The Containers and Cloud-Native Roadshow Dev Track

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 the 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.

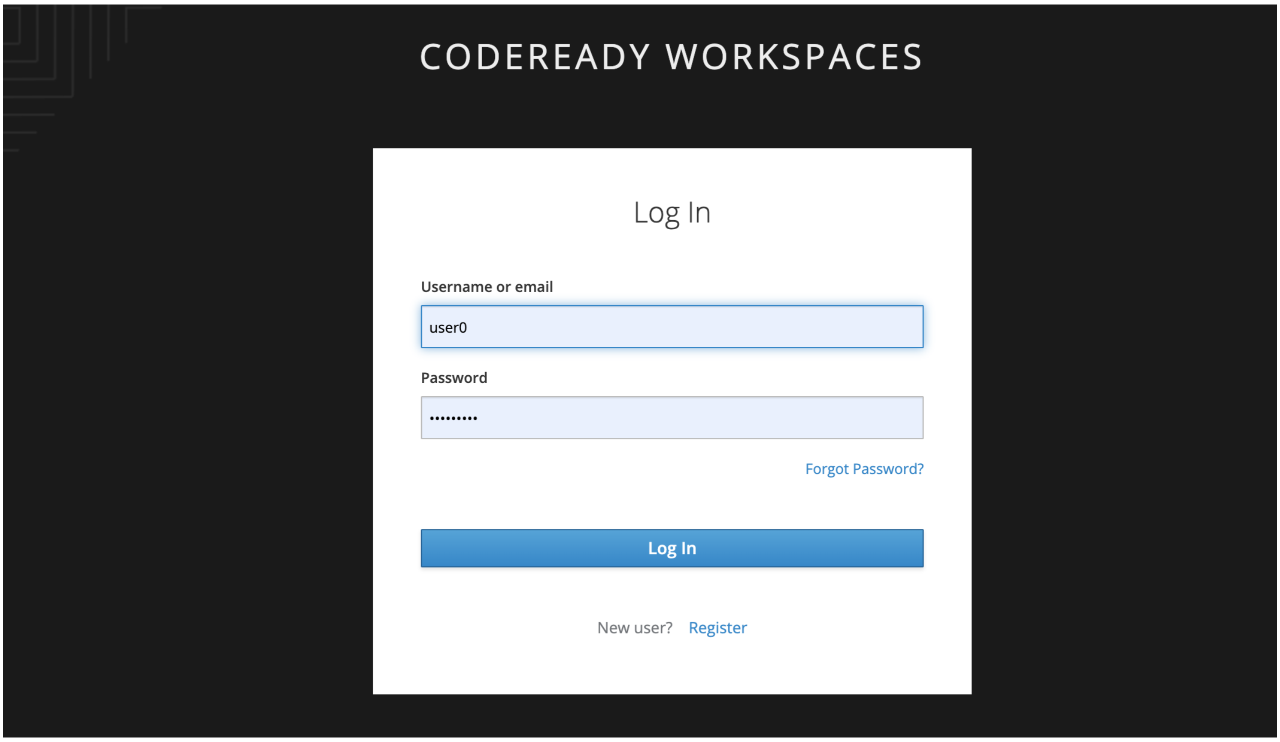
Getting Ready for the labs

> **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.

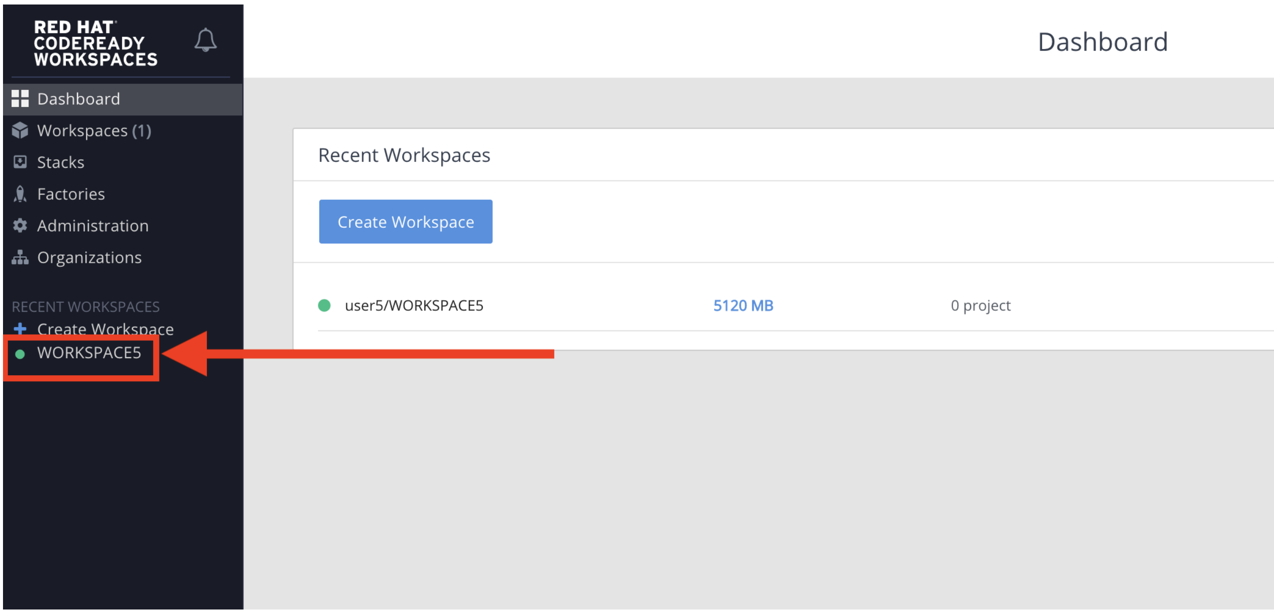
If this is the first module you are doing today

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

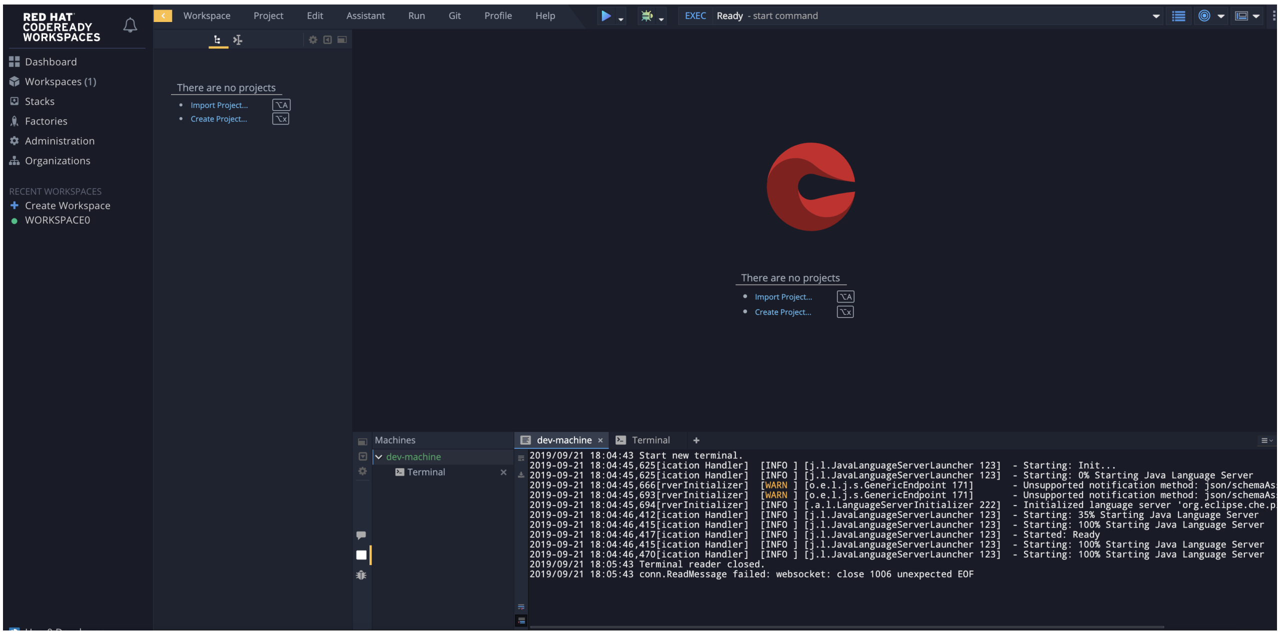
To get started, access the Che instance and log in using the username and password you've been assigned (e.g. `userXX/r3dh4t1!`):



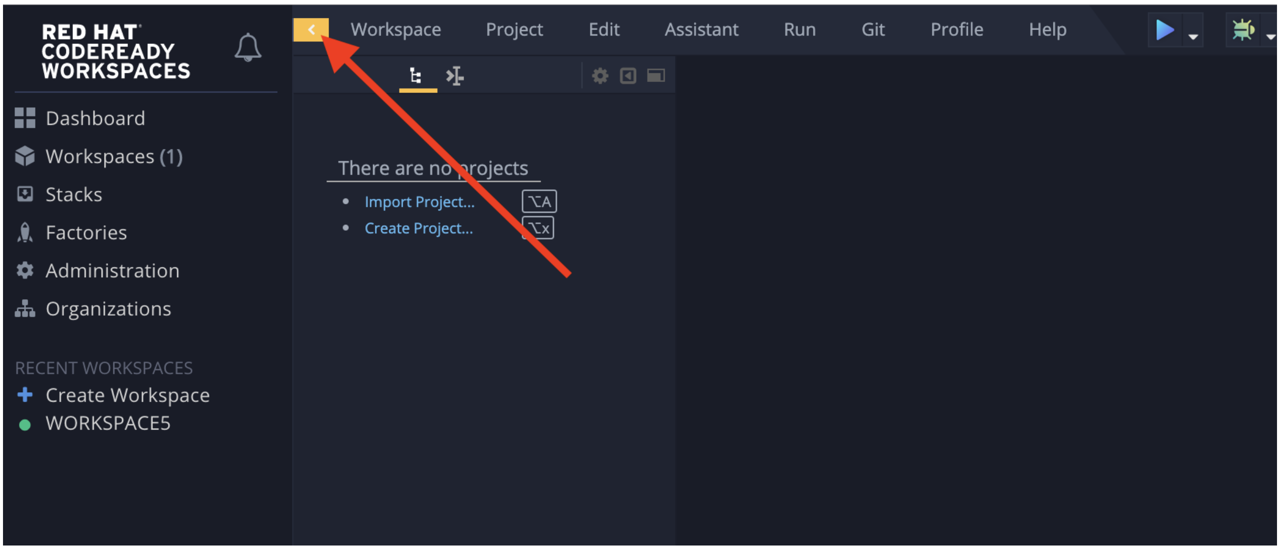
Once you log in, you'll be placed on your personal dashboard. We've pre-created workspaces for you to use. Click on the name of the pre-created workspace on the left, as shown below (the name will be different depending on your assigned number). You can also click on the name of the workspace in the 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:



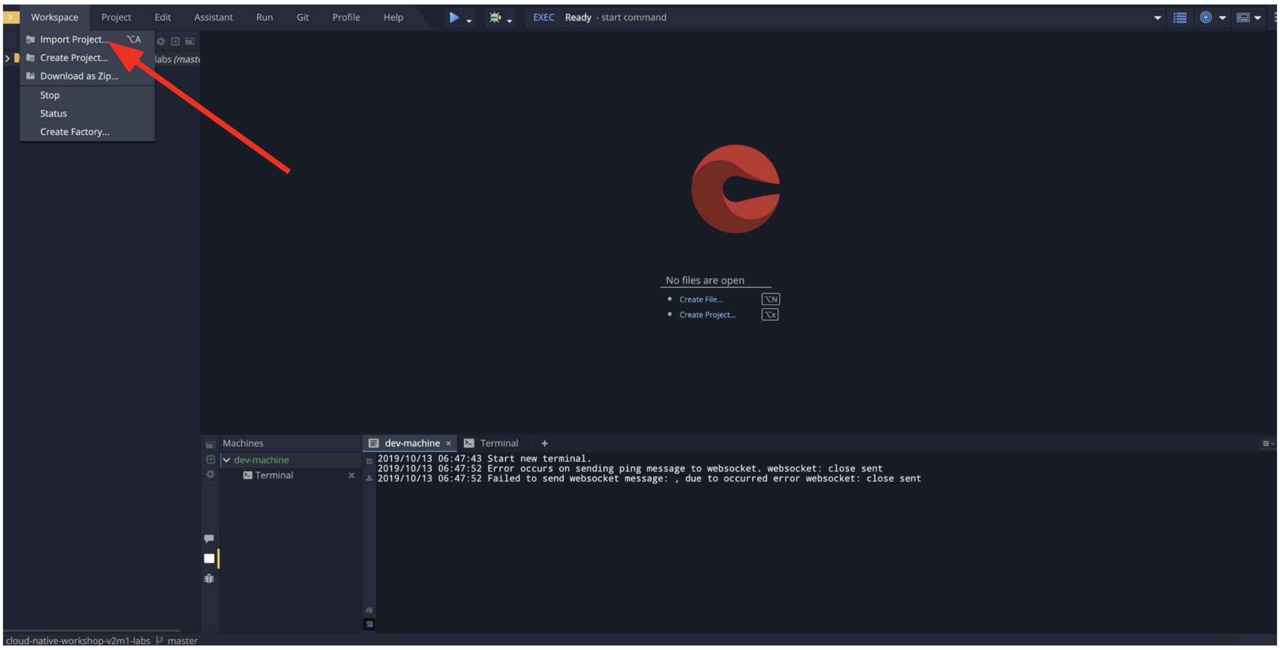
To gain extra screen space, click on the yellow arrow to hide the left menu (you won't need it):



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

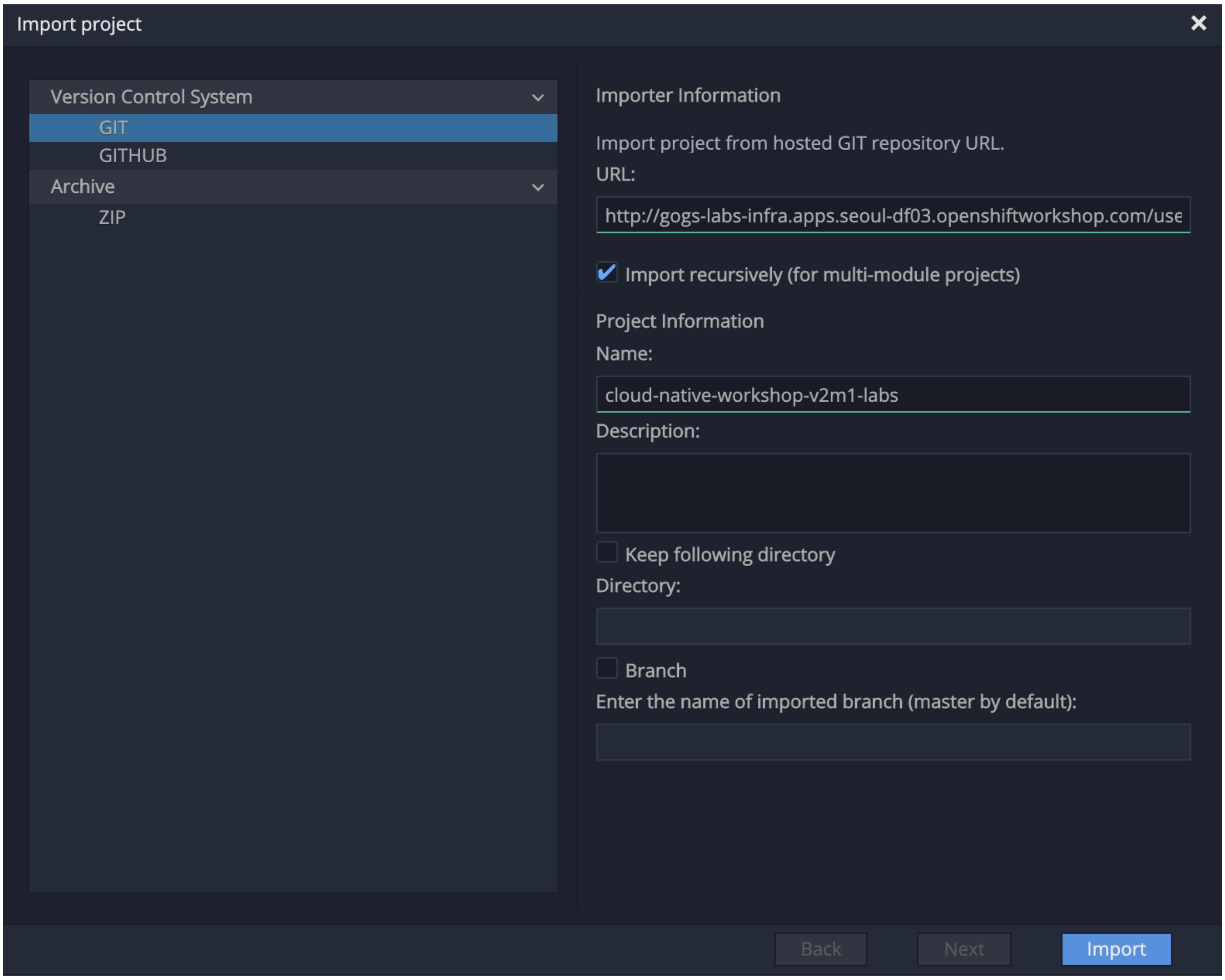
Click on the **Import Projects...** in **Workspace** menu and enter the following:



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

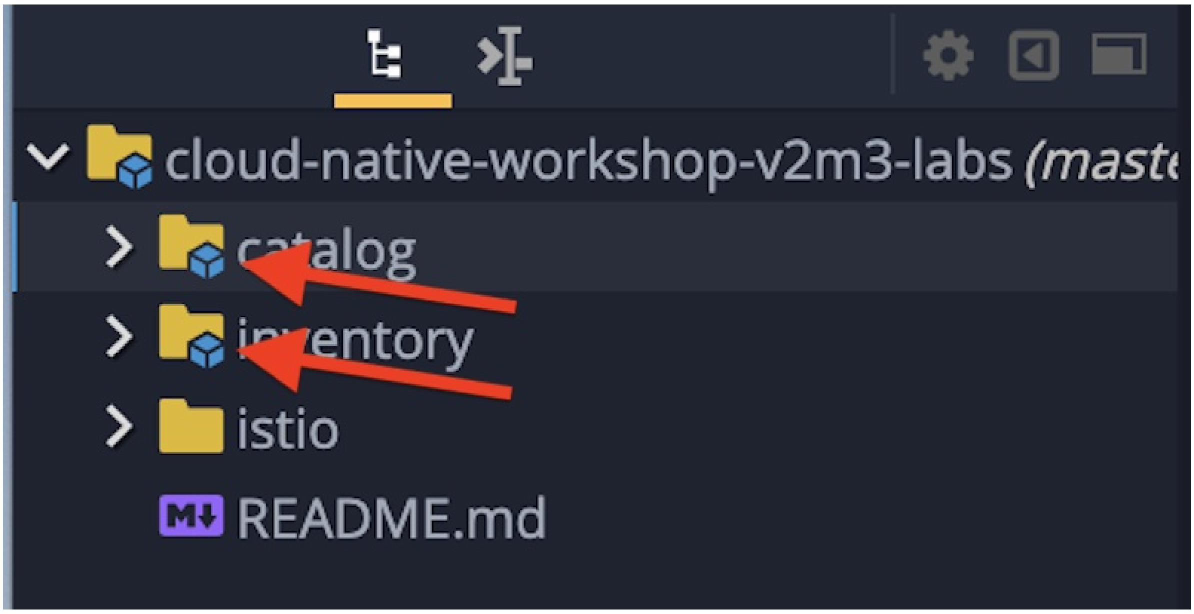
- Version Control System: `GIT`
- URL: `http://gogs-labs-infra.apps.cluster-beijing-5bc2.beijing-5bc2.open.redhat.com/userXX/cloud-native-workshop-v2m2-labs.git` (IMPORTANT: replace userXX with your lab user)
- Check `Import recursively (for multi-module projects)`
- Name: `cloud-native-workshop-v2m2-labs`

Tip: You can find GIT URL when you click on GIT URL then login with your credentials.

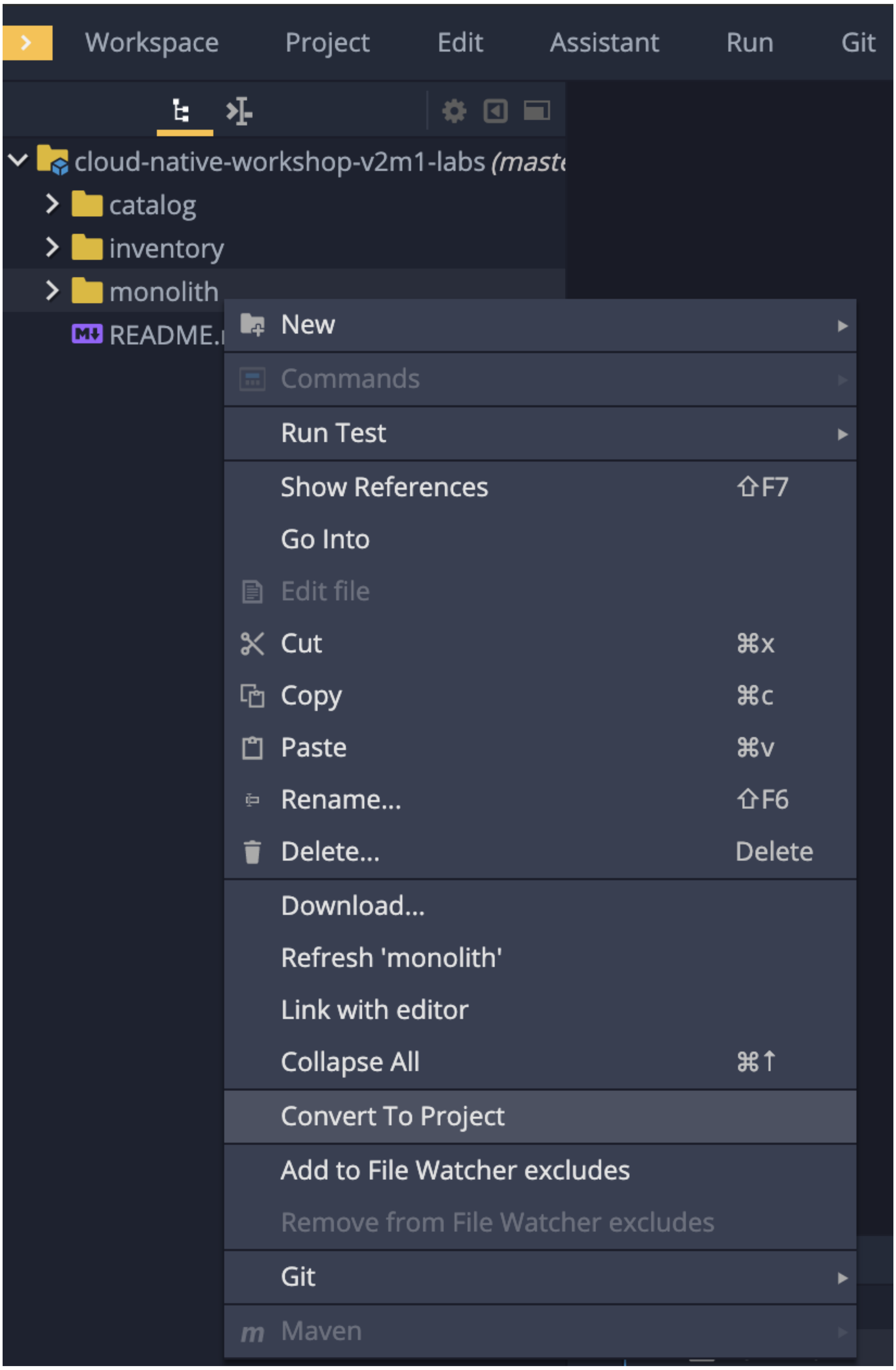


The projects are imported now into your workspace and is visible in the project explorer.

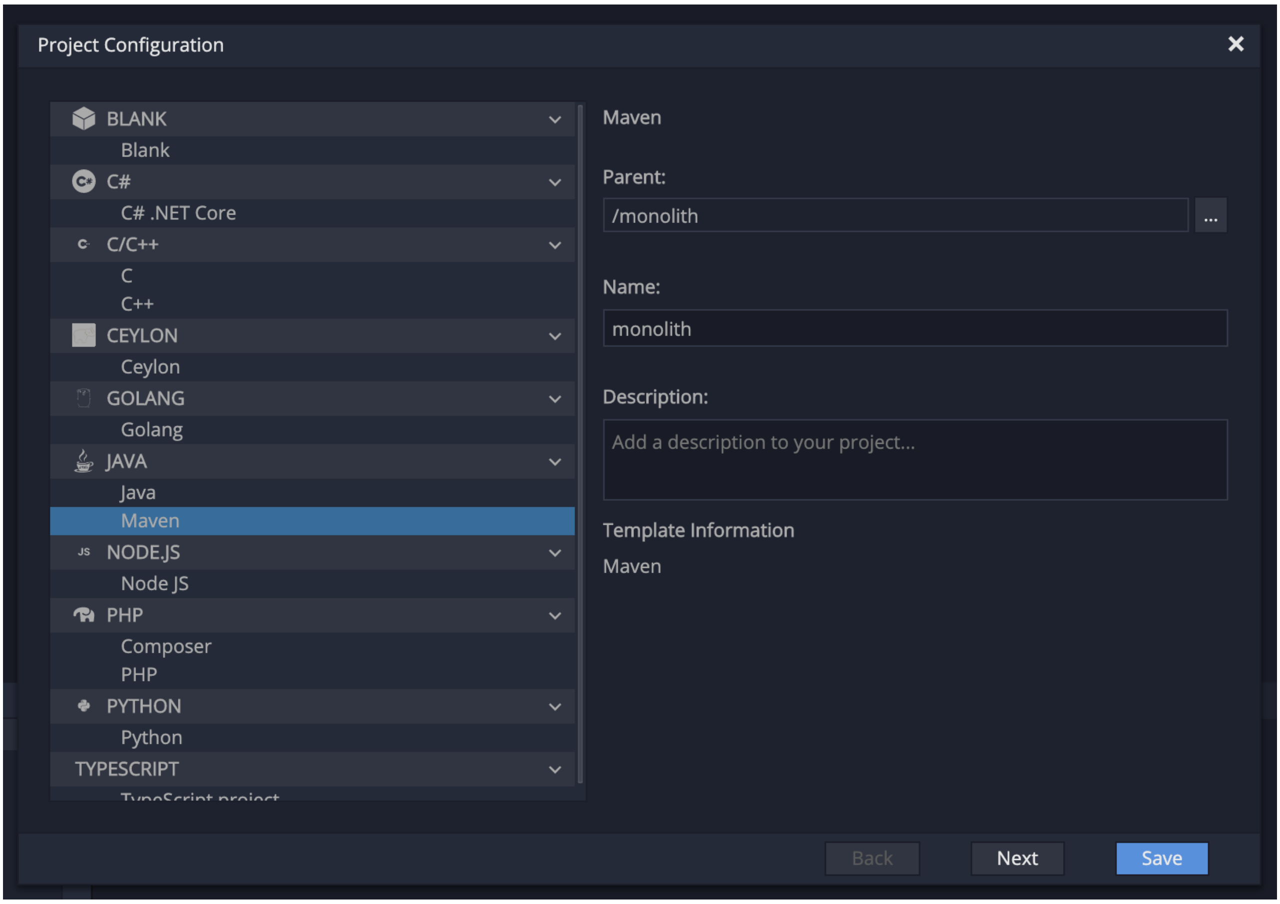
CodeReady Workspaces is a full featured IDE and provides language specific capabilities for various project types. In order to enable these capabilities, let's convert the imported project skeletons to a Maven projects. In the project explorer, right-click on each project (`monolith` , `inventory` and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

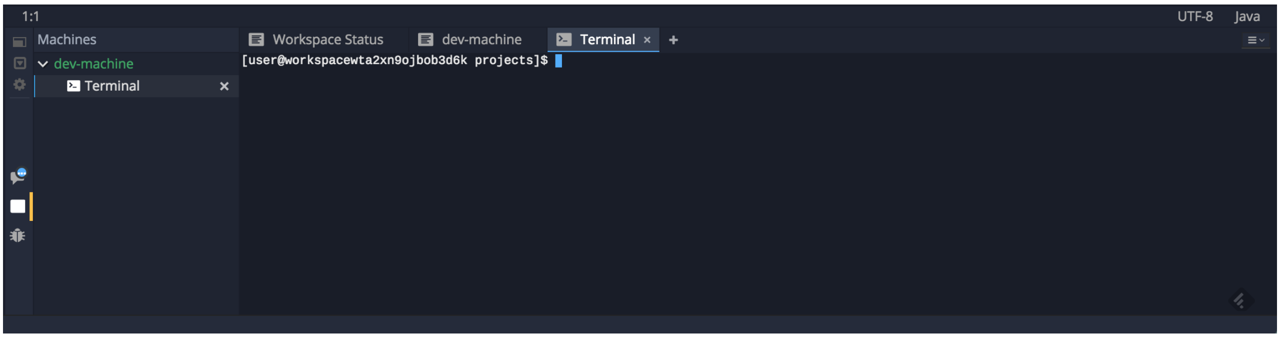


Choose `Maven` from the project configurations and then click on `Save` .



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

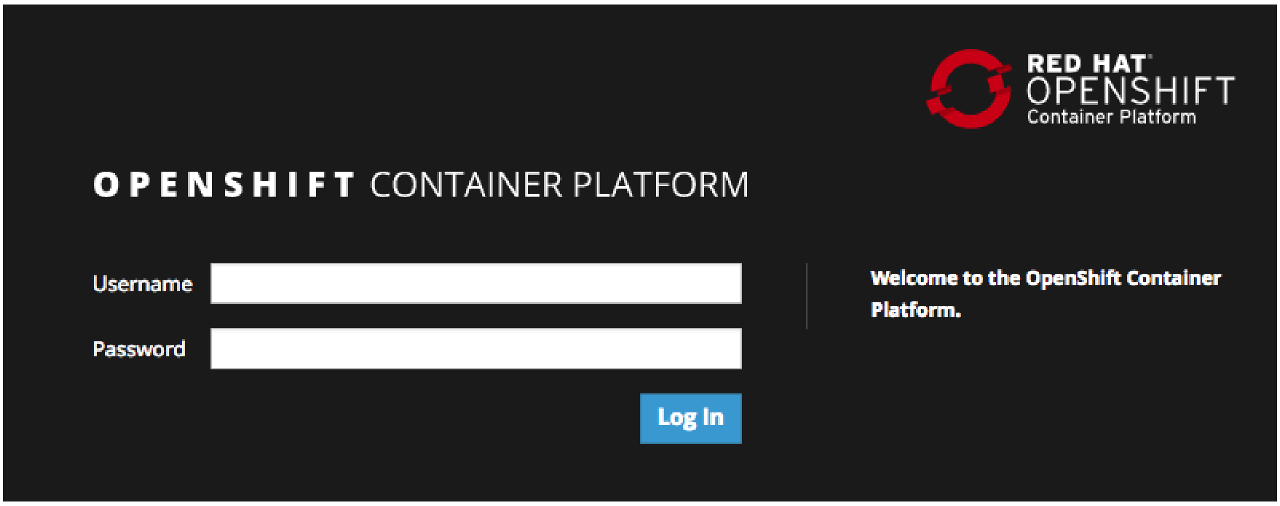
The existing **userXX-coolstore-dev** project is used as a developer environment for building new versions of the app after code changes and deploying them to the development environment.

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.

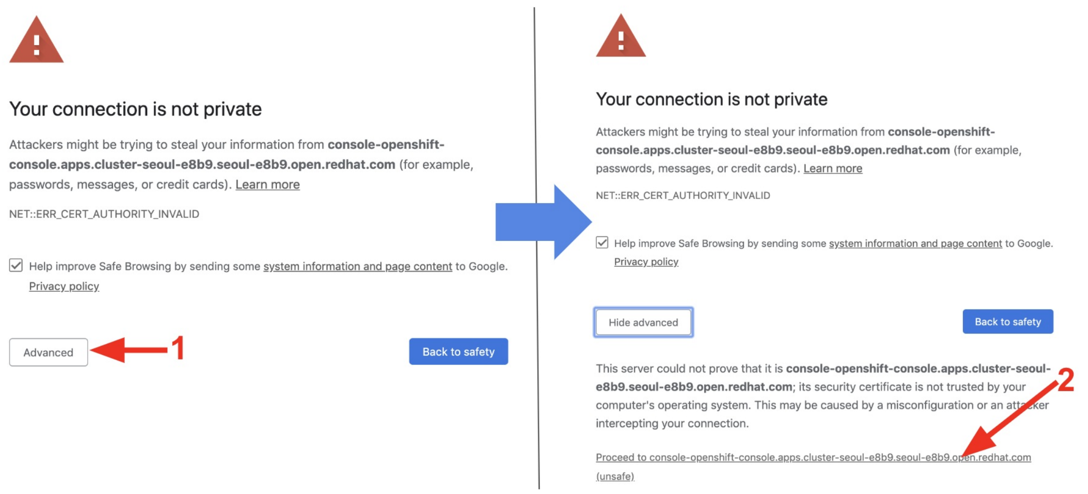
1. Create the production environment

First, open a new brower with the OpenShift web console

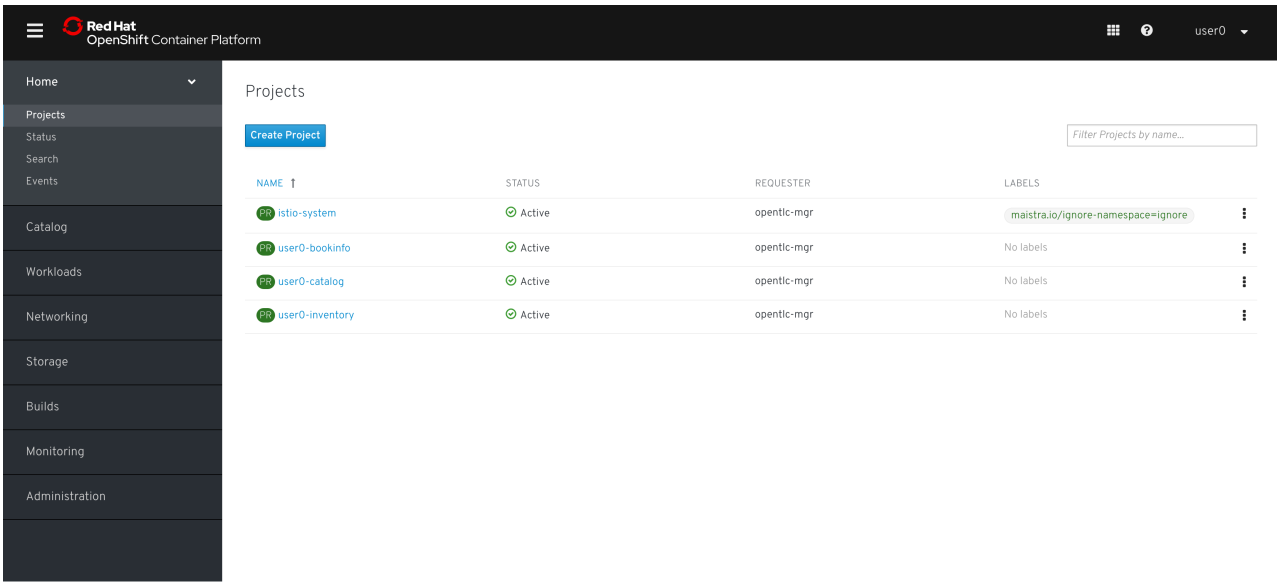


Login using:

- 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:



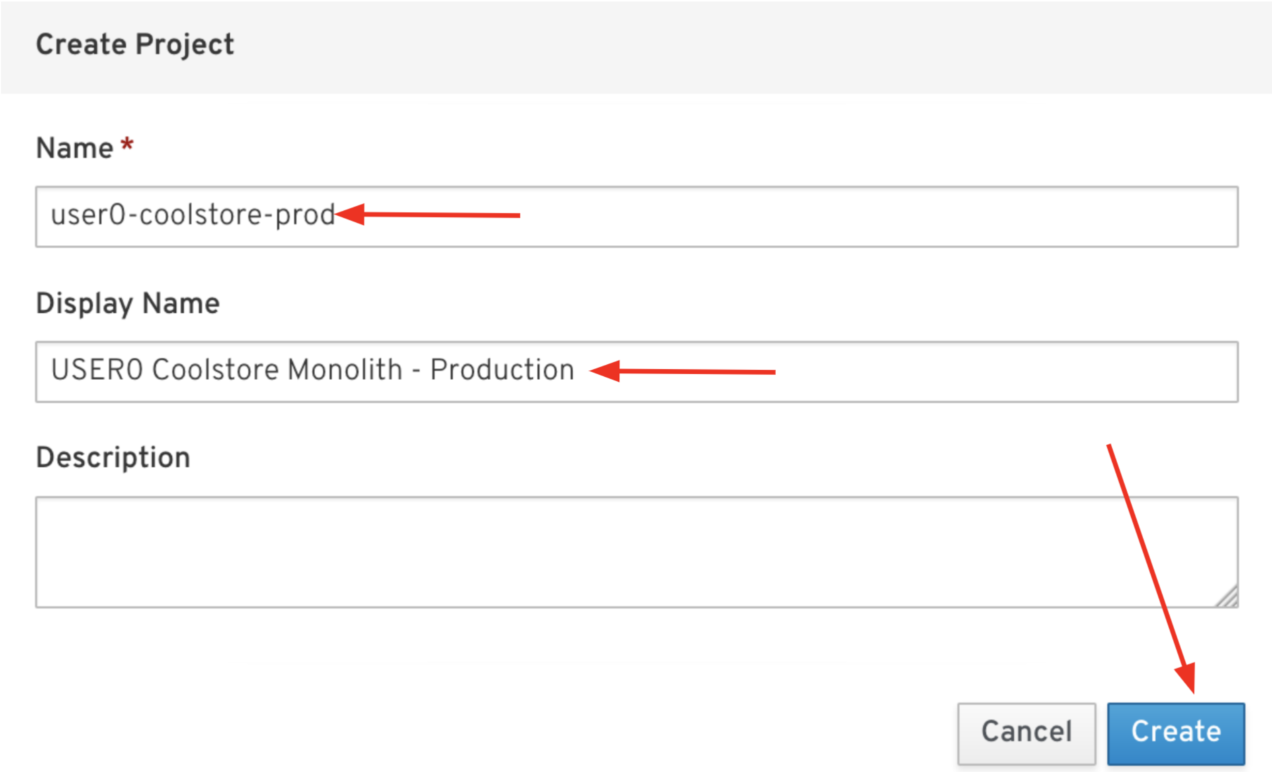
> The project displayed in the landing page depends on which labs you will run today. If you will develop `Service Mesh and Identity` then you will see pre-created projects as the above screeenshot.

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.



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.

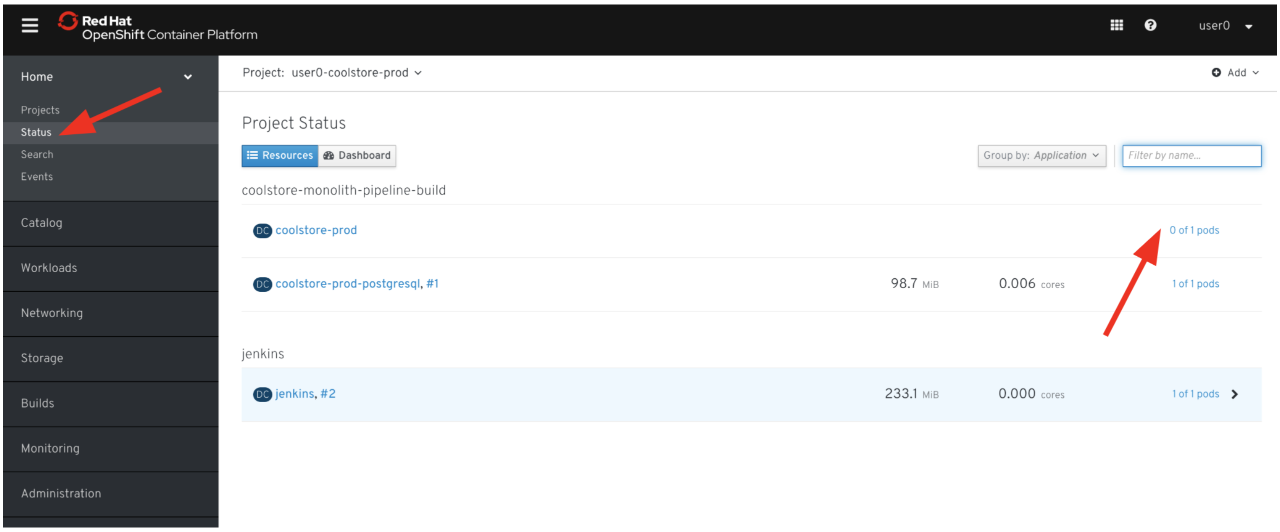
```
oc new-app --template=jenkins-ephemeral -l app=jenkins -p JENKINS_SERVICE_NAME=jenkins -p DISABLE_ADMINISTRATIVE_MONITORS=true
```

```
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:



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

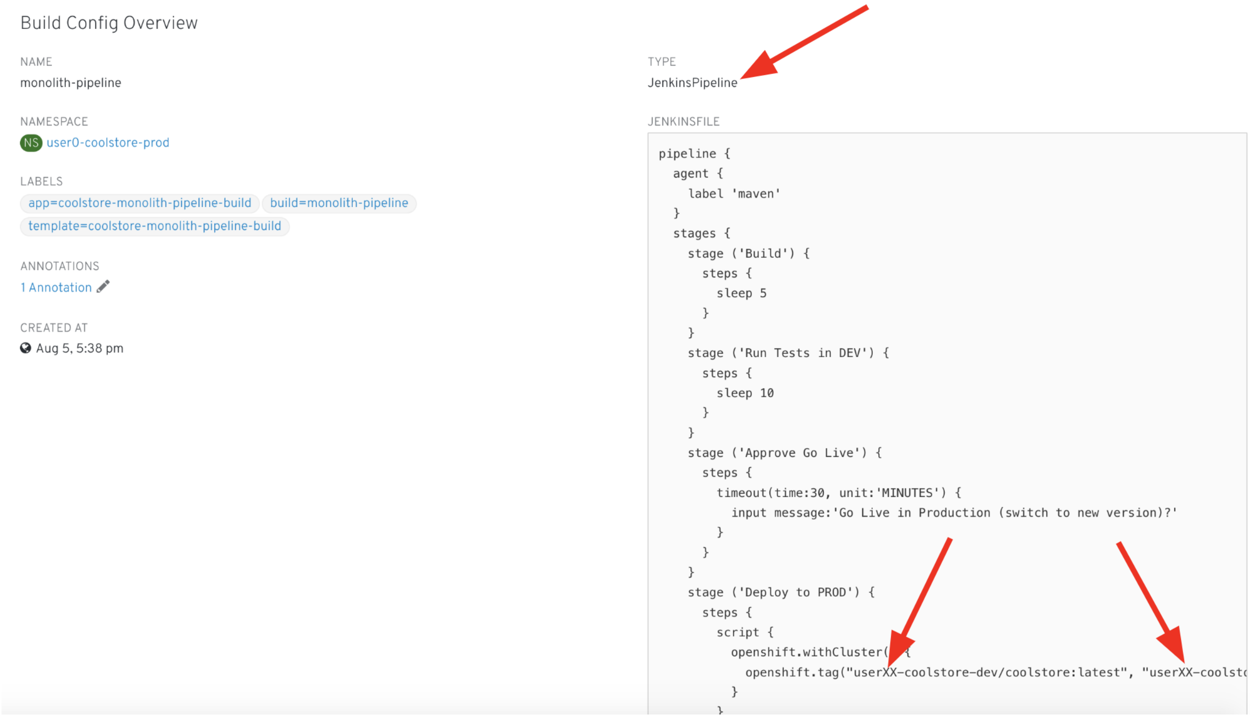
OpenShift has built-in support for CI/CD pipelines by allowing developers to define a Jenkins pipeline for execution by a Jenkins automation engine, which is automatically provisioned on-demand by OpenShift when needed.

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:



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

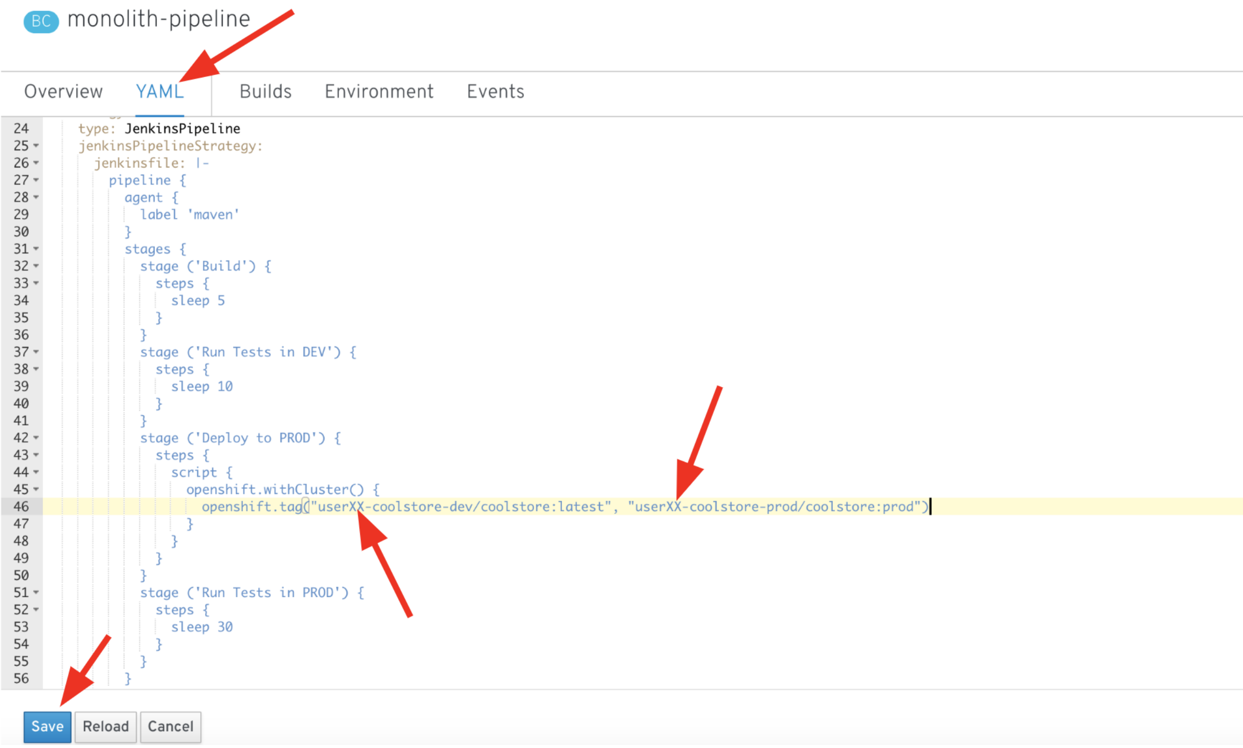
`oc describe bc/monolith-pipeline`

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**.



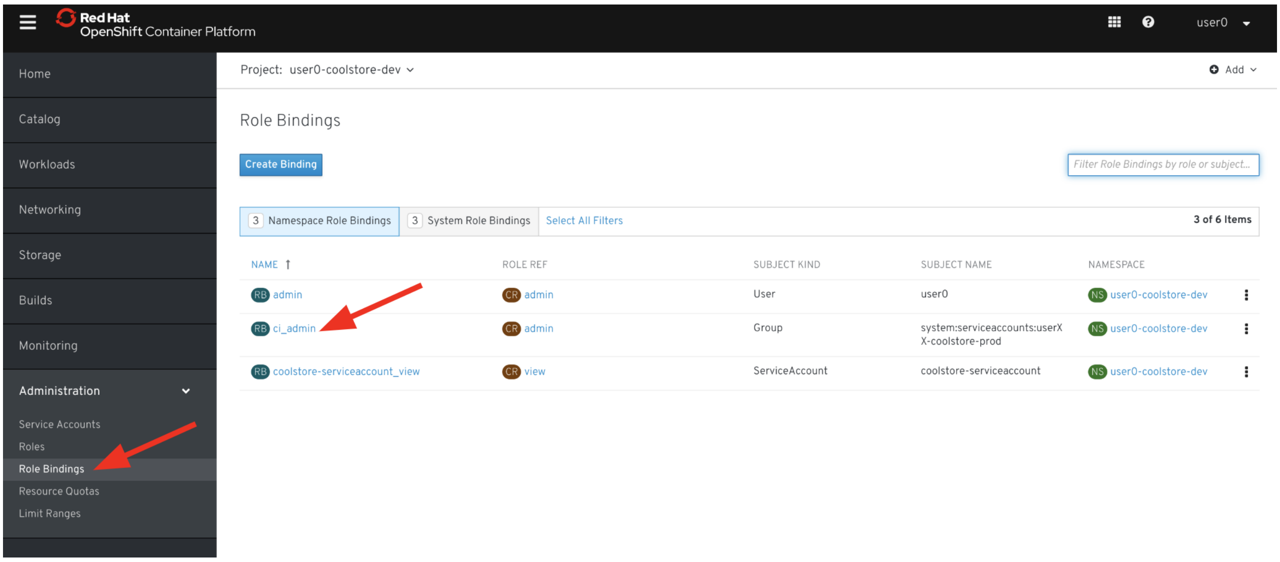
The pipeline syntax allows creating complex deployment scenarios with the possibility of defining checkpoints for manual interaction and approval processes using the large set of steps and plugins that Jenkins provides in order to adapt the pipeline to the processes used in your team. You can see a few 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.

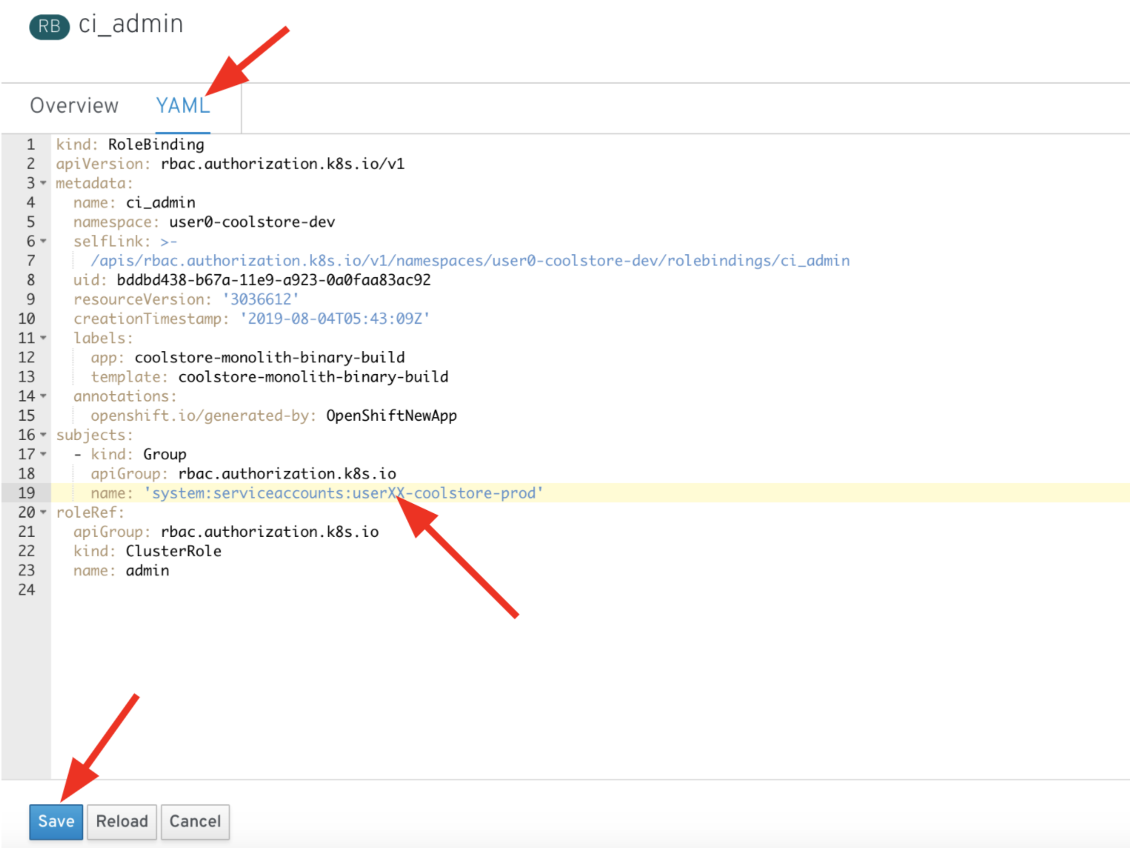
4. Promote the dev image to production using the pipeline

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**:

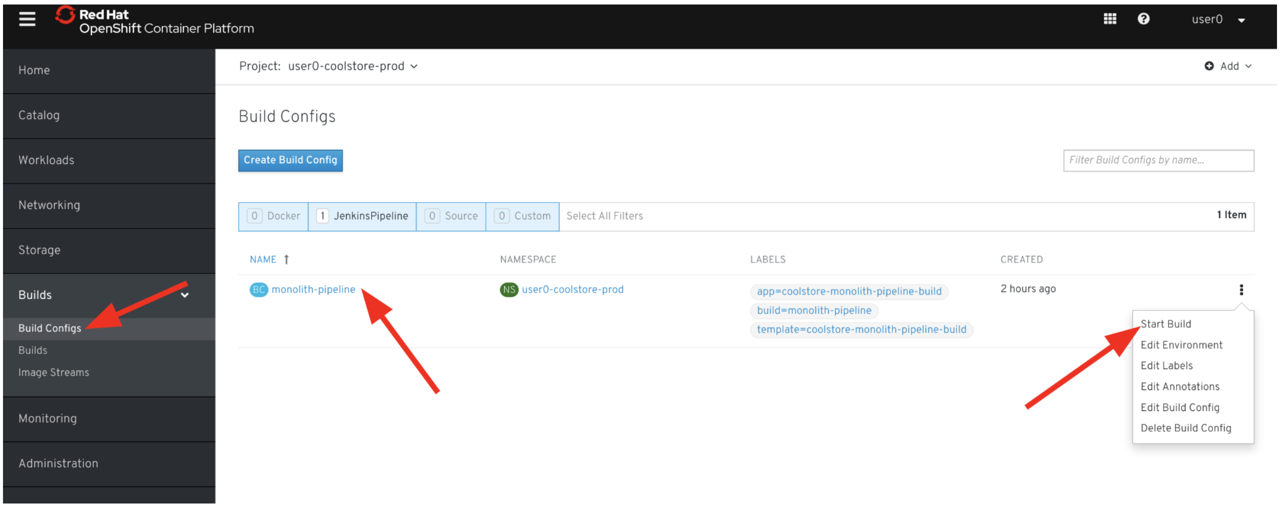


Move to **YAML** tab and replace your username with *userXX* then click on **Save**:

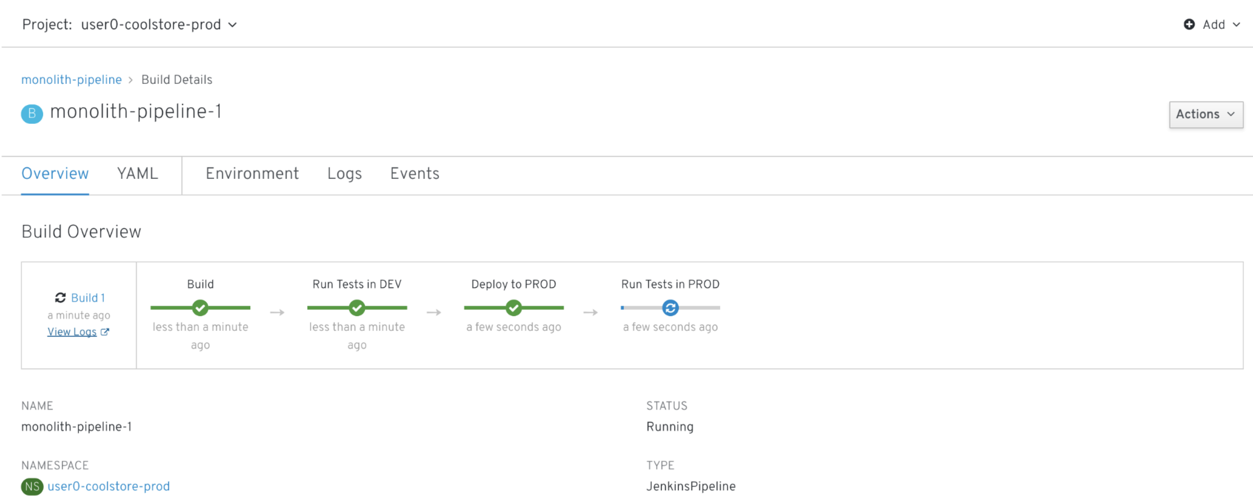


Let's invoke the build pipeline by using OpenShift web console. Open the production project in the web console.

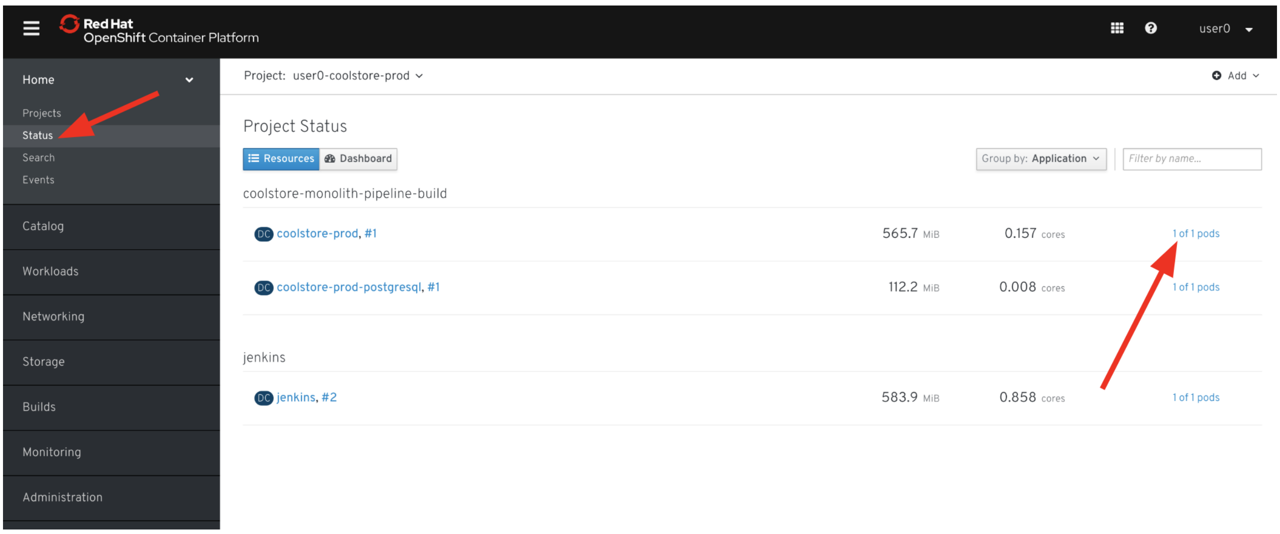
Next, navigate to *Builds* > *Build Configs* > *monolith-pipeline*, click the small menu at the far right, and click *Start Build*:



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:



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



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

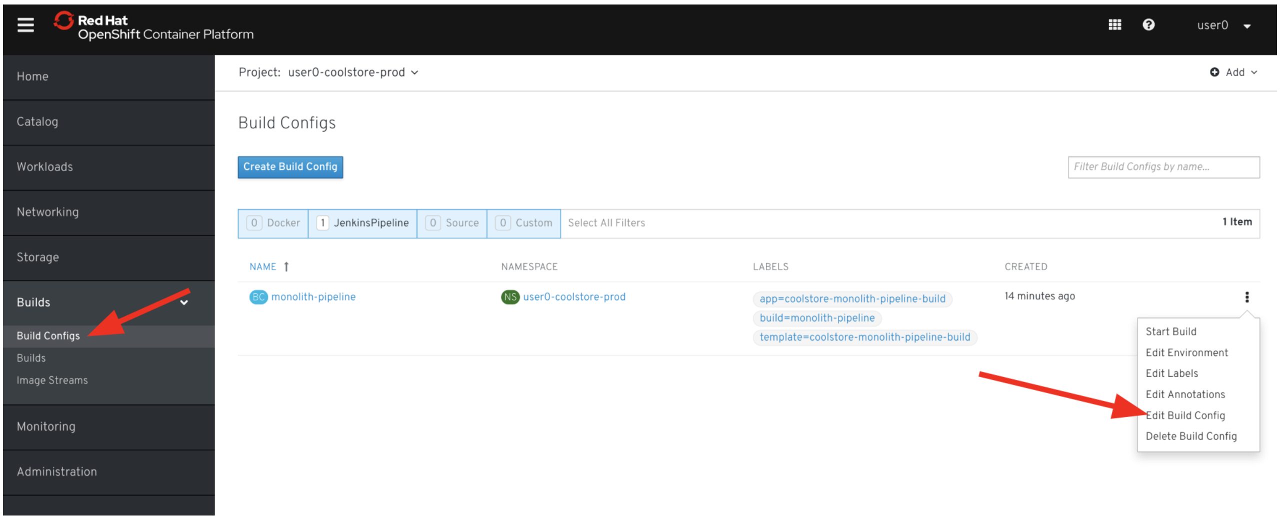
In previous steps, you used an OpenShift Pipeline to automate the process of building and deploying changes from the dev environment to production.

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*.

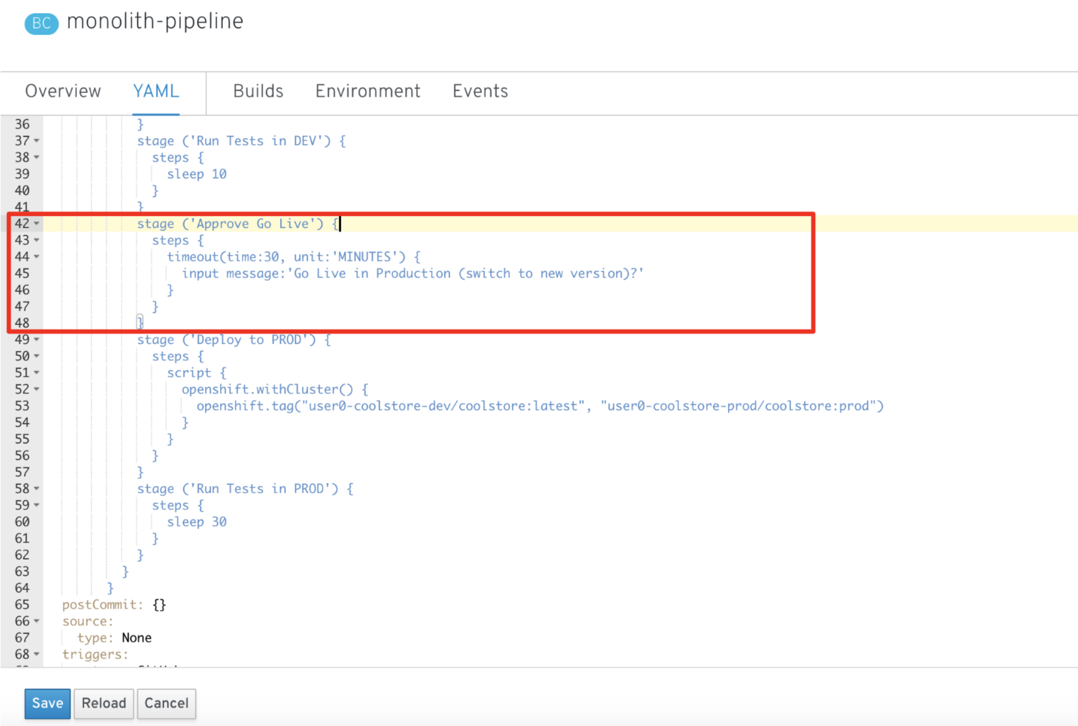


Click on **YAML** tab and add *a new stage* to the pipeline, just before the *Deploy to PROD* stage:

> 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:



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:

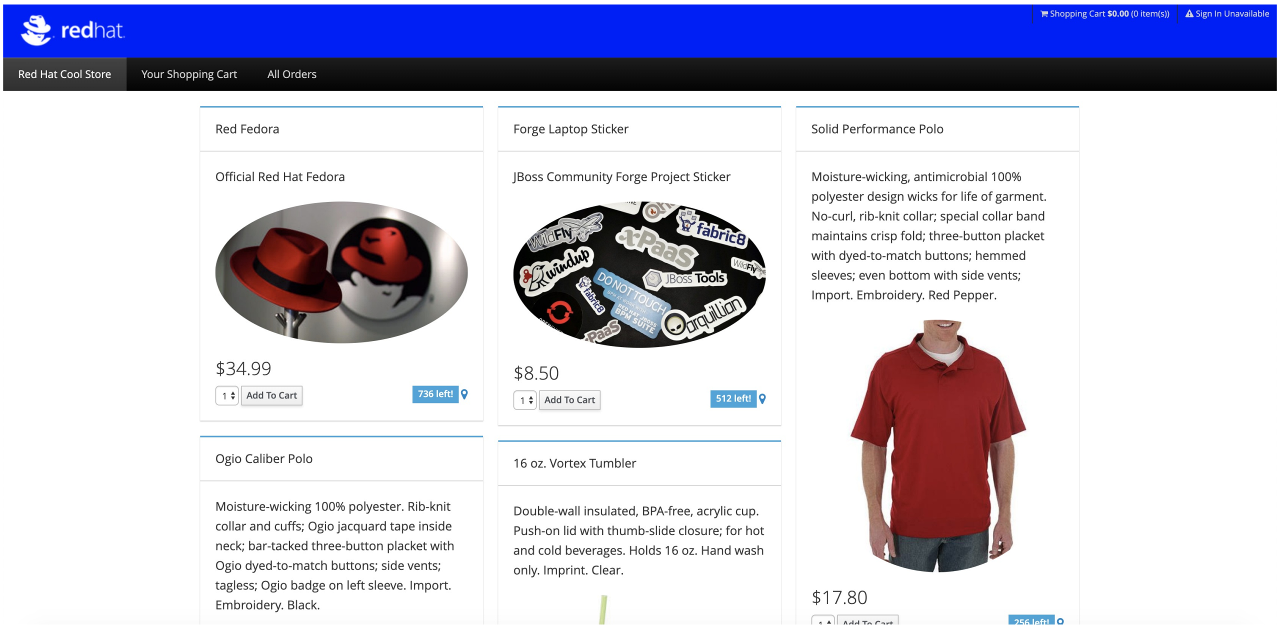
```
oc start-build -n userXX-coolstore-dev coolstore --from-file=deployments/ROOT.war --follow
```

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

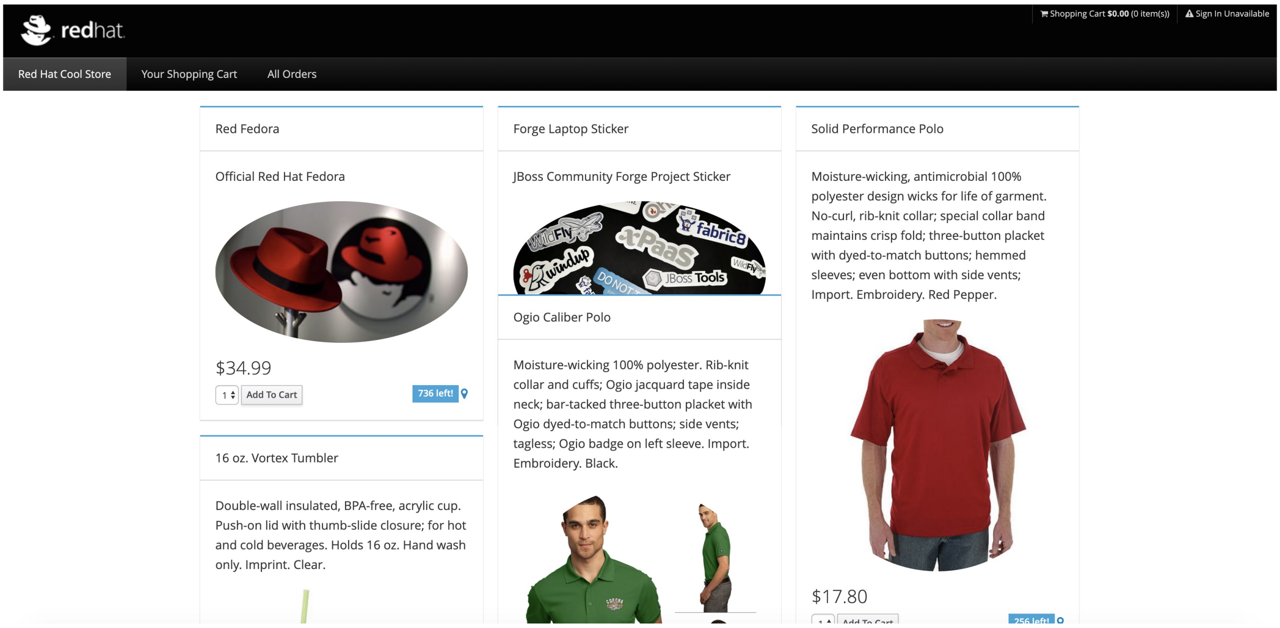
```
oc -n userXX-coolstore-dev rollout status -w dc/coolstore
```

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.



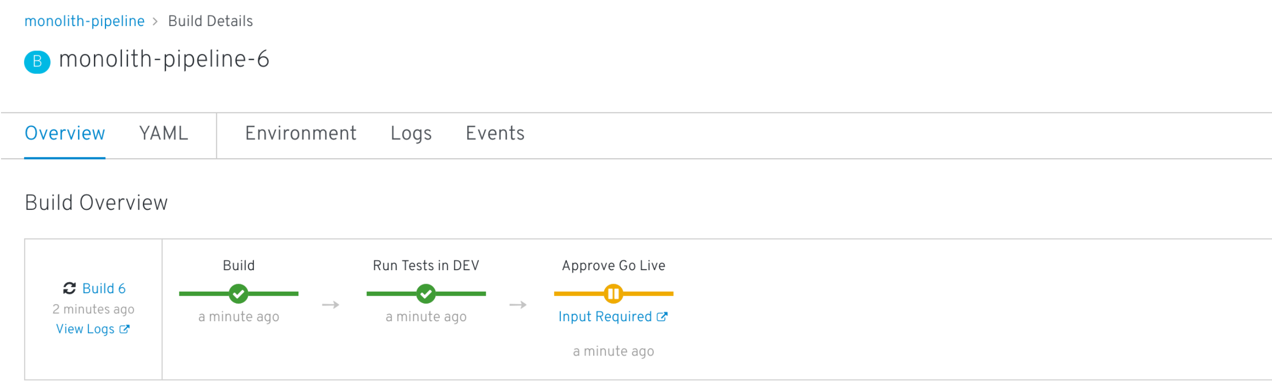
Then navigating to the `userXX-coolstore-prod` project in the OpenShift Console, and then going to *Networking* > *Routes* and clicking on the route URL for the production app. It should still be black:



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:



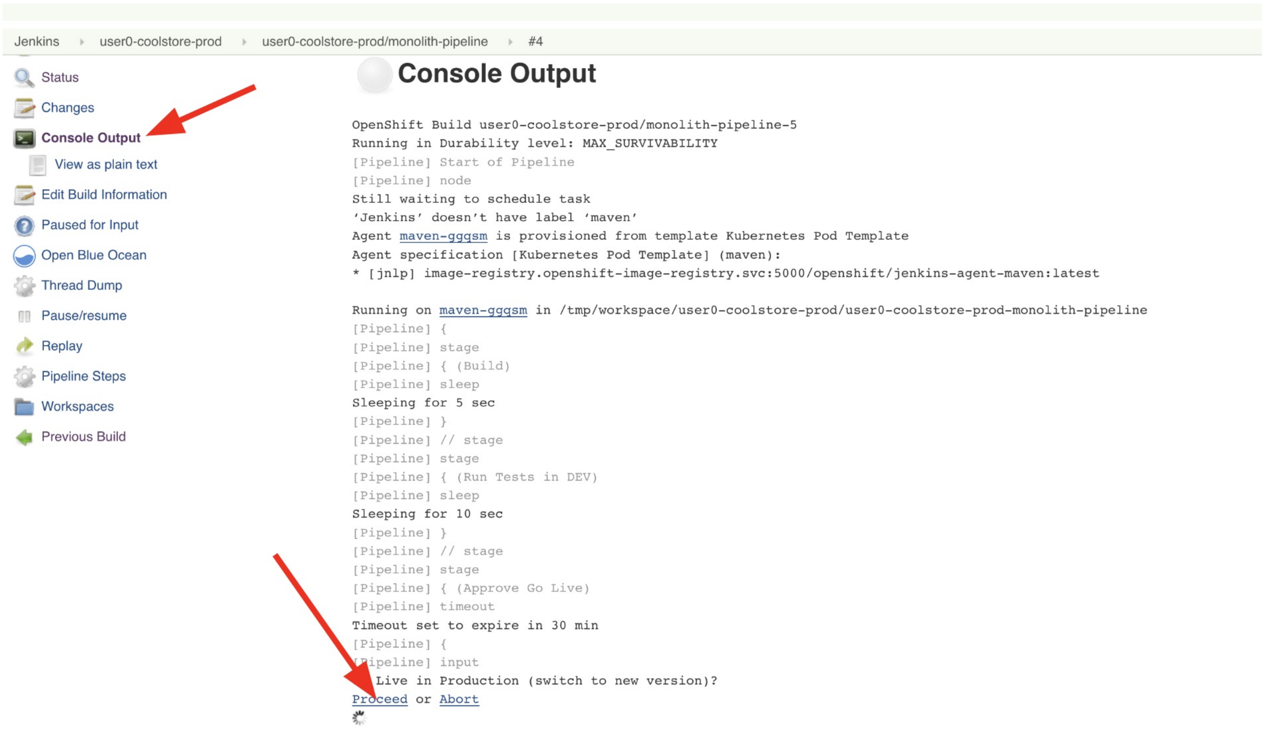
.

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`.



8. Approve the change to go live

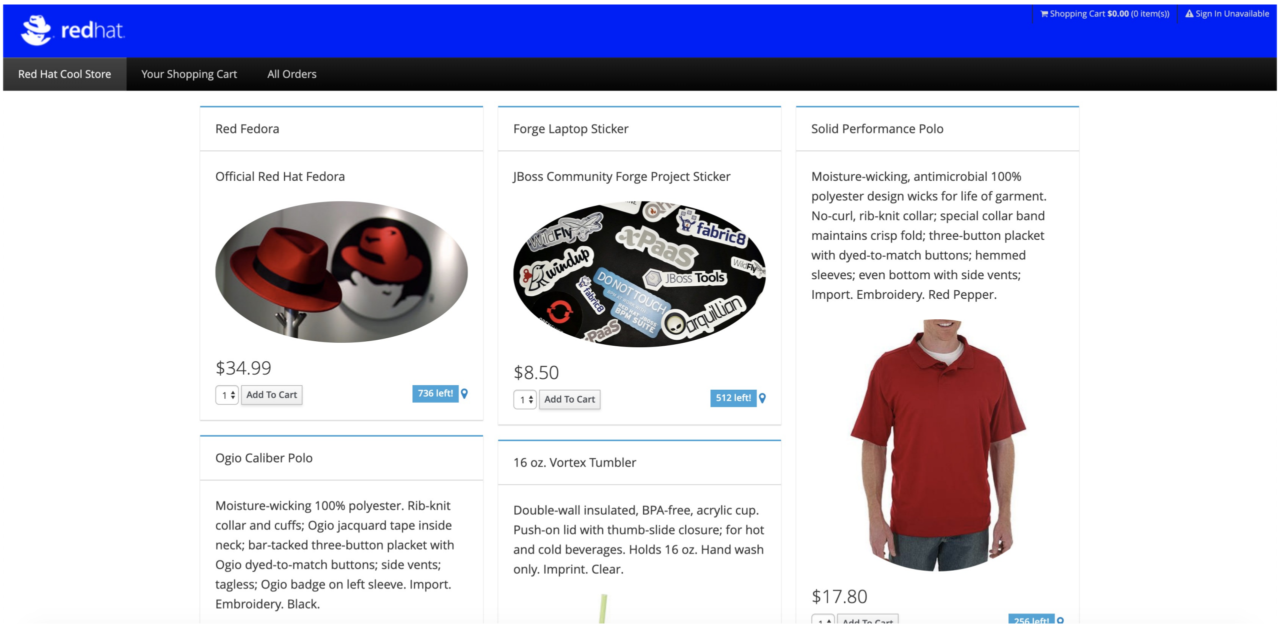
Click **Proceed**, which will approve the change to be pushed to production. You could also have clicked **Abort** which would stop the pipeline immediately in case the change was unwanted or unapproved.

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.

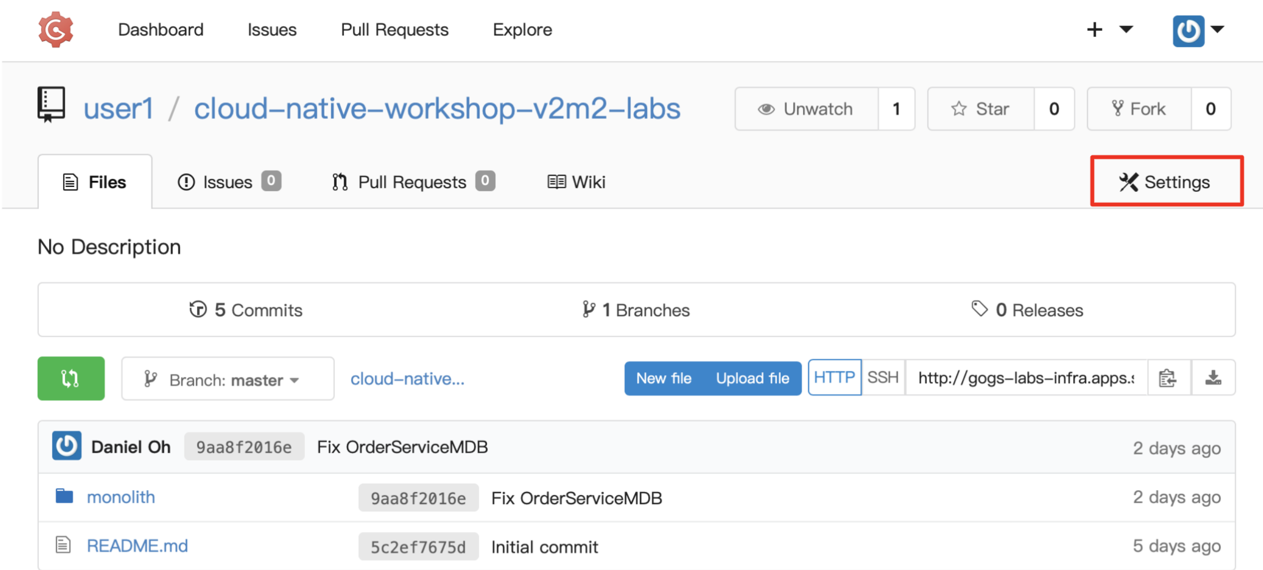
You can get see the webhook links in the OpenShift web console by going to *Builds* > *Build Configs* > *monolith-pipeline*. Look for the *Generic secret* value in the *YAML* tab. Copy this down.

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**.

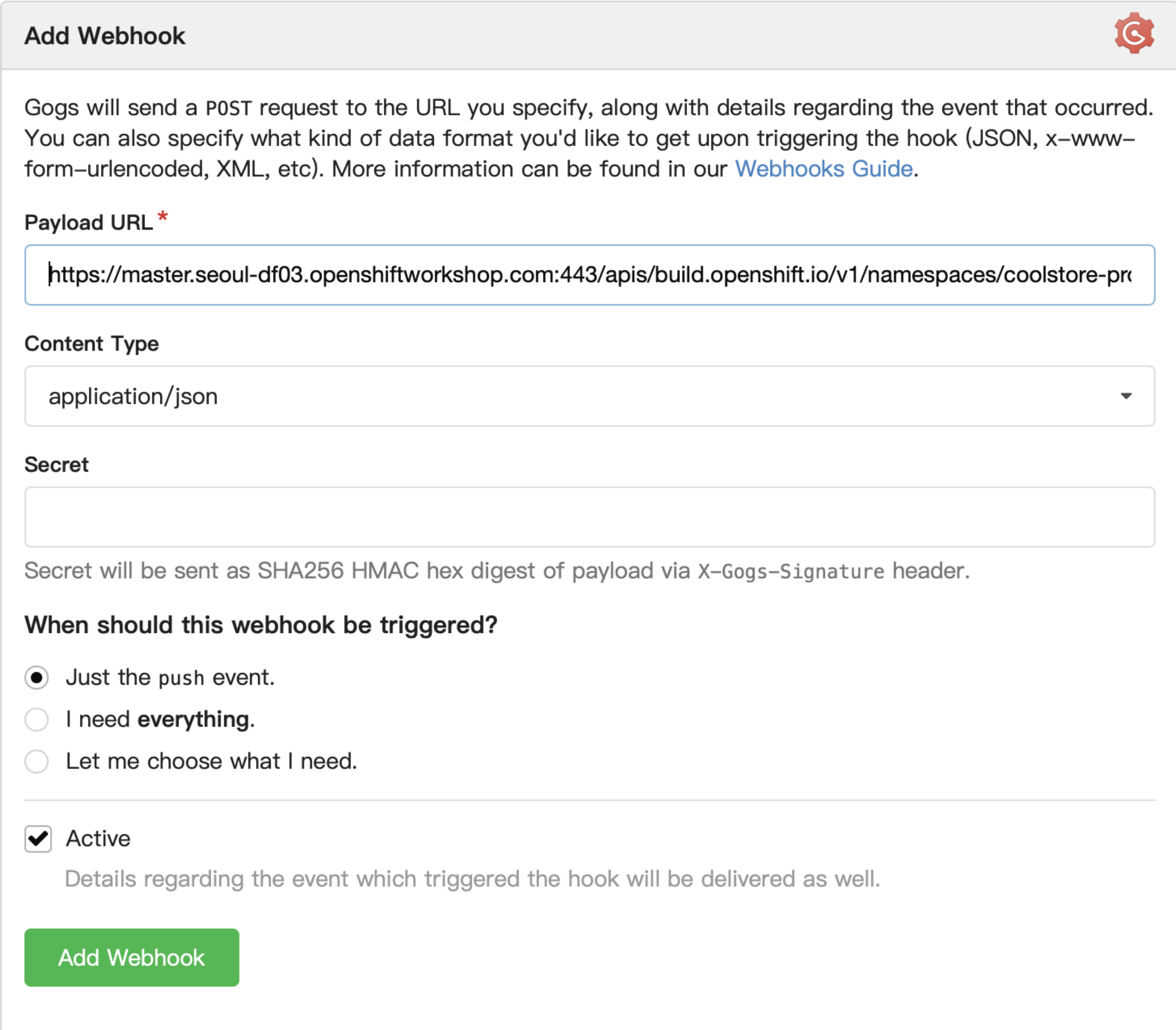


On the left menu, click on **Webhooks** and then on **Add Webhook** button and then **Gogs**.

Create a webhook with the following details:

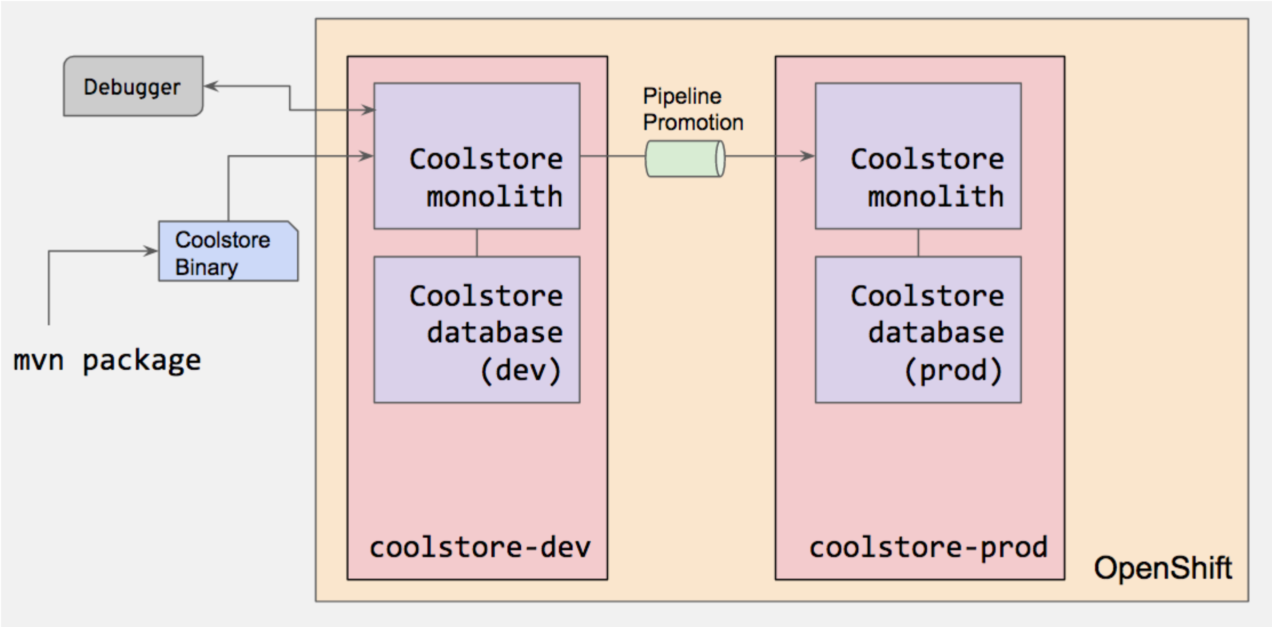
- Payload URL: paste the Generic webhook url you copied from the **monolith-pipeline** (make sure to replace the `secret` value in the URL!)
- Content type: **application/json**

Click on **Add Webhook**.



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 your 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.

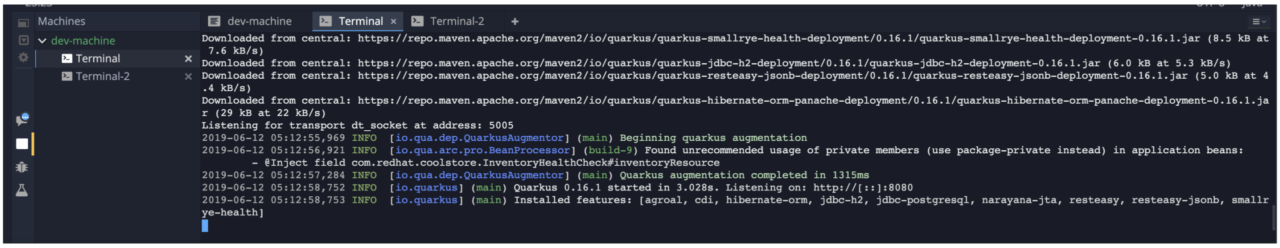
Enable remote debugging on Inventory by running the following inside the **inventory** directory (*/projects/cloud-native-workshop-v2m2-labs/inventory*) in the CodeReady Workspaces Terminal window:

`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 you choice.

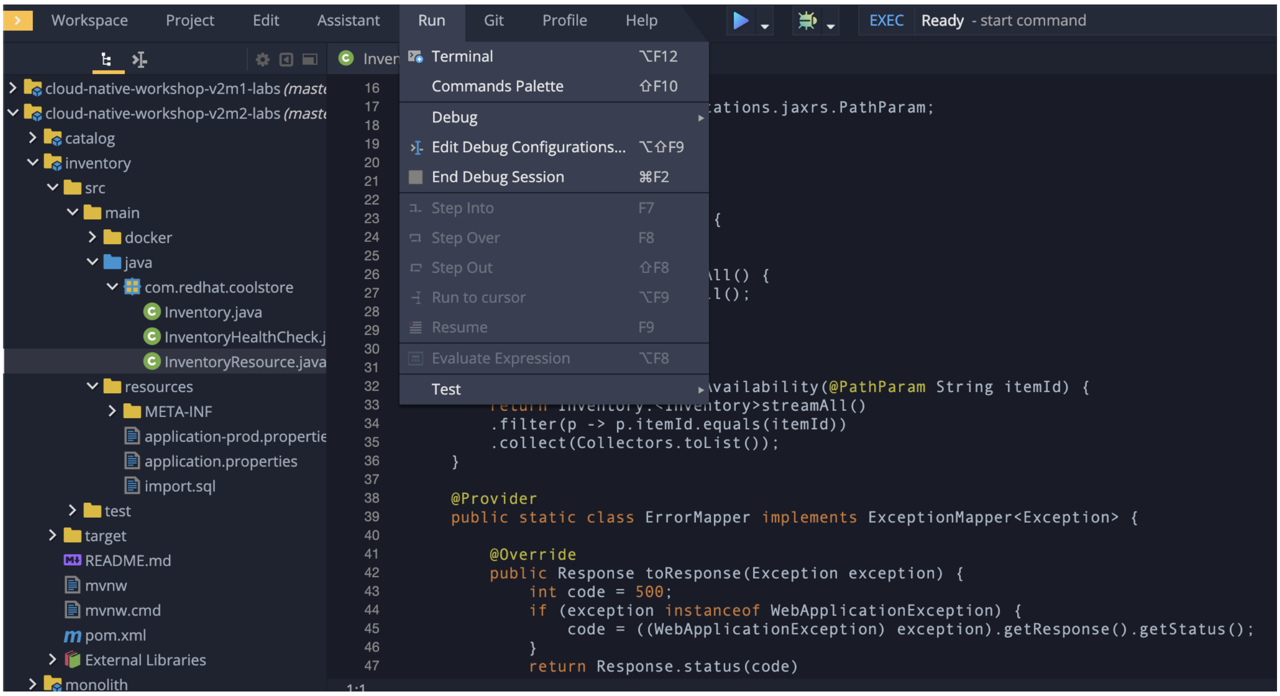
Do not wait for the command to return! The Quarkus maven plugin keeps the forwarded port open so that you can start debugging remotely.



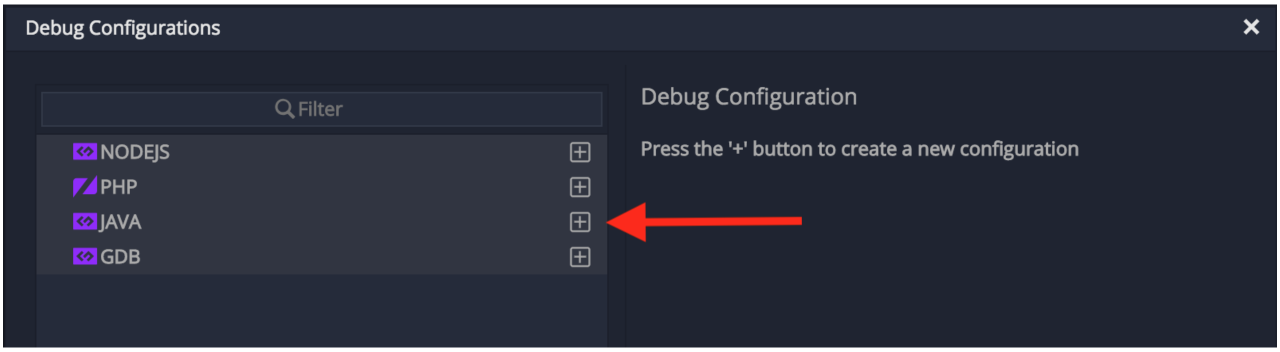
2. Remote Debug with CodeReady Workspace

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…**.

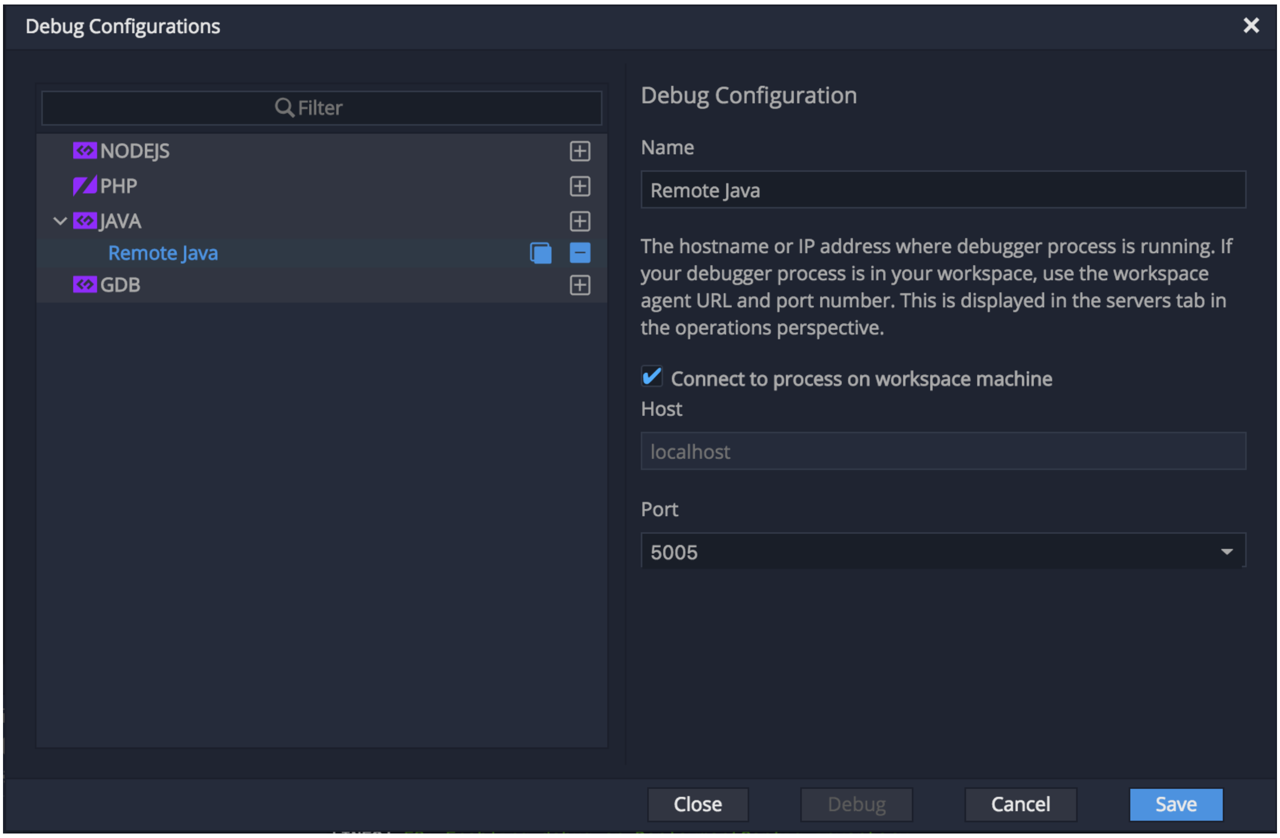


The window shows the debuggers available in CodeReady Workspace. Click on the plus sign near the Java debugger.



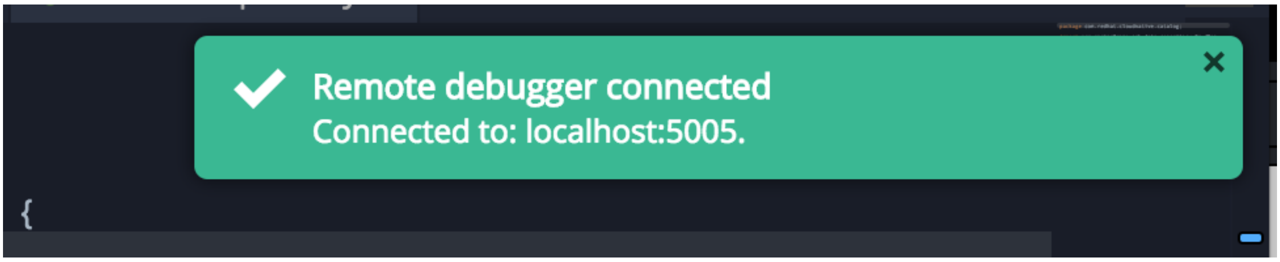
Configure the remote debugger and click on the **Save** button:

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



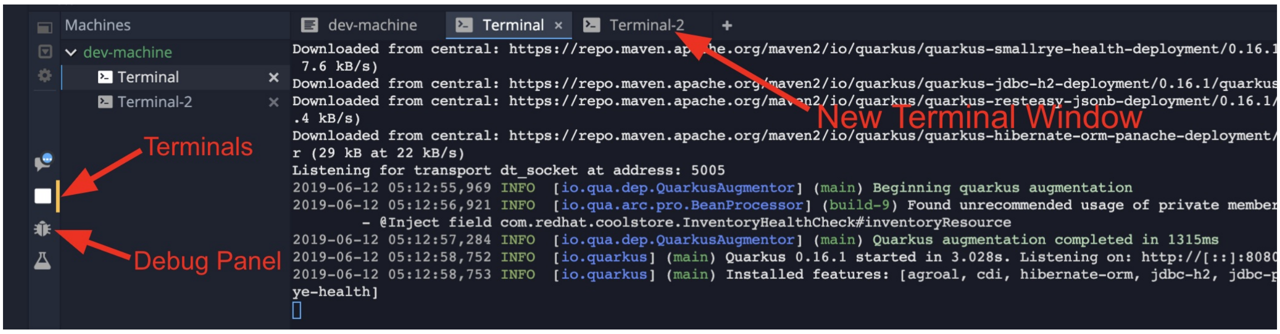
You can now click on *Save* then the *Debug* button to make CodeReady Workspaces connect to the Inventory service running on OpenShift.

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.

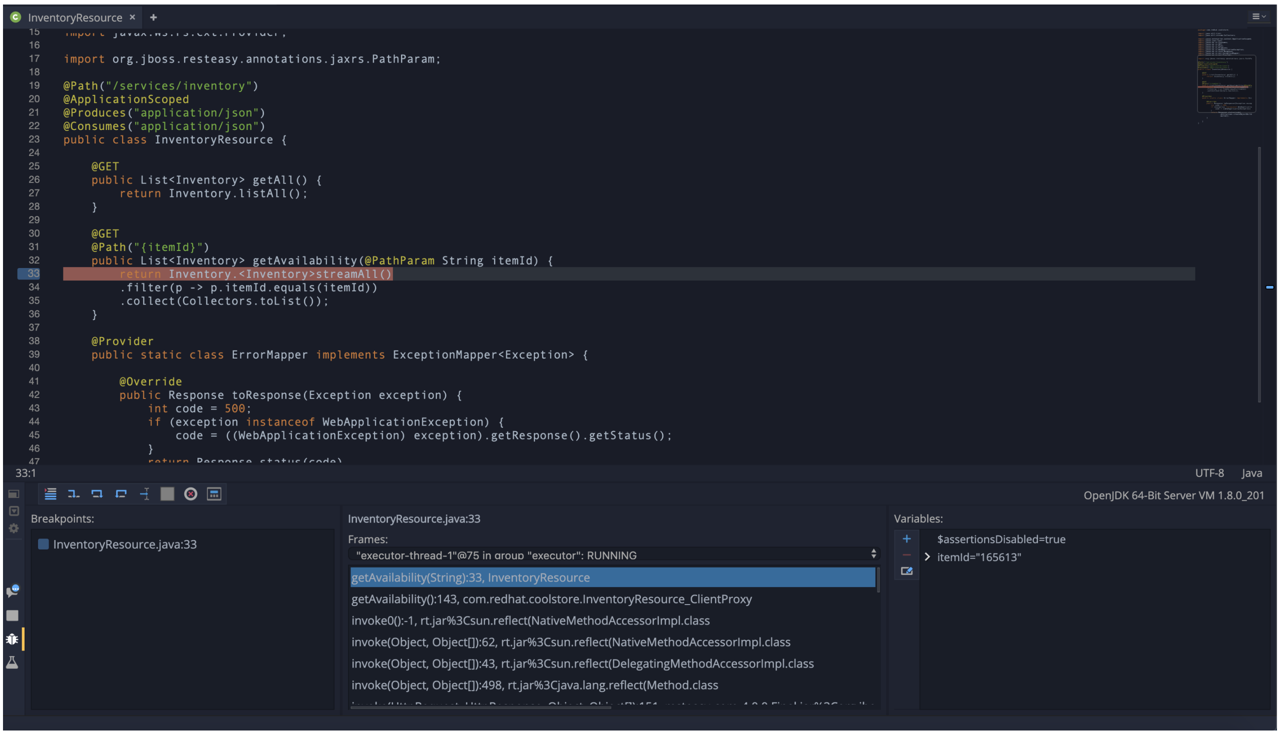
Note that you can use the the following icons to switch between debug and terminal windows.



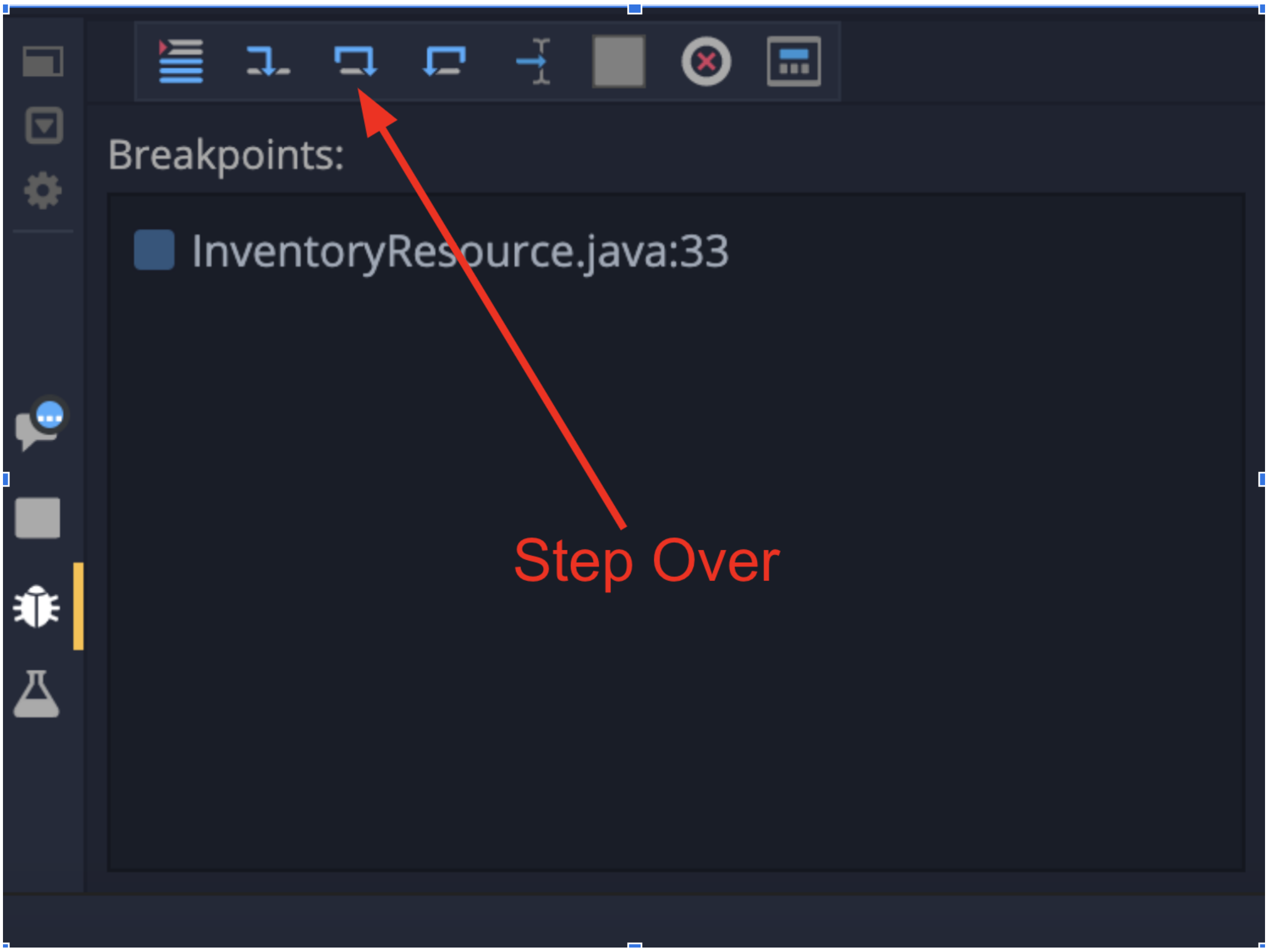
Invoke the endpoint URL of the Inventory service using **curl** command in a separate Terminal:

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

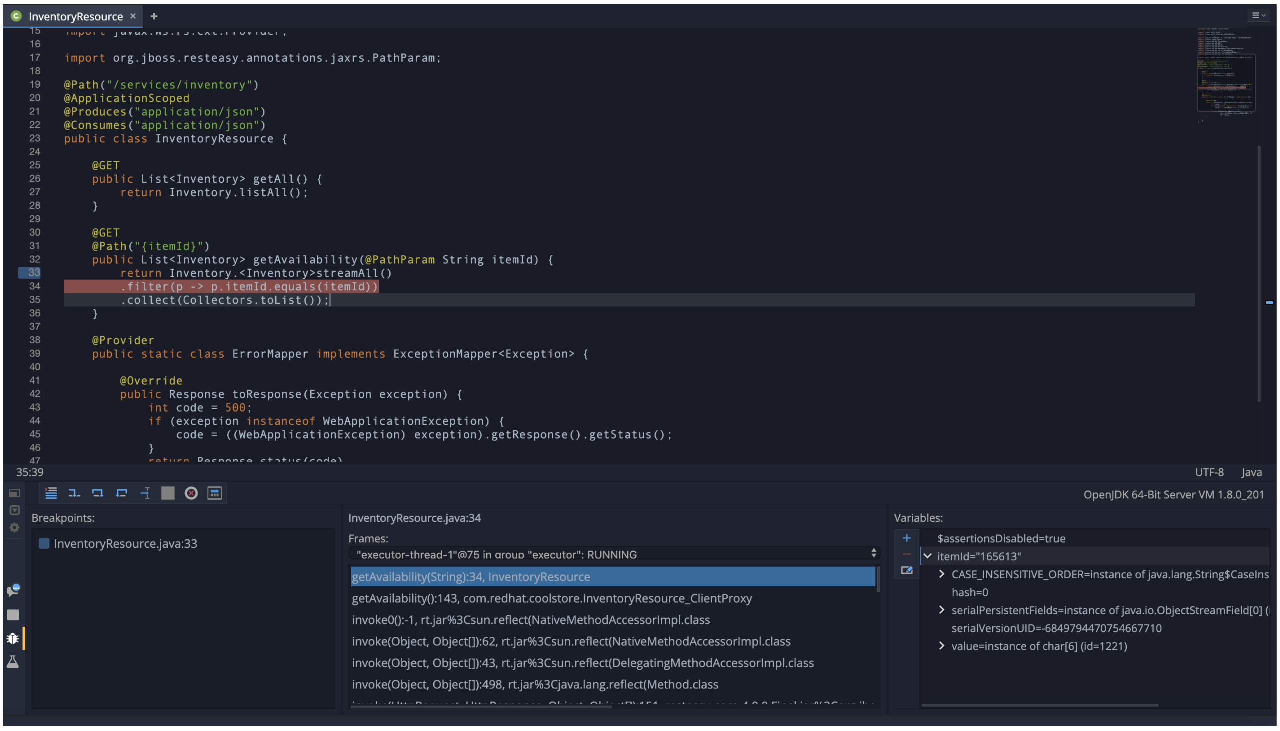
Switch back to the debug panel and notice that the code execution is paused at the breakpoint on *InventoryResource* class.



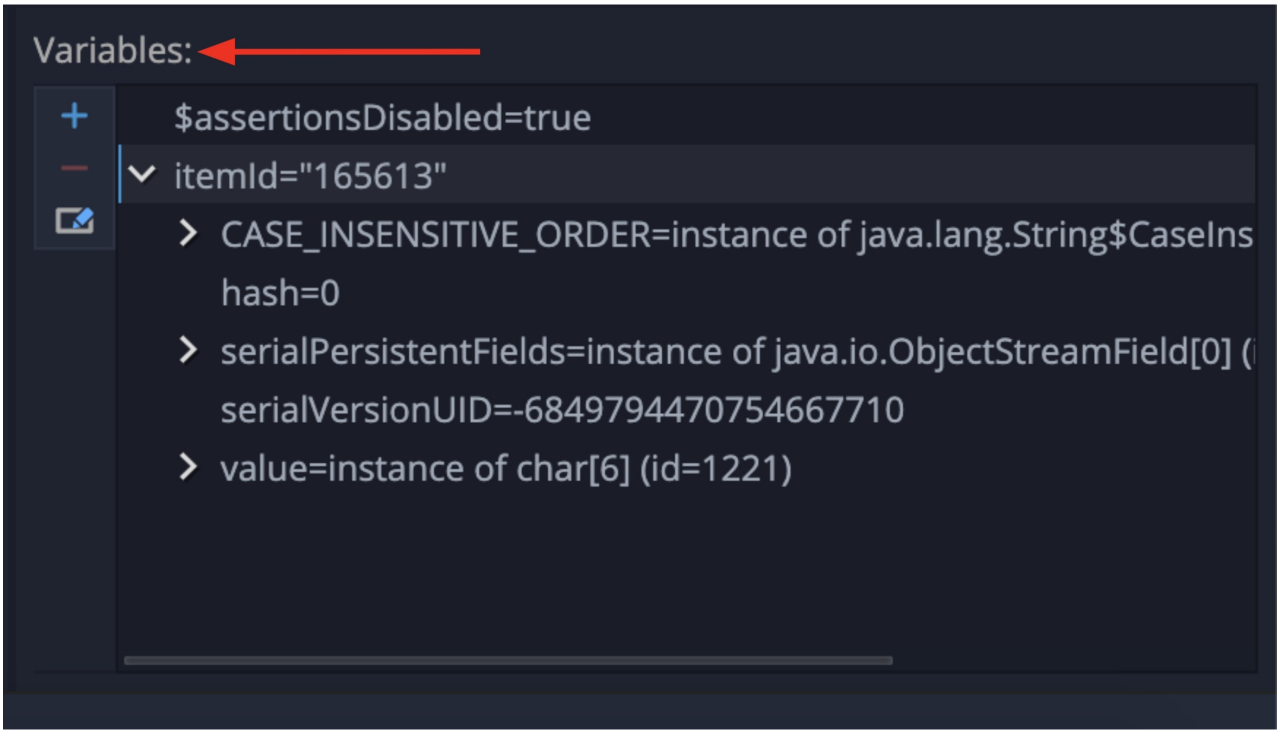
Click on the *Step Over* icon to execute one line and retrieve the inventory object for the given product id from the database.



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



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 swich 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.

> `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

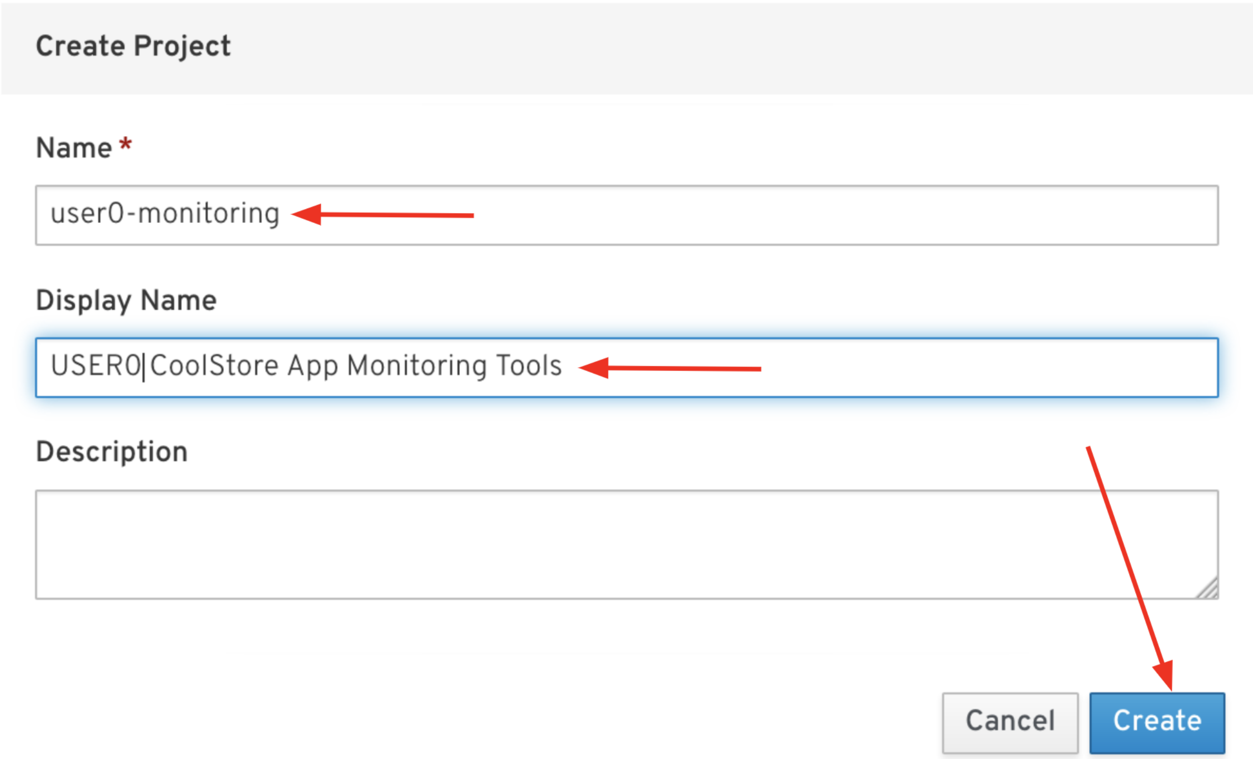
1. Create OpenShift Project

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*



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
```

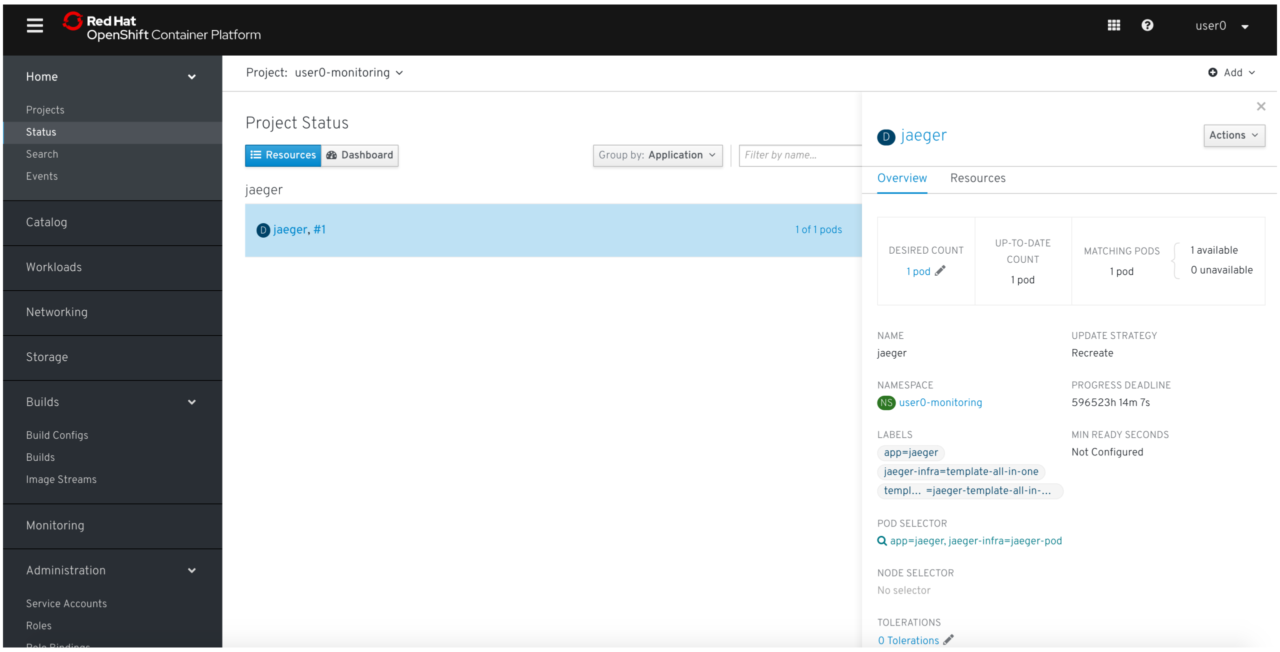
```
oc process -f /projects/cloud-native-workshop-v2m2-labs/monitoring/jaeger-all-in-one-template.yml | oc create -f -
```

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

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

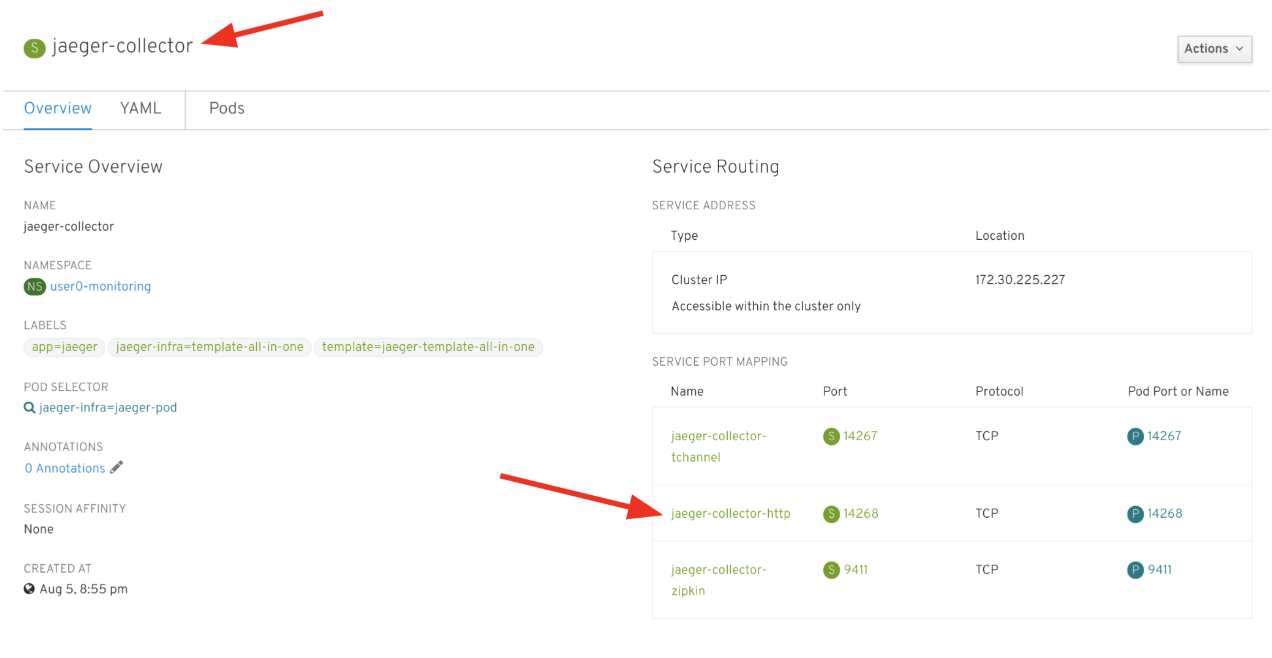
> deployment jaeger successfully rolled out

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



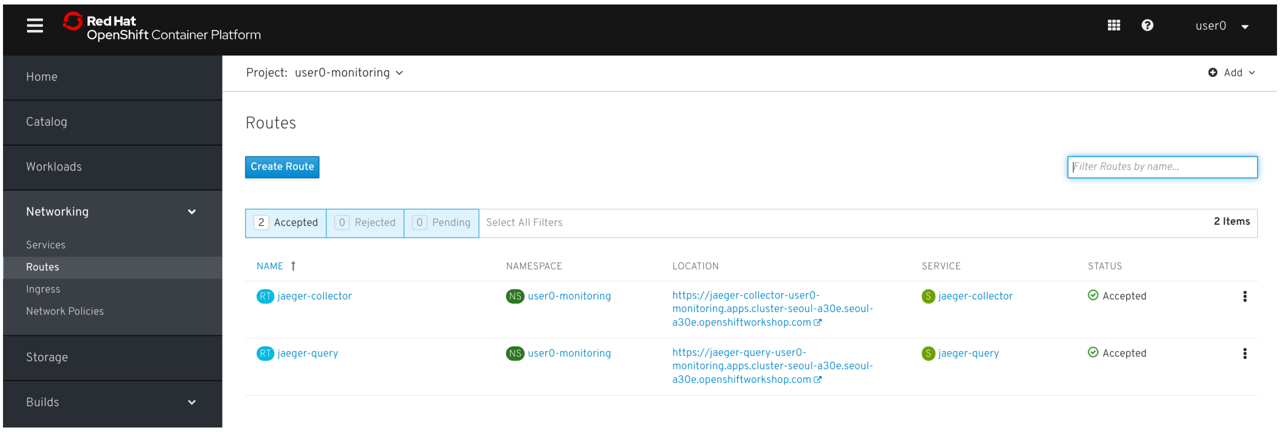
3. Examine Jaeger-Collector

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 Inventoty service later.

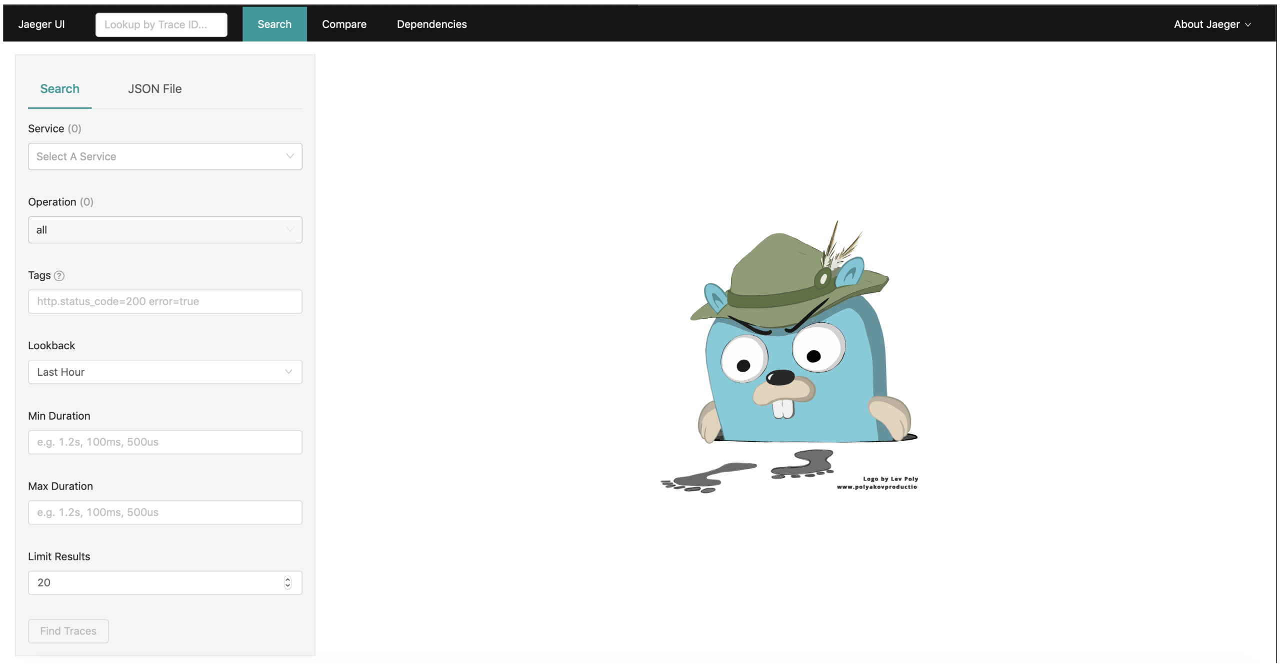


4. Observe Jaeger UI

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



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.



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.

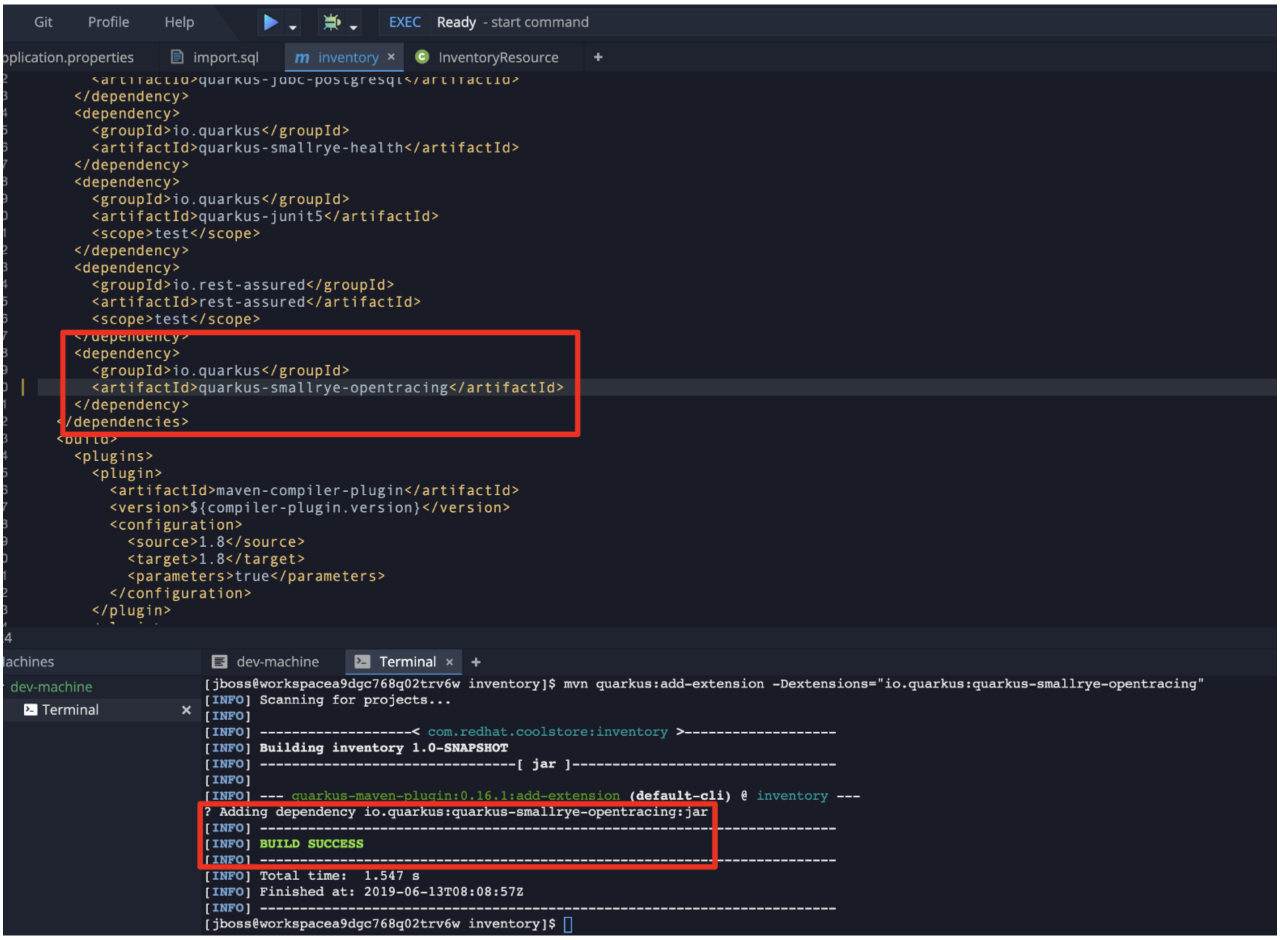
In this step, we will add Qurakus extensions to the Inventory application for using **smallrye-opentracing**. Copy the following commands to add the tracing extension via CodeReady Workspaces Terminal:

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**.



> 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.

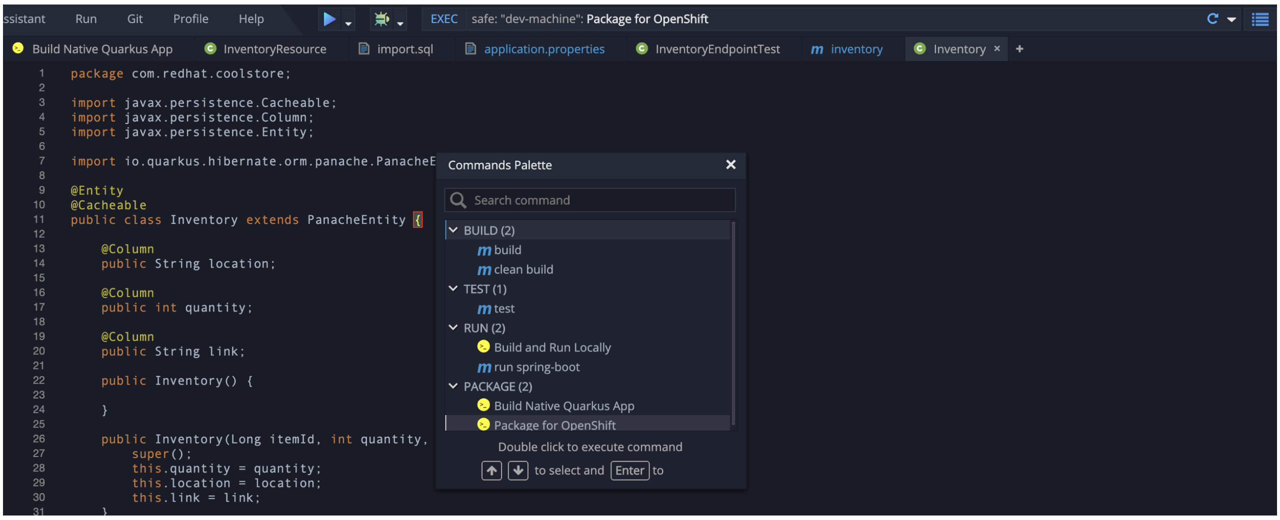
> If the `quarkus.jaeger.service-name` property (or `JAEGER_SERVICE_NAME` environment variable) is not provided then a "no-op" tracer will be configured, resulting in no tracing data being reported to the backend.

Currently the tracer can only be configured to report spans directly to the collector via HTTP, using the `quarkus.jaeger.endpoint` property (or `JAEGER_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:

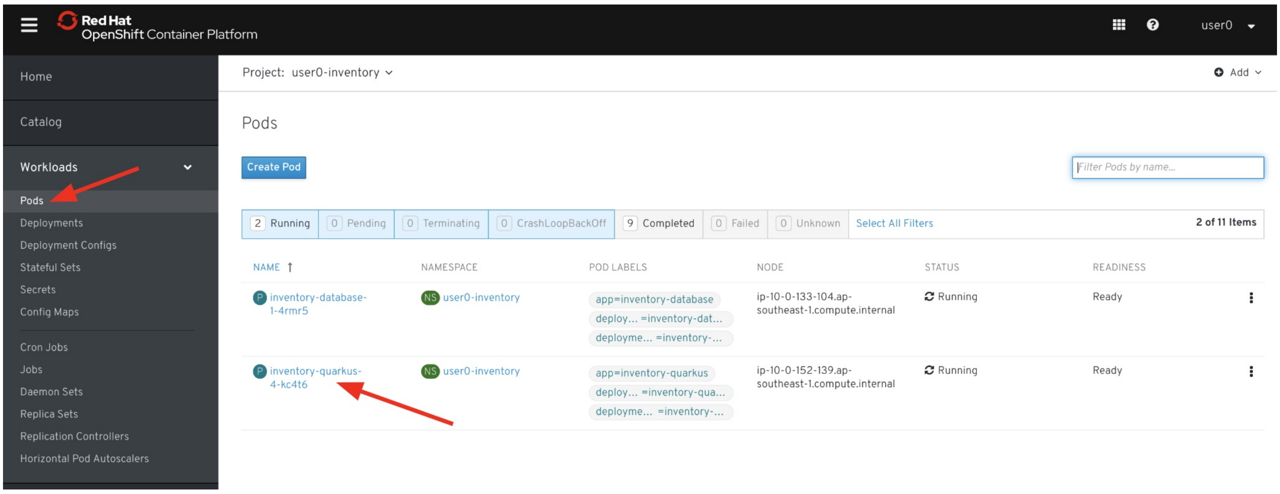
`replication controller "inventory-quarkus-XX" successfully rolled out`

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.

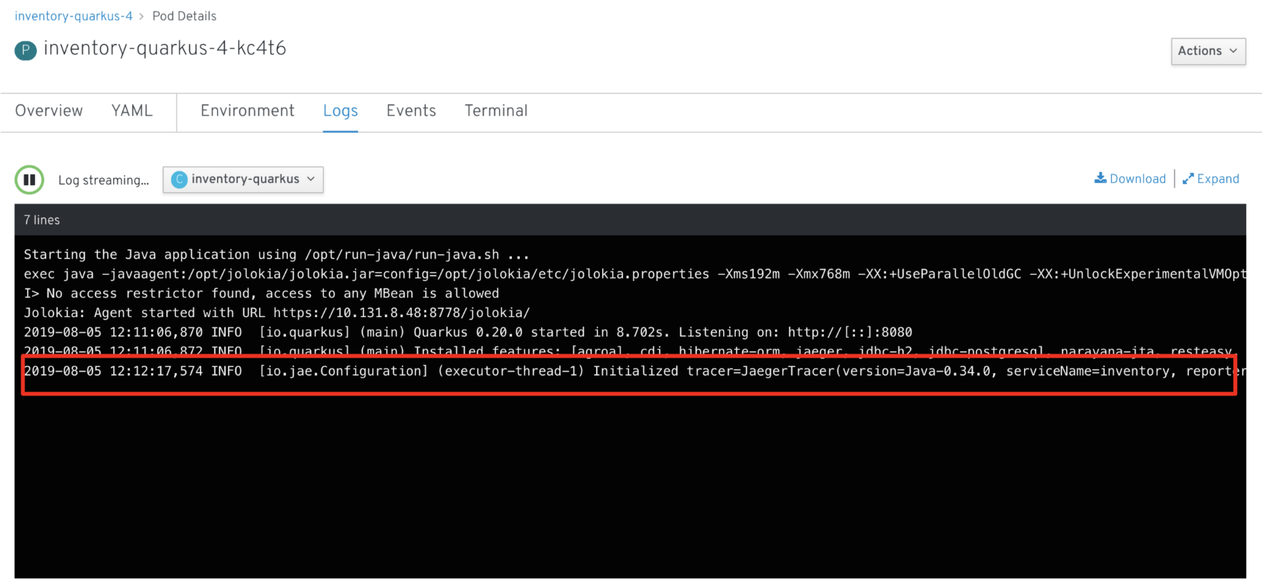
`curl http://$(oc get route inventory-quarkus -o=go-template --template='{{ .spec.host }}')/services/inventory/165613 ; echo`

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



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

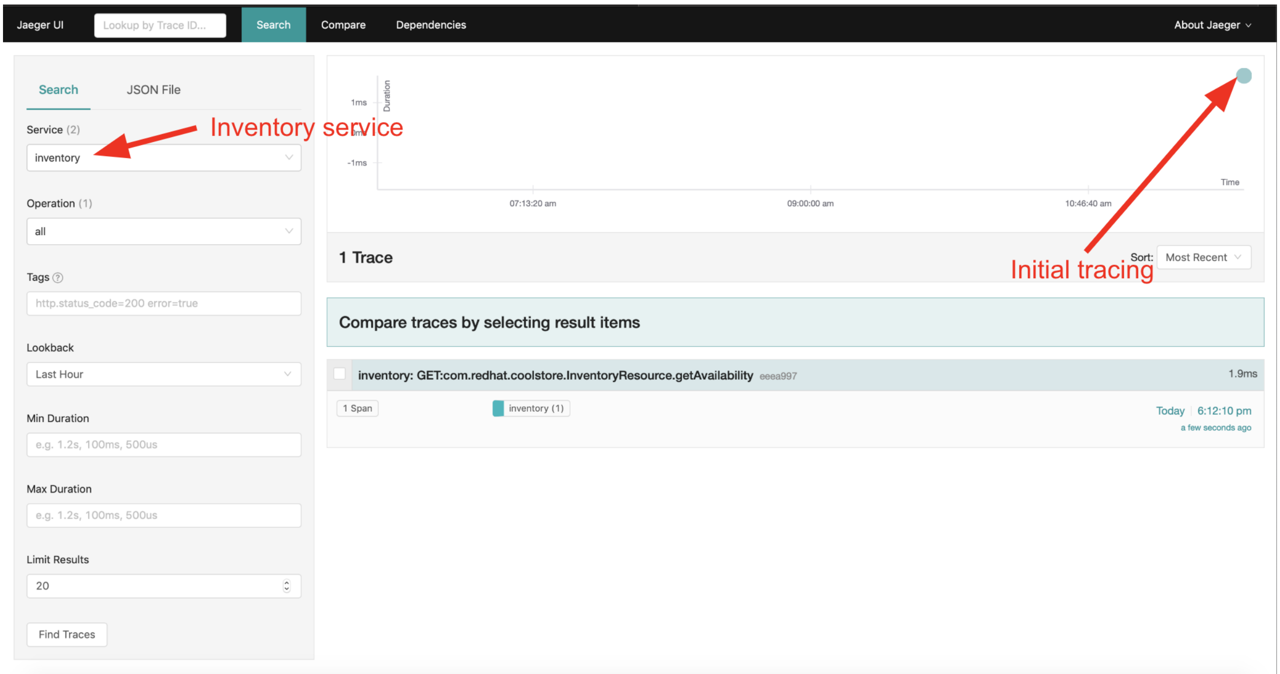
2019-08-05 12:12:17,574 INFO [io.jae.Configuration] (executor-thread-1) Initialized tracer=JaegerTracer(version=Java-0.34.0, serviceName=inventory, reporter=RemoteReporter(sender=HttpSender(), closeEnqueueTimeout=1000), sampler=ConstSampler(decision=true, tags={sampler.type=const, sampler.param=true}), tags={hostname=inventory-quarkus-4-kc4t6, jaeger.version=Java-0.34.0, ip=10.131.8.48}, zipkinSharedRpcSpan=false, expandExceptionLogs=false, useTraceId128Bit=false)



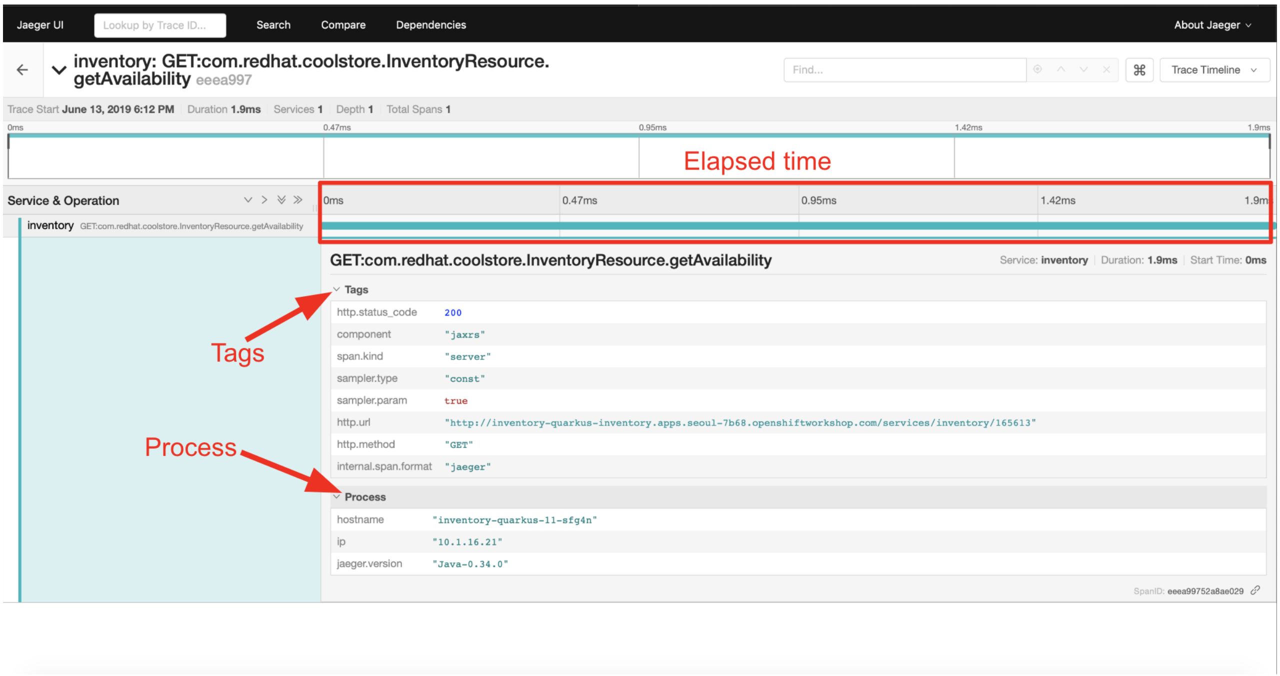
Now, reload the Jaeger UI then you will find that 2 services are created as here:

- inventory
- jaeger-query

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

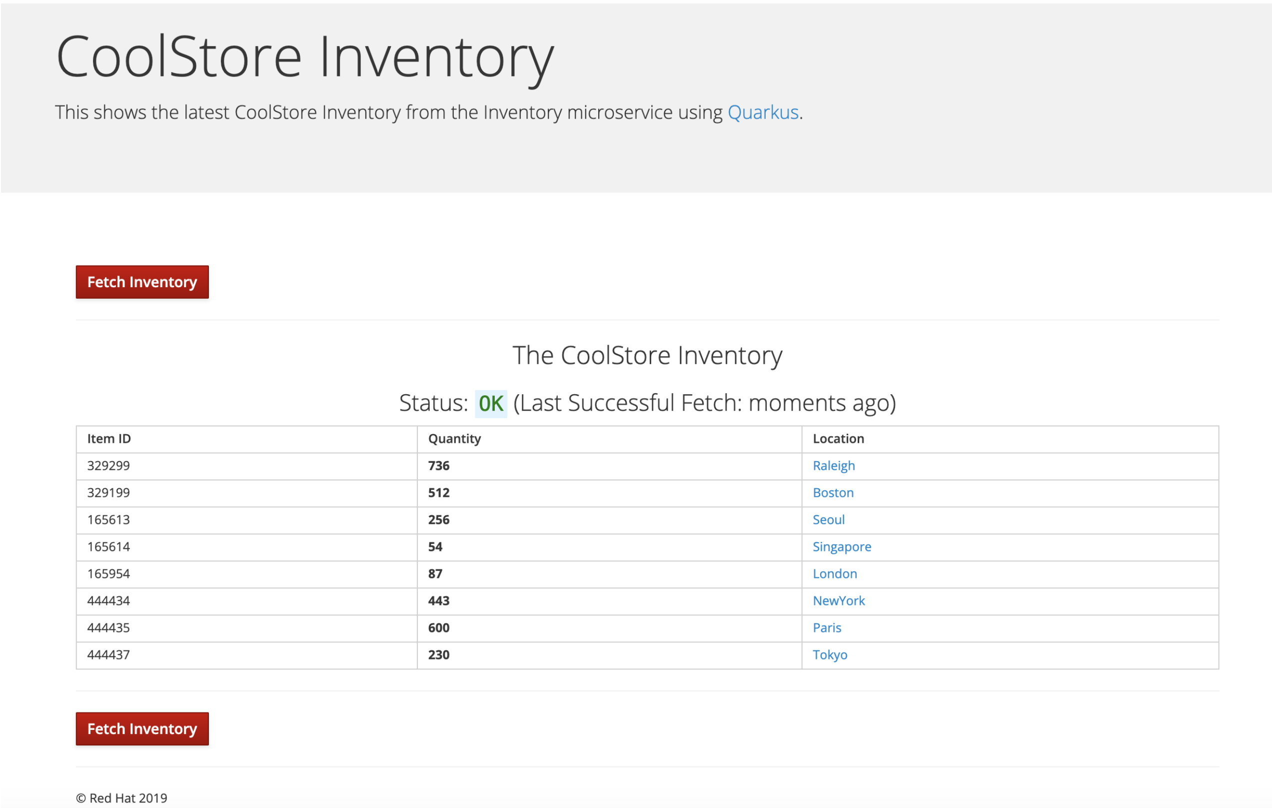


If you click on **Span** and you will see a logical unit of work in Jaeger that has an operation name, the start time of the operation, and the duration. Spans may be nested and ordered to model causal relationships:

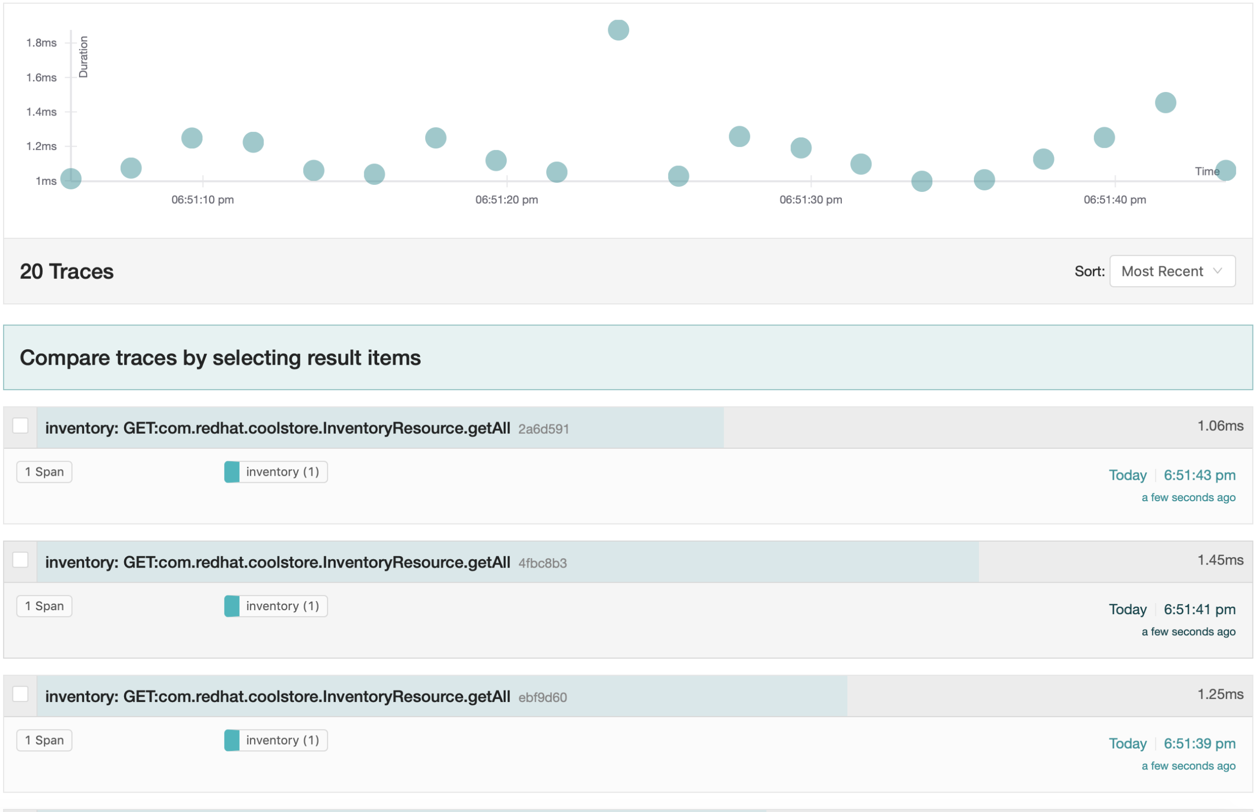


Let's make more traces! Open a new web browser to access *CoolStore Inventory Page* using its route (on *Networking* > *Routes* in the OpenShift Console):

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

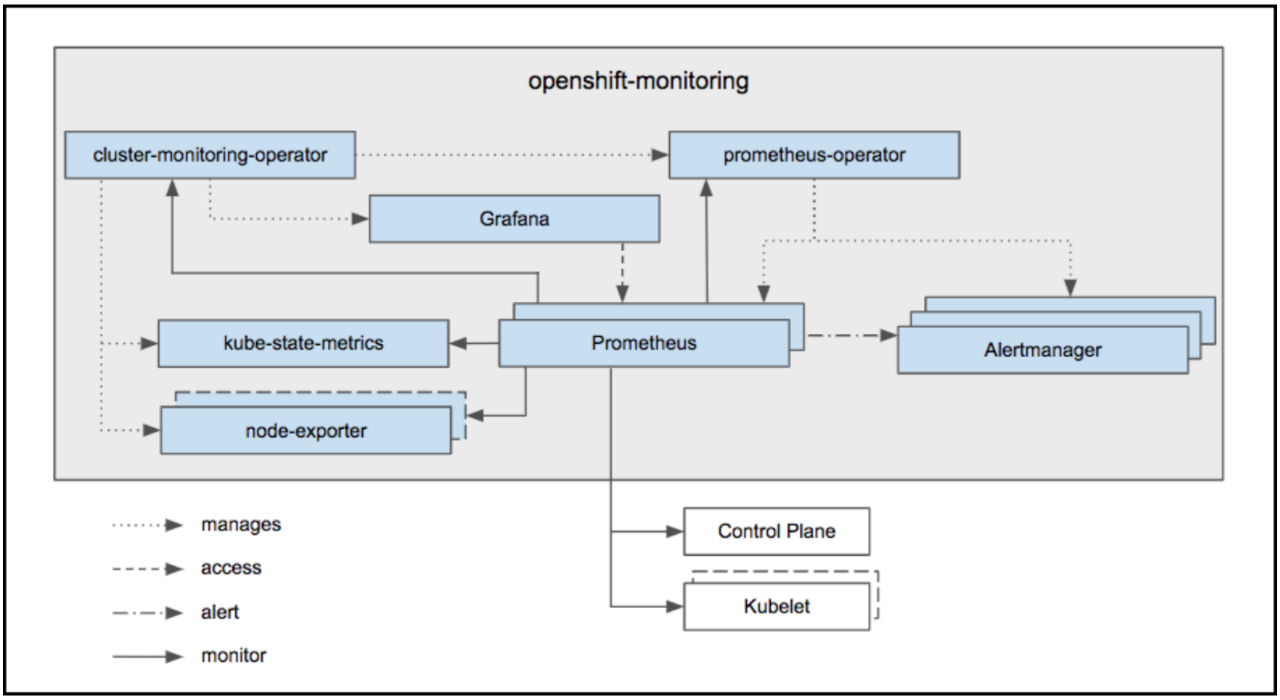


Go back to *Jaeger UI* then click on *Find Traces*. You will see dozens of traces because the Inventory page continues to calling the endpoint of Inventory service in every 2 seconds like we called via **curl** command:



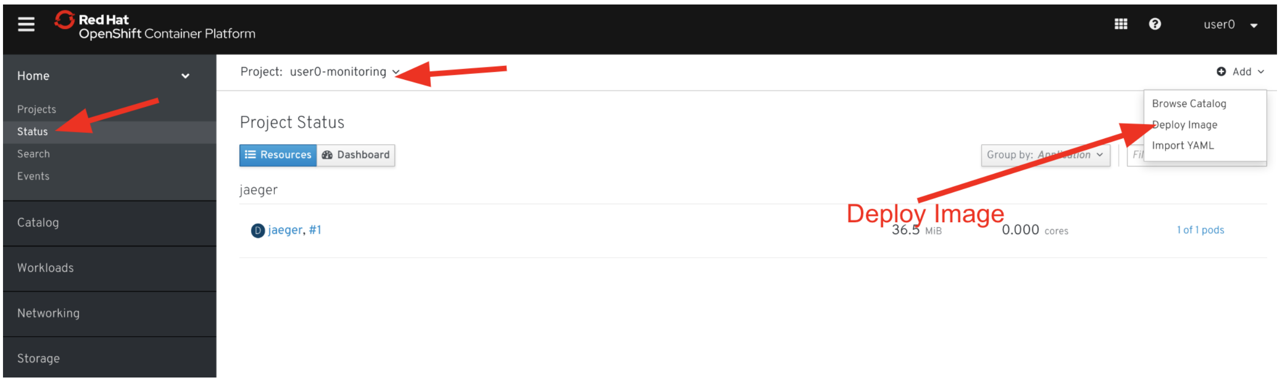
9. Deploy Prometheus and Grafana to OpenShift

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.

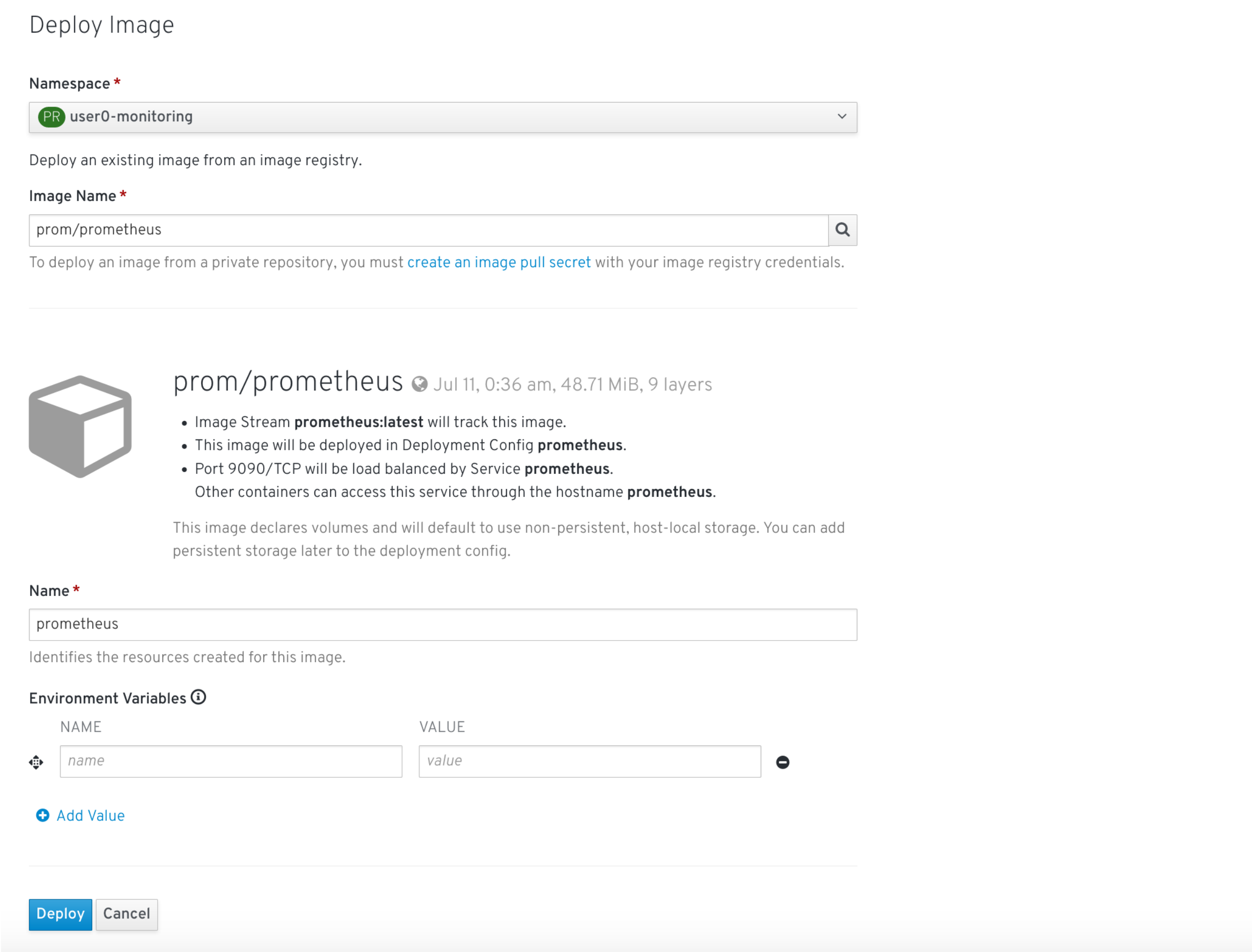


However, we will deploy custom **Prometheus** to scrape services metrics of Inventory and Catalog applications. Then we will visualize the metrics data via custom **Grafana** dashboards deployment.

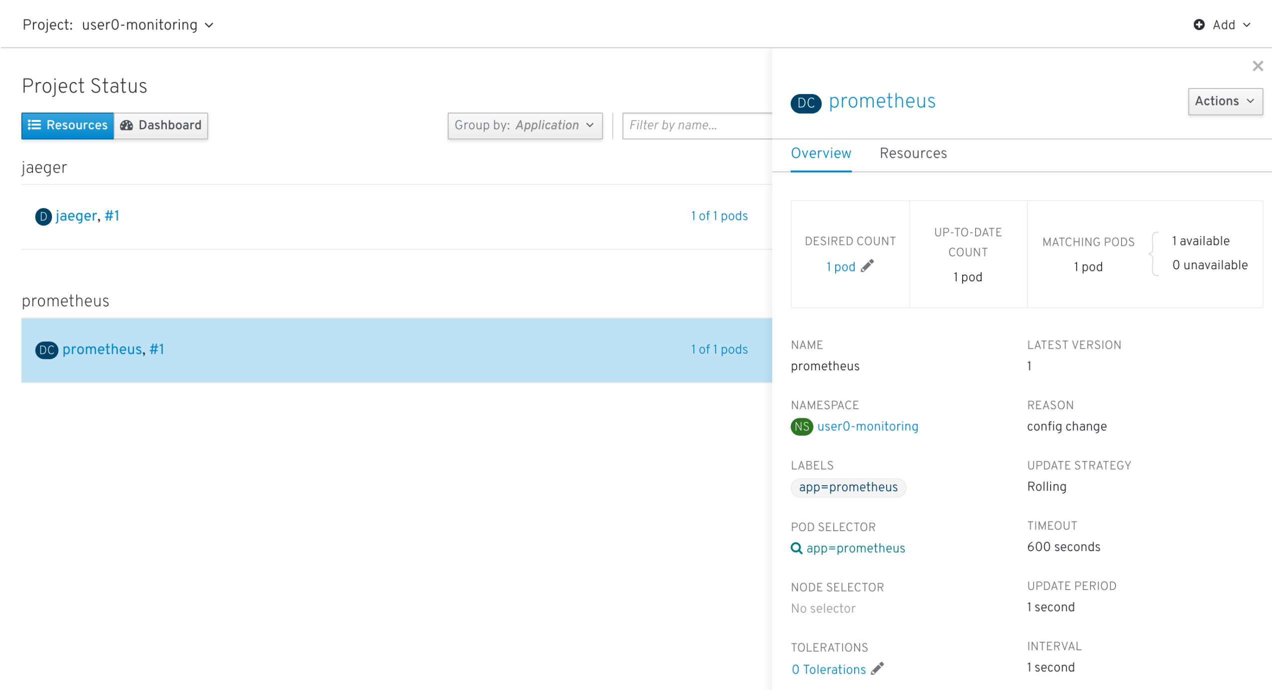
Go to *Project Status* page in *userXX-monitoring* project and click on **Deploy Image** under *Add* on the right top menu:



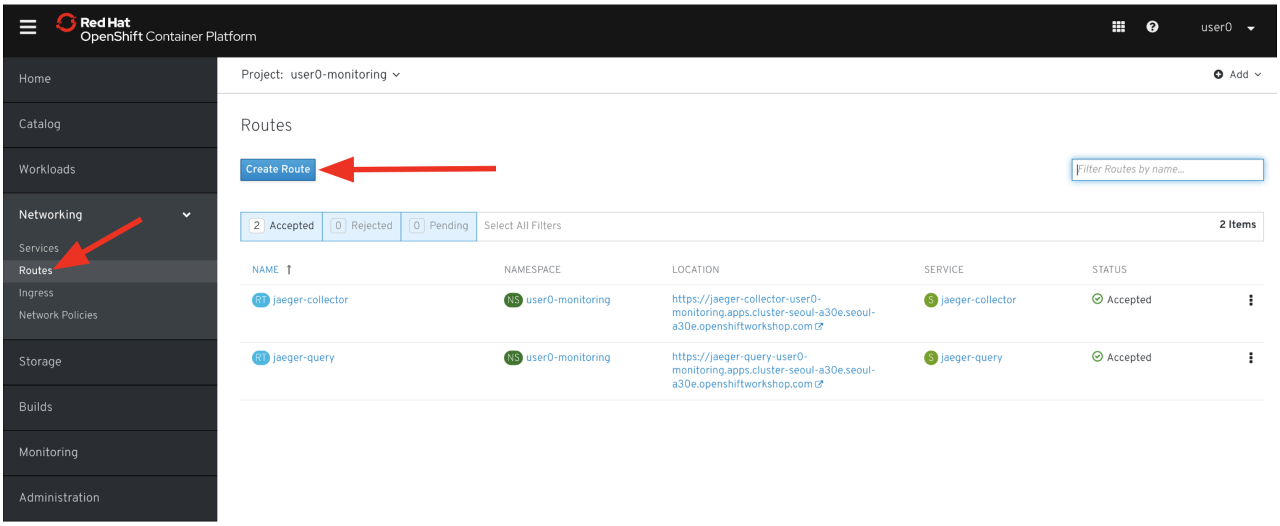
Select **Image Name** and input *prom/prometheus* to search the Prometheus container image via clicking on *Search* icon.



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

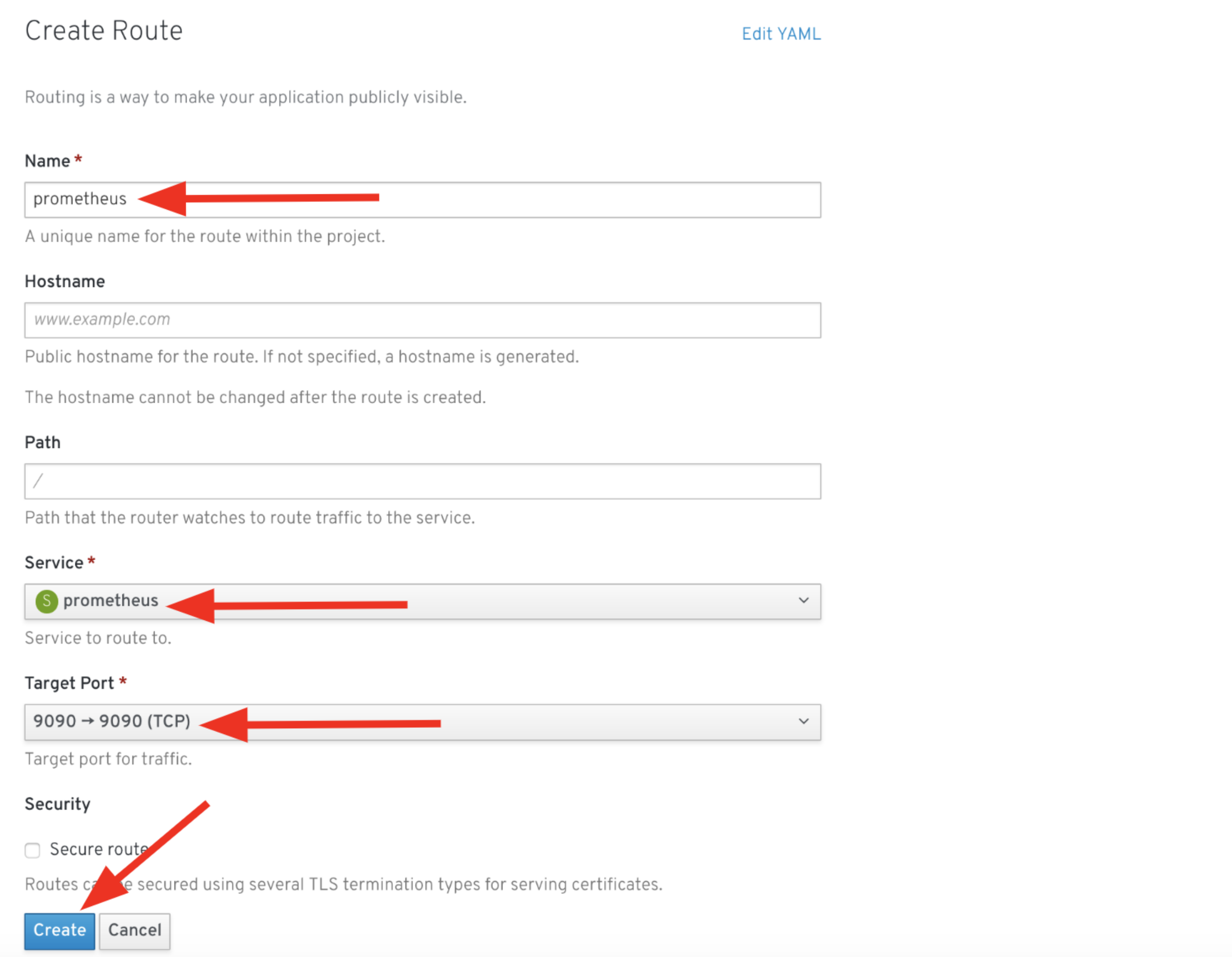


Create the route to access Prometheus web UI. Navigate *Networking* > *Routes* on the left menu and click on **Create Route**.

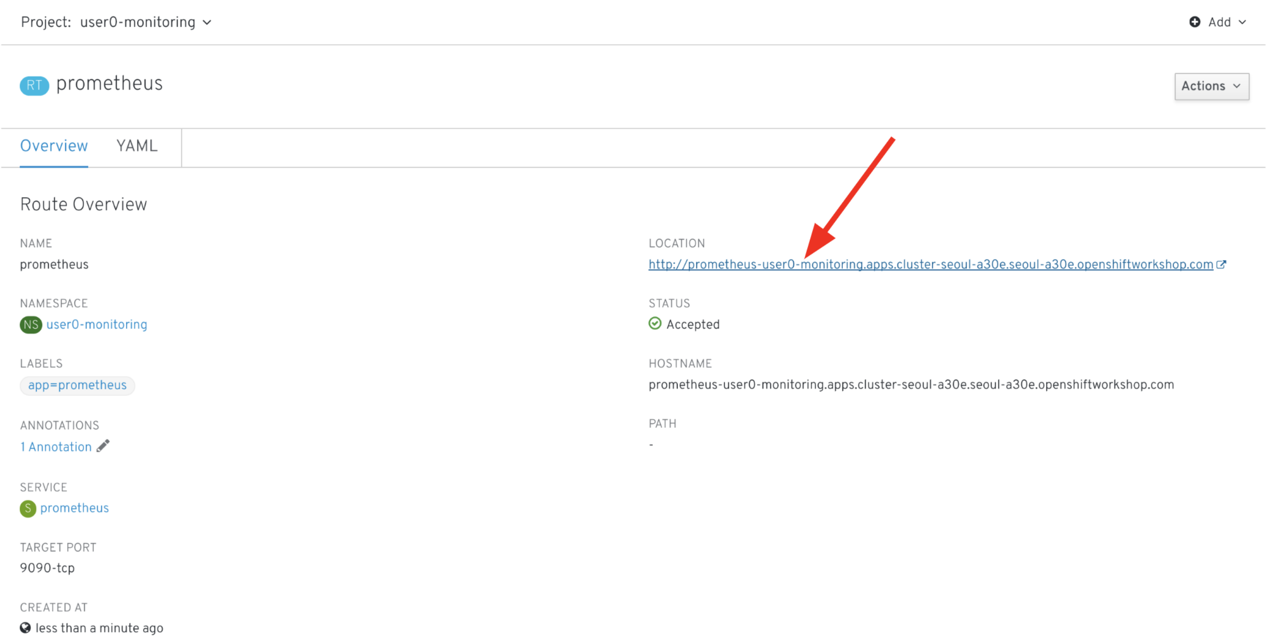


Input the following variables and keep the rest of all default variables. Click on **Create**.

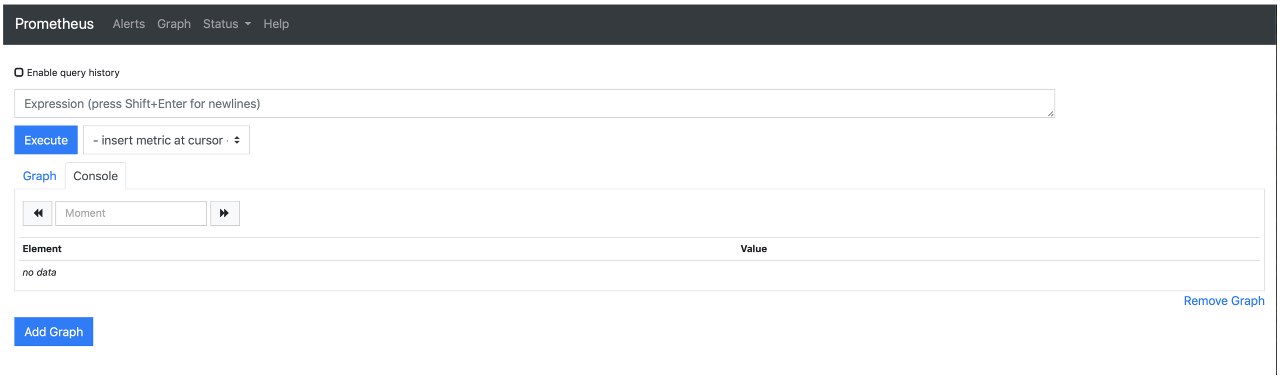
- Name: **prometheus**
- Service: **prometheus**
- Target Port: **9090 -> 9090 (TCP)**



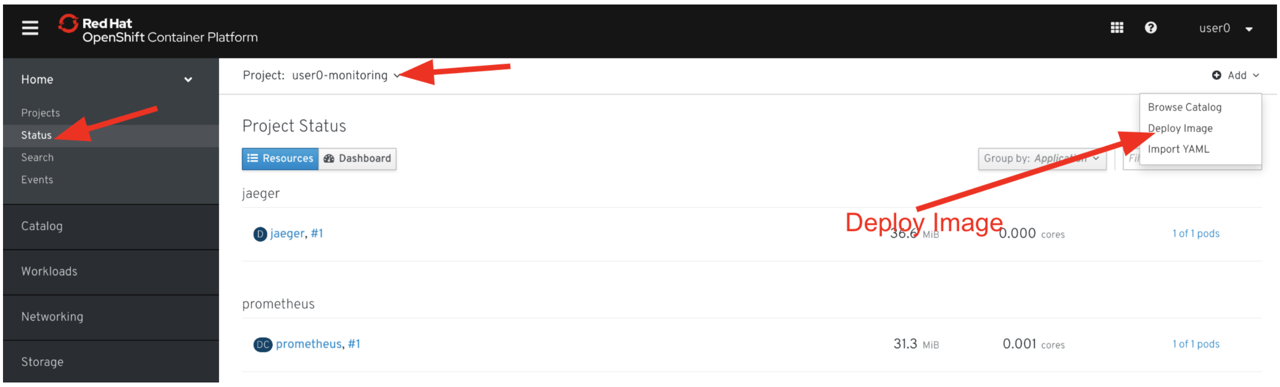
Now, you have the route URL(i.e. *http://prometheus-user0-monitoring.apps.cluster-seoul-a30e.seoul-a30e.openshiftworkshop.com*) as below and click on the URL to make sure if you can access the *Prometheus web UI*.



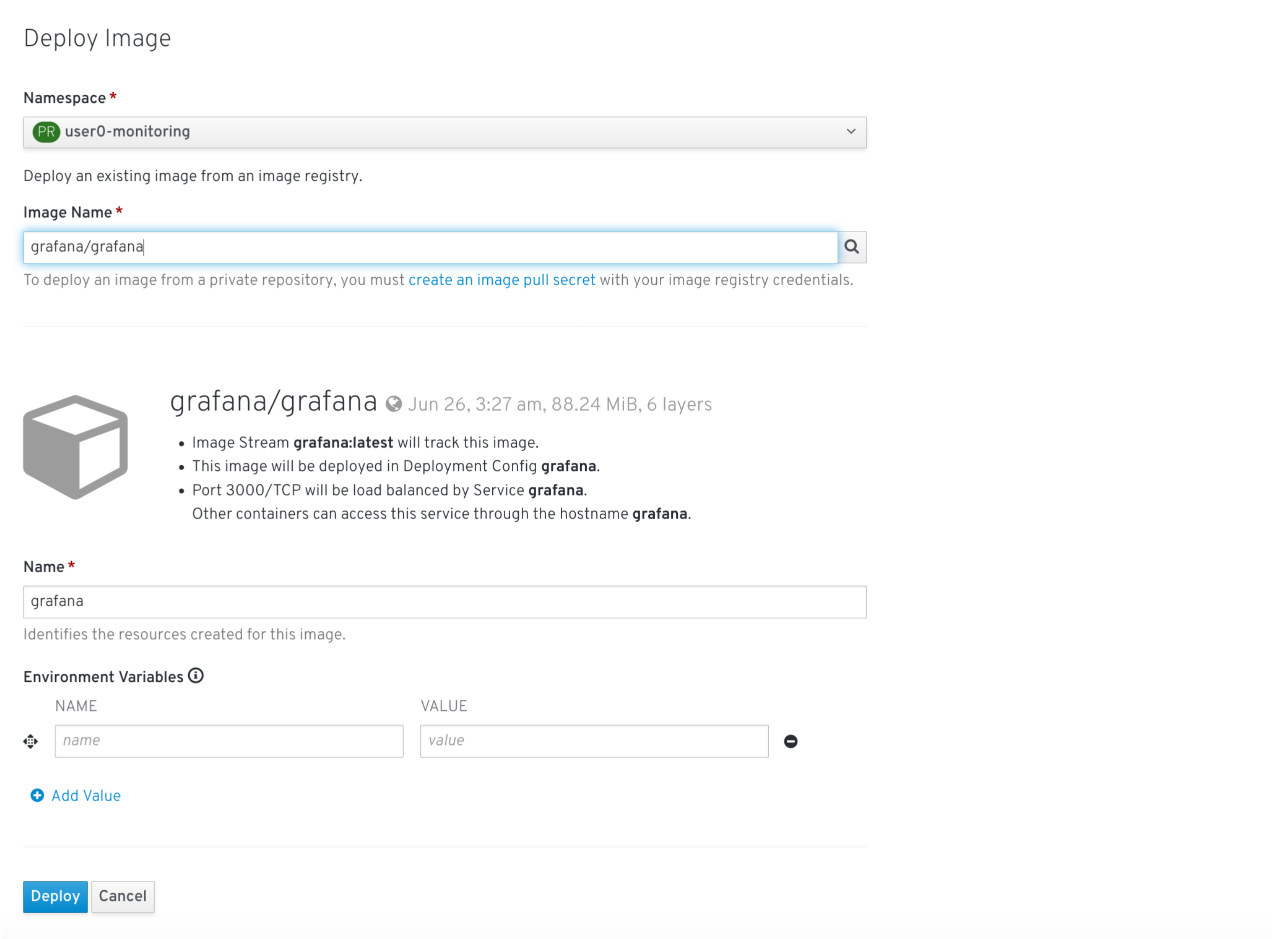
You will see the landing page of Prometheus as shown:



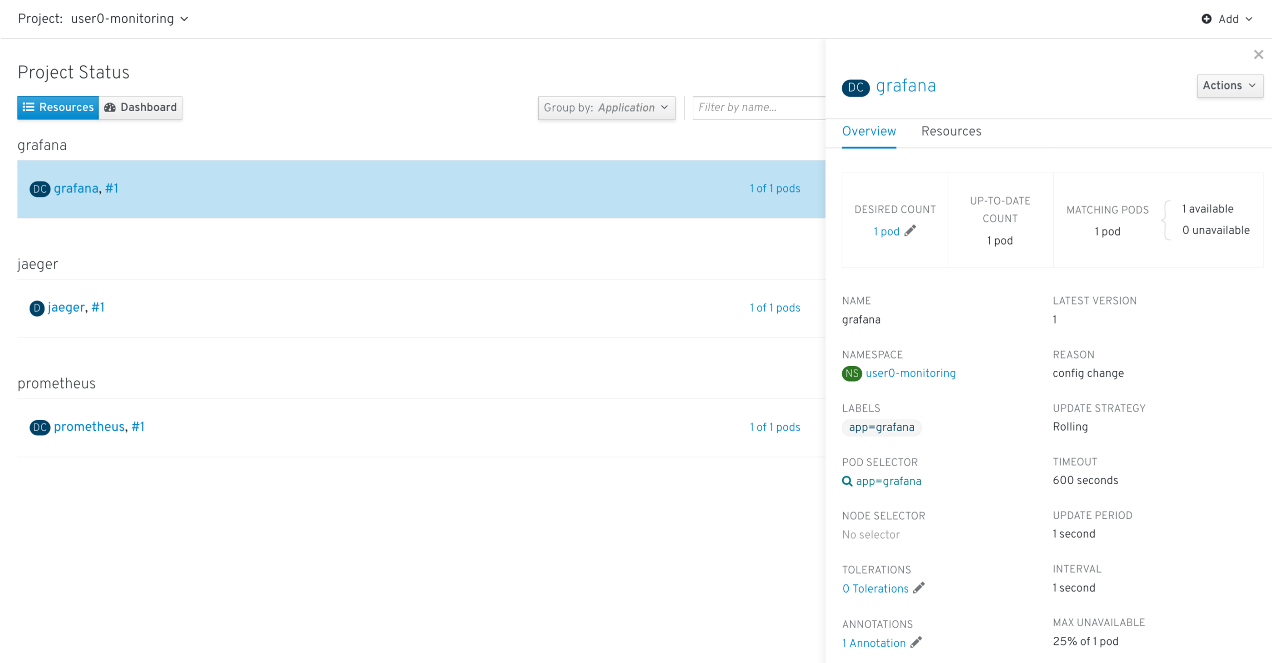
Let's deploy **Grafana Dashboards** to OpenShift. Go to *Project Status* page in *userXX-monitoring* project and click on **Deploy Image** under *Add* menu:



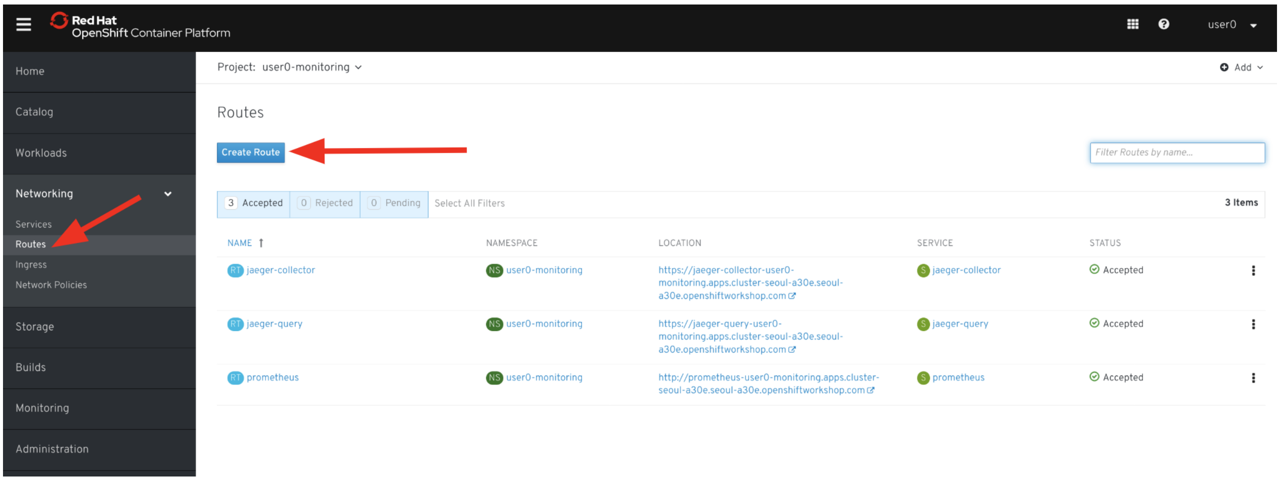
Select **Image Name** and input *grafana/grafana* to search the Prometheus container image via clicking on *Search* icon.



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

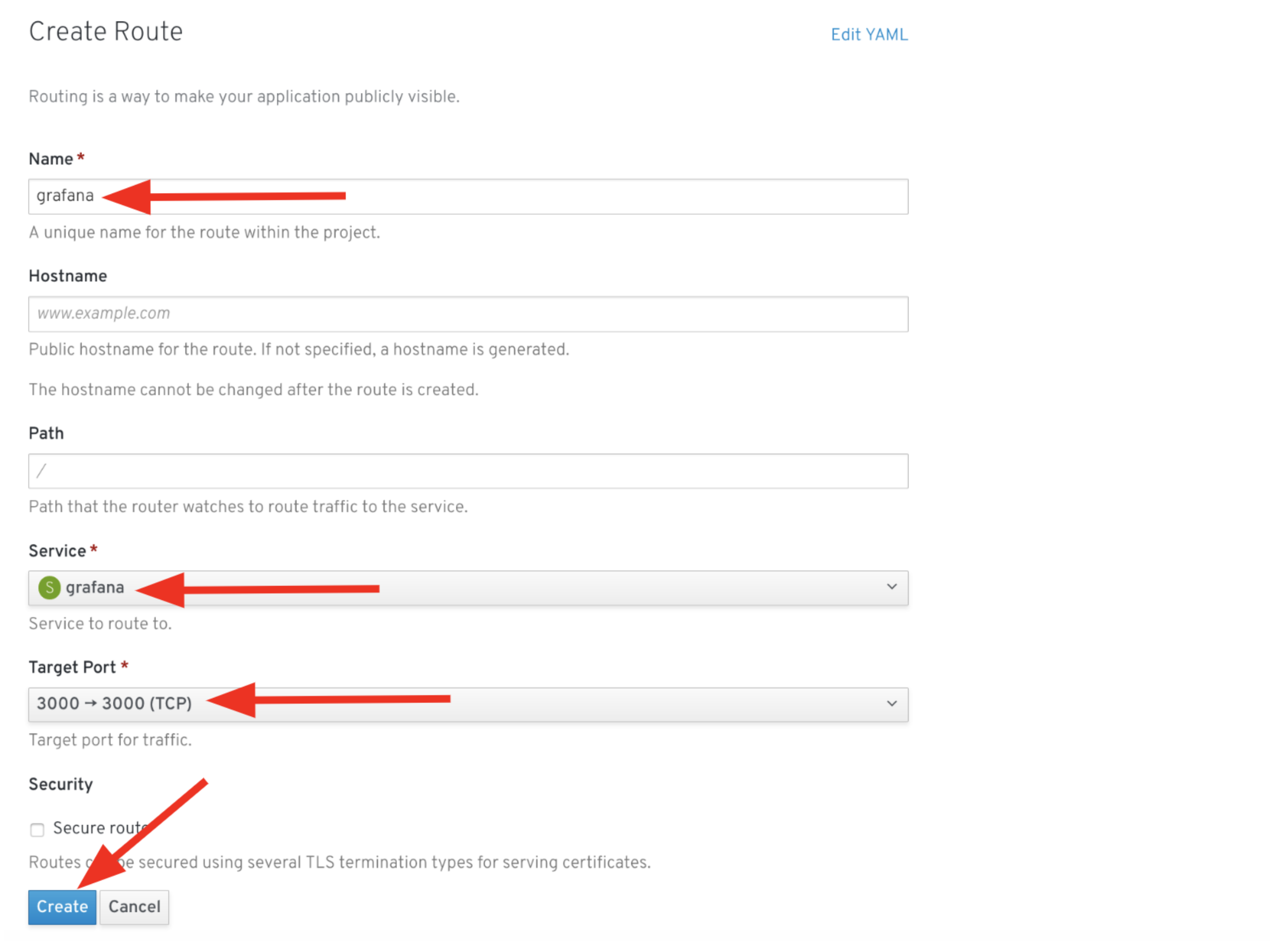


Create the route to access Grafana web UI. Navigate *Networking* > *Routes* on the left menu and click on **Create Route**.

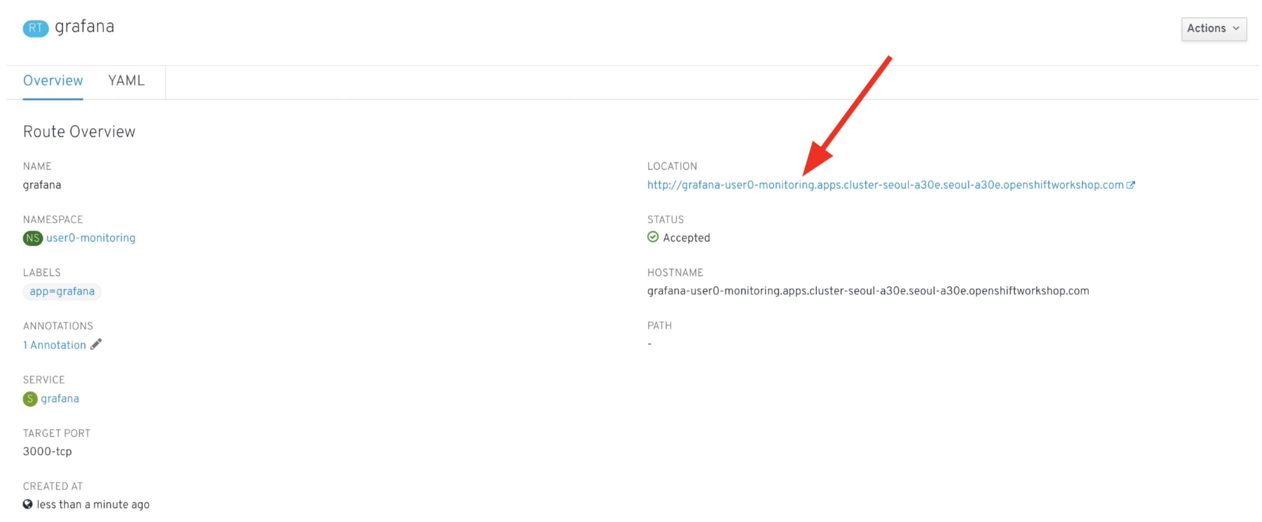


Input the following variables and keep the rest of all default variables. Click on `Create`.

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



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.

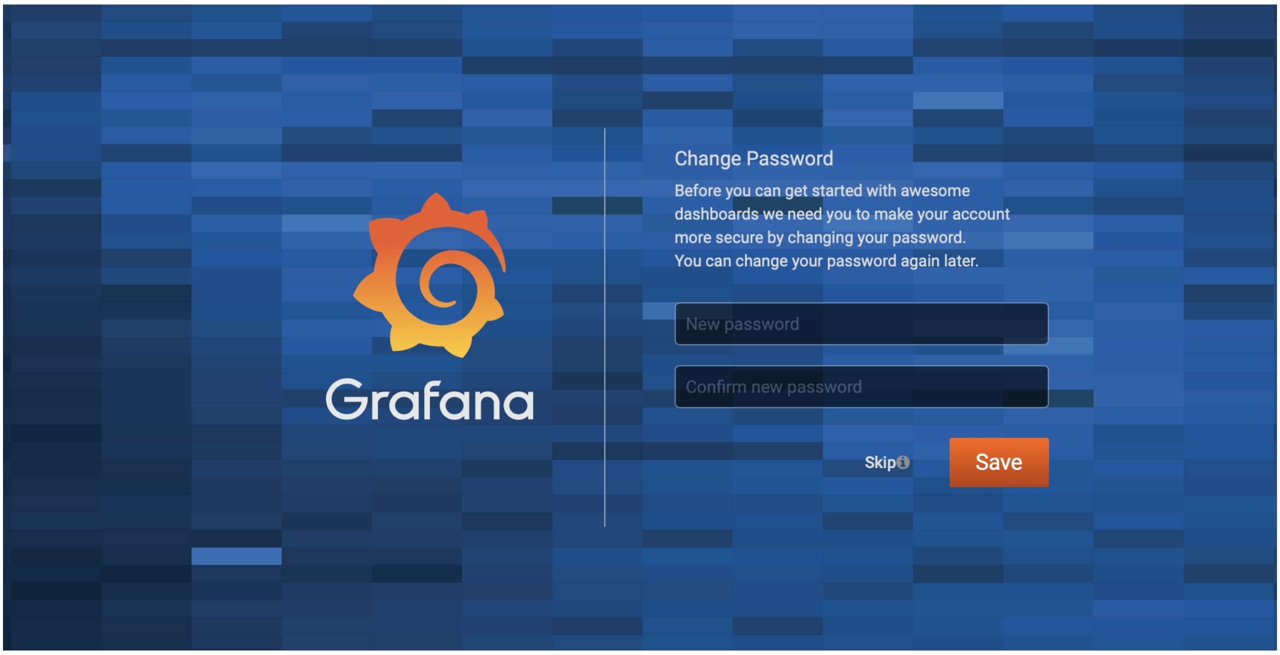


You will see the landing page of Prometheus as shown:

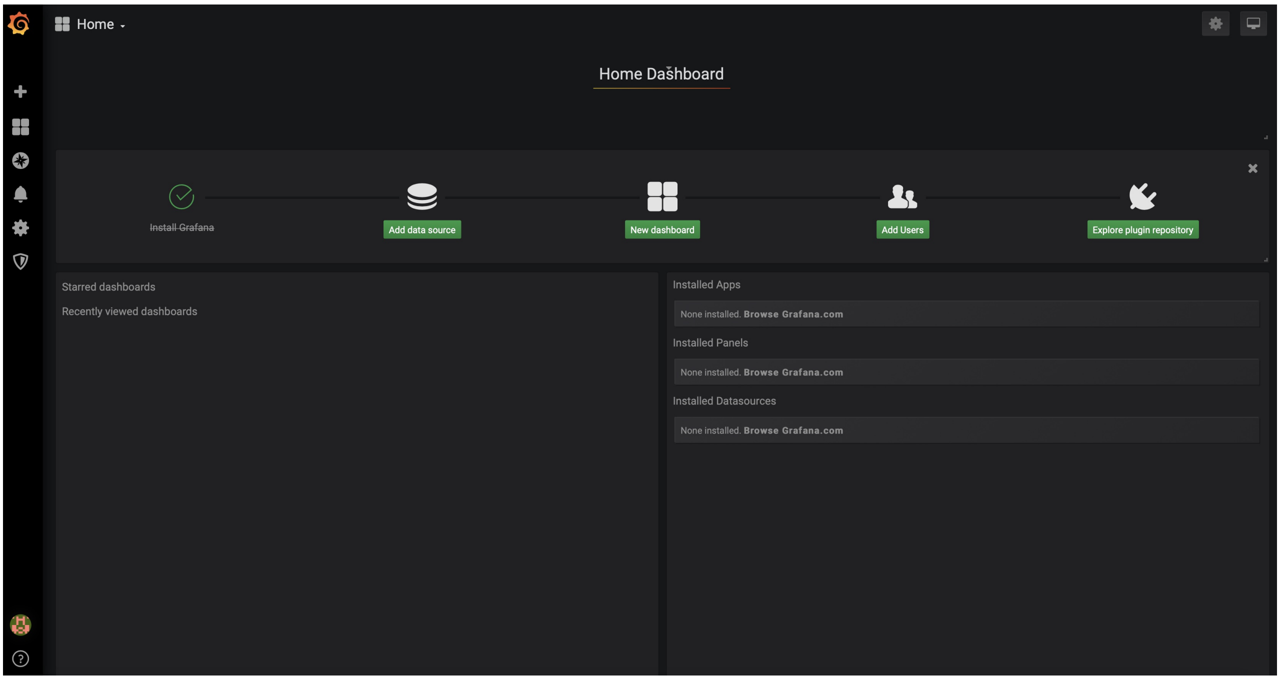
Log in Grafana web UI using the following variables:

- Username: admin
- Password: admin

Skip the Change Password.

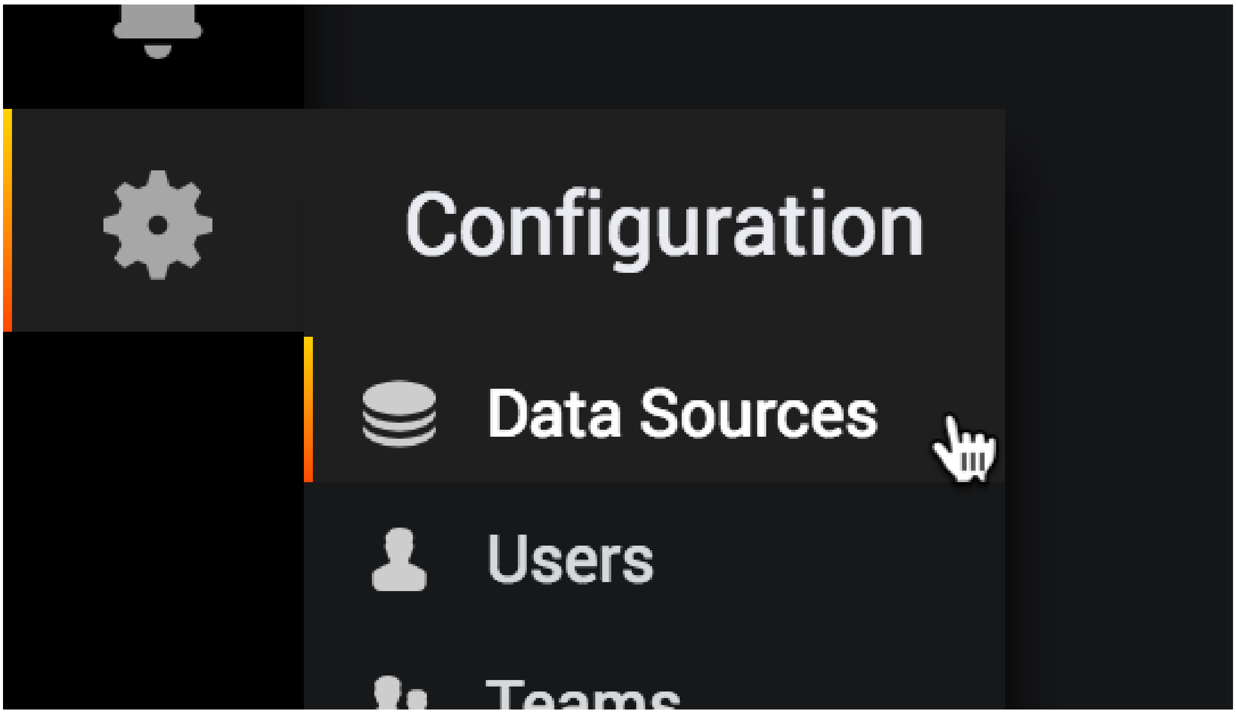


You will see the landing page of Grafana as shown:

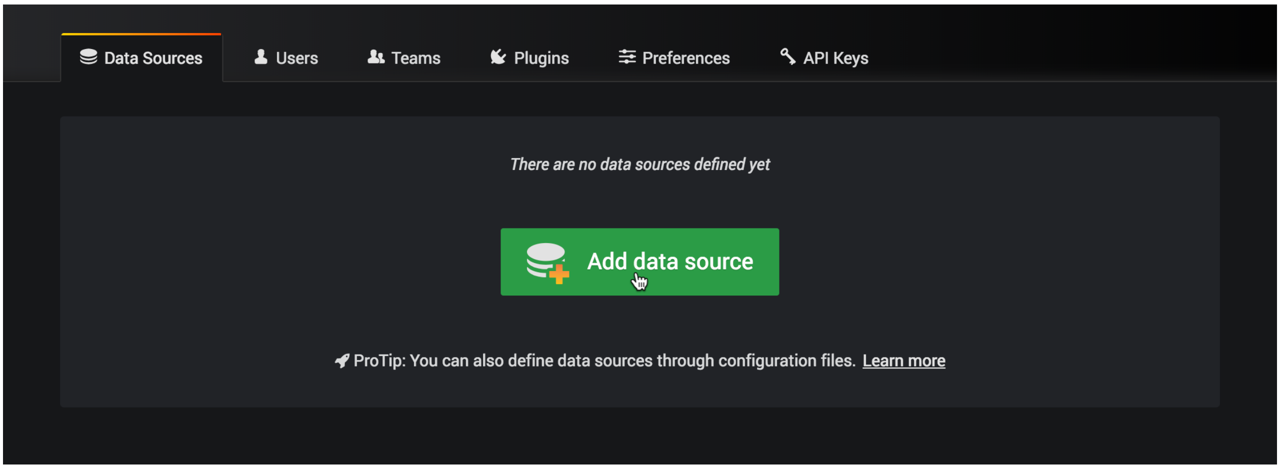


10. Add a data source to Grafana

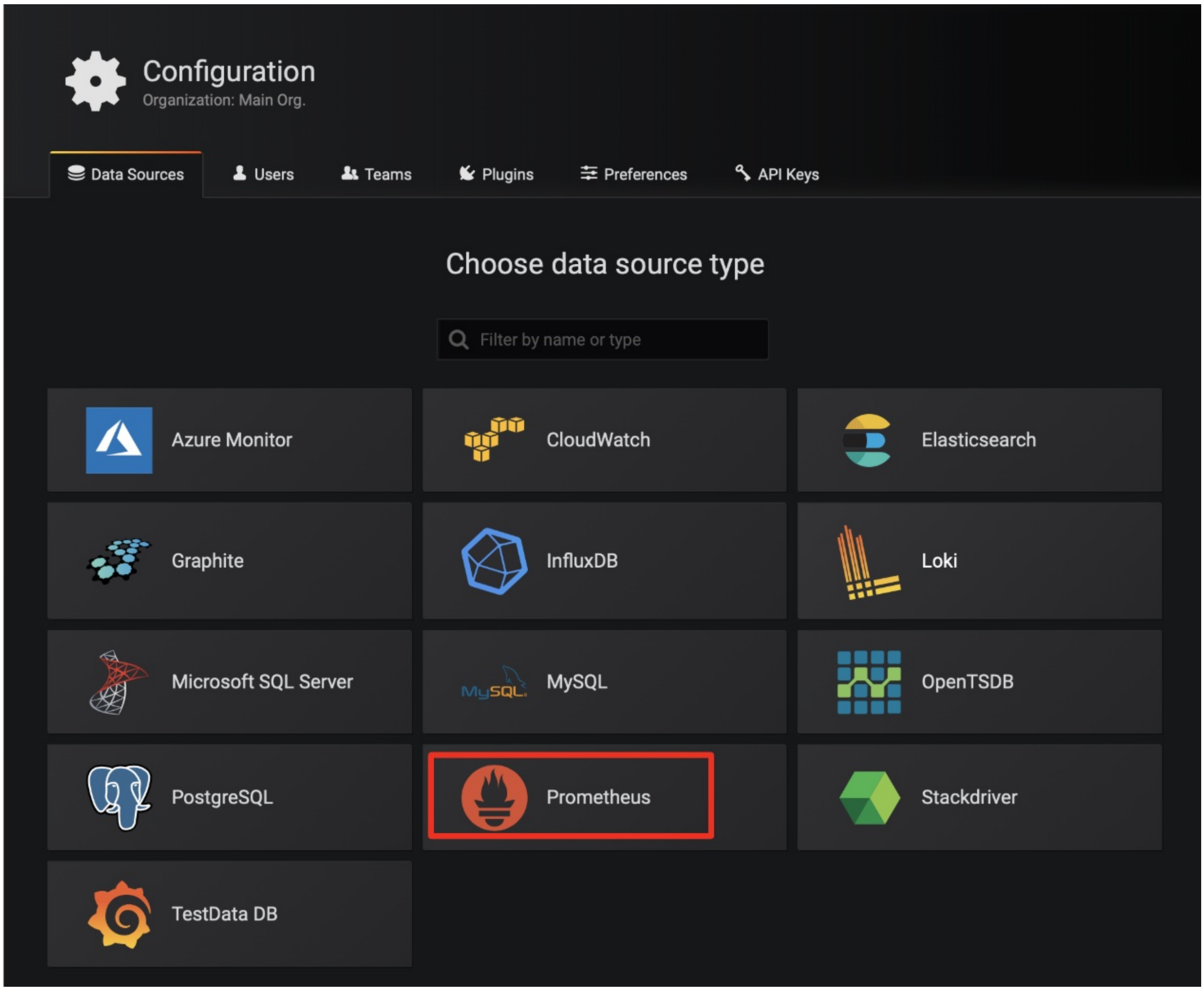
Before we create a monitoring dashboard, we need to add a data source. Go to the cog on the side menu that will show you the configuration menu. If the side menu is not visible click the Grafana icon in the upper left corner.



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.



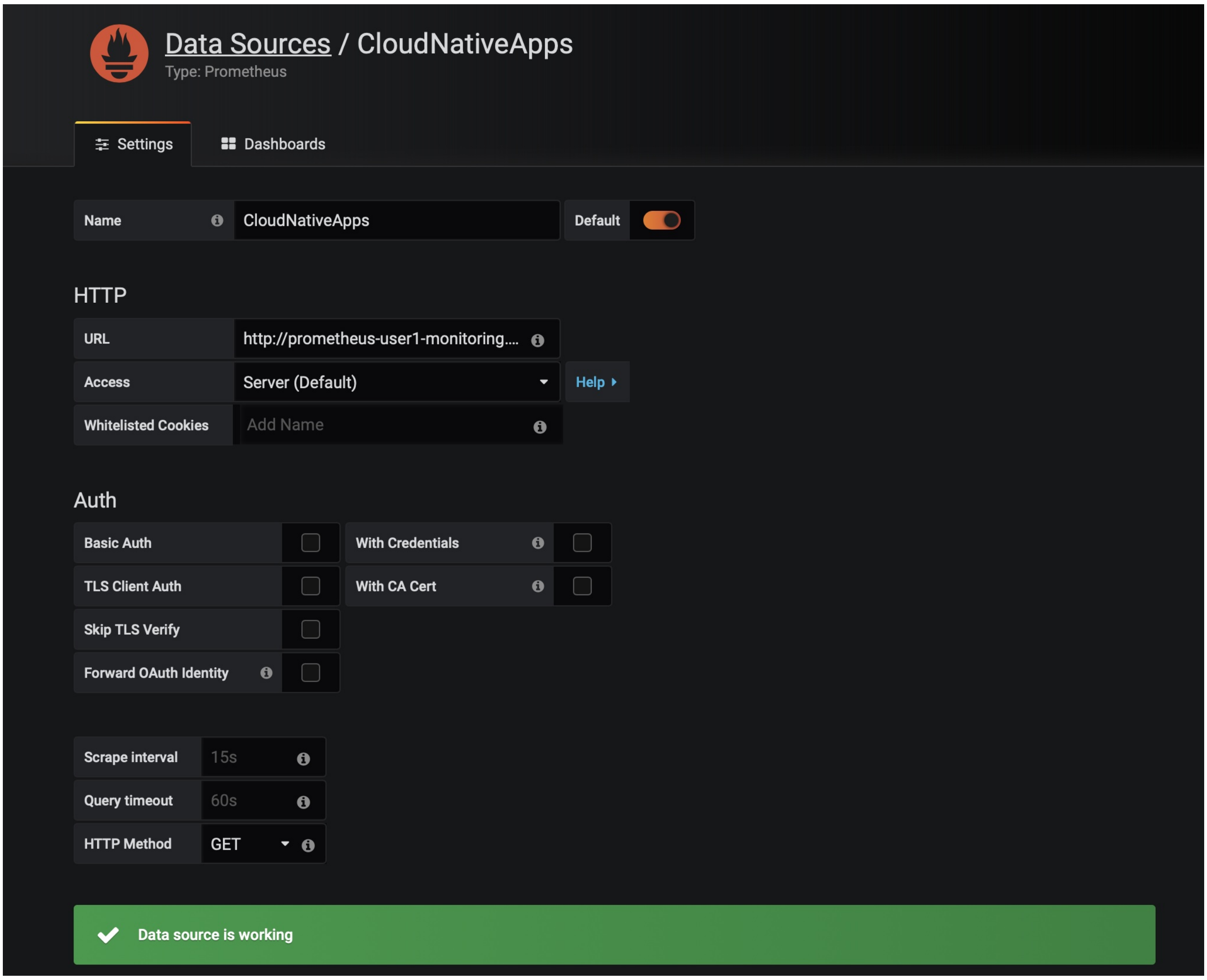
Click Add data source and select **Prometheus** as data source type.



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



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 Inverntory(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.

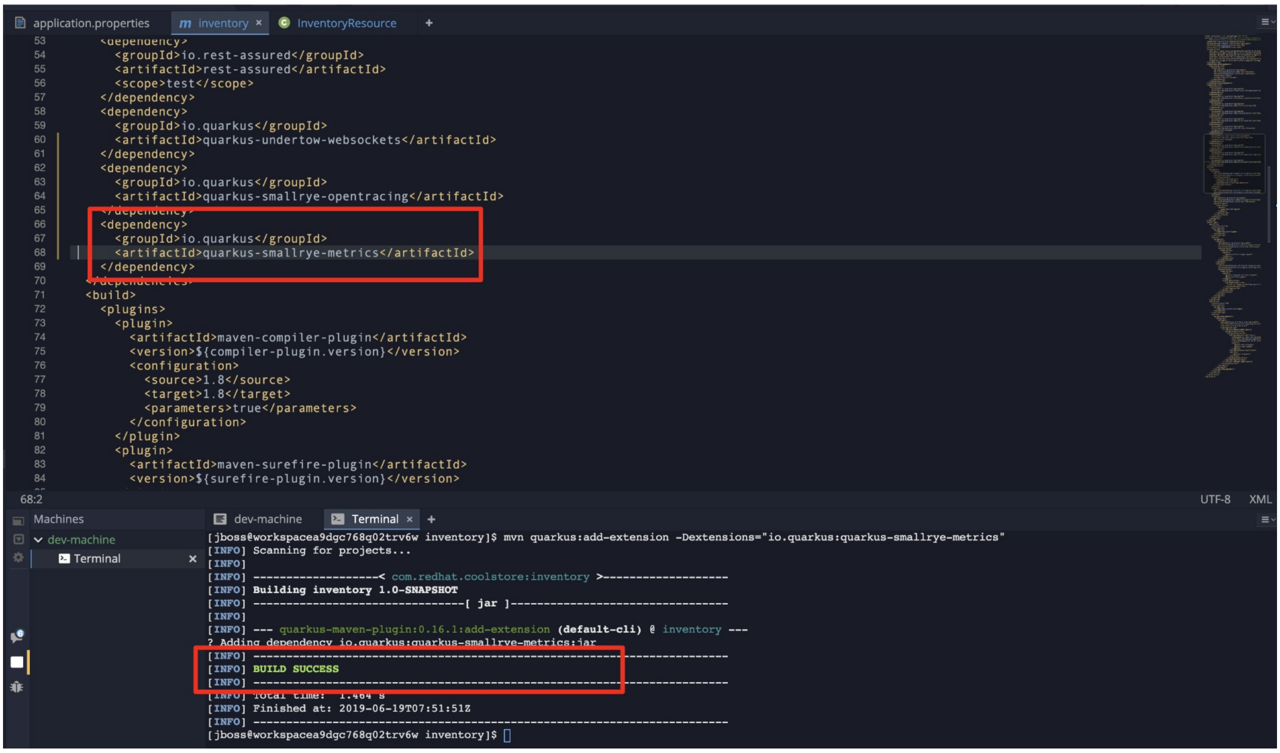
The metrics can be read remotely using JSON format or the **OpenMetrics** format, so that they can be processed by additional tools such as *Prometheus*, and stored for analysis and visualisation.

We will add Qurakus 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-metricsextension 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.



Let's add a few annotations to make sure that our desired metrics are calculated over time and can be exported for processing by *Prometheu* 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_`):

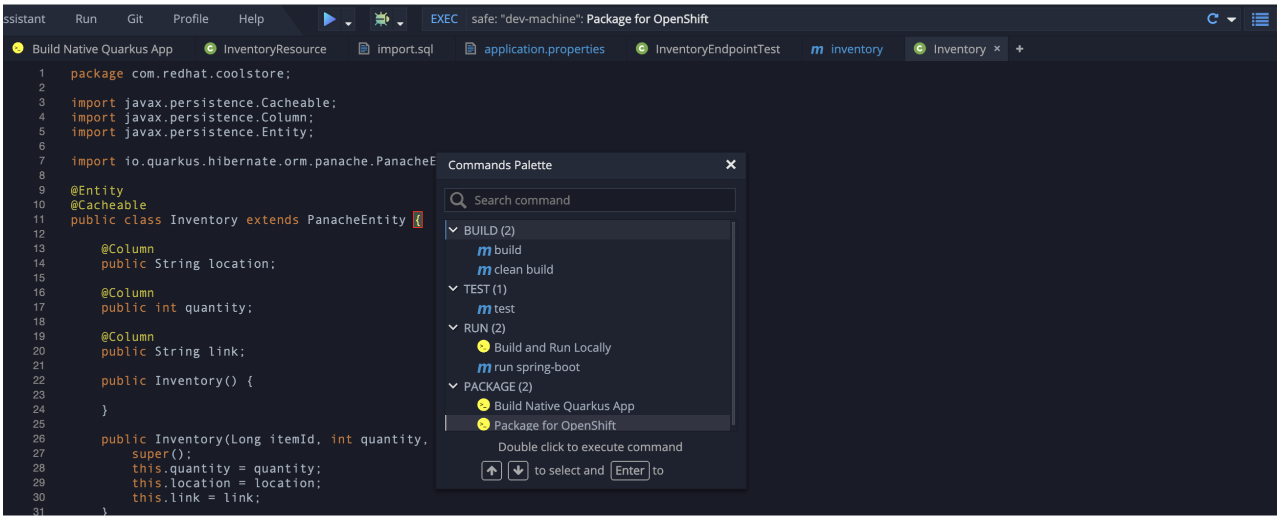
```
    @GET
    @Counted(name = "performedChecksAll", description = "How many getAll() have been performed.")
    @Timed(name = "checksTimerAll", description = "A measure of how long it takes to perform the getAll().", unit = MetricUnits.MILLISECONDS)
    public List<Inventory> getAll() {
        return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

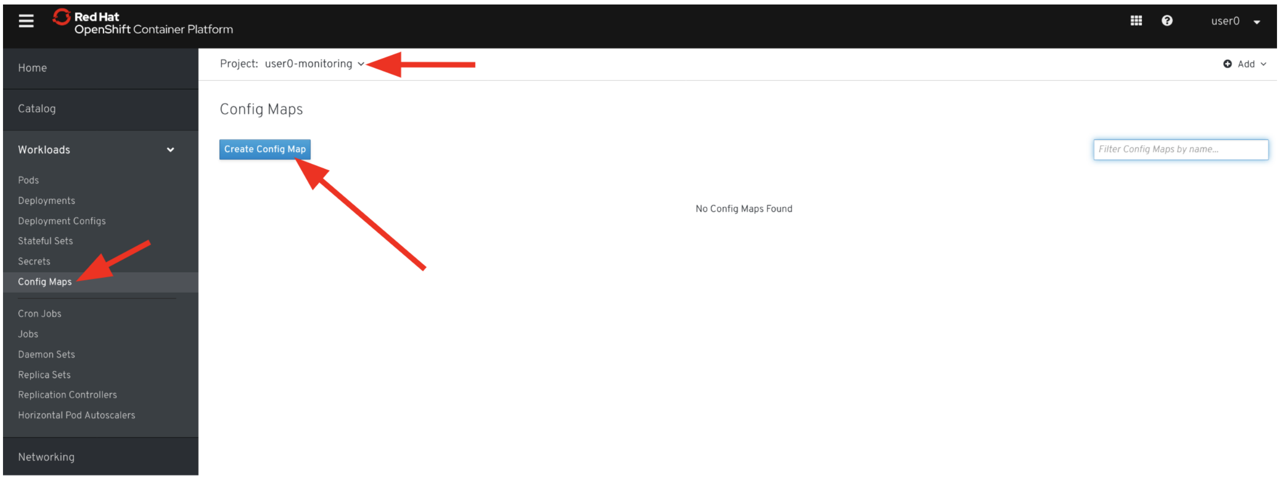
`oc start-build inventory-quarkus --from-file target/*-runner.jar --follow -n userxx-inventory`

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.

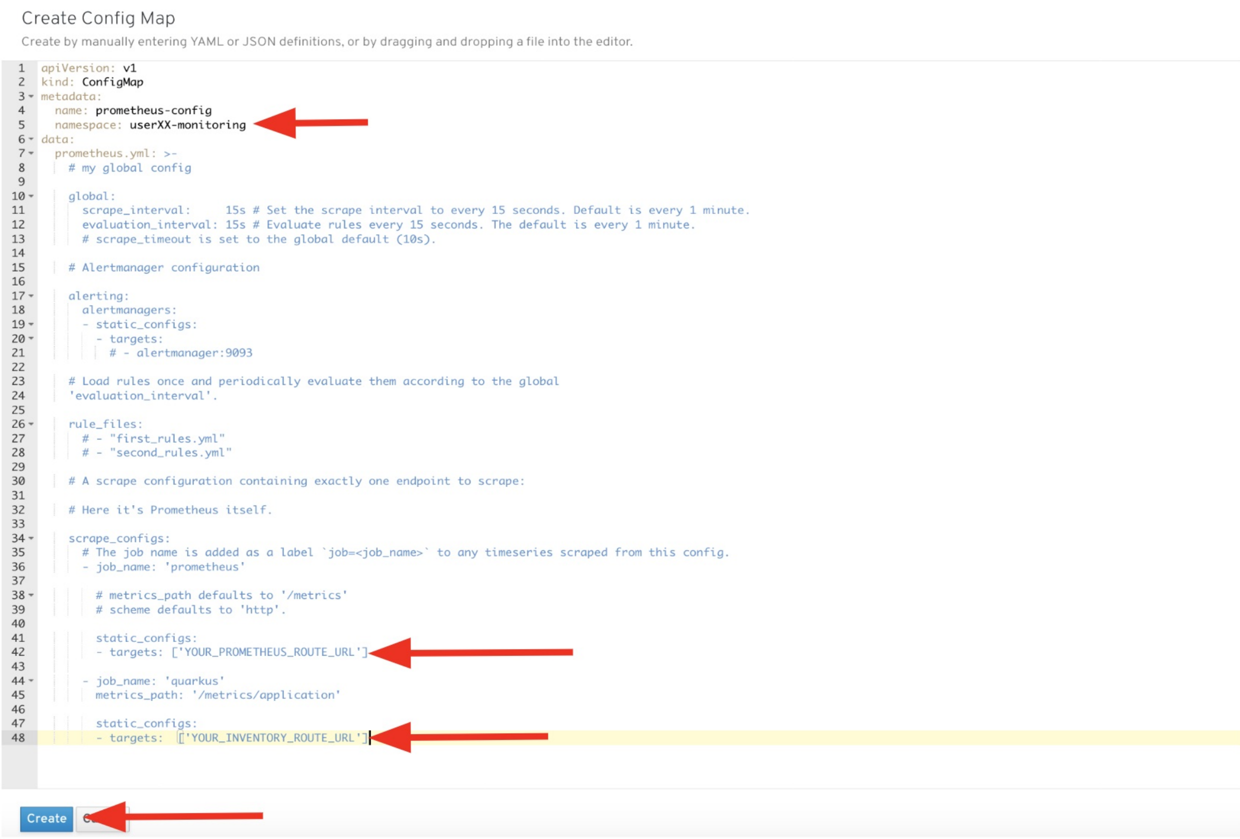


Make sure you're in the `userXX-monitoring` project in OpenShift, and click on **Create Config Maps** button to create a config map. You'll copy and paste the below code into the field.

> 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:

```yaml
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']
```



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

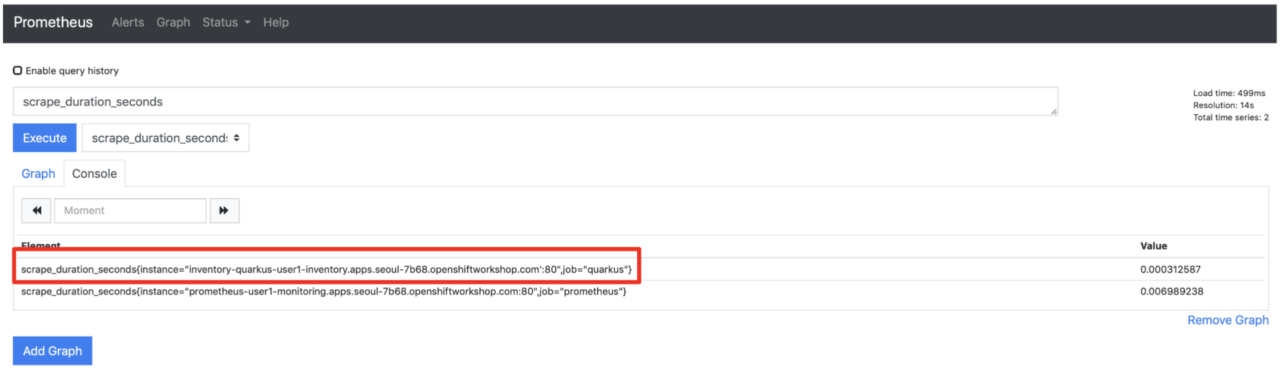
`curl $INV_URL/metrics/application`

You should something like:

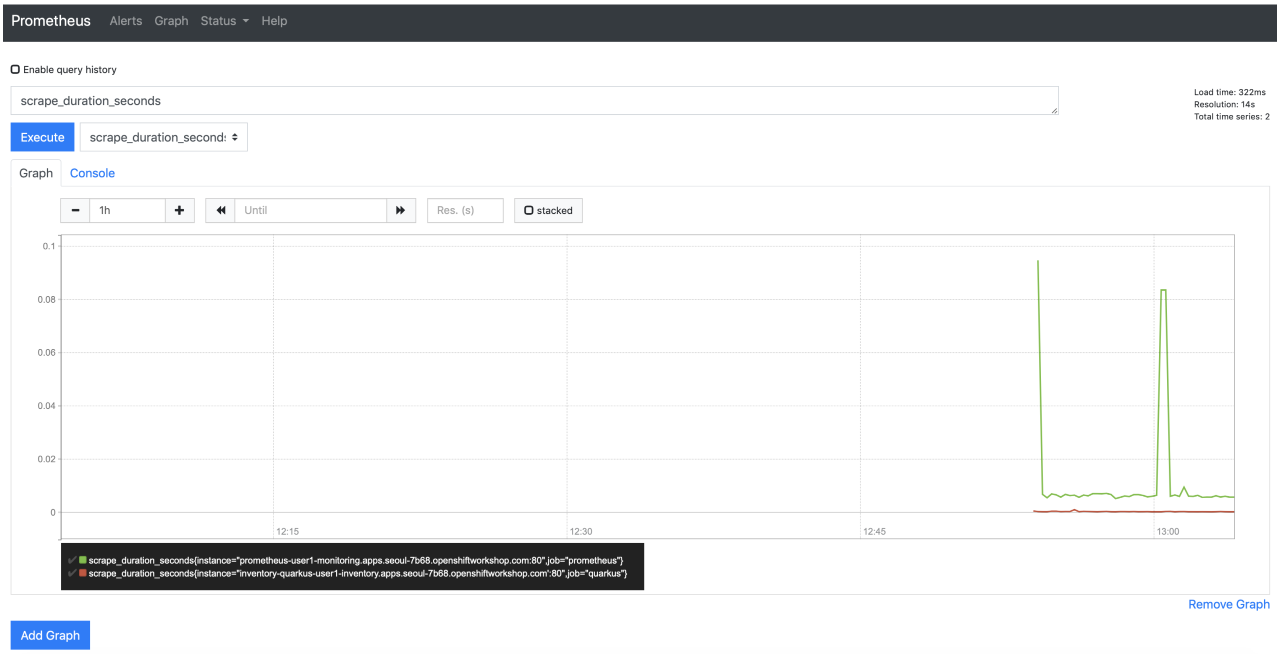
```
# TYPE application_com_redhat_coolstore_InventoryResource_checksTimerAll_stddev_seconds gauge
application_com_redhat_coolstore_InventoryResource_checksTimerAll_stddev_seconds 1.497008378628945E-4
# HELP application_com_redhat_coolstore_InventoryResource_checksTimerAll_seconds A measure of how long 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:



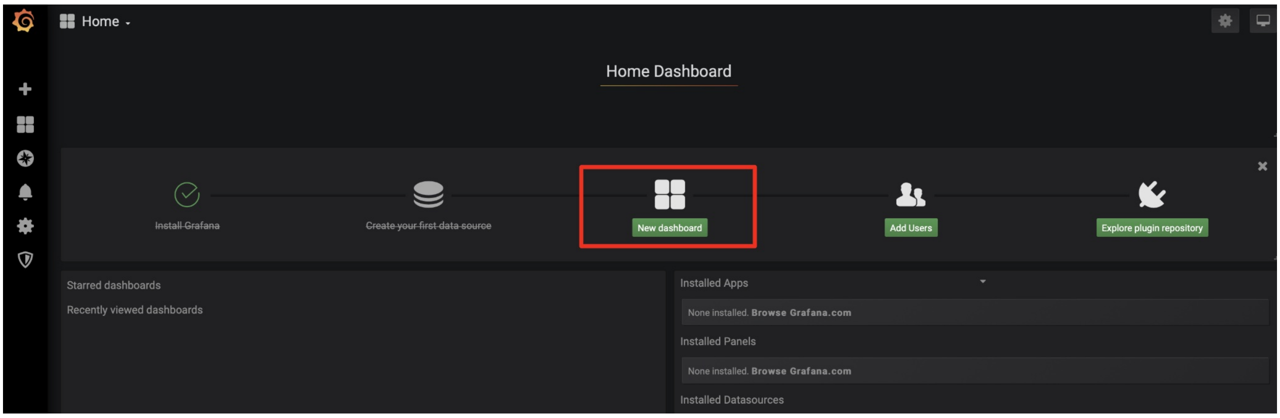
Switch to **Graph** tab:



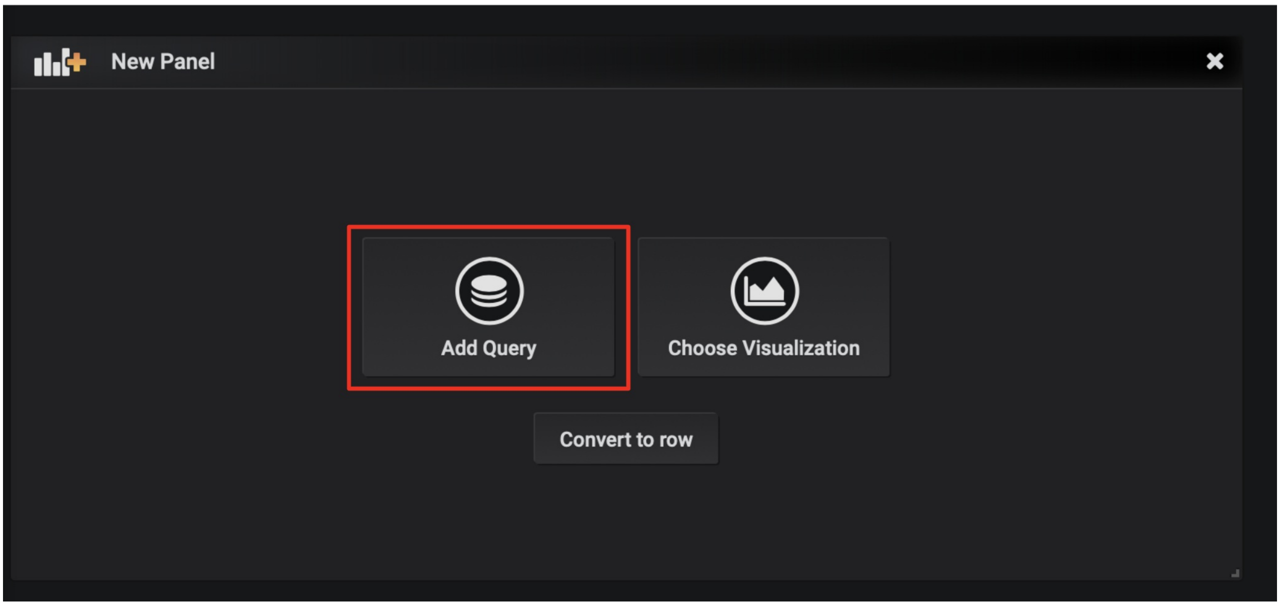
You can play with the values for time and see different data across different time ranges for this metric.

Now let’s use Grafana.

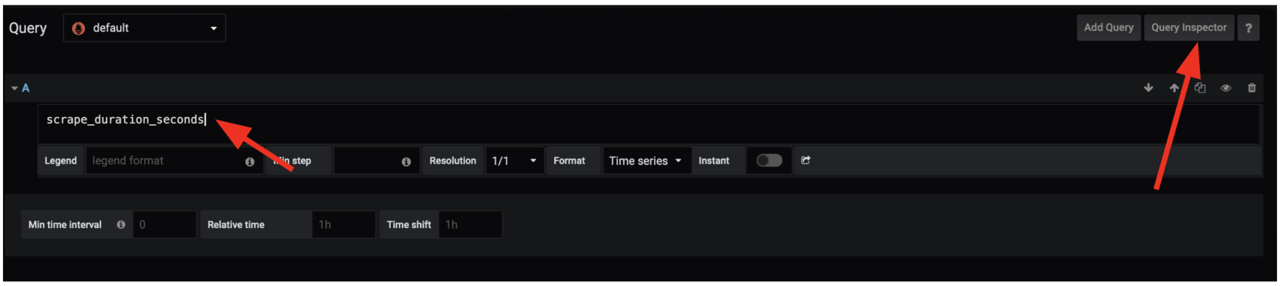
3) Open the **Grafana Web UI** (visit *Networking* > *Routes* in the `userXX-monitoring project in the OpenShift console) via a web brower and create a new *Dashboard* to review the metrics.



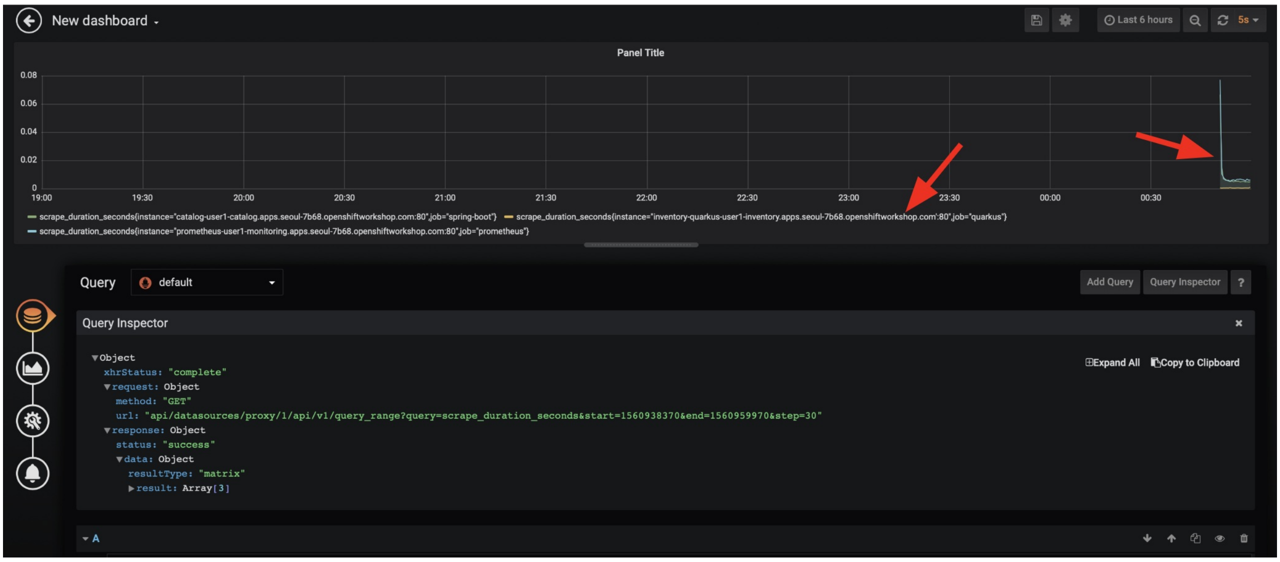
Click on **New dashboard** then select *Add Query* in a new panel:



Add **scrape_duration_seconds** in query box:



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

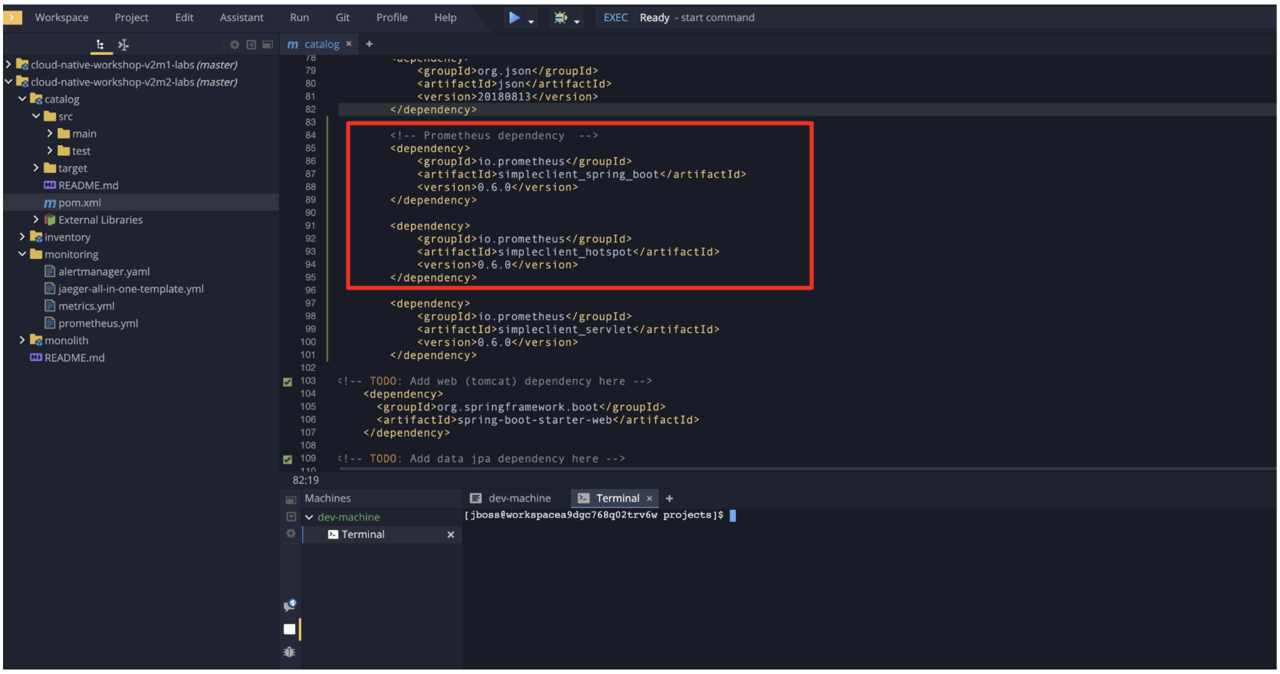


14. Utilize metrics specification for Catalog(Spring Boot)

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*:

```xml
<!-- 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>
```



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 acess 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`

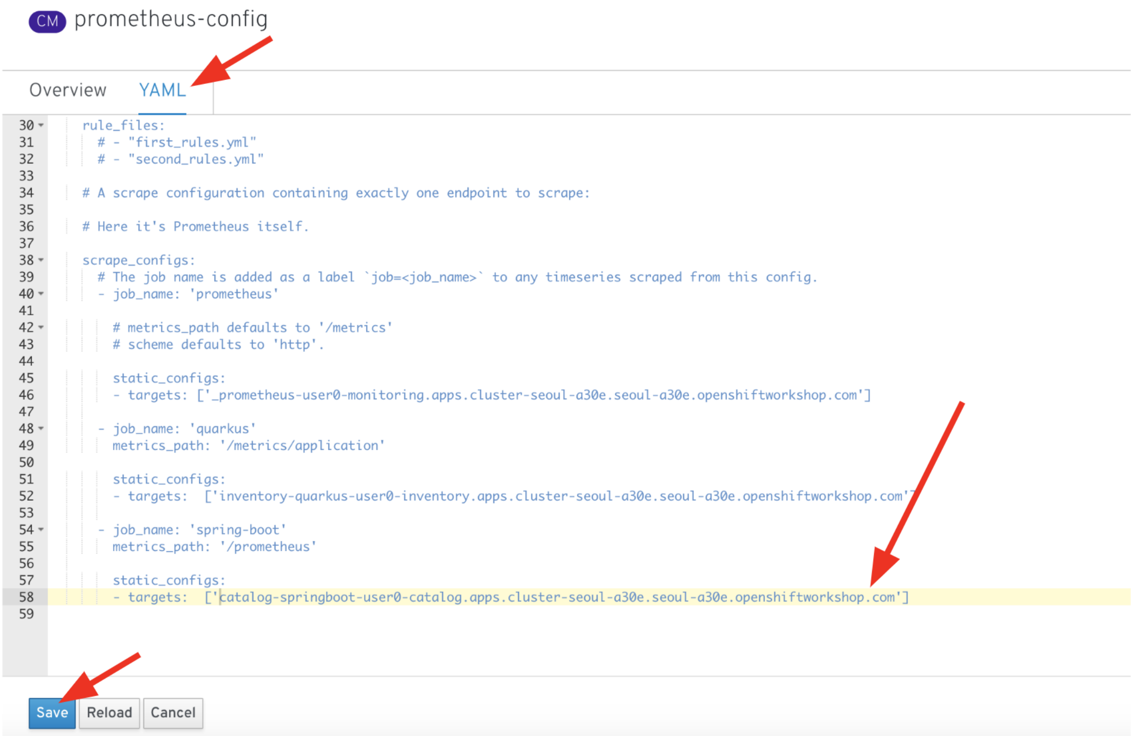
Navigate to your `userXX-monitoring` project in OpenShift console, and go to `Workloads > Config Maps > prometheus_config`. Click on the *YAML* tab.

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

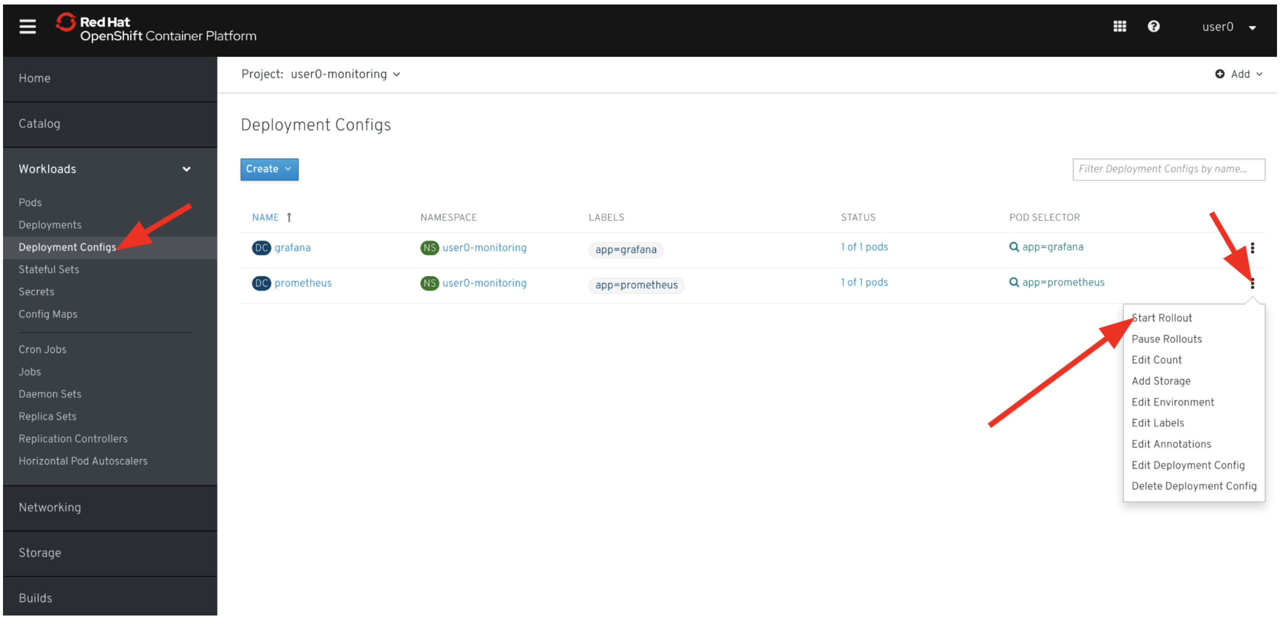
> Replace `YOUR_CATALOG_ROUTE` with the route emitted from the above `oc get route` command!

```
- job_name: 'spring-boot'
  metrics_path: '/prometheus'

  static_configs:
  - targets:  ['YOUR_CATALOG_ROUTE']
```

Click on **Save**.

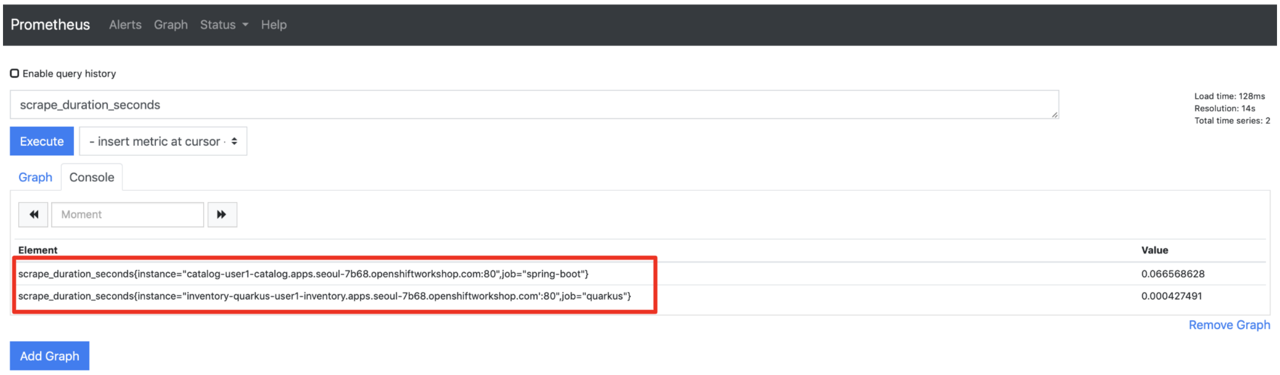


OpenShift does not automatically redeploy whenever ConfigMaps are changed, so let's force a redeployment. Select the `userXX-catalog` project in the OpenShift console, navigate to *Workloads* > *Deployment Configs* > *prometheus* and select **Start Rollout** from the *Actions* menu:

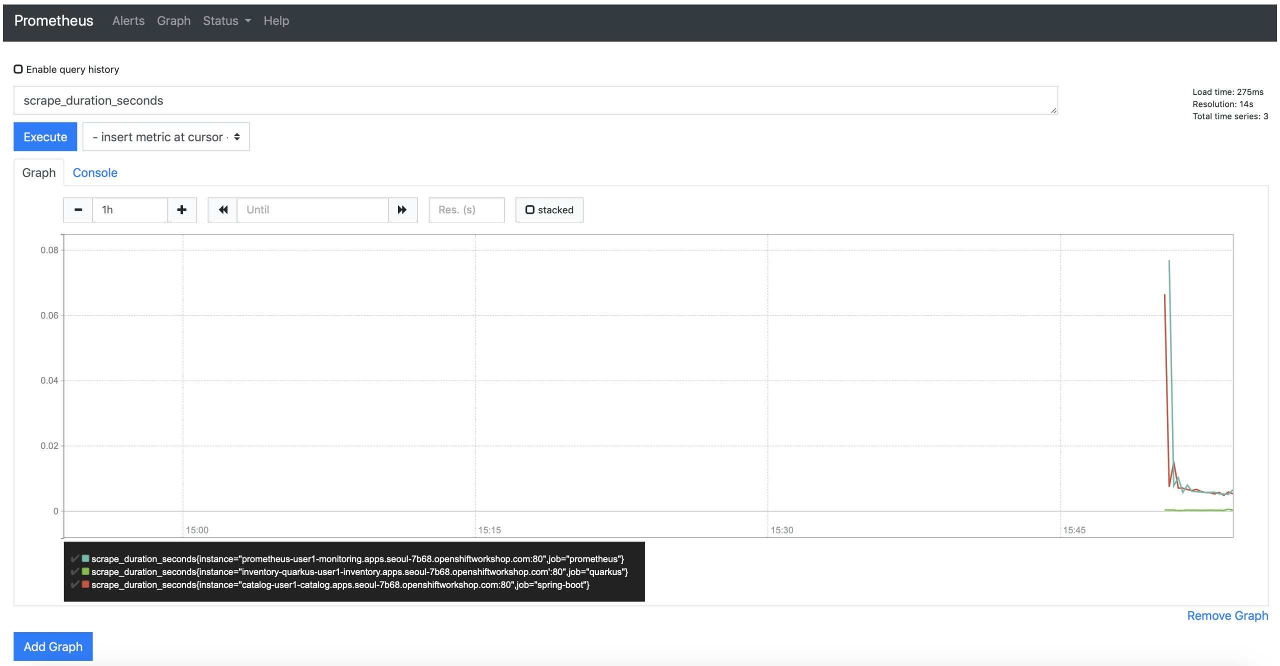


17. Observing metrics in Prometheus and Grafana

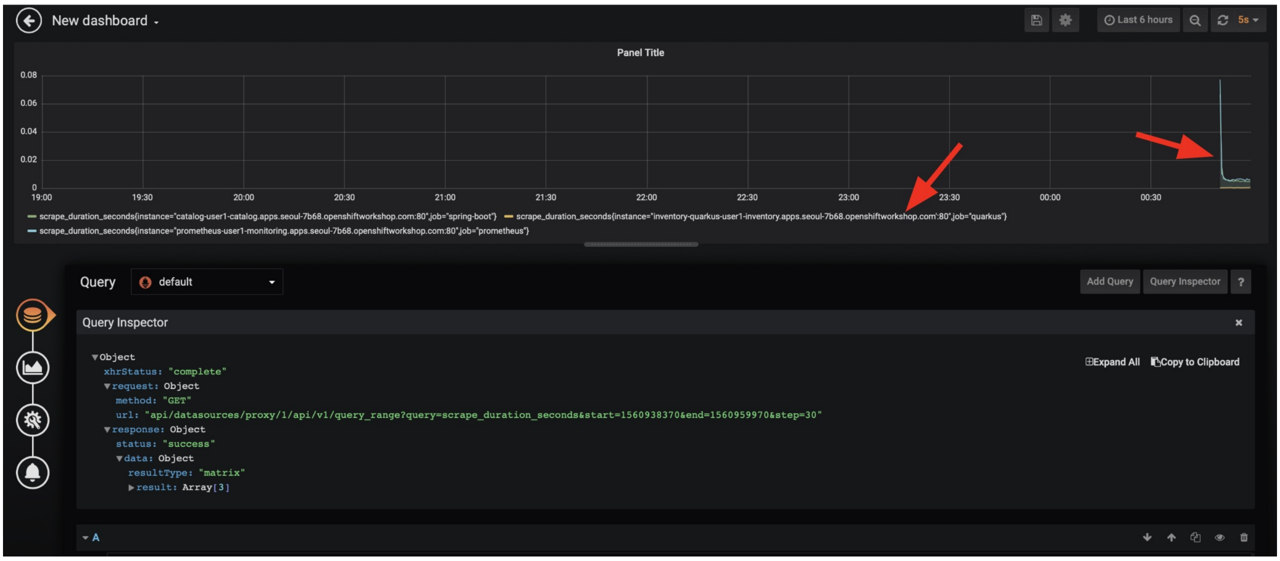
1) Open the Prometheus Web UI via a web brower and input(or select) `scrape_duration_seconds` in the query box. Click on **Execute** then you will see *quarkus job* in the metrics:



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-nativa 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.
