

The Containers and Cloud-Native Roadshow Dev Track

 guides-m2-labs-infra.apps.cluster-beijing-5bc2.beijing-

Your Workshop Environment

The Workshop Environment You Are Using

Your workshop environment consists of several components which have been pre-installed and are ready to use. Depending on which parts of the workshop you're doing, you will use one or more of:

- [Red Hat OpenShift](#) - You'll use one or more *projects* (Kubernetes namespaces) that are your own and are isolated from other workshop students
- [Red Hat CodeReady Workspaces](#) - based on **Eclipse Che**, it's a cloud-based, in-browser IDE (similar to IntelliJ IDEA, VSCode, Eclipse IDE). You've been provisioned your own personal workspace for use with this workshop. You'll write, test, and deploy code from here.
- [Red Hat Application Migration Toolkit](#) - You'll use this to migrate an existing application
- [Red Hat Runtimes](#) - a collection of cloud-native runtimes like Spring Boot, Node.js, and [Quarkus](#)
- [Red Hat AMQ Streams](#) - streaming data platform based on **Apache Kafka**
- [Red Hat SSO](#) - For authentication / authorization - based on **Keycloak**
- Other open source projects like [Gogs](#) (Git server that holds application source code), [Knative](#) (for serverless apps), [Jenkins](#) and [Tekton](#) (CI/CD pipelines), [Prometheus](#) and [Grafana](#) (monitoring apps), and more.

You'll be provided clickable URLs throughout the workshop to access the services that have been installed for you.

How to complete this workshop

Simply follow these instructions end-to-end. **You'll need to do quite a bit of copy/paste for Linux commands and source code modifications**, as well as clicking around on various consoles used in the labs. When you get to the end of each section, you can click the "Next >" button at the bottom to advance to the next topic. You can also use the menu on the left to move around the instructions at will.

The entire workshop is split into one or more *modules* - Look at the top of the screen in the header to see which module you are on. After you complete this module, your instructor may have additional modules to complete.

Good luck, and let's get started!

Advanced Cloud-Native Services

Advanced Cloud-Native Services

If you complete the **Cloud Native Workshop - Module 1**, you learned how to take an existing application to the cloud with JBoss EAP and OpenShift, and you got a glimpse into the power of OpenShift for existing applications.

In this lab, you will go deeper into how to use the OpenShift Container Platform as a developer to build and deploy applications. We'll focus on the core features of OpenShift as it relates to developers, and you'll learn typical workflows for a developer (develop, build, test, deploy, and repeat).

Let's get started

If you are not familiar with the OpenShift Container Platform, it's worth taking a few minutes to understand the basics of the platform as well as the environment that you will be using for this workshop.

The goal of OpenShift is to provide a great experience for both Developers and System Administrators to develop, deploy, and run containerized applications. Developers should love using OpenShift because it enables them to take advantage of both containerized applications and orchestration without having to know the details. Developers are free to focus on their code instead of spending time writing Dockerfiles and running docker builds.

Both Developers and Operators communicate with the OpenShift Platform via one of the following methods:

- **Command Line Interface** - The command line tool that we will be using as part of this training is called the `oc` tool. You used this briefly in the last lab. This tool is written in the Go programming language and is a single executable that is provided for Windows, OS X, and the Linux Operating Systems.
- **Web Console** - OpenShift also provides a feature rich Web Console that provides a friendly graphical interface for interacting with the platform. You can always access the Web Console using the link provided just above the Terminal window on the right:
- **REST API** - Both the command line tool and the web console actually communicate to OpenShift via the same method, the REST API. Having a robust API allows users to create their own scripts and automation depending on their specific requirements. For detailed information about the REST API, check out the [official documentation](#). You will not use the REST API directly in this workshop.

During this workshop, you will be using both the command line tool and the web console. However, it should be noted that there are plugins for several integrated

development environments as well. For example, to use OpenShift from the Eclipse IDE, you would want to use the official [JBoss Tools](#) plugin.

Now that you know how to interact with OpenShift, let's focus on some core concepts that you as a developer will need to understand as you are building your applications!

Developer Concepts

There are several concepts in OpenShift useful for developers, and in this workshop you should be familiar with them.

Projects

[Projects](#) are a top level concept to help you organize your deployments. An OpenShift project allows a community of users (or a user) to organize and manage their content in isolation from other communities. Each project has its own resources, policies (who can or cannot perform actions), and constraints (quotas and limits on resources, etc).

Projects act as a wrapper around all the application services and endpoints you (or your teams) are using for your work.

Containers

The basic units of OpenShift applications are called containers (sometimes called Linux Containers). [Linux container technologies](#) are lightweight mechanisms for isolating running processes so that they are limited to interacting with only their designated resources.

Though you do not directly interact with the Docker CLI or service when using OpenShift Container Platform, understanding their capabilities and terminology is important for understanding their role in OpenShift Container Platform and how your applications function inside of containers.

Pods

OpenShift Container Platform leverages the Kubernetes concept of a pod, which is one or more containers deployed together on one host, and the smallest compute unit that can be defined, deployed, and managed.

Pods are the rough equivalent of a machine instance (physical or virtual) to a container. Each pod is allocated its own internal IP address, therefore owning its entire port space, and containers within pods can share their local storage and networking.

Images

Containers in OpenShift are based on Docker-formatted container images. An image is a binary that includes all of the requirements for running a single container, as well as metadata describing its needs and capabilities.

You can think of it as a packaging technology. Containers only have access to resources defined in the image unless you give the container additional access when creating it. By deploying the same image in multiple containers across multiple hosts and load balancing between them, OpenShift Container Platform can provide redundancy and horizontal scaling for a service packaged into an image.

Image Streams

An image stream and its associated tags provide an abstraction for referencing Images from within OpenShift. The image stream and its tags allow you to see what images are available and ensure that you are using the specific image you need even if the image in the repository changes.

Builds

A build is the process of transforming input parameters into a resulting object. Most often, the process is used to transform input parameters or source code into a runnable image. A *BuildConfig* object is the definition of the entire build process. It can build from different sources, including a Dockerfile, a source code repository like Git, or a Jenkins Pipeline definition.

Pipelines

Pipelines allow developers to define a *Jenkins* pipeline for execution by the Jenkins pipeline plugin. The build can be started, monitored, and managed by OpenShift Container Platform in the same way as any other build type.

Pipeline workflows are defined in a *Jenkinsfile*, either embedded directly in the build configuration, or supplied in a Git repository and referenced by the build configuration.

Deployments

An OpenShift Deployment describes how images are deployed to pods, and how the pods are deployed to the underlying container runtime platform. OpenShift deployments also provide the ability to transition from an existing deployment of an image to a new one and also define hooks to be run before or after creating the replication controller.

Services

A Kubernetes service serves as an internal load balancer. It identifies a set of replicated pods in order to proxy the connections it receives to them. Backing pods can be added to or removed from a service arbitrarily while the service remains consistently available, enabling anything that depends on the service to refer to it at a consistent address.

Routes

Services provide internal abstraction and load balancing within an OpenShift environment, sometimes clients (users, systems, devices, etc.) **outside** of OpenShift need to access an application. The way that external clients are able to access applications

running in OpenShift is through the OpenShift routing layer. And the data object behind that is a *Route*.

The default OpenShift router (HAProxy) uses the HTTP header of the incoming request to determine where to proxy the connection. You can optionally define security, such as TLS, for the *Route*. If you want your *Services*, and, by extension, your *Pods*, to be accessible to the outside world, you need to create a *Route*.

Templates

Templates contain a collection of object definitions (BuildConfigs, DeploymentConfigs, Services, Routes, etc) that compose an entire working project. They are useful for packaging up an entire collection of runtime objects into a somewhat portable representation of a running application, including the configuration of the elements.

You will use several pre-defined templates to initialize different environments for the application. You've already used one in the previous lab to deploy the application into a *dev* environment, and you'll use more in this lab to provision the *production* environment as well.

Consult the [OpenShift documentation](#) for more details on these and other concepts.

Getting Ready for the labs

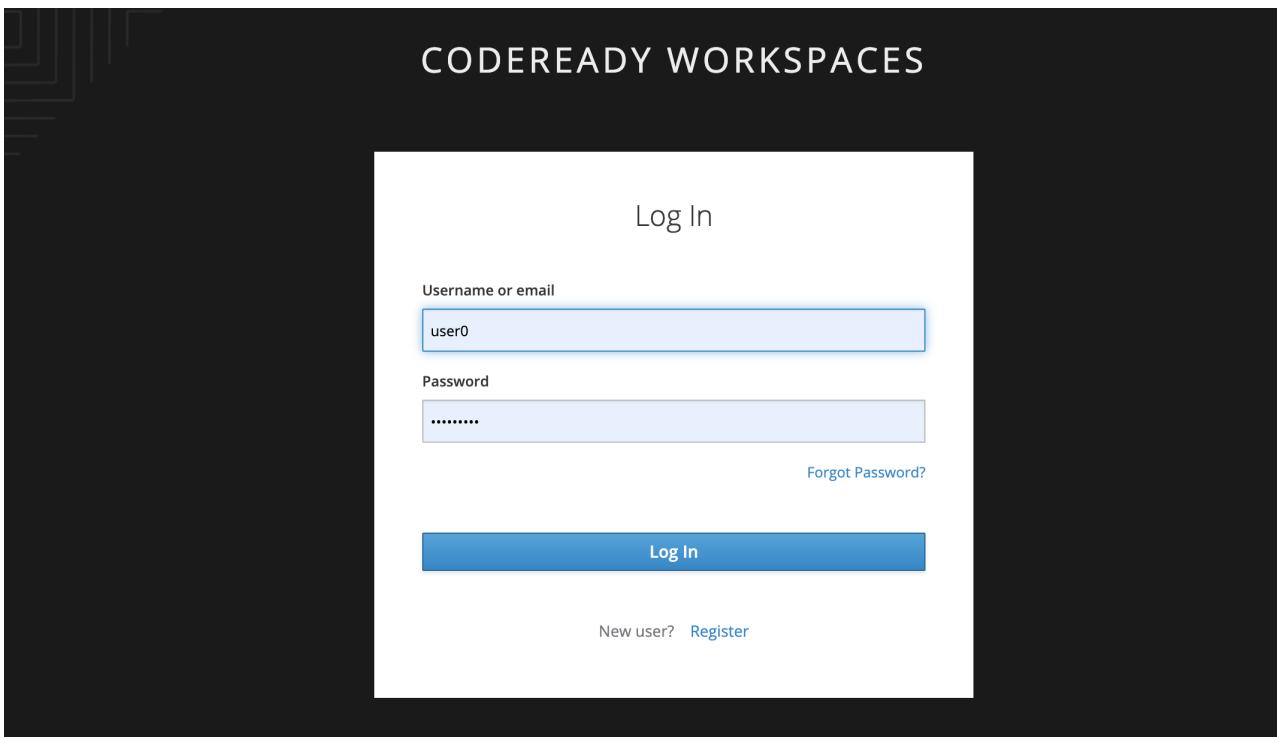
NOTE

If you've already completed the **Optimizing Existing Applications** module then you will simply need to import the code for this module. Skip down to the **Import Projects** section.

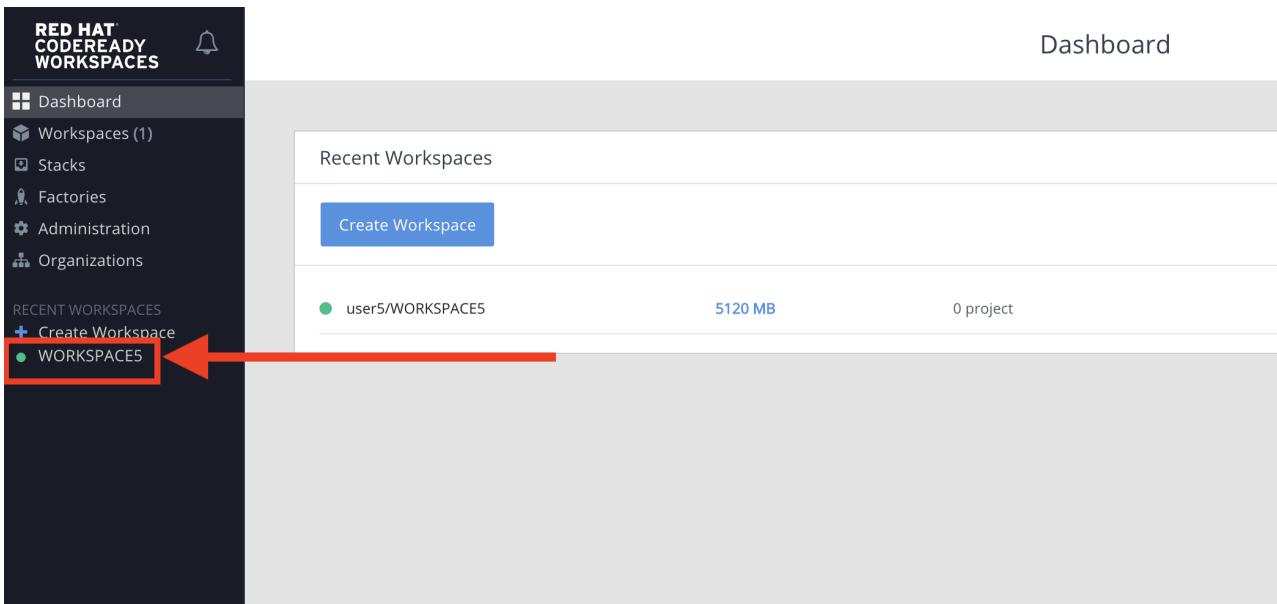
If this is the first module you are doing today

You will be using Red Hat CodeReady Workspaces, an online IDE based on [Eclipse Che](#). **Changes to files are auto-saved every few seconds**, so you don't need to explicitly save changes.

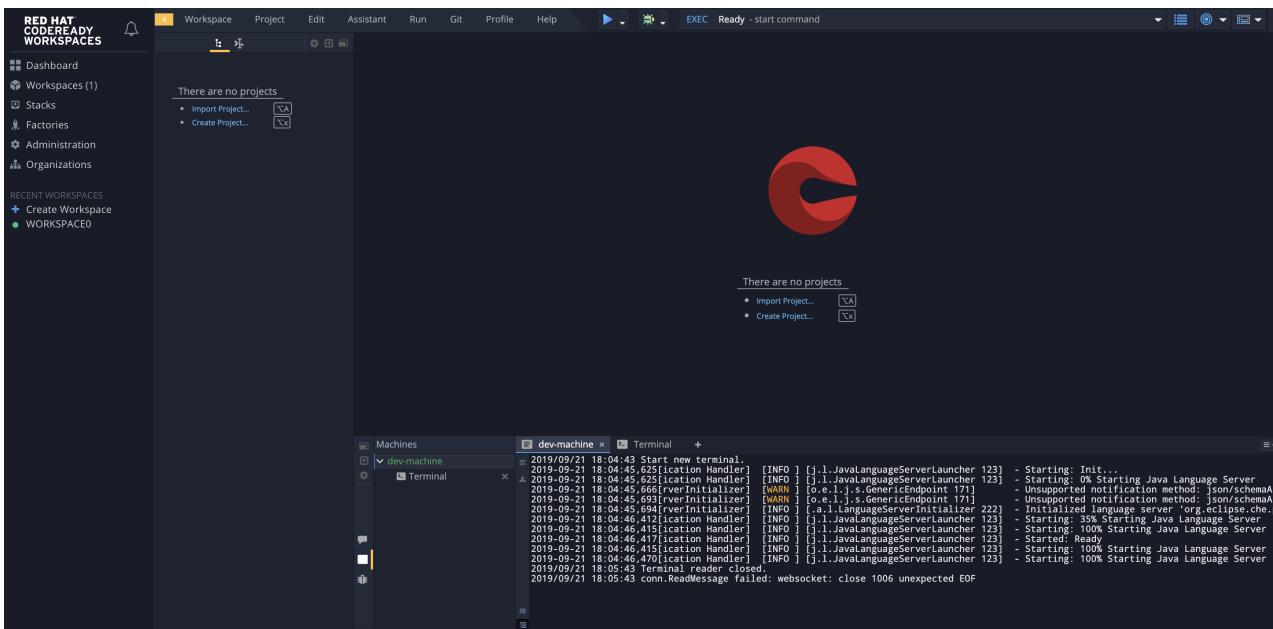
To get started, [access the Che instance](#) and log in using the username and password you've been assigned (e.g. `userXX/r3dh4t1!`):



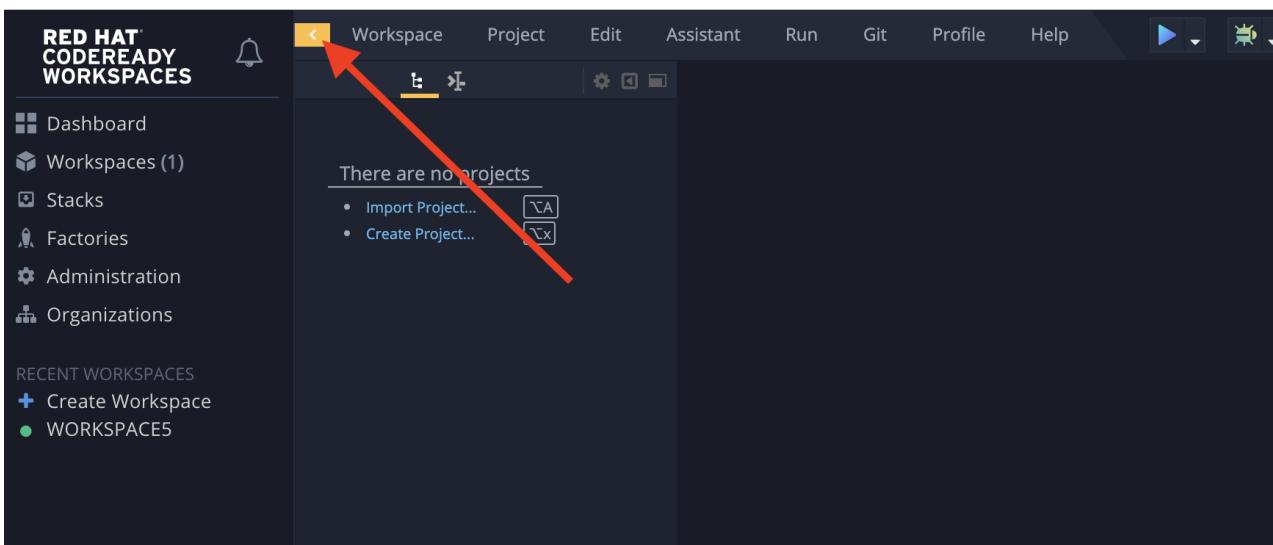
Once you log in, you'll be placed on your personal dashboard. We've pre-created workspaces for you to use. Click on the name of the pre-created workspace on the left, as shown below (the name will be different depending on your assigned number). You can also click on the name of the workspace in the center, and then click on the green button that says "OPEN" on the top right hand side of the screen:



After a minute or two, you'll be placed in the workspace:



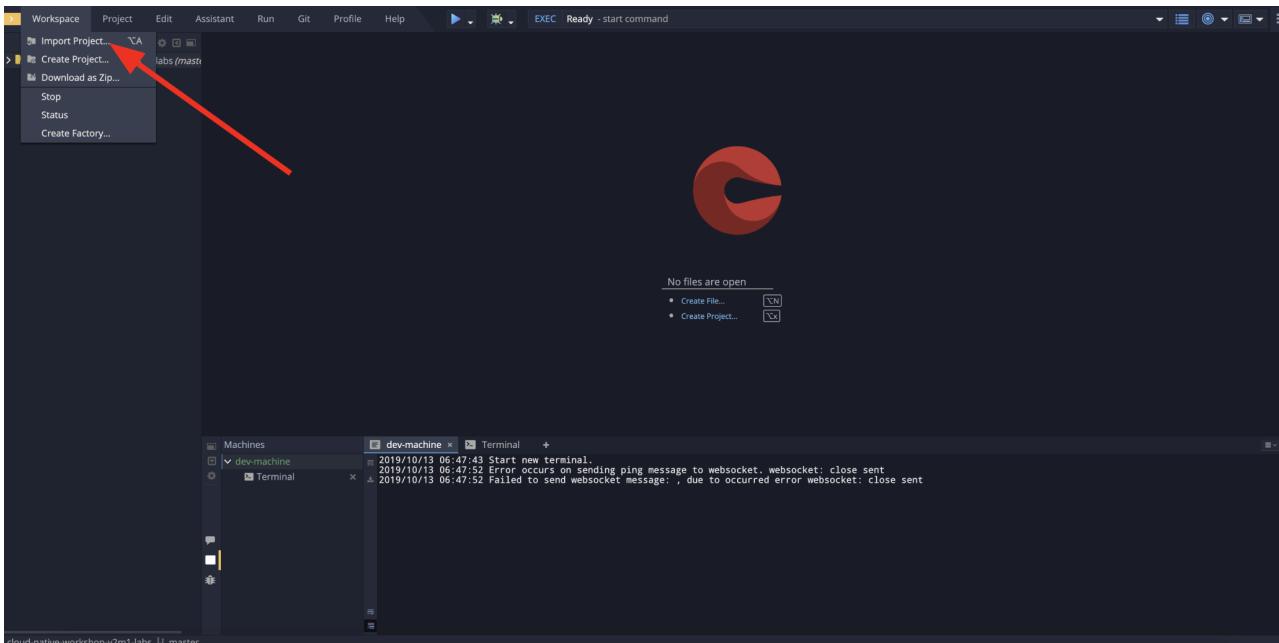
To gain extra screen space, click on the yellow arrow to hide the left menu (you won't need it):



Users of Eclipse, IntelliJ IDEA or Visual Studio Code will see a familiar layout: a project/file browser on the left, a code editor on the right, and a terminal at the bottom. You'll use all of these during the course of this workshop, so keep this browser tab open throughout. **If things get weird, you can simply reload the browser tab to refresh the view.**

Import Projects

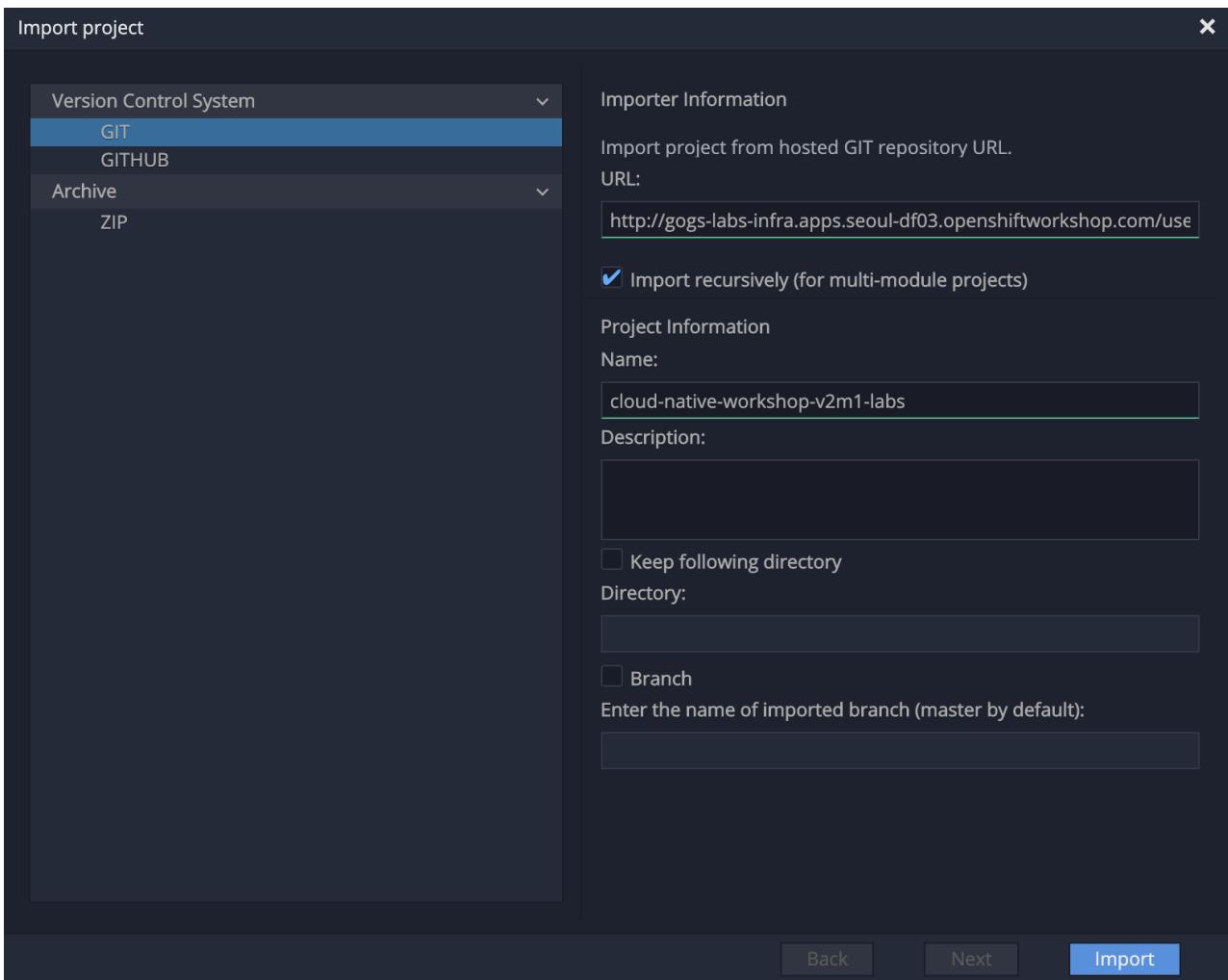
Click on the **Import Projects...** in **Workspace** menu and enter the following:



NOTE: If you've completed other modules already, then you can use *Workspace > Import Project* menu to import the project.

- Version Control System: [GIT](#)
- URL: <http://gogs-labs-infra.apps.cluster-beijing-5bc2.beijing-5bc2.open.redhat.com/userXX/cloud-native-workshop-v2m2-labs.git> (IMPORTANT: replace userXX with your lab user)
- Check [Import recursively \(for multi-module projects\)](#)
- Name: [cloud-native-workshop-v2m2-labs](#)

Tip: You can find GIT URL when you click on [GIT URL](#) then login with your credentials.

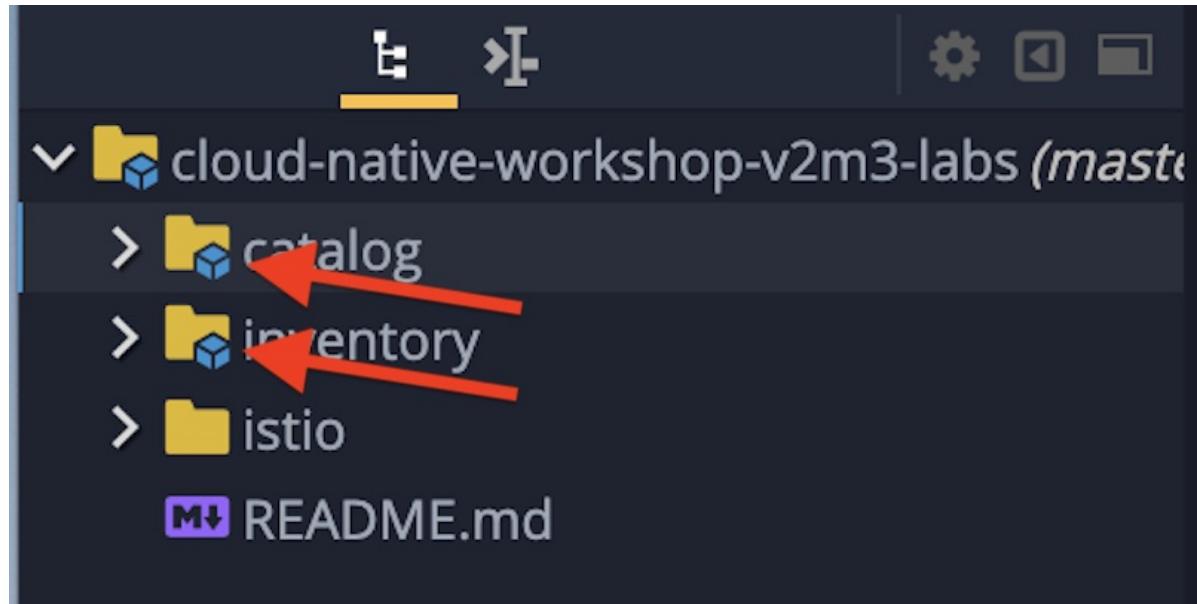


The projects are imported now into your workspace and is visible in the project explorer.

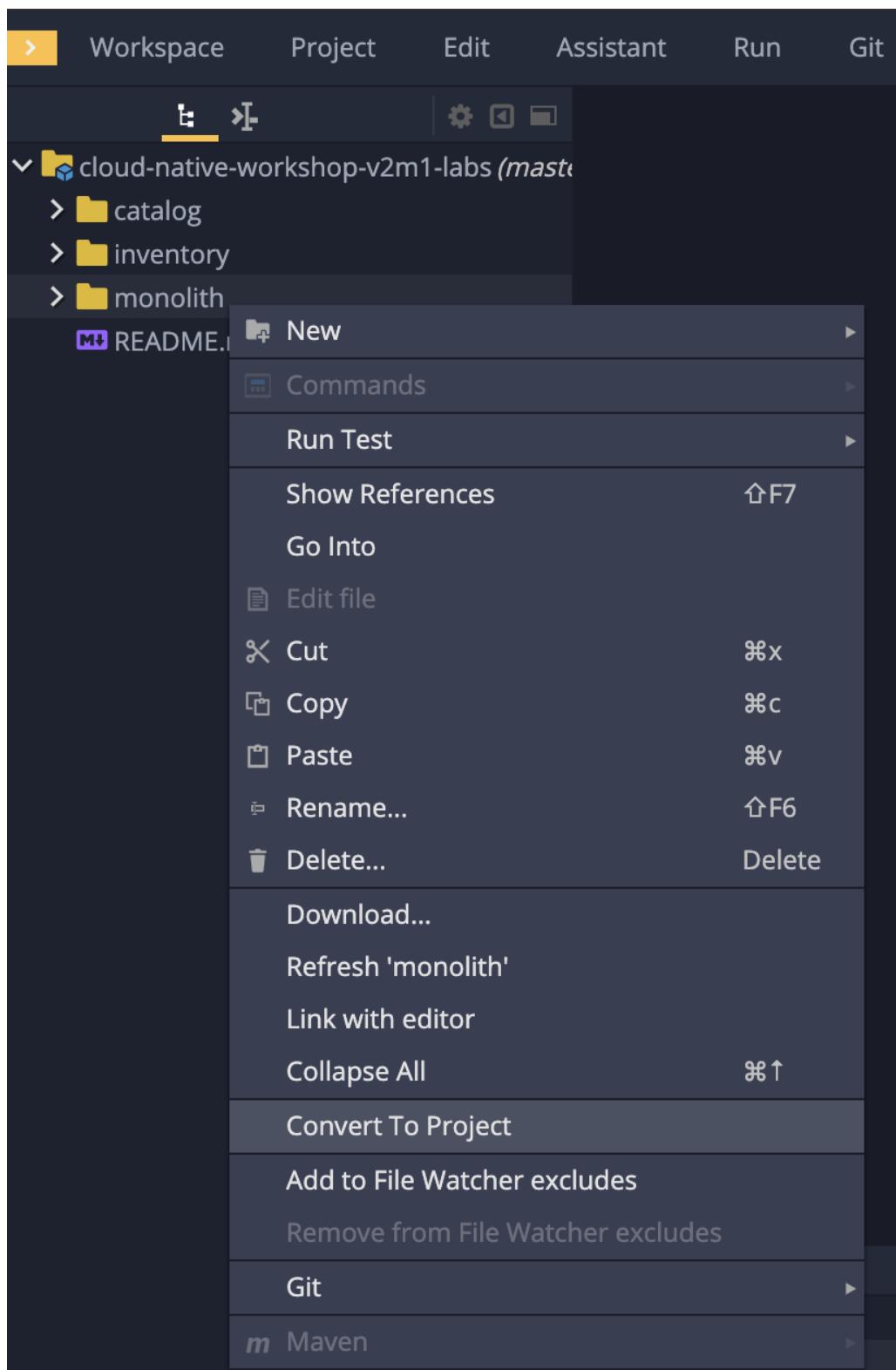
CodeReady Workspaces is a full featured IDE and provides language specific capabilities for various project types. In order to enable these capabilities, let's convert the imported project skeletons to a Maven projects. In the project explorer, right-click on each project (`monolith`, `inventory` and `catalog`) and then click on `Convert to Project` continuously.

NOTE

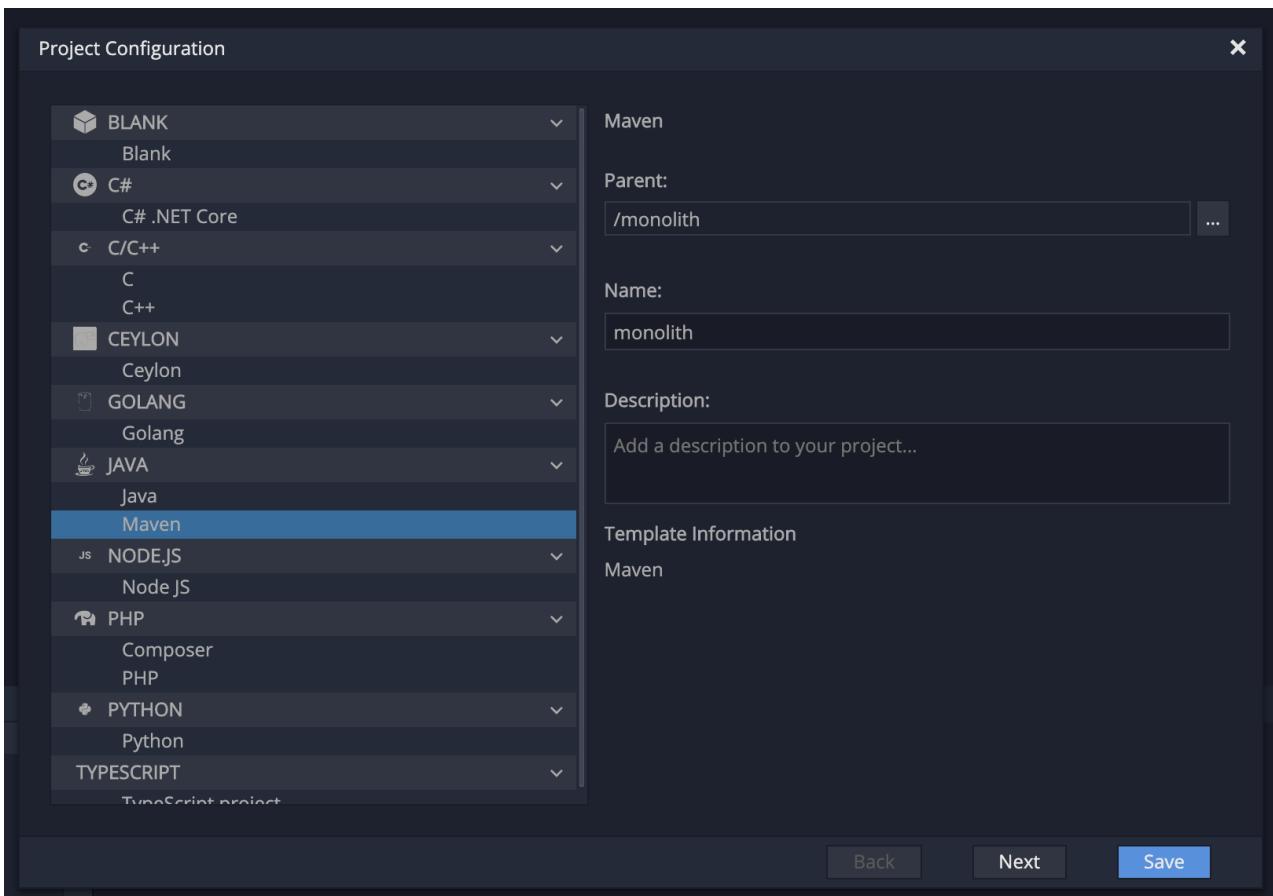
If you do not see the [Convert to Project](#) then your projects are already converted, and you should see a small icon next to each project:



If not, then convert them:

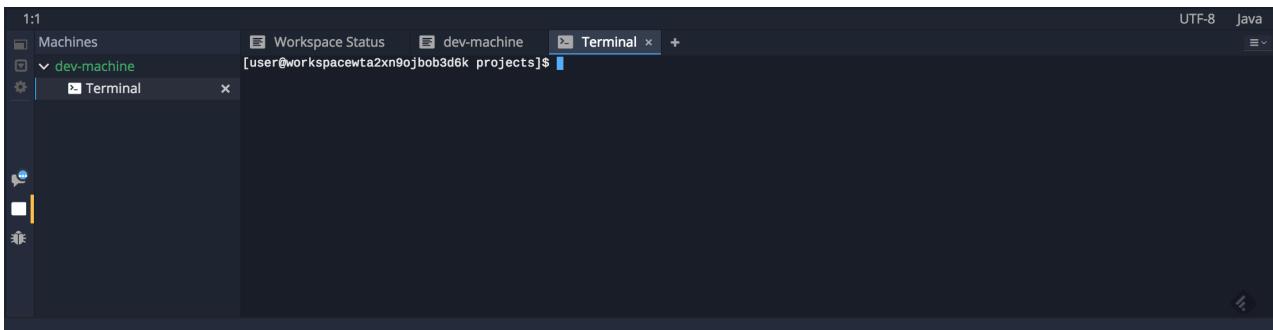


Choose **Maven** from the project configurations and then click on **Save**.



Repeat the above for all 3 projects (`monolith` , `inventory` and `catalog` projects.)

NOTE : For the rest of these labs, anytime you need to run a command in a terminal, you can use the CodeReady Workspaces Terminal window.



Login to OpenShift CLI

Although your Eclipse Che workspace is running on the Kubernetes cluster, it's running with a default restricted *Service Account* that prevents you from creating most resource types. If you've completed other modules, you're probably already logged in, but let's login again: open a Terminal and issue the following command:

```
oc login https://$KUBERNETES_SERVICE_HOST:$KUBERNETES_SERVICE_PORT --insecure-skip-tls-verify=true
```

Enter your username and password assigned to you:

- Username: userXX
- Password: r3dh4t1!

You should see like:

Login successful.

You have access to the following projects and can switch between them with 'oc project <projectname>':

```
* default
istio-system
user0-bookinfo
user0-catalog
user0-cloudnative-pipeline
user0-cloudnativeapps
user0-inventory
```

Using project "default".

Welcome! See 'oc help' to get started.

If this is the first module you are doing today

If you've already completed Module 1 (Optimizing Existing Applications), then you will already have the *CoolStore* app deployed.

If this is the first module you are completing today, you need to deploy CoolStore monolith application by running this command in a CodeReady Workspaces Terminal:

```
sh /projects/cloud-native-workshop-v2m2-labs/monolith/scripts/deploy-inventory.sh userXX
sh /projects/cloud-native-workshop-v2m2-labs/monolith/scripts/deploy-catalog.sh userXX
sh /projects/cloud-native-workshop-v2m2-labs/monolith/scripts/deploy-coolstore.sh userXX
```

| NOTE: Replace userXX with your actual username!

Wait for the commands to complete. If you see any errors, contact an instructor!

Verifying the Dev Environment

In the previous module, you created a new OpenShift project called **userXX-coolstore-dev** which represents your developer personal project in which you deployed the CoolStore monolith.

Verify Application

Let's take a moment and review the OpenShift resources that are created for the Monolith:

- Build Config: **coolstore** build config is the configuration for building the Monolith image from the source code or WAR file
- Image Stream: **coolstore** image stream is the virtual view of all coolstore container images built and pushed to the OpenShift integrated registry.
- Deployment Config: **coolstore** deployment config deploys and redeploys the Coolstore container image whenever a new coolstore container image becomes available. Similarly, the **coolstore-postgresql** does the same for the database.
- Service: **coolstore** and **coolstore-postgresql** service is an internal load balancer which identifies a set of pods (containers) in order to proxy the connections it receives to them. Backing pods can be added to or removed from a service arbitrarily while the service remains consistently available, enabling anything that depends on the service to refer to it at a consistent address (service name or IP).
- Route: **www** route registers the service on the built-in external load-balancer and assigns a public DNS name to it so that it can be reached from outside OpenShift cluster.

You can review the above resources in the [OpenShift web console](#) or using the `oc get` or `oc describe` commands (`oc describe` gives more detailed info):

You can use short synonyms for long words, like **bc** instead of **buildconfig**, **is** for **imagestream**, **dc** for **deploymentconfig**, **svc** for **service**, etc.

NOTE: Don't worry about reading and understanding the output of `oc describe`. Just make sure the command doesn't report errors!

Set the current project to *coolstore* and replace your username with **userXX**:

```
oc project userXX-coolstore-dev
```

Run these commands to inspect the elements via CodeReady Workspaces Terminal window:

```
oc get bc coolstore
```

```
oc get is coolstore
```

```
oc get dc coolstore
```

```
oc get svc coolstore
```

```
oc describe route www
```

Verify that you can access the monolith by clicking on the exposed OpenShift route to open up the sample application in a separate browser tab.

You should also be able to see both the CoolStore monolith and its database running in separate pods via CodeReady Workspaces Terminal window:

```
oc get pods -l application=coolstore
```

The output should look like this:

```
NAME          READY  STATUS   RESTARTS  AGE
coolstore-2-bpkkc    1/1    Running  0        4m
coolstore-postgresql-1-jpcb8  1/1    Running  0        9m
```

Verify Database

You can log into the running Postgres container using the following via CodeReady Workspaces Terminal window:

```
oc rsh dc/coolstore-postgresql
```

Once logged in, use the following command to execute an SQL statement to show some content from the database:

```
psql -U $POSTGRESQL_USER $POSTGRESQL_DATABASE -c 'select name from PRODUCT_CATALOG;'
```

You should see the following:

```
name
-----
Red Fedora
Forge Laptop Sticker
Solid Performance Polo
Ogio Caliber Polo
16 oz. Vortex Tumbler
Atari 2600 Joystick
Pebble Smart Watch
Oculus Rift
Lytro Camera
(9 rows)
```

Don't forget to exit the pod's shell with **exit**.

With our running project on OpenShift, in the next step we'll explore how you as a developer can work with the running app to make changes and debug the application!

Implementing Continuous Delivery

Lab1 - Automating Deployments Using Pipelines

In the previous scenarios, you deployed the Coolstore monolith using an OpenShift Template into the **userXX-coolstore-dev** Project. The template created the necessary objects (BuildConfig, DeploymentConfig, ImageStreams, Services, and Routes) and gave you as a Developer a "playground" in which to run the app, make changes and debug.

In this step, we are now going to setup a separate production environment and explore some best practices and techniques for developers and DevOps teams for getting code from the developer (**that's YOU!**) to production with less downtime and greater consistency.

Production vs. Development

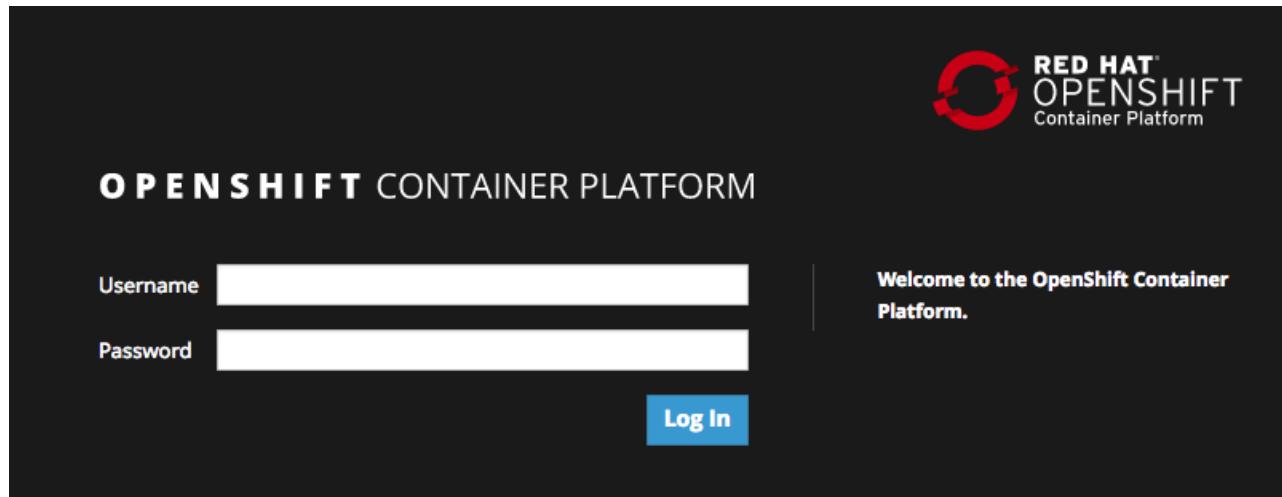
The existing **userXX-coolstore-dev** project is used as a developer environment for building new versions of the app after code changes and deploying them to the development environment.

In a real project on OpenShift, *dev*, *test* and *production* environments would typically use different OpenShift projects and perhaps even different OpenShift clusters.

For simplicity in this scenario we will only use a *dev* and *prod* environment, and no test/QA environment.

1. Create the production environment

First, open a new browser with the [OpenShift web console](#)



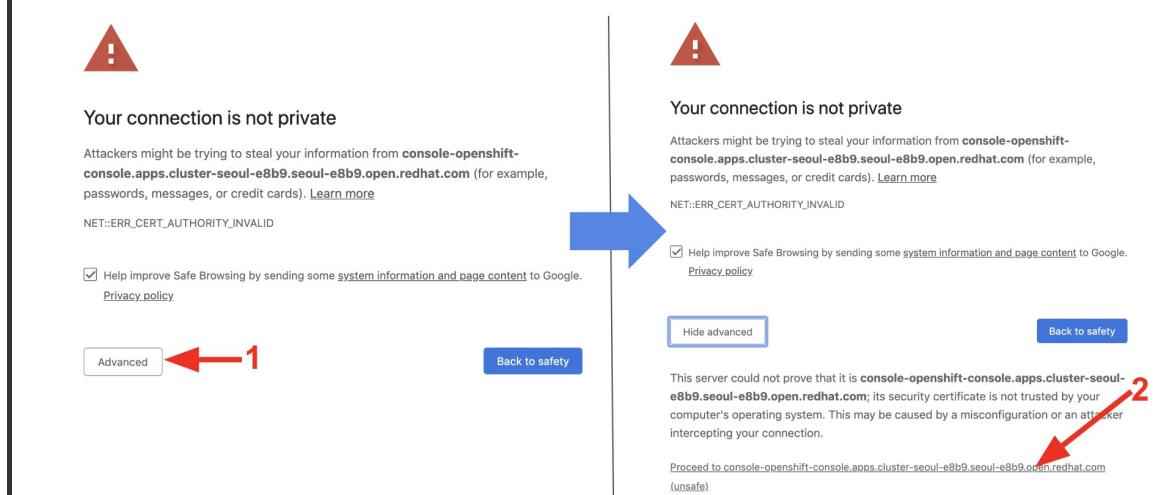
Login using:

- Username: `userXX`
- Password: `r3dh4t1!`

NOTE: Use of self-signed certificates

When you access the OpenShift web console](<https://console-openshift-console.apps.cluster-beijing-5bc2.beijing-5bc2.open.redhat.com>) or other URLs via **HTTPS** protocol, you will see browser warnings like **Your connection is not secure** since this workshop uses self-signed certificates (which you should not do in production!). For example, if you're using **Chrome**, you will see the following screen.

Click on **Advanced** then, you can access the HTTPS page when you click on **Proceed to... !!!**



Other browsers have similar procedures to accept the security exception.

You will see the OpenShift landing page:

NAME	STATUS	REQUESTER	LABELS
istio-system	Active	opentic-mgr	maistra.io/ignore-namespace=ignore
user0-bookinfo	Active	opentic-mgr	No labels
user0-catalog	Active	opentic-mgr	No labels
user0-inventory	Active	opentic-mgr	No labels

The project displayed in the landing page depends on which labs you will run today. If you will develop **Service Mesh and Identity** then you will see pre-created projects as the above screenshot.

Click **Create Project**, fill in the fields, and click **Create**:

- Name: **userXX-coolstore-prod**
- Display Name: **USERXX Coolstore Monolith - Production**
- Description: *leave this field empty*

NOTE: YOU **MUST** USE **userXX-coolstore-prod** AS THE PROJECT NAME, as this name is referenced later on and you will experience failures if you do not name it **userXX-coolstore-prod**.

This will create a new OpenShift project called **userXX-coolstore-prod** from which our production application will run.

Create Project

Name *

Display Name

Description



2. Add the production elements

In this case we'll use the production template to create the objects. Execute via CodeReady Workspaces Terminal window:

```
oc project userXX-coolstore-prod
```

And finally deploy template:

```
oc new-app --template=coolstore-monolith-pipeline-build
```

We have to deploy a **Jenkins Server** in the namespace because OpenShift 4 doesn't deploy a Jenkins server automatically when we use *Jenkins Pipeline* build strategy.

```
oc new-app --template=jenkins-ephemeral -l app=jenkins -p JENKINS_SERVICE_NAME=jenkins -p DISABLE_ADMINISTRATIVE_MONITORS=true
```

```
oc set resources dc/jenkins --limits(cpu=1, memory=2Gi) --requests(cpu=1, memory=512Mi)
```

This will use an OpenShift Template called **coolstore-monolith-pipeline-build** to construct the production application. As you probably guessed it will also include a Jenkins Pipeline to control the production application (more on this later!)

Navigate to the Web Console to see your new app and the components using this link:

Coolstore Prod Project Status at [OpenShift web console](#):

The screenshot shows the Red Hat OpenShift Container Platform Web Console. The left sidebar has a 'Status' link highlighted with a red arrow. The main content area shows the 'Project Status' for the 'coolstore-monolith-pipeline-build' project. It lists three applications: 'coolstore-prod' (status: 0 of 1 pods), 'coolstore-prod-postgresql, #1' (status: 1 of 1 pods), and 'jenkins' (status: 1 of 1 pods). Each application entry includes resource usage details like memory and cores.

You can see the production database, and an application called *Jenkins* which OpenShift uses to manage CI/CD pipeline deployments. There is no running production app just yet. The only running app is back in the *dev* environment, where you used a binary build to run the app previously.

In the next step, we'll *promote* the app from the *dev* environment to the *production* environment using an OpenShift pipeline build. Let's get going!

Promoting Apps Across Environments with Pipelines

Continuous Delivery

So far you have built and deployed the app manually to OpenShift in the *dev* environment. Although it's convenient for local development, it's an error-prone way of delivering software when extended to test and production environments.

Continuous Delivery (CD) refers to a set of practices with the intention of automating various aspects of delivery software. One of these practices is called delivery pipeline which is an automated process to define the steps a change in code or configuration has to go through in order to reach upper environments and eventually to production.

OpenShift simplifies building CI/CD Pipelines by integrating the popular [Jenkins pipelines](#) into the platform and enables defining truly complex workflows directly from within OpenShift.

The first step for any deployment pipeline is to store all code and configurations in a source code repository. In this workshop, the source code and configurations are stored in a [GitHub repository](#) we've been using. This repository has been copied locally to your environment and you've been using it ever since!

Pipelines

OpenShift has built-in support for CI/CD pipelines by allowing developers to define a [Jenkins pipeline](#) for execution by a Jenkins automation engine, which is automatically provisioned on-demand by OpenShift when needed.

The build can get started, monitored, and managed by OpenShift in the same way as any other build types e.g. S2I. Pipeline workflows are defined in a `Jenkinsfile`, either embedded directly in the build configuration, or supplied in a Git repository and referenced by the build configuration. They are written using the [Groovy scripting language](#).

As part of the production environment template you used in the last step, a Pipeline build object was created. Ordinarily the pipeline would contain steps to build the project in the *dev* environment, store the resulting image in the local repository, run the image and execute tests against it, then wait for human approval to *promote* the resulting image to other environments like test or production.

3. Inspect the Pipeline Definition

Our pipeline is somewhat simplified for the purposes of this Workshop. Inspect the contents of the pipeline by navigating *Builds > Build Configs* and click on [monolith-pipeline](#) in the [OpenShift web console](#). Then, you will see the details of `Jenkinsfile` on the right side:

Build Config Overview

NAME
monolith-pipeline

NAMESPACE
NS user0-coolstore-prod

LABELS
app=coolstore-monolith-pipeline-build build=monolith-pipeline template=coolstore-monolith-pipeline-build

ANNOTATIONS
1 Annotation

CREATED AT
Aug 5, 5:38 pm

JENKINSFILE

```
pipeline {  
    agent {  
        label 'maven'  
    }  
    stages {  
        stage ('Build') {  
            steps {  
                sleep 5  
            }  
        }  
        stage ('Run Tests in DEV') {  
            steps {  
                sleep 10  
            }  
        }  
        stage ('Approve Go Live') {  
            steps {  
                timeout(time:30, unit:'MINUTES') {  
                    input message:'Go Live in Production (switch to new version)?'  
                }  
            }  
        }  
        stage ('Deploy to PROD') {  
            steps {  
                script {  
                    openshift.withCluster()  
                    openshift.tag("userXX-coolstore-dev/coolstore:latest", "userXX-coolst  
                }  
            }  
        }  
    }  
}
```

You can also inspect this via the following command via CodeReady Workspaces Terminal window:

```
oc describe bc/monolith-pipeline
```

You can see the Jenkinsfile definition of the pipeline in the output:

Jenkinsfile contents:

```
pipeline {  
    agent {  
        label 'maven'  
    }  
    stages {  
        stage ('Build') {  
            steps {  
                sleep 5  
            }  
        }  
        stage ('Run Tests in DEV') {  
            steps {  
                sleep 10  
            }  
        }  
        stage ('Deploy to PROD') {  
            steps {  
                script {  
                    openshift.withCluster() {  
                        openshift.tag("userXX-coolstore-dev/coolstore:latest", "userXX-coolstore-  
prod/coolstore:prod")  
                    }  
                }  
            }  
        }  
        stage ('Run Tests in PROD') {  
            steps {  
                sleep 30  
            }  
        }  
    }  
}
```

NOTE: You have to replace your username with **userXX** in Jenkinsfile via clicking on **YAML** tab. For example, if your username is user0, it will be user0-coolstore-dev and user0-coolstore-prod. Don't forget to click on **Save**.

```

24 type: JenkinsPipeline
25 JenkinsPipelineStrategy:
26   jenkinsfile: |
27     pipeline {
28       agent {
29         label 'maven'
30       }
31       stages {
32         stage ('Build') {
33           steps {
34             sleep 5
35           }
36         }
37         stage ('Run Tests in DEV') {
38           steps {
39             sleep 10
40           }
41         }
42         stage ('Deploy to PROD') {
43           steps {
44             script {
45               openshift.withCluster() {
46                 openshift.tag("userXX-coolstore-dev/coolstore:latest", "userXX-coolstore-prod/coolstore:prod")
47               }
48             }
49           }
50         }
51         stage ('Run Tests in PROD') {
52           steps {
53             sleep 30
54           }
55         }
56       }
57     }

```

Save Reload Cancel

The pipeline syntax allows creating complex deployment scenarios with the possibility of defining checkpoints for manual interaction and approval processes using [the large set of steps and plugins that Jenkins provides](#) in order to adapt the pipeline to the processes used in your team. You can see a few examples of advanced pipelines in the [OpenShift GitHub Repository](#).

To simplify the pipeline in this workshop, we simulate the build and tests and skip any need for human input. Once the pipeline completes, it deploys the app from the *dev* environment to our *production* environment using the above `tag()` method within the `openshift` object, which simply re-tags the image you already created using a tag which will trigger deployment in the production environment.

4. Promote the dev image to production using the pipeline

Before promoting the dev image, you need to modify a **RoleBinding** to access the dev image by Jenkins. This allows the Jenkins service account in the **userXX-coolstore-prod** project to access the image within the **userXX-coolstore-dev** project.

Go to overview page of the `userXX-coolstore-dev` project, then navigate to *Administration > Role Bindings*. Click on **ci_admin**:

The screenshot shows the OpenShift web console with the sidebar expanded. The 'Administration' section is selected, and the 'Role Bindings' link is highlighted with a red arrow. The main content area displays a table of role bindings. One row, 'ci_admin', is highlighted with a red arrow, indicating it is the target for modification.

NAME	ROLE REF	SUBJECT KIND	SUBJECT NAME	NAMESPACE
admin	admin	User	user0	user0-coolstore-dev
ci_admin	admin	Group	system:serviceaccounts:userX-coolstore-prod	user0-coolstore-dev
coolstore-serviceaccount_view	view	ServiceAccount	coolstore-serviceaccount	user0-coolstore-dev

Move to **YAML** tab and replace your username with *userXX* then click on **Save**:

The screenshot shows the YAML editor for the 'ci_admin' role binding. The 'YAML' tab is selected. The code editor highlights the 'name' field in the 'subjects' section, which is 'system:serviceaccounts:userXX-coolstore-prod'. A red arrow points to this line. At the bottom, there are three buttons: 'Save' (highlighted with a red arrow), 'Reload', and 'Cancel'.

```

1 kind: RoleBinding
2 apiVersion: rbac.authorization.k8s.io/v1
3 metadata:
4   name: ci_admin
5   namespace: user0-coolstore-dev
6   selfLink: >-
7     /apis/rbac.authorization.k8s.io/v1/namespaces/user0-coolstore-dev/rolebindings/ci_admin
8   uid: bddbd438-b67a-11e9-a923-0a0faa83ac92
9   resourceVersion: '3036612'
10  creationTimestamp: '2019-08-04T05:43:09Z'
11  labels:
12    app: coolstore-monolith-binary-build
13    template: coolstore-monolith-binary-build
14  annotations:
15    openshift.io/generated-by: OpenShiftNewApp
16  subjects:
17    - kind: Group
18      apiGroup: rbac.authorization.k8s.io
19      name: 'system:serviceaccounts:userXX-coolstore-prod'
20  roleRef:
21    apiGroup: rbac.authorization.k8s.io
22    kind: ClusterRole
23    name: admin
24

```

Let's invoke the build pipeline by using [OpenShift web console](#). Open the production project in the web console.

Next, navigate to *Builds > Build Configs > monolith-pipeline*, click the small menu at the far right, and click *Start Build*:

Project: user0-coolstore-prod

Build Configs

Create Build Config

Filter Build Configs by name...

0 Docker | 1 JenkinsPipeline | 0 Source | 0 Custom | Select All Filters

NAME | NS | LABELS | CREATED

monolith-pipeline | user0-coolstore-prod | app=coolstore-monolith-pipeline-build, build=monolith-pipeline, template=coolstore-monolith-pipeline-build | 2 hours ago

Actions

- Start Build
- Edit Environment
- Edit Labels
- Edit Annotations
- Edit Build Config
- Delete Build Config

This will start the pipeline. *It will take a minute or two to start the pipeline!* Future runs will not take as much time as the Jenkins infrastructure will already be warmed up. You can watch the progress of the pipeline:

Project: user0-coolstore-prod

monolith-pipeline > Build Details

monolith-pipeline-1

Actions

Overview **YAML** **Environment** **Logs** **Events**

Build Overview

Build 1 a minute ago [View Logs](#)

Build less than a minute ago → Run Tests in DEV less than a minute ago → Deploy to PROD a few seconds ago → Run Tests in PROD a few seconds ago

NAME	monolith-pipeline-1	STATUS	Running
NAMESPACE	user0-coolstore-prod	TYPE	JenkinsPipeline

Once the pipeline completes, return to the Prod Project Status at [OpenShift web console](#) and notice that the application is now deployed and running!

Project: user0-coolstore-prod

Project Status

Resources **Dashboard**

Group by: Application | Filter by name...

coolstore-monolith-pipeline-build

coolstore-prod, #1	565.7 MiB	0.157 cores	1 of 1 pods
coolstore-prod-postgresql, #1	112.2 MiB	0.008 cores	1 of 1 pods
jenkins	583.9 MiB	0.858 cores	1 of 1 pods
jenkins, #2			

It may take a few moments for the container to deploy fully.

Congratulations!

You have successfully setup a development and production environment for your project and can use this workflow for future projects as well.

In the next step, we'll add a human interaction element to the pipeline, so that you as a project lead can be in charge of approving changes.

More Reading

[OpenShift Pipeline Documentation](#)

Adding Pipeline Approval Steps

In previous steps, you used an OpenShift Pipeline to automate the process of building and deploying changes from the dev environment to production.

In this step, we'll add a final checkpoint to the pipeline which will require you as the project lead to approve the final push to production.

5. Edit the pipeline

Ordinarily your pipeline definition would be checked into a source code management system like Git, and to change the pipeline you'd edit the *Jenkinsfile* in the source base. For this workshop we'll just edit it directly to add the necessary changes. You can edit it with the **oc** command but we'll use the Web Console.

Go back to *Builds > Build Configs > monolith-pipeline* then click on *Edit Build Config*.

The screenshot shows the Red Hat OpenShift Container Platform interface. The left sidebar has a navigation menu with links: Home, Catalog, Workloads, Networking, Storage, Builds (with 'Build Configs' highlighted by a red arrow), Image Streams, Monitoring, and Administration. The main content area is titled 'Build Configs' and shows a table with one item: 'monolith-pipeline'. The table columns are NAME, NAMESPACE, LABELS, and CREATED. The 'monolith-pipeline' row has a 'user0-coolstore-prod' namespace, labels 'app=coolstore-monolith-pipeline-build', 'build=monolith-pipeline', and 'template=coolstore-monolith-pipeline-build', and was created '14 minutes ago'. To the right of the table is a context menu with options: Start Build, Edit Environment, Edit Labels, Edit Annotations, Edit Build Config (which is highlighted by a red arrow), and Delete Build Config.

Click on **YAML** tab and add a new stage to the pipeline, just before the *Deploy to PROD* stage:

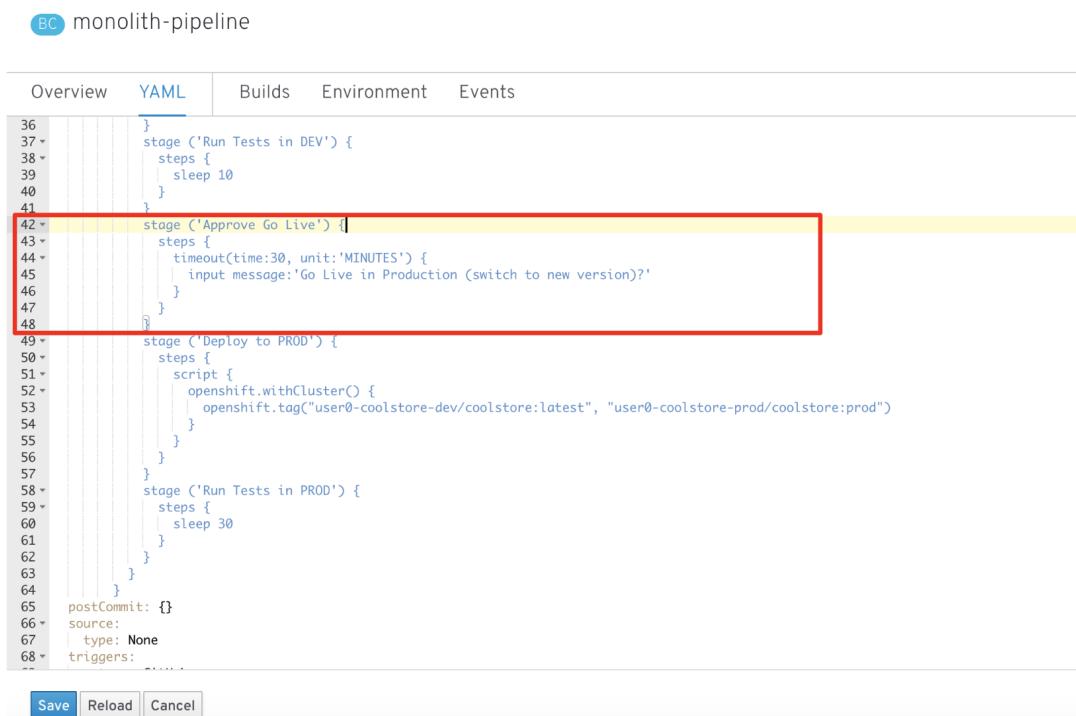
NOTE: You will need to copy and paste the below code into the right place as shown in the below image.

```

stage ('Approve Go Live') {
    steps {
        timeout(time:30, unit:'MINUTES') {
            input message:'Go Live in Production (switch to new version)?'
        }
    }
}

```

Your final pipeline should look like:



The screenshot shows a CI/CD pipeline configuration in YAML. The pipeline consists of several stages: 'Run Tests in DEV', 'Approve Go Live', 'Deploy to PROD', and 'Run Tests in PROD'. The 'Approve Go Live' stage is highlighted with a red box. At the bottom of the screen, there are three buttons: 'Save', 'Reload', and 'Cancel'.

```

monolith-pipeline

Overview YAML Builds Environment Events

36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68

```

```

stage ('Run Tests in DEV') {
    steps {
        sleep 10
    }
}

stage ('Approve Go Live') {
    steps {
        timeout(time:30, unit:'MINUTES') {
            input message:'Go Live in Production (switch to new version)?'
        }
    }
}

stage ('Deploy to PROD') {
    steps {
        script {
            openshift.withCluster() {
                openshift.tag("user0-coolstore-dev/coolstore:latest", "user0-coolstore-prod/coolstore:prod")
            }
        }
    }
}

stage ('Run Tests in PROD') {
    steps {
        sleep 30
    }
}

postCommit: 0
source:
| type: None
triggers:

```

Save Reload Cancel

Click **Save**.

6. Make a simple change to the app

With the approval step in place, let's simulate a new change from a developer who wants to change the color of the header in the coolstore to a blue background color.

First, open *monolith/src/main/webapp/app/css/coolstore.css* via CodeReady Workspace, which contains the CSS stylesheet for the CoolStore app.

Add the following CSS to turn the header bar background to blue (**Copy** to add it at the bottom):

```
.navbar-header {
    background: blue
}
```

Now we need to update the catalog endpoint in the monolith application. Copy the route URL of catalog service using following `oc` command in CodeReady Workspaces Terminal. Replace your username with `userXX`:

```
echo "http://$(oc get route -n userXX-catalog | grep catalog | awk '{print $2}')"
```

In the **monolith** project (within the root **cloud-native-workshop-v2m2-labs** project), open `catalog.js` in `src/main/webapp/app/services` and add a line as shown in the image to define the value of `baseUrl`.

```
baseUrl="http://REPLACEURL/services/products";
```

| Replace `REPLACEURL` with the URL emitted from the previous `echo` command

Next, re-build the app once more via CodeReady Workspaces Terminal:

```
cd /projects/cloud-native-workshop-v2m2-labs/monolith
```

```
mvn clean package -Popenshift
```

And re-deploy it to the dev environment using a binary build just as we did before via CodeReady Workspaces Terminal:

```
oc start-build -n userXX-coolstore-dev coolstore --from-file=deployments/ROOT.war --follow
```

Now wait for it to complete the deployment via CodeReady Workspaces Terminal:

```
oc -n userXX-coolstore-dev rollout status -w dc/coolstore
```

And verify that the blue header is visible in the dev application by navigating to the `userXX-coolstore-dev` project in the OpenShift Console, and then going to *Networking > Routes* and clicking on the route URL. It should look like the following:

| If it doesn't, you may need to do a hard browser refresh. Try holding the shift key while clicking the browser refresh button.

The screenshot shows the Red Hat Cool Store homepage with a blue header bar. The header includes the Red Hat logo, the text "Red Hat Cool Store", "Your Shopping Cart", and "All Orders". On the right side of the header, there are links for "Shopping Cart \$0.00 (0 item(s))" and "Sign In Unavailable".

Red Fedora
Official Red Hat Fedora

\$34.99
1 Add To Cart 736 left!

Forge Laptop Sticker
JBoss Community Forge Project Sticker

\$8.50
1 Add To Cart 512 left!

Solid Performance Polo
Moisture-wicking, antimicrobial 100% polyester design wicks for life of garment. No-curl, rib-knit collar; special collar band maintains crisp fold; three-button placket with dyed-to-match buttons; hemmed sleeves; even bottom with side vents; Import. Embroidery. Red Pepper.

\$17.80
1 Add To Cart 256 left!

Then navigating to the `userXX-coolstore-prod` project in the OpenShift Console, and then going to *Networking > Routes* and clicking on the route URL for the production app. It should still be black:

The screenshot shows the Red Hat Cool Store homepage with a blue header bar. The header includes the Red Hat logo, the text "Red Hat Cool Store", "Your Shopping Cart", and "All Orders". On the right side of the header, there are links for "Shopping Cart \$0.00 (0 item(s))" and "Sign In Unavailable".

Red Fedora
Official Red Hat Fedora

\$34.99
1 Add To Cart 736 left!

Forge Laptop Sticker
JBoss Community Forge Project Sticker

\$8.50
1 Add To Cart 512 left!

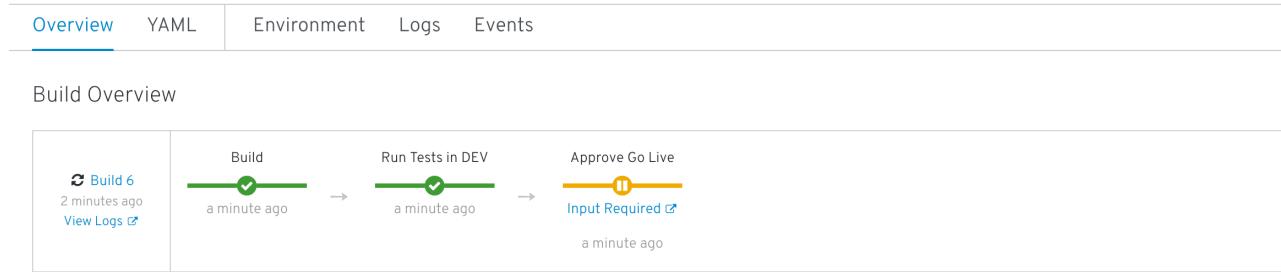
Solid Performance Polo
Moisture-wicking, antimicrobial 100% polyester design wicks for life of garment. No-curl, rib-knit collar; special collar band maintains crisp fold; three-button placket with dyed-to-match buttons; hemmed sleeves; even bottom with side vents; Import. Embroidery. Red Pepper.

\$17.80
1 Add To Cart 256 left!

We're happy with this change in dev, so let's promote the new change to prod, using the new approval step!

7. Run the pipeline again

Invoke the pipeline once more by navigating to *Builds > Build Configs > monolith-pipeline > Rebuild*. The same pipeline progress will be shown, however before deploying to prod, you will see a prompt in the pipeline:



Click on the link for **Input Required**. This will open a new tab and direct you to Jenkins itself, where you can login with the same credentials as OpenShift:

- Username: `userXX`
- Password: `r3dh4t1!`

Accept the browser certificate warning and the Jenkins/OpenShift permissions, and then you'll find yourself at the approval prompt:

Click on **Console Output** on left menu then click on **Proceed**.

The screenshot shows the Jenkins 'Console Output' page for build #4 of the 'monolith-pipeline' job. The left sidebar lists various build-related links: Status, Changes, Console Output (which is highlighted with a red arrow), View as plain text, Edit Build Information, Paused for Input, Open Blue Ocean, Thread Dump, Pause/resume, Replay, Pipeline Steps, Workspaces, and Previous Build. The main content area shows the log output for the pipeline. Near the bottom of the log, there is a prompt: 'Live in Production (switch to new version)?' followed by two buttons: 'Proceed' and 'Abort'. A red arrow points from the right side of the screen down to the 'Proceed' button.

```

Jenkins > user0-coolstore-prod > user0-coolstore-prod/monolith-pipeline > #4
>Status
Changes
Console Output (highlighted)
View as plain text
Edit Build Information
Paused for Input
Open Blue Ocean
Thread Dump
Pause/resume
Replay
Pipeline Steps
Workspaces
Previous Build

Console Output

OpenShift Build user0-coolstore-prod/monolith-pipeline-5
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] Start of Pipeline
[Pipeline] node
Still waiting to schedule task
'Jenkins' doesn't have label 'maven'
Agent maven-gggsm is provisioned from template Kubernetes Pod Template
Agent specification [Kubernetes Pod Template] (maven):
* [jnlp] image-registry.openshift-image-registry.svc:5000/openshift/jenkins-agent-maven:latest

Running on maven-gggsm in /tmp/workspace/user0-coolstore-prod/user0-coolstore-prod-monolith-pipeline
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Build)
[Pipeline] sleep
Sleeping for 5 sec
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Run Tests in DEV)
[Pipeline] sleep
Sleeping for 10 sec
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Approve Go Live)
[Pipeline] timeout
Timeout set to expire in 30 min
[Pipeline] {
[Pipeline] input
Live in Production (switch to new version)?
Proceed or Abort

```

8. Approve the change to go live

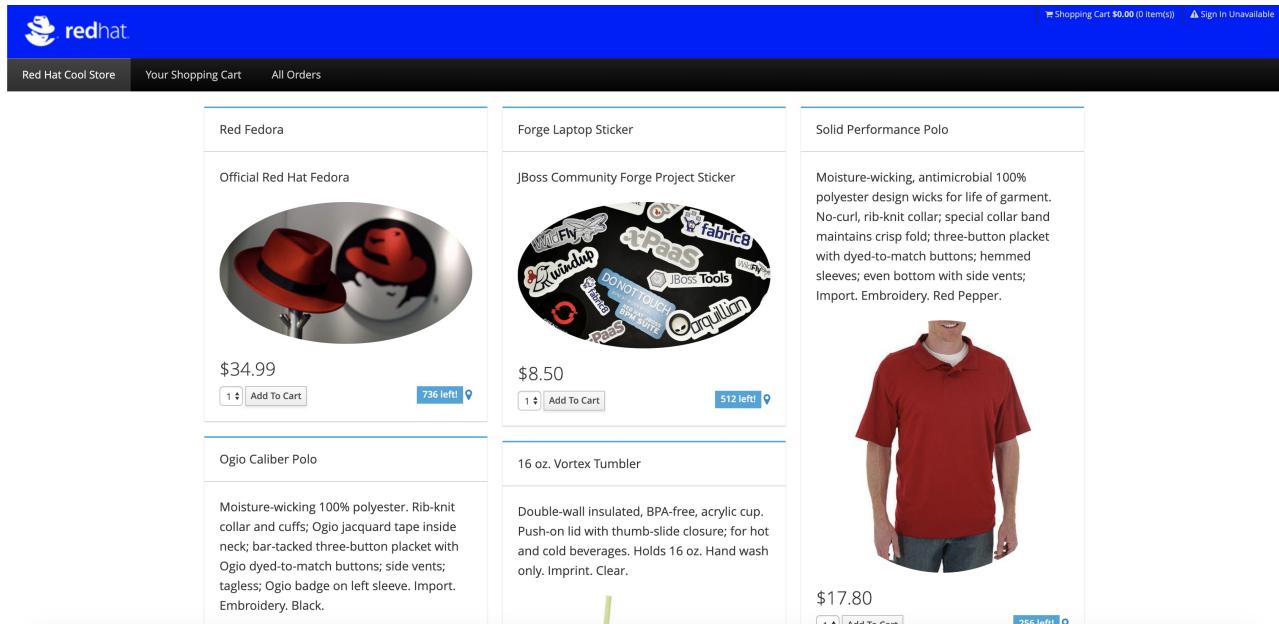
Click **Proceed**, which will approve the change to be pushed to production. You could also have clicked **Abort** which would stop the pipeline immediately in case the change was unwanted or unapproved.

Once you click *Proceed*, you will see the log file from Jenkins showing the final progress and deployment.

Wait for the production deployment to complete via CodeReady Workspaces Terminal:

```
oc rollout -n userXX-coolstore-prod status -w dc/coolstore-prod
```

Once it completes, verify that the production application has the new change (blue header):



If it doesn't, you may need to do a hard browser refresh. Try holding the shift key while clicking the browser refresh button.

9. Run the Pipeline on Every Code Change

Manually triggering the deployment pipeline to run is useful but it would be better to run the pipeline automatically on every change in code or configuration, at least to lower environments (e.g. dev and test) and ideally all the way to production with some manual approvals in-place.

In order to automate triggering the pipeline, you can define a webhook on your Git repository to notify OpenShift on every commit that is made to the Git repository and trigger a pipeline execution.

You can get see the webhook links in the [OpenShift web console](#) by going to *Builds > Build Configs > monolith-pipeline*. Look for the *Generic secret* value in the *YAML* tab. Copy this down.

Then go back to the *Overview* tab. At the bottom you'll find the *Generic webhook url* which you will need (along with the secret) in the next steps.

Go to your Git repository at <http://gogs-labs-infra.apps.cluster-beijing-5bc2.beijing-5bc2.open.redhat.com/userXX/cloud-native-workshop-v2m2-labs.git> (replace userXX with your username and open this URL in a new tab), Click **Sign In** and sign in with your credentials:

- Username: userXX (replace with your username)
- Password: r3dh4t1!

then click on **Settings**.

No Description

5 Commits 1 Branches 0 Releases

Branch: master ▾ cloud-native... New file Upload file HTTP SSH http://gogs-labs-infra.apps.svc.cluster-beijing-5bc2.beijing-5bc2.open.redhat.com/settings

Daniel Oh 9aa8f2016e Fix OrderServiceMDB	2 days ago
monolith 9aa8f2016e Fix OrderServiceMDB	2 days ago
README.md 5c2ef7675d Initial commit	5 days ago

On the left menu, click on **Webhooks** and then on **Add Webhook** button and then **Gogs**.

Create a webhook with the following details:

- Payload URL: paste the Generic webhook url you copied from the **monolith-pipeline** (make sure to replace the `secret` value in the URL!)
- Content type: **application/json**

Click on **Add Webhook**.

Add Webhook



Gogs will send a POST request to the URL you specify, along with details regarding the event that occurred. You can also specify what kind of data format you'd like to get upon triggering the hook (JSON, x-www-form-urlencoded, XML, etc). More information can be found in our [Webhooks Guide](#).

Payload URL *

`https://master.seoul-df03.openshiftworkshop.com:443/apis/build.openshift.io/v1/namespaces/coolstore-pr`

Content Type

application/json

Secret

Secret will be sent as SHA256 HMAC hex digest of payload via X-Gogs-Signature header.

When should this webhook be triggered?

- Just the push event.
- I need **everything**.
- Let me choose what I need.

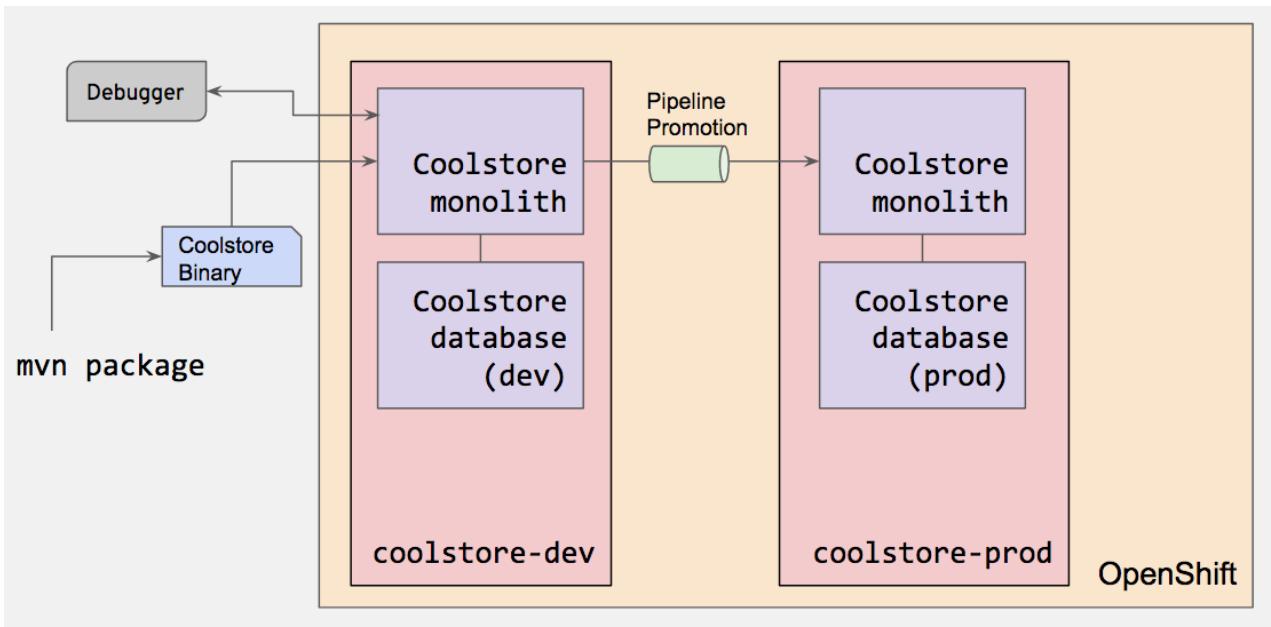
Active

Details regarding the event which triggered the hook will be delivered as well.

[Add Webhook](#)

All done. You can **click on the newly defined webhook** to see the list of *Recent Delivery*. Click on the **Test Delivery** button allows you to manually trigger the webhook for testing purposes. Click on it and verify that the *monolith-pipeline* starts running immediately (navigate to *Builds > Builds* then you should see one running. Click on it to ensure the pipeline is executing, and optionally confirm the *Approve Go Live* as before).

Congratulations! You have added a human approval step for all future developer changes. You now have two projects that can be visualized as:



Summary

In this lab, you learned how to use the OpenShift Container Platform as a developer to build, and deploy applications. You also learned how OpenShift makes your life easier as a developer, architect, and DevOps engineer.

You can use these techniques in future projects to modernize your existing applications and add a lot of functionality without major re-writes.

The monolithic application we've been using so far works great, but is starting to show its age. Even small changes to one part of the app require many teams to be involved in the push to production.

Debugging Applications

Lab2 - Debugging Applications

In this lab, you will debug the coolstore application using Java remote debugging and look into line-by-line code execution as the code runs on Quarkus.

1. Enable Remote Debugging

Remote debugging is a useful debugging technique for application development which allows looking into the code that is being executed somewhere else on a different machine and execute the code line-by-line to help investigate bugs and issues. Remote debugging is part of Java SE standard debugging architecture which you can learn more about it in [Java SE docs](#).

Quarkus in development mode enables hot deployment with background compilation, which means that when you modify your Java files and/or your resource files and refresh your browser, these changes will automatically take effect. This works too for resource files like the configuration property file.

This will also listen for a debugger on port 5005. If you want to wait for the debugger to attach before running you can pass -Ddebug on the command line. If you don't want the debugger at all you can use -Ddebug=false.

An easier approach would be to use the Quarkus maven plugin to enable remote debugging on the Inventory pod. It also forwards the default remote debugging port, 5005, from the Inventory pod to your workstation so simplify connectivity.

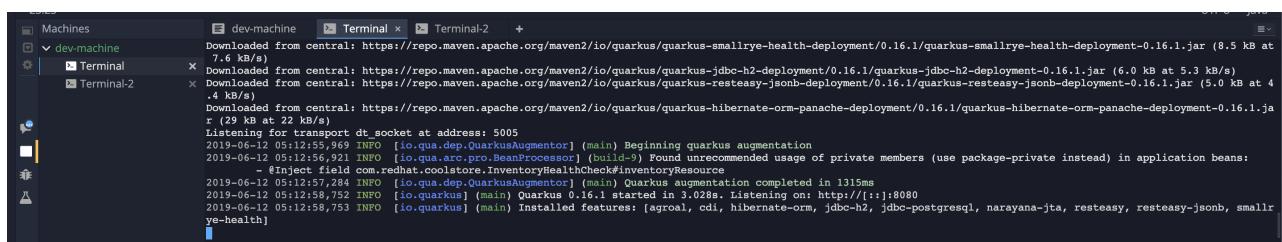
Enable remote debugging on Inventory by running the following inside the **inventory** directory (*/projects/cloud-native-workshop-v2m2-labs/inventory*) in the CodeReady Workspaces Terminal window:

```
mvn compile quarkus:dev
```

The default port for remoting debugging is **5005** but you can change the default port using **-Ddebug=port_num**

You are all set now to start debugging using the tools of your choice.

Do not wait for the command to return! The Quarkus maven plugin keeps the forwarded port open so that you can start debugging remotely.



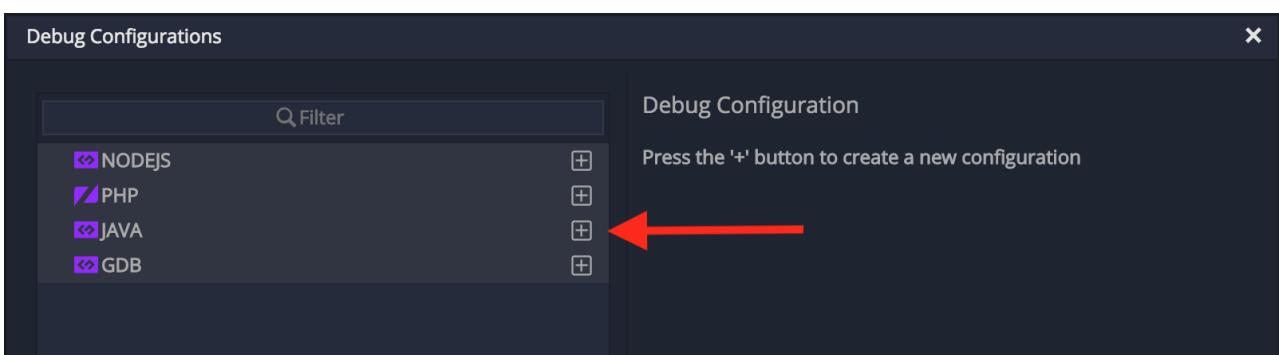
```
Downloaded from central: https://repo.maven.apache.org/maven2/io/quarkus/quarkus-smallrye-health-deployment/0.16.1/quarkus-smallrye-health-deployment-0.16.1.jar (8.5 kB at 7.6 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/io/quarkus/quarkus-jdbc-h2-deployment/0.16.1/quarkus-jdbc-h2-deployment-0.16.1.jar (6.0 kB at 5.3 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/io/quarkus/quarkus-resteasy-jsonb-deployment/0.16.1/quarkus-resteasy-jsonb-deployment-0.16.1.jar (5.0 kB at 4.4 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/io/quarkus/quarkus-hibernate-orm-panache-deployment/0.16.1/quarkus-hibernate-orm-panache-deployment-0.16.1.jar (29 kB at 22 kB/s)
Listening for transport dt_socket at address: 5005
2019-06-12 05:12:55,969 INFO [io.qua.dep.QuarkusAugmentor] (main) Beginning quarkus augmentation
2019-06-12 05:12:56,921 INFO [io.qua.arc.pro.BeanProcessor] (build-9) Found unrecommended usage of private members (use package-private instead) in application beans:
- #inject field com.redhat.cooldstore.InventoryHealthCheck.inventoryResource
2019-06-12 05:12:57,284 INFO [io.qua.dep.QuarkusAugmentor] (main) Quarkus augmentation completed in 135ms
2019-06-12 05:12:58,752 INFO [io.quarkus] (main) Quarkus 0.16.1 started in 3.028s. Listening on: http://[::]:8080
2019-06-12 05:12:58,753 INFO [io.quarkus] (main) Installed features: [agroal, cdi, hibernate-orm, jdbc-postgresql, narayana-jta, resteasy, resteasy-jsonb, smallrye-health]
```

2. Remote Debug with CodeReady Workspace

CodeReady Workspaces provides a convenience way to remotely connect to Java applications running inside containers and debug while following the code execution in the IDE.

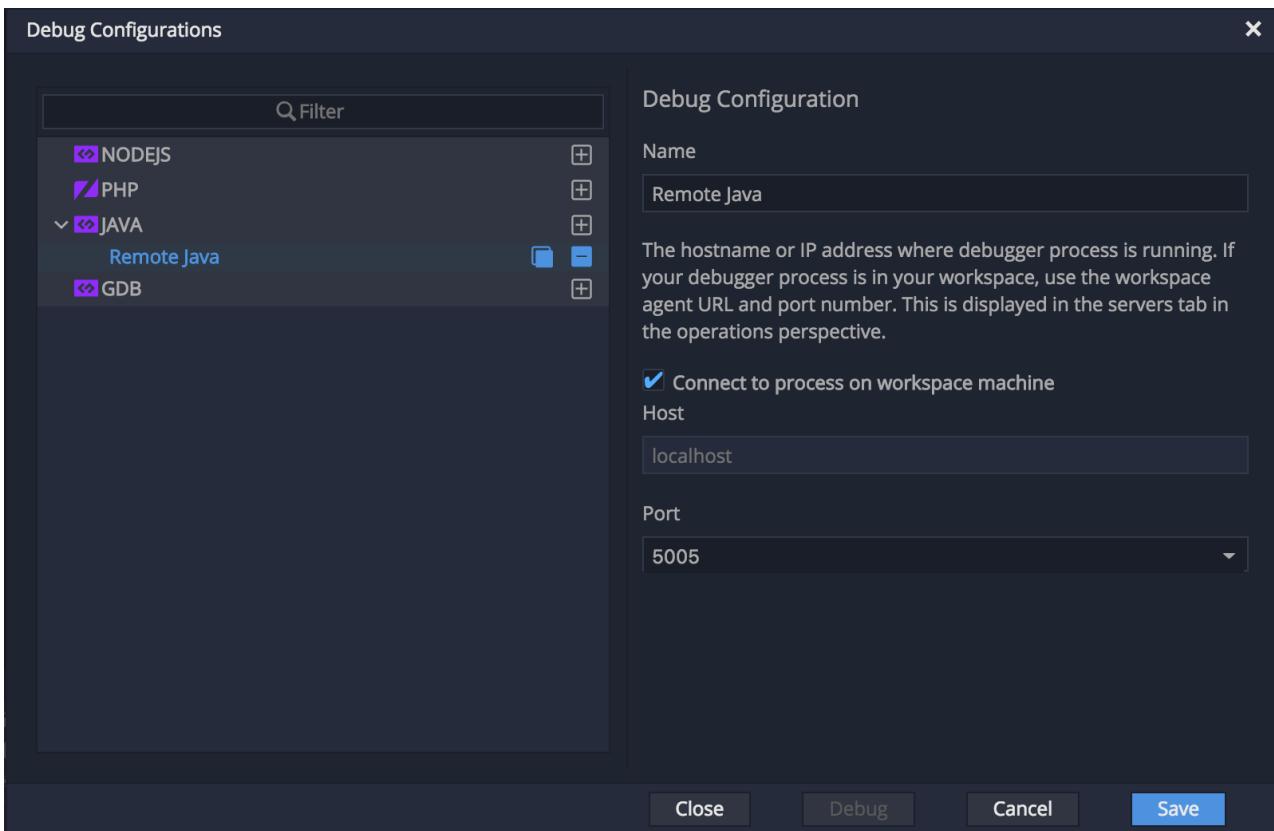
From the **Run** menu, click on **Edit Debug Configurations....**

The window shows the debuggers available in CodeReady Workspaces. Click on the plus sign near the Java debugger.



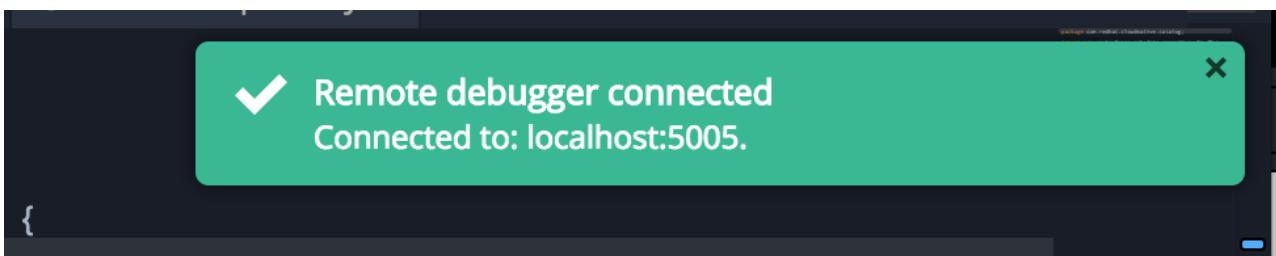
Configure the remote debugger and click on the **Save** button:

- Check **Connect to process on workspace machine**
- Port: **5005**



You can now click on *Save* then the *Debug* button to make CodeReady Workspaces connect to the Inventory service running on OpenShift.

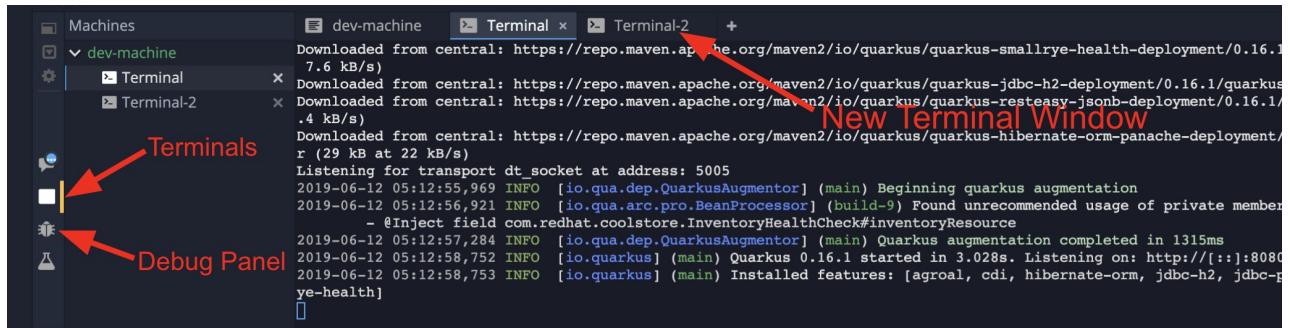
You should see a confirmation that the remote debugger is successfully connected.



Open `com.redhat.coolstore.InventoryResource` and single-click on the editor sidebar on the line number of the first line of the `getAvailability()` method to add a breakpoint to that line.

```
29
30     @GET
31     @Path("{itemId}")
32     public List<Inventory> getAvailability(@PathParam String itemId) {
33         return Inventory.<Inventory>streamAll()
34             .filter(p -> p.itemId.equals(itemId))
35             .collect(Collectors.toList());
36     } Breakpoint
37
38     @Provider
39     public static class ErrorMapper implements ExceptionMapper<Exception> {
40
41         @Override
42         public Response toResponse(Exception exception) {
43             int code = 500;
44             if (exception instanceof WebApplicationException) {
45                 code = ((WebApplicationException) exception).getResponse().getStatus();
46             }
47         }
48     }
49 }
```

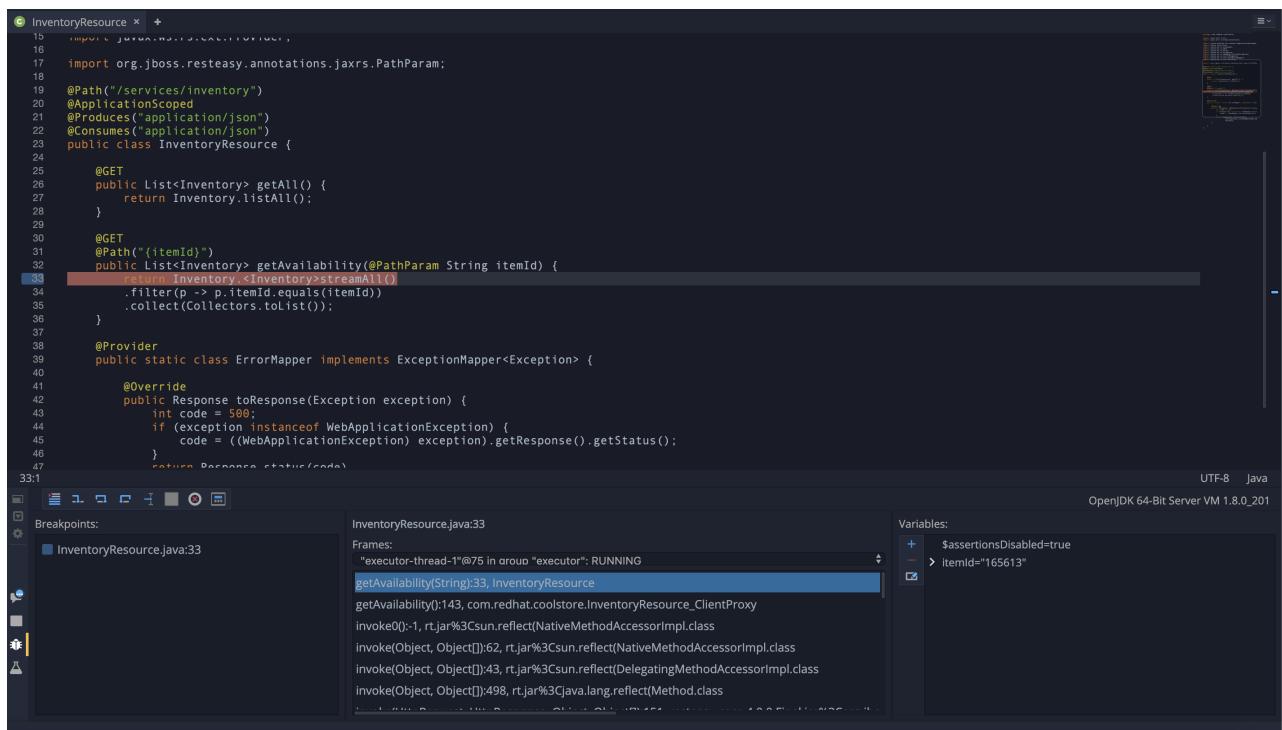
Note that you can use the the following icons to switch between debug and terminal windows.



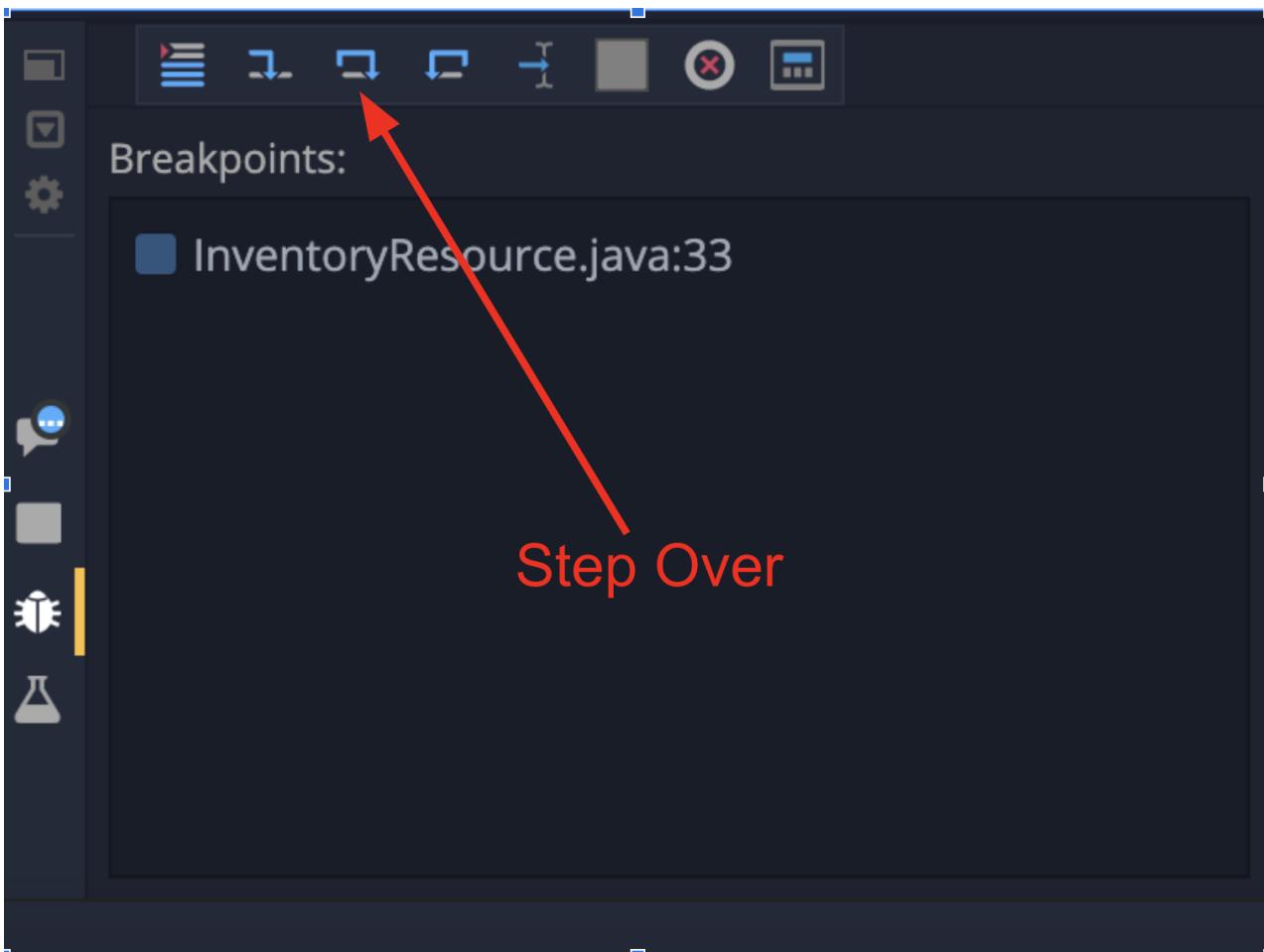
Invoke the endpoint URL of the Inventory service using `curl` command in a separate Terminal:

```
curl http://localhost:8080/services/inventory/165613 ; echo
```

Switch back to the debug panel and notice that the code execution is paused at the breakpoint on `InventoryResource` class.



Click on the *Step Over* icon to execute one line and retrieve the inventory object for the given product id from the database.



The *Variables* panel allows you to see (and change) local variables.

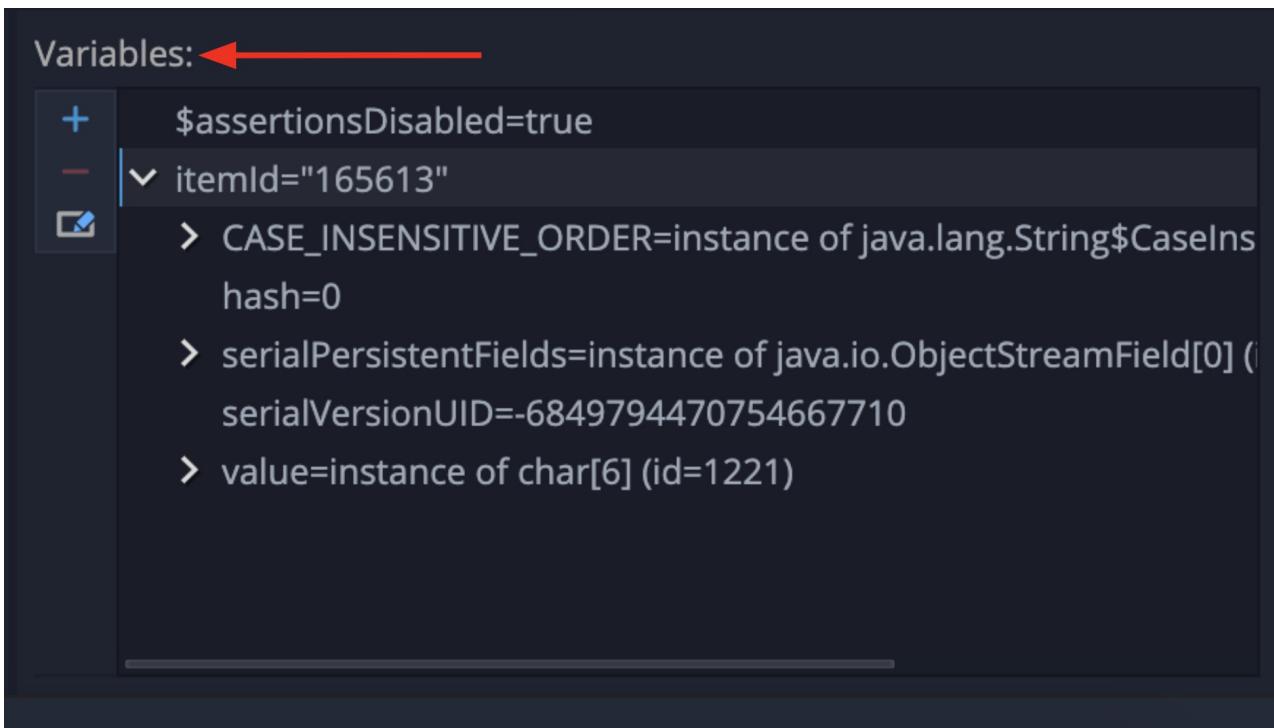
The screenshot shows the IntelliJ IDEA interface with the code editor displaying `InventoryResource.java`. The code defines a REST endpoint for getting inventory items. The Variables panel on the right shows local variables for the current frame, including `itemId` set to "165613". The Breakpoints panel at the bottom shows a single breakpoint at line 33 of `InventoryResource.java`.

```

15 import io.swagger.annotations.Api;
16 import io.swagger.annotations.ApiOperation;
17 import org.jboss.resteasy.annotations.jaxrs.PathParam;
18
19 @Path("/services/inventory")
20 @ApplicationScoped
21 @Produces("application/json")
22 @Consumes("application/json")
23 public class InventoryResource {
24
25     @GET
26     public List<Inventory> getAll() {
27         return Inventory.listAll();
28     }
29
30     @GET
31     @Path("{itemId}")
32     public List<Inventory> getAvailability(@PathParam String itemId) {
33         return Inventory.<Inventory>streamAll()
34             .filter(p -> p.itemId.equals(itemId))
35             .collect(Collectors.toList());
36     }
37
38     @Provider
39     public static class ErrorMapper implements ExceptionMapper<Exception> {
40
41         @Override
42         public Response toResponse(Exception exception) {
43             int code = 500;
44             if (exception instanceof WebApplicationException) {
45                 code = ((WebApplicationException) exception).getResponse().getStatus();
46             }
47             return Response.status(code);
48         }
49     }
50 }

```

This would allow you to see for example value of `itemId` variable passed into the method during execution.



Look at the **Variables** window. You can see the value of *itemId* along with the Stream variables used to retrieve the inventory.

The product id(165613) is a unique value to retrieve certain data from the Inventory database. If you might have unexpected result or errors in development, this debugging feature will help you find the root cause quickly then you will eventually fix the issue.

3. Resume Debug and Confirm the Result

Click on the *Resume* icon to continue the code execution and then on the stop icon to end the debug session. When you switch to *Terminal-2* window, you will see the result of **curl** command.

```
[{"id":3,"itemId":"165613","link":"http://maps.google.com/?q=Seoul","location":"Seoul","quantity":256}]
```

If you wait too long, the **curl** command may timeout and you won't get any output. You can repeat the process to see it again.

```
(jboss@workspacex8e2ycqnrdqidh projects)$ curl http://localhost:8080/services/inventory/165613 ; echo
[{"id":3,"itemId":"165613","link":"http://maps.google.com/?q=Seoul","location":"Seoul","quantity":256}]
```

NOTE : Make sure to stop Quarkus development mode via **Close** terminal. Next, you need to open a new Terminal in CodeReady Workspaces then change the directory once again via **cd** command that you executed previously.

Congratulations!

Application Monitoring

Lab3 - Application Monitoring

In the previous labs, you learned how to debug cloud-native apps to fix errors quickly using CodeReady Workspaces with Quarkus framework, and you got a glimpse into the power of Quarkus for developer joy.

You will now begin observing applications in term of a distributed transaction, performance and latency because as cloud-native applications are developed quickly, a distributed architecture in production gets ultimately complex in two areas: networking and observability. Later on we'll explore how you can better manage and monitor the application using service mesh.

In this lab, you will monitor coolstore applications using [Jaeger](#) and [Prometheus](#).



Jaeger is an open source distributed tracing tool for monitoring and troubleshooting microservices-based distributed systems, including:

- Distributed context propagation
- Distributed transaction monitoring
- Root cause analysis
- Service dependency analysis
- Performance and latency optimization

Prometheus is an open source systems monitoring and alerting tool that fits in recording any numeric time series, including:

- Multi-dimensional time series data by metric name and key/value pairs
- No reliance on distributed storage
- Time series collection over HTTP
- Pushing time series is supported via an intermediary gateway
- Service discovery or static configuration

1. Create OpenShift Project

In this step, we will deploy our new monitoring tools for our CoolStore application, so create a separate project to house it and keep it separate from our monolith and our other microservices we already created previously.

In the [OpenShift web console](#), create a new project for the *Monitoring* tools:

Click **Create Project**, fill in the fields, and click **Create**:

- Name: **userXX-monitoring**
- Display Name: **USERXX CoolStore App Monitoring Tools**
- Description: *leave this field empty*

Create Project

Name *

Display Name

Description



This will take you to the project overview. There's nothing there yet, but that's about to change.

2. Deploy Jaeger to OpenShift

This template uses an in-memory storage with a limited functionality for local testing and development. Do not use this template in production environments, although there are a number of parameters in the template to constrain the maximum number of traces and the amount of CPU/Memory consumed to prevent node instability.

Install everything in the current namespace via CodeReady Workspaces Terminal:

```
oc project userXX-monitoring
```

```
oc process -f /projects/cloud-native-workshop-v2m2-labs/monitoring/jaeger-all-in-one-template.yml | oc create -f -
```

You can also check if the deployment is complete via CodeReady Workspaces Terminal:

```
oc rollout status -w deployment.apps/jaeger
```

deployment jaeger successfully rolled out

When you navigate the **Project Status** page in OpenShift console, you will see as below:

The screenshot shows the OpenShift Container Platform interface. The left sidebar includes Home, Projects, Status, Search, Events, Catalog, Workloads, Networking, Storage, Builds (Build Configs, Builds, Image Streams), Monitoring, and Administration (Service Accounts, Roles, Role Bindings). The main area is titled 'Project Status' for 'user0-monitoring'. It displays a 'Resources' section with a 'jaeger' pod listed. The pod details show: NAME: jaeger, UPDATE STRATEGY: Recreate; NAMESPACE: user0-monitoring, PROGRESS DEADLINE: 59d6523h 14m 7s; LABELS: app=jaeger, jaeger-infra=template-all-in-one, templ...=jaeger-template-all-in...; POD SELECTOR: app=jaeger, jaeger-infra=jaeger-pod; NODE SELECTOR: No selector; TOLERATIONS: 0 Tolerations. The pod status is '1 available'.

3. Examine Jaeger-Collector

Collector is by default accessible only to services running inside the cluster. We will use *jaeger-collector-http* service with port 14268 to gather tracing data of Inventory service later.

Service Overview

NAME: jaeger-collector

NAMESPACE: user0-monitoring

LABELS: app=jaeger, jaeger-infra=template-all-in-one, template=jaeger-template-all-in-one

POD SELECTOR: jaeger-infra=jaeger-pod

ANNOTATIONS: 0 Annotations

SESSION AFFINITY: None

CREATED AT: Aug 5, 8:55 pm

Service Routing

Type	Location
Cluster IP	172.30.225.227 Accessible within the cluster only

Service Port Mapping

Name	Port	Protocol	Pod Port or Name
jaeger-collector-tchannel	S 14267	TCP	P 14267
jaeger-collector-http	S 14268	TCP	P 14268
jaeger-collector-zipkin	S 9411	TCP	P 9411

4. Observe Jaeger UI

Once you deployed Jaeger to OpenShift, navigate to *Networking > Routes* and you will see that the route was generated automatically.

NAME	NAMESPACE	LOCATION	SERVICE	STATUS
jaeger-collector	user0-monitoring	https://jaeger-collector-user0-monitoring.apps.cluster-seoul-a30e.seoul-a30e.openshiftworkshop.com:14267	jaeger-collector	Accepted
jaeger-query	user0-monitoring	https://jaeger-query-user0-monitoring.apps.cluster-seoul-a30e.seoul-a30e.openshiftworkshop.com:14268	jaeger-query	Accepted

Click on the route URL for **jaeger-query**. This is the UI for Jaeger, but currently we have no apps being monitored so it's rather useless. Don't worry! We will utilize tracing data in the next step.



The image shows the Jaeger UI search interface. On the left, there is a sidebar with various search filters: Service (0), Operation (0), Tags (http.status_code=200 error=true), Lookback (Last Hour), Min Duration (e.g. 1.2s, 100ms, 500us), Max Duration (e.g. 1.2s, 100ms, 500us), and Limit Results (set to 20). At the bottom of the sidebar is a 'Find Traces' button. To the right of the sidebar, there is a large search input field containing the text 'all'. At the top right of the page, there are links for 'Jaeger UI', 'Lookup by Trace ID...', 'Search', 'Compare', 'Dependencies', and 'About Jaeger'.

5. Utilizing OpenTracing with Inventory(Quarkus)

We have a catalog service written with Spring Boot that calls the inventory service written with Quarkus as part of our cloud-native application. These applications are easy to trace using Jaeger.

In this step, we will add Quarkus extensions to the Inventory application for using **smallrye-opentracing**. Copy the following commands to add the tracing extension via CodeReady Workspaces Terminal:

Go to *inventory* directory and add the extension with these commands:

```
cd /projects/cloud-native-workshop-v2m2-labs/inventory
```

```
mvn quarkus:add-extension -Dextensions="opentracing"
```

If builds successfully (you will see *BUILD SUCCESS*), you will see *smallrye-opentracing* dependency in **pom.xml**.

```

application.properties import.sql m inventory x InventoryResource +
2     <artifactId>quarkus-jdbc-postgresql</artifactId>
3   </dependency>
4   <dependency>
5     <groupId>io.quarkus</groupId>
6     <artifactId>quarkus-smallrye-health</artifactId>
7   </dependency>
8   <dependency>
9     <groupId>io.quarkus</groupId>
10    <artifactId>quarkus-junit5</artifactId>
11    <scope>test</scope>
12  </dependency>
13  <dependency>
14    <groupId>io.rest-assured</groupId>
15    <artifactId>rest-assured</artifactId>
16    <scope>test</scope>
17  </dependency>
18  <dependency>
19    <groupId>io.quarkus</groupId>
20    <artifactId>quarkus-smallrye-opentracing</artifactId>
21  </dependency>
22 </dependencies>
23 </project>
24 <plugins>
25   <plugin>
26     <artifactId>maven-compiler-plugin</artifactId>
27     <version>${compiler-plugin.version}</version>
28     <configuration>
29       <source>1.8</source>
30       <target>1.8</target>
31       <parameters>true</parameters>
32     </configuration>
33   </plugin>
34 </plugins>

```

[jboss@workspacea9dgc768g02trv6w inventory]\$ mvn quarkus:add-extension -Dextensions="io.quarkus:quarkus-smallrye-opentracing"
[INFO] Scanning for projects...
[INFO] [INFO] -----< com.redhat.coolstore:inventory >-----
[INFO] Building inventory 1.0-SNAPSHOT
[INFO] [INFO] -----[jar]-----
[INFO] [INFO] --- quarkus-maven-plugin:0.16.1:add-extension (default-cli) @ inventory ---
? Adding dependency io.quarkus:quarkus-smallrye-opentracing:jar
[INFO] [INFO] BUILD SUCCESS
[INFO] [INFO] -----
[INFO] Total time: 1.547 s
[INFO] Finished at: 2019-06-13T08:08:57Z
[INFO] -----
[jboss@workspacea9dgc768g02trv6w inventory]\$

NOTE: There are many [more extensions](#) for Quarkus for popular frameworks like [CodeReady Workspaces](#), [Vert.x](#), [Apache Camel](#), [Infinispan](#), Spring DI compatibility (e.g. `@Autowired`), and more.

6. Create the configuration

Before getting started with this step, confirm your **jaeger-collector** service in *userXX-monitoring* project via **oc** command in CodeReady Workspaces Terminal:

```
oc get svc -n userXX-monitoring | grep jaeger
```

```

jaeger-agent    ClusterIP    None      <none>
5775/UDP,6831/UDP,6832/UDP,5778/TCP  4d3h
jaeger-collector ClusterIP    172.30.225.227 <none>
14267/TCP,14268/TCP,9411/TCP        4d3h
jaeger-query    LoadBalancer  172.30.88.160 af514f3d7b77711e98f2c06fc57e3ee7-
2124798632.ap-southeast-1.elb.amazonaws.com 80:31616/TCP          4d3h

```

The easiest way to configure the **Jaeger tracer** is to set up in the application(i.e. inventory).

Open [src/main/resources/application.properties](#) file and add the following configuration via CodeReady Workspaces Terminal:

You need to replace **userXX** with your username in the configuration.

```
# Jaeger configuration
quarkus.jaeger.service-name=inventory
quarkus.jaeger.sampler-type=const
quarkus.jaeger.sampler-param=1
quarkus.jaeger.endpoint=http://jaeger-collector.userXX-monitoring:14268/api/traces
```

You can also specify the configuration using environment variables or JVM properties. See [Jaeger Features](#).

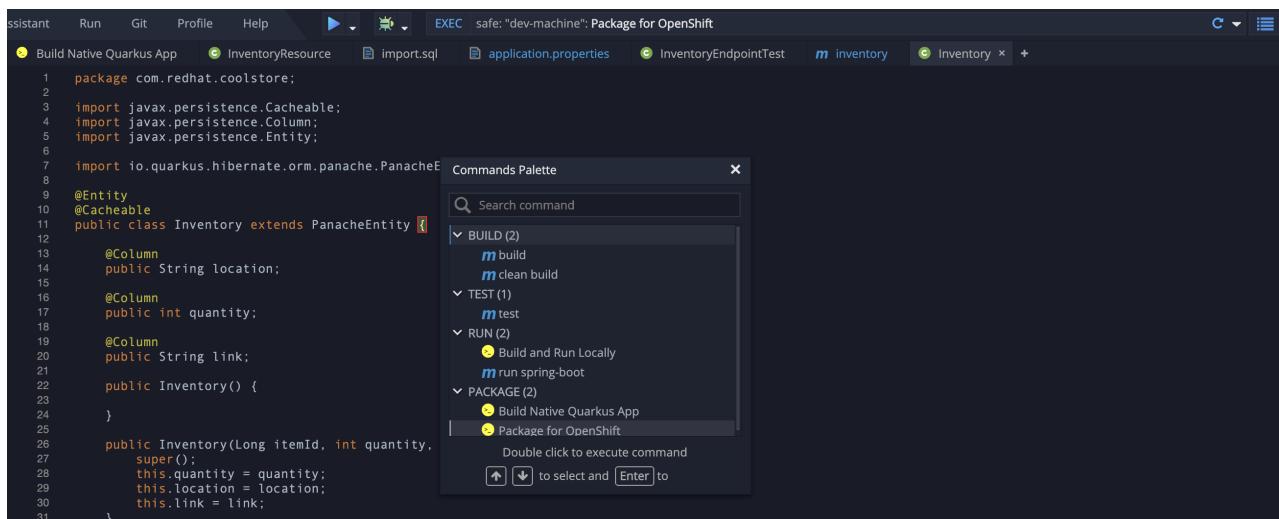
If the `quarkus.jaeger.service-name` property (or `JAAGER_SERVICE_NAME` environment variable) is not provided then a “no-op” tracer will be configured, resulting in no tracing data being reported to the backend.

Currently the tracer can only be configured to report spans directly to the collector via HTTP, using the `quarkus.jaeger.endpoint` property (or `JAAGER_ENDPOINT` environment variable). Support for using the Jaeger agent, via UDP, will be available in a future version.

NOTE: there is no tracing specific code included in the application. By default, requests sent to this endpoint will be traced without any code changes being required. It is also possible to enhance the tracing information. For more information on this, please see the [MicroProfile OpenTracing specification](#).

7. Re-Deploy to OpenShift

Repackage the inventory application via clicking on **Package for OpenShift** in *Commands Palette*:



Start and watch the build, which will take about a minute to complete:

```
oc start-build inventory-quarkus --from-file target/*-runner.jar --follow -n userXX-inventory
```

You should see a **Push successful** at the end of the build output and it. To verify that deployment is started and completed automatically, run the following command via CodeReady Workspaces Terminal :

```
oc rollout status -w dc/inventory-quarkus -n userXX-inventory
```

And wait for the result as below:

```
replication controller "inventory-quarkus-XX" successfully rolled out
```

8. Observing Jaeger Tracing

In order to trace networking and data transaction, we will call the Inventory service via **curl** commands via CodeReady Workspaces Terminal: Be sure to use your route URL of Inventory.

```
curl http://$(oc get route inventory-quarkus -o=go-template --template='{{ .spec.host }}')/services/inventory/165613 ; echo
```

Go to *Workloads > Pods* in the left menu and click on **inventory-quarkus-xxxxxx**.

NAME	NAMESPACE	POD LABELS	NODE	STATUS	READINESS
inventory-database-1-4rrm5	user0-inventory	app=inventory-database deploy...=inventory-dat... deployme...=inventory-...	ip-10-0-133-104.ap-southeast-1.compute.internal	Running	Ready
inventory-quarkus-4-kc4t6	user0-inventory	app=inventory-quarkus deploy...=inventory-qua... deployme...=inventory-...	ip-10-0-152-139.ap-southeast-1.compute.internal	Running	Ready

Click on **Logs** tab and you will see that tracer is initialized after you call the Inventory service at the first time.

```
2019-08-05 12:12:17,574 INFO [io.jae.Configuration] (executor-thread-1) Initialized tracer=JaegerTracer(version=Java-0.34.0, serviceName=inventory, reporter=RemoteReporter(sender=HttpSender(), closeEnqueueTimeout=1000), sampler=ConstSampler(decision=true, tags={sampler.type=const, sampler.param=true}), tags={hostname=inventory-quarkus-4-kc4t6, jaeger.version=Java-0.34.0, ip=10.131.8.48}, zipkinSharedRpcSpan=false, expandExceptionLogs=false, useTraceld128Bit=false)
```

inventory-quarkus-4 > Pod Details

inventory-quarkus-4-kc4t6

Actions ▾

Overview YAML Environment **Logs** Events Terminal

Log streaming... inventory-quarkus ▾

7 lines

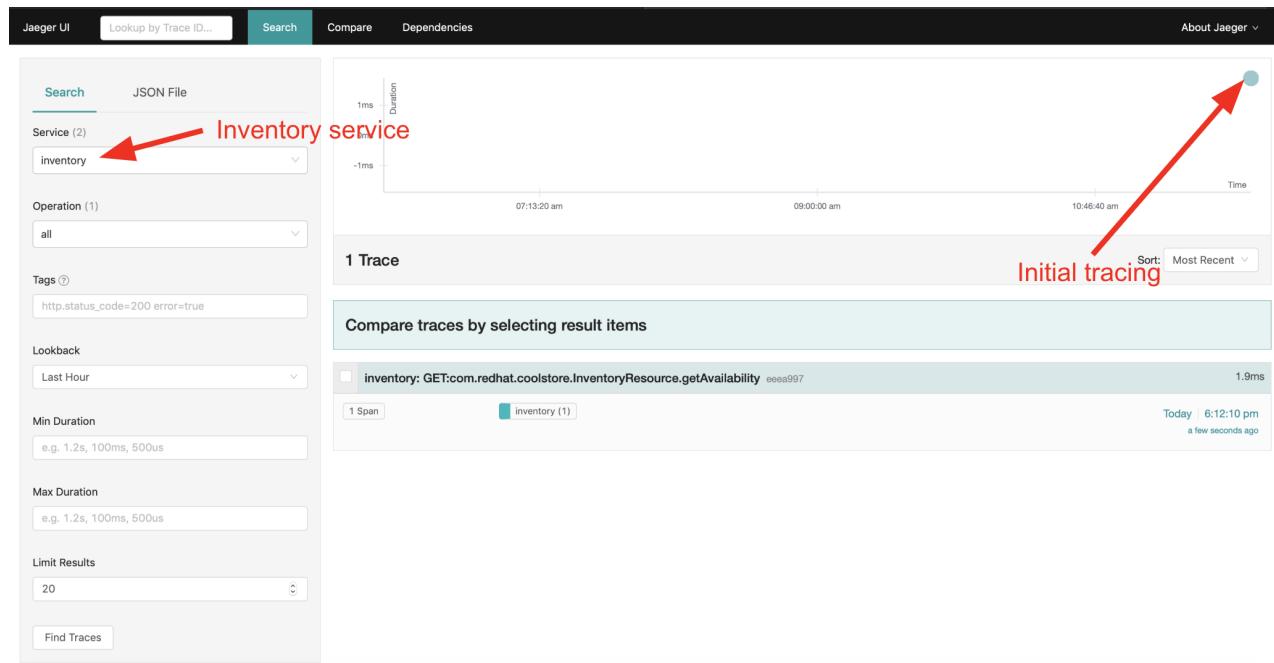
```
Starting the Java application using /opt/run-java/run-java.sh ...
exec java -javaagent:/opt/jolokia/jolokia.jar=config=/opt/jolokia/etc/jolokia.properties -Xms192m -Xmx768m -XX:+UseParallelOldGC -XX:+UnlockExperimentalVMOpt
Is No access restrictor found, access to any MBean is allowed
Jolokia: Agent started with URL https://10.131.8.48:8778/jolokia/
2019-08-05 12:11:06,870 INFO [io.quarkus] (main) Quarkus 0.20.0 started in 8.702s. Listening on: http://[::]:8080
2019-08-05 12:11:06,872 INFO [io.quarkus] (main) Installed features: [agroal cdi hibernate-orm jaeger idb=h2 idb-postgresql narayana-itx resteasy]
2019-08-05 12:12:17,574 INFO [io.jae.Configuration] (executor-thread-1) Initialized tracer=JaegerTracer(version=Java-0.34.0, serviceName=inventory, reporter=)
```

[Download](#) | [Expand](#)

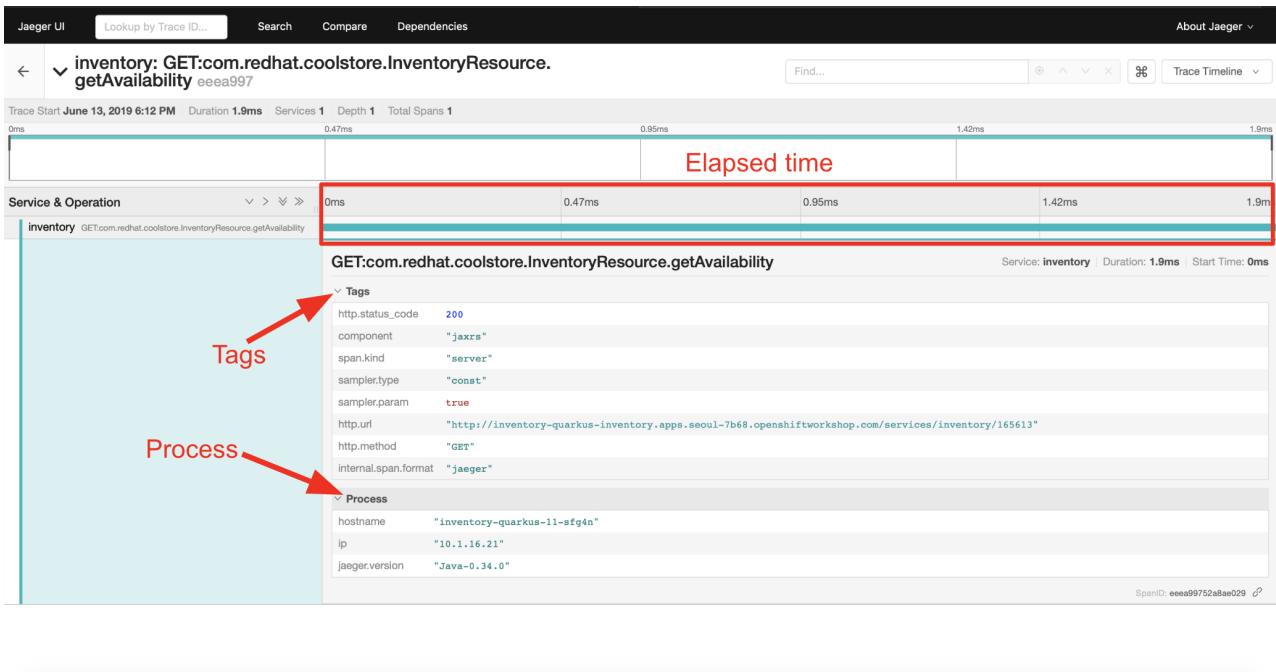
Now, reload the Jaeger UI then you will find that 2 services are created as here:

- inventory
- jaeger-query

Select the **inventory** service and then click on **Find Traces** and observe the first trace in the graph:



If you click on **Span** and you will see a logical unit of work in Jaeger that has an operation name, the start time of the operation, and the duration. Spans may be nested and ordered to model causal relationships:



Let's make more traces! Open a new web browser to access *CoolStore Inventory Page* using its route (on *Networking > Routes* in the OpenShift Console):

If you do not see the `inventory` route listed, be sure you've chosen the `userXX-inventory` project in the Project selector dropdown!

CoolStore Inventory

This shows the latest CoolStore Inventory from the Inventory microservice using Quarkus.

[Fetch Inventory](#)

The CoolStore Inventory

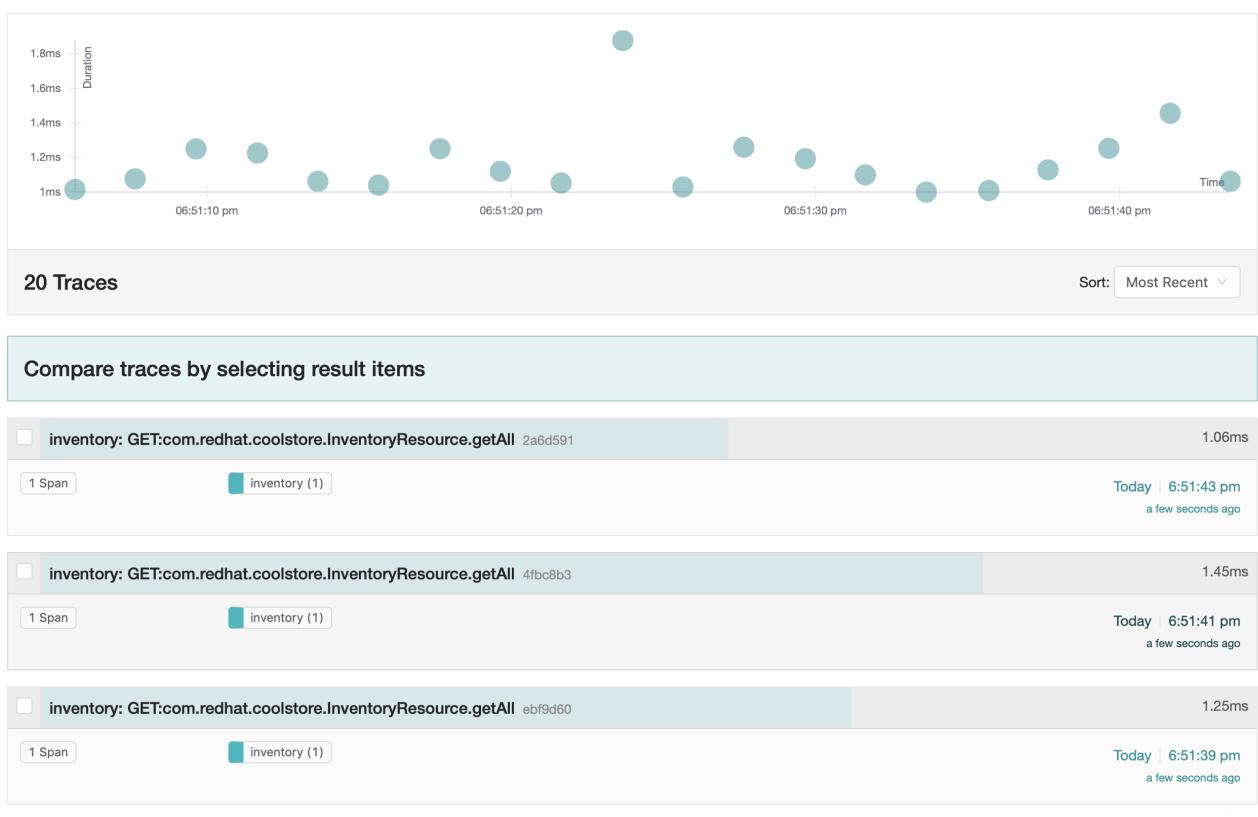
Status: **OK** (Last Successful Fetch: moments ago)

Item ID	Quantity	Location
329299	736	Raleigh
329199	512	Boston
165613	256	Seoul
165614	54	Singapore
165954	87	London
444434	443	NewYork
444435	600	Paris
444437	230	Tokyo

[Fetch Inventory](#)

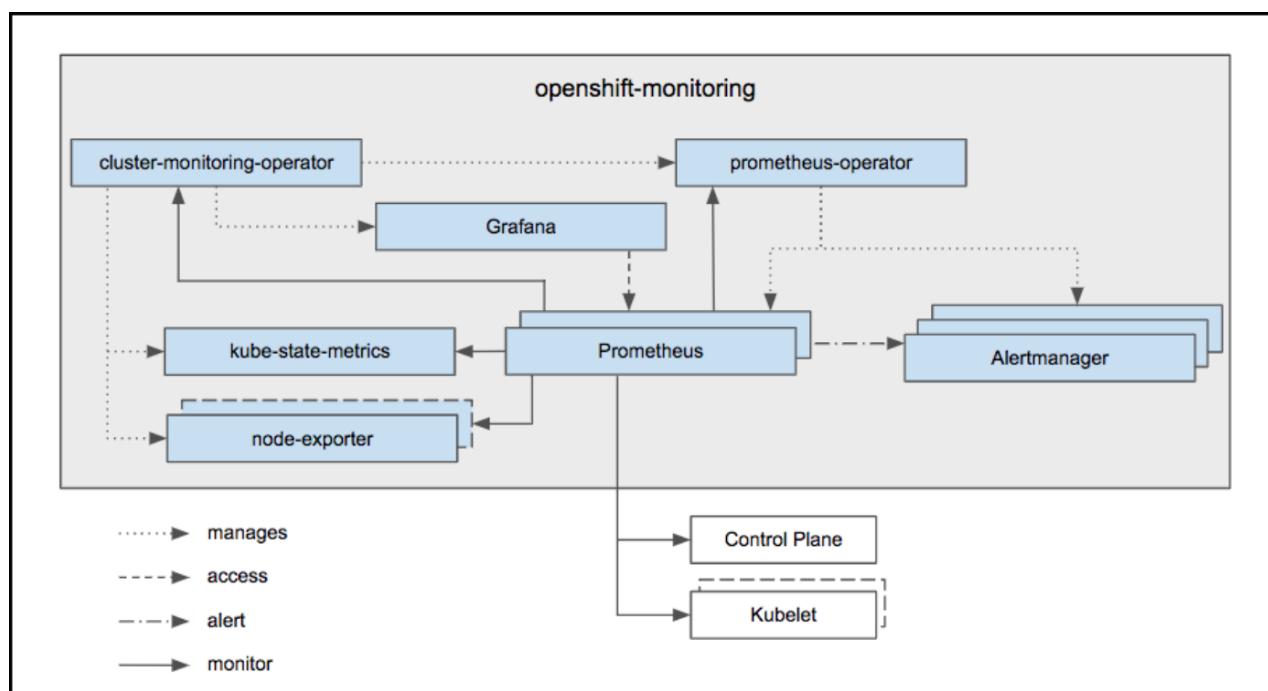
© Red Hat 2019

Go back to *Jaeger UI* then click on *Find Traces*. You will see dozens of traces because the Inventory page continues to calling the endpoint of Inventory service in every 2 seconds like we called via `curl` command:



9. Deploy Prometheus and Grafana to OpenShift

OpenShift Container Platform ships with a pre-configured and self-updating monitoring stack that is based on the [Prometheus](#) open source project and its wider eco-system. It provides monitoring of cluster components and ships with a set of alerts to immediately notify the cluster administrator about any occurring problems and a set of [Grafana](#) dashboards.



However, we will deploy custom **Prometheus** to scrape services metrics of Inventory

and Catalog applications. Then we will visualize the metrics data via custom **Grafana** dashboards deployment.

Go to *Project Status* page in *userXX-monitoring* project and click on **Deploy Image** under *Add* on the right top menu:

The screenshot shows the Red Hat OpenShift Container Platform interface. The top navigation bar includes the Red Hat logo, 'OpenShift Container Platform', and a user dropdown for 'user0'. The left sidebar has links for Home, Projects (which is selected and highlighted with a red arrow), Status, Search, Events, Catalog, Workloads, Networking, and Storage. The main content area is titled 'Project Status' for the 'user0-monitoring' project. It shows a single pod named 'jaeger'. In the top right, there's a 'Add' dropdown menu with options: 'Browse Catalog', 'Deploy Image' (which is highlighted with a red arrow), and 'Import YAML'. Below the menu, there are buttons for 'Group by: Application' and 'Filter'.

Select **Image Name** and input *prom/prometheus* to search the Prometheus container image via clicking on *Search icon*.

Deploy Image

This screenshot shows the 'Deploy Image' configuration page. The 'Namespace' field is set to 'user0-monitoring'. The 'Image Name' field contains 'prom/prometheus'. Below the form, there's a detailed description of the image: 'prom/prometheus' (Jul 11, 0:36 am, 48.71 MiB, 9 layers). It lists several bullet points about the image, such as tracking via Image Stream, deployment in Deployment Config, port 9090 load balancing, and access via hostname. A note says the image declares volumes and uses non-persistent host-local storage. The 'Name' field is set to 'prometheus'. The 'Environment Variables' section shows a table with one entry: 'name' with 'value'. There are 'Add Value' and 'Delete' buttons for the table. At the bottom are 'Deploy' and 'Cancel' buttons.

Once you find the image correctly as the above page, click on **Deploy**. It takes 1 ~ 2 mins to deploy a pod.

The screenshot shows the Red Hat OpenShift Container Platform interface. On the left, the 'Project Status' section displays two pods: 'jaeger' and 'prometheus'. The 'prometheus' pod is highlighted with a blue background. On the right, the 'DC prometheus' details page is shown, providing an overview of the pod's status, desired count (1 pod), up-to-date count (1 pod), and matching pods (1 available, 0 unavailable). The pod itself is listed with its name, namespace (user0-monitoring), labels (app=prometheus), pod selector (Q app=prometheus), node selector (No selector), tolerations (0 Tolerations), latest version (1), reason (config change), update strategy (Rolling), timeout (600 seconds), update period (1 second), and interval (1 second).

Create the route to access Prometheus web UI. Navigate *Networking > Routes* on the left menu and click on **Create Route**.

The screenshot shows the 'Routes' page under the 'Networking' section. A red arrow points to the 'Create Route' button at the top left of the table. The table lists two existing routes: 'jaeger-collector' and 'jaeger-query', both of which are accepted and have their status set to 'Accepted'. The table includes columns for NAME, LOCATION, SERVICE, and STATUS.

NAME	LOCATION	SERVICE	STATUS
jaeger-collector	https://jaeger-collector-user0-monitoring.apps.cluster-seoul-a30e.seoul-a30e.openshiftworkshop.com:9090	jaeger-collector	Accepted
jaeger-query	https://jaeger-query-user0-monitoring.apps.cluster-seoul-a30e.seoul-a30e.openshiftworkshop.com:9090	jaeger-query	Accepted

Input the following variables and keep the rest of all default variables. Click on **Create**.

- Name: **prometheus**
- Service: **prometheus**
- Target Port: **9090 -> 9090 (TCP)**

Create Route

[Edit YAML](#)

Routing is a way to make your application publicly visible.

Name *

A unique name for the route within the project.

Hostname

Public hostname for the route. If not specified, a hostname is generated.

The hostname cannot be changed after the route is created.

Path

Path that the router watches to route traffic to the service.

Service *

Service to route to.

Target Port *

Target port for traffic.

Security

Secure route

Routes can be secured using several TLS termination types for serving certificates.



Now, you have the route URL(i.e. <http://prometheus-user0-monitoring.apps.cluster-seoul-a30e.seoul-a30e.openshiftworkshop.com>) as below and click on the URL to make sure if you can access the *Prometheus web UI*.

Project: user0-monitoring 

 prometheus 

[Overview](#) [YAML](#)

Route Overview

NAME prometheus	LOCATION http://prometheus-user0-monitoring.apps.cluster-seoul-a30e.seoul-a30e.openshiftworkshop.com 
NAMESPACE  user0-monitoring	STATUS  Accepted
LABELS 	HOSTNAME prometheus-user0-monitoring.apps.cluster-seoul-a30e.seoul-a30e.openshiftworkshop.com
ANNOTATIONS  1 Annotation 	PATH -
SERVICE  prometheus	
TARGET PORT 9090-tcp	
CREATED AT  less than a minute ago	



You will see the landing page of Prometheus as shown:

Prometheus Alerts Graph Status Help

Enable query history

Expression (press Shift+Enter for newlines)

Execute - insert metric at cursor

Graph Console

Element	Value
no data	

Add Graph

Let's deploy **Grafana Dashboards** to OpenShift. Go to *Project Status* page in *userXX-monitoring* project and click on **Deploy Image** under *Add* menu:

Red Hat OpenShift Container Platform

Home

Projects

Status → **Project: user0-monitoring**

Search

Events

Catalog

Workloads

Networking

Storage

Project Status

Resources Dashboard

jaeger

jaeger, #1

prometheus

prometheus, #1

36.0 MiB 0.000 cores 1 of 1 pods

31.3 MiB 0.001 cores 1 of 1 pods

Add

Browse Catalog

Deploy Image

Import YAML

Select **Image Name** and input *grafana/grafana* to search the Prometheus container image via clicking on *Search* icon.

Deploy Image

Namespace *

PR user0-monitoring

Deploy an existing image from an image registry.

Image Name *

grafana/grafana

To deploy an image from a private repository, you must [create an image pull secret](#) with your image registry credentials.

 **grafana/grafana** Jun 26, 3:27 am, 88.24 MiB, 6 layers

- Image Stream **grafana:latest** will track this image.
- This image will be deployed in Deployment Config **grafana**.
- Port 3000/TCP will be load balanced by Service **grafana**.

Other containers can access this service through the hostname **grafana**.

Name *

grafana

Identifies the resources created for this image.

Environment Variables ⓘ

NAME	VALUE
name	value

+ Add Value

Deploy Cancel

Once you find the image correctly as the above screenshot, click on **Deploy**. It takes 1 ~ 2 mins to deploy a pod.

The screenshot shows the 'Project Status' section of the OpenShift web interface. It lists several projects: 'grafana', 'jaeger', and 'prometheus'. Under 'grafana', a single pod named 'grafana, #1' is shown with a status of '1 of 1 pods'. To the right, a detailed view of the 'grafana' deployment is provided. The deployment has a 'Desired Count' of 1 pod, which is 'Up-to-date'. It has 1 matching pod available. The deployment configuration includes: NAME: grafana, LATEST VERSION: 1; NAMESPACE: user0-monitoring, REASON: config change; LABELS: app=grafana; POD SELECTOR: Q app=grafana; NODE SELECTOR: No selector; TOLERATIONS: 0 Tolerations; ANNOTATIONS: 1 Annotation. The update strategy is Rolling, with a TIMEOUT of 600 seconds and an UPDATE PERIOD of 1 second. The interval for annotations is 1 second, and the max unavailable is 25% of 1 pod.

Create the route to access Grafana web UI. Navigate *Networking > Routes* on the left menu and click on **Create Route**.

The screenshot shows the 'Routes' section of the Red Hat OpenShift Container Platform interface. On the left sidebar, under 'Networking', the 'Routes' option is highlighted with a red arrow pointing to it. A blue button labeled 'Create Route' is visible on the main screen. The table lists three existing routes: 'jeager-collector', 'jeager-query', and 'prometheus', all of which are accepted and have their status set to 'Accepted'. The table columns include: NAME, NAMESPACE, LOCATION, SERVICE, and STATUS.

NAME	NAMESPACE	LOCATION	SERVICE	STATUS
jeager-collector	user0-monitoring	https://jeager-collector-user0-monitoring.apps.cluster-seoul-a30e.seoul-a30e.openshiftworkshop.com:8443	jeager-collector	Accepted
jeager-query	user0-monitoring	https://jeager-query-user0-monitoring.apps.cluster-seoul-a30e.seoul-a30e.openshiftworkshop.com:8443	jeager-query	Accepted
prometheus	user0-monitoring	http://prometheus-user0-monitoring.apps.cluster-seoul-a30e.seoul-a30e.openshiftworkshop.com:9090	prometheus	Accepted

Input the following variables and keep the rest of all default variables. Click on **Create** .

- Name: **grafana**
- Service: **grafana**
- Target Port: **3000 -> 3000 (TCP)**

Create Route

[Edit YAML](#)

Routing is a way to make your application publicly visible.

Name * 

A unique name for the route within the project.

Hostname

Public hostname for the route. If not specified, a hostname is generated.

The hostname cannot be changed after the route is created.

Path

Path that the router watches to route traffic to the service.

Service * 

Service to route to.

Target Port * 

Target port for traffic.

Security Secure routes

Routes can be secured using several TLS termination types for serving certificates.

 **Create**  Cancel

Now, you have the route URL(i.e. <http://grafana-user0-monitoring.apps.cluster-seoul-a30e.seoul-a30e.openshiftworkshop.com>) and click on the URL to make sure if you can access the Grafana web UI.

RT grafana

Actions ▾

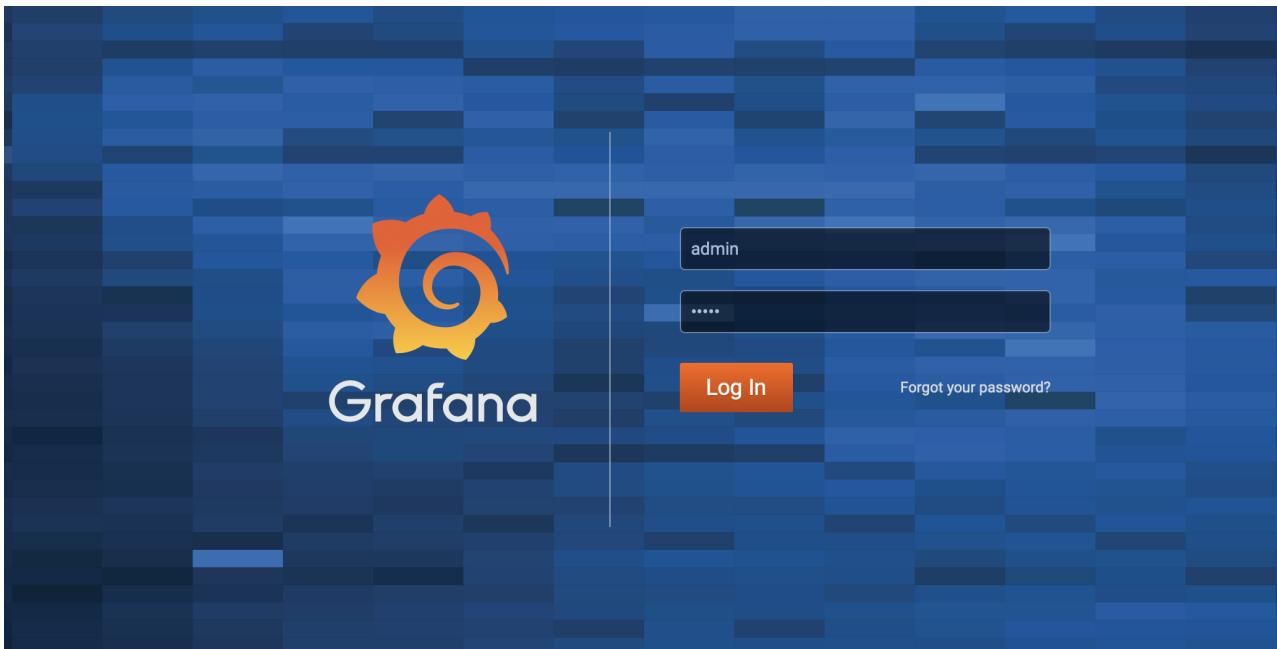
Overview	YAML
--------------------------	----------------------

Route Overview

NAME grafana	LOCATION http://grafana-user0-monitoring.apps.cluster-seoul-a30e.seoul-a30e.openshiftworkshop.com 
NAMESPACE  user0-monitoring	STATUS  Accepted
LABELS  app=grafana	HOSTNAME grafana-user0-monitoring.apps.cluster-seoul-a30e.seoul-a30e.openshiftworkshop.com
ANNOTATIONS  1 Annotation 	PATH -
SERVICE  grafana	
TARGET PORT 3000-tcp	
CREATED AT  less than a minute ago	



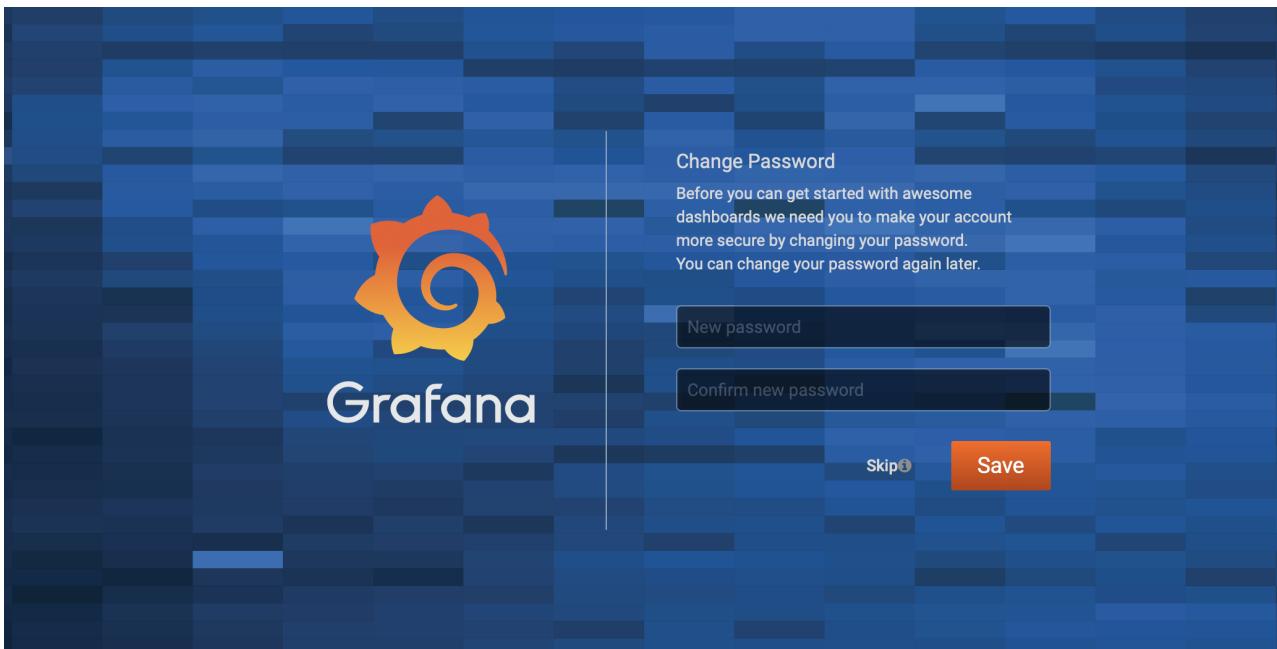
You will see the landing page of Prometheus as shown:



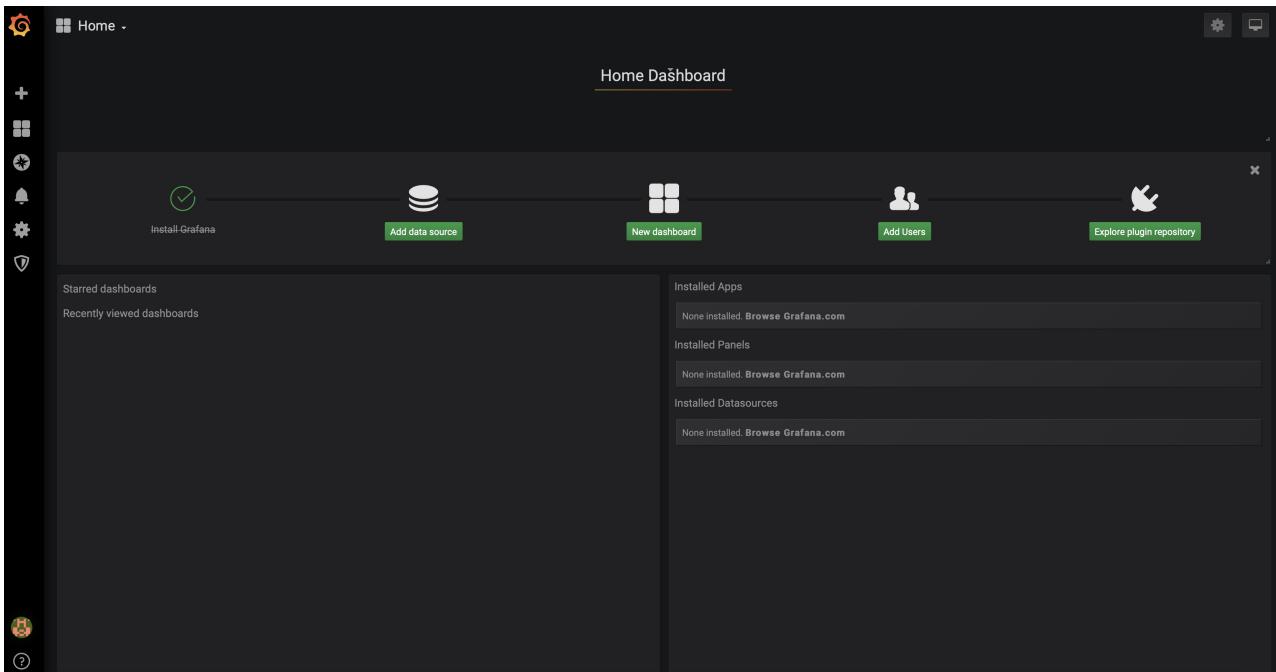
Log in Grafana web UI using the following variables:

- Username: admin
- Password: admin

Skip the Change Password.

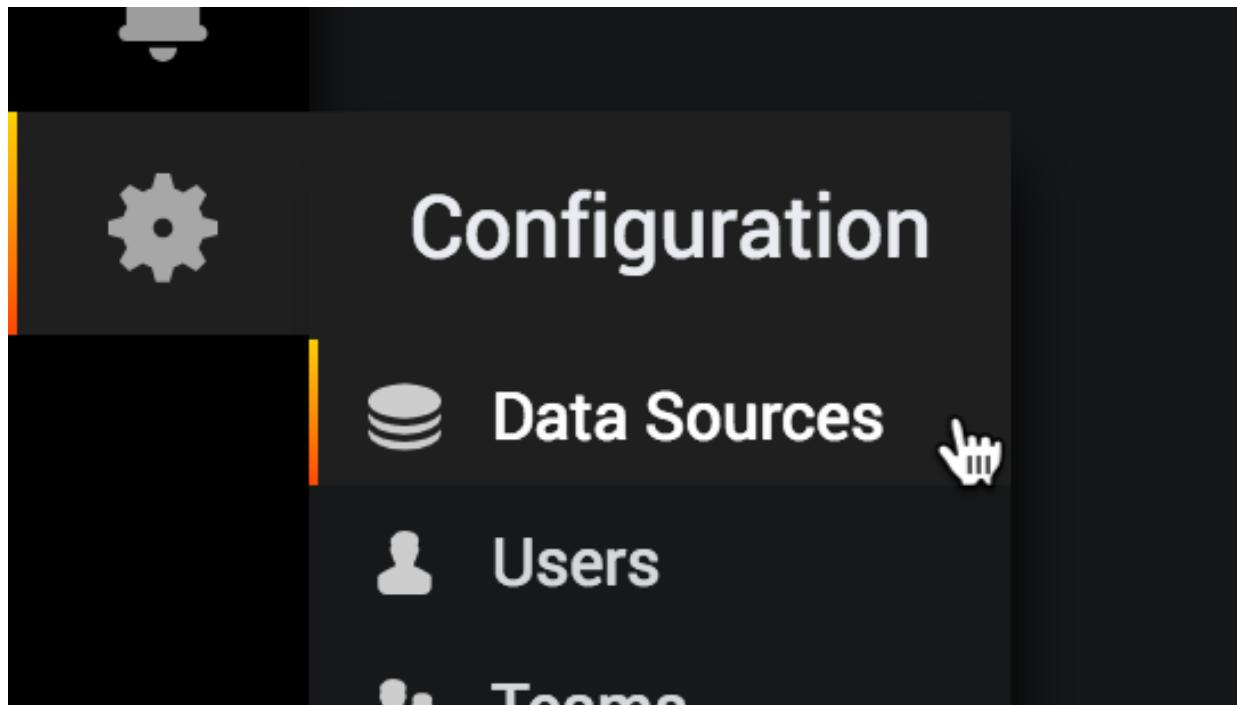


You will see the landing page of Grafana as shown:



10. Add a data source to Grafana

Before we create a monitoring dashboard, we need to add a data source. Go to the cog on the side menu that will show you the configuration menu. If the side menu is not visible click the Grafana icon in the upper left corner.



Click on data sources of the configuration menu and you'll be taken to the data sources page where you can add and edit data sources.

The screenshot shows the Grafana interface for managing data sources. At the top, there are navigation links: 'Data Sources' (highlighted in orange), 'Users', 'Teams', 'Plugins', 'Preferences', and 'API Keys'. Below this, a message states 'There are no data sources defined yet'. A prominent green button with the text 'Add data source' and a gear icon is centered. A small note at the bottom says 'ProTip: You can also define data sources through configuration files. [Learn more](#)'.

Click Add data source and select **Prometheus** as data source type.

The screenshot shows the 'Configuration' page in Grafana. It features a 'Choose data source type' section with a search bar labeled 'Filter by name or type'. Below the search bar is a grid of nine data source types, each with an icon and a name. The 'Prometheus' option is highlighted with a red box. The other options are: Azure Monitor, CloudWatch, Elasticsearch, Graphite, InfluxDB, Loki, Microsoft SQL Server, MySQL, OpenTSDB, PostgreSQL, Stackdriver, and TestData DB.

Choose data source type		
Azure Monitor	CloudWatch	Elasticsearch
Graphite	InfluxDB	Loki
Microsoft SQL Server	MySQL	OpenTSDB
PostgreSQL	Prometheus	Stackdriver
TestData DB		

Next, input the following variables to configure the dashboard. Make sure to replace the default HTTP URL with your *Prometheus Route URL* which should be <http://prometheus.userXX-monitoring:9090> (replace `userXX with your username).

Click on **Save & Test** then you will see the *Data source is working* message.

- Name: CloudNativeApps
- HTTP URL: <http://prometheus.userXX-monitoring:9090>

The screenshot shows the Granana interface for managing data sources. A specific data source named "CloudNativeApps" is selected, which is of type "Prometheus". The configuration includes fields for the URL (set to "http://prometheus-user1-monitoring..."), access method (set to "Server (Default)"), and various authentication and scraping options. A green status bar at the bottom confirms that the data source is working.

Now Granana is set up to pull collected metrics from Prometheus as they are collected from the application(s) you are monitoring. We'll use this later.

11. Utilize metrics specification for Inventory(Quarkus)

In this step, we will learn how *Inventory(Quarkus)* application can utilize the MicroProfile Metrics specification through the **SmallRye Metrics extension**. *MicroProfile Metrics* allows applications to gather various metrics and statistics that provide insights into what is happening inside the application.

The metrics can be read remotely using JSON format or the **OpenMetrics** format, so that they can be processed by additional tools such as *Prometheus*, and stored for analysis and visualisation.

We will add Quarkus extensions to the Inventory application for using *smallrye-metrics* and we'll use the Quarkus Maven Plugin. Copy the following commands to add the *smallrye-metricsexception* via CodeReady Workspaces Terminal:

Go to *inventory`* directory:

```
mvn quarkus:add-extension -Dextensions="metrics"
```

If builds successfully (you will see *BUILD SUCCESS*), you will see *smallrye-opentracing* dependency in *pom.xml* automatically.

```
application.properties
m inventory + InventoryResource
53 <dependency>
54   <groupId>io.rest-assured</groupId>
55   <artifactId>rest-assured</artifactId>
56   <scope>test</scope>
57 </dependency>
58 <dependency>
59   <groupId>io.quarkus</groupId>
60   <artifactId>quarkus-undertow-websockets</artifactId>
61 </dependency>
62 <dependency>
63   <groupId>io.quarkus</groupId>
64   <artifactId>quarkus-smallrye-opentracing</artifactId>
65 </dependency>
66 <dependency>
67   <groupId>io.quarkus</groupId>
68   <artifactId>quarkus-smallrye-metrics</artifactId>
69 </dependency>
70 </dependencies>
71 <build>
72   <plugins>
73     <plugin>
74       <artifactId>maven-compiler-plugin</artifactId>
75       <version>${compiler-plugin.version}</version>
76       <configuration>
77         <source>1.8</source>
78         <target>1.8</target>
79         <parameters>true</parameters>
80       </configuration>
81     </plugin>
82     <plugin>
83       <artifactId>maven-surefire-plugin</artifactId>
84       <version>${surefire-plugin.version}</version>
85     </plugin>
86   </plugins>
87 </build>
88 <!--
89   <dependency>
90     <groupId>io.smallrye.metrics</groupId>
91     <artifactId>smallrye-metrics</artifactId>
92     <version>0.16.1</version>
93   </dependency>
94 -->
95 <!--
96   <dependency>
97     <groupId>io.smallrye.opentracing</groupId>
98     <artifactId>smallrye-opentracing</artifactId>
99     <version>0.16.1</version>
100   </dependency>
101 -->
```

Machines dev-machine Terminal +

```
[jboss@workspacea9dgc768g02trv6w inventory]$ mvn quarkus:add-extension -Dextensions="io.quarkus:quarkus-smallrye-metrics"
[INFO] Scanning for projects...
[INFO] -----
[INFO] -----< com.redhat.coolstore:inventory >-----
[INFO] Building inventory 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- quarkus-maven-plugin:0.16.1:add-extension (default-cli) @ inventory ---
[INFO] ? Adding dependency io.quarkus:quarkus-smallrye-metrics:jar
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] -----
[INFO] Total time: 1.404 s
[INFO] Finished at: 2019-06-19T07:51:51Z
[INFO] -----
```

Let's add a few annotations to make sure that our desired metrics are calculated over time and can be exported for processing by *Prometheus* and *Grafana*.

The metrics that we will gather are these:

- `performedChecksAll` : A counter of how many times `getAll()` has been performed.
- `checksTimerAll` : A measure of how long it takes to perform the `getAll()` method
- `performedChecksAvail` : A counter of how many times `getAvailability()` is called
- `checksTimerAvail` : A measure of how long it takes to perform the `getAvailability()` method

Open **InventoryResource** file in `src/main/java/com/redhat/coolstore` and replace the two methods `getAll()` and `getAvailability()` with the below code which adds several annotations for custom metrics (`@Counted`, `@Timed_`):

```

    @GET
    @Counted(name = "performedChecksAll", description = "How many getAll() have been
performed.")
    @Timed(name = "checksTimerAll", description = "A measure of how long it takes to perform the
getAll().", unit = MetricUnits.MILLISECONDS)
    public List<Inventory> getAll() {
        return Inventory.listAll();
    }

    @GET
    @Counted(name = "performedChecksAvail", description = "How many getAvailability() have been
performed.")
    @Timed(name = "checksTimerAvail", description = "A measure of how long it takes to perform the
getAvailability().", unit = MetricUnits.MILLISECONDS)
    @Path("{itemId}")
    public List<Inventory> getAvailability(@PathParam String itemId) {
        return Inventory.<Inventory>streamAll()
            .filter(p -> p.itemId.equals(itemId))
            .collect(Collectors.toList());
    }
}

```

Add the necessary imports at the top:

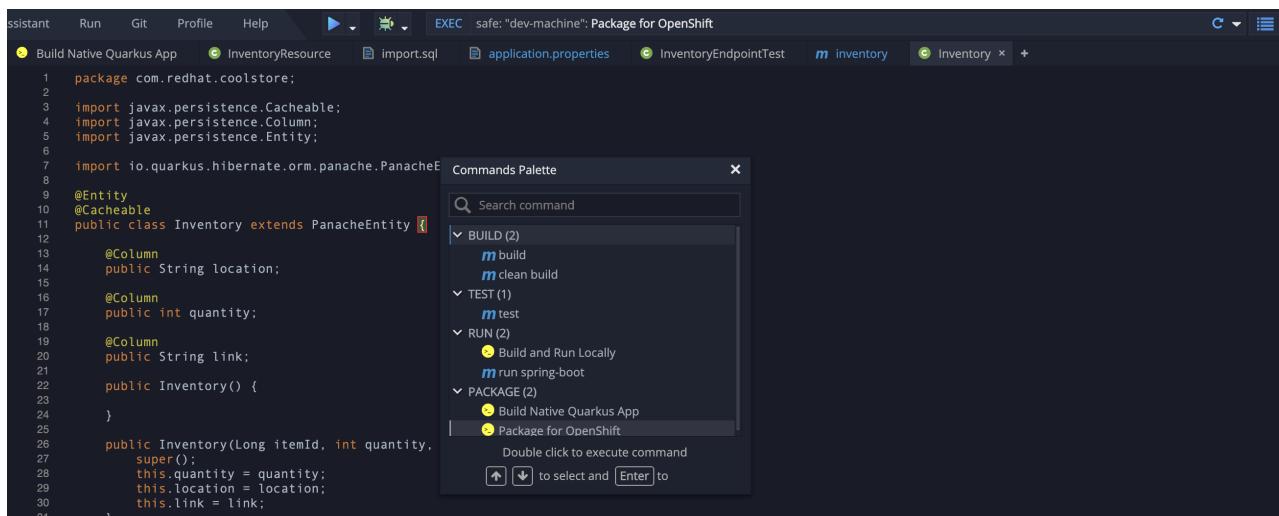
```

import org.eclipse.microprofile.metrics.MetricUnits;
import org.eclipse.microprofile.metrics.annotation.Counted;
import org.eclipse.microprofile.metrics.annotation.Timed;

```

12. Redeploy to OpenShift

Repackage the inventory application via clicking on **Package for OpenShift** in *Commands Palette*:



Or you can run a maven plugin command directly in Terminal:

```
mvn clean package -DskipTests (make sure you're in the inventory directory)
```

Start and watch the build, which will take about a minute to complete:

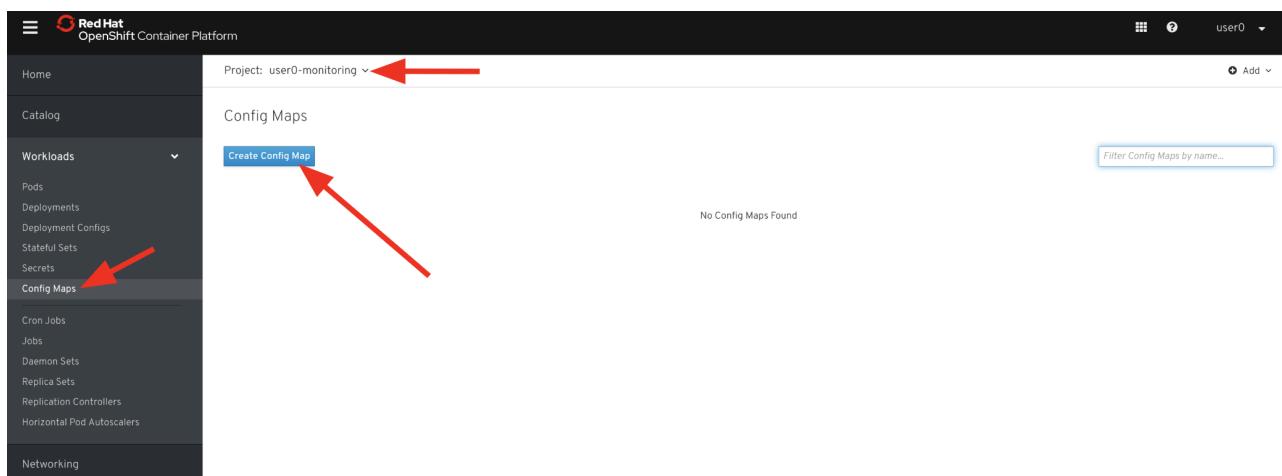
```
oc start-build inventory-quarkus --from-file target/*-runner.jar --follow -n userxx-inventory
```

Finally, make sure it's actually done rolling out:

```
oc rollout status -w dc/inventory-quarkus -n userxx-inventory
```

Go to the *userXX-monitoring* project in [OpenShift web console](#) and then on the left sidebar, *Workloads > Config Maps*.

Now we will reconfigure Prometheus so that it knows about our application.



Make sure you're in the `userXX-monitoring` project in OpenShift, and click on **Create Config Maps** button to create a config map. You'll copy and paste the below code into the field.

In the below `ConfigMap` code, you need to replace `userXX-monitoring` with your username prefix (e.g. `user9-monitoring`), and replace `YOUR_PROMETHEUS_ROUTE` and `YOUR_INVENTORY_ROUTE` with values from your environment, so that Prometheus knows where to scrape metrics from. The values you need can be discovered by running the following commands in the Terminal:

- Prometheus Route: `oc get route prometheus -n userxx-monitoring -o=go-template --template='{{ .spec.host }}'`
- Inventory Route: `oc get route inventory-quarkus -n userxx-inventory -o=go-template --template='{{ .spec.host }}'`

Paste in this code and then replace the values as shown in the image below:

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
  namespace: userXX-monitoring
data:
  prometheus.yml: >-
    # my global config

    global:
      scrape_interval: 15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.
      evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.
      # scrape_timeout is set to the global default (10s).

    # Alertmanager configuration

    alerting:
      alertmanagers:
        - static_configs:
          - targets:
            # - alertmanager:9093

    # Load rules once and periodically evaluate them according to the global
    # 'evaluation_interval'.

    rule_files:
      # - "first_rules.yml"
      # - "second_rules.yml"

    # A scrape configuration containing exactly one endpoint to scrape:

    # Here it's Prometheus itself.

    scrape_configs:
      # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
      - job_name: 'prometheus'

        # metrics_path defaults to '/metrics'
        # scheme defaults to 'http'.

      static_configs:
        - targets: ['YOUR_PROMETHEUS_ROUTE_URL']

      - job_name: 'quarkus'
        metrics_path: '/metrics/application'

      static_configs:
        - targets: ['YOUR_INVENTORY_ROUTE_URL']

```

Create Config Map

Create by manually entering YAML or JSON definitions, or by dragging and dropping a file into the editor.

```

1 apiVersion: v1
2 kind: ConfigMap
3 metadata:
4   name: prometheus-config
5   namespace: userXX-monitoring ←
6 data:
7   prometheus.yml: >-
8     # my global config
9
10  global:
11    scrape_interval: 15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.
12    evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.
13    # scrape_timeout is set to the global default (10s).
14
15  # Alertmanager configuration
16
17  alerting:
18    alertmanagers:
19      - static_configs:
20        - targets:
21          - # - alertmanager:9093
22
23  # Load rules once and periodically evaluate them according to the global
24  'evaluation_interval'.
25
26  rule_files:
27    - "first_rules.yml"
28    - "second_rules.yml"
29
30  # A scrape configuration containing exactly one endpoint to scrape:
31
32  # Here it's Prometheus itself.
33
34  scrape_configs:
35    # The job name is added as a label 'job=<job_name>' to any timeseries scraped from this config.
36    - job_name: 'prometheus'
37
38    # metrics_path defaults to '/metrics'
39    # scheme defaults to 'http'.
40
41    static_configs:
42      - targets: ['YOUR_PROMETHEUS_ROUTE_URL'] ←
43
44    - job_name: 'quarkus'
45      metrics_path: '/metrics/application'
46
47    static_configs:
48      - targets: ['YOUR_INVENTORY_ROUTE_URL'] ←

```

Create ←

Config maps hold key-value pairs and in the above command a `prometheus-config` config map is created with `prometheus.yml` as the key and the above content as the value. Whenever a config map is injected into a container, it would appear as a file with the same name as the key, at specified path on the filesystem.

Confirm you created the config map using the terminal command:

```
oc describe cm prometheus-config -n userXX-monitoring (replace userXX with your username)
```

Next, we need to *mount* this ConfigMap in the filesystem of the Prometheus container so that it can read it. Run this command to alter the Prometheus deployment to mount it (replace `userXX` with your username)

```
oc set volume -n userXX-monitoring dc/prometheus --add -t configmap --configmap-name=prometheus-config -m /etc/prometheus/prometheus.yml --sub-path=prometheus.yml
```

This will trigger a new deployment. Wait for it with:

```
oc rollout status -w dc/prometheus -n userXX-monitoring (replace userXX with your username)
```

13. Generate some values for the metrics

Let's write a loop to call our inventory service multiple times. First, get the URL to it (replace `userXX` with your username): `INV_URL=$(oc get route inventory-quarkus -n userxx-inventory -o=go-template --template='{{ .spec.host }}')`

Next, run this in the same Terminal:

```
for i in {1..50}; do curl http://${INV_URL}/services/inventory/329199 ; echo ; date ; sleep 1; done
```

This will continually access the inventory project and cause it to generate metrics.

Let's review the generated metrics. We have 3 ways to view the metrics:

- `curl` commands
- Prometheus Web UI
- Grafana Dashboards

Let's look at it with `curl` in a separate terminal:

```
INV_URL=$(oc get route inventory-quarkus -n userxx-inventory -o=go-template --template='{{ .spec.host }}')
```

and then

```
curl $INV_URL/metrics/application
```

You should something like:

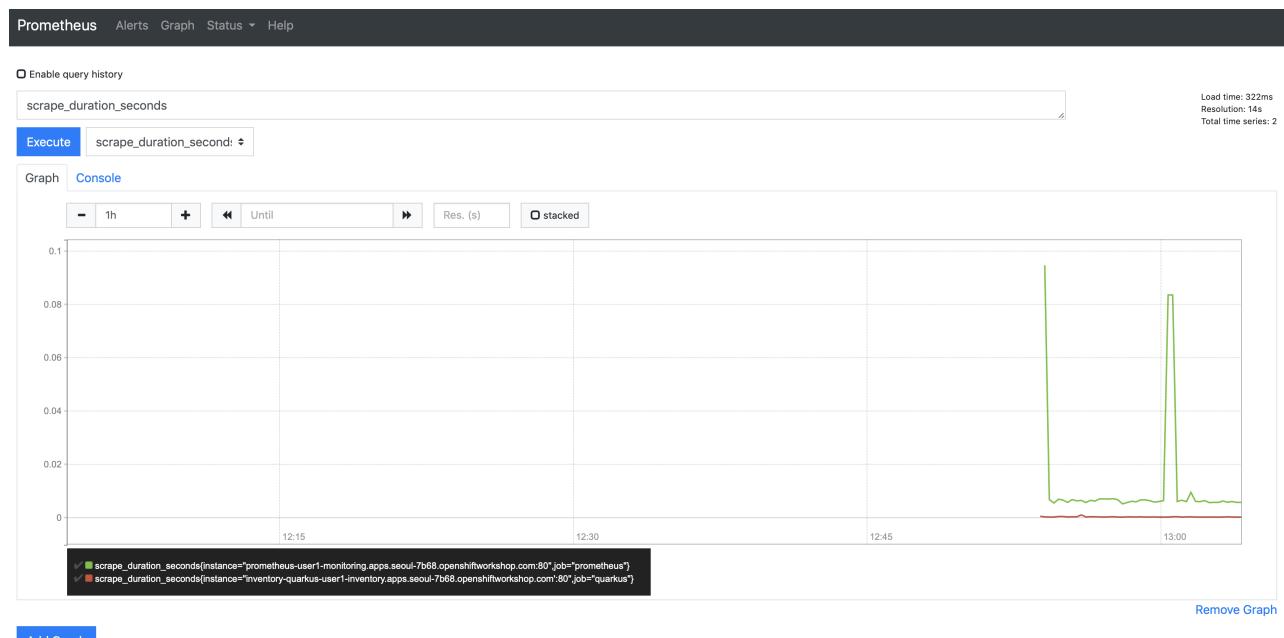
```
# TYPE application_com_redhat_coolstore_InventoryResource_checksTimerAll_stddev_seconds gauge
application_com_redhat_coolstore_InventoryResource_checksTimerAll_stddev_seconds
1.497008378628945E-4
# HELP application_com_redhat_coolstore_InventoryResource_checksTimerAll_seconds A measure of
how long it takes to perform the getAll().
# TYPE application_com_redhat_coolstore_InventoryResource_checksTimerAll_seconds summary
application_com_redhat_coolstore_InventoryResource_checksTimerAll_seconds_count 3107.0
application_com_redhat_coolstore_InventoryResource_checksTimerAll_seconds{quantile="0.5"}
0.001503754
application_com_redhat_coolstore_InventoryResource_checksTimerAll_seconds{quantile="0.75"}
0.001594015
application_com_redhat_coolstore_InventoryResource_checksTimerAll_seconds{quantile="0.95"}
0.001782487
application_com_redhat_coolstore_InventoryResource_checksTimerAll_seconds{quantile="0.98"}
0.001871631
application_com_redhat_coolstore_InventoryResource_checksTimerAll_seconds{quantile="0.99"}
0.001887301
application_com_redhat_coolstore_InventoryResource_checksTimerAll_seconds{quantile="0.999"}
0.001952198
```

This shows the raw metrics the application is collecting.

Now let's use Prometheus. Open the **Prometheus Web UI** via a web brower and input `scrape_duration_seconds` in the query box. This is a metric from Prometheus itself indicating how long it takes to scrape metrics. Click on **Execute** then you will see *quarkus job* in the metrics:

The screenshot shows the Prometheus Web UI interface. At the top, there are tabs for Prometheus, Alerts, Graph, Status, and Help. Below the tabs, there is a checkbox for 'Enable query history'. A search bar contains the query `scrape_duration_seconds`. To the right of the search bar, it says 'Load time: 499ms', 'Resolution: 14s', and 'Total time series: 2'. Below the search bar, there are two buttons: 'Execute' (which is highlighted in blue) and 'scrape_duration_second: ⚏'. Underneath these buttons are two tabs: 'Graph' (which is highlighted in blue) and 'Console'. A 'Moment' button is also present. The main area displays a table with two rows. The first row has a red border and contains the metric name `scrape_duration_seconds` followed by its value `0.000312587` and the label `instance="inventory-quarkus-user1-inventory.apps.seoul-7b68.openshiftworkshop.com:80",job="quarkus"`. The second row contains the metric name `scrape_duration_seconds` followed by its value `0.006989238` and the label `instance="prometheus-user1-monitoring.apps.seoul-7b68.openshiftworkshop.com:80",job="prometheus"`. On the far right of the table, there is a 'Value' column. At the bottom right of the table, there is a link 'Remove Graph'. At the very bottom left, there is a 'Add Graph' button.

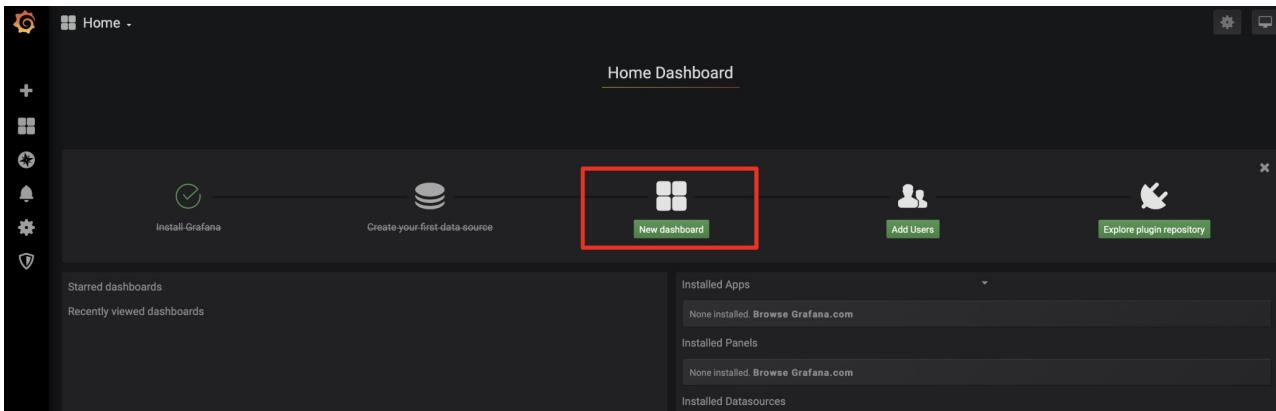
Switch to **Graph** tab:



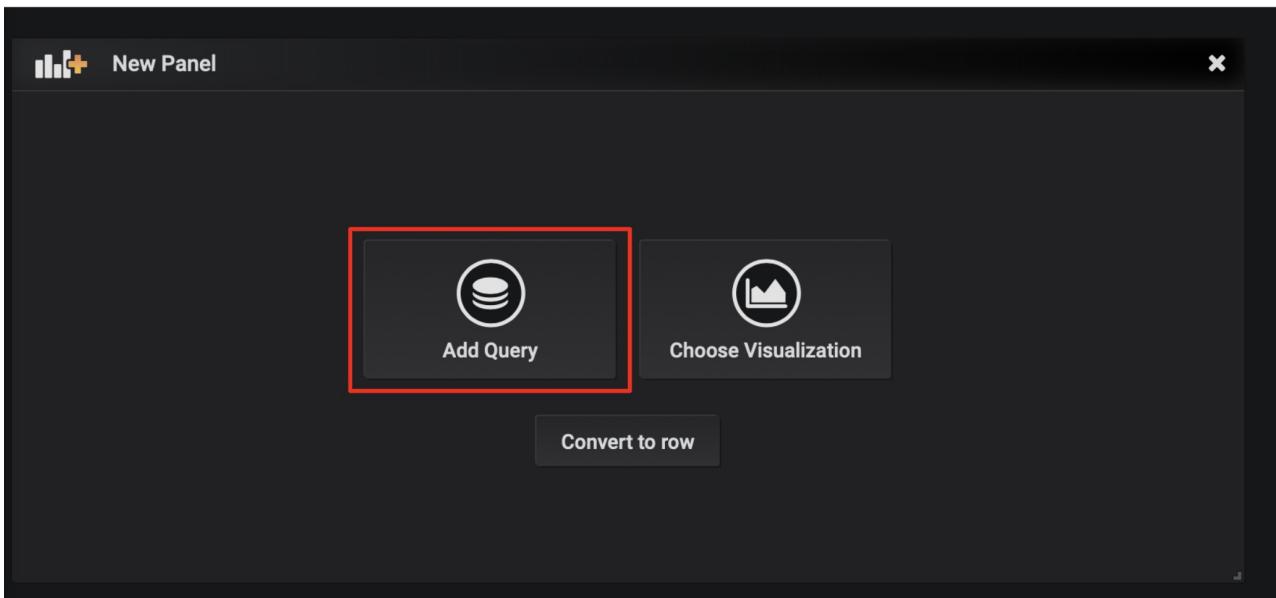
You can play with the values for time and see different data across different time ranges for this metric.

Now let's use Grafana.

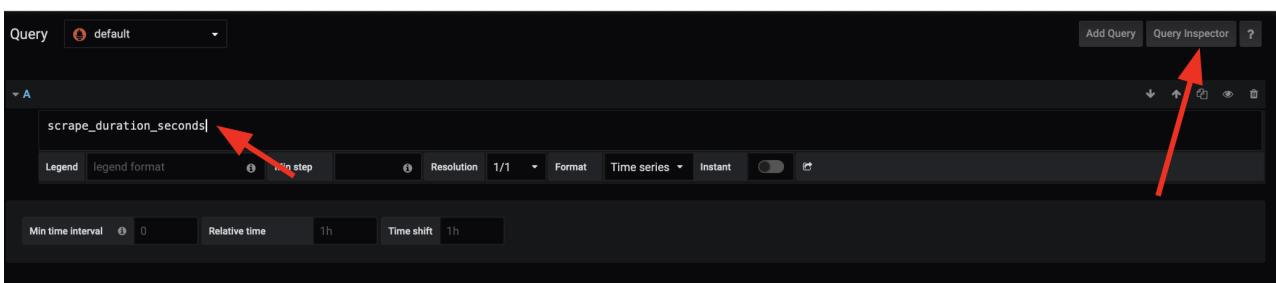
3) Open the **Grafana Web UI** (visit *Networking > Routes* in the `userXX-monitoring project in the OpenShift console) via a web brower and create a new *Dashboard* to review the metrics.



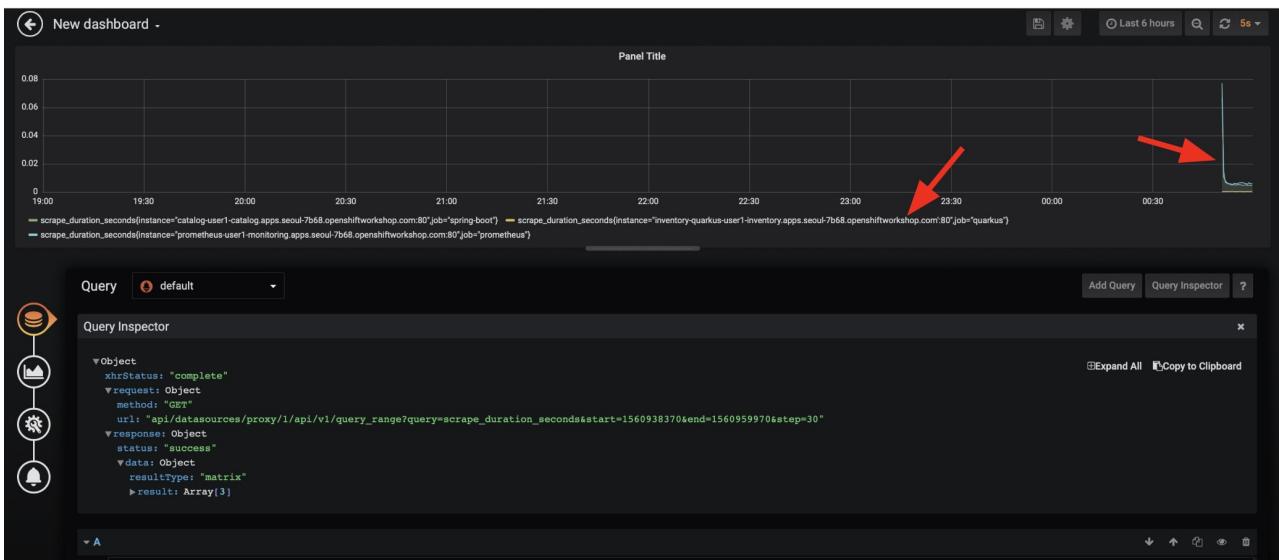
Click on **New dashboard** then select *Add Query* in a new panel:



Add **scrape_duration_seconds** in query box:



Click on **Query Inspector** then you will see *inventory-quarkus metrics* and change **5s** to refresh dashboard:



14. Utilize metrics specification for Catalog(Spring Boot)

In this step, we will learn how to export metrics to *Prometheus* from *Spring Boot* application by using the [Prometheus JVM Client](#).

Go to *Catalog* project directory and open *pom.xml* to add the following *Prometheus dependencies*:

```
<!-- Prometheus dependency -->
<dependency>
    <groupId>io.prometheus</groupId>
    <artifactId>simpleclient_spring_boot</artifactId>
    <version>0.6.0</version>
</dependency>

<dependency>
    <groupId>io.prometheus</groupId>
    <artifactId>simpleclient_hotspot</artifactId>
    <version>0.6.0</version>
</dependency>

<dependency>
    <groupId>io.prometheus</groupId>
    <artifactId>simpleclient_servlet</artifactId>
    <version>0.6.0</version>
</dependency>
```

```
<dependency>
    <groupId>org.json</groupId>
    <artifactId>json</artifactId>
    <version>20180813</version>
</dependency>

<!-- Prometheus dependency -->
<dependency>
    <groupId>io.prometheus</groupId>
    <artifactId>simpleclient_spring_boot</artifactId>
    <version>0.6.0</version>
</dependency>

<dependency>
    <groupId>io.prometheus</groupId>
    <artifactId>simpleclient_hotspot</artifactId>
    <version>0.6.0</version>
</dependency>

<dependency>
    <groupId>io.prometheus</groupId>
    <artifactId>simpleclient_web</artifactId>
    <version>0.6.0</version>
</dependency>

<!-- TODO: Add web (tomcat) dependency here -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<!-- TODO: Add data jpa dependency here -->
```

Next, let's create a new class to configure our metrics for Spring.

In the `catalog` project in CodeReady, open the empty `src/main/java/com/redhat/coolstore/MonitoringConfig.java` class and add the following code to it:

```

package com.redhat.coolstore;

import io.prometheus.client.exporter.MetricsServlet;
import io.prometheus.client.hotspot.DefaultExports;
import io.prometheus.client.spring.boot.SpringBootMetricsCollector;
import org.springframework.boot.actuate.endpoint.PublicMetrics;
import org.springframework.boot.web.servlet.ServletRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.Collection;

@Configuration
class MonitoringConfig {

    @Bean
    SpringBootMetricsCollector springBootMetricsCollector(Collection<PublicMetrics> publicMetrics) {

        SpringBootMetricsCollector springBootMetricsCollector = new
        SpringBootMetricsCollector(publicMetrics);
        springBootMetricsCollector.register();

        return springBootMetricsCollector;
    }

    @Bean
    ServletRegistrationBean servletRegistrationBean() {
        DefaultExports.initialize();
        return new ServletRegistrationBean(new MetricsServlet(), "/prometheus");
    }
}

```

15. Re-Build and Re-Deploy to OpenShift

Build and deploy the Catalog project using the following command, which will rebuild and redeploy to OpenShift:

```
cd /projects/cloud-native-workshop-v2m2-labs/catalog
```

and then

```
mvn clean package spring-boot:repackage -DskipTests
```

The build and deploy may take a minute or two. Wait for it to complete. You should see a **BUILD SUCCESS** at the end of the build output.

And then re-build the container image, which will take about a minute to complete:

```
oc start-build -n userNN-catalog catalog-springboot --from-file target/catalog-1.0.0-
SNAPSHOT.jar --follow (replace userNN with your username!)
```

Once the build is done, it will automatically start a new deployment. Wait for it to complete:

```
oc rollout status -w dc/catalog-springboot
```

Wait for that command to report replication controller “catalog-springboot-XX” successfully rolled out before continuing.

NOTE: Even if the rollout command reports success the application may not be ready yet and the reason for that is that we currently don't have any liveness check configured, but we will add that in the next steps.

Let's access the metrics from the catalog service. Access them via `curl` with these commands:

```
CAT_URL=$(oc get route catalog-springboot -n userXX-catalog -o=go-template --template='{{ .spec.host }}')
```

and then:

```
curl $CAT_URL/prometheus
```

You will see a similar output as here:

```
# HELP jvm_gc_collection_seconds Time spent in a given JVM garbage collector in seconds.
# TYPE jvm_gc_collection_seconds summary
jvm_gc_collection_seconds_count{gc="PS Scavenge",} 6.0
jvm_gc_collection_seconds_sum{gc="PS Scavenge",} 0.125
jvm_gc_collection_seconds_count{gc="PS MarkSweep",} 7.0
jvm_gc_collection_seconds_sum{gc="PS MarkSweep",} 1.017
# HELP jvm_classes_loaded The number of classes that are currently loaded in the JVM
# TYPE jvm_classes_loaded gauge
jvm_classes_loaded 9238.0
# HELP jvm_classes_loaded_total The total number of classes that have been loaded since the JVM has started execution
# TYPE jvm_classes_loaded_total counter
jvm_classes_loaded_total 9238.0
# HELP jvm_classes_unloaded_total The total number of classes that have been unloaded since the JVM has started execution
# TYPE jvm_classes_unloaded_total counter
jvm_classes_unloaded_total 0.0
# HELP process_cpu_seconds_total Total user and system CPU time spent in seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 16.45
...
```

This is the raw output from the application that Prometheus will periodically read (“scrape”).

16. Add Catalog(Spring Boot) Job

You'll need the catalog route once again, which you can discover using this in the Terminal:

```
oc get route catalog-springboot -n userXX-catalog -o=go-template --template='{{ .spec.host }}'; echo
```

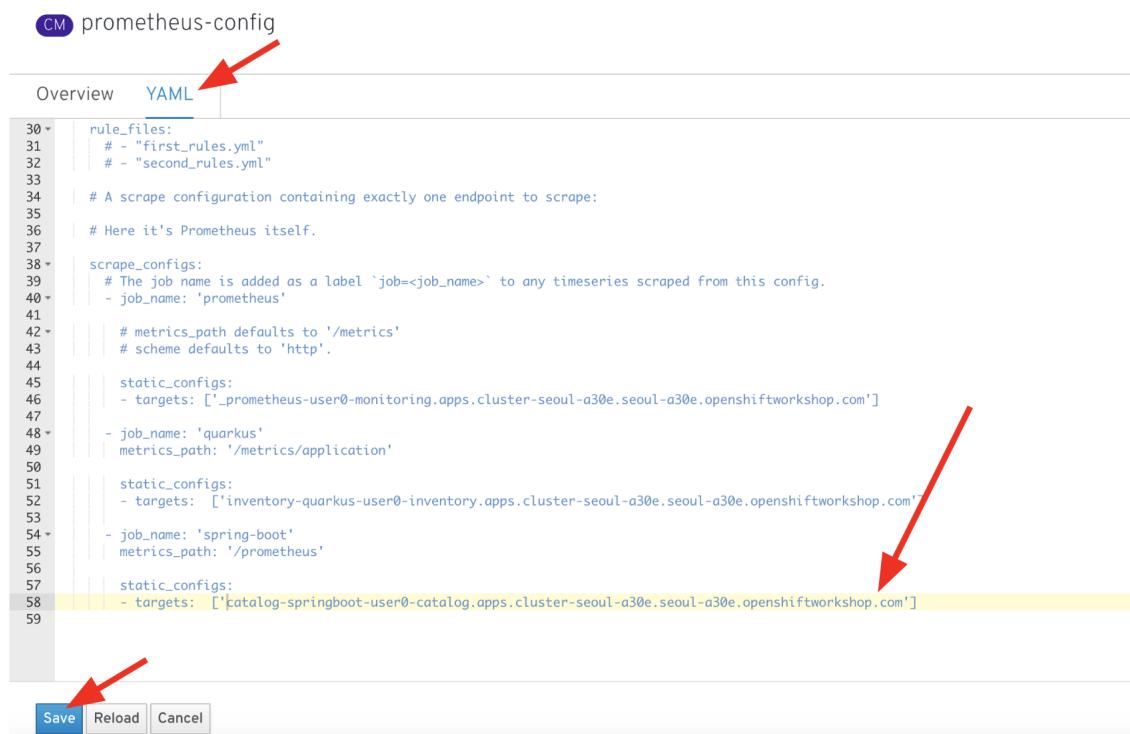
Navigate to your `userXX-monitoring` project in OpenShift console, and go to [Workloads > Config Maps > prometheus_config](#). Click on the `YAML` tab.

Edit to add the following contents below the existing `job_name` elements (and with the same indentation):

Replace `YOUR_CATALOG_ROUTE` with the route emitted from the above `oc get route` command!

```
- job_name: 'spring-boot'  
  metrics_path: '/prometheus'  
  
  static_configs:  
    - targets: ['YOUR_CATALOG_ROUTE']
```

Click on **Save**.



OpenShift does not automatically redeploy whenever ConfigMaps are changed, so let's force a redeployment. Select the `userXX-catalog` project in the OpenShift console, navigate to `Workloads > Deployment Configs > prometheus` and select **Start Rollout** from the Actions menu:

The screenshot shows the Red Hat OpenShift Container Platform interface. The left sidebar has a 'Workloads' section with various options like Pods, Deployments, Stateful Sets, etc. The 'Deployment Configs' option is highlighted with a red arrow. On the right, there's a table of deployment configurations. One row for 'prometheus' is selected, and a red arrow points to the three-dot menu icon next to it. A context menu is open, listing options such as 'Start Rollout', 'Pause Rollouts', 'Edit Count', 'Add Storage', 'Edit Environment', 'Edit Labels', 'Edit Annotations', 'Edit Deployment Config', and 'Delete Deployment Config'.

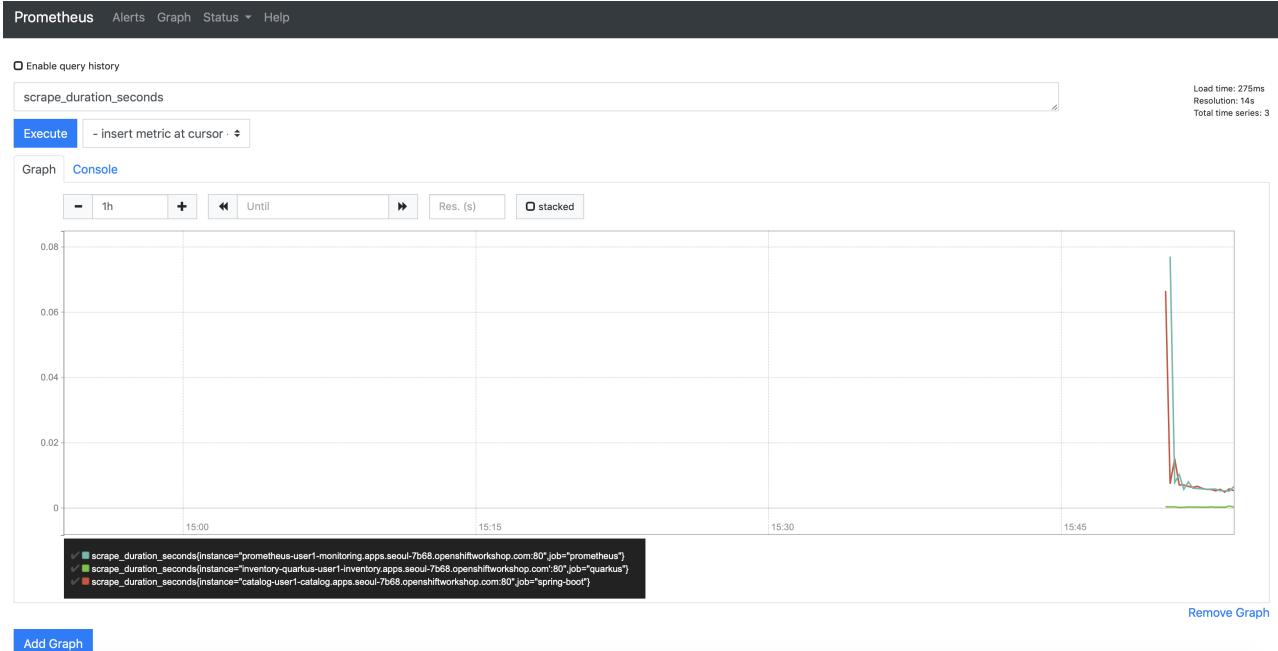
17. Observing metrics in Prometheus and Grafana

1) Open the Prometheus Web UI via a web brower and input(or select) `scrape_duration_seconds` in the query box. Click on **Execute then you will see *quarkus job* in the metrics:**

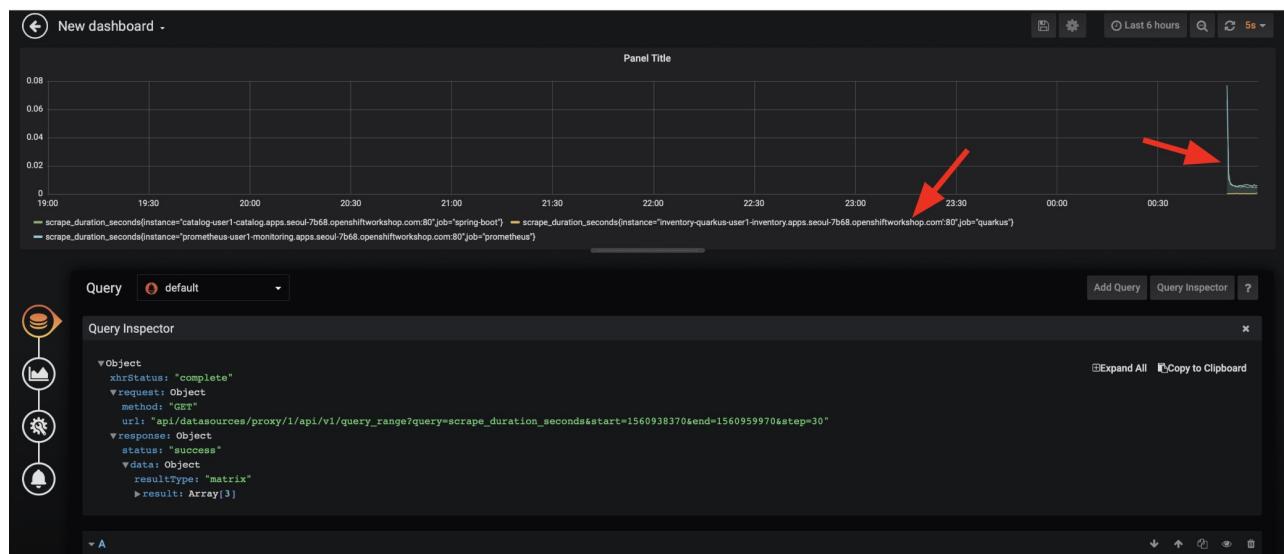
The screenshot shows the Prometheus Web UI. At the top, there are tabs for Prometheus, Alerts, Graph, Status, and Help. Below that is a search bar with 'scrape_duration_seconds' and an 'Execute' button. Underneath is a 'Graph' tab and a 'Console' tab. The main area shows a table of results. The first row, for 'spring-boot', has a value of 0.066568628. The second row, for 'quarkus', has a value of 0.000427491. The entire results table is highlighted with a red box.

Element	Value
scrape_duration_seconds(instance="catalog-user1-catalog.apps.seoul-7b68.openshiftworkshop.com:80";job="spring-boot")	0.066568628
scrape_duration_seconds(instance="inventory-quarkus-user1-inventory.apps.seoul-7b68.openshiftworkshop.com:80";job="quarkus")	0.000427491

Switch to **Graph** tab:



2) Open the Grafana Web UI via a web brower and access your existing Dashboard. Try to add a new Query with some of the application metrics.



Summary

In this lab, you learned how to monitor cloud-native applications using Jaeger, Prometheus, and Grafana. You also learned how Quarkus makes your observation tasks easier as a developer and operator. You can use these techniques in future projects to observe your distributed cloud-native applications.