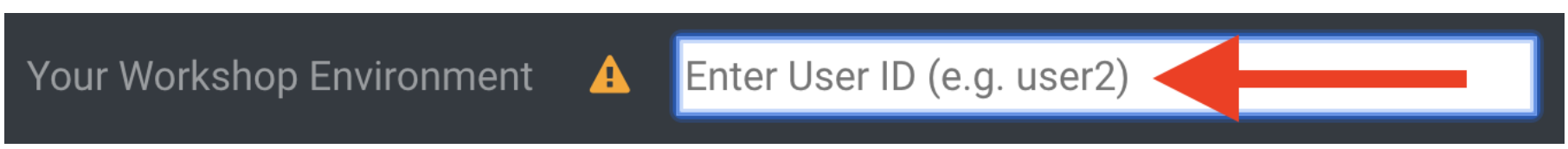
Your Workshop Environment

First Step: Confirm Your Username!

Look in the box at the top of your screen. Is your username set already? If so it will look like this:

If your username is properly set, then you can move on. **If not, in the above box, enter the user ID you were assigned** like this:



This will customize the links and copy/paste code for this workshop. If you accidently type the wrong username, just click the green recycle icon to reset it.

Throughout this lab you'll discover how Quarkus can make your development of cloud native apps faster and more productive.

Click-to-Copy

You will see various code and command blocks throughout these exercises which can be copy/pasted directly by clicking anywhere on the block of text:

```java
/* A sample Java snippet that you can copy/paste by clicking */
public class CopyMeDirectly {
    public static void main(String[] args) {
        System.out.println("You can copy this whole class with a click!");
    }
}
```

Simply click once and the whole block is copied to your clipboard, ready to be pasted with `CTRL` + `V` (or `Command` + `V` on Mac OS).

There are also Linux shell commands that can also be copied and pasted into a Terminal in your Development Environment:

```sh
echo "This is a bash shell command that you can copy/paste by clicking"
```

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 (https://www.openshift.com/) - You'll use one or more **projects** (Kubernetes namespaces) that are your own and are isolated from other workshop students
- Red Hat CodeReady Workspaces (https://developers.redhat.com/products/codeready-workspaces/overview) - 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 (https://developers.redhat.com/products/rhamt) - You'll use this to migrate an existing application
- Red Hat Runtimes (https://www.redhat.com/en/products/runtimes) - a collection of cloud-native runtimes like Spring Boot, Node.js, and Quarkus (https://quarkus.io)
- Red Hat AMQ Streams (https://www.redhat.com/en/technologies/jboss-middleware/amq) - streaming data platform based on **Apache Kafka**
- Red Hat SSO (https://access.redhat.com/products/red-hat-single-sign-on) - For authentication / authorization - based on **Keycloak**
- Other open source projects like Knative (https://knative.dev) (for serverless apps), Jenkins (https://jenkins.io/) and Tekton (https://cloud.google.com/tekton/) (CI/CD pipelines), Prometheus (https://prometheus.io) and Grafana (https://grafana.com) (monitoring apps), and more.

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

Red Hat offers the fully supported Red Hat Build of Quarkus(RHBQ) (https://access.redhat.com/products/quarkus) with support and maintenance of Quarkus. In this workhop, you will use Quarkus to develop Kubernetes-native microservices and deploy them to OpenShift. Quarkus is one of the runtimes included in Red Hat Runtimes (https://www.redhat.com/en/products/runtimes). Learn more about RHBQ (https://access.redhat.com/documentation/en-us/red_hat_build_of_quarkus).

How to complete this workshop

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.

Good luck, and let's get started!

Advanced Cloud-Native Services

If you completed 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 modernizing existing applications with microservices. If you did not, or you started but didn't finish, there are scripts below you can run to catch up.

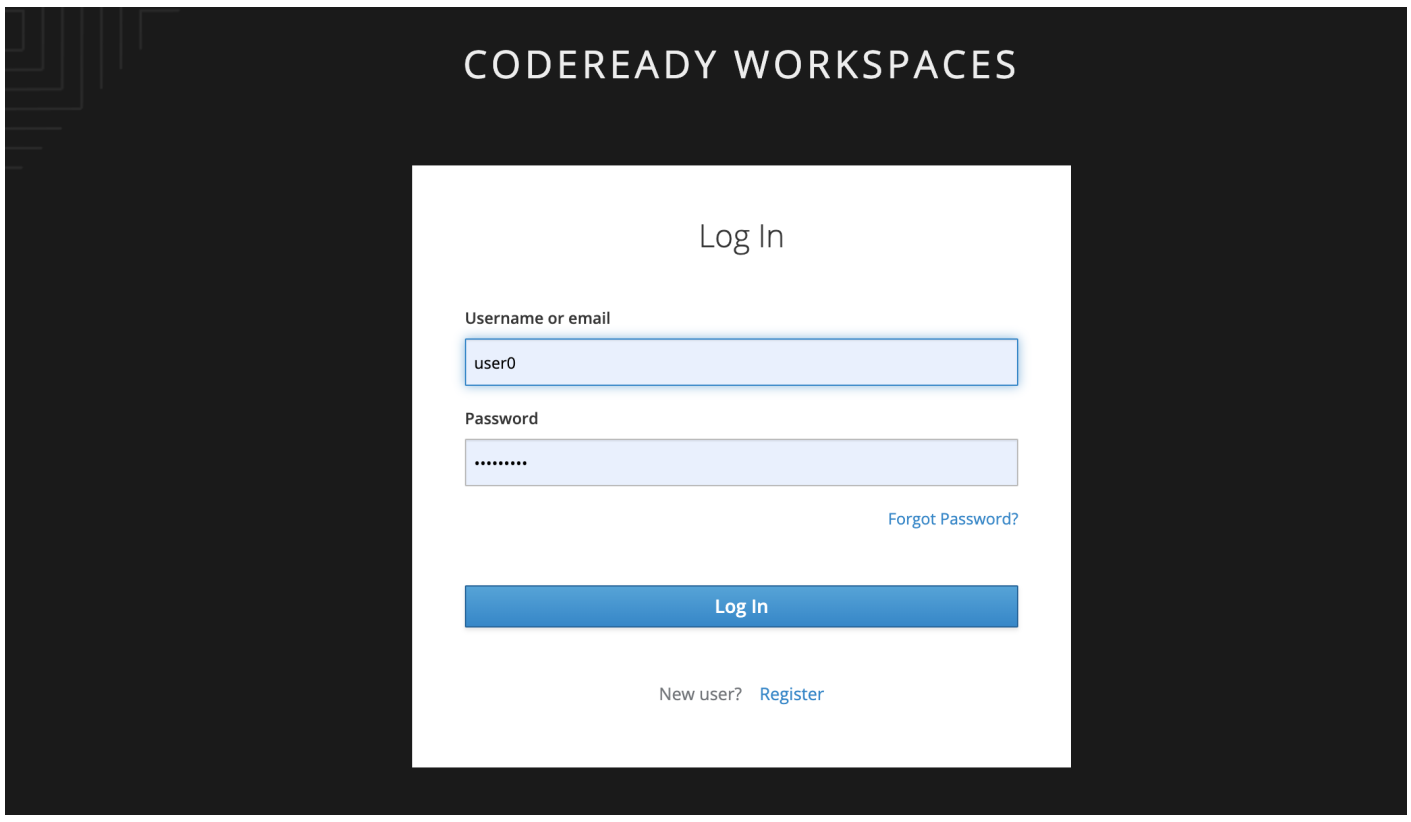
In this lab, you will go deeper into how to use the OpenShift Container Platform as a developer to build and deploy applications. We'll focus on the core features of OpenShift as it relates to developers, and you'll learn typical workflows for a developer (develop, build, test, deploy, and repeat).

Getting Ready for the labs

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.

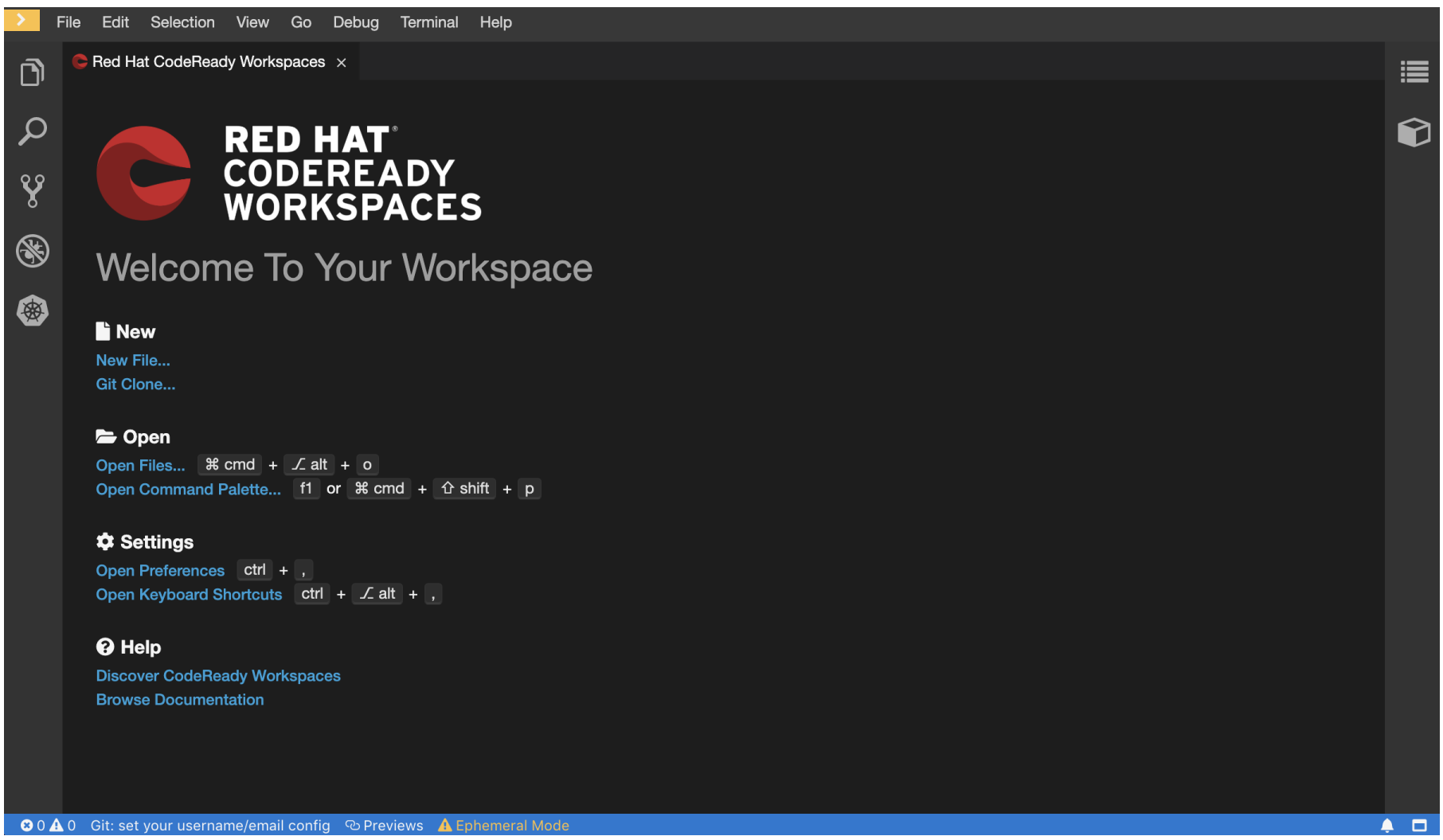
You will be using Red Hat CodeReady Workspaces, an online IDE based on Eclipse Che (https://www.eclipse.org/che/). **Changes to files are auto-saved every few seconds**, so you don't need to explicitly save changes.

To get started, access the CodeReady Workspaces instance (https://codeready-labs-infra.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com) and log in using the username and password you've been assigned (e.g. `user16/r3dh4t1!`):



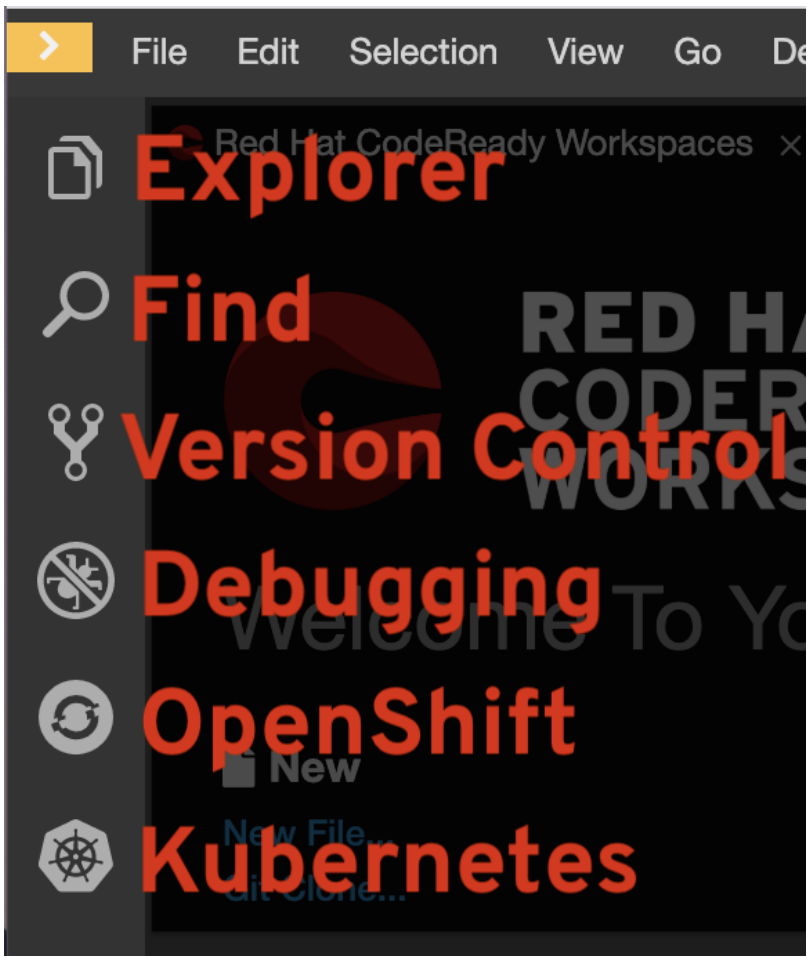
Once you log in, you'll be placed on your personal dashboard. Click on the name of the pre-created workspace on the left, as shown below (the name will be different depending on your assigned number). You can also click on the name of the workspace in the center, and then click on the green user16-namespace that says *Open* on the top right hand side of the screen.

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



This IDE is based on Eclipse Che (which is in turn based on MicroSoft VS Code editor).

You can see icons on the left for navigating between project explorer, search, version control (e.g. Git), debugging, and other plugins. You'll use these during the course of this workshop. Feel free to click on them and see what they do:



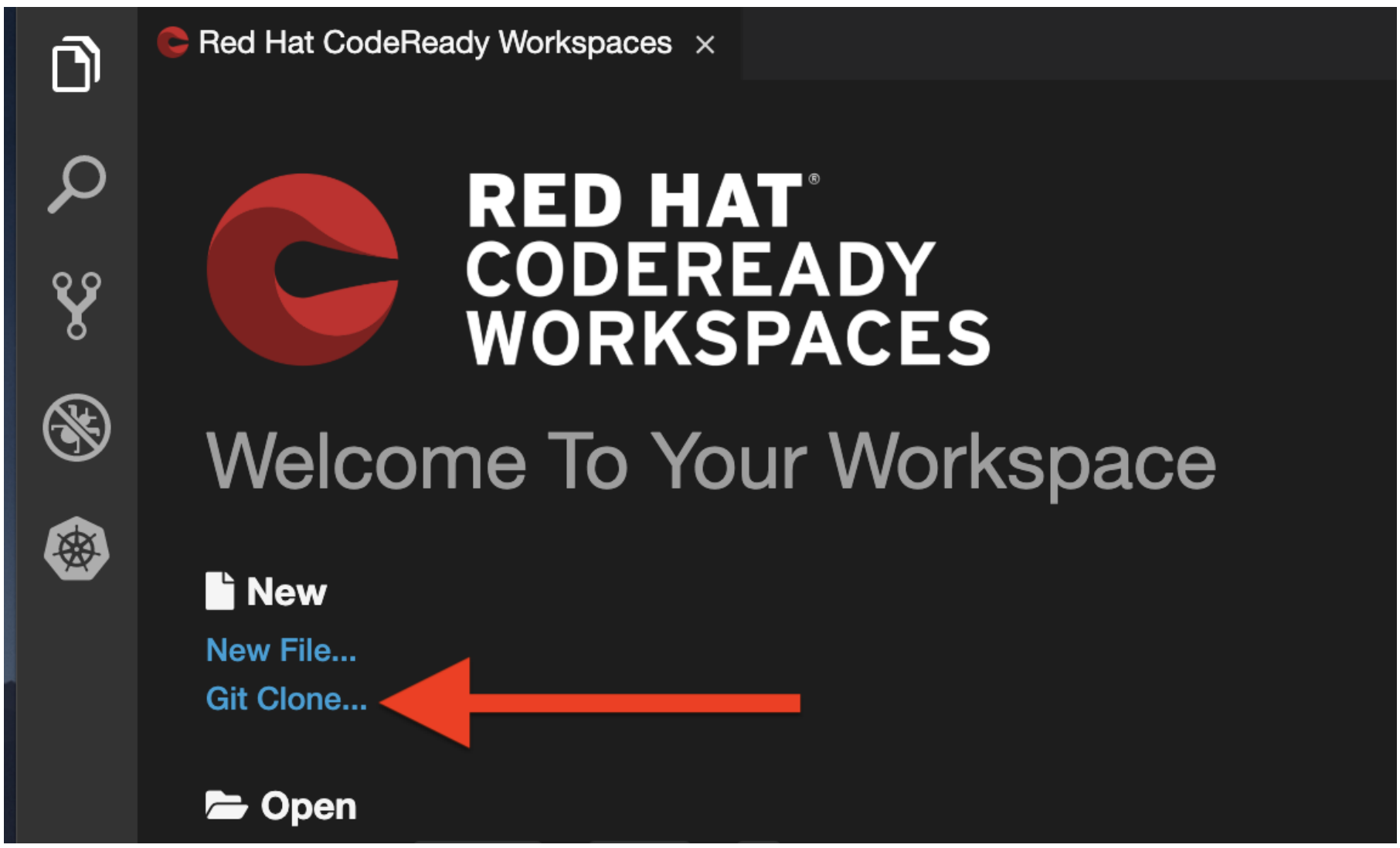
If things get weird or your browser appears, you can simply reload the browser tab to refresh the view.

Many features of CodeReady Workspaces are accessed via **Commands**. You can see a few of the commands listed with links on the home page (e.g. *New File..*, *Git Clone..*, and others).

If you ever need to run commands that you don't see in a menu, you can press `F1` to open the command window, or the more traditional `Control` + `SHIFT` + `P` (or `Command` + `SHIFT` + `P` on Mac OS X).

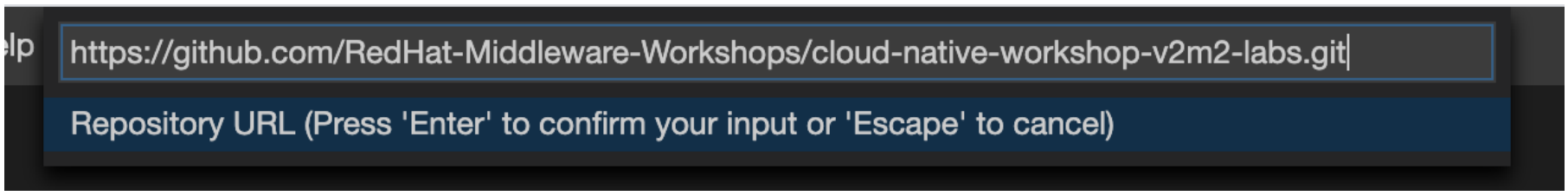
Import Projects

Let's import the project source code for this lab. Click on **Git Clone..** (or type `F1`, enter 'git' and click on the auto-completed *Git Clone..*)

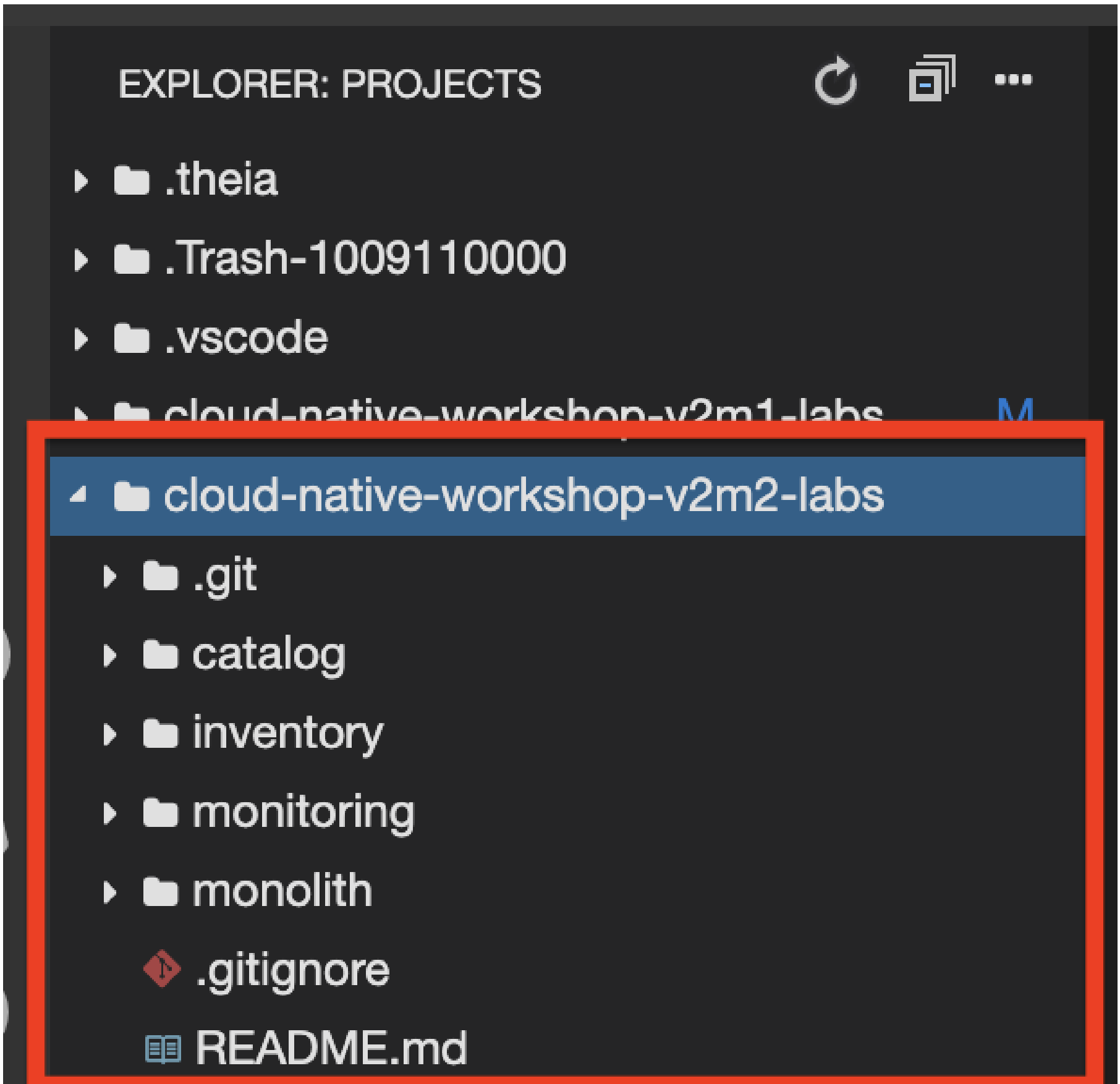


Step through the prompts, using the following value for **Repository URL**. If you use **FireFox**, it may end up pasting extra spaces at the end, so just press backspace after pasting:

```
https://github.com/RedHat-Middleware-Workshops/cloud-native-workshop-v2m2-labs.git
```



The project is imported into your workspace and is visible in the project explorer:

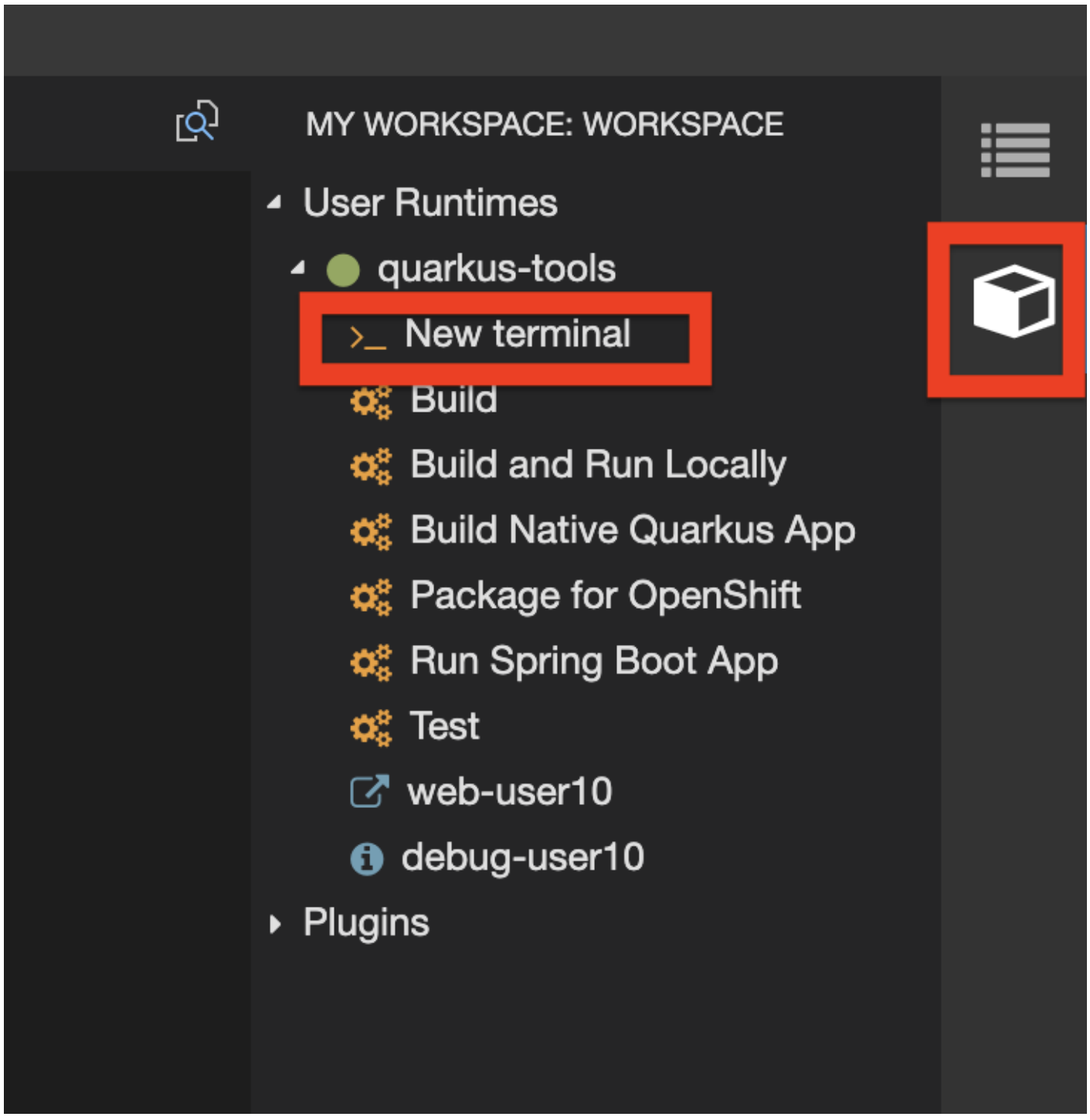


IMPORTANT: Check out proper Git branch

To make sure you're using the right version of the project files, run this command in a CodeReady Terminal:

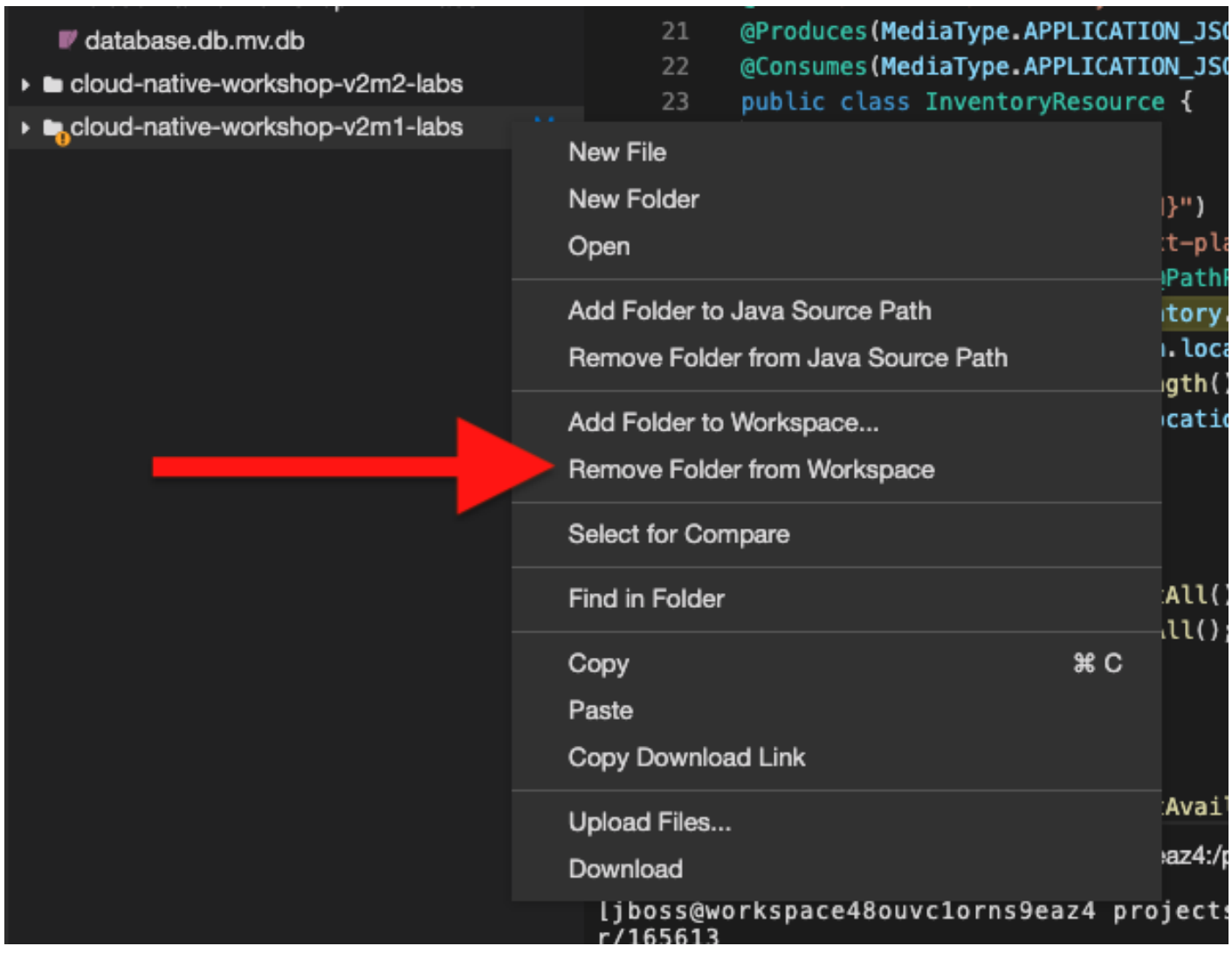
```sh
cd $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m2-labs && git checkout ocp-4.4
```

The Terminal window in CodeReady Workspaces. You can open a terminal window for any of the containers running in your Developer workspace. For the rest of these labs, anytime you need to run a command in a terminal, you can use the **>_ New Terminal** command on the right:



Remove other projects

If you've completed other modules today (such as `cloud-native-workshop-v2m1-labs`), remove them from your workspace by right-clicking on the project name in the explorer and choose **Remove from Workspace** and accept the warning. Be sure not to delete the new project you just imported for this lab!

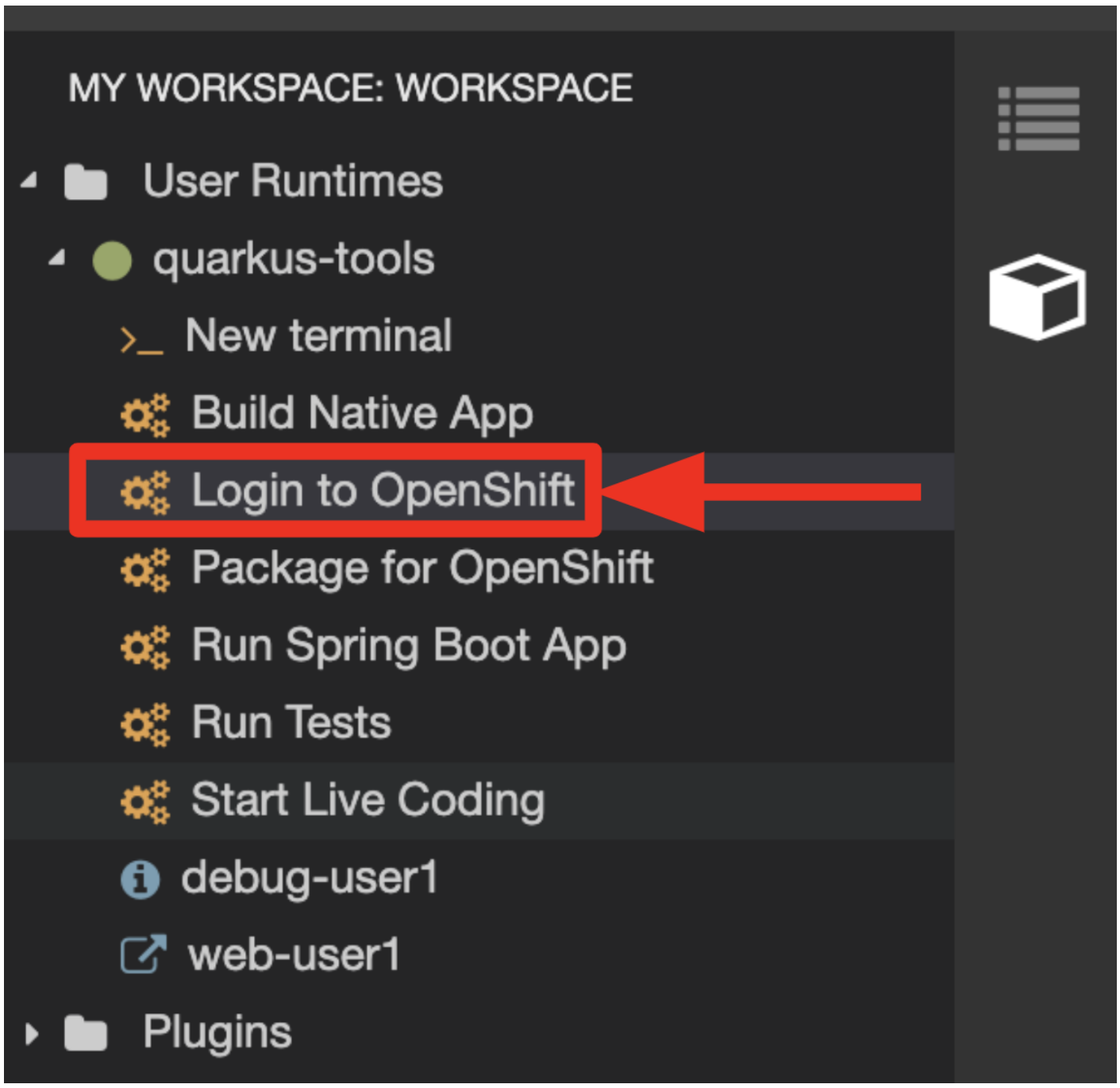


Login to OpenShift

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: click on **Login to OpenShift**, and enter your given credentials:

- Username: `user16`

- Password: `r3dh4t1!`



You should see something like this (the project names may be different):

```
NONE
Login successful.

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

  * user16-bookinfo
    user16-catalog
    user16-cloudnative-pipeline
    user16-cloudnativeapps
    user16-inventory
    user16-istio-system

Using project "user16-bookinfo".
Welcome! See 'oc help' to get started.
```

After you log in using **Login to OpenShift**, the terminal is no longer usable as a regular terminal. You can close the terminal window. You will still be logged in when you open more terminals later!

Do this if this is the first module you're doing today, otherwise continue to the Verifying the Dev Environment!

These commands reset and re-play all the steps from module 1 and should take 4-5 minutes to finish.

You do not need to run these if you already completed the Optimizing Existing Applications lab and have created `catalog` and `inventory` and the CoolStore `monolith` projects!

```
SH
sh /projects/cloud-native-workshop-v2m2-labs/monolith/scripts/deploy-inventory.sh user16 && \
sh /projects/cloud-native-workshop-v2m2-labs/monolith/scripts/deploy-catalog.sh user16 && \
sh /projects/cloud-native-workshop-v2m2-labs/monolith/scripts/deploy-coolstore.sh user16
```

It sometimes takes time to create a new build image for network latency in OpenShift. So if you got failed to deploy catalog-service with **Error from server (NotFound): services "catalog-springboot" not found**. Please try again with delay via the following command:

```
SH
sh /projects/cloud-native-workshop-v2m2-labs/monolith/scripts/deploy-inventory.sh user16 && \
sh /projects/cloud-native-workshop-v2m2-labs/monolith/scripts/deploy-catalog.sh user16 3m && \
sh /projects/cloud-native-workshop-v2m2-labs/monolith/scripts/deploy-coolstore.sh user16
```

Wait for the commands to complete.

Verifying the Dev Environment

In the previous module, you created a new OpenShift project called **user16-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. Anything that depends on the service can 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.

When referring to Kubernetes or OpenShift object types in `oc` commands, you can use short synonyms for long words, like **bc** instead of **buildconfig**, **is** for **imagestream**, **dc** for **deploymentconfig**, **svc** for **service**, etc.

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

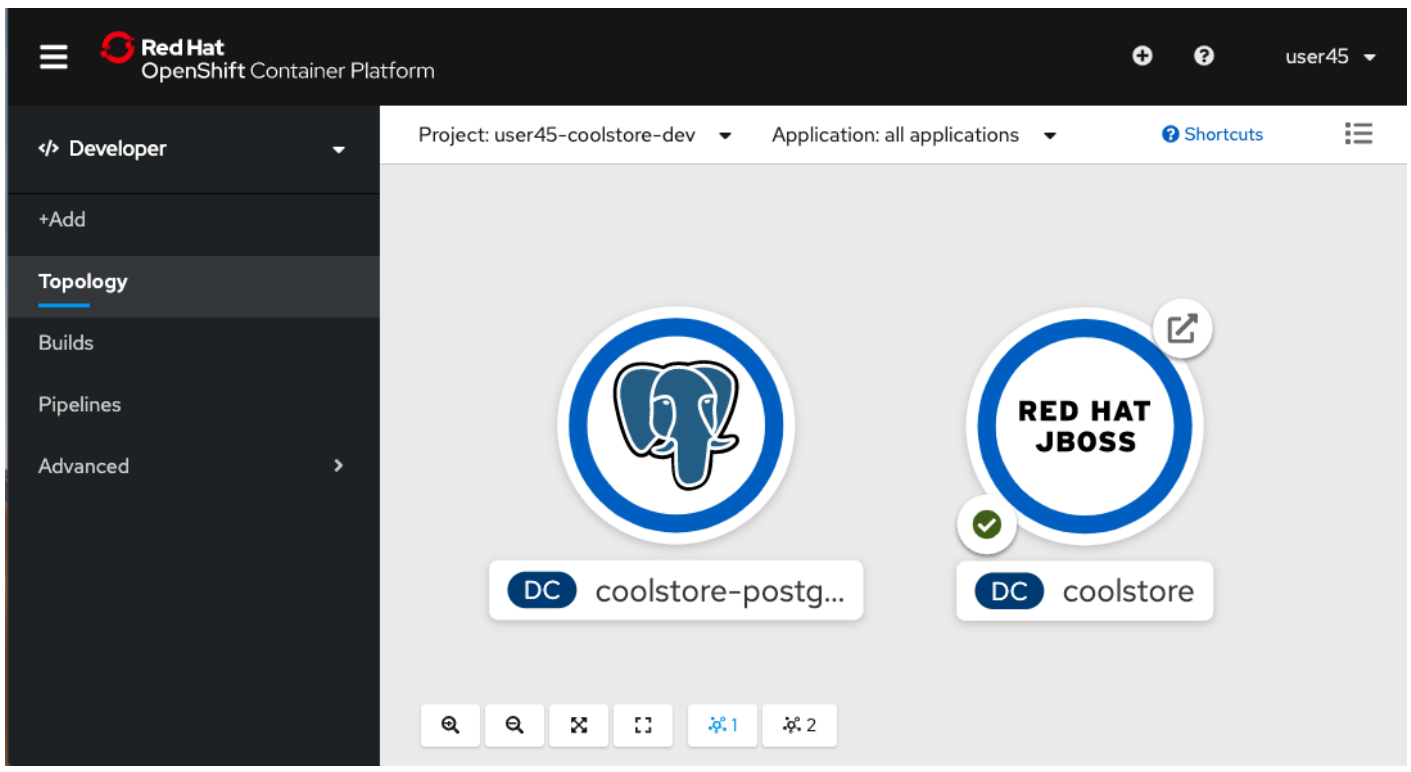
Run these commands to inspect the elements via CodeReady Workspaces Terminal window:

```
oc project user16-coolstore-dev && \
oc get bc coolstore && \
oc get is coolstore && \
oc get dc coolstore && \
oc get svc coolstore && \
oc describe route www
```

SH

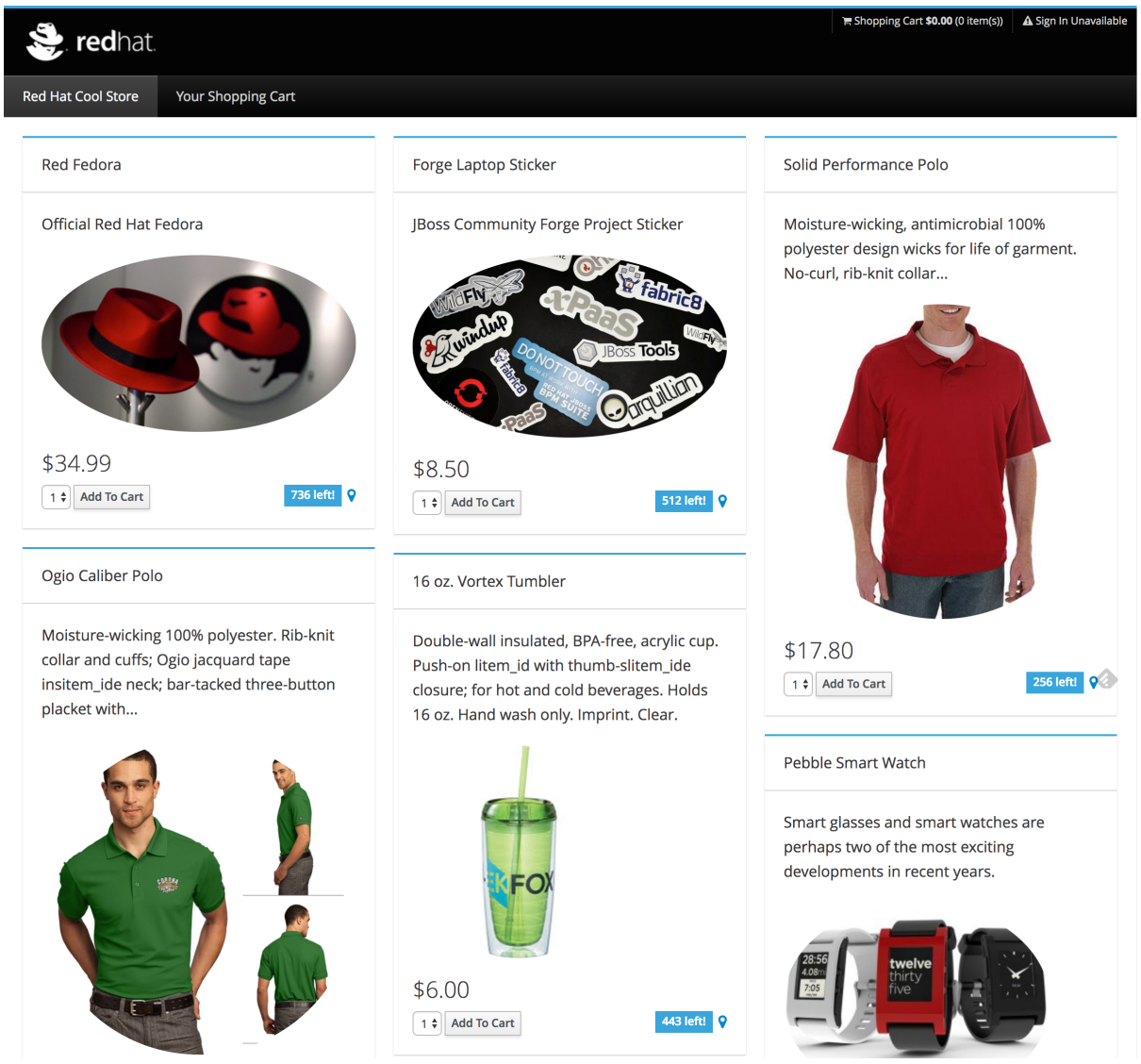
You should get a valid value for each, and no errors!

Visit the Topology View (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-coolstore-dev) to see the coolstore app (and its database). The **Topology** view in the *Developer* perspective of the OpenShift web console provides a visual representation of all the applications within a project, their build status, and the components and services associated with them. You'll visit this often:



Verify that you can access the monolith by clicking on the route link (the arrow) to access the running monolith:

The coolstore web interface should look like:



Verify Database

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

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

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

```sh
psql -U $POSTGRESQL_USER $POSTGRESQL_DATABASE -c 'select name,price from PRODUCT_CATALOG;' | cat; exit
```

You should see the following:

```
          name          | price
------------------------+-------
 Red Fedora             | 34.99
 Forge Laptop Sticker   |   8.5
 Solid Performance Polo |  17.8
 Ogio Caliber Polo      | 28.75
 16 oz. Vortex Tumbler  |     6
 Pebble Smart Watch     |    24
 Oculus Rift            |   106
 Lytro Camera           |  44.3
 Atari 2600 Joystick    |   240
(9 rows)
```

This shows the content of the monolith's database.

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

In the previous scenarios, you deployed the Coolstore monolith using an OpenShift Template into the **user16-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

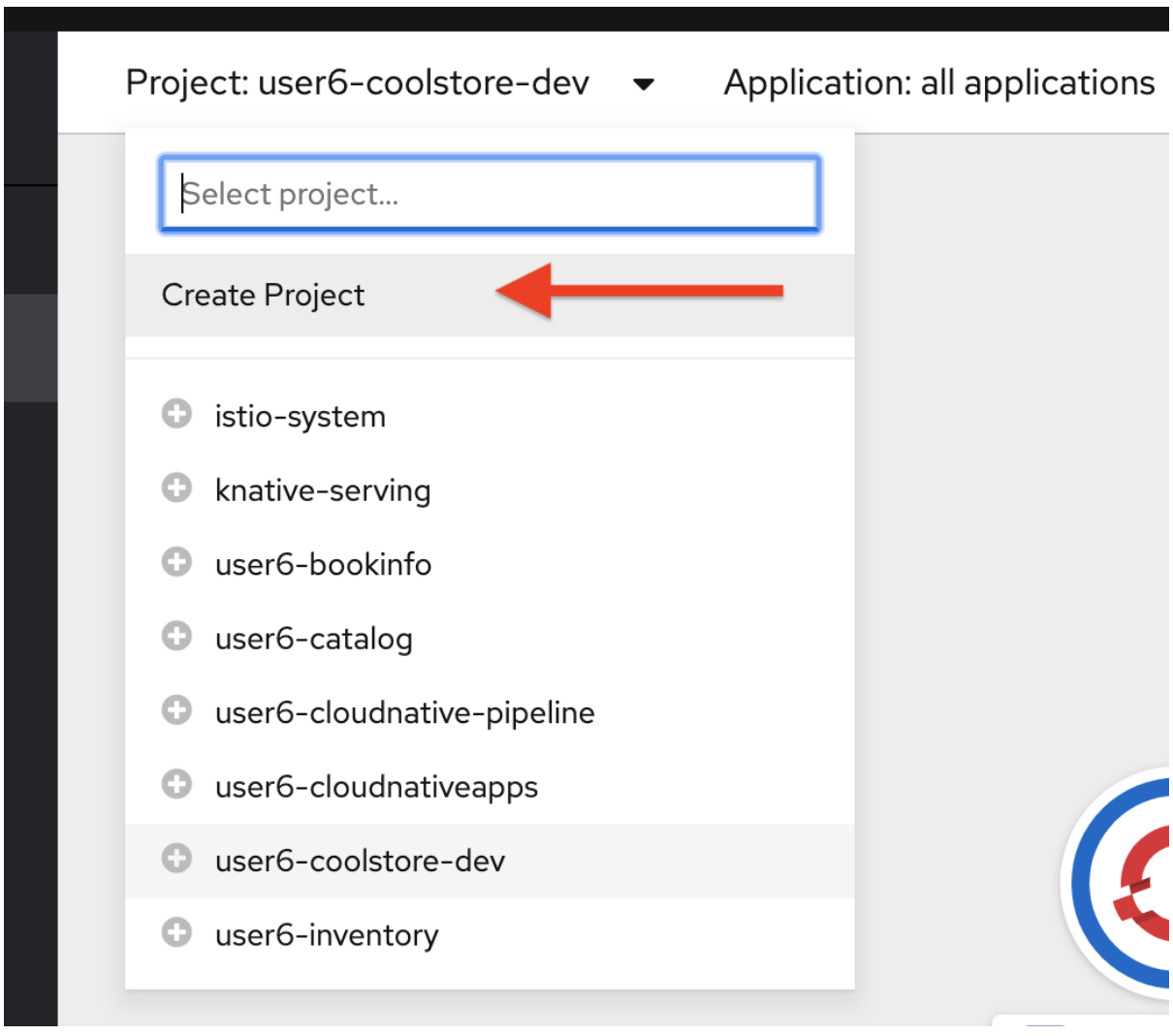
The existing **user16-coolstore-dev** project is used as a developer environment for building new versions of the app after code changes and deploying them to the development environment.

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

Back in the Topology View (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-coolstore-dev), click on the project drop-down and select **Create Project**.



Fill in the fields, and click `Create`:

- Name: **user16-coolstore-prod**
- Display Name: **user16 Coolstore Monolith - Production**
- Description: *leave this field empty*

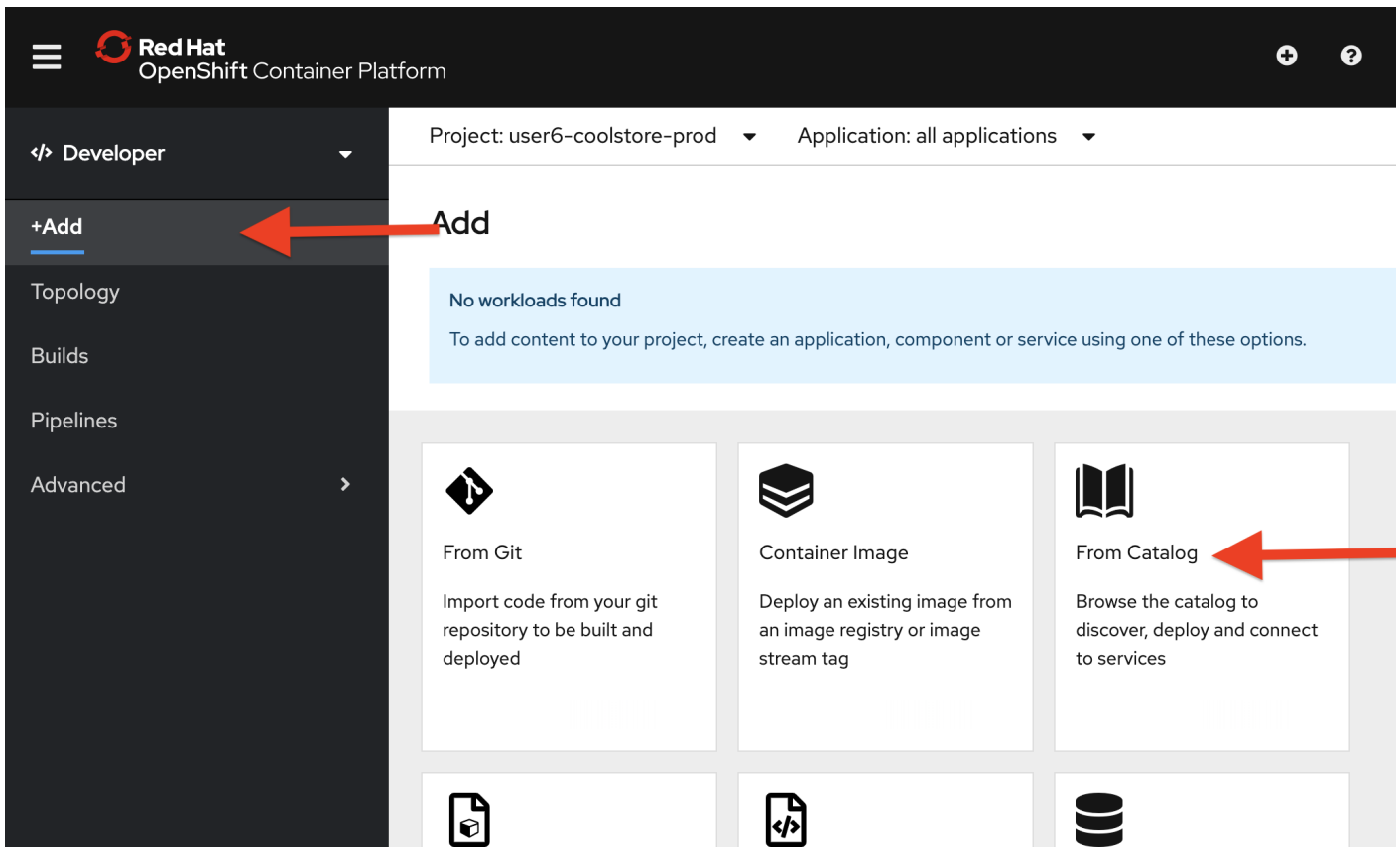
> YOU `MUST` USE `user16-coolstore-prod` AS THE PROJECT NAME, as this name is referenced later on and you will experience failures if you do not name it `user16-coolstore-prod`.

This will create a new OpenShift project called **user16-coolstore-prod** from which our production application will run.

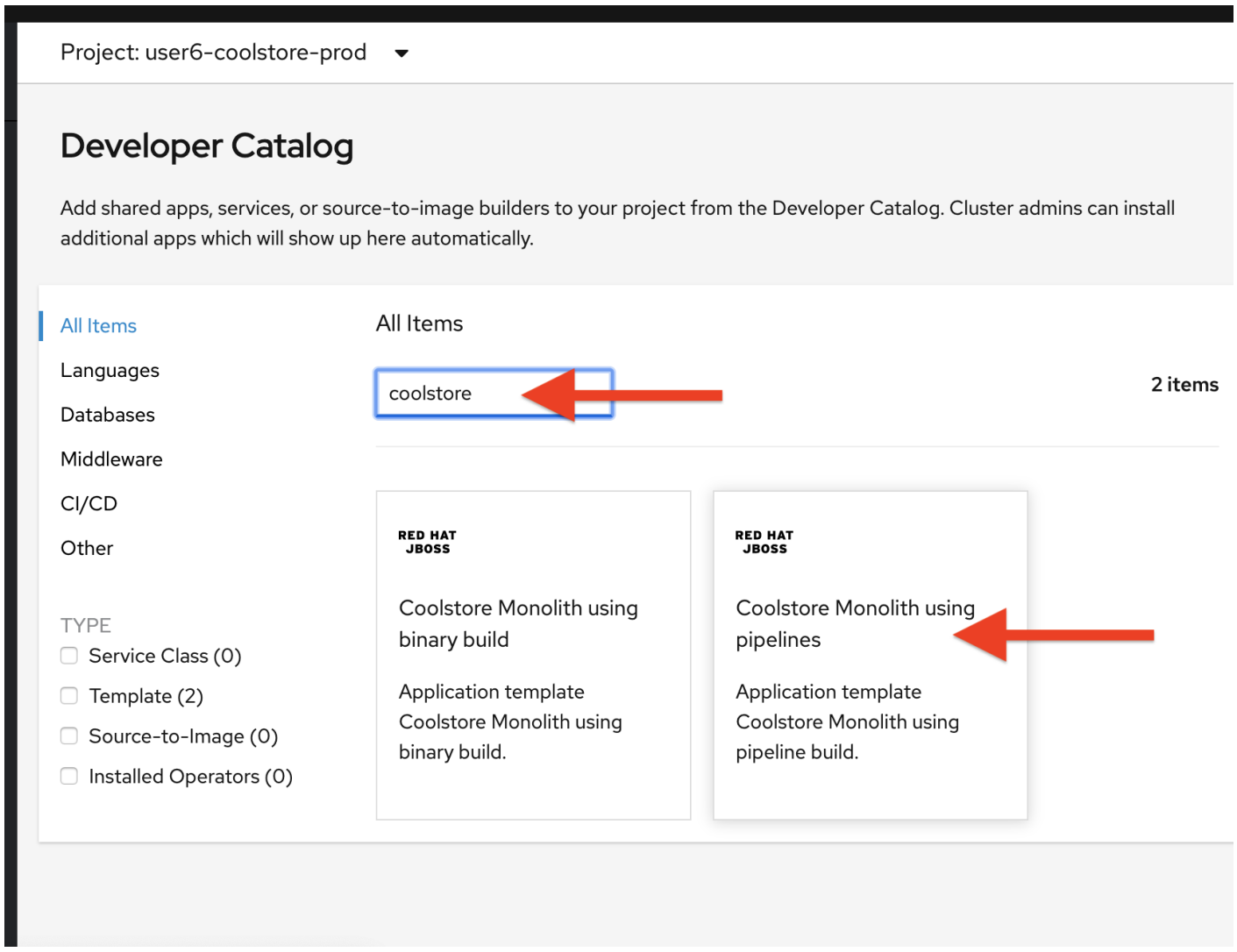
2. Add the production elements

In this case we'll use the production template to create the objects.

We've pre-installed an application *template* for use. Click the **+Add** and then **From Catalog** item:

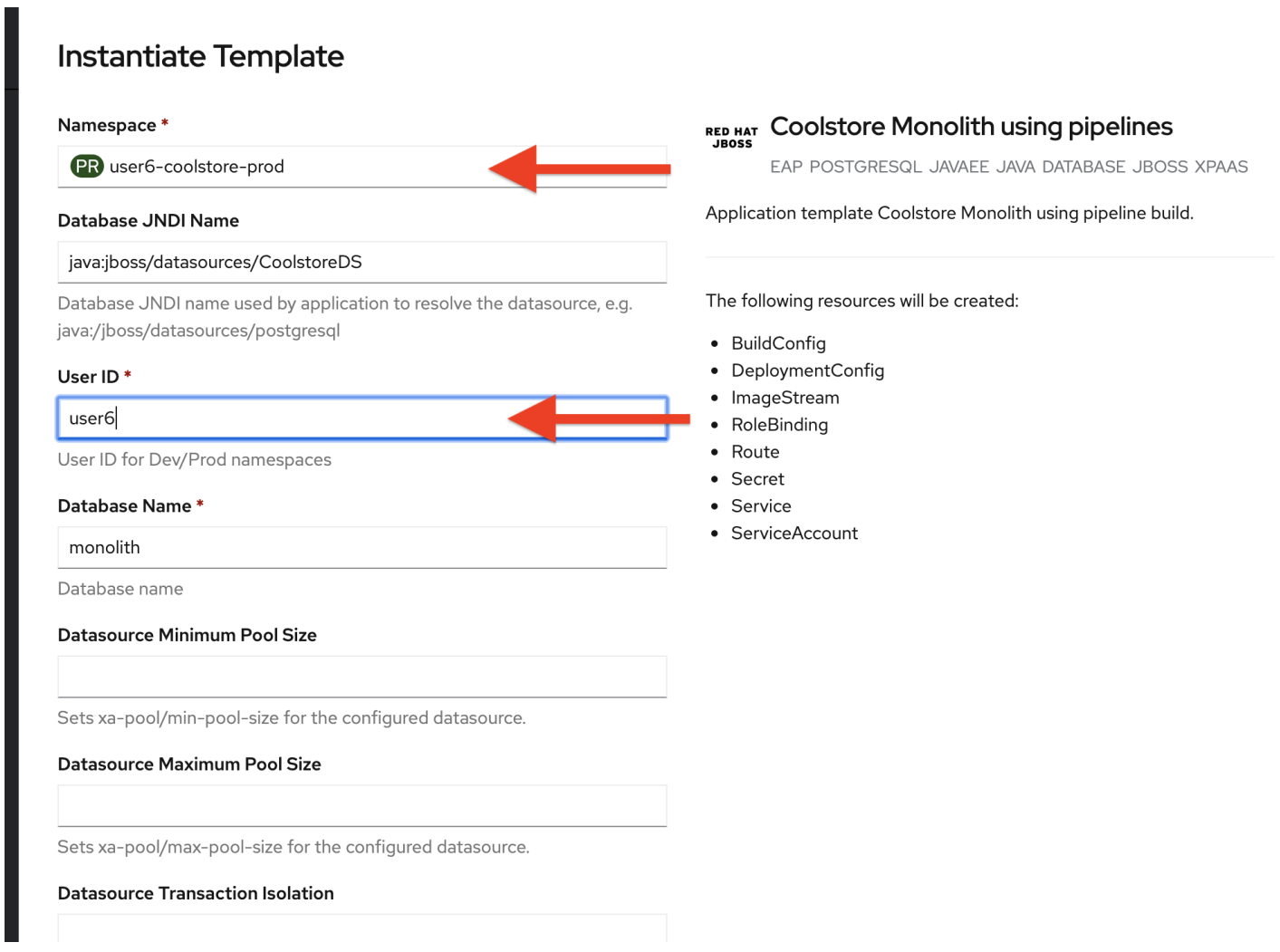


In the search box, type in `coolstore` and choose *Coolstore Monolith using pipelines* and then click **Instantiate Template**.



Fill in the following fields:

- **Namespace**: `user16-coolstore-prod` (this should already be selected)
- **User ID**: `user16`



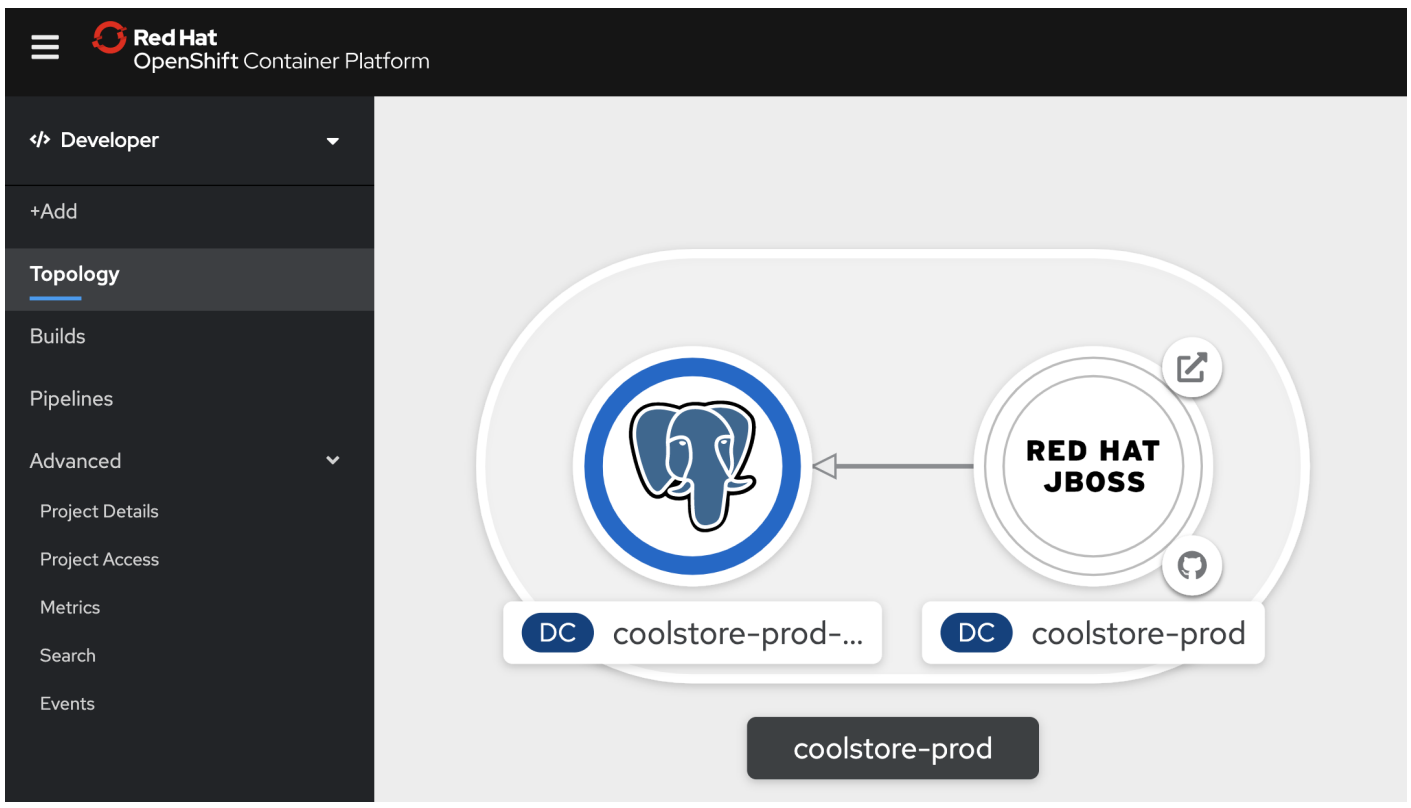
Keep the other values as-is and scroll down and click **Create**.

Go to the Topology View (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-coolstore-prod) to see the elements that were deployed.

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

Label the components so that they get proper icons by running this command in the CodeReady Terminal:

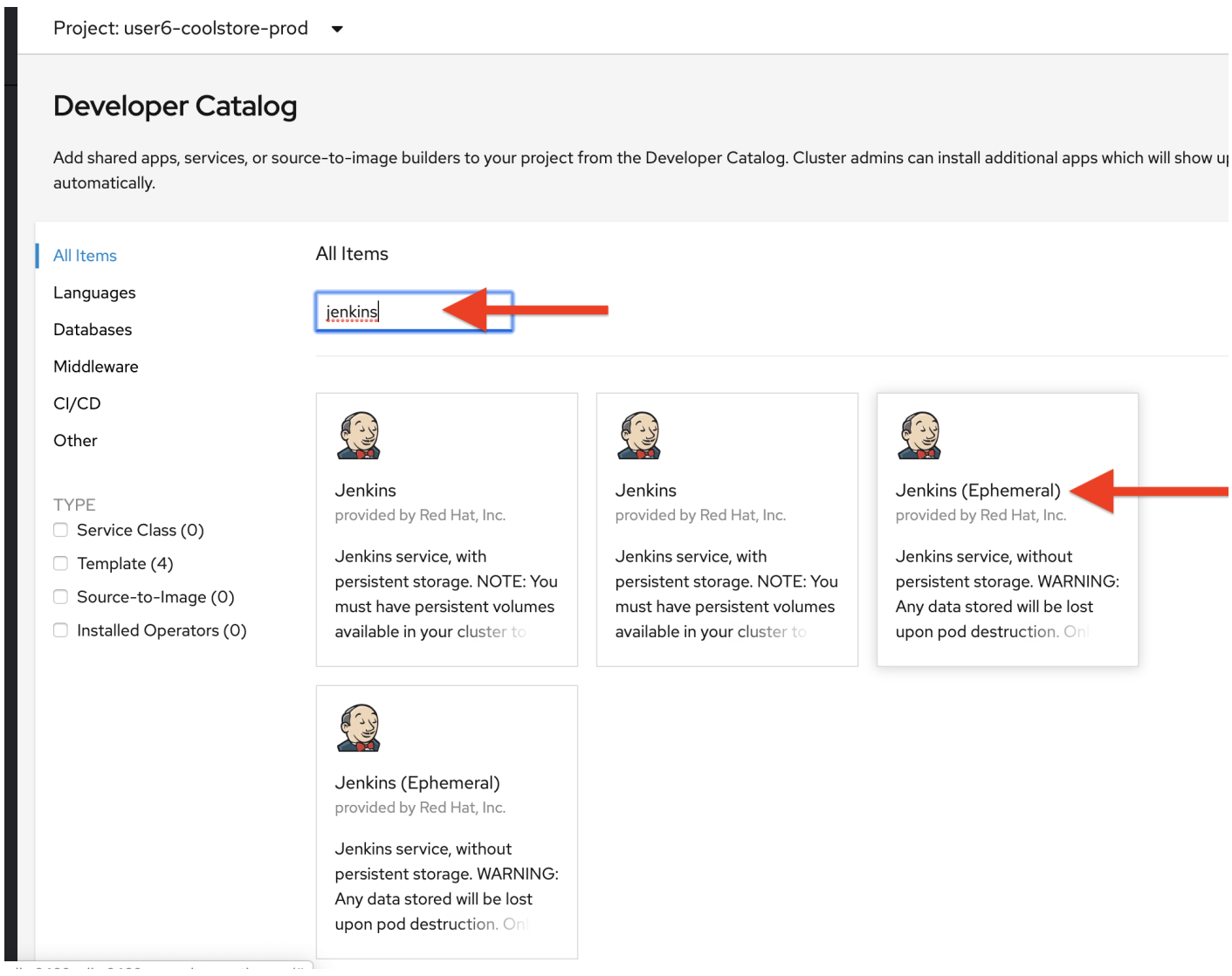
```sh
oc project user16-coolstore-prod && \
oc label dc/coolstore-prod-postgresql app.openshift.io/runtime=postgresql --overwrite && \
oc label dc/coolstore-prod app.openshift.io/runtime=jboss --overwrite && \
oc label dc/coolstore-prod-postgresql app.kubernetes.io/part-of=coolstore-prod --overwrite && \
oc label dc/coolstore-prod app.kubernetes.io/part-of=coolstore-prod --overwrite && \
oc annotate dc/coolstore-prod app.openshift.io/connects-to=coolstore-prod-postgresql --overwrite && \
oc annotate dc/coolstore-prod app.openshift.io/vcs-uri=https://github.com/RedHat-Middleware-Workshops/cloud-native-workshop-v2m2-labs.git --
overwrite && \
oc annotate dc/coolstore-prod app.openshift.io/vcs-ref=ocp-4.4 --overwrite
```



You can see the *postgres* database running (with a dark blue circle), and the coolstore monolith, which has not yet been deployed or started. In previous labs we deployed manually from a binary build of our app in our developer project. In this lab we will use a *CI/CD pipeline* to build and deploy automatically go our production environment.

We will use a **Jenkins Server** in our project and use a *Jenkins Pipeline* build strategy.

Click **Add** then click **From Catalog**, type in `jenkins` in the search box, and choose the FIRST **Jenkins (ephemeral)** item:



Click **Instantiate Template** and change the following fields, leaving other fields alone:

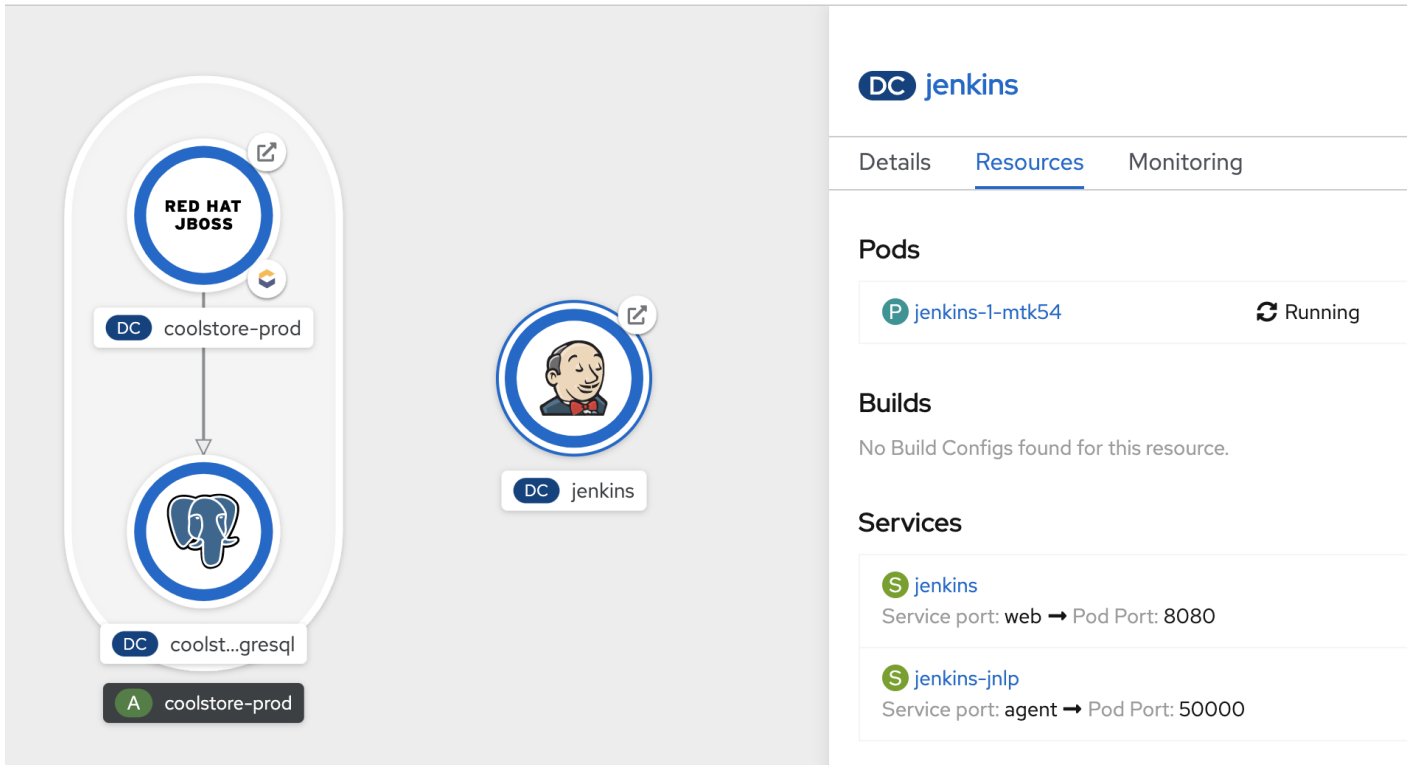
- **Namespace**: `user16-coolstore-prod` (this should already be selected)
- **Memory Limit**: `2Gi`
- **Disable memory intensive administrative monitors**: `true`

Click **Create**.

Let's label the new Jenkins server:

```sh
oc label dc/jenkins app.openshift.io/runtime=jenkins --overwrite
```

Back on the Topology View (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-coolstore-prod) 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

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 (https://jenkins.io/doc/book/pipeline/overview/) into the platform and enables defining truly complex workflows directly from within OpenShift. OpenShift 4 also introduces Tekton Pipelines (https://www.openshift.com/learn/topics/pipelines), an evolution of CI/CD for Kubernetes, which you might be exploring in a later module if it's on the agenda for today.

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 (https://github.com/RedHat-Middleware-Workshops/cloud-native-workshop-v2m2-labs) we've been using.

OpenShift has built-in support for **Jenkins CI/CD pipelines** by allowing developers to define a Jenkins pipeline (https://jenkins.io/solutions/pipeline/) for execution by a Jenkins automation engine.

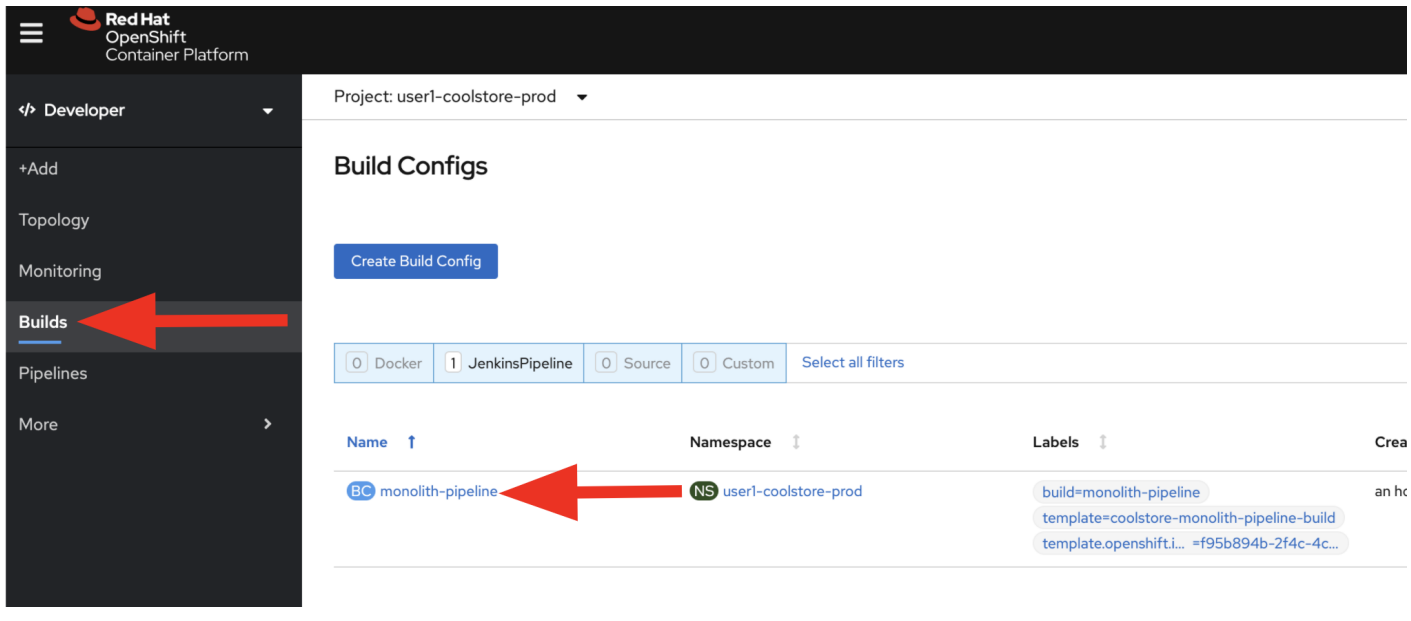
The build can get started, monitored, and managed by OpenShift in the same way as any other build types e.g. S2I. Pipeline workflows are defined in a `Jenkinsfile`, either embedded directly in the build configuration, or supplied in Git repository and referenced by the build configuration. They are written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

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.

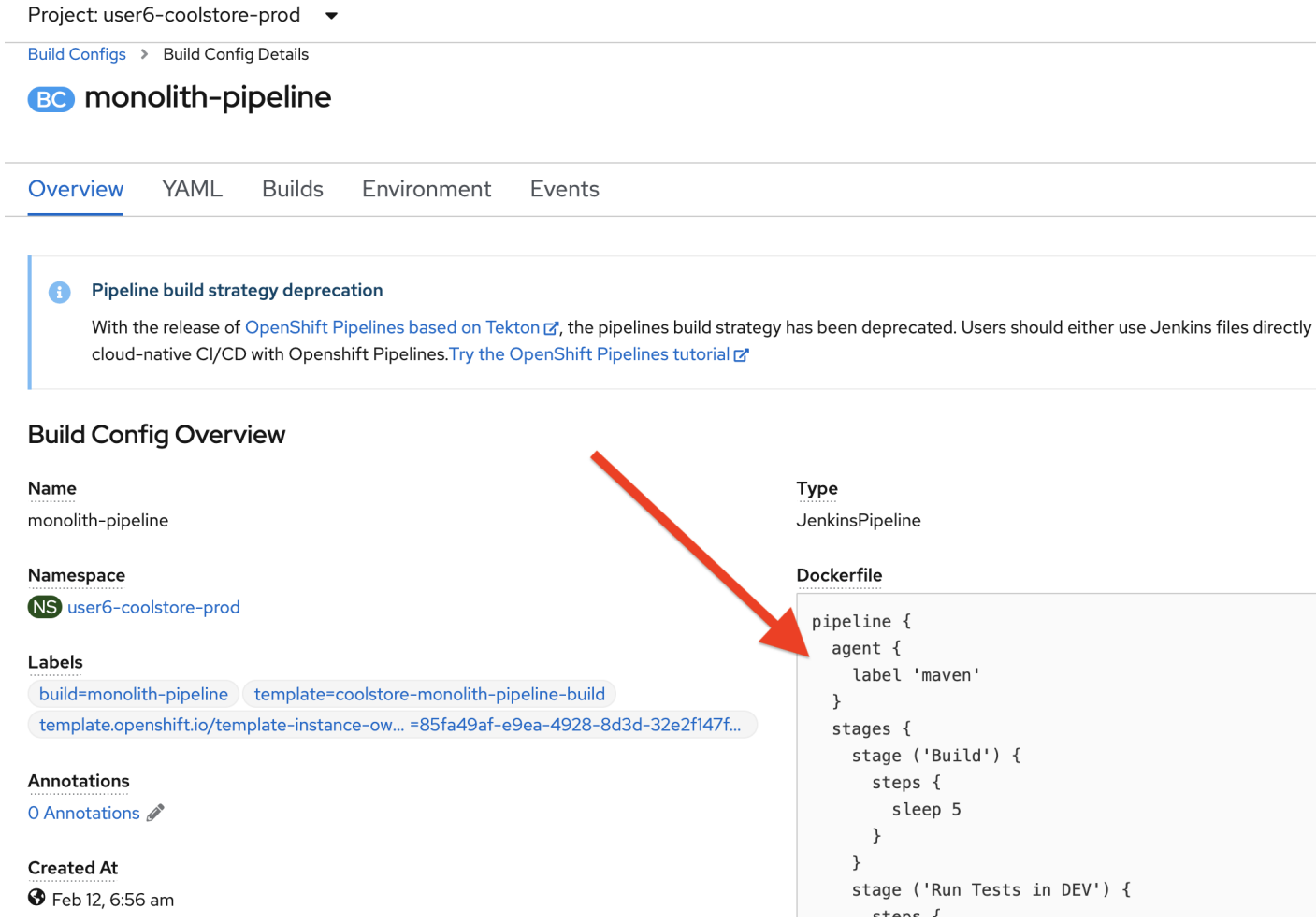
3. Inspect the Pipeline Definition

You may notice a *Pipelines* menu item on the left menu of the OpenShift Console. This menu item is for exploring *Tekton Pipelines*, which is a newer pipeline technology based on the Tekton project. There are other modules as part of this workshop that explore Tekton. For now we'll concentrate on Jenkins.

Our pipeline is somewhat simplified for the purposes of this Workshop. Inspect the contents of the pipeline by navigating to the Build Config page (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/k8s/ns/user16-coolstore-prod/buildconfigs) and click on `monolith-pipeline`:



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 -n user16-coolstore-prod
```

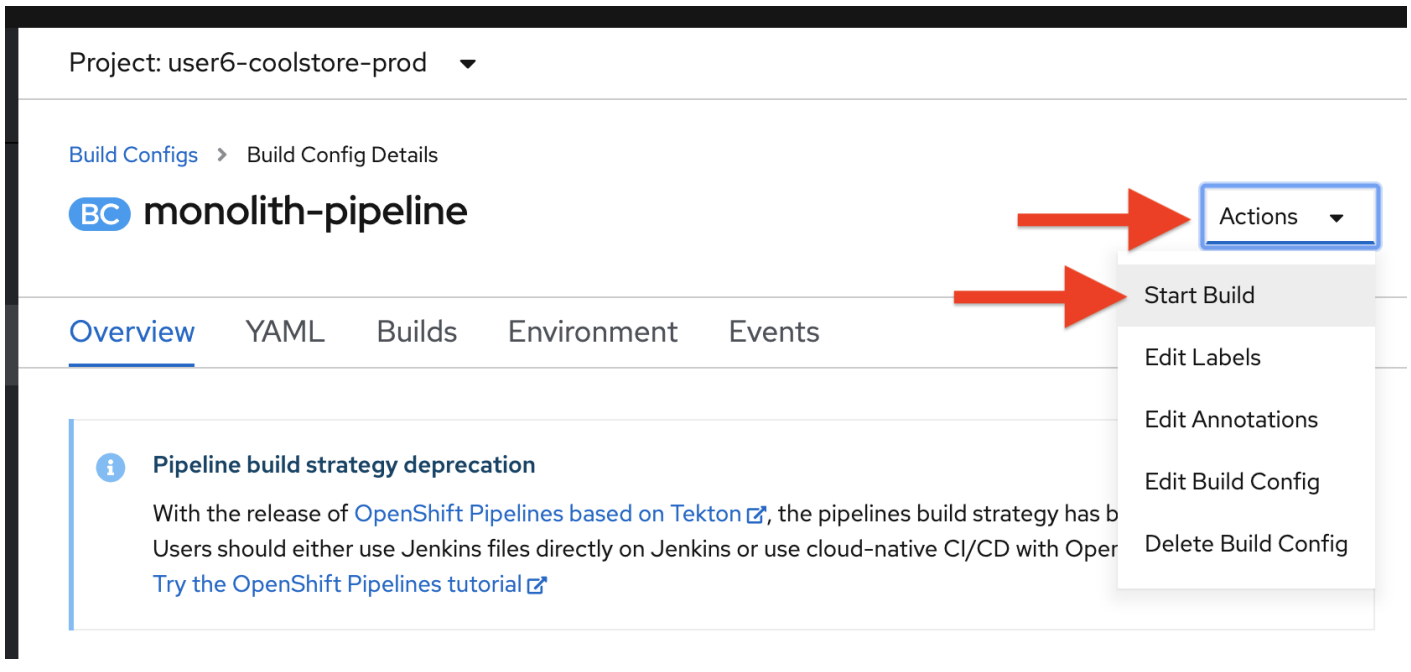
The pipeline syntax allows creating complex deployment scenarios with the possibility of defining checkpoints for manual interaction and approval processes using the large set of steps and plugins that Jenkins provides (https://jenkins.io/doc/pipeline/steps/) in order to adapt the pipeline to the processes used in your team.

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.

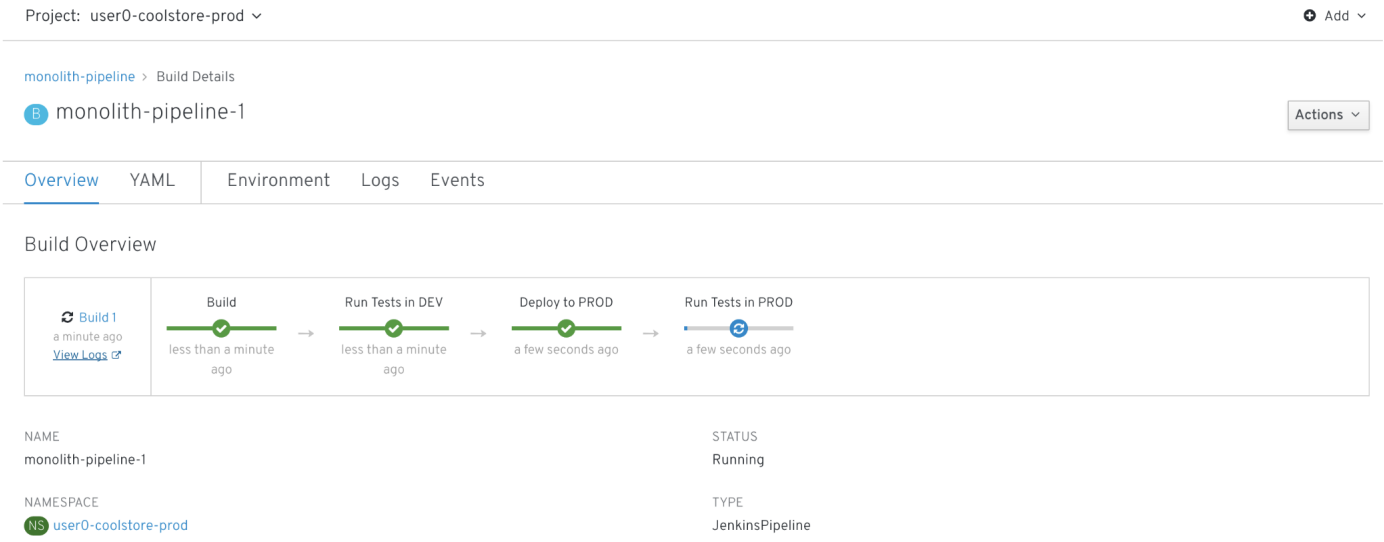
4. Promote the dev image to production using the pipeline

Let's invoke the build pipeline.

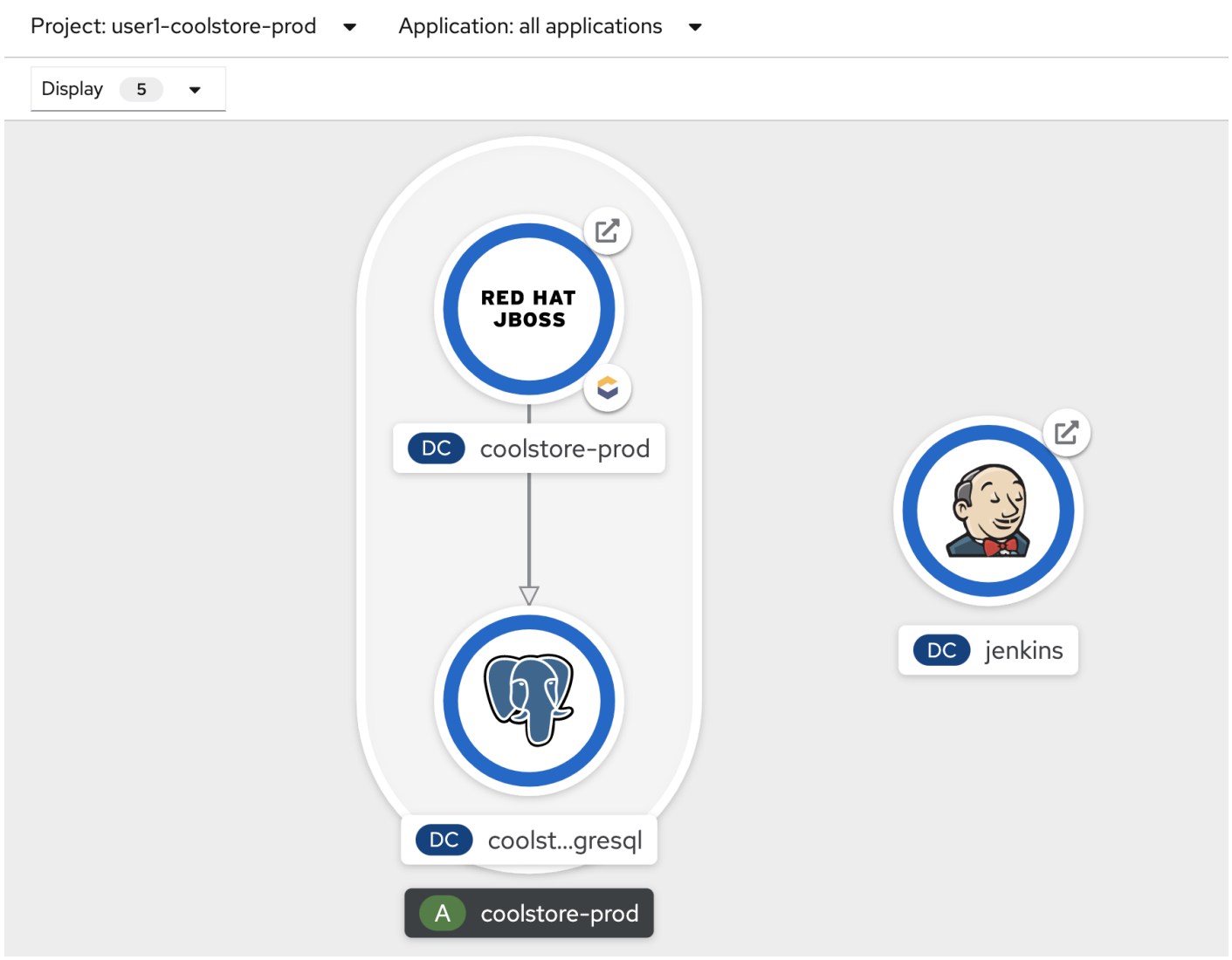
On the Pipeline Details Page (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/k8s/ns/user16-coolstore-prod/buildconfigs/monolith-pipeline), select *Actions > Start Build*:



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 Topology View (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-coolstore-prod) 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.

5. 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.

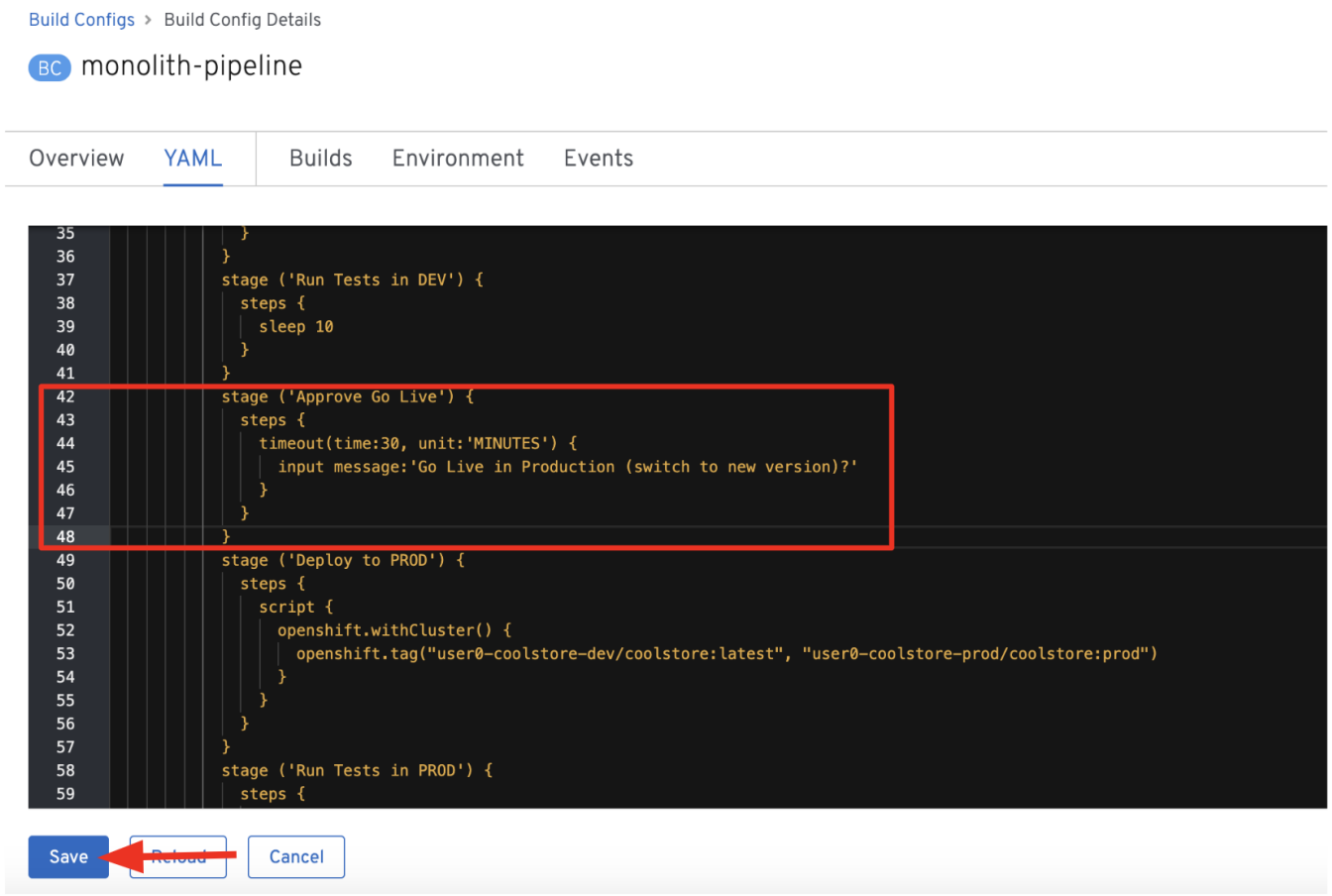
Ordinarily your pipeline definition would be checked into a source code management system like Git, and to change the pipeline you'd edit the *Jenkinsfile* in the source base. For this workshop we'll just edit it directly to add the necessary changes. You can edit it with the **oc** command but we'll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

Back on the Pipeline Details Page (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/k8s/ns/user16-coolstore-prod/buildconfigs/monolith-pipeline) click on the **YAML** tab. Add *a new stage* to the pipeline, just before the *Deploy to PROD* stage:

You will need to copy and paste the below code into the right place of **BuildConfig** as shown in the below image.

```groovy
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, in CodeReady, in the `cloud-native-workshop-v2m2-labs` project, open *monolith/src/main/webapp/app/css/coolstore.css*, 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):

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

Now we need to update the catalog endpoint in the monolith application. Run the following commands in a Terminal to update the baseUrl to the proper value with your username:

```sh
JSPATH="$CHE_PROJECTS_ROOT/cloud-native-workshop-v2m2-labs/monolith/src/main/webapp/app/services/catalog.js"
CATALOGHOST=$(oc get route -n user16-catalog catalog-springboot -o jsonpath="{.spec.host}")
sed -i 's/REPLACEURL/'$CATALOGHOST'/' "$JSPATH"
```

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

```sh
mvn clean package -Popenshift -DskipTests -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m2-labs/monolith
```

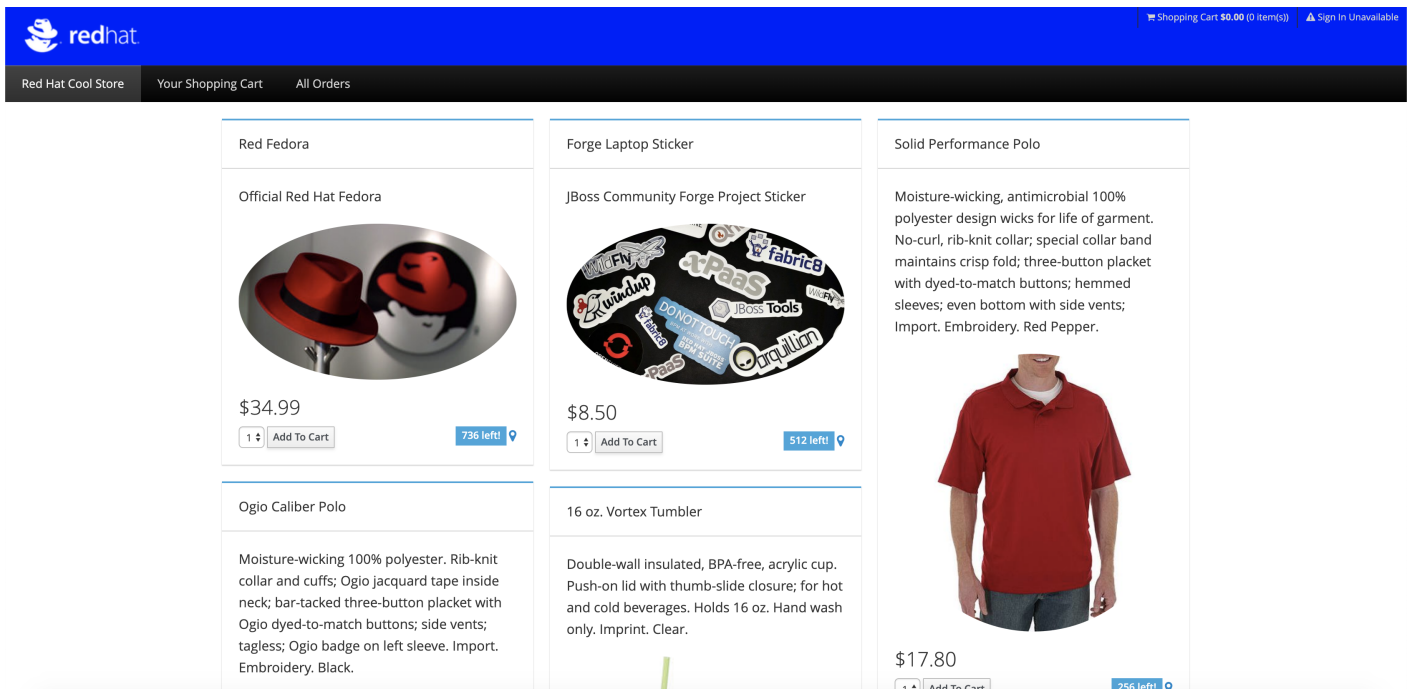
And re-deploy it to the **dev** environment using a binary build just as we did before via CodeReady Workspaces Terminal:

```
oc start-build -n user16-coolstore-dev coolstore --from-file=$CHE_PROJECTS_ROOT/cloud-native-workshop-v2m2-labs/monolith/deployments/ROOT.war --follow
```
SH

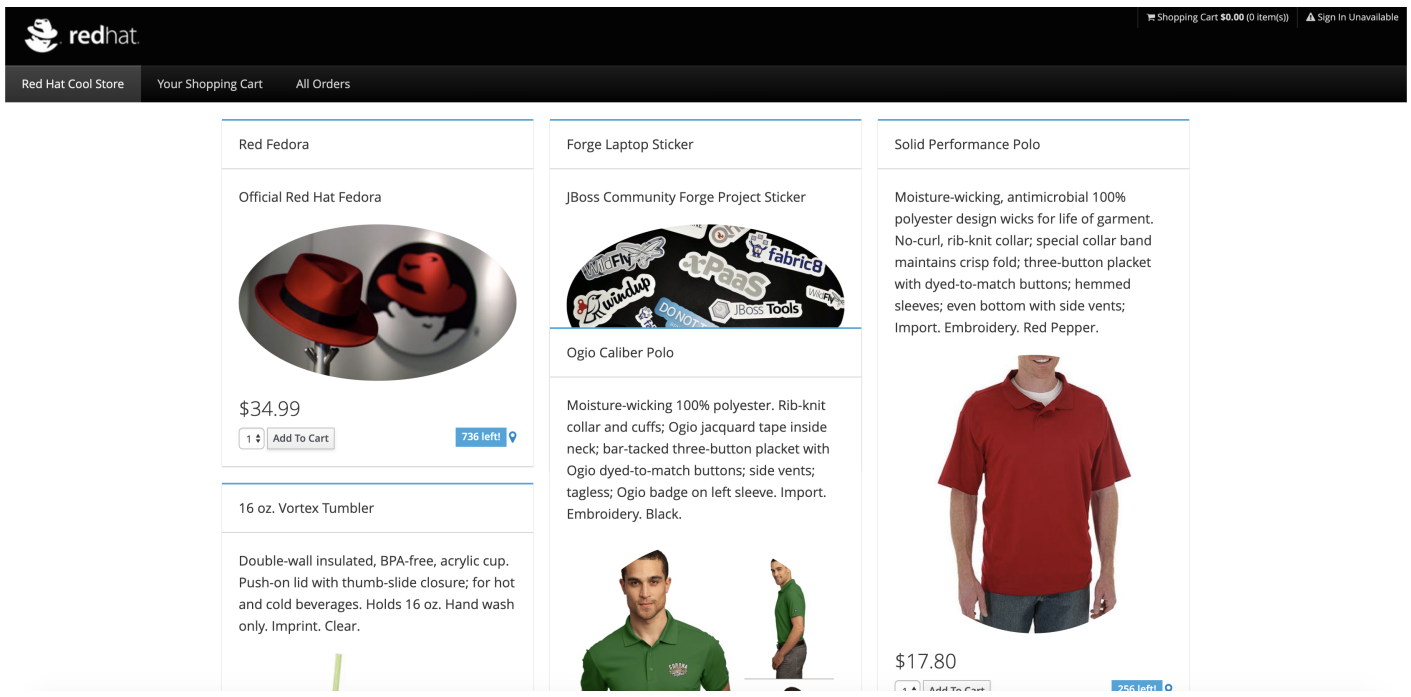
Wait for it to complete the deployment via CodeReady Workspaces Terminal:

```
oc -n user16-coolstore-dev rollout status -w dc/coolstore
```
SH

And verify that the blue header is visible in the **dev** application by navigating to the Coolstore Dev Web frontend (http://www-user16-coolstore-dev.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com). It should look like the following:



Confirm the Coolstore Prod Web frontend (http://www-user16-coolstore-prod.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com) is still using the original black header:

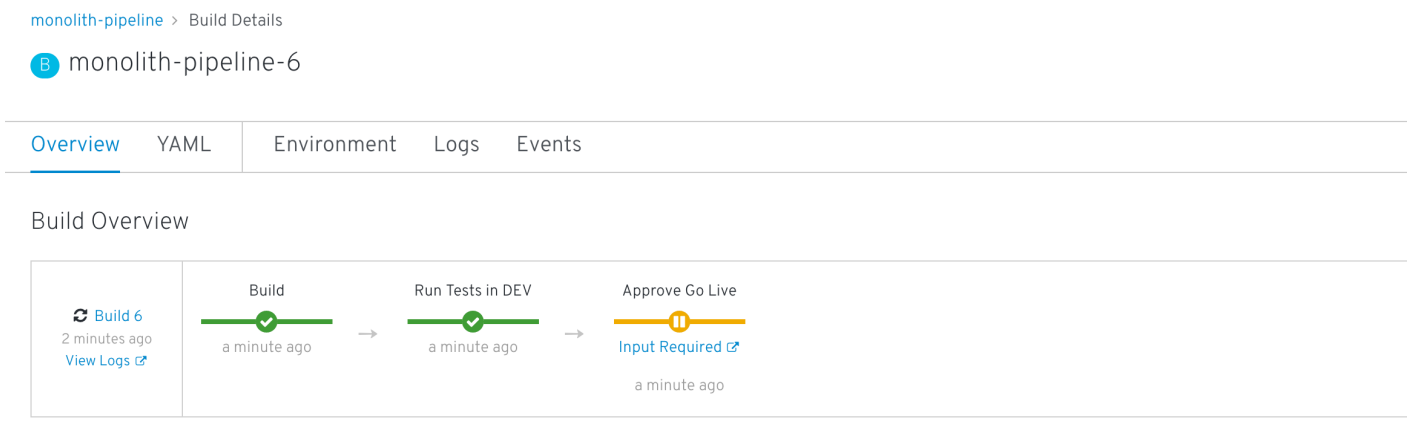


We're happy with this change in **dev**, so let's promote the new change to **prod**, using the new approval step!

7. Run the pipeline again

Invoke the pipeline once more by navigating to the Pipeline Details Page (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/k8s/ns/user16-coolstore-prod/buildconfigs/monolith-pipeline), select *Actions > Start Build*.

The same pipeline progress will be shown, however before deploying to prod, you will see a prompt in the pipeline:

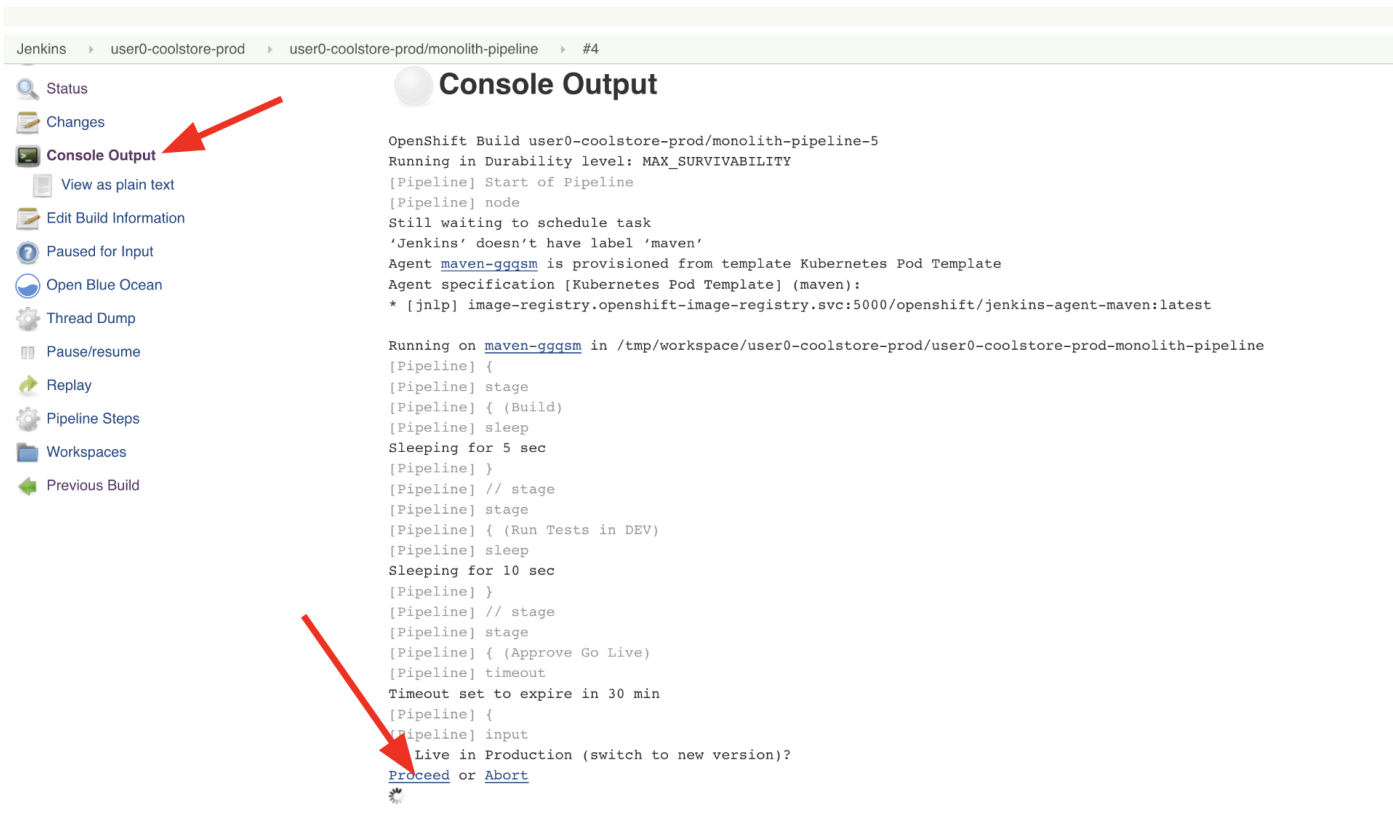


Click on the link for **Input Required**. This will open a new tab and direct you to Jenkins itself, where you can login with the same credentials as OpenShift:

- Username: `user16`
- 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`.



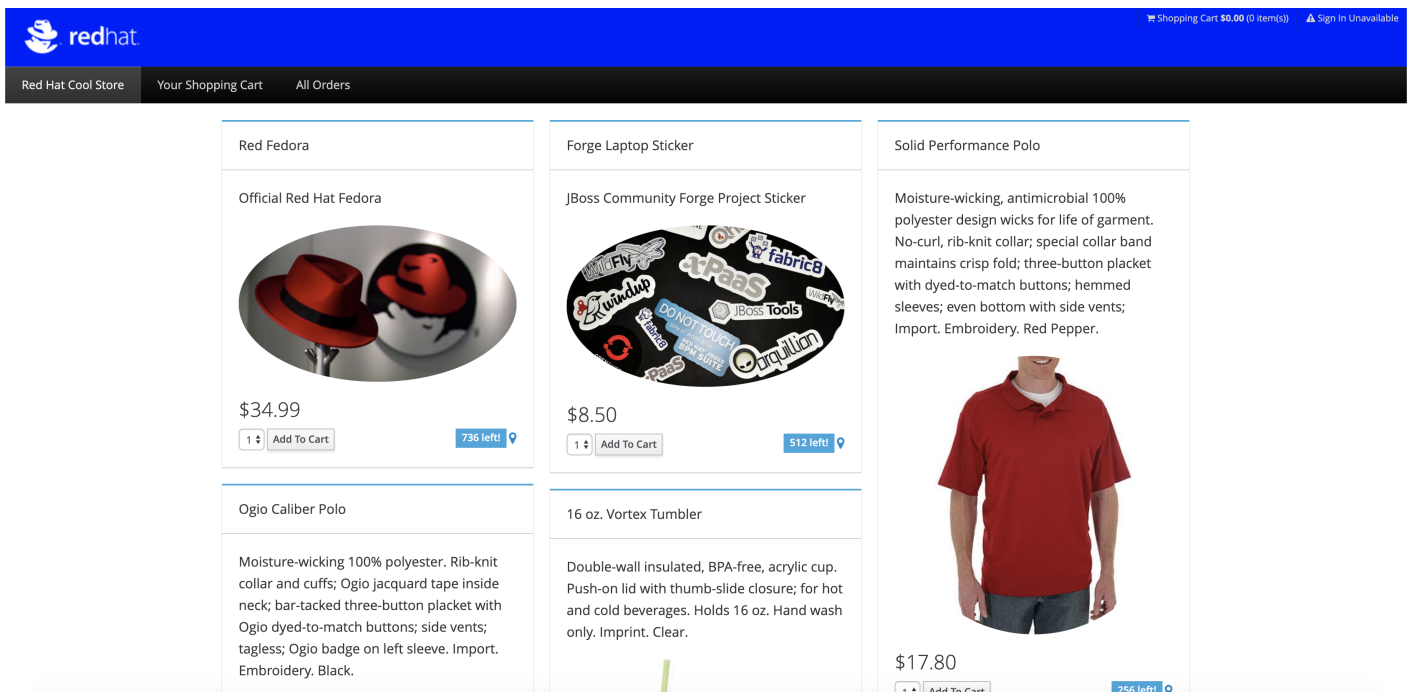
8. Approve the change to go live

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.

On the Production Topology View (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-coolstore-prod), wait for the production deployment to complete (and you get all blue circles!).

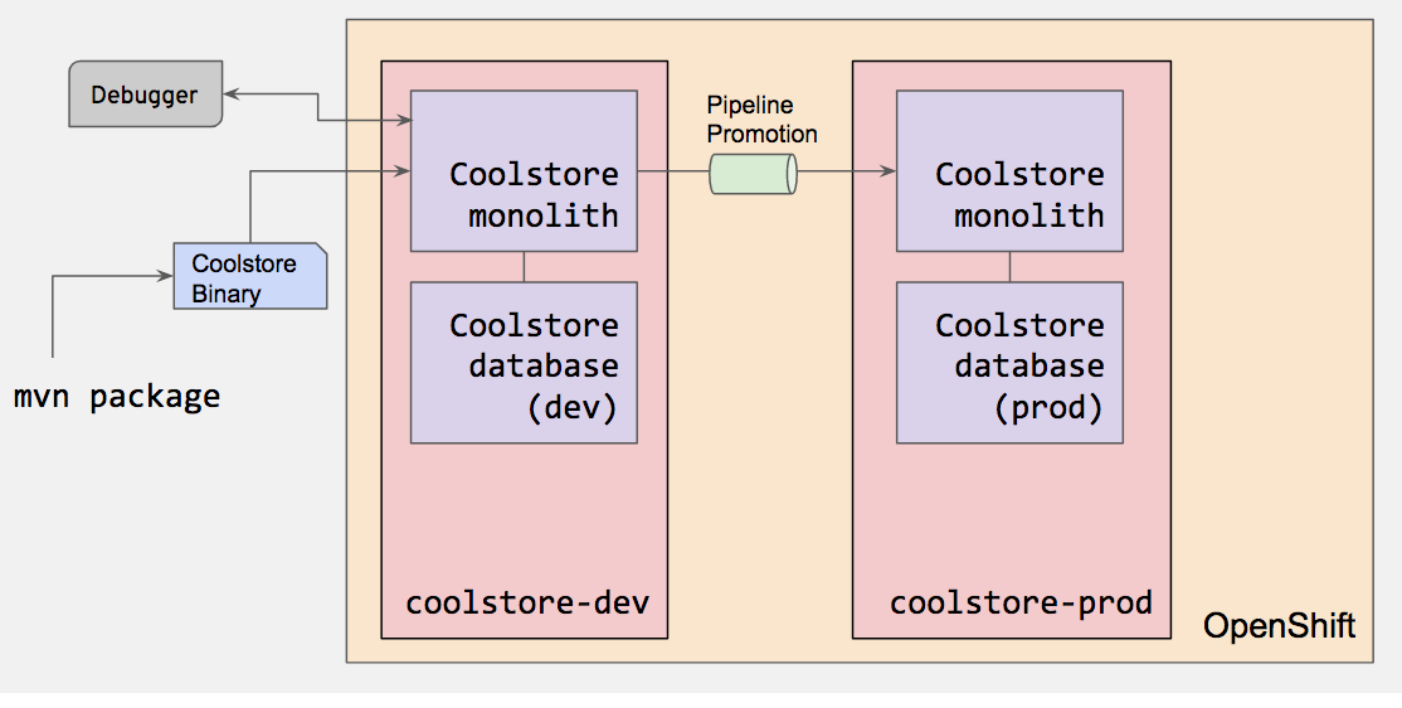
Once it completes, verify that the Coolstore Prod Web frontend (http://www-user16-coolstore-prod.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com) has the new change (blue header):



It may take a few seconds for the new app to be available, since we've not configured a proper health check and are not using *Rolling Updates* when deploying new apps. Real production environments would use this to ensure no downtime.

If you don't see a blue header, or you get *Application Not Available* errors, wait a few moments and reload. If you are still getting a black header, try holding the shift key while clicking the browser refresh button to force a hard refresh.

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

In this lab, you will debug the coolstore microservice 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 (https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/architecture.html).

Quarkus in development mode enables "Live Coding" 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 properties files and even `pom.xml` changes.

When run in Developer Mode (i.e. `mvn quarkus:dev`), Quarkus will also listen for debugging sessions on port `5005` (by default). 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`.

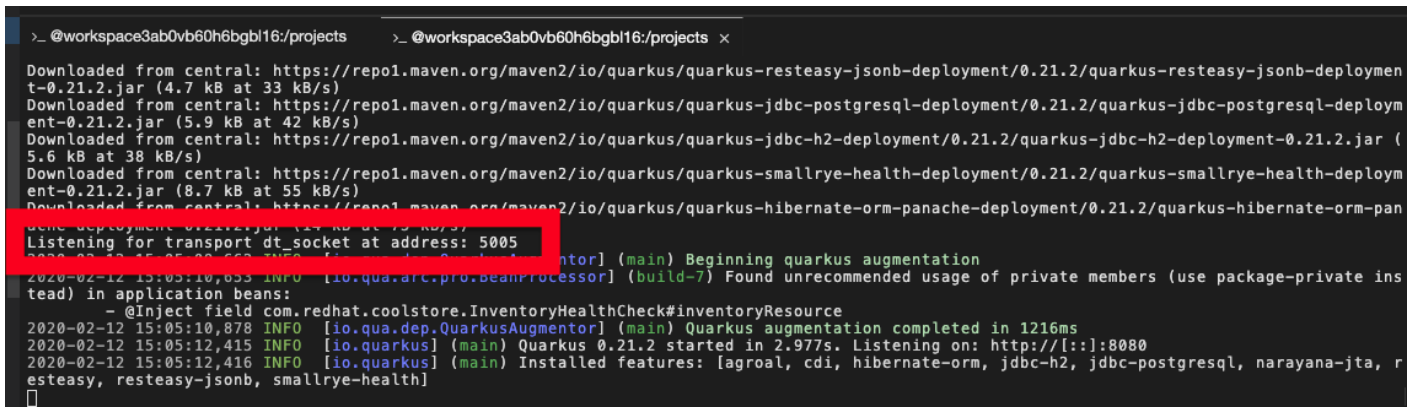
Start up the Inventory microservice locally, in development mode, using this command in a CodeReady Terminal: Enable remote debugging on Inventory by running the following command in the CodeReady Workspaces Terminal window:

```
mvn quarkus:dev -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m2-labs/inventory
```
SH

Close any popups about port 5005 or 8080 when the app starts up

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. Add a bug

Let's add a new endpoint that has a bug we will fix using the debugger.

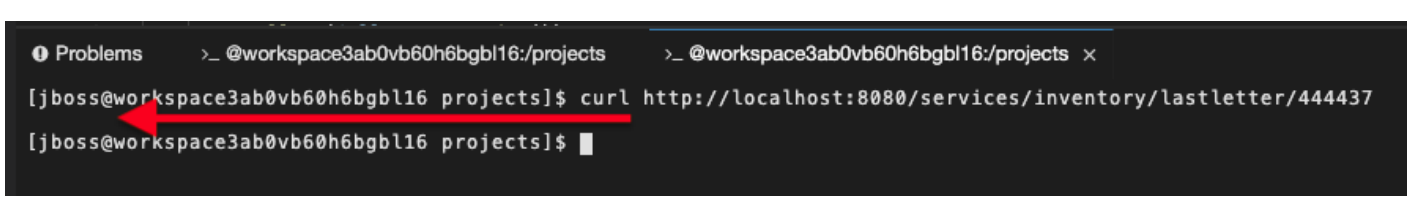
Go back to the *Explorer* view, and under the `cloud-native-workshop-v2m2-labs` project, open the `inventory/src/main/java/com/redhat/coolstore/InventoryResource.java` file. Add a new method which has an off-by-one error:

```java
@GET
@Path("/lastletter/{itemId}")
@Produces("application/text-plain")
public String lastLetter(@PathParam("itemId") String itemId) {
    Inventory item = Inventory.find("itemId", itemId).firstResult();
    String location = item.location;
    int len = location.length();
    String lastLetter = location.substring(len);
    return lastLetter;
}
```

This method retrieves the location of inventory, and outputs the last letter in the name of the location. After adding this method, try it out with item `165613`, which has inventory in `Seoul` and we'd expect the last letter `l` to be output. Open another Terminal and run the command:

```sh
curl http://localhost:8080/services/inventory/lastletter/165613
```

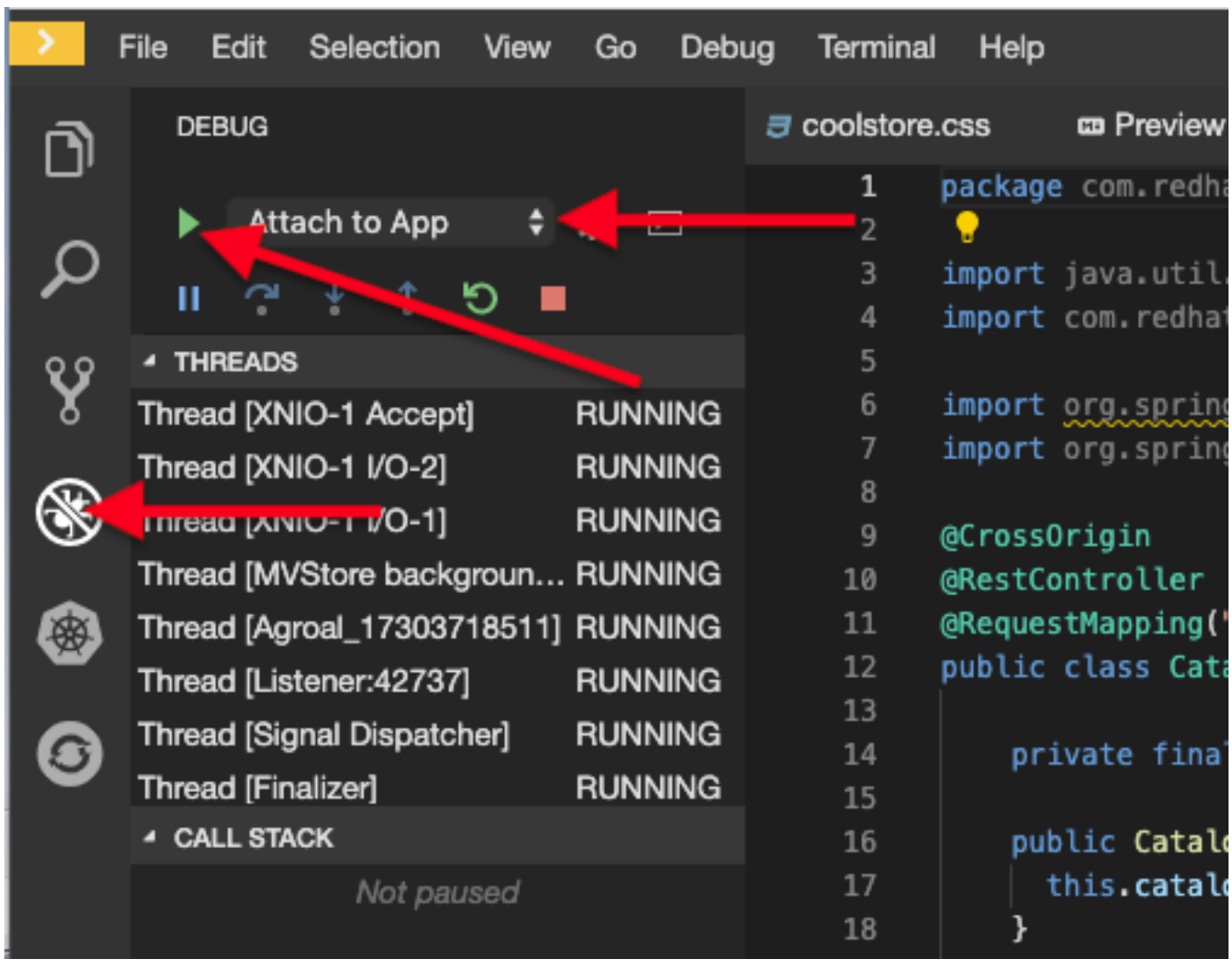
You don't see the letter `l` do you? You should see:



There's no `l`! You can probably spot the error, but let's walk through the debugger.

2. Debug with CodeReady Workspaces

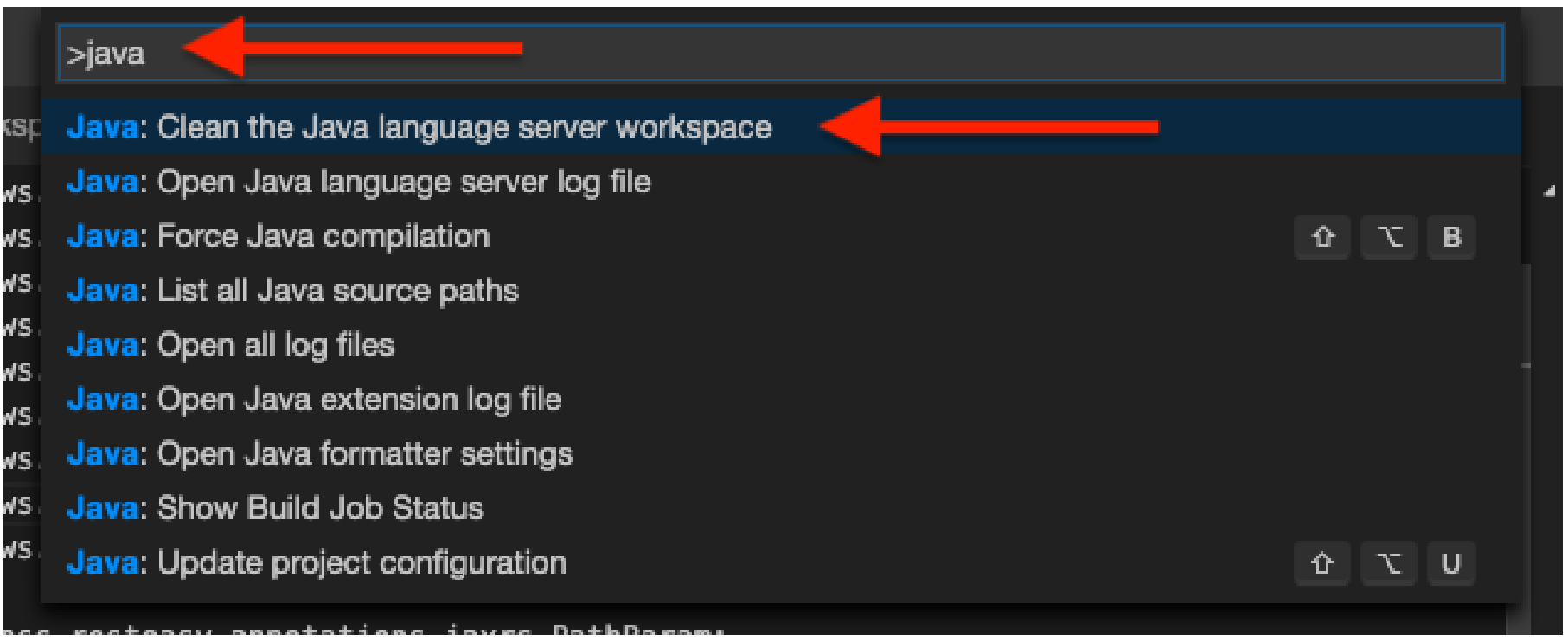
Click on the Debug icon on the left, select *Attach to App* in the drop-down, and click the green *Start Debugging* Icon:



You may see `[Warn] The debugger and the debuggee are running in different versions of JVMs. You could see wrong source mapping results.` You can ignore this warning, as long as the JVM versions are in the same major version family it won't be a problem.

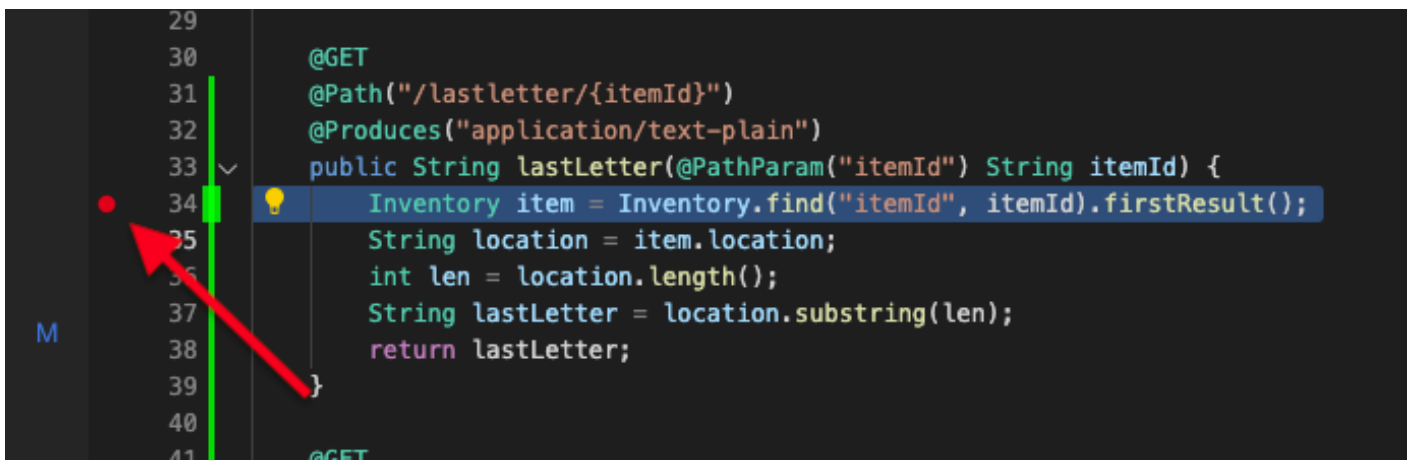
You should see a list of Threads in the debug console.

If you get errors or strange messages about Java Language Server or other failures, you may need to restart the Java Language Server. To do this, press `F1` to open the command window, or the more traditional `Control` + `SHIFT` + `P` (or `Command` + `SHIFT` + `P` on Mac OS X). You can also use the **View > Find Command...**. Type `java` and click on the command named **Java: Clean the Java language server workspace** as shown:



This will restart the Java language server. Once it's restarted and you're back in your workspace, click the *Attach to App* green icon once again to attach the debugger and proceed below.

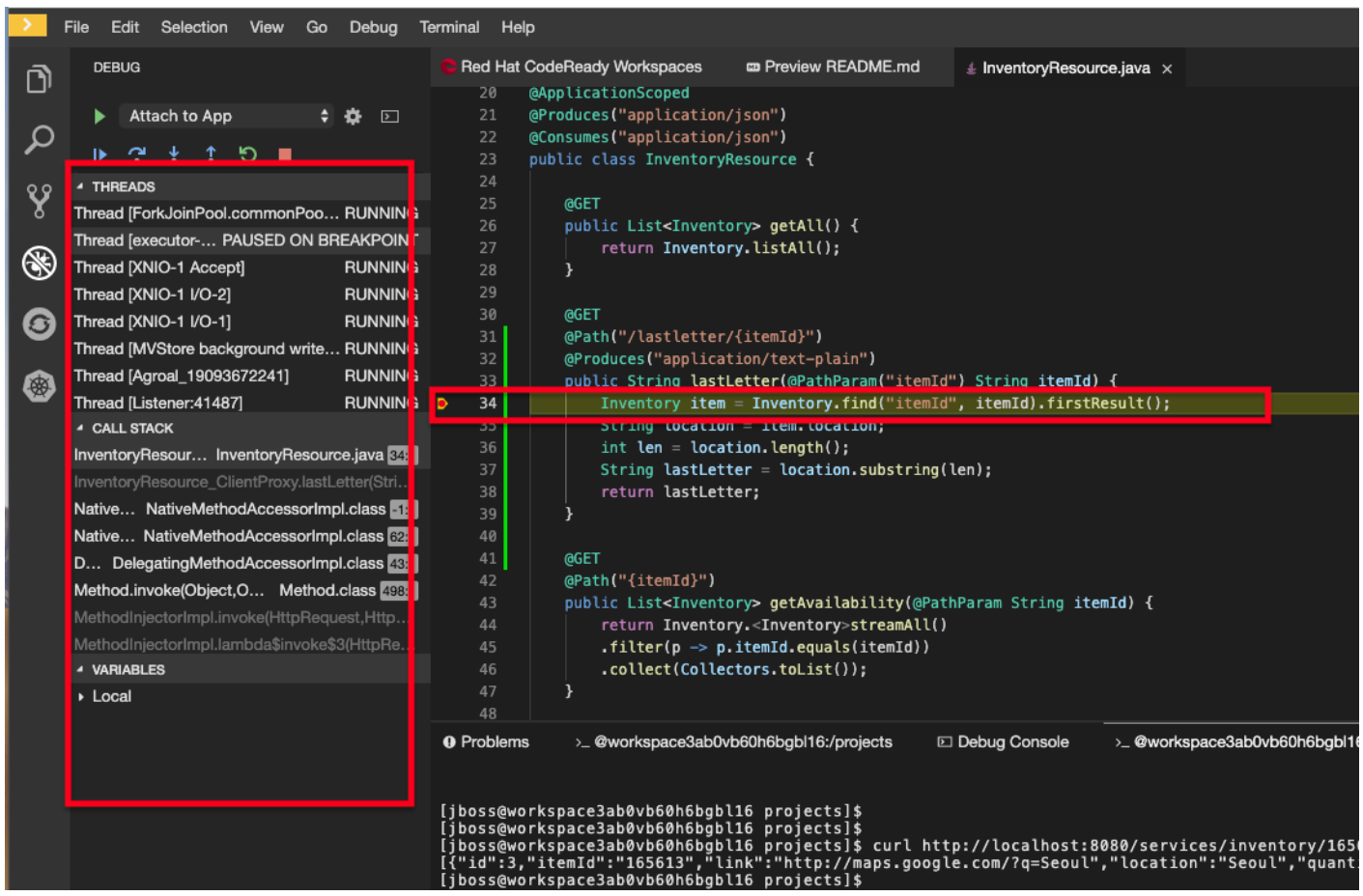
Go back to the *Explorer* view. In the new method you added, add a breakpoint by clicking to the left of the first line in that method to cause a red dot to appear, as shown:



Open a new Terminal and invoke the new method using the same `curl` command:

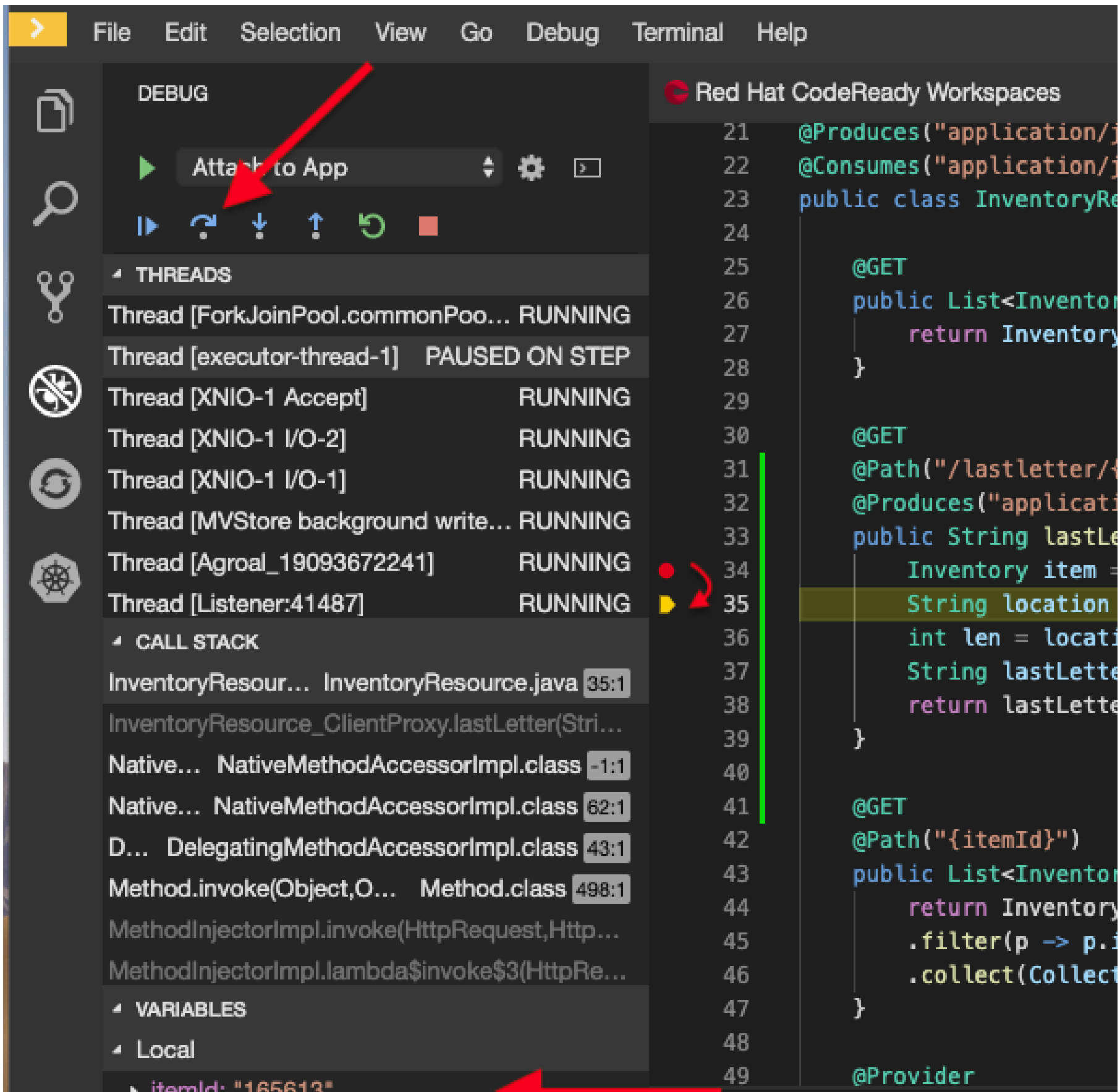
```sh
curl http://localhost:8080/services/inventory/lastletter/165613
```

This command will appear to hang, while the debugger intercepts the call. Switch back to the Debugger view to see the state:

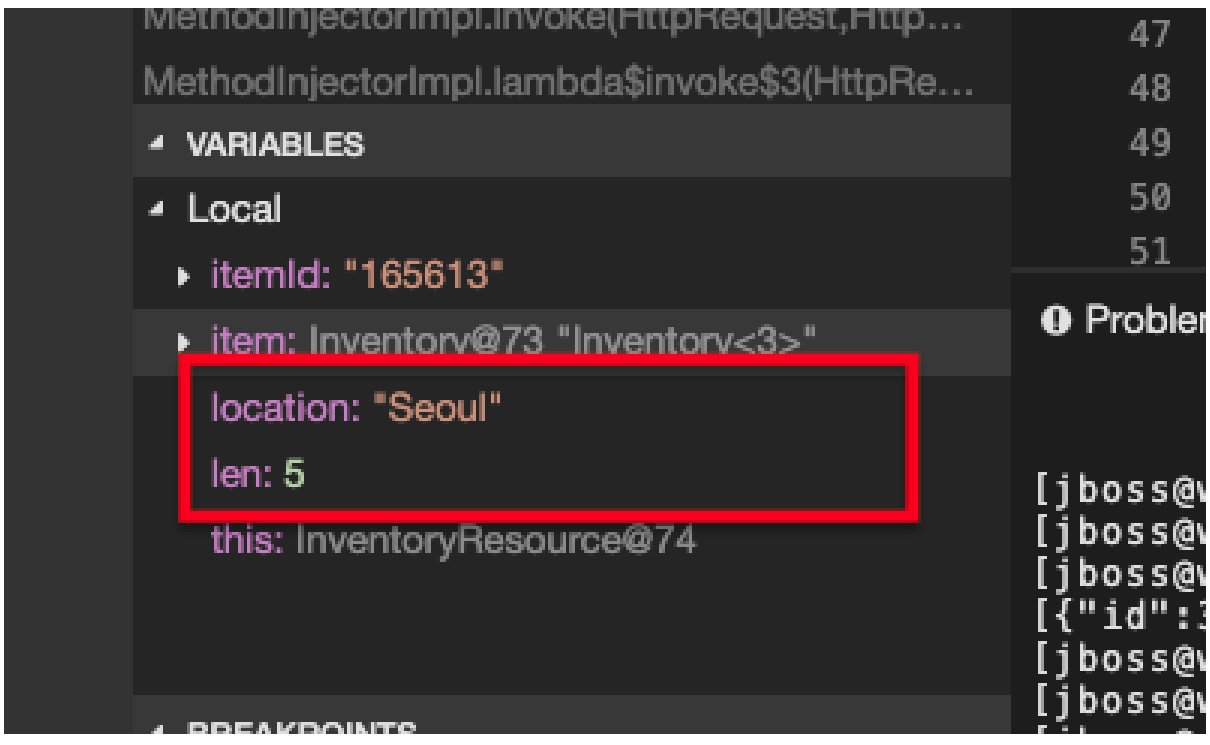


Click on the *Step Over* icon to execute one line and retrieve the inventory object for the given product id from the database and see the yellow cursor advance one line.

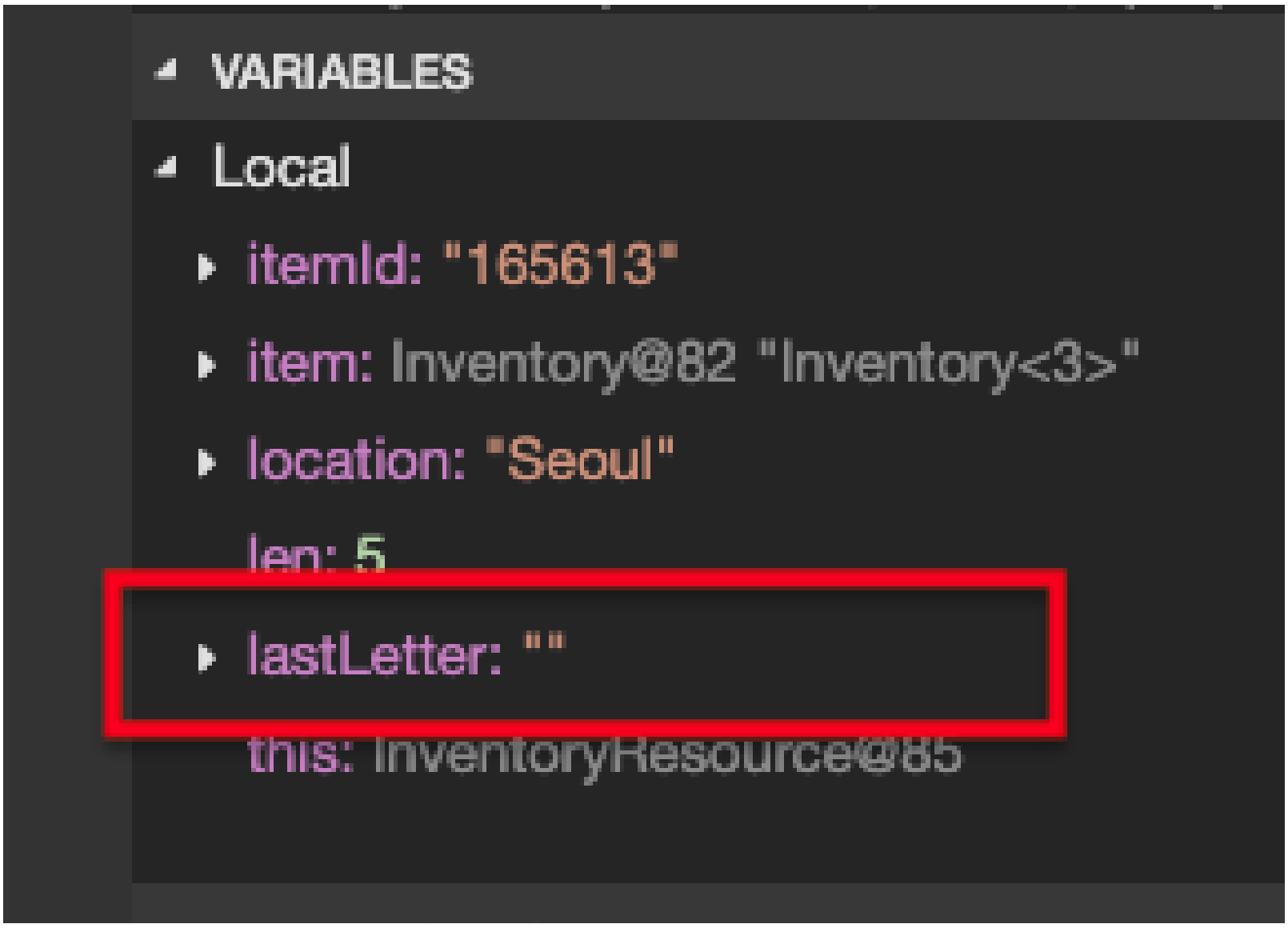
Expand the *Local Variables* in the lower left, to see the variables for `itemId` passed to the method and `item` element retrieved from the database. If you don't see them, click on the small arrow next to *Locals* to expand the list.



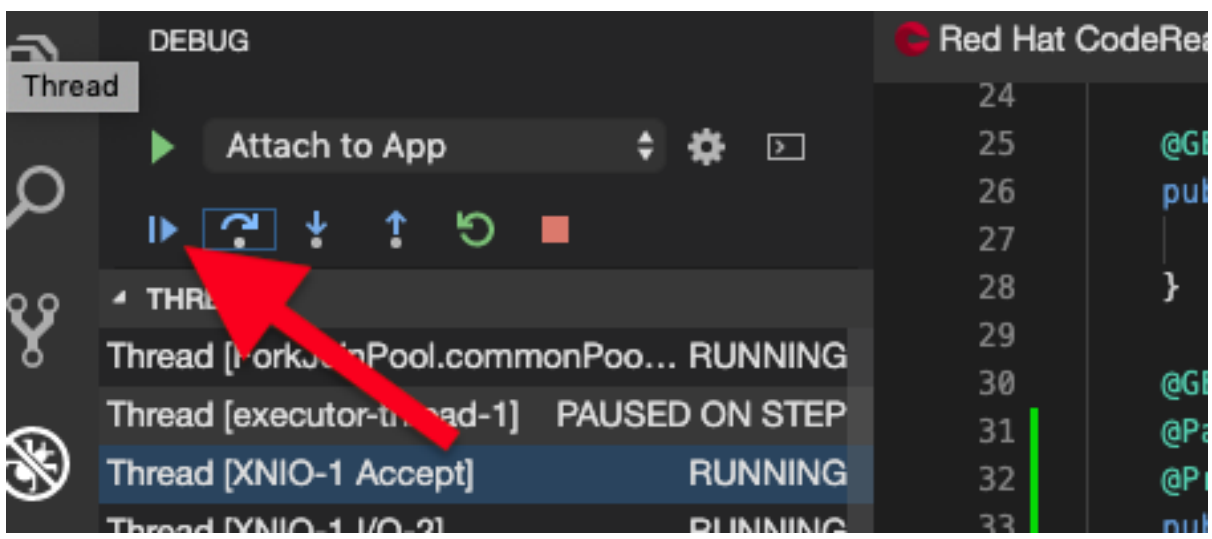
Step over 2 more times, and note the value of `location` is `Seoul` and `len` is set to the length of `Seoul` which is `5`.



One more step-over, and you can clearly see that `lastLetter` is blank (empty string). It's an off-by-one error!



Click the Continue button to let the method finish and return the empty value to `curl`:



3. Fix and Confirm

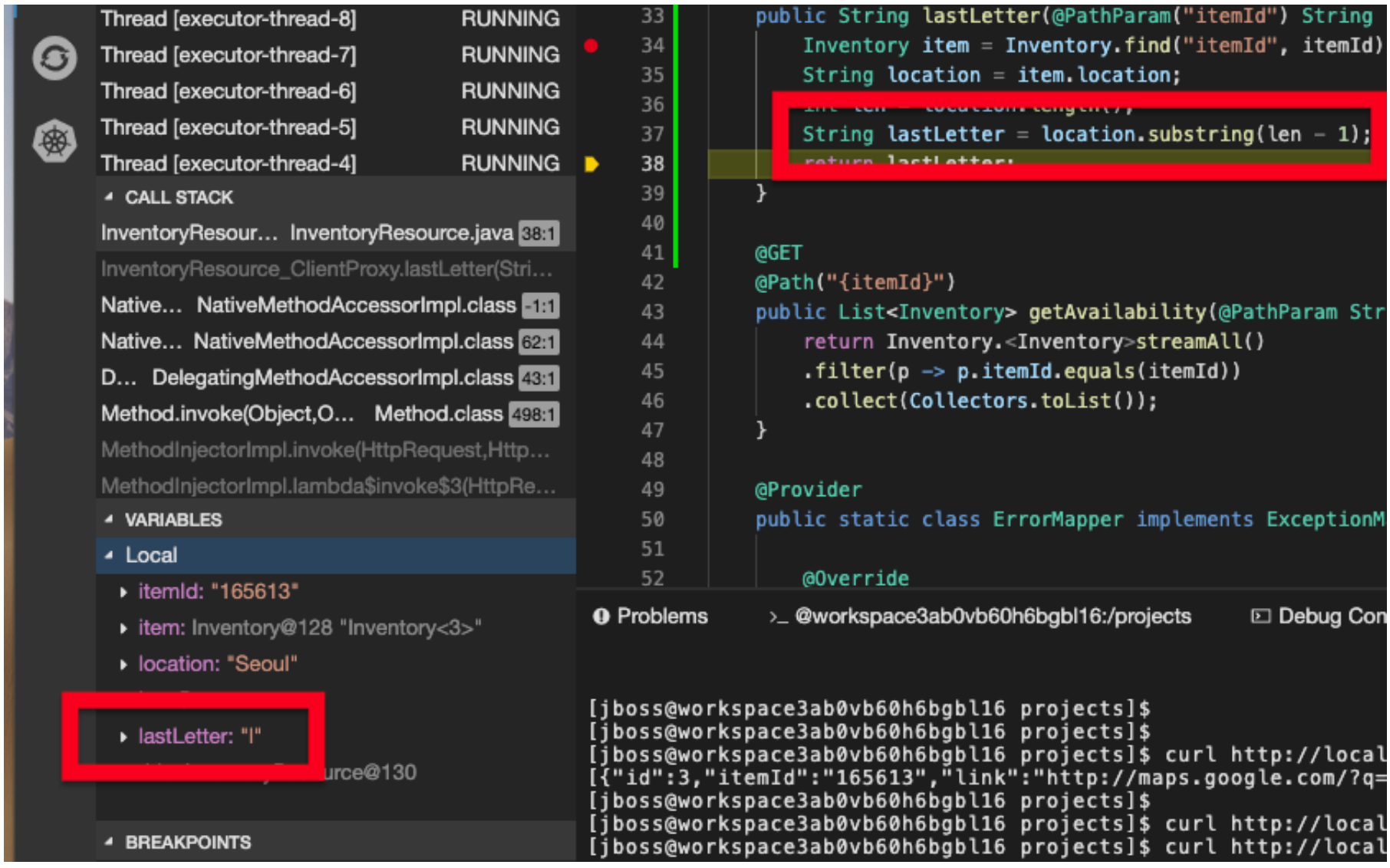
The off-by-one error can be fixed simply by fixing the call to `substring()`. Fix the bug by changing the line with `substring()` to read:

```java
        String lastLetter = location.substring(len - 1);
```

Execute the command again:

```sh
curl http://localhost:8080/services/inventory/lastletter/165613
```

The debugger will again catch the execution. Step through with the debugger as you did previously and confirm the value of `lastLetter` is `l` and is properly returned when the method is finished:



Click on the *Stop Debugger* (red box) to disconnect the debugger, then stop the app by typing `CTRL-C` in the Terminal in which the app runs.

Congratulations!

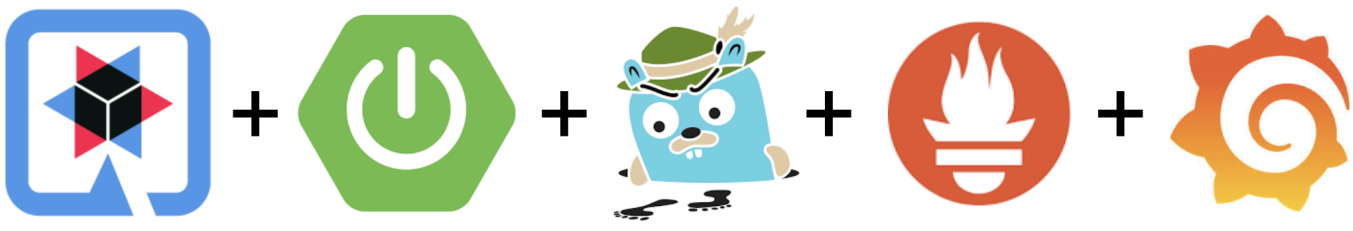
Quarkus apps are just like any other Java app, so debugging is straightforward and supported by many IDEs and CLIs out there. Combined with Live Reload and CodeReady Workspaces, it makes development quick and (relatively) painless!

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 (https://www.jaegertracing.io/) and Prometheus (https://prometheus.io/).



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:

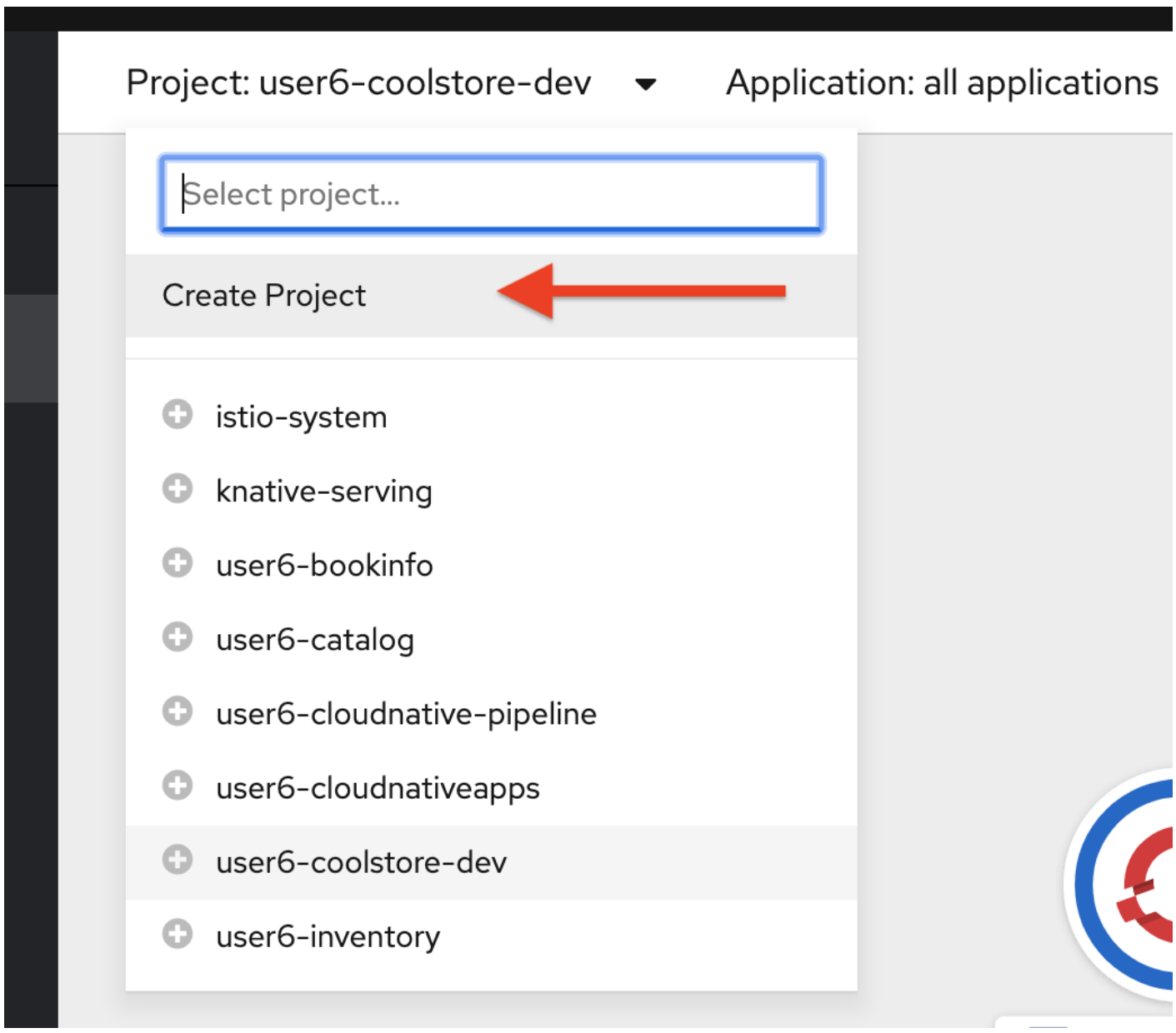
- Multi-dimensional time series data by metric name and key/value pairs

- 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

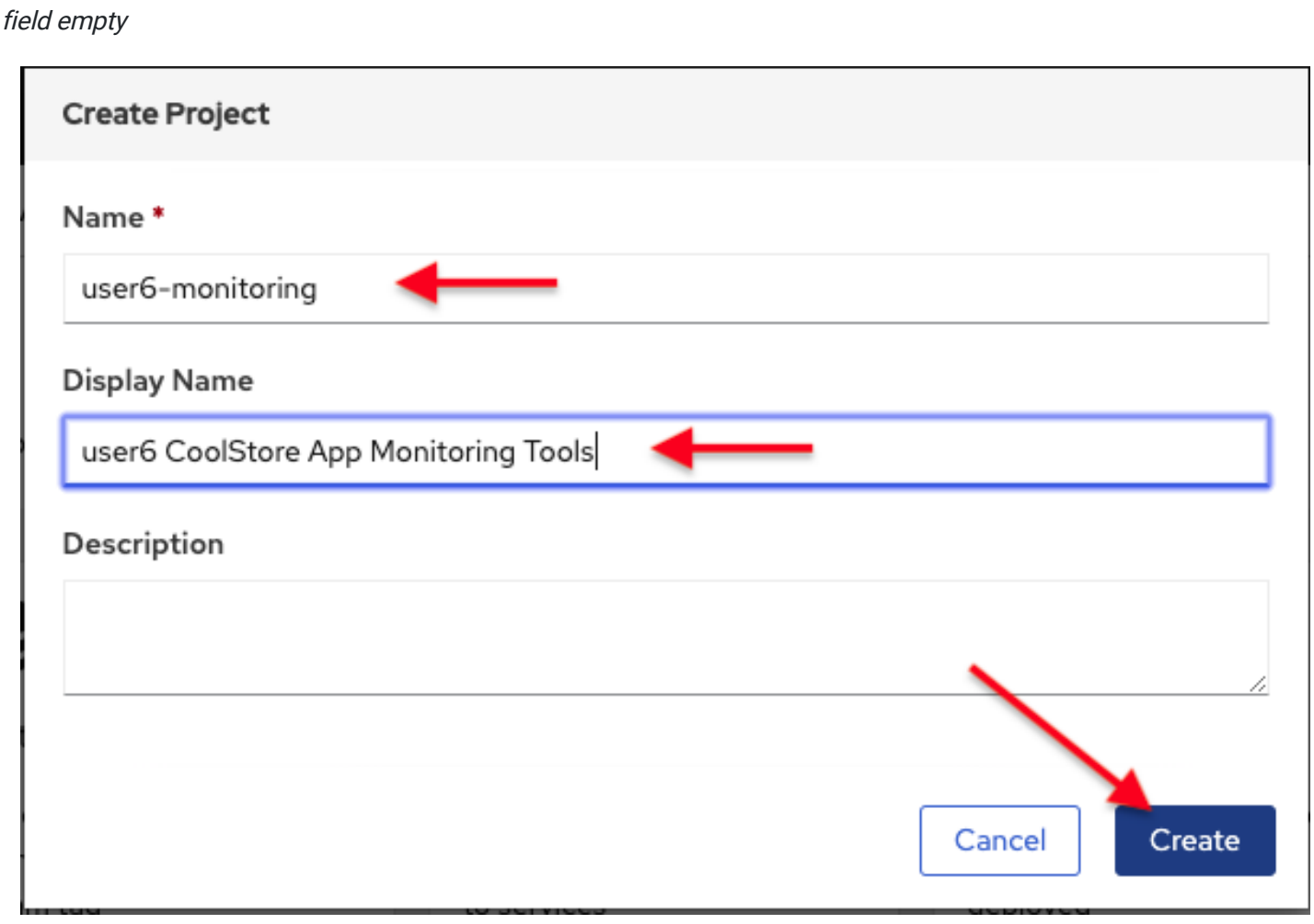
In this step, we will deploy our new monitoring tools for our CoolStore application, so create a separate project to house it and keep it separate from our monolith and our other microservices we already created previously.

Back in the Topology View (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-coolstore-dev), click on the Project drop-down and select *Create Project*.



Fill in the fields, and click **Create**:

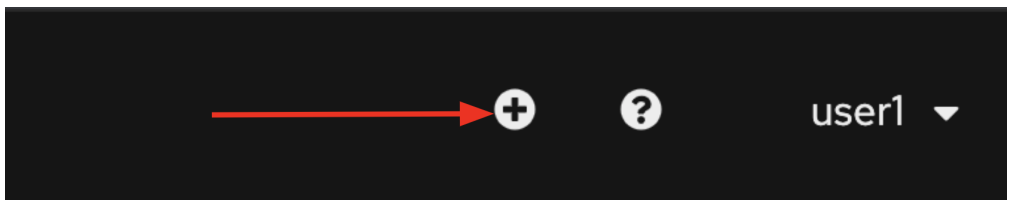
- Name: `user16-monitoring`
- Display Name: `user16 CoolStore App Monitoring Tools`
- Description: *leave this field empty*



2. Deploy Jaeger to OpenShift

In the new project, click `+Add`, select *From Catalog*.

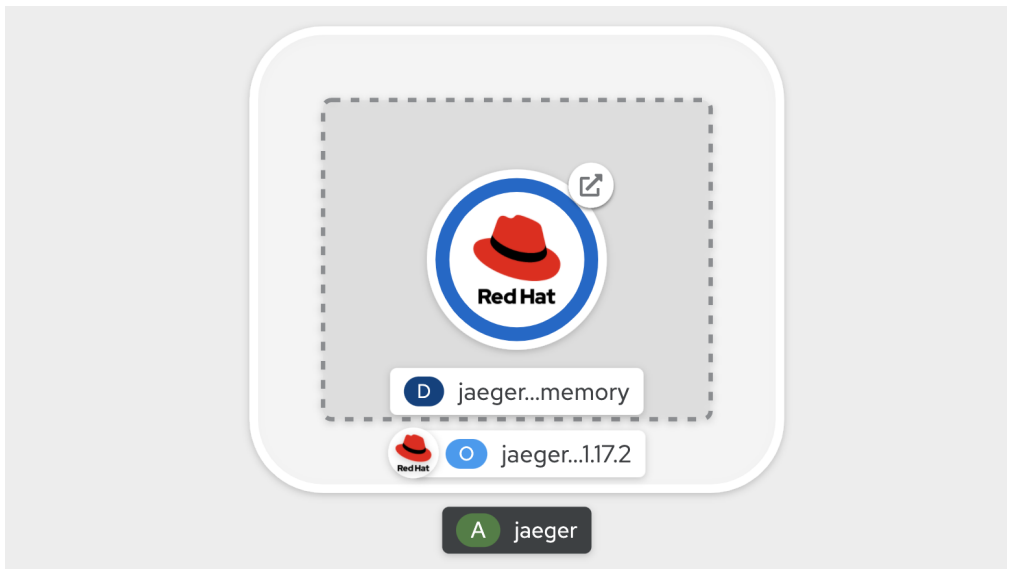
To deploy a jaeger server, click on `+` icon on the right top corner:



Copy the following *Service* in `YAML` editor then click on **Create**:

```yaml
apiVersion: jaegertracing.io/v1
kind: Jaeger
metadata:
  name: jaeger-all-in-one-inmemory
  namespace: user16-monitoring
```

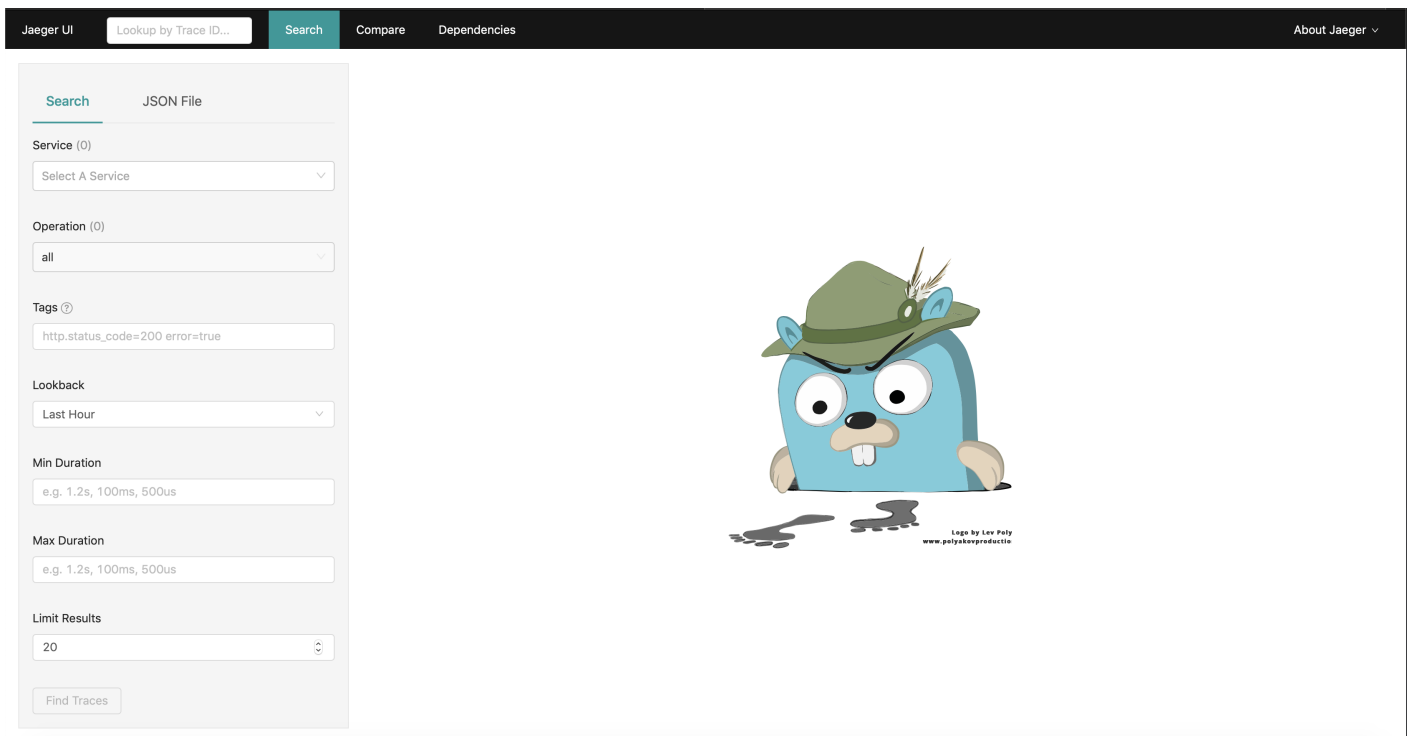
In the Topology View (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-monitoring) you can see Jaeger deploying:



4. Observe Jaeger UI

Once the deployment completes (dark blue circles!), open the Jaeger UI (https://jaeger-all-in-one-inmemory-user16-monitoring.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com).

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 the Quarkus extensions to the Inventory application for using **smallrye-opentracing**. Run the following commands to add the tracing extension via CodeReady Workspaces Terminal:

```sh
mvn quarkus:add-extension -Dextensions="smallrye-opentracing" -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m2-labs/inventory
```

You should see:

```
✅ Adding extension io.quarkus:quarkus-smallrye-opentracing
```

and `BUILD SUCCESS`. This ensures that the extension's dependencies are added to `pom.xml` for the inventory microservice.


There are many more extensions (https://quarkus.io/extensions/) for Quarkus for popular frameworks like Vert.x (https://vertx.io/), Apache Camel (http://camel.apache.org/), Infinispan (http://infinispan.org/), Spring DI compatibility (e.g. `@Autowired`), and more.

6. Create the configuration

Before getting started with this step, confirm the **jaeger-collector** service by visiting the Topology View (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-monitoring), and click on the *jaeger* deployment and select the *Resources* tab to view the services exposed by Jaeger:

We will configure our app to use the `http-c-binary-trft` service on port `14268` to report traces.

In the `inventory` project under `cloud-native-workshop-v2m2-labs` workspace, open `src/main/resources/application.properties` file and add the following configuration via CodeReady Workspaces Terminal:

```
# Jaeger configuration
%prod.quarkus.jaeger.service-name=inventory
%prod.quarkus.jaeger.sampler-type=const
%prod.quarkus.jaeger.sampler-param=1
%prod.quarkus.jaeger.endpoint=http://jaeger-all-in-one-inmemory-collector.user16-monitoring.svc.cluster.local:14268/api/traces
```
PROPERTIES

You can also specify the configuration using environment variables or JVM properties. See Jaeger Features (https://www.jaegertracing.io/docs/1.12/client-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.

There is no tracing specific code included in the application. By default, RESTful requests sent to the app will be traced without **any** code changes being required thanks to MicroProfile Tracing. It is also possible to enhance the tracing information and manually trace other methods and classes. For more information on this, please see the MicroProfile OpenTracing specification (https://github.com/eclipse/microprofile-opentracing/blob/master/spec/src/main/asciidoc/microprofile-opentracing.asciidoc).

7. Re-Deploy to OpenShift

Repackage and re-deploy the inventory application via the Terminal:

```
oc project user16-inventory && \
mvn package -DskipTests -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m2-labs/inventory
```
SH

In the console and on the Inventory Topology View (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-inventory), wait for the re-build and re-deployment to complete.

8. Observing Jaeger Tracing

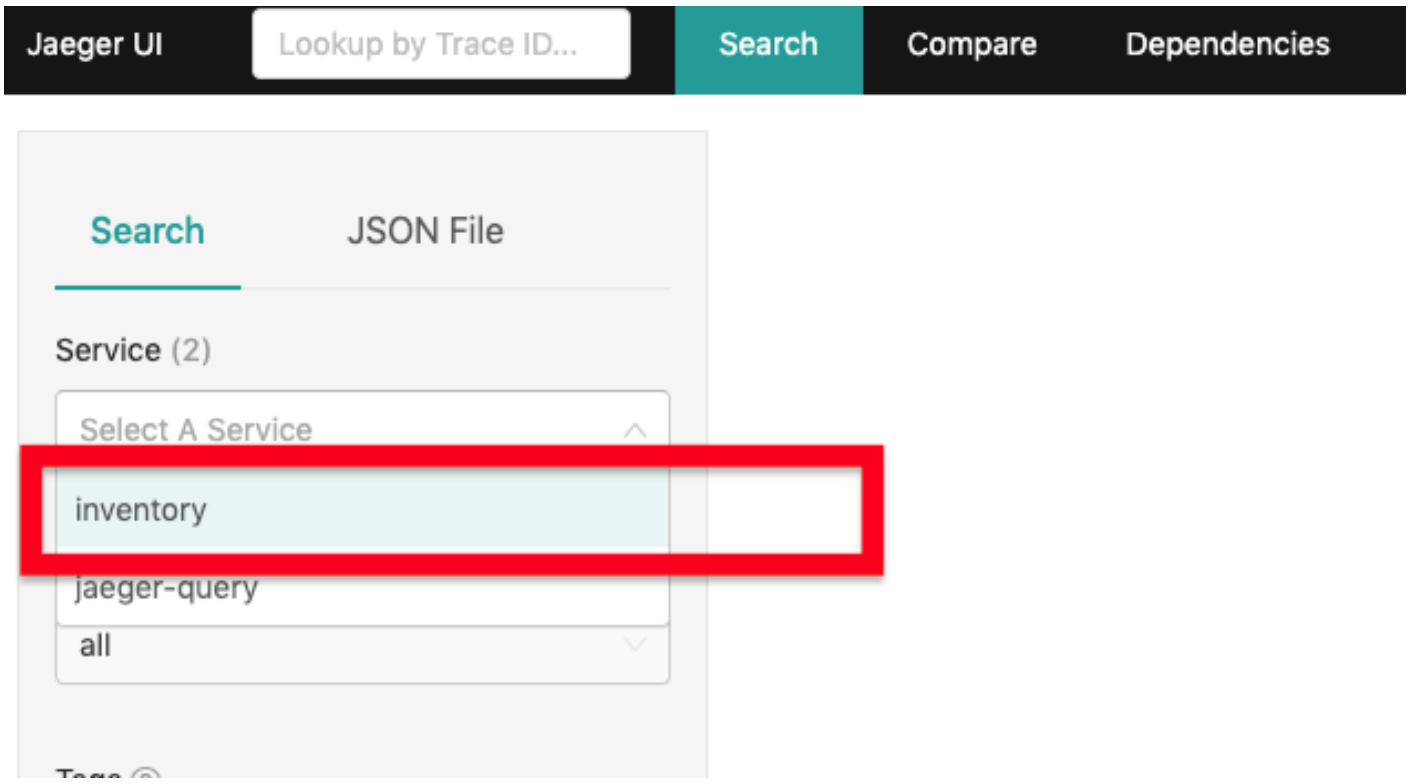
In order to trace networking and data transaction, we will call the Inventory service using **curl** commands via CodeReady Workspaces Terminal.

To generate traces, call the inventory 10 times:

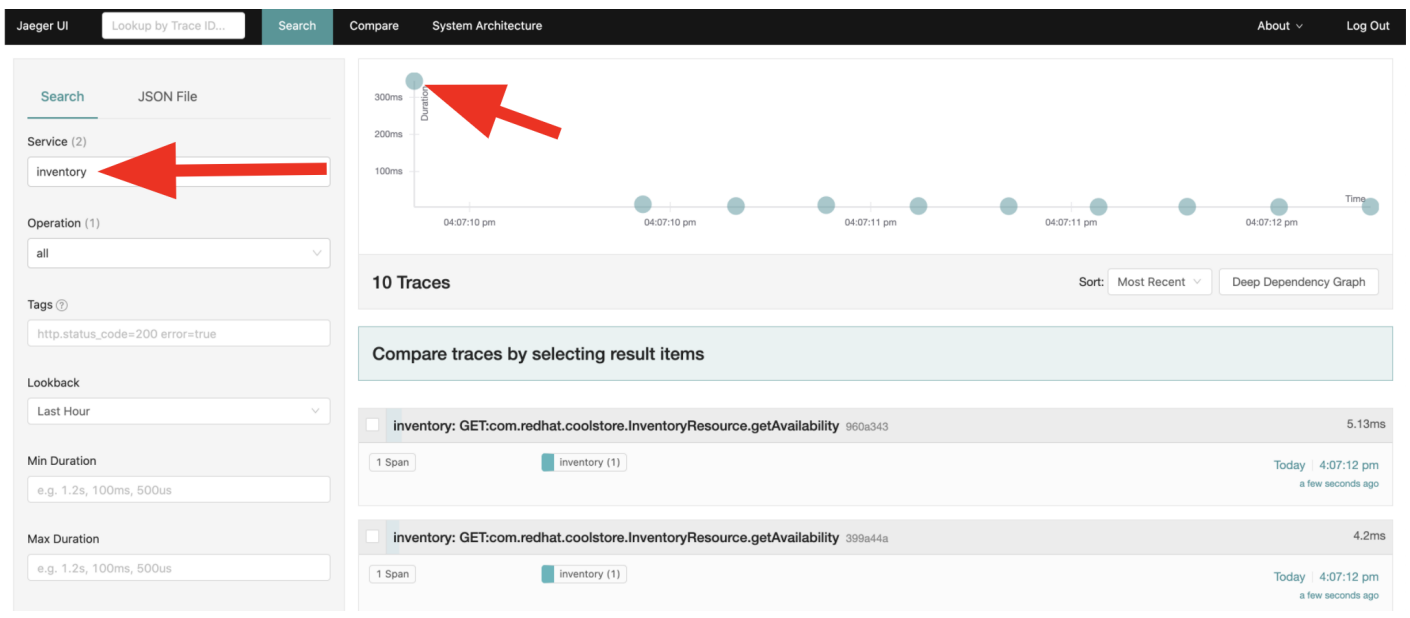
```sh
URL="http://$(oc get route -n user16-inventory inventory -o jsonpath="{.spec.host}")"

for i in $(seq 1 10) ; do
  curl -s $URL/services/inventory/165613
  sleep .2
done
```

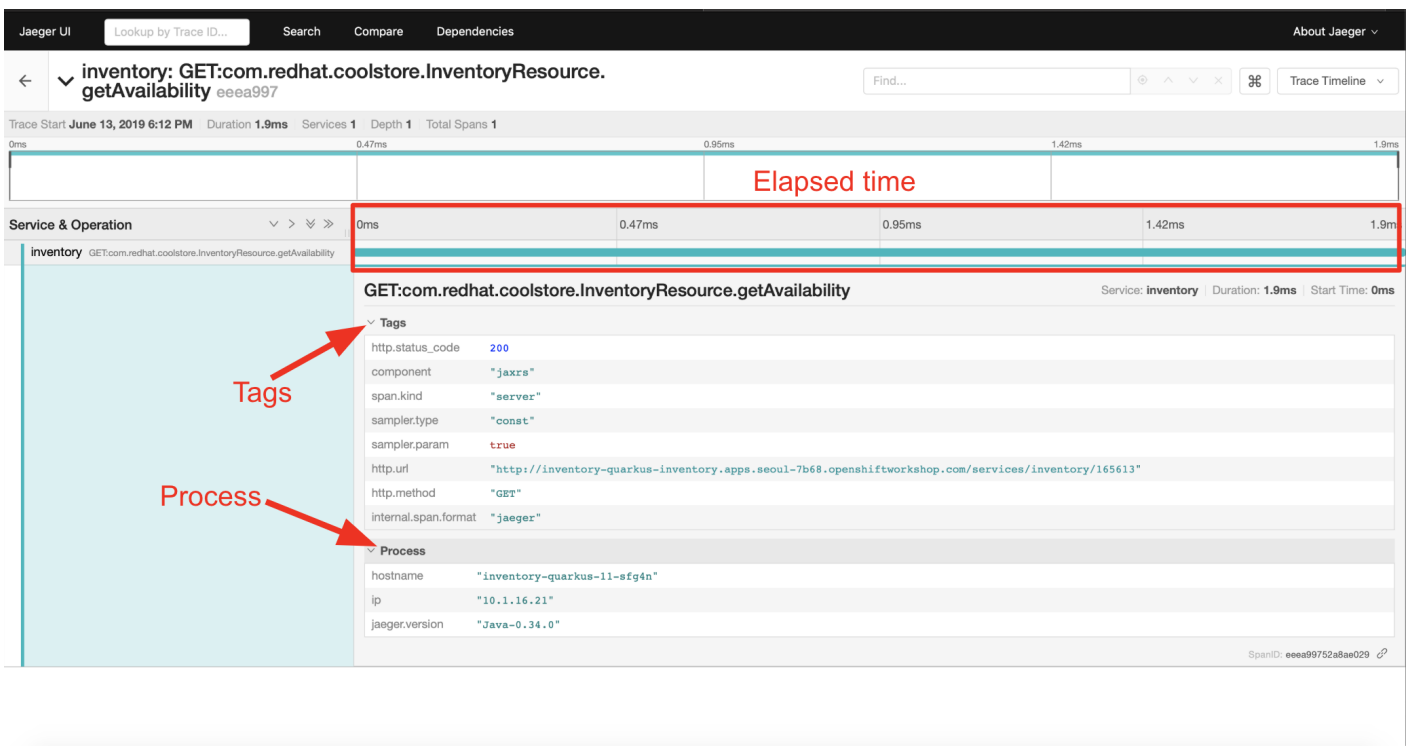
Now, reload the Jaeger UI (http://jaeger-query-user16-monitoring.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com) and you will find that a new `inventory` service appears alongside the service for Jaeger itself:



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



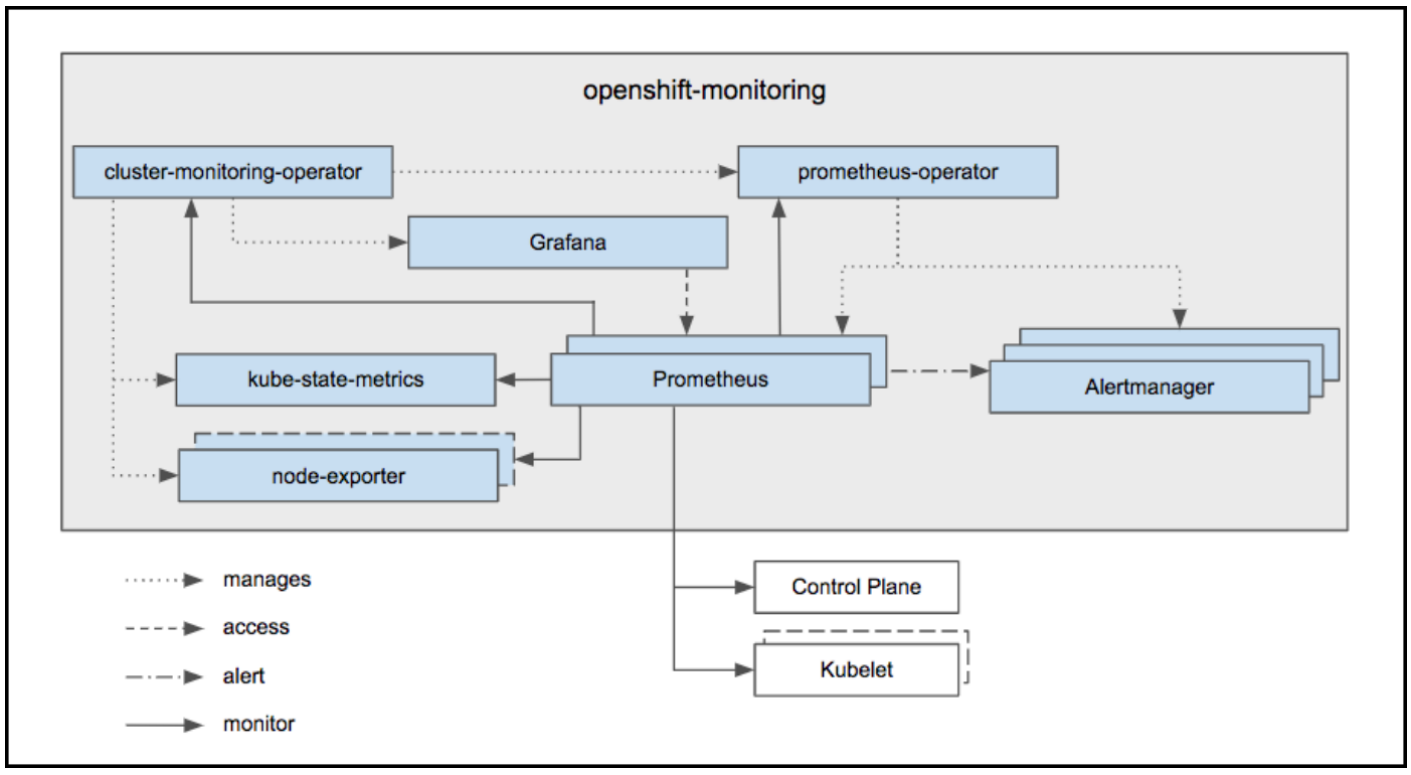
If you click on the single **Span** and you will see a logical unit of work in Jaeger that has an operation name, the start time of the operation, and the duration. Spans may be nested and ordered to model causal relationships:



As applications get more complex and many microservices are calling each other, these spans and traces will become more complex but also reveal issues with the app.

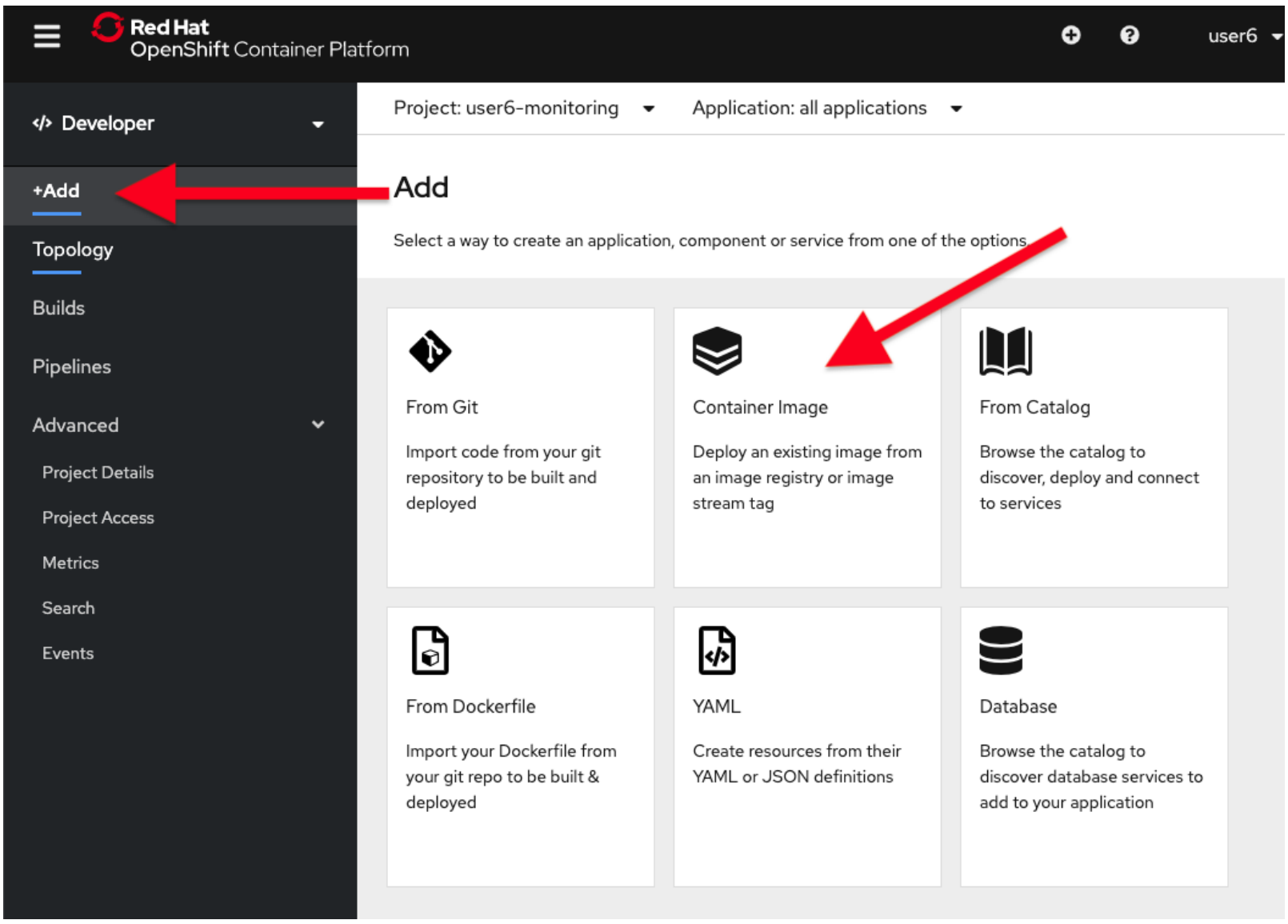
9. Deploy Prometheus and Grafana to OpenShift

OpenShift Container Platform ships with a pre-configured and self-updating monitoring stack that is based on the Prometheus (https://prometheus.io) 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 (https://grafana.com/) 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.

On the Monitoring Topology View (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-monitoring), click on `+Add`, and choose "Container Image"

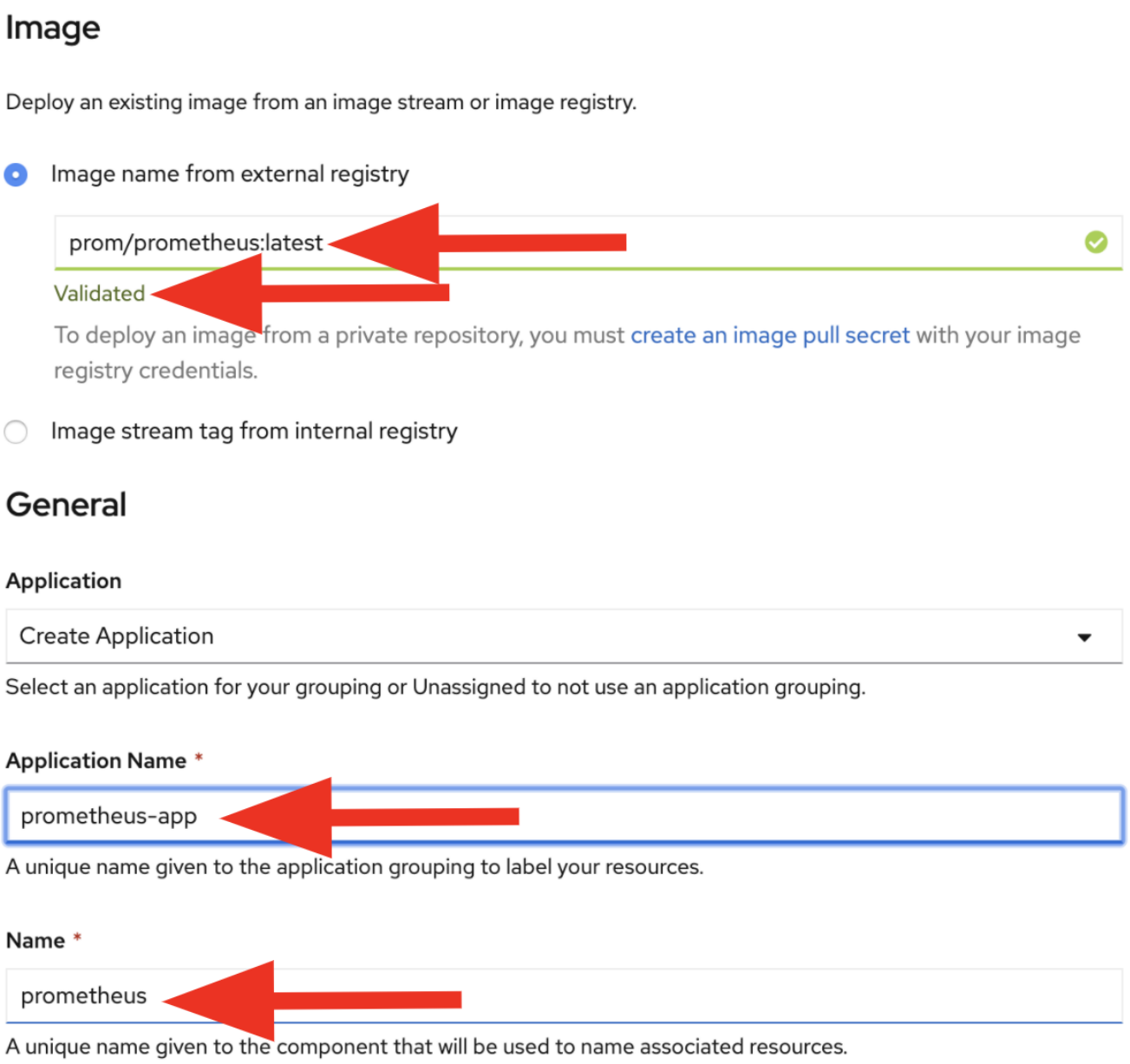


Fill out the following fields:

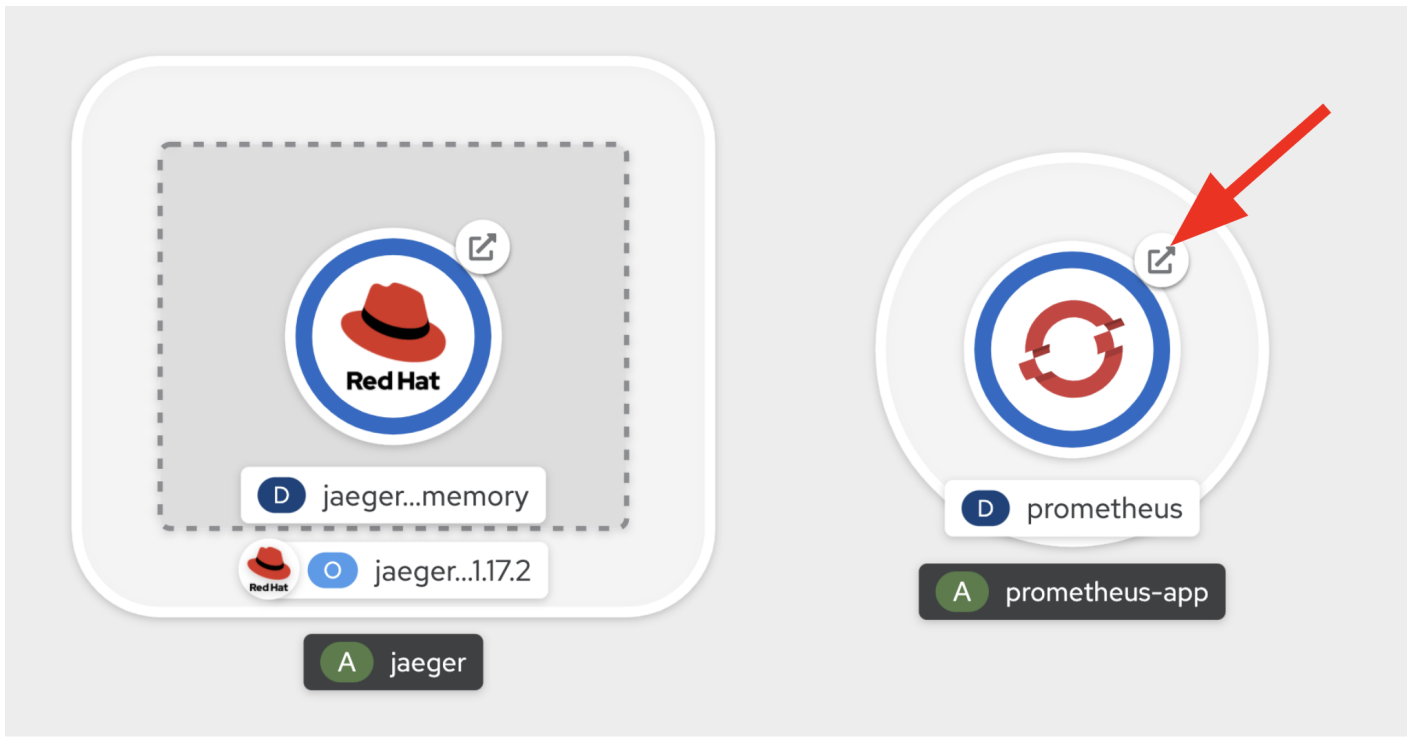
- **Image Name**: `prom/prometheus:latest`
- **Application Name**: `prometheus-app`
- **Name**: `prometheus`

Press **Enter** then you will see **green checked** icon and **Validated** in 30 seconds.

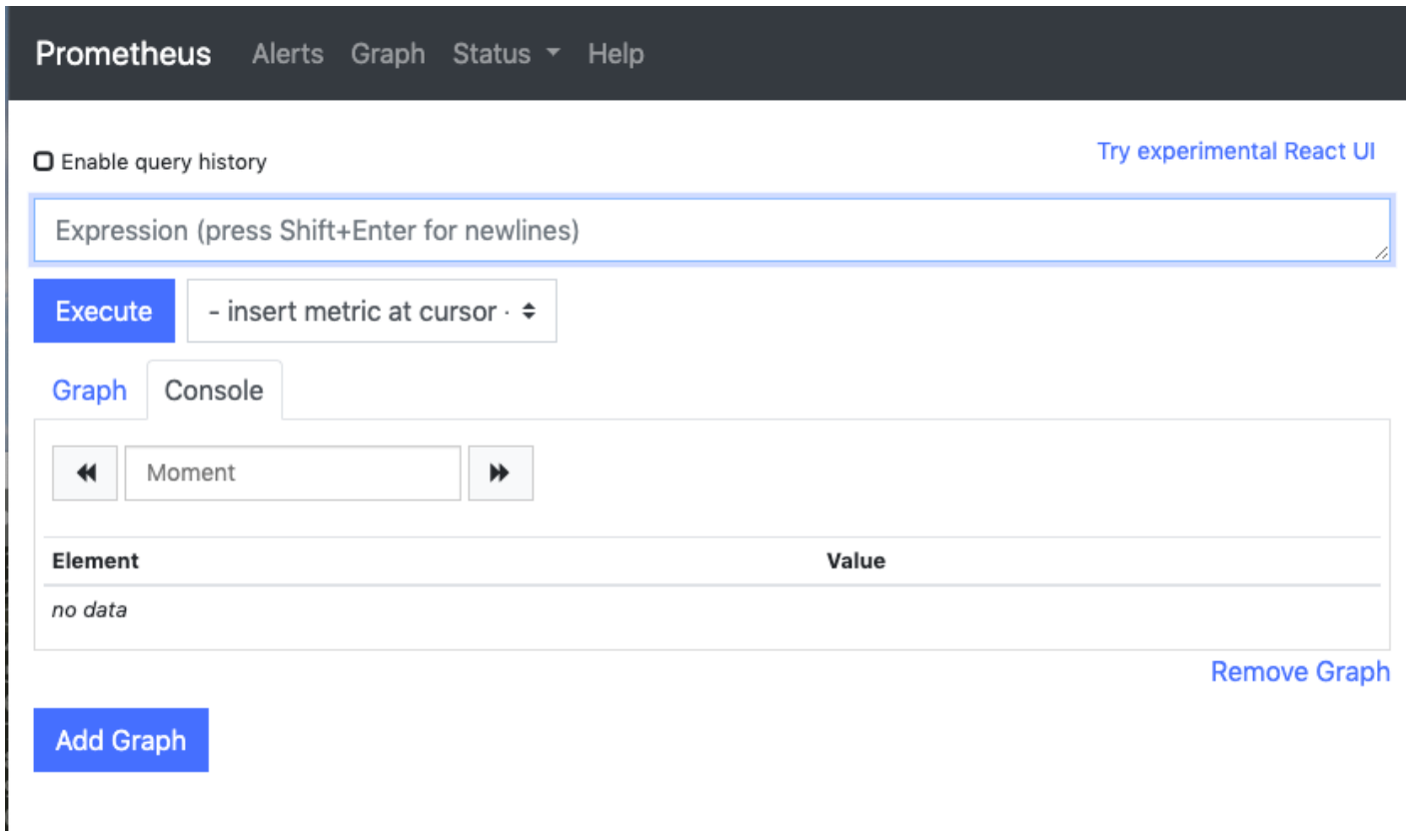
Leave the rest as-is and click **Create**:



On the Topology view, you'll see prometheus spinning up. Once it completes, click on the arrow to access the prometheus query UI:



Which should load the Prometheus Web UI (we'll use this later):



Deploy Grafana

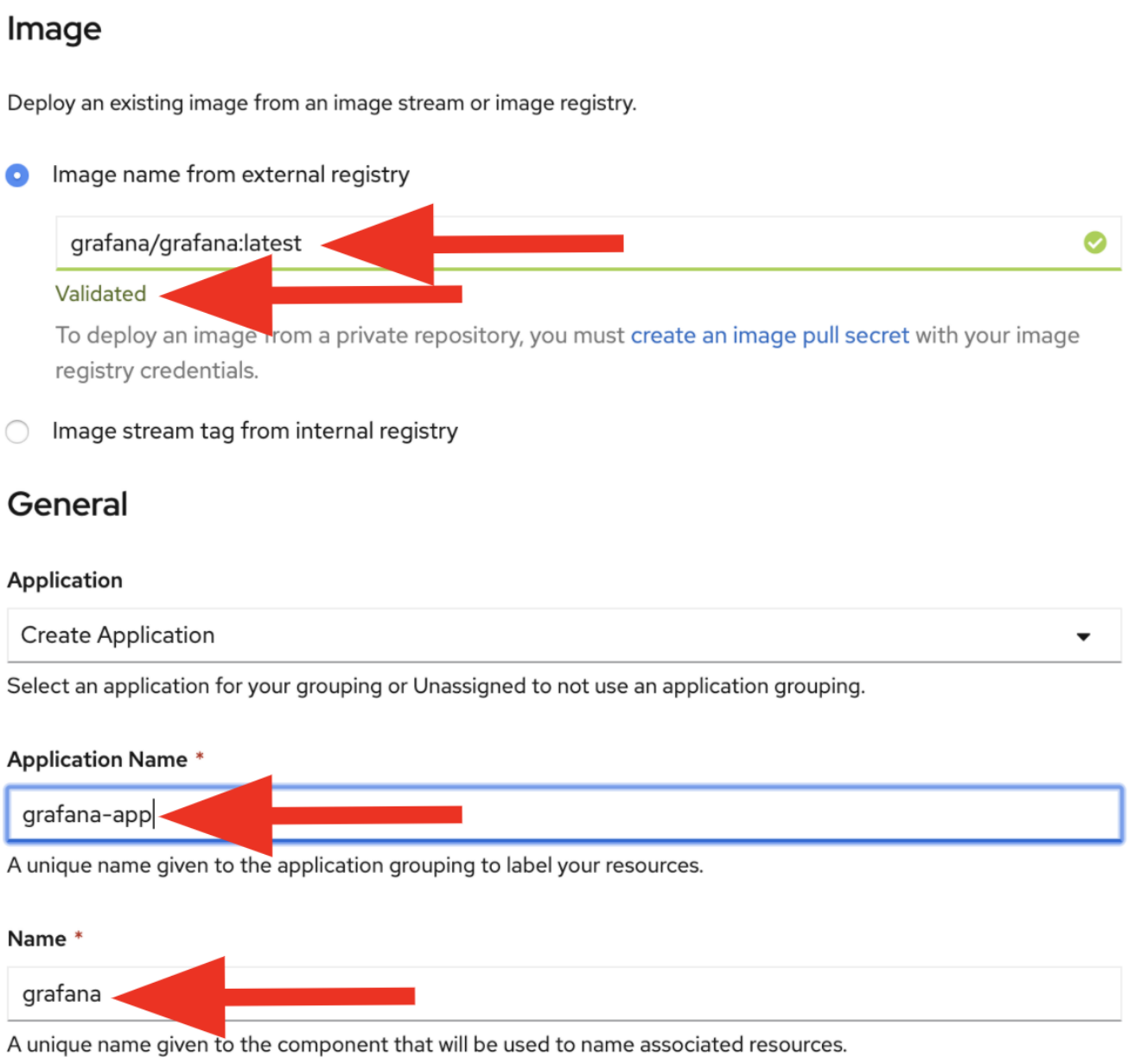
Follow the same process as before: On the Monitoring Topology View (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-monitoring), click on `+Add`, and choose "Container Image", and fill in the fields:

- **Image Name**: `grafana/grafana:latest`
- **Application**: `grafana-app`
- **Name**: `grafana`

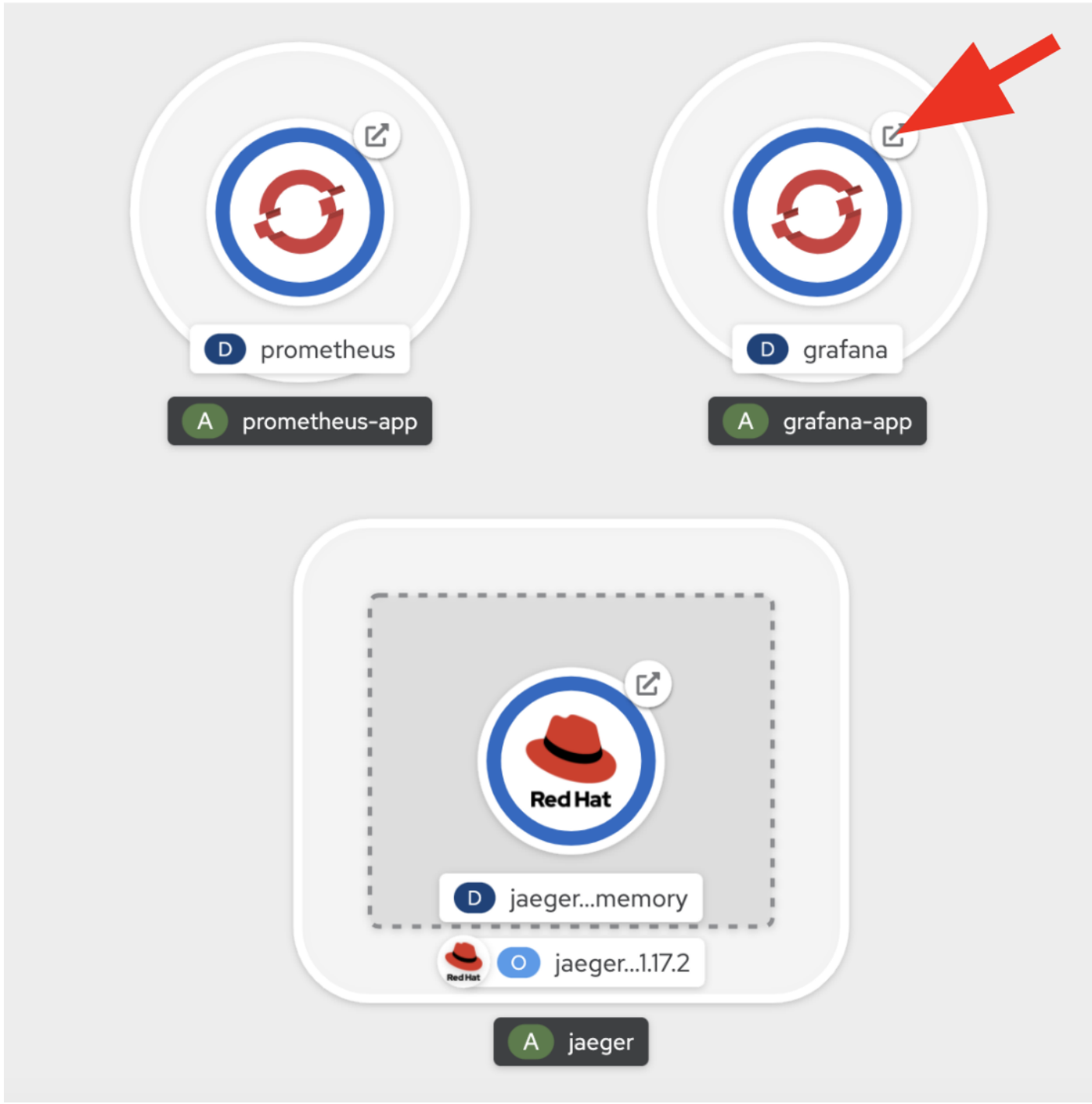
Click the "Magnifying Glass" search icon next to the image name to confirm the image exists.

Press **Enter** then you will see **green checked** icon and **Validated** in 30 seconds.

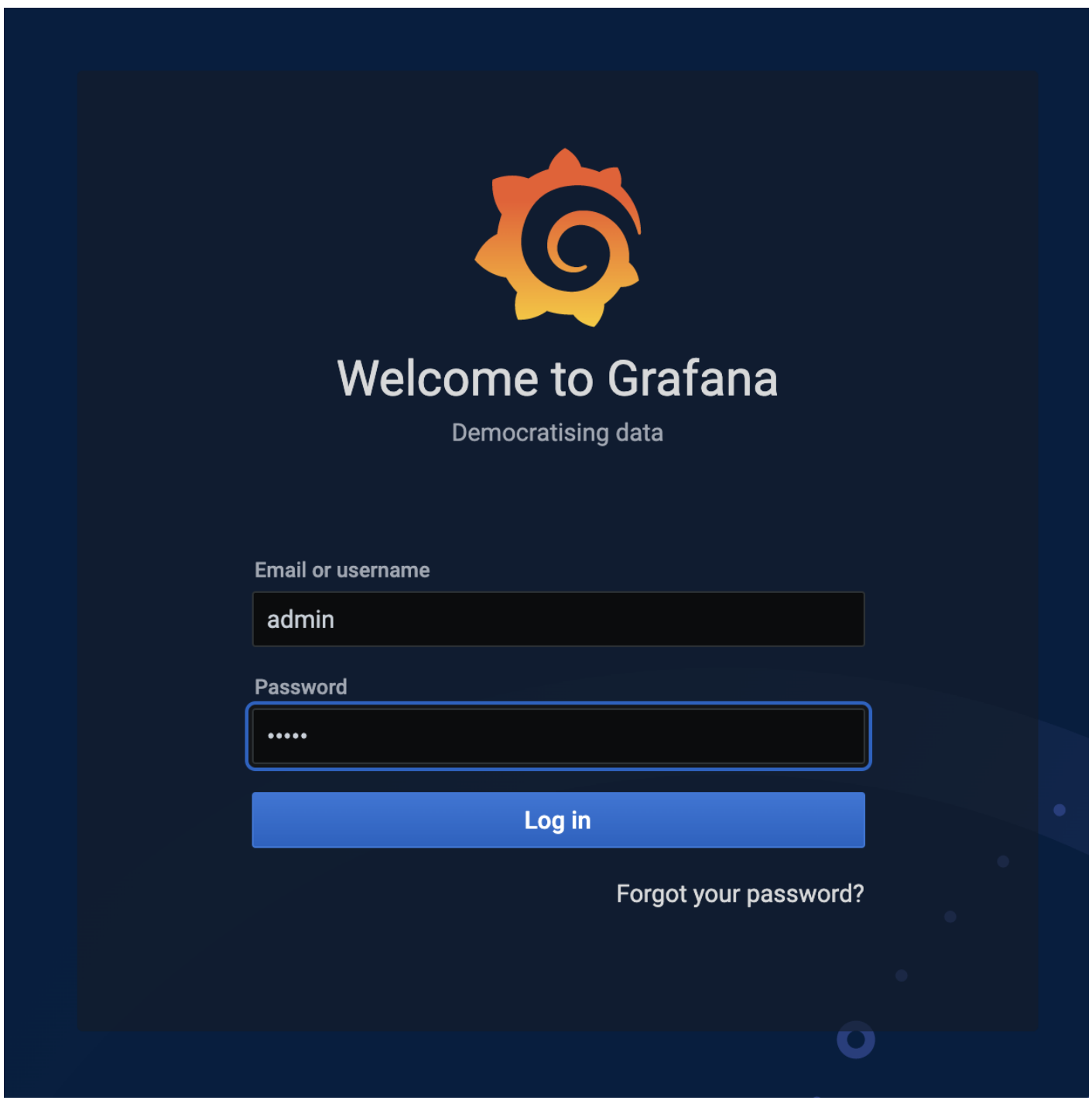
Leave the rest as-is and click **Create**:



On the Topology view, you'll see Grafana spinning up. Once it completes, click on the arrow to access the Grafana UI:



Which should load the Grafana Web UI:

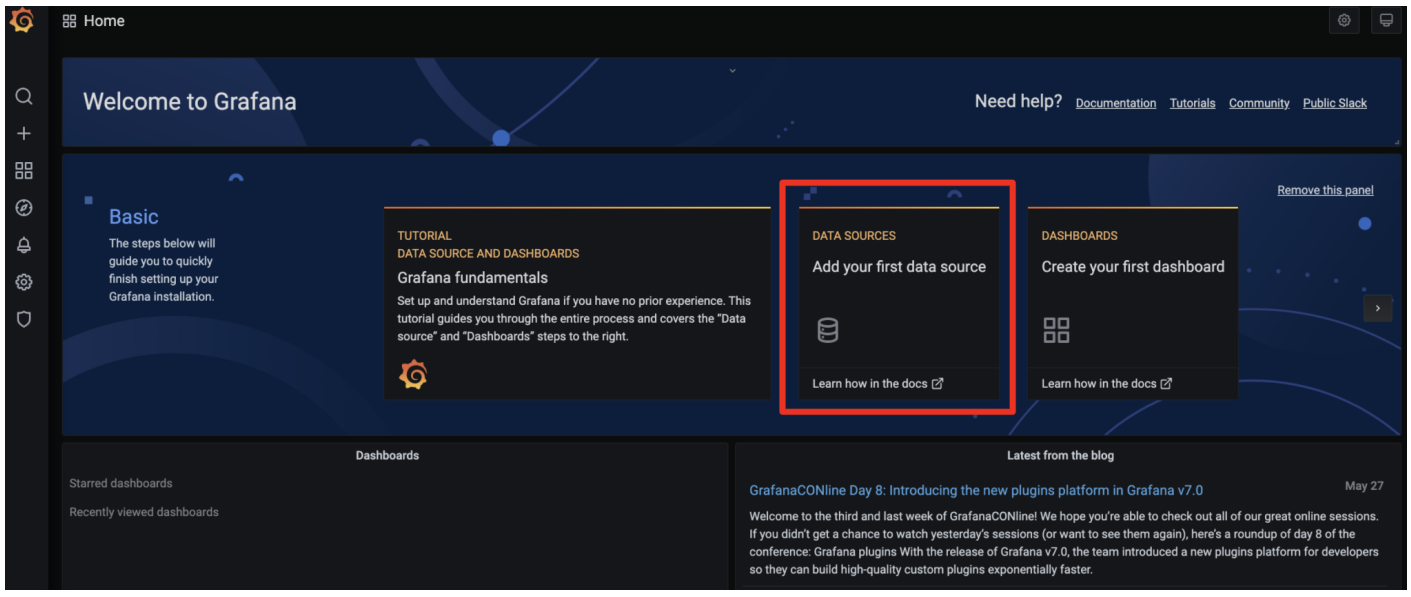


Log into Grafana web UI using the following values:

- Username: `admin`
- Password: `admin`

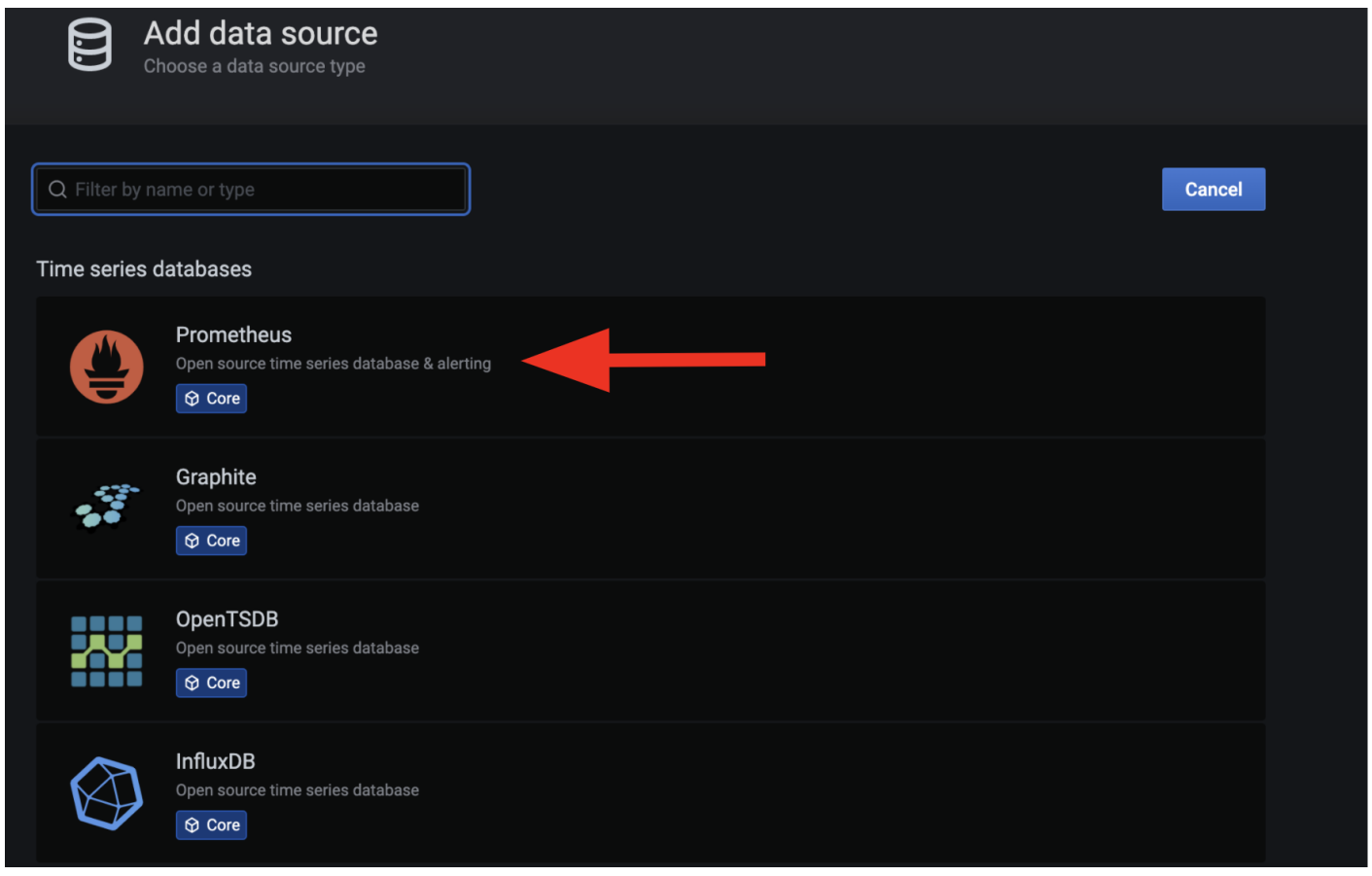
Skip the New Password (or change it to something else that you can remember)

You will see the landing page of Grafana as shown:



10. Add a data source to Grafana

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



Fill out the form with the following values:

- **URL**: `http://prometheus.user16-monitoring:9090`

Click on **Save & Test** and confirm you get a success message:

At this point Granana is set up to pull collected metrics from Prometheus as they are collected from the application(s) you are monitoring.

11. Utilize metrics specification for Inventory Microservice

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.

Add the necessary Quarkus extensions to the Inventory application for using *smallrye-metrics* using the following commands in a CodeReady terminal:

```sh
mvn quarkus:add-extension -Dextensions="metrics" -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m2-labs/inventory
```

You should see in the output:

```
✅ Adding extension io.quarkus:quarkus-smallrye-metrics
```
CONSOLE

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

In the *cloud-native-workshop-v2m2-labs/inventory* project, open `src/main/java/com/redhat/coolstore/InventoryResource.java`. Replace the two methods *getAll()* and *getAvailability()* with the below code which adds several annotations for custom metrics (**@Counted**, **@Timed**):

```java
    @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());
    }
```
JAVA

Add the necessary imports at the top:

```java
import org.eclipse.microprofile.metrics.MetricUnits;
import org.eclipse.microprofile.metrics.annotation.Counted;
import org.eclipse.microprofile.metrics.annotation.Timed;
```
JAVA

12. Redeploy to OpenShift

Repackage and redeploy the inventory application:

```sh
oc project user16-inventory && \
mvn clean package -DskipTests -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m2-labs/inventory
```
SH

You should get `BUILD SUCCESS` and then the application should be re-deployed. Watch the Inventory Topology View (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-inventory) until the app is re-deployed.

Once it's done you should be able to see the raw metrics exposed by the app with this command in a Terminal:

```sh
curl http://inventory-user16-inventory.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/metrics
```
SH

You will see a bunch of metrics in the OpenMetrics format:

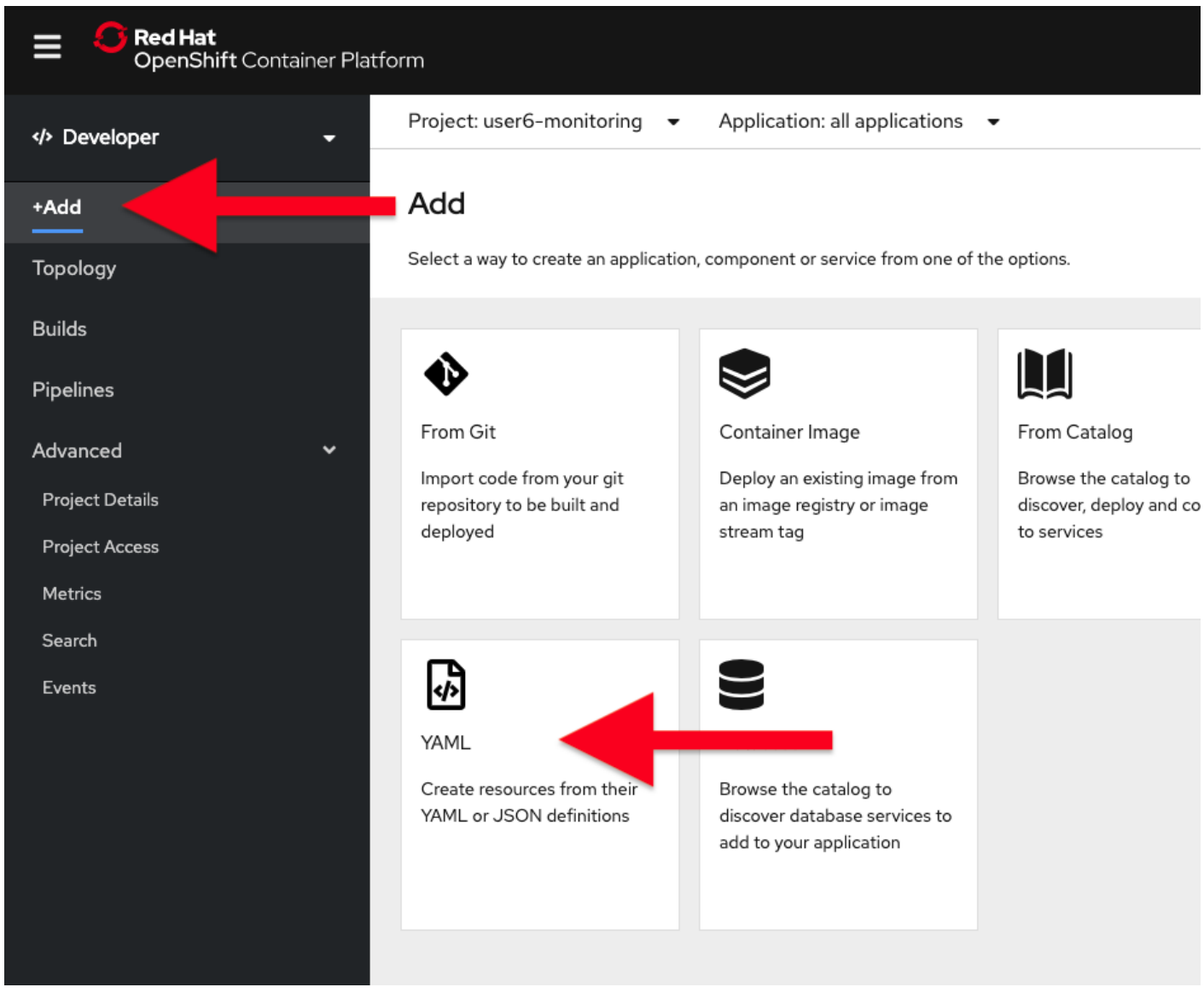
```
# HELP vendor_memoryPool_usage_bytes Current usage of the memory pool denoted by the 'name' tag
# TYPE vendor_memoryPool_usage_bytes gauge
vendor_memoryPool_usage_bytes{name="PS Survivor Space"} 916920.0
# HELP vendor_memoryPool_usage_bytes Current usage of the memory pool denoted by the 'name' tag
# TYPE vendor_memoryPool_usage_bytes gauge
vendor_memoryPool_usage_bytes{name="PS Old Gen"} 1.489556E7
# HELP vendor_memory_maxNonHeap_bytes Displays the maximum amount of used non-heap memory in bytes.
# TYPE vendor_memory_maxNonHeap_bytes gauge
vendor_memory_maxNonHeap_bytes 4.52984832E8
# HELP vendor_memory_usedNonHeap_bytes Displays the amount of used non-heap memory in bytes.
# TYPE vendor_memory_usedNonHeap_bytes gauge
vendor_memory_usedNonHeap_bytes 5.4685184E7
... and more
```
CONSOLE

This is what Prometheus will use to access and index the metrics from our app when we deploy it to the cluster. But first you must tell Prometheus about it!

Configure Prometheus ConfigMap

To instruct Prometheus to scrape metrics from our app, we need to create a Kubernetes *ConfigMap*.

On the Monitoring Topology View (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-monitoring), click `+Add` on the left, and this time choose **YAML**:



In the empty box, paste in the following YAML code:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
  namespace: user16-monitoring
data:
  prometheus.yml: >-
    scrape_configs:
      - job_name: 'prometheus'
        static_configs:
        - targets: ['localhost:9090']

      - job_name: 'inventory-quarkus'
        scrape_interval: 10s
        scrape_timeout: 5s
        static_configs:
        - targets: ['inventory.user16-inventory.svc.cluster.local:8080']
```

And click **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.

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:

```sh
oc set volume -n user16-monitoring deployment/prometheus --add -t configmap --configmap-name=prometheus-config -m /etc/prometheus/prometheus.yml --sub-path=prometheus.yml && \
oc rollout status -n user16-monitoring -w deployment/prometheus
```

This will trigger a new deployment of prometheus. Wait for it to finish!

13. Generate some values for the metrics

Now that Prometheus is scraping values from our app, let's write a loop to call our inventory service multiple times so we can observe the metrics. Do this with the following commands:

```sh
URL=http://$(oc get route -n user16-inventory inventory -o jsonpath="{.spec.host}")

for i in $(seq 1 600) ; do
  curl -s $URL/services/inventory/165613
  curl -s $URL/services/inventory
  sleep 1
done
```

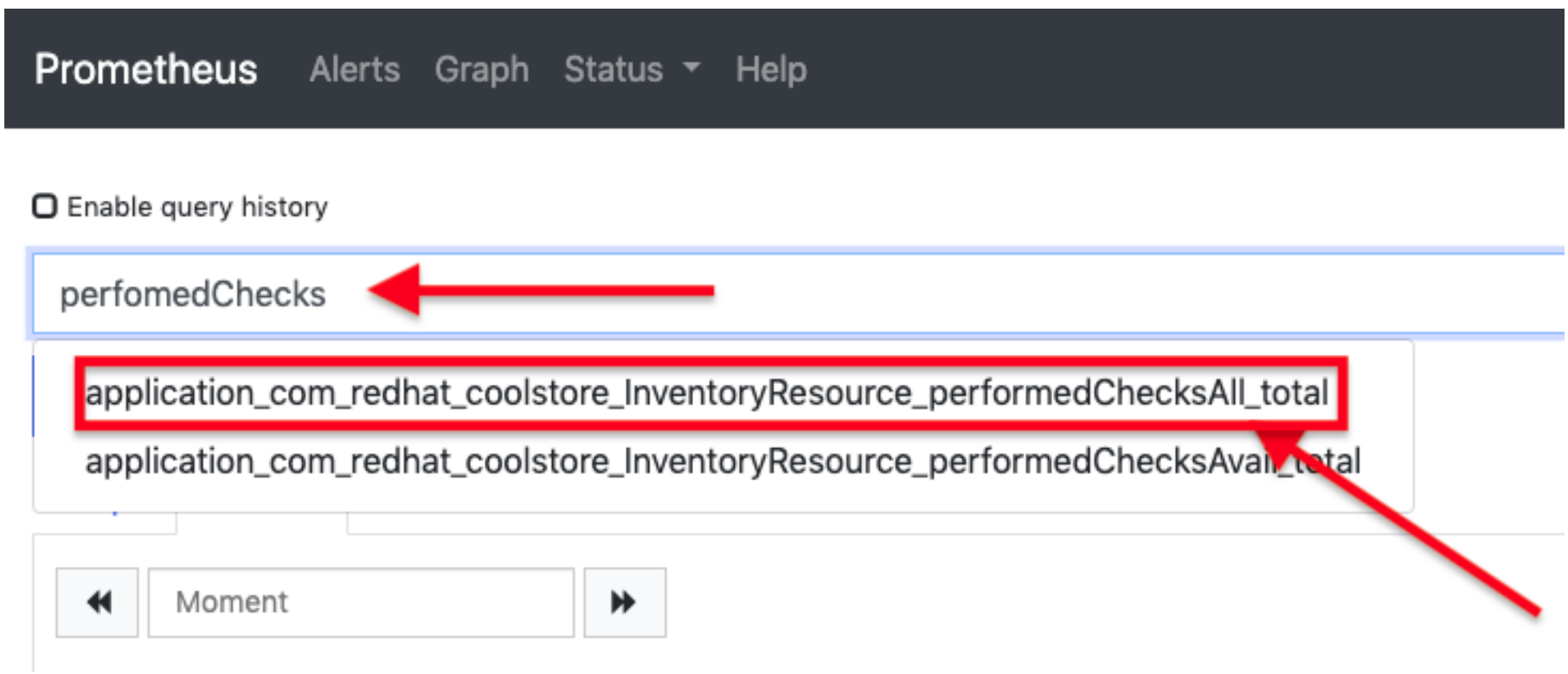
Leave this loop running (it will end after 600 seconds, or 10 minutes)

We have 3 ways to view the metrics:

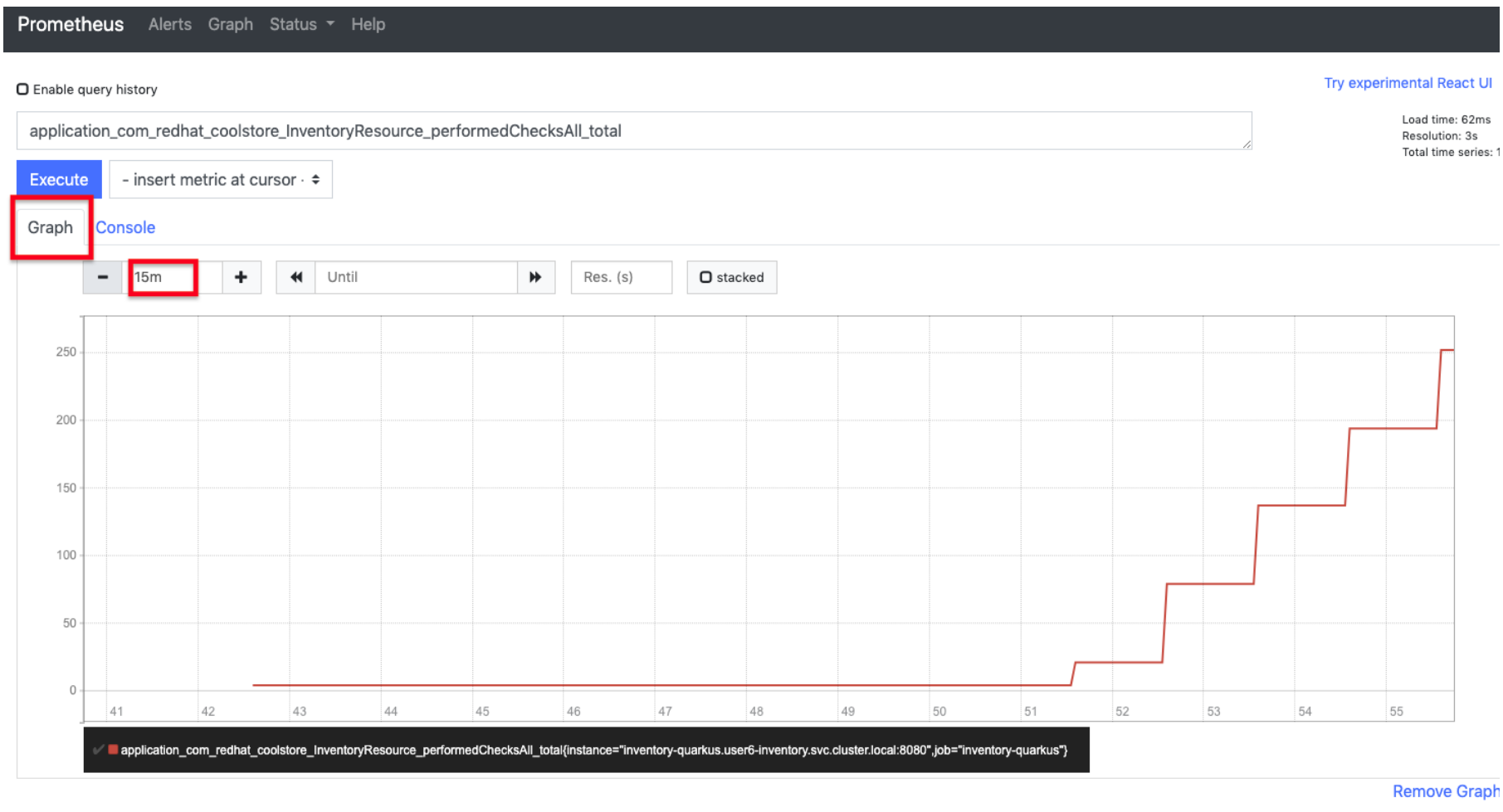
1. `curl` commands (which you already did)
2. Prometheus Queries
3. Grafana Dashboards

Using Prometheus

Open the Prometheus UI (http://prometheus-user16-monitoring.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com) and input `performedChecks` and select the auto-completed value:



Switch to **Graph** tab:

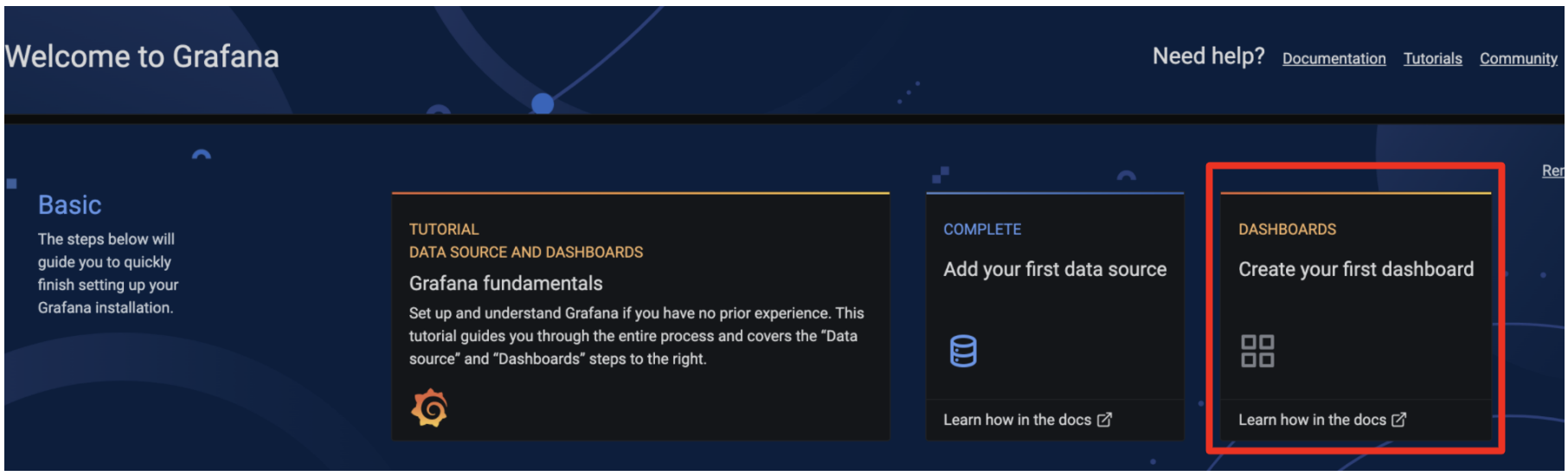


You can play with the values for time and see different data across different time ranges for this metric. There are many other metrics you can query for, and perform quite complex queries using Prom QL (https://prometheus.io/docs/prometheus/latest/querying/basics/) (Prometheus Query Language).

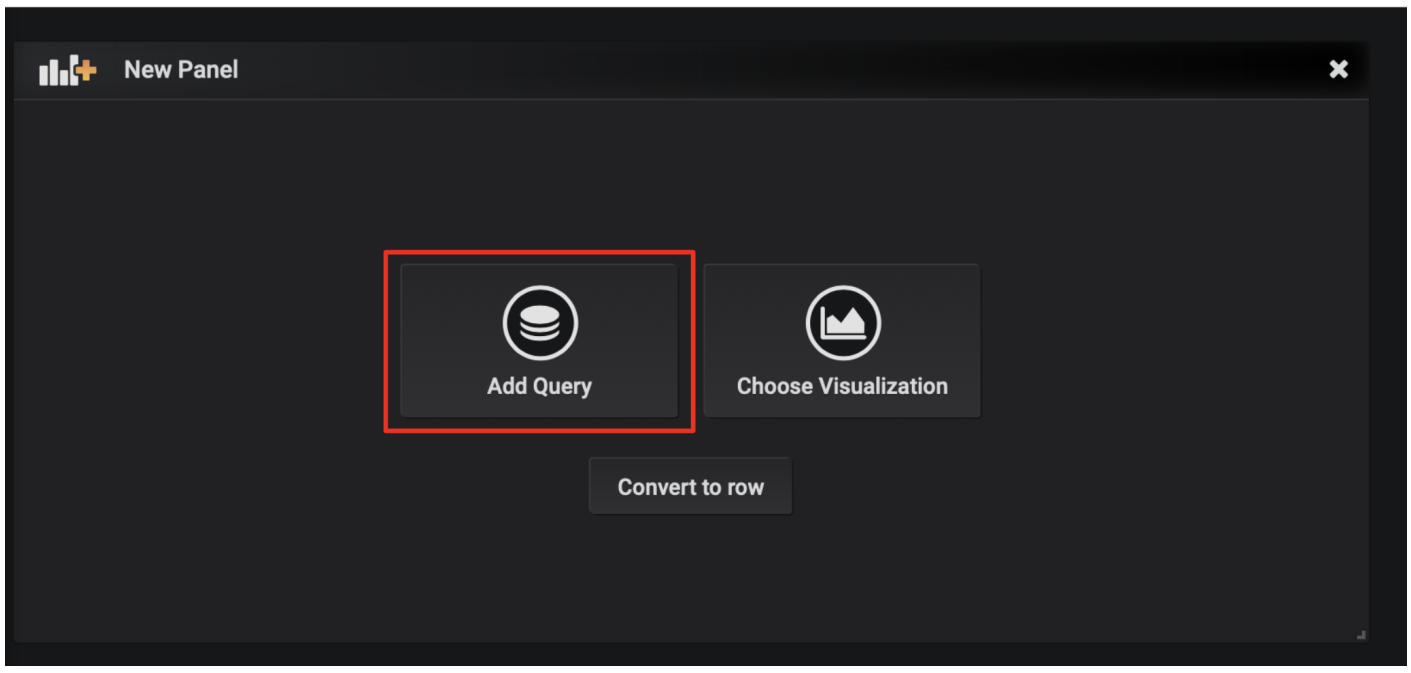
Using Grafana

Open the Grafana UI (http://grafana-user16-monitoring.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com).

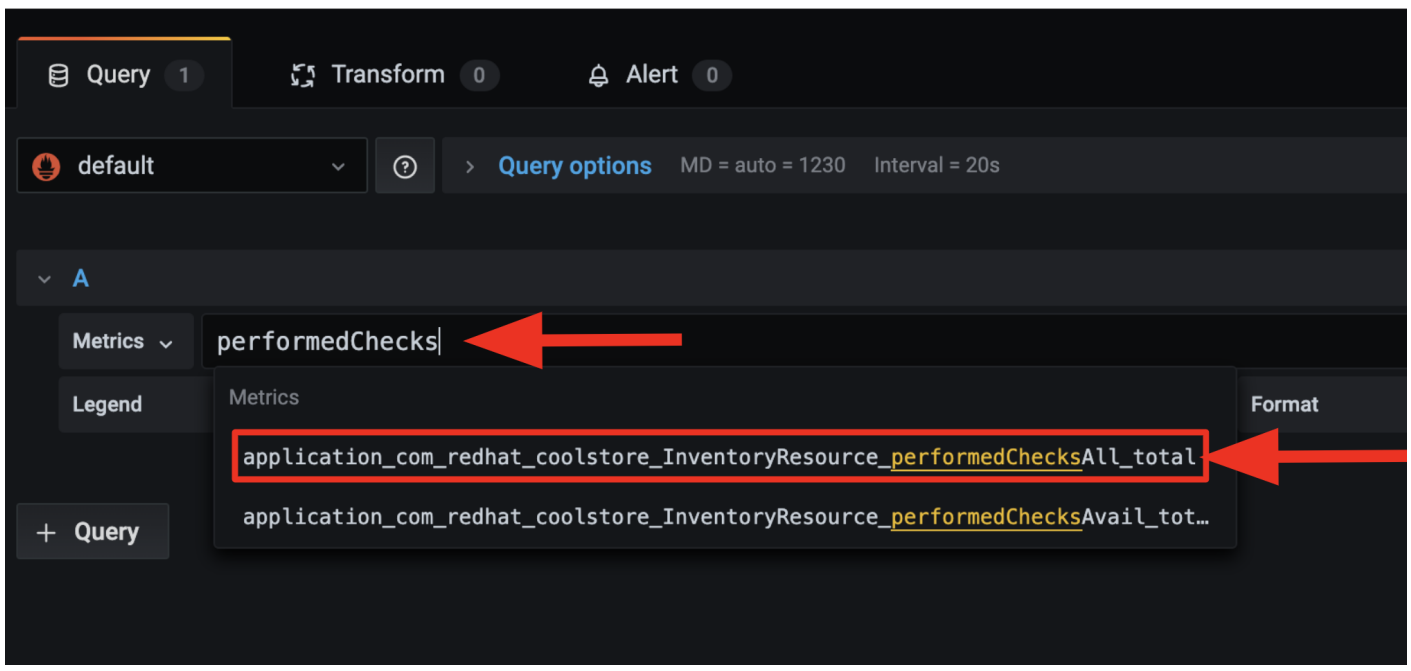
Select **New Dashboard** to create a new *Dashboard* to review the metrics.



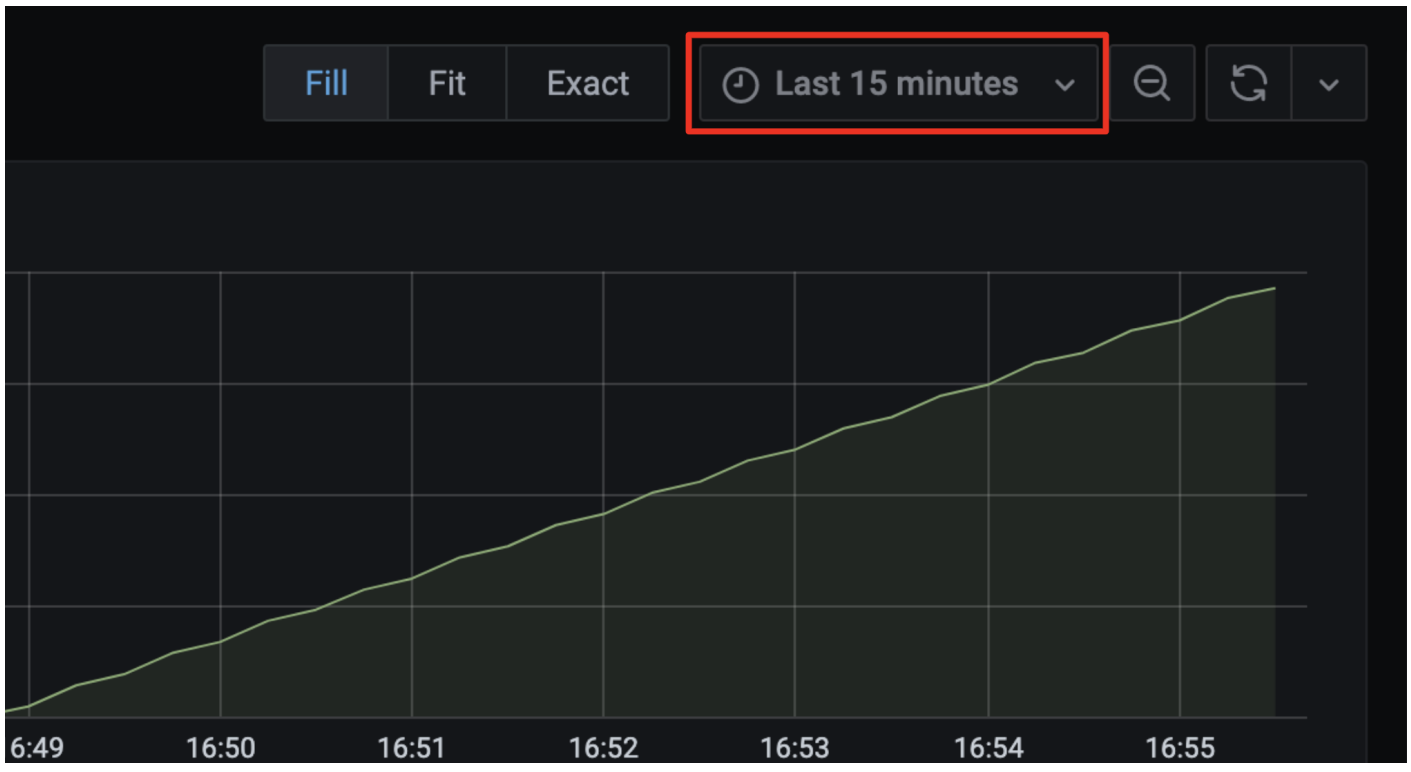
Click on **Add new panel** to add a new panel with a query:



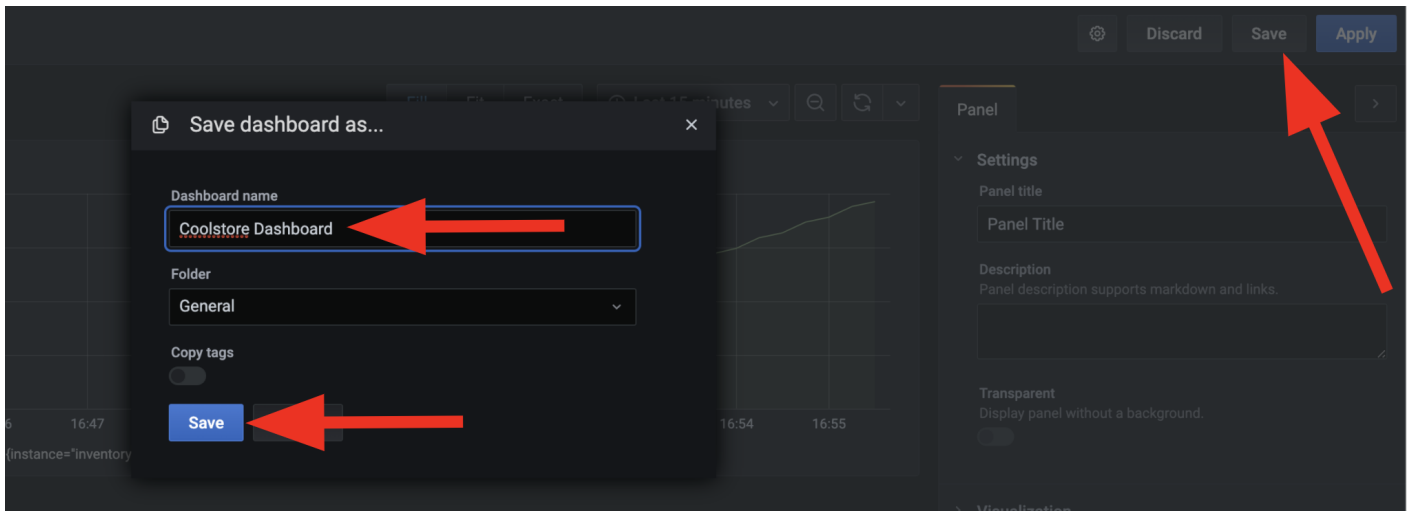
Type in `performedChecks` in the *Metrics* field, and choose the first auto-completed value:



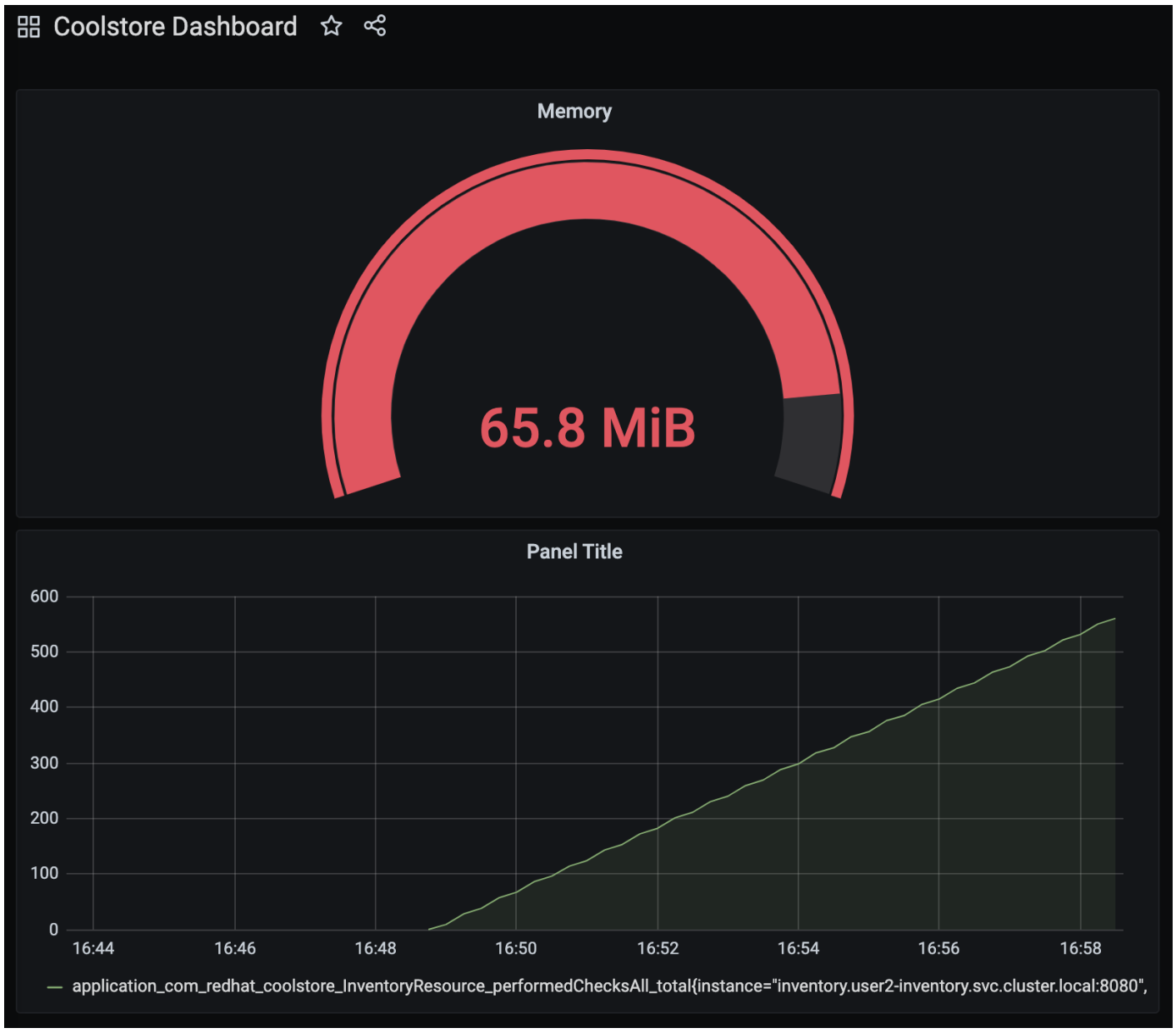
Press ENTER while the cursor is in the field, and the values should begin showing up. Choose **Last 15 Minutes** in the drop-down as shown:



You can fine tune the display, along with the type of graph (bar, line, gauge, etc). Using other options. When done, click the **Save** button, give your new dashboard a name, and click **Save**:



This is optional, but you can add more Panels if you like, for example: The JVM RSS Value `process_resident_memory_bytes` (set the Visualization to `Gauge` and the Unit in *Field* tab to `bytes(IEC)` on the *Visualization*, and the title to `Memory` on the *Panel Title*). It could look like:



You can see more examples of more complex dashboard, and even import them into your own dashboards from the Grafana Labs Dashboard community (https://grafana.com/grafana/dashboards).

Extra Credit: Spring Boot

If you feel up to it, Spring Boot can also expose Metrics which can be collected by Prometheus and displayed with Grafana. To add metrics support to your Catalog service written with Spring Boot, you'll need to:

1. Add dependencies for Spring Boot Actuator and Prometheus
2. Configure `application-openshift.properties` with config values
3. Re-build and Re-deploy the app to OpenShift (in the user16-catalog project) using commands from previous modules
4. Edit the Prometheus *ConfigMap* to add another scrape job pointing at `catalog-springboot.user16-catalog.svc.cluster.local:8080`
5. Re-deploy Prometheus to pick up the new config
6. Attempt to query Prometheus for the Spring Boot metrics

It is beyond the scope of this lab, but if you're interested, give it a go if you have extra time!

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.
