE

- When **githost** is not specified, the given certificate is used for all HTTPS repositories.
- Certificate files are typically stored as Base64 ASCII files, such as. **.pem**, **.crt**, **.ca-bundle**. Also, they can be encoded as binary data, for example, **.cer**. All **Secrets** that hold certificate files should use the Base64 ASCII certificate rather than the binary data certificate.

2. Add the required labels to the ConfigMap:

```
$ oc label configmap che-git-self-signed-cert \
  app.kubernetes.io/part-of=che.eclipse.org -n openshift-devspaces
```

3. Configure OpenShift Dev Spaces operand to use self-signed certificates for Git repositories. See Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource" .

```
spec:
  devEnvironments:
    trustedCerts:
      gitTrustedCertsConfigMapName: che-git-self-signed-cert
```

Verification steps

- Create and start a new workspace. Every container used by the workspace mounts a special volume that contains a file with the self-signed certificate. The container's **/etc/gitconfig** file contains information about the Git server host (its URL) and the path to the certificate in the **http** section (see Git documentation about git-config).

Example 3.11. Contents of an /etc/gitconfig file

```
[http "https://10.33.177.118:3000"]
sslCAInfo = /etc/config/che-git-tls-creds/certificate
```

Additional resources

- Section 3.1.1, "Using dsc to configure the **CheCluster** Custom Resource during installation"
- Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource"
- Section 3.7.3, "Importing untrusted TLS certificates to Dev Spaces" .

3.4.4. Configuring workspaces nodeSelector

This section describes how to configure **nodeSelector** for Pods of OpenShift Dev Spaces workspaces.

Procedure

OpenShift Dev Spaces uses the **CHE_WORKSPACE_POD_NODE__SELECTOR** environment variable to configure **nodeSelector**. This variable can contain a set of comma-separated **key=value** pairs to form the nodeSelector rule, or **NULL** to disable it.

```
CHE_WORKSPACE_POD_NODE__SELECTOR=disktype=ssd,cpu=xlarge,[key=value]
```

IMPORTANT

nodeSelector must be configured during OpenShift Dev Spaces installation. This prevents existing workspaces from failing to run due to volumes affinity conflict caused by existing workspace PVC and Pod being scheduled in different zones.

To avoid Pods and PVCs to be scheduled in different zones on large, multizone clusters, create an additional **StorageClass** object (pay attention to the **allowedTopologies** field), which will coordinate the PVC creation process.

Pass the name of this newly created **StorageClass** to OpenShift Dev Spaces through the **CHE_INFRA_KUBERNETES_PVC_STORAGE__CLASS__NAME** environment variable. A default empty value of this variable instructs OpenShift Dev Spaces to use the cluster's default **StorageClass**.

Additional resources

- Section 3.1.1, "Using dsc to configure the **CheCluster** Custom Resource during installation"
- Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource"

3.4.5. Open VSX registry URL

To search and install extensions, the Visual Studio Code editor uses an embedded Open VSX registry instance. You can also configure OpenShift Dev Spaces to use another Open VSX registry instance rather than the embedded one.

Procedure

- Set the URL of your Open VSX registry instance in the CheCluster Custom Resource **spec.components.pluginRegistry.openVSXURL** field.

```
spec:
  components:
# [...]
    pluginRegistry:
      openVSXURL: <your_open_vsx_registy>
# [...]
```

Additional resources

- Section 3.1.2, “Using the CLI to configure the CheCluster Custom Resource”
- Open VSX registry

3.5. CACHING IMAGES FOR FASTER WORKSPACE START

To improve the start time performance of OpenShift Dev Spaces workspaces, use the Image Puller, a OpenShift Dev Spaces-agnostic component that can be used to pre-pull images for OpenShift clusters. The Image Puller is an additional OpenShift deployment which creates a *DaemonSet* that can be configured to pre-pull relevant OpenShift Dev Spaces workspace images on each node. These images would already be available when a OpenShift Dev Spaces workspace starts, therefore improving the workspace start time.

The Image Puller provides the following parameters for configuration.

Table 3.16. Image Puller parameters

Parameter	Usage	Default
CACHING_INTERVAL_HOURS	DaemonSets health checks interval in hours	**"1"**
CACHING_MEMORY_REQUEST	The memory request for each cached image while the puller is running. See Section 3.5.2, “Defining the memory settings”.	**10Mi**
CACHING_MEMORY_LIMIT	The memory limit for each cached image while the puller is running. See Section 3.5.2, “Defining the memory settings”.	**20Mi**

Parameter	Usage	Default
CACHING_CPU_REQUEST	The processor request for each cached image while the puller is running	**.05** or 50 millicores
CACHING_CPU_LIMIT	The processor limit for each cached image while the puller is running	**.2** or 200 millicores
DAEMONSET_NAME	Name of DaemonSet to create	**kubernetes-image-puller**
DEPLOYMENT_NAME	Name of the Deployment to create	**kubernetes-image-puller**
NAMESPACE	OpenShift project containing DaemonSet to create	**k8s-image-puller**
IMAGES	Semicolon-separated list of images to pull, in the format ***<name1>=<image1>;<name2>=<image2>***. See Section 3.5.1, "Defining the list of images".	
NODE_SELECTOR	Node selector to apply to the pods created by the DaemonSet	**'{}'**
AFFINITY	Affinity applied to pods created by the DaemonSet	**'{}'**
IMAGE_PULL_SECRETS	List of image pull secrets, in the format **pullsecret1;...** to add to pods created by the DaemonSet. Those secrets need to be in the image puller's namespace and a cluster administrator must create them.	**""**

Additional resources

- Section 3.5.1, "Defining the list of images"
- Section 3.5.2, "Defining the memory settings".
- Section 3.5.3, "Installing Image Puller on OpenShift using the web console"
- Section 3.5.4, "Installing Image Puller on OpenShift using CLI"
- Kubernetes Image Puller source code repository

3.5.1. Defining the list of images

The Image Puller can pre-pull most images, including scratch images such as **che-machine-exec**. However, images that mount volumes in the Dockerfile, such as **traefik**, are not supported for pre-pulling on OpenShift 3.11.

Procedure

1. Gather a list of relevant container images to pull by navigating to the **https://devspaces-*<openshift_deployment_name>*.*<domain_name>*/plugin-registry/v3/external_images.txt** URL.

2. Determine images from the list for pre-pulling. For faster workspace startup times, consider pulling workspace related images such as **universal-developer-image**, che-code`, and **che-gateway**.

Additional resources

- Section 3.5.3, “Installing Image Puller on OpenShift using the web console”
- Section 3.5.4, “Installing Image Puller on OpenShift using CLI”

3.5.2. Defining the memory settings

Define the memory requests and limits parameters to ensure pulled containers and the platform have enough memory to run.

Prerequisites

- Section 3.5.1, “Defining the list of images”

Procedure

1. To define the minimal value for **CACHING_MEMORY_REQUEST** or **CACHING_MEMORY_LIMIT**, consider the necessary amount of memory required to run each of the container images to pull.

2. To define the maximal value for **CACHING_MEMORY_REQUEST** or **CACHING_MEMORY_LIMIT**, consider the total memory allocated to the DaemonSet Pods in the cluster:

   ```
   (memory limit) * (number of images) * (number of nodes in the cluster)
   ```

   Pulling 5 images on 20 nodes, with a container memory limit of **20Mi** requires **2000Mi** of memory.

Additional resources

- Section 3.5.3, “Installing Image Puller on OpenShift using the web console”
- Section 3.5.4, “Installing Image Puller on OpenShift using CLI”

3.5.3. Installing Image Puller on OpenShift using the web console

You can install the community supported Kubernetes Image Puller Operator on OpenShift using the OpenShift web console.

Prerequisites

- Section 3.5.1, "Defining the list of images"
- Section 3.5.2, "Defining the memory settings".
- An OpenShift web console session by a cluster administrator. See Accessing the web console.

Procedure

1. Install the community supported Kubernetes Image Puller Operator. See Installing from OperatorHub using the web console.
2. Create a kubernetes-image-puller **KubernetesImagePuller** operand from the community supported Kubernetes Image Puller Operator. See Creating applications from installed Operators.

3.5.4. Installing Image Puller on OpenShift using CLI

You can install the Kubernetes Image Puller on OpenShift by using OpenShift **oc** management tool.

Prerequisites

- Section 3.5.1, "Defining the list of images".
- Section 3.5.2, "Defining the memory settings".
- An active **oc** session with administrative permissions to the OpenShift cluster. See Getting started with the OpenShift CLI.

Procedure

1. Clone the Image Puller repository and get in the directory containing the OpenShift templates:

```
$ git clone https://github.com/che-incubator/kubernetes-image-puller
$ cd kubernetes-image-puller/deploy/openshift
```

2. Configure the **app.yaml**, **configmap.yaml** and **serviceaccount.yaml** OpenShift templates using following parameters:

Table 3.17. Image Puller OpenShift templates parameters in app.yaml

Value	Usage	Default
DEPLOYMENT_NAME	The value of **DEPLOYMENT_NAME** in the ConfigMap	**kubernetes-image-puller**
IMAGE	Image used for the **kubernetes-image-puller** deployment	**registry.redhat.io/devspaces/imagepuller-rhel8:3.5**

Value	Usage	Default
IMAGE_TAG	The image tag to pull	**latest**
SERVICEACCOUNT_NAME	The name of the ServiceAccount created and used by the deployment	**kubernetes-image-puller**

Table 3.18. Image Puller OpenShift templates parameters in **configmap.yaml**

Value	Usage	Default
CACHING_CPU_LIMIT	The value of **CACHING_CPU_LIMIT** in the ConfigMap	**.2**
CACHING_CPU_REQUEST	The value of **CACHING_CPU_REQUEST** in the ConfigMap	**.05**
CACHING_INTERVAL_HOURS	The value of **CACHING_INTERVAL_HOURS** in the ConfigMap	**"1"**
CACHING_MEMORY_LIMIT	The value of **CACHING_MEMORY_LIMIT** in the ConfigMap	**"20Mi"**
CACHING_MEMORY_REQUEST	The value of **CACHING_MEMORY_REQUEST** in the ConfigMap	**"10Mi"**
DAEMONSET_NAME	The value of **DAEMONSET_NAME** in the ConfigMap	**kubernetes-image-puller**
DEPLOYMENT_NAME	The value of **DEPLOYMENT_NAME** in the ConfigMap	**kubernetes-image-puller**
IMAGES	The value of **IMAGES** in the ConfigMap	**"undefined"**
NAMESPACE	The value of **NAMESPACE** in the ConfigMap	**k8s-image-puller**

Value	Usage	Default
NODE_SELECTOR	The value of **NODE_SELECTOR** in the ConfigMap	**"{}"**

Table 3.19. Image Puller OpenShift templates parameters in serviceaccount.yaml

Value	Usage	Default
SERVICEACCOUNT_NAME	The name of the ServiceAccount created and used by the deployment	**kubernetes-image-puller**

3. Create an OpenShift project to host the Image Puller:

```
$ oc new-project <k8s-image-puller>
```

4. Process and apply the templates to install the puller:

```
$ oc process -f serviceaccount.yaml | oc apply -f -
$ oc process -f configmap.yaml | oc apply -f -
$ oc process -f app.yaml | oc apply -f -
```

Verification steps

1. Verify the existence of a *<kubernetes-image-puller>* deployment and a *<kubernetes-image-puller>* DaemonSet. The DaemonSet needs to have a Pod for each node in the cluster:

```
$ oc get deployment,daemonset,pod --namespace <k8s-image-puller>
```

2. Verify the values of the *<kubernetes-image-puller>* **ConfigMap**.

```
$ oc get configmap <kubernetes-image-puller> --output yaml
```

3.6. CONFIGURING OBSERVABILITY

To configure OpenShift Dev Spaces observability features, see:

- Section 3.6.1, “Che-Theia workspaces”
- Section 3.6.2, “Configuring server logging”
- Section 3.6.3, “Collecting logs using dsc”
- Section 3.6.4.2, “Monitoring the Dev Workspace Operator”
- Section 3.6.4.3, “Monitoring Dev Spaces Server”

3.6.1. Che-Theia workspaces

3.6.1.1. Telemetry overview

Telemetry is the explicit and ethical collection of operation data. By default, telemetry is not available in Red Hat OpenShift Dev Spaces, but in the Che-Theia editor there is an abstract API that allows enabling telemetry using the plugin mechanism and in the **chectl** command line tool usage data can be collected using segment. This approach is used in the "Eclipse Che hosted by Red Hat " service where telemetry is enabled for every Che-Theia workspace.

This documentation includes a guide describing how to make your own telemetry client for Red Hat OpenShift Dev Spaces, followed by an overview of the Red Hat OpenShift Dev Spaces Woopra Telemetry Plugin.

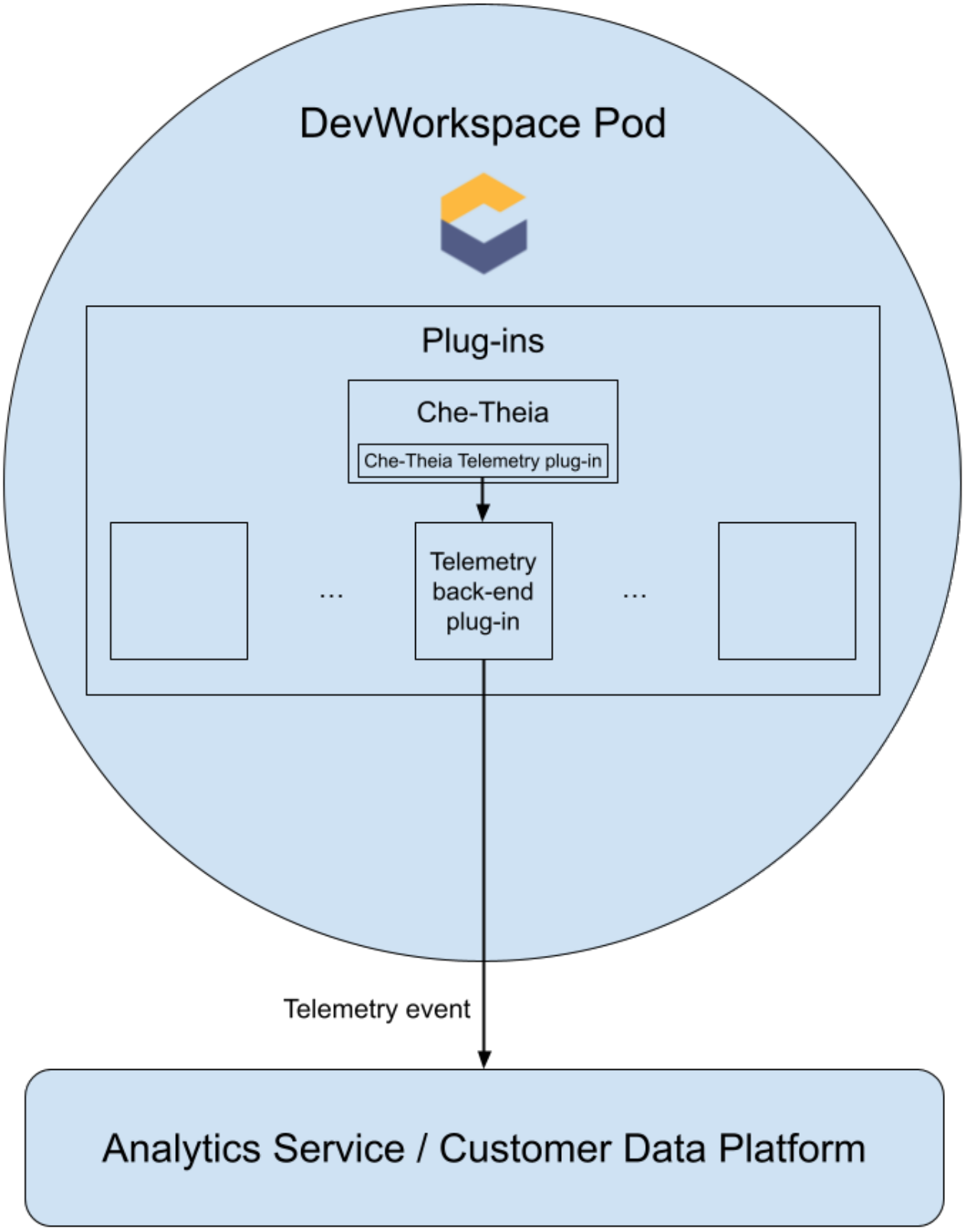
3.6.1.2. Use cases

Red Hat OpenShift Dev Spaces telemetry API allows tracking:

- Duration of a workspace utilization
- User-driven actions such as file editing, committing, and pushing to remote repositories.
- Programming languages and devfiles used in workspaces.

3.6.1.3. How it works

When a Dev Workspace starts, the **che-theia** container starts the telemetry plugin which is responsible for sending telemetry events to a backend. If the **$DEVWORKSPACE_TELEMETRY_BACKEND_PORT** environment variable is set in the Dev Workspace Pod, the telemetry plugin sends events to a backend listening at that port. The backend turns received events into a backend-specific representation of the events and sends them to the configured analytics backend (for example, Segment or Woopra).



3.6.1.4. Events sent to the backend by the Che-Theia telemetry plugin

Event	Description
WORKSPACE_OPENED	Sent when Che-Theia starts running

Event	Description
COMMIT_LOCALLY	Sent when a commit was made locally with the **git.commit** Theia command
PUSH_TO_REMOTE	Sent when a Git push was made with the **git.push** Theia command
EDITOR_USED	Sent when a file was changed within the editor

Other events such as **WORKSPACE_INACTIVE** and **WORKSPACE_STOPPED** can be detected within the back-end plugin.

3.6.1.5. The Woopra telemetry plugin

The Woopra Telemetry Plugin is a plugin built to send telemetry from a Red Hat OpenShift Dev Spaces installation to Segment and Woopra. This plugin is used by Eclipse Che hosted by Red Hat , but any Red Hat OpenShift Dev Spaces deployment can take advantage of this plugin. There are no dependencies other than a valid Woopra domain and Segment Write key. The devfile v2 for the plugin, plugin.yaml, has four environment variables that can be passed to the plugin:

- **WOOPRA_DOMAIN** - The Woopra domain to send events to.
- **SEGMENT_WRITE_KEY** - The write key to send events to Segment and Woopra.
- **WOOPRA_DOMAIN_ENDPOINT** - If you prefer not to pass in the Woopra domain directly, the plugin will get it from a supplied HTTP endpoint that returns the Woopra Domain.
- **SEGMENT_WRITE_KEY_ENDPOINT** - If you prefer not to pass in the Segment write key directly, the plugin will get it from a supplied HTTP endpoint that returns the Segment write key.

To enable the Woopra plugin on the Red Hat OpenShift Dev Spaces installation:

Procedure

- Deploy the **plugin.yaml** devfile v2 file to an HTTP server with the environment variables set correctly.

1. Configure the **CheCluster** Custom Resource. See Section 3.1.2, “Using the CLI to configure the CheCluster Custom Resource”.

```
spec:
  devEnvironments:
    defaultPlugins:
    - editor: eclipse/che-theia/next    (1)
      plugins:                          (2)
      - 'https://your-web-server/plugin.yaml'
```

(1) The **editorId** to set the telemetry plugin for.

(2) The URL to the telemetry plugin’s devfile v2 definition.

Additional resources

- Section 3.1.1, “Using dsc to configure the **CheCluster** Custom Resource during installation”
- Section 3.1.2, “Using the CLI to configure the CheCluster Custom Resource”

3.6.1.6. Creating a telemetry plugin

This section shows how to create an **AnalyticsManager** class that extends **AbstractAnalyticsManager** and implements the following methods:

- **isEnabled()** - determines whether the telemetry backend is functioning correctly. This can mean always returning **true**, or have more complex checks, for example, returning **false** when a connection property is missing.
- **destroy()** - cleanup method that is run before shutting down the telemetry backend. This method sends the **WORKSPACE_STOPPED** event.
- **onActivity()** - notifies that some activity is still happening for a given user. This is mainly used to send **WORKSPACE_INACTIVE** events.
- **onEvent()** - submits telemetry events to the telemetry server, such as **WORKSPACE_USED** or **WORKSPACE_STARTED**.
- **increaseDuration()** - increases the duration of a current event rather than sending many events in a small frame of time.

The following sections cover:

- Creating a telemetry server to echo events to standard output.
- Extending the OpenShift Dev Spaces telemetry client and implementing a user’s custom backend.
- Creating a **plugin.yaml** file representing a Dev Workspace plugin for the custom backend.
- Specifying of a location of a custom plugin to OpenShift Dev Spaces by setting the **workspacesDefaultPlugins** attribute from the **CheCluster** custom resource.

3.6.1.6.1. Getting started

This document describes the steps required to extend the OpenShift Dev Spaces telemetry system to communicate with to a custom backend:

1. Creating a server process that receives events
2. Extending OpenShift Dev Spaces libraries to create a backend that sends events to the server
3. Packaging the telemetry backend in a container and deploying it to an image registry
4. Adding a plugin for your backend and instructing OpenShift Dev Spaces to load the plugin in your Dev Workspaces

A finished example of the telemetry backend is available here.

CREATING A SERVER THAT RECEIVES EVENTS

For demonstration purposes, this example shows how to create a server that receives events from our telemetry plugin and writes them to standard output.

For production use cases, consider integrating with a third-party telemetry system (for example, Segment, Woopra) rather than creating your own telemetry server. In this case, use your provider's APIs to send events from your custom backend to their system.

The following Go code starts a server on port **8080** and writes events to standard output:

Example 3.12. main.go

```
package main

import (
 "io/ioutil"
 "net/http"

 "go.uber.org/zap"
)

var logger *zap.SugaredLogger

func event(w http.ResponseWriter, req *http.Request) {
 switch req.Method {
 case "GET":
  logger.Info("GET /event")
 case "POST":
  logger.Info("POST /event")
 }
 body, err := req.GetBody()
 if err != nil {
  logger.With("err", err).Info("error getting body")
  return
 }
 responseBody, err := ioutil.ReadAll(body)
 if err != nil {
  logger.With("error", err).Info("error reading response body")
  return
 }
 logger.With("body", string(responseBody)).Info("got event")
}

func activity(w http.ResponseWriter, req *http.Request) {
 switch req.Method {
 case "GET":
  logger.Info("GET /activity, doing nothing")
 case "POST":
  logger.Info("POST /activity")
  body, err := req.GetBody()
  if err != nil {
   logger.With("error", err).Info("error getting body")
   return
  }
  responseBody, err := ioutil.ReadAll(body)
  if err != nil {
   logger.With("error", err).Info("error reading response body")
   return
  }
  logger.With("body", string(responseBody)).Info("got activity")
```

```
 }
}

func main() {

 log, _ := zap.NewProduction()
 logger = log.Sugar()

 http.HandleFunc("/event", event)
 http.HandleFunc("/activity", activity)
 logger.Info("Added Handlers")

 logger.Info("Starting to serve")
 http.ListenAndServe(":8080", nil)
}
```

Create a container image based on this code and expose it as a deployment in OpenShift in the **openshift-devspaces** project. The code for the example telemetry server is available at telemetry-server-example. To deploy the telemetry server, clone the repository and build the container:

```
$ git clone https://github.com/che-incubator/telemetry-server-example
$ cd telemetry-server-example
$ podman build -t registry/organization/telemetry-server-example:latest .
$ podman push registry/organization/telemetry-server-example:latest
```

Both **manifest_with_ingress.yaml** and **manifest_with_route** contain definitions for a Deployment and Service. The former also defines a Kubernetes Ingress, while the latter defines an OpenShift Route.

In the manifest file, replace the **image** and **host** fields to match the image you pushed, and the public hostname of your OpenShift cluster. Then run:

```
$ kubectl apply -f manifest_with_[ingress|route].yaml -n openshift-devspaces
```

3.6.1.6.2. Creating the back-end project

NOTE

For fast feedback when developing, it is recommended to do development inside a Dev Workspace. This way, you can run the application in a cluster and receive events from the front-end telemetry plugin.

1. Maven Quarkus project scaffolding:

```
mvn io.quarkus:quarkus-maven-plugin:2.7.1.Final:create \
    -DprojectGroupId=mygroup -DprojectArtifactId=devworkspace-telemetry-example-plugin \
-DprojectVersion=1.0.0-SNAPSHOT
```

2. Remove the files under **src/main/java/mygroup** and **src/test/java/mygroup**.

3. Consult the GitHub packages for the latest version and Maven coordinates of **backend-base**.

4. Add the following dependencies to your **pom.xml**:

Example 3.13. **pom.xml**

```
<!-- Required -->
<dependency>
    <groupId>org.eclipse.che.incubator.workspace-telemetry</groupId>
    <artifactId>backend-base</artifactId>
    <version>LATEST VERSION FROM PREVIOUS STEP</version>
</dependency>


<!-- Used to make http requests to the telemetry server -->
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-rest-client</artifactId>
</dependency>
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-rest-client-jackson</artifactId>
</dependency>
```

5. Create a personal access token with **read:packages** permissions to download the **org.eclipse.che.incubator.workspace-telemetry:backend-base** dependency from GitHub packages.

6. Add your GitHub username, personal access token and **che-incubator** repository details in your **~/.m2/settings.xml** file:

Example 3.14. **settings.xml**

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <servers>
    <server>
      <id>che-incubator</id>
      <username>YOUR GITHUB USERNAME</username>
      <password>YOUR GITHUB TOKEN</password>
    </server>
  </servers>

  <profiles>
    <profile>
      <id>github</id>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <repositories>
        <repository>
          <id>central</id>
          <url>https://repo1.maven.org/maven2</url>
          <releases><enabled>true</enabled></releases>
          <snapshots><enabled>false</enabled></snapshots>
          </repository>
          <repository>
```

```
            <id>che-incubator</id>
            <url>https://maven.pkg.github.com/che-incubator/che-workspace-telemetry-
client</url>
          </repository>
        </repositories>
      </profile>
    </profiles>
</settings>
```

3.6.1.6.3. Creating a concrete implementation of AnalyticsManager and adding specialized logic

Create two files in your project under **src/main/java/mygroup**:

- **MainConfiguration.java** - contains configuration provided to **AnalyticsManager**.
- **AnalyticsManager.java** - contains logic specific to the telemetry system.

Example 3.15. MainConfiguration.java

```
package org.my.group;

import java.util.Optional;

import javax.enterprise.context.Dependent;
import javax.enterprise.inject.Alternative;

import org.eclipse.che.incubator.workspace.telemetry.base.BaseConfiguration;
import org.eclipse.microprofile.config.inject.ConfigProperty;

@Dependent
@Alternative
public class MainConfiguration extends BaseConfiguration {
    @ConfigProperty(name = "welcome.message")  1
    Optional<String> welcomeMessage;  2
}
```

1 A MicroProfile configuration annotation is used to inject the **welcome.message** configuration.

For more details on how to set configuration properties specific to your backend, see the Quarkus Configuration Reference Guide.

Example 3.16. AnalyticsManager.java

```
package org.my.group;

import java.util.HashMap;
import java.util.Map;

import javax.enterprise.context.Dependent;
import javax.enterprise.inject.Alternative;
import javax.inject.Inject;
```

```
import org.eclipse.che.incubator.workspace.telemetry.base.AbstractAnalyticsManager;
import org.eclipse.che.incubator.workspace.telemetry.base.AnalyticsEvent;
import org.eclipse.che.incubator.workspace.telemetry.finder.DevWorkspaceFinder;
import org.eclipse.che.incubator.workspace.telemetry.finder.UsernameFinder;
import org.eclipse.microprofile.rest.client.inject.RestClient;
import org.slf4j.Logger;

import static org.slf4j.LoggerFactory.getLogger;

@Dependent
@Alternative
public class AnalyticsManager extends AbstractAnalyticsManager {

    private static final Logger LOG = getLogger(AbstractAnalyticsManager.class);

    public AnalyticsManager(MainConfiguration mainConfiguration, DevWorkspaceFinder
devworkspaceFinder, UsernameFinder usernameFinder) {
        super(mainConfiguration, devworkspaceFinder, usernameFinder);

        mainConfiguration.welcomeMessage.ifPresentOrElse(      (1)
            (str) -> LOG.info("The welcome message is: {}", str),
            () -> LOG.info("No welcome message provided")
        );
    }

    @Override
    public boolean isEnabled() {
        return true;
    }

    @Override
    public void destroy() {}

    @Override
    public void onEvent(AnalyticsEvent event, String ownerId, String ip, String userAgent, String
resolution, Map<String, Object> properties) {
        LOG.info("The received event is: {}", event);          (2)
    }

    @Override
    public void increaseDuration(AnalyticsEvent event, Map<String, Object> properties) { }

    @Override
    public void onActivity() {}
}
```

1. Log the welcome message if it was provided.
2. Log the event received from the front-end plugin.

Since **org.my.group.AnalyticsManager** and **org.my.group.MainConfiguration** are alternative beans, specify them using the **quarkus.arc.selected-alternatives** property in **src/main/resources/application.properties**.

Example 3.17. application.properties

```
quarkus.arc.selected-alternatives=MainConfiguration,AnalyticsManager
```

3.6.1.6.4. Running the application within a Dev Workspace

1. Set the **DEVWORKSPACE_TELEMETRY_BACKEND_PORT** environment variable in the Dev Workspace. Here, the value is set to **4167**.

```
spec:
  template:
    attributes:
      workspaceEnv:
        - name: DEVWORKSPACE_TELEMETRY_BACKEND_PORT
          value: '4167'
```

2. Restart the Dev Workspace from the Red Hat OpenShift Dev Spaces dashboard.

3. Run the following command within a Dev Workspace's terminal window to start the application. Use the **--settings** flag to specify path to the location of the **settings.xml** file that contains the GitHub access token.

```
$ mvn --settings=settings.xml quarkus:dev -
Dquarkus.http.port=${DEVWORKSPACE_TELEMETRY_BACKEND_PORT}
```

The application now receives telemetry events through port **4167** from the front-end plugin.

Verification steps

1. Verify that the following output is logged:

```
INFO  [org.ecl.che.inc.AnalyticsManager] (Quarkus Main Thread) No welcome message
provided
INFO  [io.quarkus] (Quarkus Main Thread) devworkspace-telemetry-example-plugin 1.0.0-
SNAPSHOT on JVM (powered by Quarkus 2.7.2.Final) started in 0.323s. Listening on:
http://localhost:4167
INFO  [io.quarkus] (Quarkus Main Thread) Profile dev activated. Live Coding activated.
INFO  [io.quarkus] (Quarkus Main Thread) Installed features: [cdi, kubernetes-client, rest-
client, rest-client-jackson, resteasy, resteasy-jsonb, smallrye-context-propagation, smallrye-
openapi, swagger-ui, vertx]
```

2. To verify that the **onEvent()** method of **AnalyticsManager** receives events from the front-end plugin, press the **l** key to disable Quarkus live coding and edit any file within the IDE. The following output should be logged:

```
INFO  [io.qua.dep.dev.RuntimeUpdatesProcessor] (Aesh InputStream Reader) Live reload
disabled
INFO  [org.ecl.che.inc.AnalyticsManager] (executor-thread-2) The received event is: Edit
Workspace File in Che
```

3.6.1.6.5. Implementing isEnabled()

For the purposes of the example, this method always returns **true** whenever it is called.

Example 3.18. AnalyticsManager.java

```
@Override
public boolean isEnabled() {
    return true;
}
```

It is possible to put more complex logic in **isEnabled()**. For example, the hosted OpenShift Dev Spaces Woopra backend checks that a configuration property exists before determining if the backend is enabled.

3.6.1.6.6. Implementing onEvent()

onEvent() sends the event received by the backend to the telemetry system. For the example application, it sends an HTTP POST payload to the **/event** endpoint from the telemetry server.

Sending a POST request to the example telemetry server
For the following example, the telemetry server application is deployed to OpenShift at the following URL: **http://little-telemetry-server-che.apps-crc.testing**, where **apps-crc.testing** is the ingress domain name of the OpenShift cluster.

1. Set up the RESTEasy REST Client by creating **TelemetryService.java**

   Example 3.19. TelemetryService.java

   ```
   package org.my.group;

   import java.util.Map;

   import javax.ws.rs.Consumes;
   import javax.ws.rs.POST;
   import javax.ws.rs.Path;
   import javax.ws.rs.core.MediaType;
   import javax.ws.rs.core.Response;

   import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;

   @RegisterRestClient
   public interface TelemetryService {
       @POST
       @Path("/event") (1)
       @Consumes(MediaType.APPLICATION_JSON)
       Response sendEvent(Map<String, Object> payload);
   }
   ```

   (1) The endpoint to make the **POST** request to.

2. Specify the base URL for **TelemetryService** in the **src/main/resources/application.properties** file:

Example 3.20. **application.properties**

```
org.my.group.TelemetryService/mp-rest/url=http://little-telemetry-server-che.apps-crc.testing
```

3. Inject **TelemetryService** into **AnalyticsManager** and send a **POST** request in **onEvent()**

Example 3.21. **AnalyticsManager.java**

```
@Dependent
@Alternative
public class AnalyticsManager extends AbstractAnalyticsManager {
    @Inject
    @RestClient
    TelemetryService telemetryService;

...

@Override
public void onEvent(AnalyticsEvent event, String ownerId, String ip, String userAgent,
String resolution, Map<String, Object> properties) {
    Map<String, Object> payload = new HashMap<String, Object>(properties);
    payload.put("event", event);
    telemetryService.sendEvent(payload);
}
```

This sends an HTTP request to the telemetry server and automatically delays identical events for a small period of time. The default duration is 1500 milliseconds.

3.6.1.6.7. Implementing **increaseDuration()**

Many telemetry systems recognize event duration. The **AbstractAnalyticsManager** merges similar events that happen in the same frame of time into one event. This implementation of **increaseDuration()** is a no-op. This method uses the APIs of the user's telemetry provider to alter the event or event properties to reflect the increased duration of an event.

Example 3.22. **AnalyticsManager.java**

```
@Override
public void increaseDuration(AnalyticsEvent event, Map<String, Object> properties) {}
```

3.6.1.6.8. Implementing **onActivity()**

Set an inactive timeout limit, and use **onActivity()** to send a **WORKSPACE_INACTIVE** event if the last event time is longer than the timeout.

Example 3.23. **AnalyticsManager.java**

```
public class AnalyticsManager extends AbstractAnalyticsManager {
```

```
    ...

    private long inactiveTimeLimit = 60000 * 3;

    ...

    @Override
    public void onActivity() {
        if (System.currentTimeMillis() - lastEventTime >= inactiveTimeLimit) {
            onEvent(WORKSPACE_INACTIVE, lastOwnerId, lastIp, lastUserAgent, lastResolution,
commonProperties);
        }
    }
```

3.6.1.6.9. Implementing destroy()

When **destroy()** is called, send a **WORKSPACE_STOPPED** event and shutdown any resources such as connection pools.

Example 3.24. AnalyticsManager.java

```
@Override
public void destroy() {
    onEvent(WORKSPACE_STOPPED, lastOwnerId, lastIp, lastUserAgent, lastResolution,
commonProperties);
}
```

Running **mvn quarkus:dev** as described in Section 3.6.1.6.4, "Running the application within a Dev Workspace" and terminating the application with **Ctrl+C** sends a **WORKSPACE_STOPPED** event to the server.

3.6.1.6.10. Packaging the Quarkus application

See the Quarkus documentation for the best instructions to package the application in a container. Build and push the container to a container registry of your choice.

Sample Dockerfile for building a Quarkus image running with JVM

Example 3.25. Dockerfile.jvm

```
FROM registry.access.redhat.com/ubi8/openjdk-11:1.11

ENV LANG='en_US.UTF-8' LANGUAGE='en_US:en'

COPY --chown=185 target/quarkus-app/lib/ /deployments/lib/
COPY --chown=185 target/quarkus-app/*.jar /deployments/
COPY --chown=185 target/quarkus-app/app/ /deployments/app/
COPY --chown=185 target/quarkus-app/quarkus/ /deployments/quarkus/

EXPOSE 8080
USER 185

ENTRYPOINT ["java", "-Dquarkus.http.host=0.0.0.0", "-
```

```
Djava.util.logging.manager=org.jboss.logmanager.LogManager", "-
Dquarkus.http.port=${DEVWORKSPACE_TELEMETRY_BACKEND_PORT}", "-jar",
"/deployments/quarkus-run.jar"]
```

To build the image, run:

```
mvn package && \
podman build -f src/main/docker/Dockerfile.jvm -t image:tag .
```

Sample Dockerfile for building a Quarkus native image

Example 3.26. Dockerfile.native

```
FROM registry.access.redhat.com/ubi8/ubi-minimal:8.5
WORKDIR /work/
RUN chown 1001 /work \
    && chmod "g+rwX" /work \
    && chown 1001:root /work
COPY --chown=1001:root target/*-runner /work/application

EXPOSE 8080
USER 1001

CMD ["./application", "-Dquarkus.http.host=0.0.0.0", "-
Dquarkus.http.port=$DEVWORKSPACE_TELEMETRY_BACKEND_PORT}"]
```

To build the image, run:

```
mvn package -Pnative -Dquarkus.native.container-build=true && \
podman build -f src/main/docker/Dockerfile.native -t image:tag .
```

3.6.1.6.11. Creating a plugin.yaml for your plugin

Create a **plugin.yaml** devfile v2 file representing a Dev Workspace plugin that runs your custom backend in a Dev Workspace Pod. For more information about devfile v2, see Devfile v2 documentation

Example 3.27. plugin.yaml

```
schemaVersion: 2.1.0
metadata:
  name: devworkspace-telemetry-backend-plugin
  version: 0.0.1
  description: A Demo telemetry backend
  displayName: Devworkspace Telemetry Backend
components:
  - name: devworkspace-telemetry-backend-plugin
    attributes:
      workspaceEnv:
        - name: DEVWORKSPACE_TELEMETRY_BACKEND_PORT
          value: '4167'
    container:
      image: YOUR IMAGE          (1)
```

```
    env:
      - name: WELCOME_MESSAGE    2
        value: 'hello world!'
```

1 Specify the container image built from Section 3.6.1.6.10, “Packaging the Quarkus application”.

2 Set the value for the **welcome.message** optional configuration property from Example 4.

Typically, the user deploys this file to a corporate web server. This guide demonstrates how to create an Apache web server on OpenShift and host the plugin there.

Create a **ConfigMap** object that references the new **plugin.yaml** file.

```
$ oc create configmap --from-file=plugin.yaml -n openshift-devspaces telemetry-plugin-yaml
```

Create a deployment, a service, and a route to expose the web server. The deployment references this **ConfigMap** object and places it in the **/var/www/html** directory.

Example 3.28. manifest.yaml

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: apache
spec:
  replicas: 1
  selector:
    matchLabels:
      app: apache
  template:
    metadata:
      labels:
        app: apache
    spec:
      volumes:
        - name: plugin-yaml
          configMap:
            name: telemetry-plugin-yaml
            defaultMode: 420
      containers:
        - name: apache
          image: 'registry.redhat.io/rhscl/httpd-24-rhel7:latest'
          ports:
            - containerPort: 8080
              protocol: TCP
          resources: {}
          volumeMounts:
            - name: plugin-yaml
              mountPath: /var/www/html
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 25%
```

```
      maxSurge: 25%
  revisionHistoryLimit: 10
  progressDeadlineSeconds: 600
---
kind: Service
apiVersion: v1
metadata:
  name: apache
spec:
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
  selector:
    app: apache
  type: ClusterIP
---
kind: Route
apiVersion: route.openshift.io/v1
metadata:
  name: apache
spec:
  host: apache-che.apps-crc.testing
  to:
    kind: Service
    name: apache
    weight: 100
  port:
    targetPort: 8080
  wildcardPolicy: None
```

```
$ oc apply -f manifest.yaml
```

Verification steps

- After the deployment has started, confirm that **plugin.yaml** is available in the web server:

```
$ curl apache-che.apps-crc.testing/plugin.yaml
```

3.6.1.6.12. Specifying the telemetry plugin in a Dev Workspace

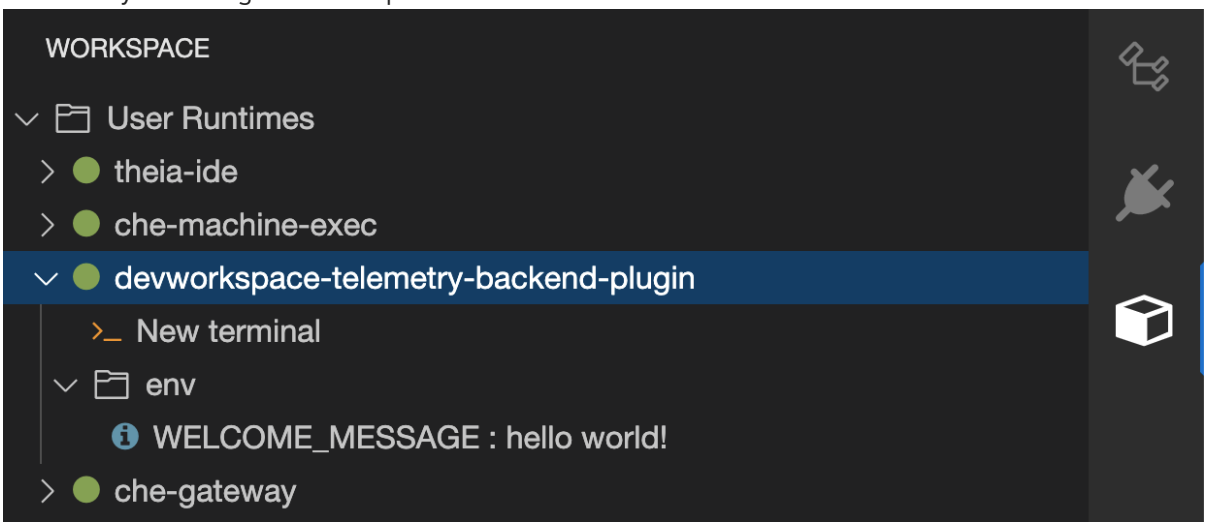
1. Add the following to the **components** field of an existing Dev Workspace:

```
components:
  ...
  - name: telemetry-plugin
    plugin:
      uri: http://apache-che.apps-crc.testing/plugin.yaml
```

2. Start the Dev Workspace from the OpenShift Dev Spaces dashboard.

Verification steps

1. Verify that the telemetry plugin container is running in the Dev Workspace pod. Here, this is verified by checking the Workspace view within the editor.



2. Edit files within the editor and observe their events in the example telemetry server's logs.

3.6.1.6.13. Applying the telemetry plugin for all Dev Workspaces

Set the telemetry plugin as a default plugin. Default plugins are applied on Dev Workspace startup for new and existing Dev Workspaces.

- Configure the **CheCluster** Custom Resource. See Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource".

```
spec:
  devEnvironments:
    defaultPlugins:
    - editor: eclipse/che-theia/next      (1)
      plugins:                            (2)
      - 'http://apache-che.apps-crc.testing/plugin.yaml'
```

(1) The editor identification to set the default plugins for.

(2) List of URLs to devfile v2 plugins.

Additional resources

- Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource" .

Verification steps

1. Start a new or existing Dev Workspace from the Red Hat OpenShift Dev Spaces dashboard.
2. Verify that the telemetry plugin is working by following the verification steps for Section 3.6.1.6.12, "Specifying the telemetry plugin in a Dev Workspace" .

3.6.2. Configuring server logging

It is possible to fine-tune the log levels of individual loggers available in the OpenShift Dev Spaces server.

The log level of the whole OpenShift Dev Spaces server is configured globally using the **cheLogLevel** configuration property of the Operator. See Section 3.1.3, "**CheCluster** Custom Resource fields reference". To set the global log level in installations not managed by the Operator, specify the **CHE_LOG_LEVEL** environment variable in the **che** ConfigMap.

It is possible to configure the log levels of the individual loggers in the OpenShift Dev Spaces server using the **CHE_LOGGER_CONFIG** environment variable.

3.6.2.1. Configuring log levels

Procedure

- Configure the **CheCluster** Custom Resource. See Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource".

```
spec:
  components:
    cheServer:
      extraProperties:
        CHE_LOGGER_CONFIG: "<key1=value1,key2=value2>" 1
```

1 Comma-separated list of key-value pairs, where keys are the names of the loggers as seen in the OpenShift Dev Spaces server log output and values are the required log levels.

Example 3.29. Configuring debug mode for the WorkspaceManager

```
spec:
  components:
    cheServer:
      extraProperties:
        CHE_LOGGER_CONFIG:
"org.eclipse.che.api.workspace.server.WorkspaceManager=DEBUG"
```

Additional resources

- Section 3.1.1, "Using dsc to configure the **CheCluster** Custom Resource during installation"
- Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource"

3.6.2.2. Logger naming

The names of the loggers follow the class names of the internal server classes that use those loggers.

3.6.2.3. Logging HTTP traffic

Procedure

- To log the HTTP traffic between the OpenShift Dev Spaces server and the API server of the Kubernetes or OpenShift cluster, configure the **CheCluster** Custom Resource. See Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource" .

```
spec:
  components:
    cheServer:
      extraProperties:
        CHE_LOGGER_CONFIG: "che.infra.request-logging=TRACE"
```

Additional resources

- Section 3.1.1, “Using dsc to configure the **CheCluster** Custom Resource during installation”
- Section 3.1.2, “Using the CLI to configure the CheCluster Custom Resource”

3.6.3. Collecting logs using dsc

An installation of Red Hat OpenShift Dev Spaces consists of several containers running in the OpenShift cluster. While it is possible to manually collect logs from each running container, **dsc** provides commands which automate the process.

Following commands are available to collect Red Hat OpenShift Dev Spaces logs from the OpenShift cluster using the **dsc** tool:

dsc server:logs

Collects existing Red Hat OpenShift Dev Spaces server logs and stores them in a directory on the local machine. By default, logs are downloaded to a temporary directory on the machine. However, this can be overwritten by specifying the **-d** parameter. For example, to download OpenShift Dev Spaces logs to the **/home/user/che-logs/** directory, use the command

```
dsc server:logs -d /home/user/che-logs/
```

When run, **dsc server:logs** prints a message in the console specifying the directory that will store the log files:

```
Red Hat OpenShift Dev Spaces logs will be available in '/tmp/chectl-logs/1648575098344'
```

If Red Hat OpenShift Dev Spaces is installed in a non-default project, **dsc server:logs** requires the **-n <NAMESPACE>** paremeter, where **<NAMESPACE>** is the OpenShift project in which Red Hat OpenShift Dev Spaces was installed. For example, to get logs from OpenShift Dev Spaces in the **my-namespace** project, use the command

```
dsc server:logs -n my-namespace
```

dsc server:deploy

Logs are automatically collected during the OpenShift Dev Spaces installation when installed using **dsc**. As with **dsc server:logs**, the directory logs are stored in can be specified using the **-d** parameter.

Additional resources

- "dsc reference documentation"

3.6.4. Monitoring with Prometheus and Grafana

You can collect and view the OpenShift Dev Spaces metrics with a running instance of Prometheus and Grafana on the cluster.

- Section 3.6.4.1, “Installing Prometheus and Grafana”
- Section 3.6.4.2, “Monitoring the Dev Workspace Operator”
- Section 3.6.4.3, “Monitoring Dev Spaces Server”

3.6.4.1. Installing Prometheus and Grafana

You can install Prometheus and Grafana by applying **template.yaml**. The **template.yaml** file in this example provides a monitoring stack of basic configuration, Deployments and Services to get started with Prometheus and Grafana.

Alternatively, you can use the Prometheus Operator and Grafana Operator.

Prerequisites

- oc

Procedure

To install Prometheus and Grafana by using **template.yaml**:

1. Create a new project, **monitoring**, for Prometheus and Grafana:

```
$ oc new-project monitoring
```

2. Apply **template.yaml** in the **monitoring** project:

```
$ oc apply -f template.yaml -n monitoring
```

Example 3.30. template.yaml

```
---
apiVersion: v1
kind: Service
metadata:
  name: grafana
  labels:
    app: grafana
spec:
  ports:
  - name: 3000-tcp
    port: 3000
    protocol: TCP
    targetPort: 3000
  selector:
    app: grafana
---
apiVersion: v1
kind: Service
metadata:
  name: prometheus
```

```
  labels:
    app: prometheus
spec:
  ports:
  - name: 9090-tcp
    port: 9090
    protocol: TCP
    targetPort: 9090
  selector:
    app: prometheus
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: grafana
  name: grafana
spec:
  selector:
    matchLabels:
      app: grafana
  template:
    metadata:
      labels:
        app: grafana
    spec:
      containers:
      - image: registry.redhat.io/rhel8/grafana:7
        name: grafana
        ports:
        - containerPort: 3000
          protocol: TCP
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: prometheus
  name: prometheus
spec:
  selector:
    matchLabels:
      app: prometheus
  template:
    metadata:
      labels:
        app: prometheus
    spec:
      serviceAccountName: prometheus
      containers:
      - image: quay.io/prometheus/prometheus:v2.36.0
        name: prometheus
        ports:
        - containerPort: 9090
          protocol: TCP
        volumeMounts:
```

```
        - mountPath: /prometheus
          name: volume-data
        - mountPath: /etc/prometheus/prometheus.yml
          name: volume-config
          subPath: prometheus.yml
      volumes:
      - emptyDir: {}
        name: volume-data
      - configMap:
          defaultMode: 420
          name: prometheus-config
        name: volume-config
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
data:
  prometheus.yml: ""
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: prometheus
---
```

Additional resources

- First steps with Prometheus
- Installing Grafana

3.6.4.2. Monitoring the Dev Workspace Operator

You can configure an example monitoring stack to process metrics exposed by the Dev Workspace Operator.

3.6.4.2.1. Collecting Dev Workspace Operator metrics with Prometheus

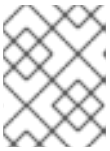
To use Prometheus to collect, store, and query metrics about the Dev Workspace Operator:

Prerequisites

- The **devworkspace-controller-metrics** Service is exposing metrics on port **8443**. This is preconfigured by default.
- The **devworkspace-webhookserver** Service is exposing metrics on port **9443**. This is preconfigured by default.
- Prometheus 2.26.0 or later is running. The Prometheus console is running on port **9090** with a corresponding Service. See First steps with Prometheus.

Procedure

1. Create a ClusterRoleBinding to bind the ServiceAccount associated with Prometheus to the devworkspace-controller-metrics-reader ClusterRole. For the example monitoring stack, the name of the ServiceAccount to be used is **prometheus**.



NOTE

Without the ClusterRoleBinding, you cannot access Dev Workspace metrics because access is protected with role-based access control (RBAC).

Example 3.31. ClusterRoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: devworkspace-controller-metrics-binding
subjects:
  - kind: ServiceAccount
    name: prometheus
    namespace: monitoring
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: devworkspace-controller-metrics-reader
```

2. Configure Prometheus to scrape metrics from port **8443** exposed by the **devworkspace-controller-metrics** Service and from port **9443** exposed by the **devworkspace-webhookserver** Service.

NOTE

The example monitoring stack already creates the **prometheus-config** ConfigMap with an empty configuration. To provide the Prometheus configuration details, edit the **data** field of the ConfigMap.

Example 3.32. Prometheus configuration

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
  namespace: monitoring
data:
  prometheus.yml: |-
    global:
      scrape_interval: 5s 1
      evaluation_interval: 5s 2
    scrape_configs: 3
      - job_name: 'DevWorkspace'
        scheme: https
        authorization:
          type: Bearer
          credentials_file: '/var/run/secrets/kubernetes.io/serviceaccount/token'
```

```
      tls_config:
        insecure_skip_verify: true
      static_configs:
        - targets: ['devworkspace-controller-metrics.<DWO_project>:8443'] 4
    - job_name: 'DevWorkspace webhooks'
      scheme: https
      authorization:
        type: Bearer
        credentials_file: '/var/run/secrets/kubernetes.io/serviceaccount/token'
      tls_config:
        insecure_skip_verify: true
      static_configs:
        - targets: ['devworkspace-webhookserver.<DWO_project>:9443'] 5
```

1. The rate at which a target is scraped.
2. The rate at which the recording and alerting rules are re-checked.
3. The resources that Prometheus monitors. In the default configuration, two jobs, **DevWorkspace** and **DevWorkspace webhooks**, scrape the time series data exposed by the **devworkspace-controller-metrics** and **devworkspace-webhookserver** Services.
4. The scrape target for the metrics from port **8443**. Replace ***<DWO_project>*** with the project where the **devworkspace-controller-metrics Service** is located.
5. The scrape target for the metrics from port **9443**. Replace ***<DWO_project>*** with the project where the **devworkspace-webhookserver Service** is located.

3. Scale the **Prometheus** Deployment down and up to read the updated ConfigMap from the previous step.

```
$ oc scale --replicas=0 deployment/prometheus -n monitoring && oc scale --replicas=1 deployment/prometheus -n monitoring
```

Verification

1. Use port forwarding to access the **Prometheus** Service locally:

```
$ oc port-forward svc/prometheus 9090:9090 -n monitoring
```

2. Verify that all targets are up by viewing the targets endpoint at **localhost:9090/targets**.
3. Use the Prometheus console to view and query metrics:
   - View metrics at **localhost:9090/metrics**.
   - Query metrics from **localhost:9090/graph**.
     For more information, see Using the expression browser.

Additional resources

- Configuring Prometheus

- Querying Prometheus
- Prometheus metric types

3.6.4.2.2. Dev Workspace-specific metrics

The following tables describe the Dev Workspace-specific metrics exposed by the **devworkspace-controller-metrics** Service.

Table 3.20. Metrics

Name	Type	Description	Labels
devworkspace_started_total	Counter	Number of Dev Workspace starting events.	**source**, **routingclass**
devworkspace_started_success_total	Counter	Number of Dev Workspaces successfully entering the **Running** phase.	**source**, **routingclass**
devworkspace_fail_total	Counter	Number of failed Dev Workspaces.	**source**, **reason**
devworkspace_startup_time	Histogram	Total time taken to start a Dev Workspace, in seconds.	**source**, **routingclass**

Table 3.21. Labels

Name	Description	Values
source	The **controller.devfile.io/devworkspace-source** label of the Dev Workspace.	**string**
routingclass	The **spec.routingclass** of the Dev Workspace.	**"basic|cluster|cluster-tls|web-terminal"**
reason	The workspace startup failure reason.	**"BadRequest|InfrastructureFailure|Unknown"**

Table 3.22. Startup failure reasons

Name	Description
BadRequest	Startup failure due to an invalid devfile used to create a Dev Workspace.

Name	Description
InfrastructureFailure	Startup failure due to the following errors: **CreateContainerError**, **RunContainerError**, **FailedScheduling**, **FailedMount**.
Unknown	Unknown failure reason.

3.6.4.2.3. Viewing Dev Workspace Operator metrics on Grafana dashboards

To view the Dev Workspace Operator metrics on Grafana with the example dashboard:

Prerequisites

- Prometheus is collecting metrics. See Section 3.6.4.2.1, "Collecting Dev Workspace Operator metrics with Prometheus".
- Grafana version 7.5.3 or later.
- Grafana is running on port **3000** with a corresponding Service. See Installing Grafana.

Procedure

1. Add the data source for the Prometheus instance. See Creating a Prometheus data source .
2. Import the example **grafana-dashboard.json** dashboard.

Verification steps

- Use the Grafana console to view the Dev Workspace Operator metrics dashboard. See Section 3.6.4.2.4, "Grafana dashboard for the Dev Workspace Operator" .

Additional resources

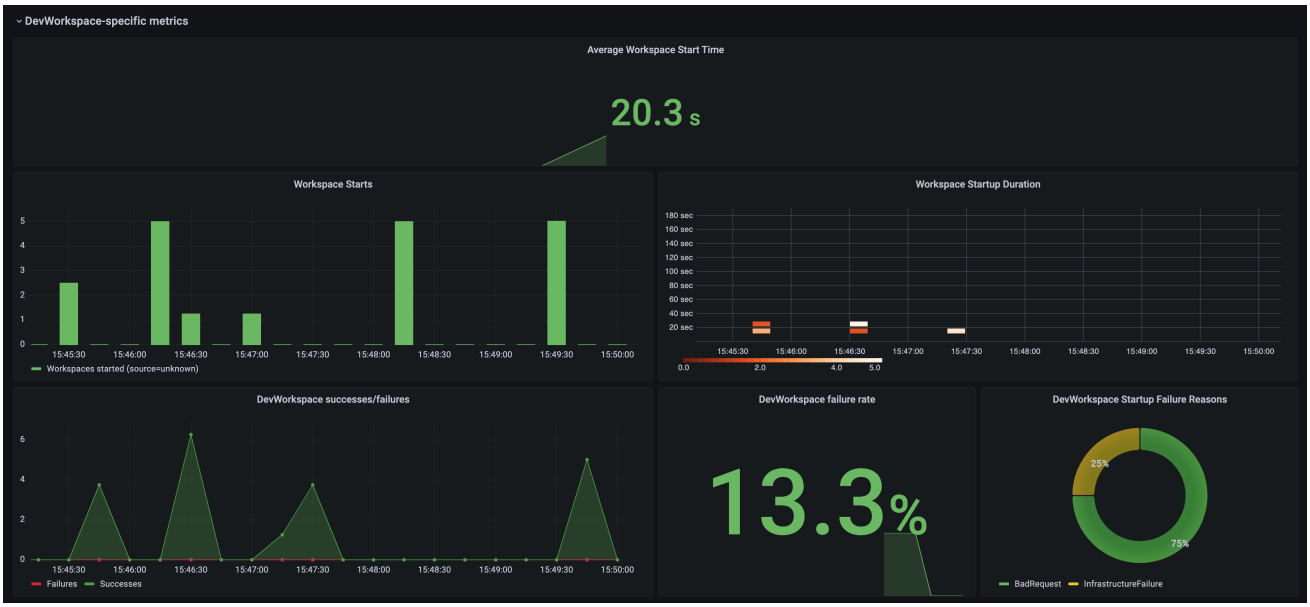
- Prometheus data source
- Import dashboard

3.6.4.2.4. Grafana dashboard for the Dev Workspace Operator

The example Grafana dashboard based on **grafana-dashboard.json** displays the following metrics from the Dev Workspace Operator.

The Dev Workspace-specific metrics panel

Figure 3.1. The Dev Workspace-specific metrics panel



Average workspace start time

The average workspace startup duration.

Workspace starts

The number of successful and failed workspace startups.

Workspace startup duration

A heatmap that displays workspace startup duration.

Dev Workspace successes / failures

A comparison between successful and failed Dev Workspace startups.

Dev Workspace failure rate

The ratio between the number of failed workspace startups and the number of total workspace startups.

Dev Workspace startup failure reasons

A pie chart that displays the distribution of workspace startup failures:

- **BadRequest**
- **InfrastructureFailure**
- **Unknown**

The Operator metrics panel (part 1)

Figure 3.2. The Operator metrics panel (part 1)

Webhooks in flight

A comparison between the number of different webhook requests.

Work queue duration

A heatmap that displays how long the reconcile requests stay in the work queue before they are handled.

Webhooks latency (/mutate)

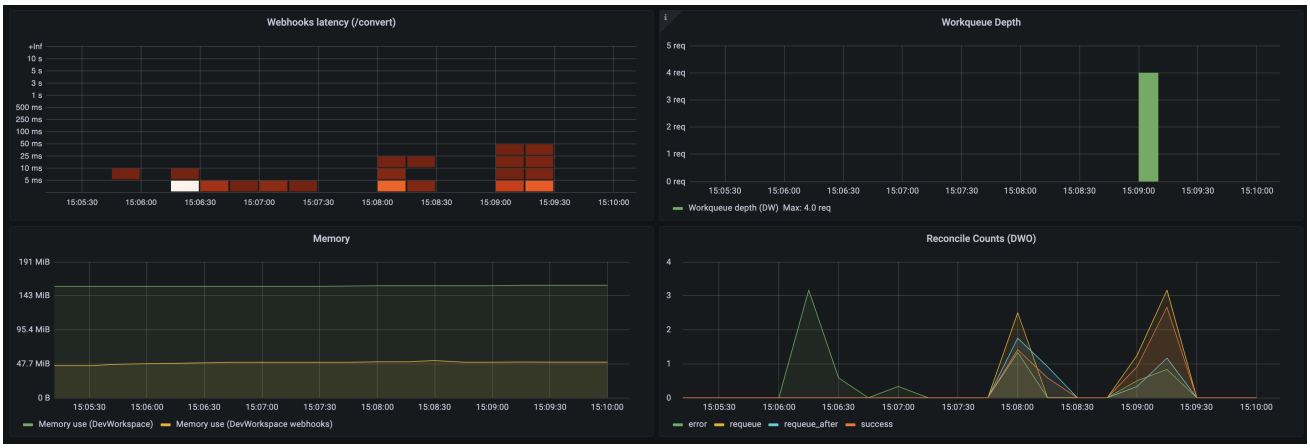
A heatmap that displays the /**mutate** webhook latency.

Reconcile time

A heatmap that displays the reconcile duration.

The Operator metrics panel (part 2)

Figure 3.3. The Operator metrics panel (part 2)



Webhooks latency (/convert)

A heatmap that displays the /**convert** webhook latency.

Work queue depth

The number of reconcile requests that are in the work queue.

Memory

Memory usage for the Dev Workspace controller and the Dev Workspace webhook server.

Reconcile counts (DWO)

The average per-second number of reconcile counts for the Dev Workspace controller.

3.6.4.3. Monitoring Dev Spaces Server

You can configure OpenShift Dev Spaces to expose JVM metrics such as JVM memory and class loading for OpenShift Dev Spaces Server.

3.6.4.3.1. Enabling and exposing OpenShift Dev Spaces Server metrics

OpenShift Dev Spaces exposes the JVM metrics on port **8087** of the **che-host** Service. You can configure this behaviour.

Procedure

- Configure the **CheCluster** Custom Resource. See Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource".

```
spec:
  components:
    metrics:
      enable: <boolean> 1
```

1 **true** to enable, **false** to disable.

3.6.4.3.2. Collecting OpenShift Dev Spaces Server metrics with Prometheus

To use Prometheus to collect, store, and query JVM metrics for OpenShift Dev Spaces Server:

Prerequisites

- OpenShift Dev Spaces is exposing metrics on port **8087**. See Enabling and exposing OpenShift Dev Spaces server JVM metrics.
- Prometheus 2.26.0 or later is running. The Prometheus console is running on port **9090** with a corresponding Service. See First steps with Prometheus.

Procedure

1. Configure Prometheus to scrape metrics from port **8087**.

NOTE

The example monitoring stack already creates the **prometheus-config** ConfigMap with an empty configuration. To provide the Prometheus configuration details, edit the **data** field of the ConfigMap.

Example 3.33. Prometheus configuration

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
data:
  prometheus.yml: |-
    global:
```

```
    scrape_interval:     5s          (1)
    evaluation_interval: 5s          (2)
  scrape_configs:                    (3)
    - job_name: 'OpenShift Dev Spaces Server'
      static_configs:
        - targets: ['che-host.<OpenShift Dev Spaces_project>:8087'] (4)
```

(1) The rate at which a target is scraped.

(2) The rate at which the recording and alerting rules are re-checked.

(3) The resources that Prometheus monitors. In the default configuration, a single job, **OpenShift Dev Spaces Server**, scrapes the time series data exposed by OpenShift Dev Spaces Server.

(4) The scrape target for the metrics from port **8087**. Replace ***<OpenShift Dev Spaces_project>*** with the OpenShift Dev Spaces project. The default OpenShift Dev Spaces project is **openshift-devspaces**.

2. Scale the **Prometheus** Deployment down and up to read the updated ConfigMap from the previous step.

```
$ oc scale --replicas=0 deployment/prometheus -n monitoring && oc scale --replicas=1 deployment/prometheus -n monitoring
```

Verification

1. Use port forwarding to access the **Prometheus** Service locally:

```
$ oc port-forward svc/prometheus 9090:9090 -n monitoring
```

2. Verify that all targets are up by viewing the **targets** endpoint at **localhost:9090/targets**.
3. Use the Prometheus console to view and query metrics:
   - View metrics at **localhost:9090/metrics**.
   - Query metrics from **localhost:9090/graph**.
     For more information, see Using the expression browser.

Additional resources

- Configuring Prometheus
- Querying Prometheus
- Prometheus metric types

3.6.4.3.3. Viewing OpenShift Dev Spaces Server metrics on Grafana dashboards

To view the OpenShift Dev Spaces Server metrics on Grafana:

Prerequisites

Prerequisites

- Prometheus is collecting metrics on the OpenShift Dev Spaces cluster. See Section 3.6.4, "Monitoring with Prometheus and Grafana".
- Grafana 6.0 or later is running on port **3000** with a corresponding Service. See Installing Grafana.

Procedure

1. Add the data source for the Prometheus instance. See Creating a Prometheus data source .
2. Import the example dashboard. See Import dashboard.
3. View the OpenShift Dev Spaces JVM metrics in the Grafana console:

Figure 3.4. OpenShift Dev Spaces server JVM dashboard

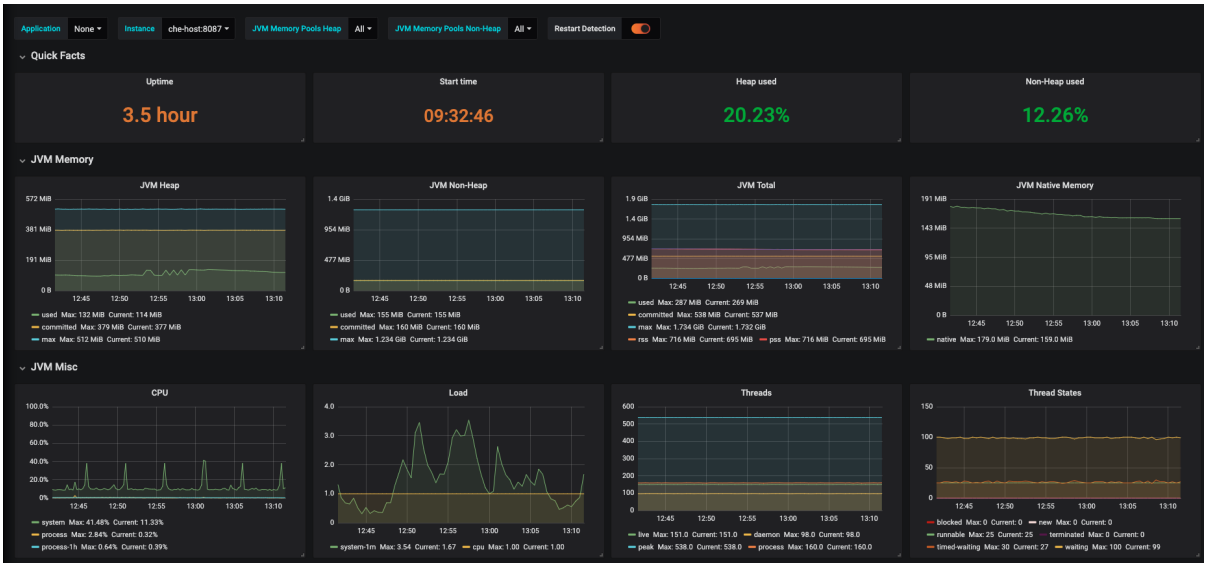


Figure 3.5. Quick Facts

Figure 3.6. JVM Memory

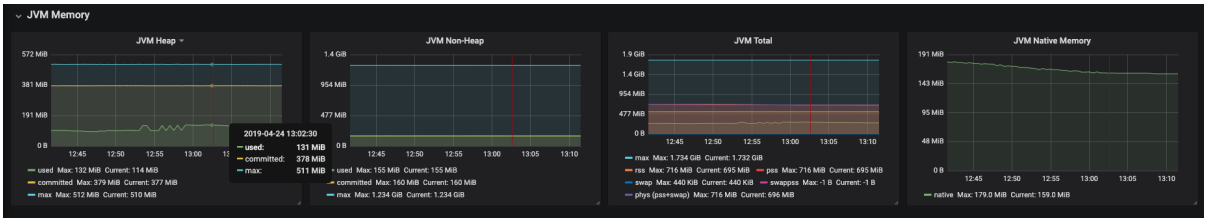


Figure 3.7. JVM Misc

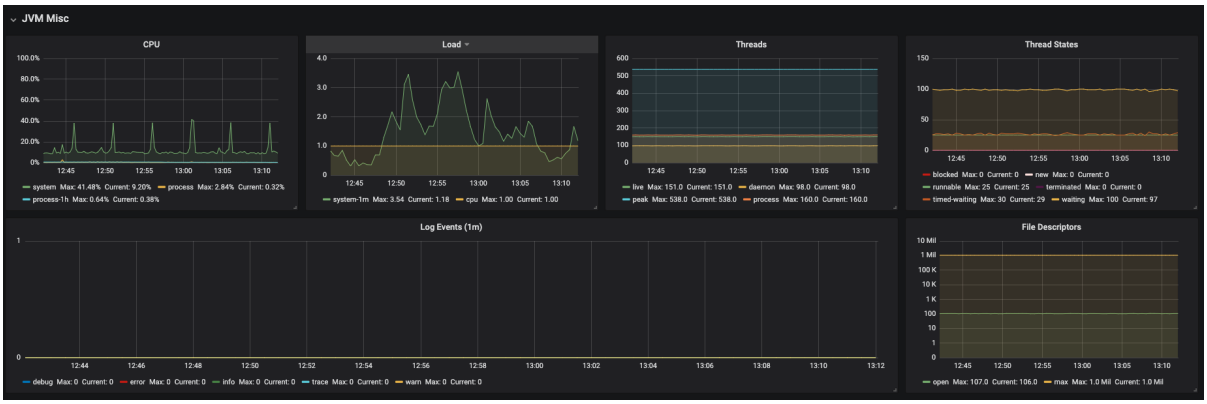


Figure 3.8. JVM Memory Pools (heap)

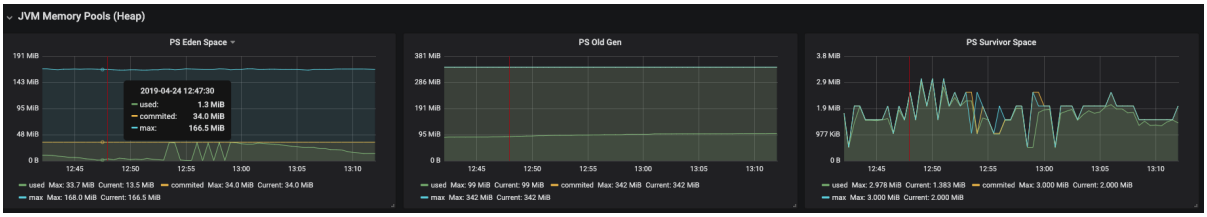


Figure 3.9. JVM Memory Pools (Non-Heap)

Figure 3.10. Garbage Collection

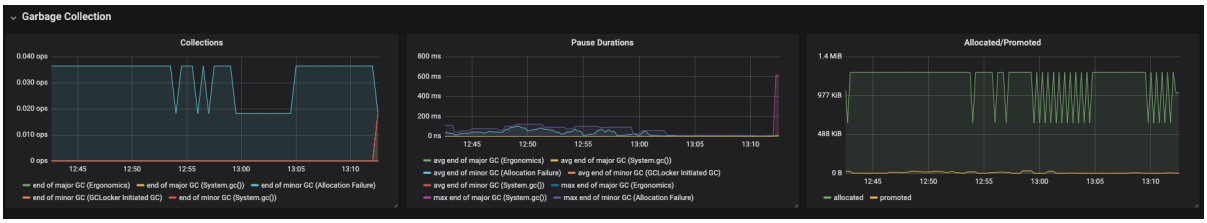


Figure 3.11. Class loading

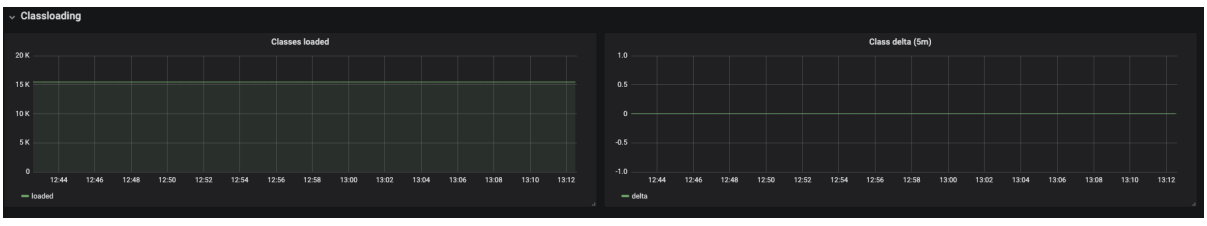
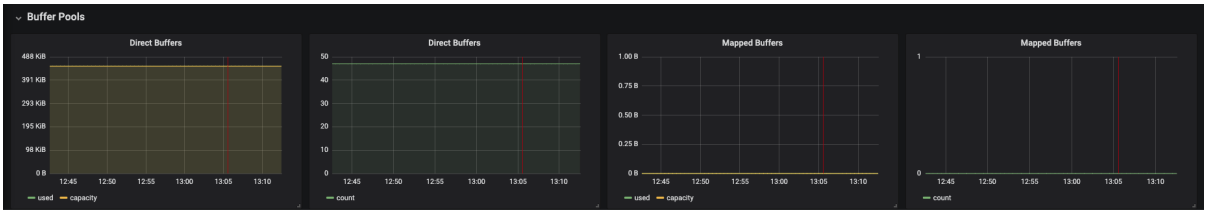


Figure 3.12. Buffer Pools



3.7. CONFIGURING NETWORKING

3.7.1. Configuring network policies

By default, all Pods in a OpenShift cluster can communicate with each other even if they are in different namespaces. In the context of OpenShift Dev Spaces, this makes it possible for a workspace Pod in one user project to send traffic to another workspace Pod in a different user project.

For security, multitenant isolation could be configured by using NetworkPolicy objects to restrict all incoming communication to Pods in a user project. However, Pods in the OpenShift Dev Spaces project must be able to communicate with Pods in user projects.

Prerequisites

- The OpenShift cluster has network restrictions such as multitenant isolation.

Procedure

- Apply the **allow-from-openshift-devspaces** NetworkPolicy to each user project. The **allow-from-openshift-devspaces** NetworkPolicy allows incoming traffic from the OpenShift Dev Spaces namespace to all Pods in the user project.

Example 3.34. allow-from-openshift-devspaces.yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-openshift-devspaces
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: openshift-devspaces  1
  podSelector: {}  2
  policyTypes:
  - Ingress
```

1. The OpenShift Dev Spaces namespace. The default is **openshift-devspaces**.
2. The empty **podSelector** selects all Pods in the project.

Additional resources

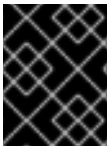
- Section 3.2, "Configuring projects"
- Network isolation
- Configuring multitenant isolation with network policy

3.7.2. Configuring Dev Spaces hostname

This procedure describes how to configure OpenShift Dev Spaces to use custom hostname.

Prerequisites

- An active **oc** session with administrative permissions to the destination OpenShift cluster. See Getting started with the CLI.
- The certificate and the private key files are generated.



IMPORTANT

To generate the pair of a private key and certificate, the same certification authority (CA) must be used as for other OpenShift Dev Spaces hosts.

IMPORTANT

Ask a DNS provider to point the custom hostname to the cluster ingress.

Procedure

1. Pre-create a project for OpenShift Dev Spaces:

```
$ oc create project openshift-devspaces
```

2. Create a TLS secret:

```
$ oc create secret TLS <tls_secret_name> \ 1
--key <key_file> \ 2
--cert <cert_file> \ 3
-n openshift-devspaces
```

1. The TLS secret name
2. A file with the private key
3. A file with the certificate

3. Add the required labels to the secret:

```
$ oc label secret <tls_secret_name> \ 1
app.kubernetes.io/part-of=che.eclipse.org -n openshift-devspaces
```

1. The TLS secret name

4. Configure the **CheCluster** Custom Resource. See Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource".

```
spec:
  networking:
    hostname: <hostname>  1
    tlsSecretName: <secret>  2
```

1. Custom Red Hat OpenShift Dev Spaces server hostname
2. The TLS secret name

5. If OpenShift Dev Spaces has been already deployed, wait until the rollout of all OpenShift Dev Spaces components finishes.

Additional resources

- Section 3.1.1, "Using dsc to configure the **CheCluster** Custom Resource during installation"
- Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource"

3.7.3. Importing untrusted TLS certificates to Dev Spaces

OpenShift Dev Spaces components communications with external services are encrypted with TLS. They require TLS certificates signed by trusted Certificate Authorities (CA). Therefore, you must import into OpenShift Dev Spaces all untrusted CA chains in use by an external service such as:

- A proxy
- An identity provider (OIDC)
- A source code repositories provider (Git)

OpenShift Dev Spaces uses labeled config maps in OpenShift Dev Spaces project as sources for TLS certificates. The config maps can have an arbitrary amount of keys with a random amount of certificates each.

NOTE

When an OpenShift cluster contains cluster-wide trusted CA certificates added through the cluster-wide-proxy configuration, OpenShift Dev Spaces Operator detects them and automatically injects them into a config map with the **config.openshift.io/inject-trusted-cabundle="true"** label. Based on this annotation, OpenShift automatically injects the cluster-wide trusted CA certificates inside the **ca-bundle.crt** key of the config map.

Prerequisites

- An active **oc** session with administrative permissions to the destination OpenShift cluster. See Getting started with the CLI.
- The **openshift-devspaces** project exists.
- For each CA chain to import: the root CA and intermediate certificates, in PEM format, in a **ca-cert-for-devspaces-*<count>*.pem** file.

Procedure

1. Concatenate all CA chains PEM files to import, into the **custom-ca-certificates.pem** file, and remove the return character that is incompatible with the Java truststore.

```
$ cat ca-cert-for-{prod-id-short}-*.pem | tr -d '\r' > custom-ca-certificates.pem
```

2. Create the **custom-ca-certificates** config map with the required TLS certificates:

```
$ oc create configmap custom-ca-certificates \
    --from-file=custom-ca-certificates.pem \
    --namespace=openshift-devspaces
```

3. Label the **custom-ca-certificates** config map:

```
$ oc label configmap custom-ca-certificates \
    app.kubernetes.io/component=ca-bundle \
    app.kubernetes.io/part-of=che.eclipse.org \
    --namespace=openshift-devspaces
```

4. Deploy OpenShift Dev Spaces if it hasn't been deployed before. Otherwise, wait until the rollout of OpenShift Dev Spaces components finishes.

5. Restart running workspaces for the changes to take effect.

Verification steps

1. Verify that the config map contains your custom CA certificates. This command returns your custom CA certificates in PEM format:

```
$ oc get configmap \
    --namespace=openshift-devspaces \
    --output='jsonpath={.items[0:].data.custom-ca-certificates\.pem}' \
    --selector=app.kubernetes.io/component=ca-bundle,app.kubernetes.io/part-
of=che.eclipse.org
```

2. Verify OpenShift Dev Spaces pod contains a volume mounting the **ca-certs-merged** config map:

```
$ oc get pod \
    --selector=app.kubernetes.io/component=devspaces \
    --output='jsonpath={.items[0].spec.volumes[0:].configMap.name}' \
    --namespace=openshift-devspaces \
    | grep ca-certs-merged
```

3. Verify the OpenShift Dev Spaces server container has your custom CA certificates. This command returns your custom CA certificates in PEM format:

```
$ oc exec -t deploy/devspaces \
    --namespace=openshift-devspaces \
    -- cat /public-certs/custom-ca-certificates.pem
```

4. Verify in the OpenShift Dev Spaces server logs that the imported certificates count is not null:

```
$ oc logs deploy/devspaces --namespace=openshift-devspaces \
    | grep custom-ca-certificates.pem
```

5. List the SHA256 fingerprints of your certificates:

```
$ for certificate in ca-cert*.pem ;
  do openssl x509 -in $certificate -digest -sha256 -fingerprint -noout | cut -d= -f2;
  done
```

6. Verify that OpenShift Dev Spaces server Java truststore contains certificates with the same fingerprint:

```
$ oc exec -t deploy/devspaces --namespace=openshift-devspaces -- \
    keytool -list -keystore /home/user/cacerts \
    | grep --after-context=1 custom-ca-certificates.pem
```

7. Start a workspace, get the project name in which it has been created: *<workspace_namespace>*, and wait for the workspace to be started.

8. Verify that the **che-trusted-ca-certs** config map contains your custom CA certificates. This command returns your custom CA certificates in PEM format:

```
$ oc get configmap che-trusted-ca-certs \
    --namespace=<workspace_namespace> \
    --output='jsonpath={.data.custom-ca-certificates\.custom-ca-certificates\.pem}'
```

9. Verify that the workspace pod mounts the **che-trusted-ca-certs** config map:

```
$ oc get pod \
    --namespace=<workspace_namespace> \
    --selector='controller.devfile.io/devworkspace_name=<workspace_name>' \
    --output='jsonpath={.items[0:].spec.volumes[0:].configMap.name}' \
    | grep che-trusted-ca-certs
```

10. Verify that the **universal-developer-image** container (or the container defined in the workspace devfile) mounts the **che-trusted-ca-certs** volume:

```
$ oc get pod \
    --namespace=<workspace_namespace> \
    --selector='controller.devfile.io/devworkspace_name=<workspace_name>' \
    --output='jsonpath={.items[0:].spec.containers[0:]}' \
    | jq 'select (.volumeMounts[].name == "che-trusted-ca-certs") | .name'
```

11. Get the workspace pod name *<workspace_pod_name>*:

```
$ oc get pod \
    --namespace=<workspace_namespace> \
    --selector='controller.devfile.io/devworkspace_name=<workspace_name>' \
    --output='jsonpath={.items[0:].metadata.name}' \
```

12. Verify that the workspace container has your custom CA certificates. This command returns your custom CA certificates in PEM format:

```
$ oc exec <workspace_pod_name> \
    --namespace=<workspace_namespace> \
    -- cat /public-certs/custom-ca-certificates.custom-ca-certificates.pem
```

Additional resources

- Section 3.4.3, "Git with self-signed certificates".

3.7.4. Configuring OpenShift Route

You can configure OpenShift Route labels and annotations, if your organization requires them.

Prerequisites

- An active **oc** session with administrative permissions to the destination OpenShift cluster. See Getting started with the CLI.
- An instance of OpenShift Dev Spaces running in OpenShift.

Procedure

- Configure the **CheCluster** Custom Resource. See Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource".

```
spec:
  networking:
    labels: <labels> 1
    annotations: <annotations> 2
```

1. An unstructured key value map of labels for OpenShift Route.
2. An unstructured key value map of annotations for OpenShift Route.

Additional resources

- Section 3.1.1, "Using dsc to configure the **CheCluster** Custom Resource during installation"
- Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource"

3.7.5. Configuring OpenShift Route

You can configure labels, annotations, and domains for OpenShift Route to work with Router Sharding.

Prerequisites

- An active **oc** session with administrative permissions to the OpenShift cluster. See Getting started with the OpenShift CLI.
- **dsc**. See: Section 2.1, "Installing the dsc management tool".

Procedure

- Configure the **CheCluster** Custom Resource. See Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource".

```
spec:
  networking:
    labels: <labels> 1
    domain: <domain> 2
    annotations: <annotations> 3
```

1. An unstructured key value map of labels that the target ingress controller uses to filter the set of Routes to service.
2. The DNS name serviced by the target ingress controller.
3. An unstructured key value map stored with a resource.

Additional resources

- Section 3.1.1, "Using dsc to configure the **CheCluster** Custom Resource during installation"
- Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource"

3.8. CONFIGURING STORAGE

WARNING

OpenShift Dev Spaces does not support the Network File System (NFS) protocol.

- Section 3.8.1, "Installing Dev Spaces using storage classes"

3.8.1. Installing Dev Spaces using storage classes

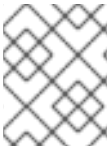
To configure OpenShift Dev Spaces to use a configured infrastructure storage, install OpenShift Dev Spaces using storage classes. This is especially useful when a user wants to bind a persistent volume provided by a non-default provisioner. To do so, a user binds this storage for the OpenShift Dev Spaces data saving and sets the parameters for that storage. These parameters can determine the following:

- A special host path
- A storage capacity
- A volume mod
- Mount options
- A file system
- An access mode

- A storage type
- And many others

OpenShift Dev Spaces has two components that require persistent volumes to store data:

- A PostgreSQL database.
- A OpenShift Dev Spaces workspaces. OpenShift Dev Spaces workspaces store source code using volumes, for example /**projects** volume.



NOTE

OpenShift Dev Spaces workspaces source code is stored in the persistent volume only if a workspace is not ephemeral.

Persistent volume claims facts:

- OpenShift Dev Spaces does not create persistent volumes in the infrastructure.
- OpenShift Dev Spaces uses persistent volume claims (PVC) to mount persistent volumes.
- The OpenShift Dev Spaces server creates persistent volume claims.
  A user defines a storage class name in the OpenShift Dev Spaces configuration to use the storage classes feature in the OpenShift Dev Spaces PVC. With storage classes, a user configures infrastructure storage in a flexible way with additional storage parameters. It is also possible to bind a static provisioned persistent volumes to the OpenShift Dev Spaces PVC using the class name.

Procedure

Use CheCluster Custom Resource definition to define storage classes:

1. Define storage class names: configure the **CheCluster** Custom Resource, and install OpenShift Dev Spaces. See Section 3.1.1, “Using dsc to configure the **CheCluster** Custom Resource during installation”.

```
spec:
  components:
    database:
      pvc:
        # keep blank unless you need to use a non default storage class for PostgreSQL PVC
        storageClass: 'postgres-storage'
  devEnvironments:
    storage:
      pvc:
        # keep blank unless you need to use a non default storage class for workspace PVC(s)
        storageClass: 'workspace-storage'
```

2. Define the persistent volume for a PostgreSQL database in a **che-postgres-pv.yaml** file:

   che-postgres-pv.yaml file

```
apiVersion: v1
kind: PersistentVolume
metadata:
```

```
  name: postgres-pv-volume
  labels:
    type: local
spec:
  storageClassName: postgres-storage
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/data/che/postgres"
```

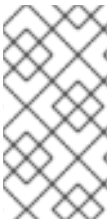
3. Define the persistent volume for a OpenShift Dev Spaces workspace in a **che-postgres-pv.yaml** file:

   che-workspace-pv.yaml file

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: workspace-pv-volume
  labels:
    type: local
spec:
  storageClassName: workspace-storage
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/data/che/workspace"
```

4. Bind the two persistent volumes:

```
$ kubectl apply -f che-workspace-pv.yaml -f che-postgres-pv.yaml
```



NOTE

You must provide valid file permissions for volumes. You can do it using storage class configuration or manually. To manually define permissions, define **storageClass#mountOptions uid** and **gid**. PostgreSQL volume requires **uid=26** and **gid=26**.

Additional resources

- Section 3.1.1, “Using dsc to configure the **CheCluster** Custom Resource during installation”
- Section 3.1.2, “Using the CLI to configure the CheCluster Custom Resource”

3.9. MANAGING IDENTITIES AND AUTHORIZATIONS

This section describes different aspects of managing identities and authorizations of Red Hat OpenShift Dev Spaces.

3.9.1. Configuring OAuth for Git providers

You can configure OAuth between OpenShift Dev Spaces and Git providers, enabling users to work with remote Git repositories:

- Section 3.9.1.1, "Configuring OAuth 2.0 for GitHub"
- Section 3.9.1.2, "Configuring OAuth 2.0 for GitLab"
- Configuring OAuth 1.0 for a Bitbucket Server or OAuth 2.0 for the Bitbucket Cloud
- Section 3.9.1.5, "Configuring OAuth 2.0 for Microsoft Azure DevOps Services"

3.9.1.1. Configuring OAuth 2.0 for GitHub

To enable users to work with a remote Git repository that is hosted on GitHub:

1. Set up the GitHub OAuth App (OAuth 2.0).
2. Apply the GitHub OAuth App Secret.

3.9.1.1.1. Setting up the GitHub OAuth App

Set up a GitHub OAuth App using OAuth 2.0.

Prerequisites

- You are logged in to GitHub.
- **base64** is installed in the operating system you are using.

Procedure

1. Go to https://github.com/settings/applications/new.
2. Enter the following values:
   a. **Application name**: **OpenShift Dev Spaces**
   b. **Homepage URL**: **https://devspaces-*<openshift_deployment_name>*.*<domain_name>*/**
   c. **Authorization callback URL**: **https://devspaces-*<openshift_deployment_name>*.*<domain_name>*/api/oauth/callback**
3. Click **Register application**.
4. Click **Generate new client secret**.
5. Copy the **GitHub OAuth Client ID** and encode it to Base64 for use when applying the GitHub OAuth App Secret:

```
$ echo -n '<github_oauth_client_id>' | base64
```

6. Copy the **GitHub OAuth Client Secret** and encode it to Base64 for use when applying the GitHub OAuth App Secret:

```
$ echo -n '<github_oauth_client_secret>' | base64
```

Additional resources

- GitHub Docs: Creating an OAuth App

3.9.1.1.2. Applying the GitHub OAuth App Secret

Prepare and apply the GitHub OAuth App Secret.

Prerequisites

- Setting up the GitHub OAuth App is completed.
- The Base64-encoded values, which were generated when setting up the GitHub OAuth App, are prepared:
  - **GitHub OAuth Client ID**
  - **GitHub OAuth Client Secret**
- An active **oc** session with administrative permissions to the destination OpenShift cluster. See Getting started with the CLI.

Procedure

1. Prepare the Secret:

```
kind: Secret
apiVersion: v1
metadata:
  name: github-oauth-config
  namespace: openshift-devspaces (1)
  labels:
    app.kubernetes.io/part-of: che.eclipse.org
    app.kubernetes.io/component: oauth-scm-configuration
  annotations:
    che.eclipse.org/oauth-scm-server: github
    che.eclipse.org/scm-server-endpoint: <github_server_url> (2)
type: Opaque
data:
  id: <Base64_GitHub_OAuth_Client_ID> (3)
  secret: <Base64_GitHub_OAuth_Client_Secret> (4)
```

(1) The OpenShift Dev Spaces namespace. The default is **openshift-devspaces**.

(2) A **GitHub Enterprise Server URL** By default **https://github.com** is used for the **SAAS** version.

(3) The Base64-encoded **GitHub OAuth Client ID**.

(4) The Base64-encoded **GitHub OAuth Client Secret**

2. Apply the Secret:

```
$ oc apply -f - <<EOF
<Secret_prepared_in_the_previous_step>
EOF
```

3. Verify in the output that the Secret is created.

3.9.1.2. Configuring OAuth 2.0 for GitLab

To enable users to work with a remote Git repository that is hosted using a GitLab instance:

1. Set up the GitLab authorized application (OAuth 2.0).
2. Apply the GitLab authorized application Secret.

3.9.1.2.1. Setting up the GitLab authorized application

Set up a GitLab authorized application using OAuth 2.0.

Prerequisites

- You are logged in to GitLab.
- **base64** is installed in the operating system you are using.

Procedure

1. Click your avatar and go to **Edit profile → Applications**.
2. Enter **OpenShift Dev Spaces** as the **Name**.
3. Enter **https://devspaces-*<openshift_deployment_name>*.*<domain_name>*/api/oauth/callback** as the **Redirect URI**.
4. Check the **Confidential** and **Expire access tokens** checkboxes.
5. Under **Scopes**, check the **api**, **write_repository**, and **openid** checkboxes.
6. Click **Save application**.
7. Copy the **GitLab Application ID** and encode it to Base64 for use when applying the GitLab-authorized application Secret:

```
$ echo -n '<gitlab_application_id>' | base64
```

8. Copy the **GitLab Client Secret** and encode it to Base64 for use when applying the GitLab-authorized application Secret:

```
$ echo -n '<gitlab_client_secret>' | base64
```

Additional resources

- GitLab Docs: Authorized applications

3.9.1.2.2. Applying the GitLab-authorized application Secret

Prepare and apply the GitLab-authorized application Secret.

Prerequisites

- Setting up the GitLab authorized application is completed.
- The Base64-encoded values, which were generated when setting up the GitLab authorized application, are prepared:
  - **GitLab Application ID**
  - **GitLab Client Secret**
- An active **oc** session with administrative permissions to the destination OpenShift cluster. See Getting started with the CLI.

Procedure

1. Prepare the Secret:

```
kind: Secret
apiVersion: v1
metadata:
  name: gitlab-oauth-config
  namespace: openshift-devspaces (1)
  labels:
    app.kubernetes.io/part-of: che.eclipse.org
    app.kubernetes.io/component: oauth-scm-configuration
  annotations:
    che.eclipse.org/oauth-scm-server: gitlab
    che.eclipse.org/scm-server-endpoint: <gitlab_server_url> (2)
type: Opaque
data:
  id: <Base64_GitLab_Application_ID> (3)
  secret: <Base64_GitLab_Client_Secret> (4)
```

(1) The OpenShift Dev Spaces namespace. The default is **openshift-devspaces**.

(2) The **GitLab server URL** Use **https://gitlab.com** for the **SAAS** version.

(3) The Base64-encoded **GitLab Application ID**.

(4) The Base64-encoded **GitLab Client Secret**.

2. Apply the Secret:

```
$ oc apply -f - <<EOF
<Secret_prepared_in_the_previous_step>
EOF
```

3. Verify in the output that the Secret is created.

3.9.1.3. Configuring OAuth 1.0 for a Bitbucket Server

To enable users to work with a remote Git repository that is hosted on a Bitbucket Server:

1. Set up an application link (OAuth 1.0) on the Bitbucket Server.
2. Apply an application link Secret for the Bitbucket Server.

3.9.1.3.1. Setting up an application link on the Bitbucket Server

Set up an application link for OAuth 1.0 on the Bitbucket Server.

Prerequisites

- You are logged in to the Bitbucket Server.
- **openssl** is installed in the operating system you are using.
- **base64** is installed in the operating system you are using.

Procedure

1. On a command line, run the commands to create the necessary files for the next steps and for use when applying the application link Secret:

```
$ openssl genrsa -out private.pem 2048 && \
openssl pkcs8 -topk8 -inform pem -outform pem -nocrypt -in private.pem -out
privatepkcs8.pem && \
cat privatepkcs8.pem | sed 's/-----BEGIN PRIVATE KEY-----//g' | sed 's/-----END PRIVATE
KEY-----//g' | tr -d '\n' | base64 | tr -d '\n' > privatepkcs8-stripped.pem && \
openssl rsa -in private.pem -pubout > public.pub && \
cat public.pub | sed 's/-----BEGIN PUBLIC KEY-----//g' | sed 's/-----END PUBLIC KEY-----//g'
| tr -d '\n' > public-stripped.pub && \
openssl rand -base64 24 > bitbucket-consumer-key && \
openssl rand -base64 24 > bitbucket-shared-secret
```

2. Go to **Administration → Application Links**.
3. Enter **https://devspaces-*<openshift_deployment_name>*.*<domain_name>*/** into the URL field and click **Create new link**.
4. Under **The supplied Application URL has redirected once**, check the **Use this URL** checkbox and click **Continue**.
5. Enter **OpenShift Dev Spaces** as the **Application Name**.
6. Select **Generic Application** as the **Application Type**.
7. Enter **OpenShift Dev Spaces** as the **Service Provider Name**.
8. Paste the content of the **bitbucket-consumer-key** file as the **Consumer key**.
9. Paste the content of the **bitbucket-shared-secret** file as the **Shared secret**.
10. Enter ***<bitbucket_server_url>*/plugins/servlet/oauth/request-token** as the **Request Token URL**.

11. Enter ***<bitbucket_server_url>*/plugins/servlet/oauth/access-token** as the **Access token URL**.
12. Enter ***<bitbucket_server_url>*/plugins/servlet/oauth/authorize** as the **Authorize URL**.
13. Check the **Create incoming link** checkbox and click **Continue**.
14. Paste the content of the **bitbucket-consumer-key** file as the **Consumer Key**.
15. Enter **OpenShift Dev Spaces** as the **Consumer name**.
16. Paste the content of the **public-stripped.pub** file as the **Public Key** and click **Continue**.

Additional resources

- Atlassian Documentation: Link to other applications

3.9.1.3.2. Applying an application link Secret for the Bitbucket Server

Prepare and apply the application link Secret for the Bitbucket Server.

Prerequisites

- The application link is set up on the Bitbucket Server.
- The following files, which were created when setting up the application link, are prepared:
  - **privatepkcs8-stripped.pem**
  - **bitbucket-consumer-key**
  - **bitbucket-shared-secret**
- An active **oc** session with administrative permissions to the destination OpenShift cluster. See Getting started with the CLI.

Procedure

1. Encode the content of the **bitbucket-consumer-key** file to Base64:

```
$ echo -n '<bitbucket-consumer-key file content>' | base64
```

2. Encode the content of the **bitbucket-shared-secret** file to Base64:

```
$ echo -n '<bitbucket-shared-secret file content>' | base64
```

3. Prepare the Secret:

```
kind: Secret
apiVersion: v1
metadata:
  name: bitbucket-oauth-config
  namespace: openshift-devspaces 1
  labels:
    app.kubernetes.io/component: oauth-scm-configuration
    app.kubernetes.io/part-of: che.eclipse.org
```

```
  annotations:
    che.eclipse.org/oauth-scm-server: bitbucket
    che.eclipse.org/scm-server-endpoint: <bitbucket_server_url> 2
type: Opaque
data:
  private.key: <Base64_content_of_privatepkcs8-stripped.pem> 3
  consumer.key: <Base64_content_of_bitbucket-consumer-key> 4
  shared_secret: <Base64_content_of_bitbucket-shared-secret> 5
```

1. The OpenShift Dev Spaces namespace. The default is **openshift-devspaces**.
2. The URL of the Bitbucket Server.
3. The content of the **privatepkcs8-stripped.pem** file, which was Base64-encoded when the file was generated.
4. The content of the **bitbucket-consumer-key** file that you manually encoded to Base64 in step 1.
5. The content of the **bitbucket-shared-secret** file that you manually encoded to Base64 in step 2.

4. Apply the Secret:

```
$ oc apply -f - <<EOF
<Secret_prepared_in_the_previous_step>
EOF
```

5. Verify in the output that the Secret is created.

3.9.1.4. Configuring OAuth 2.0 for the Bitbucket Cloud

You can enable users to work with a remote Git repository that is hosted in the Bitbucket Cloud:

1. Set up an OAuth consumer (OAuth 2.0) in the Bitbucket Cloud.
2. Apply an OAuth consumer Secret for the Bitbucket Cloud.

3.9.1.4.1. Setting up an OAuth consumer in the Bitbucket Cloud

Set up an OAuth consumer for OAuth 2.0 in the Bitbucket Cloud.

Prerequisites

- You are logged in to the Bitbucket Cloud.
- **base64** is installed in the operating system you are using.

Procedure

1. Click your avatar and go to the **All workspaces** page.
2. Select a workspace and click it.

3. Go to **Settings → OAuth consumers → Add consumer**.
4. Enter **OpenShift Dev Spaces** as the **Name**.
5. Enter **https://devspaces-*<openshift_deployment_name>*.*<domain_name>*/api/oauth/callback** as the **Callback URL**.
6. Under **Permissions**, check all of the **Account** and **Repositories** checkboxes, and click **Save**.
7. Expand the added consumer and then copy the **Key** value and encode it to Base64 for use when applying the Bitbucket OAuth consumer Secret:

```
$ echo -n '<bitbucket_oauth_consumer_key>' | base64
```

8. Copy the **Secret** value and encode it to Base64 for use when applying the Bitbucket OAuth consumer Secret:

```
$ echo -n '<bitbucket_oauth_consumer_secret>' | base64
```

Additional resources

- Bitbucket Docs: Use OAuth on Bitbucket Cloud

3.9.1.4.2. Applying an OAuth consumer Secret for the Bitbucket Cloud

Prepare and apply an OAuth consumer Secret for the Bitbucket Cloud.

Prerequisites

- The OAuth consumer is set up in the Bitbucket Cloud.
- The Base64-encoded values, which were generated when setting up the Bitbucket OAuth consumer, are prepared:
  - Bitbucket OAuth consumer Key
  - Bitbucket OAuth consumer Secret
- An active **oc** session with administrative permissions to the destination OpenShift cluster. See Getting started with the CLI.

Procedure

1. Prepare the Secret:

```
kind: Secret
apiVersion: v1
metadata:
  name: bitbucket-oauth-config
  namespace: openshift-devspaces 1
  labels:
    app.kubernetes.io/part-of: che.eclipse.org
    app.kubernetes.io/component: oauth-scm-configuration
  annotations:
```

```
    che.eclipse.org/oauth-scm-server: bitbucket
type: Opaque
data:
  id: <Base64_Bitbucket_Oauth_Consumer_Key> 2
  secret: <Base64_Bitbucket_Oauth_Consumer_Secret> 3
```

1. The OpenShift Dev Spaces namespace. The default is **openshift-devspaces**.
2. The Base64-encoded **Bitbucket OAuth consumer Key**.
3. The Base64-encoded **Bitbucket OAuth consumer Secret**.

2. Apply the Secret:

```
$ oc apply -f - <<EOF
<Secret_prepared_in_the_previous_step>
EOF
```

3. Verify in the output that the Secret is created.

3.9.1.5. Configuring OAuth 2.0 for Microsoft Azure DevOps Services

To enable users to work with a remote Git repository that is hosted on Microsoft Azure Repos:

1. Set up the Microsoft Azure DevOps Services OAuth App (OAuth 2.0).
2. Apply the Microsoft Azure DevOps Services OAuth App Secret.

3.9.1.5.1. Setting up the Microsoft Azure DevOps Services OAuth App

Set up a Microsoft Azure DevOps Services OAuth App using OAuth 2.0.

Prerequisites

- You are logged in to Microsoft Azure DevOps Services.
- Third-party applications can access resources in your organization through OAuth. See Manage security & app access policies.
- **base64** is installed in the operating system you are using.

Procedure

1. Visit https://app.vsaex.visualstudio.com/app/register/.
2. Enter the following values:
   a. **Company name**: **OpenShift Dev Spaces**
   b. **Application name**: **OpenShift Dev Spaces**
   c. **Application website**: **https://devspaces-*<openshift_deployment_name>*.*<domain_name>*/**

d. **Authorization callback URL**:
https://devspaces-*<openshift_deployment_name>*.*<domain_name>*/api/oauth/callback

3. In **Select Authorized scopes**, select **Code (read and write)**.

4. Click **Create application**.

5. Copy the **App ID** and encode it to Base64 for use when applying the Microsoft Azure DevOps Services OAuth App Secret:

```
$ echo -n '<microsoft_azure_devops_services_oauth_app_id>' | base64
```

6. Click **Show** to display the **Client Secret**.

7. Copy the **Client Secret** and encode it to Base64 for use when applying the Microsoft Azure DevOps Services OAuth App Secret:

```
$ echo -n '<microsoft_azure_devops_services_oauth_client_secret>' | base64
```

Additional resources

- Authorize access to REST APIs with OAuth 2.0

3.9.1.5.2. Applying the Microsoft Azure DevOps Services OAuth App Secret

Prepare and apply the Microsoft Azure DevOps Services Secret.

Prerequisites

- Setting up the Microsoft Azure DevOps Services OAuth App is completed.
- The Base64-encoded values, which were generated when setting up the Microsoft Azure DevOps Services OAuth App, are prepared:
  - **App ID**
  - **Client Secret**
- An active **oc** session with administrative permissions to the destination OpenShift cluster. See Getting started with the CLI.

Procedure

1. Prepare the Secret:

```
kind: Secret
apiVersion: v1
metadata:
  name: azure-devops-oauth-config
  namespace: openshift-devspaces (1)
  labels:
    app.kubernetes.io/part-of: che.eclipse.org
    app.kubernetes.io/component: oauth-scm-configuration
  annotations:
    che.eclipse.org/oauth-scm-server: azure-devops
```

```
type: Opaque
data:
  id: <Base64_Microsoft_Azure_DevOps_Services_OAuth_App_ID> (2)
  secret: <Base64_Microsoft_Azure_DevOps_Services_OAuth_Client_Secret> (3)
```

(1) The OpenShift Dev Spaces namespace. The default is **openshift-devspaces**.

(2) The Base64-encoded Microsoft Azure DevOps Services OAuth **App ID**.

(3) The Base64-encoded Microsoft Azure DevOps Services OAuth **Client Secret**.

2. Apply the Secret:

```
$ oc apply -f - <<EOF
<Secret_prepared_in_the_previous_step>
EOF
```

3. Verify in the output that the Secret is created.

3.9.2. Configuring the administrative user

To execute actions that require administrative privileges on OpenShift Dev Spaces server, such as deleting user data, activate a user with administrative privileges. The default installation enables the administrative privileges for the **admin** user, regardless of its existence on OpenShift.

Procedure

- Configure the **CheCluster** Custom Resource to set the *<admin>* user with administrative privileges. See Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource" .

```
spec:
  components:
    cheServer:
      extraProperties:
        CHE_SYSTEM_ADMIN__NAME: '<admin>'
```

Additional resources

- Section 3.1.1, "Using dsc to configure the **CheCluster** Custom Resource during installation"
- Section 3.1.2, "Using the CLI to configure the CheCluster Custom Resource"

3.9.3. Removing user data in compliance with the GDPR

You can remove a user's data in compliance with the General Data Protection Regulation (GDPR) that enforces the right of individuals to have their personal data erased.

WARNING

Removing user data as follows is irreversible! All removed data is deleted and unrecoverable!

Prerequisites

- An active session with administrative permissions for OpenShift Dev Spaces. See Section 3.9.2, "Configuring the administrative user".
- An active **oc** session with administrative permissions for the OpenShift cluster. See Getting started with the OpenShift CLI.

Procedure

1. If the OpenShift Dev Spaces instance has been upgraded since a previous release in which user data were stored in a PostgreSQL database, then first use the deprecated endpoints:
   a. Go to **https://devspaces-*<openshift_deployment_name>*.*<domain_name>*/swagger/#/user/find_1**.
   b. Select **Try it out → name**: *<username>* → **Execute** to get the user **id**.
   c. Find the **id** value in the **Response body**.
   If the response is **404**, which means the user is not in the database, skip to step two.
   d. Go to **https://devspaces-*<openshift_deployment_name>*.*<domain_name>*/swagger/#/user/remove**.
   e. Select **Try it out → id**: *<id>* → **Execute** to remove the user's data that is managed by the OpenShift Dev Spaces server.
   f. Verify that you got a **204** response.
2. Delete the user project to remove all OpenShift resources bound to the user, such as workspaces, Secrets, and ConfigMaps.

```
$ oc delete namespace <username>-devspaces
```

Additional resources

- Chapter 5, *Using the Dev Spaces server API*
- Section 3.2.1, "Configuring project name"
- Chapter 7, *Uninstalling Dev Spaces*

CHAPTER 4. MANAGING IDE EXTENSIONS

IDEs use extensions or plugins to extend their functionality, and the mechanism for managing extensions differs between IDEs.

- Section 4.1, “Extensions for Microsoft Visual Studio Code - Open Source”

4.1. EXTENSIONS FOR MICROSOFT VISUAL STUDIO CODE - OPEN SOURCE

To manage extensions, this IDE uses one of these Open VSX registry instances:

- The public, primary open-vsx.org registry.
- The embedded instance of the Open VSX registry that runs in the **plugin-registry** pod of OpenShift Dev Spaces to support air-gapped, offline, and proxy-restricted environments. The embedded Open VSX registry contains only a subset of the extensions published on open-vsx.org. This subset can be customized.
- A standalone Open VSX registry instance, deployed on a network accessible from OpenShift Dev Spaces workspace pods.

4.1.1. Selecting an Open VSX registry instance

The Open VSX registry at **https://open-vsx.org** is the default if resolved from within your organization’s cluster. If not, then the embedded Open VSX registry within the OpenShift Dev Spaces **plugin-registry** pod is the default.

If the default Open VSX registry instance is not what you need, you can select another Open VSX registry instance as follows.

Procedure

- Edit the **openVSXURL** value in the **CheCluster** custom resource:

```
spec:
  components:
    pluginRegistry:
      openVSXURL: "<url_of_an_open_vsx_registry_instance>"
```

TIP

  - The default **openVSXURL** value is **https://open-vsx.org**.
  - To select the embedded Open VSX registry instance in the **plugin-registry** pod, use **openVSXURL: ''**. See the next section for how to customize the list of included extensions.
  - You can also point **openVSXURL** at the URL of a standalone Open VSX registry instance if its URL is accessible from within your organization’s cluster and not blocked by a proxy.

4.1.2. Adding or removing extensions in the embedded Open VSX registry instance

You can add or remove extensions in the embedded Open VSX registry instance deployed by OpenShift Dev Spaces to support offline and proxied environments.

This will create a custom build of the Open VSX registry, which can be used in your organization's workspaces.

TIP

To get the latest security fixes after a OpenShift Dev Spaces update, rebuild your container based on the latest tag or SHA.

Procedure

1. Get the publisher and extension names of each chosen extension:
   a. Find the extension on the Open VSX registry website and copy the URL of the extension's listing page.
   b. Extract the *<publisher>* and *<extension>* names from the copied URL:

      ```
      https://www.open-vsx.org/extension/<publisher>/<extension>
      ```

      #### TIP

      If the extension is only available from Microsoft Visual Studio Marketplace, but not Open VSX, you can ask the extension publisher to also publish it on open-vsx.org according to these instructions, potentially using this GitHub action.

      If the extension publisher is unavailable or unwilling to publish the extension to open-vsx.org, and if there is no Open VSX equivalent of the extension, consider reporting an issue to the Open VSX team.

2. Download or fork and clone the plugin registry repository.
3. For each extension that you need to add or remove, edit the **openvsx-sync.json** file:
   - To add extensions, add the publisher and extension names to the **openvsx-sync.json** file.
   - To remove extensions, remove the publisher and extension names from the **openvsx-sync.json** file.
   - Use the following JSON syntax:

     ```
     {
       "id": "<publisher>.<extension>"
     }
     ```

TIP

- The latest extension version on open-vsx.org is the default. Alternatively, you can add **"version": "*<extension_version>*"** on a new line to specify a version.
- If you have a closed-source extension or an extension developed only for internal use in your organization, you can add the extension directly from a **.vsix** file by using a URL accessible to your custom plugin registry container:

```
{
    "id": "<publisher>.<extension>",
    "download": "<url_to_download_vsix_file>",
    "version": "<extension_version>"
}
```

- Read the Terms of Use for the Microsoft Visual Studio Marketplace before using its resources.

4. Build the plugin registry container image and publish it to a container registry like quay.io:

a.
```
$ ./build.sh -o <username> -r quay.io -t custom
```

b.
```
$ podman push quay.io/<username/plugin_registry:custom>
```

5. Edit the **CheCluster** custom resource in your organization's cluster to point to the image (for example, on quay.io) and save the changes:

```
spec:
  components:
    pluginRegistry:
      deployment:
        containers:
          - image: quay.io/<username/plugin_registry:custom>
      openVSXURL: ''
```

Verification

1. Check that the **plugin-registry** pod has restarted and is running.
2. Restart the workspace and check the available extensions in the **Extensions** view of the workspace IDE.

CHAPTER 5. USING THE DEV SPACES SERVER API

To manage OpenShift Dev Spaces server workloads, use the Swagger web user interface to navigate OpenShift Dev Spaces server API.

Procedure

- Navigate to the Swagger API web user interface: **https://devspaces-*<openshift_deployment_name>*.*<domain_name>*/swagger**.

Additional resources

- Swagger

CHAPTER 6. UPGRADING DEV SPACES

This chapter describes how to upgrade from CodeReady Workspaces 3.1 to OpenShift Dev Spaces 3.5.

6.1. UPGRADING THE CHECTL MANAGEMENT TOOL

This section describes how to upgrade the **dsc** management tool.

Procedure

- Section 2.1, “Installing the dsc management tool” .

6.2. SPECIFYING THE UPDATE APPROVAL STRATEGY

The Red Hat OpenShift Dev Spaces Operator supports two upgrade strategies:

Automatic

The Operator installs new updates when they become available.

Manual

New updates need to be manually approved before installation begins.

You can specify the update approval strategy for the Red Hat OpenShift Dev Spaces Operator by using the OpenShift web console.

Prerequisites

- An OpenShift web console session by a cluster administrator. See Accessing the web console.
- An instance of OpenShift Dev Spaces that was installed by using Red Hat Ecosystem Catalog.

Procedure

1. In the OpenShift web console, navigate to **Operators → Installed Operators**.
2. Click **Red Hat OpenShift Dev Spaces**in the list of installed Operators.
3. Navigate to the **Subscription** tab.
4. Configure the **Update approval** strategy to **Automatic** or **Manual**.

Additional resources

- Changing the update channel for an Operator

6.3. UPGRADING DEV SPACES USING THE OPENSHIFT WEB CONSOLE

You can manually approve an upgrade from an earlier minor version using the Red Hat OpenShift Dev Spaces Operator from the Red Hat Ecosystem Catalog in the OpenShift web console.

Prerequisites

- An OpenShift web console session by a cluster administrator. See Accessing the web console.
- An instance of OpenShift Dev Spaces that was installed by using the Red Hat Ecosystem Catalog.
- The approval strategy in the subscription is **Manual**. See Section 6.2, "Specifying the update approval strategy".

Procedure

- Manually approve the pending Red Hat OpenShift Dev Spaces Operator upgrade. See Manually approving a pending Operator upgrade.

Verification steps

1. Navigate to the OpenShift Dev Spaces instance.
2. The 3.5 version number is visible at the bottom of the page.

Additional resources

- Manually approving a pending Operator upgrade

6.4. UPGRADING DEV SPACES USING THE CLI MANAGEMENT TOOL

This section describes how to upgrade from the previous minor version using the CLI management tool.

Prerequisites

- An administrative account on OpenShift.
- A running instance of a previous minor version of CodeReady Workspaces, installed using the CLI management tool on the same instance of OpenShift, in the **openshift-devspaces** OpenShift project.
- **dsc** for OpenShift Dev Spaces version 3.5. See: Section 2.1, "Installing the dsc management tool".

Procedure

1. Save and push changes back to the Git repositories for all running CodeReady Workspaces 3.1 workspaces.
2. Shut down all workspaces in the CodeReady Workspaces 3.1 instance.
3. Upgrade OpenShift Dev Spaces:

```
$ dsc server:update -n openshift-devspaces
```

NOTE

For slow systems or internet connections, add the **--k8spodwaittimeout=1800000** flag option to extend the Pod timeout period to 1800000 ms or longer.

Verification steps

1. Navigate to the OpenShift Dev Spaces instance.
2. The 3.5 version number is visible at the bottom of the page.

6.5. UPGRADING DEV SPACES IN A RESTRICTED ENVIRONMENT

This section describes how to upgrade Red Hat OpenShift Dev Spaces and perform minor version updates by using the CLI management tool in a restricted environment.

Prerequisites

- The OpenShift Dev Spaces instance was installed on OpenShift using the **dsc --installer operator** method in the **openshift-devspaces** project. See Section 2.4, "Installing Dev Spaces in a restricted environment".
- The OpenShift cluster has at least 64 GB of disk space.
- The OpenShift cluster is ready to operate on a restricted network, and the OpenShift control plane has access to the public internet. See About disconnected installation mirroring and Using Operator Lifecycle Manager on restricted networks.
- An active **oc** session with administrative permissions to the OpenShift cluster. See Getting started with the OpenShift CLI.
- An active **oc registry** session to the **registry.redhat.io** Red Hat Ecosystem Catalog. See: Red Hat Container Registry authentication.
- **opm**. See Installing the **opm** CLI.
- **jq**. See Downloading **jq**.
- **podman**. See Installing Podman.
- An active **skopeo** session with administrative access to the *<my_registry>* registry. See Installing Skopeo, Authenticating to a registry , and Mirroring images for a disconnected installation .
- **dsc** for OpenShift Dev Spaces version 3.5. See Section 2.1, "Installing the dsc management tool".

Procedure

1. Download and execute the mirroring script to install a custom Operator catalog and mirror the related images: prepare-restricted-environment.sh.

```
$ bash prepare-restricted-environment.sh \
  --ocp_ver "4.12" \
  --devworkspace_operator_index "registry.redhat.io/redhat/redhat-operator-index:v4.12" \
  --devworkspace_operator_version "v0.19.0" \
  --prod_operator_index "registry.redhat.io/redhat/redhat-operator-index:v4.12" \
  --prod_operator_package_name "devspaces-operator" \
  --prod_operator_version "v3.5.0" \
  --my_registry "<my_registry>" \
  --my_catalog "<my_catalog>"
```

2. In all running workspaces in the CodeReady Workspaces 3.1 instance, save and push changes back to the Git repositories.

3. Stop all workspaces in the CodeReady Workspaces 3.1 instance.

4. Run the following command:

```
$ dsc server:update --che-operator-image="$TAG" -n openshift-devspaces --
k8spodwaittimeout=1800000
```

Verification steps

1. Navigate to the OpenShift Dev Spaces instance.

2. The 3.5 version number is visible at the bottom of the page.

Additional resources

- Red Hat-provided Operator catalogs
- Managing custom catalogs

6.6. REPAIRING THE DEV WORKSPACE OPERATOR ON OPENSHIFT

Under certain conditions, such as OLM restart or cluster upgrade, the Dev Spaces Operator for OpenShift Dev Spaces might automatically install the Dev Workspace Operator even when it is already present on the cluster. In that case, you can repair the Dev Workspace Operator on OpenShift as follows:

Prerequisites

- An active **oc** session as a cluster administrator to the destination OpenShift cluster. See Getting started with the CLI.
- On the **Installed Operators** page of the OpenShift web console, you see multiple entries for the Dev Workspace Operator or one entry that is stuck in a loop of **Replacing** and **Pending**.

Procedure

1. Delete the **devworkspace-controller** namespace that contains the failing pod.

2. Update **DevWorkspace** and **DevWorkspaceTemplate** Custom Resource Definitions (CRD) by setting the conversion strategy to **None** and removing the entire **webhook** section:

```
spec:
  ...
  conversion:
    strategy: None
status:
...
```

TIP

You can find and edit the **DevWorkspace** and **DevWorkspaceTemplate** CRDs in the **Administrator** perspective of the OpenShift web console by searching for **DevWorkspace** in **Administration → CustomResourceDefinitions**.

NOTE

The **DevWorkspaceOperatorConfig** and **DevWorkspaceRouting** CRDs have the conversion strategy set to **None** by default.

3. Remove the Dev Workspace Operator subscription:

```
$ oc delete sub devworkspace-operator \
-n openshift-operators 1
```

1 **openshift-operators** or an OpenShift project where the Dev Workspace Operator is installed.

4. Get the Dev Workspace Operator CSVs in the *<devworkspace_operator.vX.Y.Z>* format:

```
$ oc get csv | grep devworkspace
```

5. Remove each Dev Workspace Operator CSV:

```
$ oc delete csv <devworkspace_operator.vX.Y.Z> \
-n openshift-operators 1
```

1 **openshift-operators** or an OpenShift project where the Dev Workspace Operator is installed.

6. Re-create the Dev Workspace Operator subscription:

```
$ cat <<EOF | oc apply -f -
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: devworkspace-operator
  namespace: openshift-operators
spec:
  channel: fast
  name: devworkspace-operator
  source: redhat-operators
  sourceNamespace: openshift-marketplace
  installPlanApproval: Automatic 1
  startingCSV: devworkspace-operator.v0.19.0
EOF
```

1 **Automatic** or **Manual**.

IMPORTANT

For **installPlanApproval: Manual**, in the **Administrator** perspective of the OpenShift web console, go to **Operators → Installed Operators** and select the following for the **Dev Workspace Operator**: **Upgrade available → Preview InstallPlan → Approve**.

7. In the **Administrator** perspective of the OpenShift web console, go to **Operators → Installed Operators** and verify the **Succeeded** status of the **Dev Workspace Operator**.

CHAPTER 7. UNINSTALLING DEV SPACES

WARNING

Uninstalling OpenShift Dev Spaces removes all OpenShift Dev Spaces-related user data!

Use **oc** to uninstall the OpenShift Dev Spaces instance.

Prerequisites

- **dsc**. See: Section 2.1, “Installing the dsc management tool” .

Procedure

- Remove the OpenShift Dev Spaces instance:

```
$ dsc server:delete
```

TIP

The **--delete-namespace** option removes the OpenShift Dev Spaces namespace.

The **--delete-all** option removes the Dev Workspace Operator and the related resources.