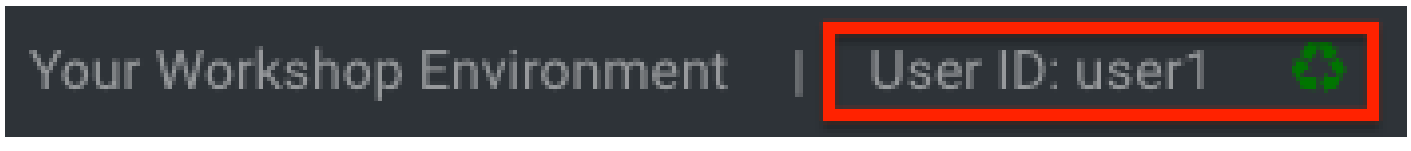
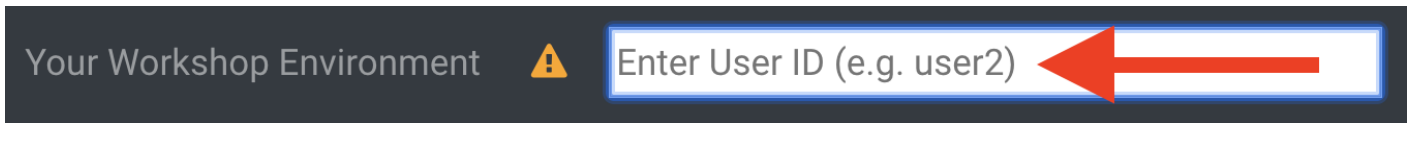
Your Workshop Environment

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.

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!

Getting Started with Application Migration Toolkit

In this module you'll work with an existing Java EE application designed for a retail webshop. The current version of the webshop is a Java EE application built for Oracle Weblogic Application Server. As part of a modernization strategy you've decided to move this application to JBoss EAP, containerize it, and run it on a Kubernetes platform with OpenShift.

What is Red Hat Application Migration Toolkit?

Red Hat Application Migration Toolkit

Red Hat Application Migration Toolkit (RHAMT) is an extensible and customizable rule-based tool that helps simplify migration of Java applications.

It is used by organizations for:

- Planning and work estimation
- Identifying migration issues and providing solutions
- Detailed reporting
- Using built-in rules and migration paths
- Rule extension and customizability
- Ability to analyze source code or application archives

Read more about it in the RHAMT documentation (https://access.redhat.com/documentation/en/red-hat-application-migration-toolkit)

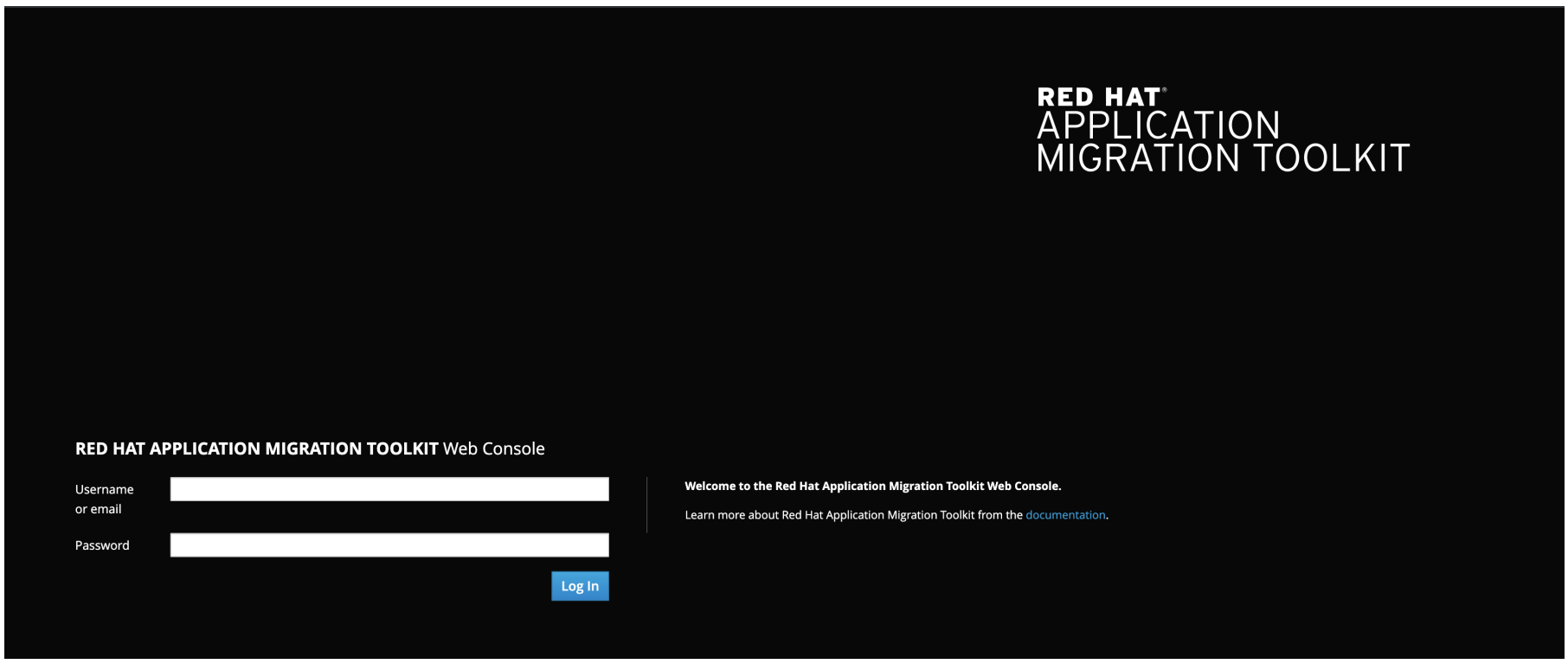
Lab 1 - Login to RHAMT and analyze app

In this step we will analyze a monolithic application built for use with Oracle® WebLogic Server (WLS). This application is a Java EE application using a number of different technologies, including standard Java EE APIs as well as proprietary Weblogic APIs and best practices.

For this lab, we will use the RHAMT Web Console on top of OpenShift Container Platform.

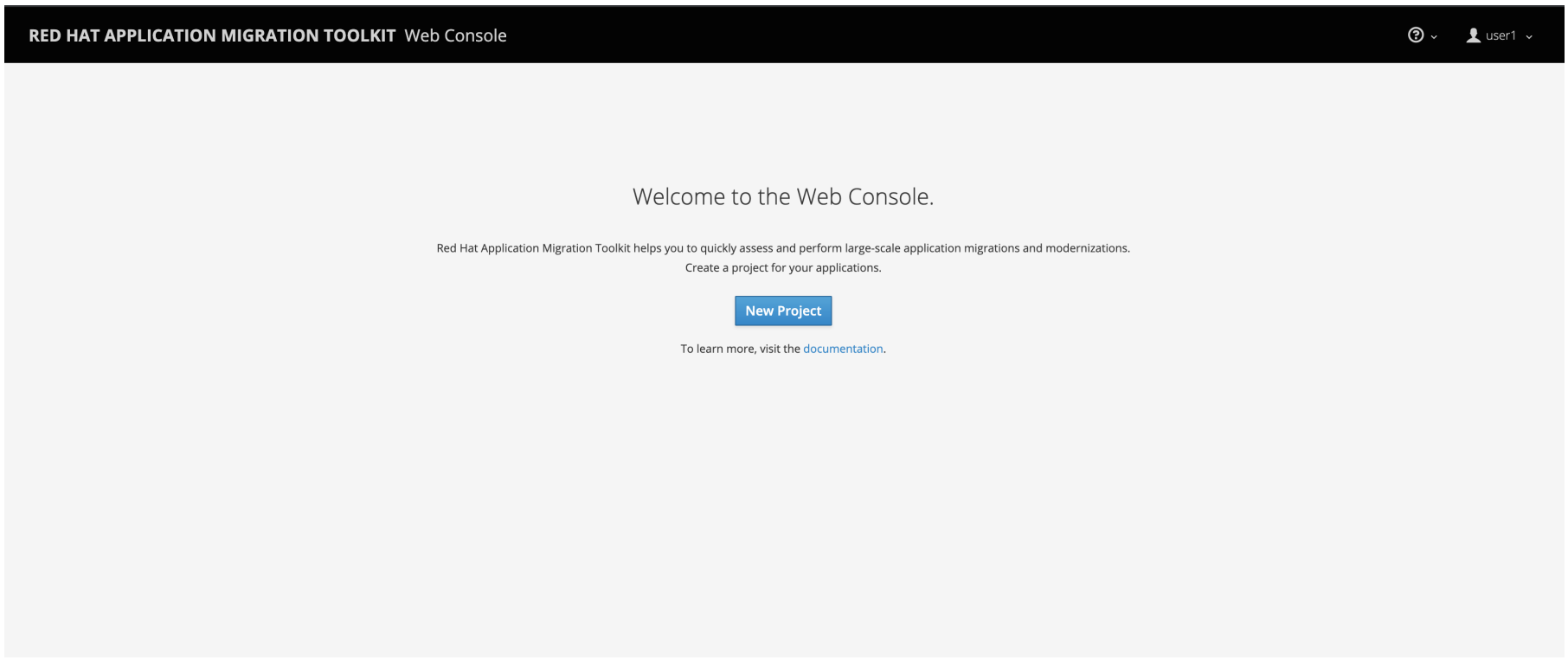
1. Login to the RHAMT web console in OpenShift cluster

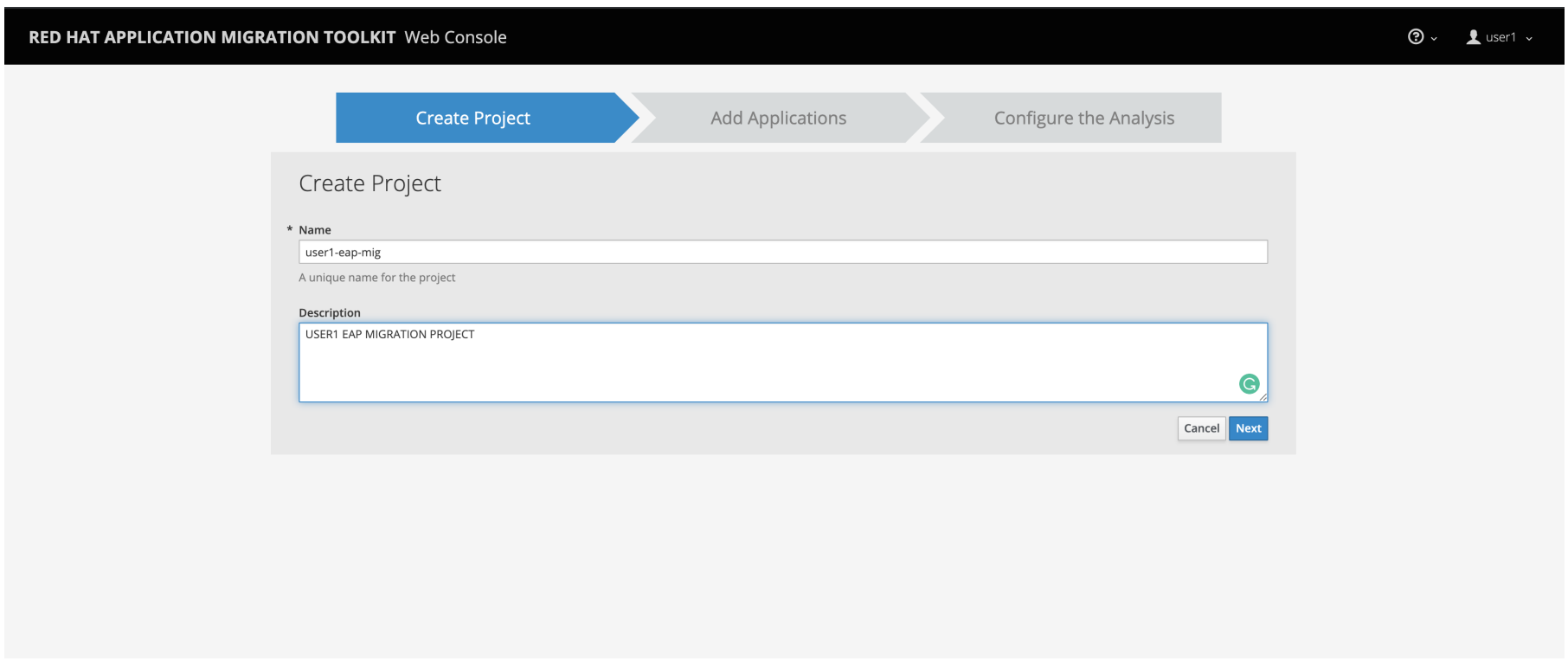
To get started, access the Red Hat Application Migration Toolkit (http://rhamt-web-console-labs-infra.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com) and log in using the username and password you've been assigned (e.g. `user16/r3dh4t1!`):



2. Create a new project

Click **New Project**. Input a name and optional description to create a project. Name the project `user16-eap-migration` to avoid conflicts with other attendees.

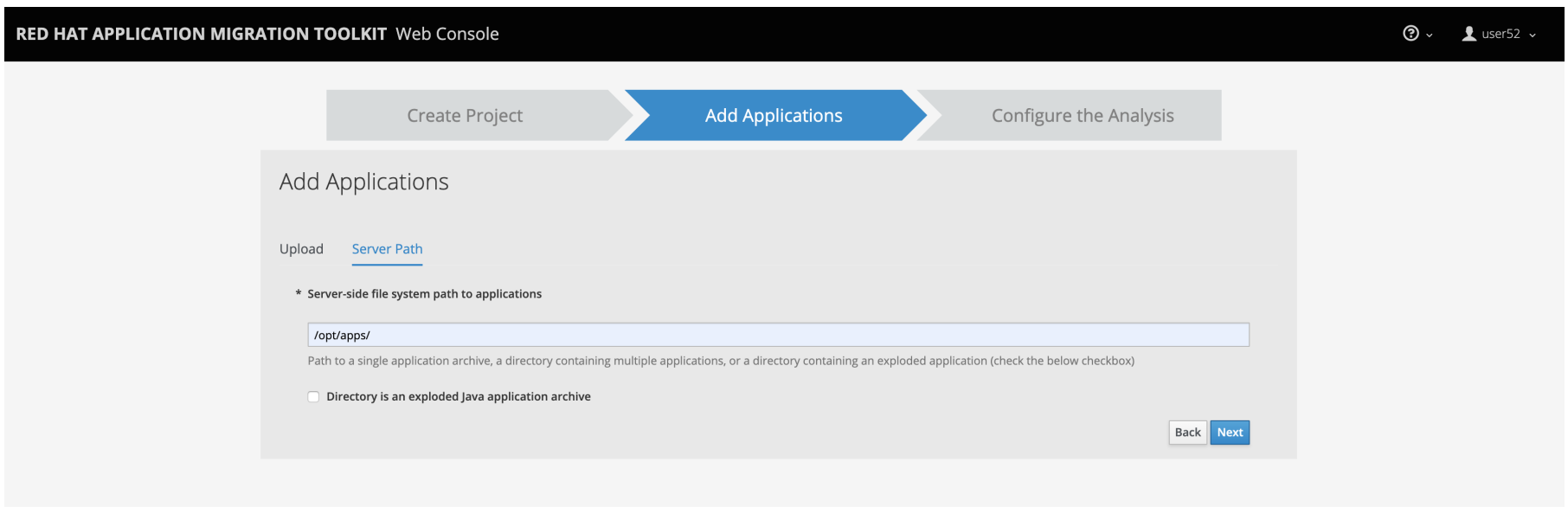




3. Add the monolith application to the project

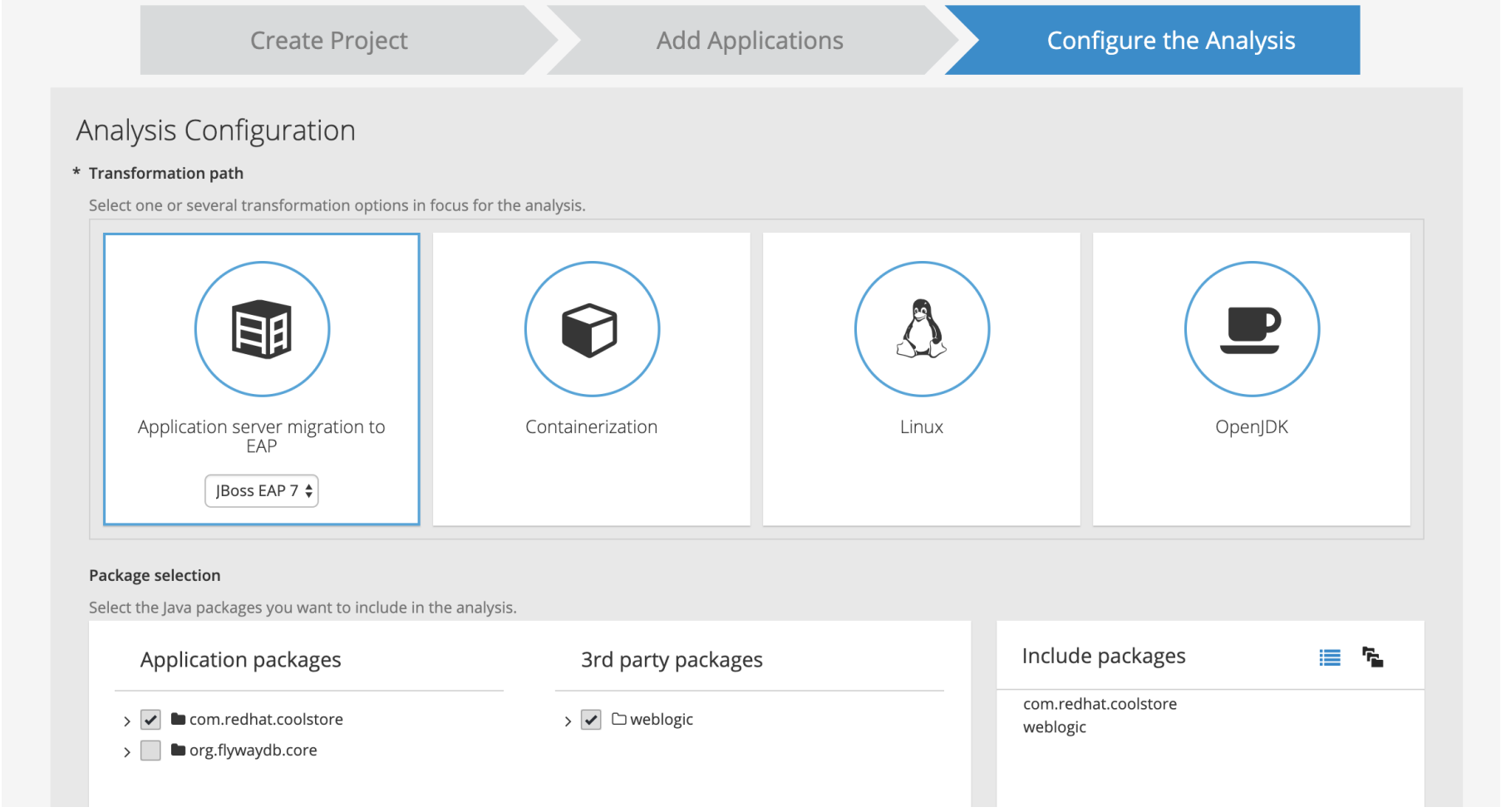
Select **Server Path** to analyze our monolithic application:

- Server Path: `/opt/apps`



4. Select *Migration to JBoss EAP 7* in Transformation Path

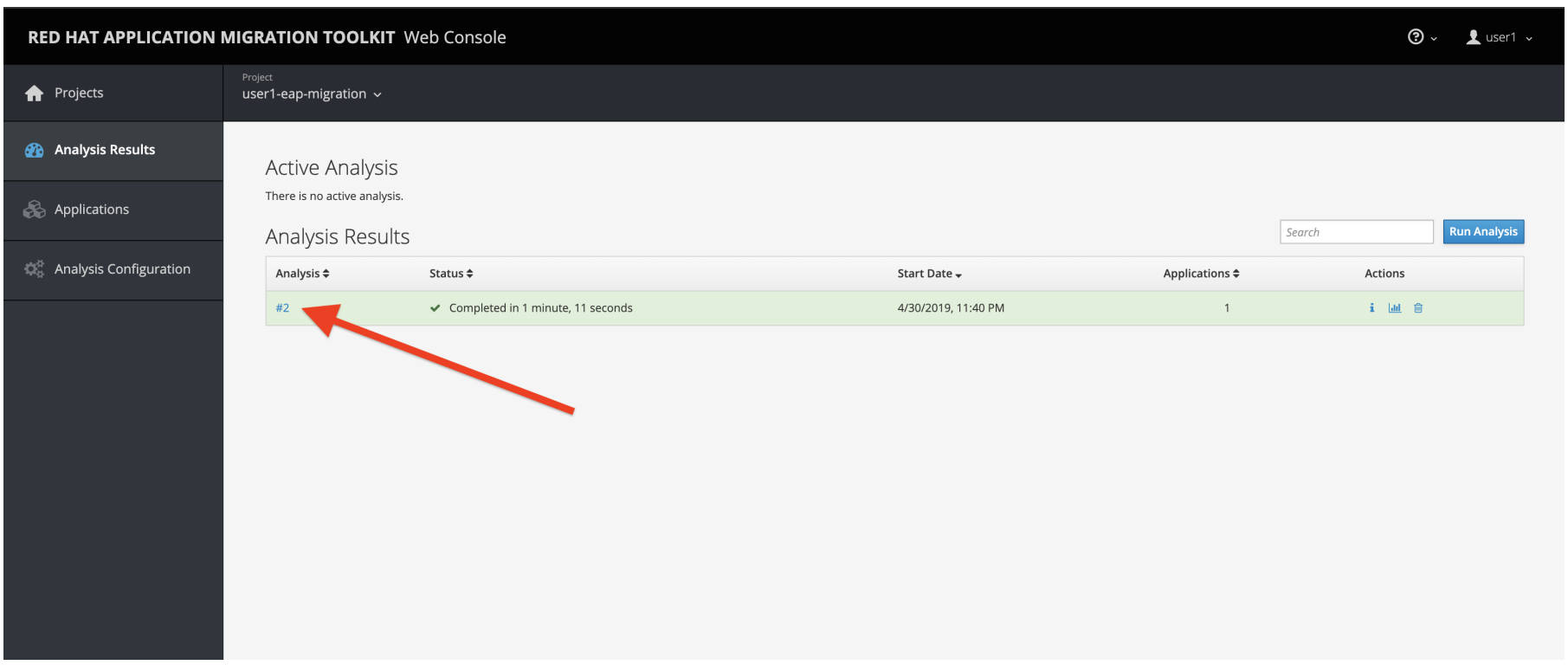
Choose the `com` and `weblogic` checkboxes to include these packages during analysis and click the **Save & Run** button. You will be taken to Analysis Results dashboard page, wait until the analysis is complete (it will take a minute or two).



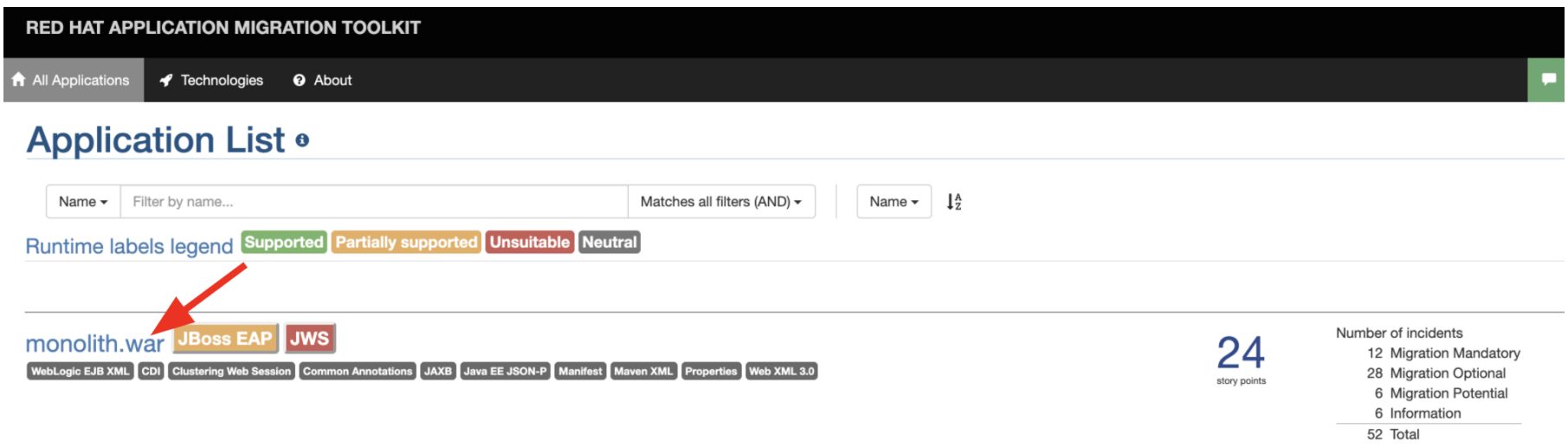
5. Go to the Active Analysis page and click on the latest when it's completed

> Your report may be *queued* for a few seconds. Soon you will see a progress bar, then when the report is complete you can continue. If it seems to be queued for more than a minute, try refreshing the browser page.

Click the #1 link (or #2) to see the report:

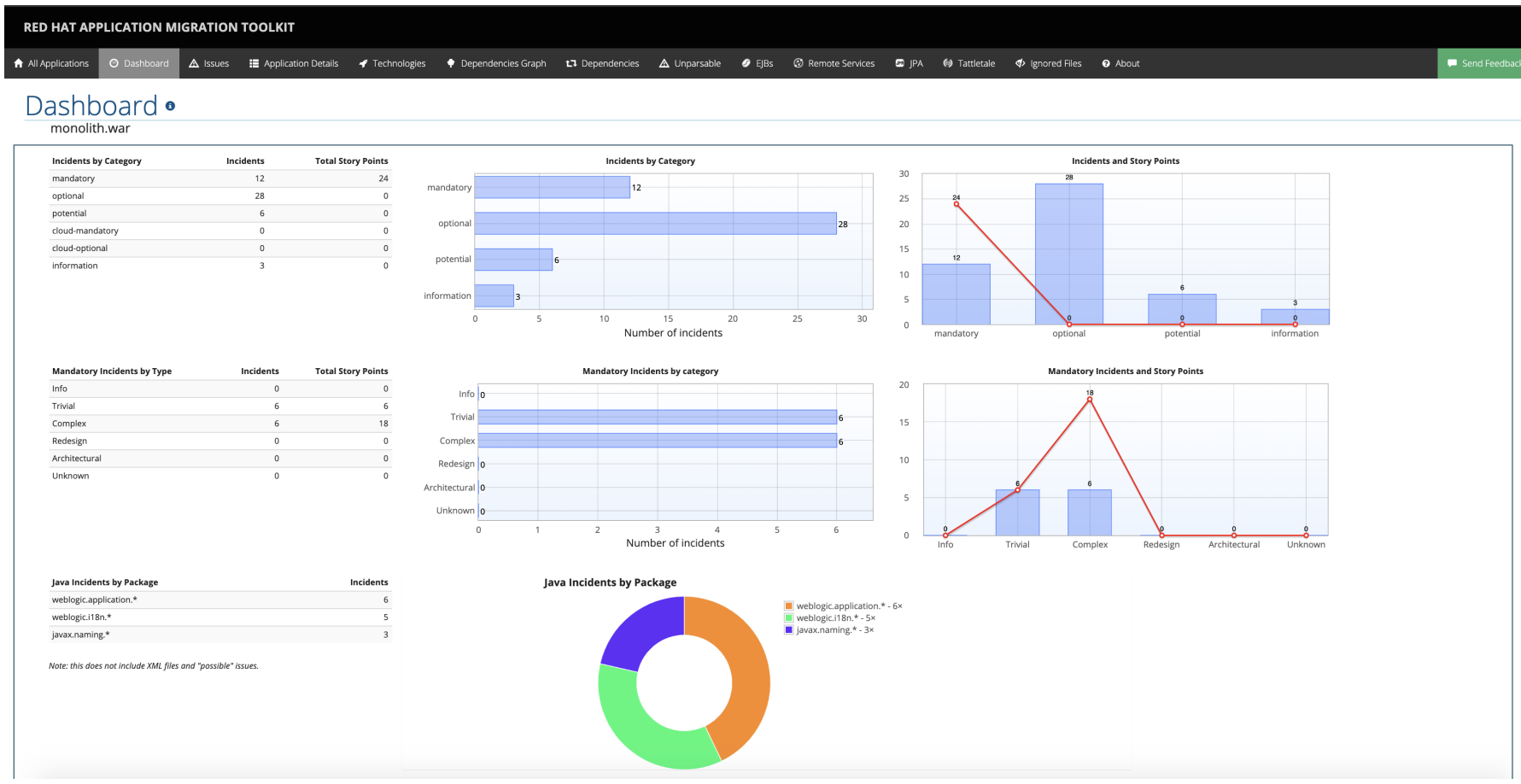


6. Review the report



The main landing page of the report lists the applications that were processed. Each row contains a high-level overview of the story points, number of incidents, and technologies encountered in that application.

Click on the `monolith.war` link to access details for the project:



7. Understanding the report

The Dashboard gives an overview of the entire application migration effort. It summarizes:

- The incidents and story points by category
- The incidents and story points by level of effort of the suggested changes
- The incidents by package

> Story points are an abstract metric commonly used in Agile software development to estimate the relative level of effort needed to implement a feature or change. Red Hat Application Migration Toolkit uses story points to express the level of effort needed to migrate particular application constructs, and the application as a whole. The level of effort will vary greatly depending on the size and complexity of the application(s) to migrate.

You can use this report to estimate how easy/hard each app is, and make decisions about which apps to migrate, which to refactor, and which to leave alone. In this case we will do a straight migration to JBoss EAP.

On to the next step to change the code!

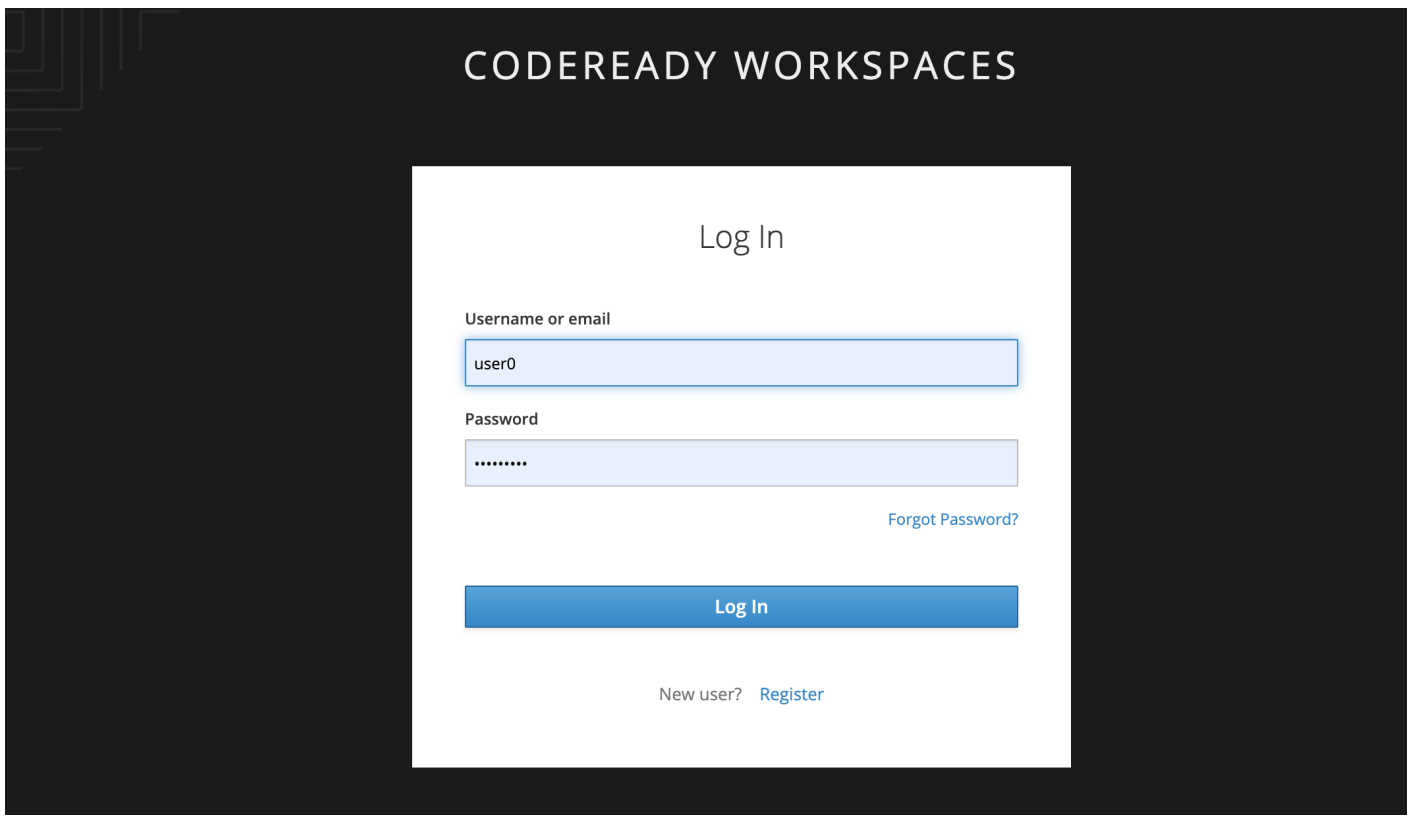
Migrate to JBoss EAP

In this step you will migrate some Weblogic-specific code in the app to use standard (Jakarta EE) interfaces.

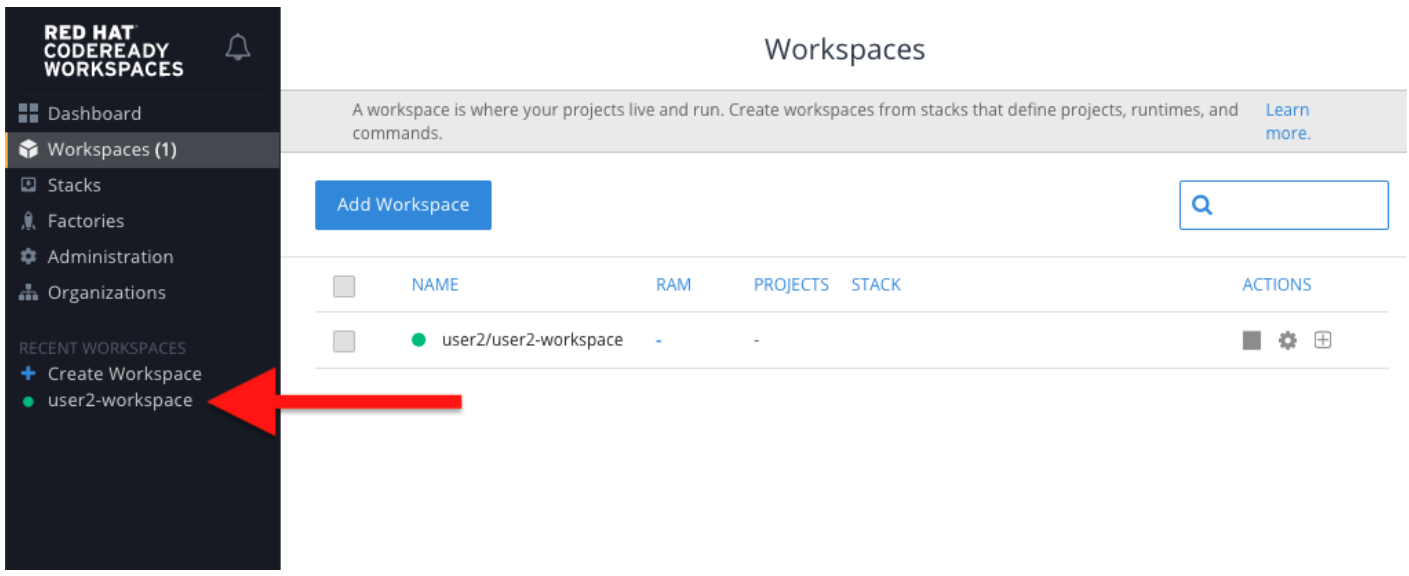
1. Access Your Development Environment

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.

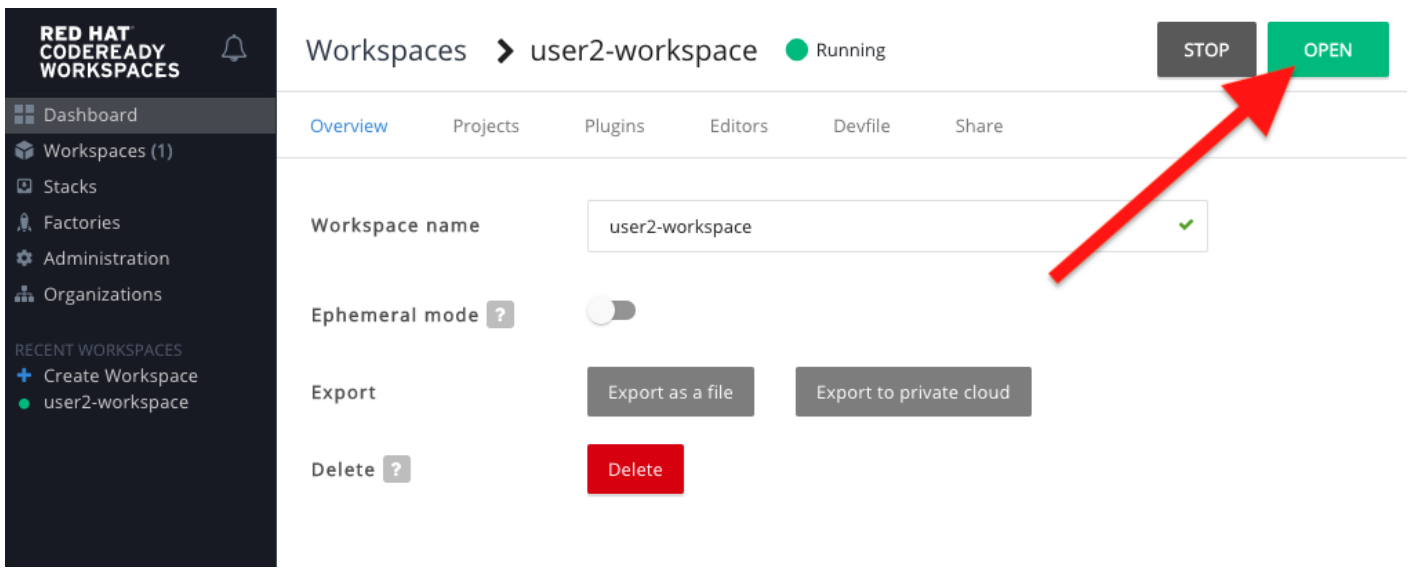
To get started, access the CodeReady Workspaces instance (https://codeready-labs-infra.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com) and log in using the username and password you've been assigned (e.g. `user16/r3dh4t1!`):



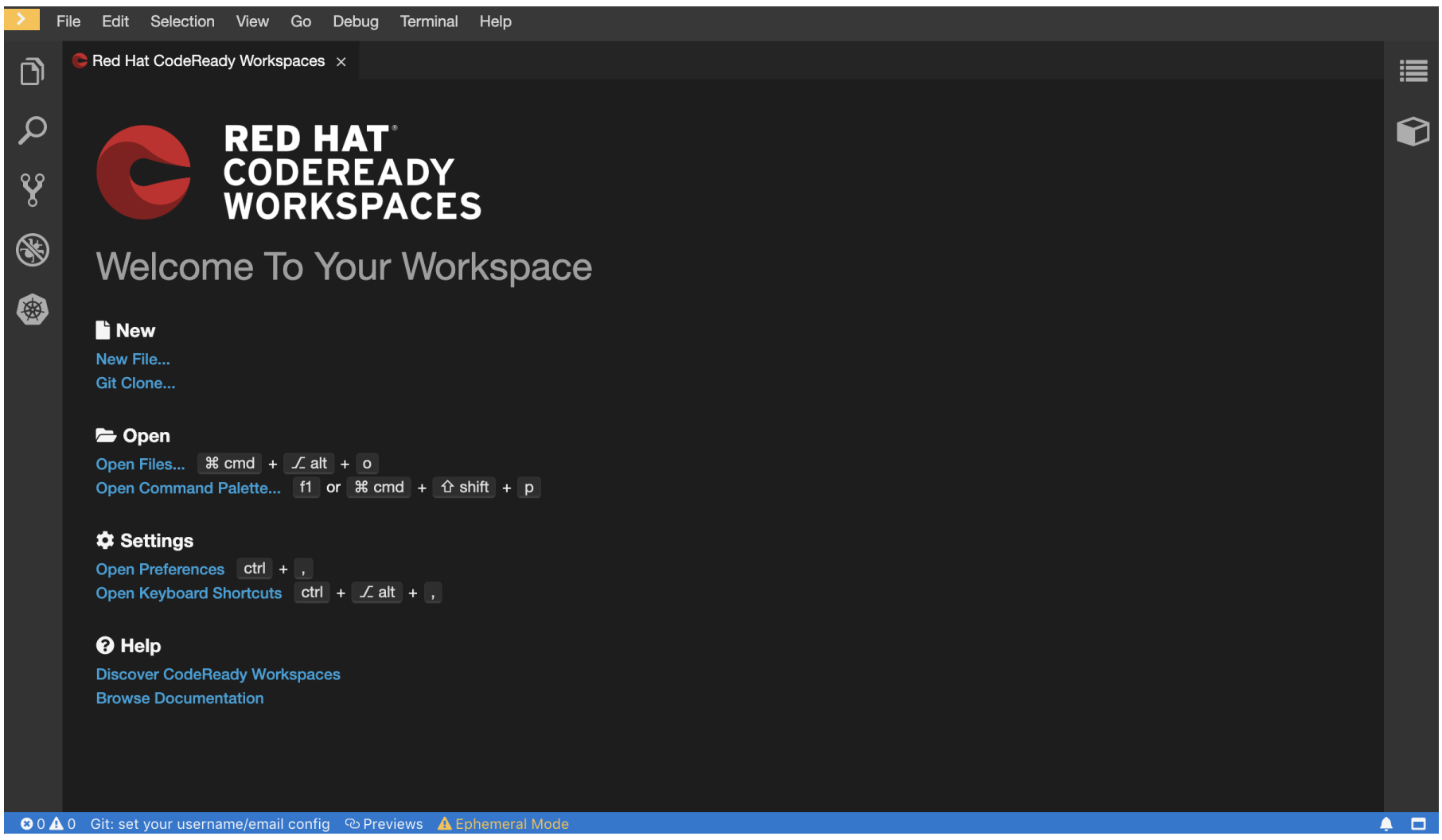
Once you log in, you'll be placed on your personal dashboard. Click on the name of the pre-created workspace on the left, as shown below (the name will be different depending on your assigned number).



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:

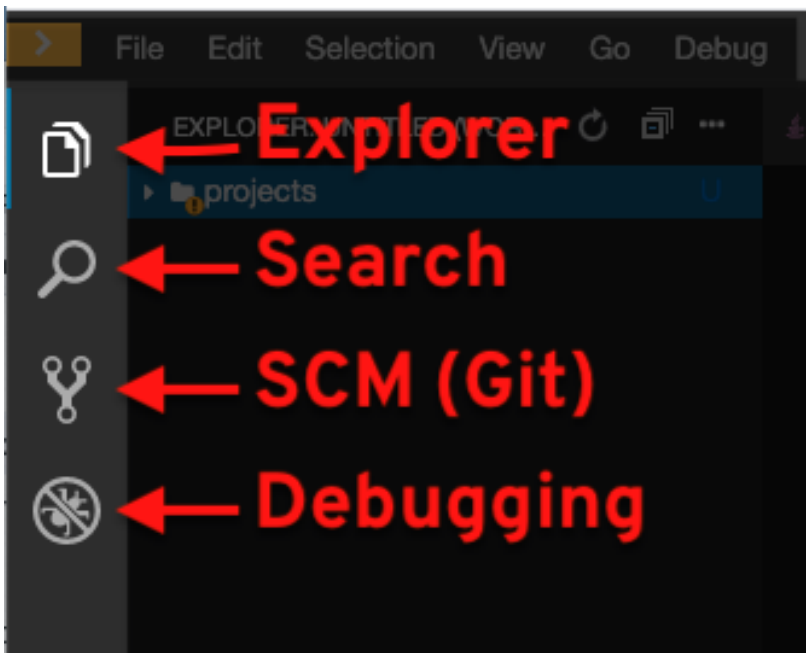


After a minute or two, you'll be placed in the workspace:



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:

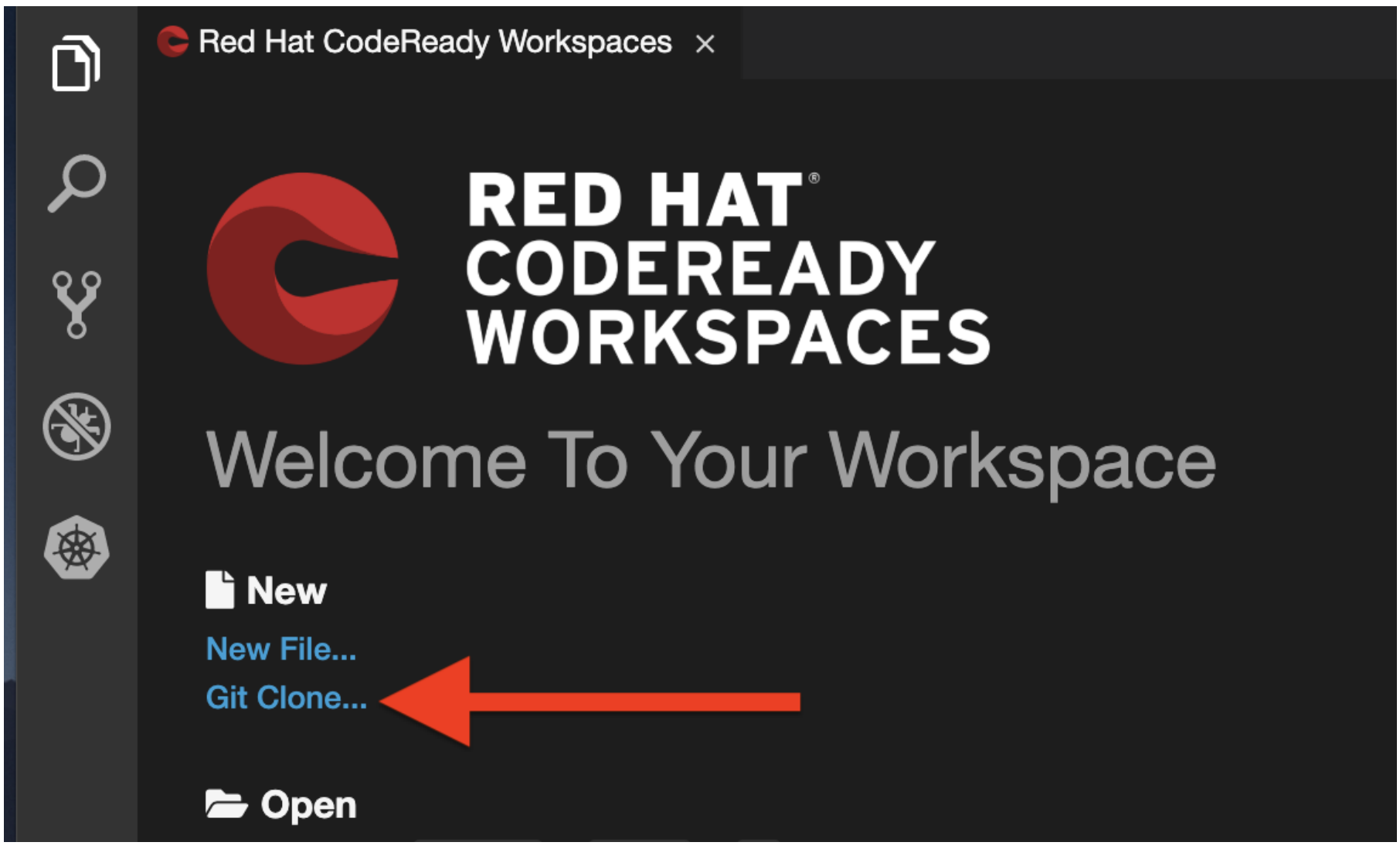


If things get weird or your browser appears, you can simply reload the browser tab to refresh the view.

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).

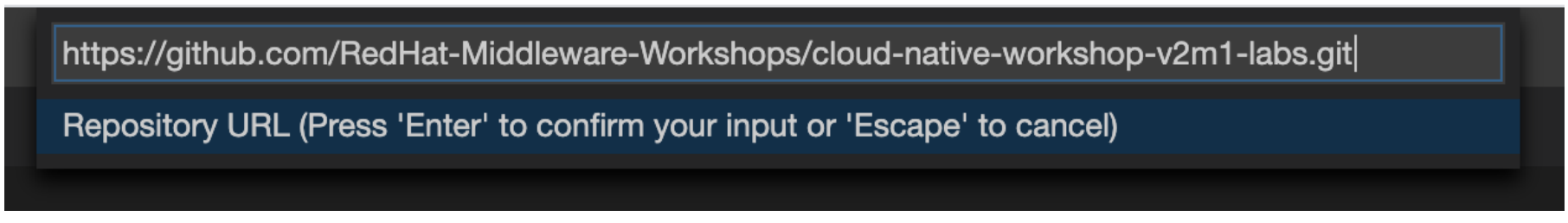
If you ever need to run commands that you don't see in a menu, you can press `F1` to open the command window, or the more traditional `Control` + `SHIFT` + `P` (or `Command` + `SHIFT` + `P` on Mac OS X).

Let's import our first project. Click on **Git Clone..** (or type `F1`, enter 'git' and click on the auto-completed *Git Clone..*)

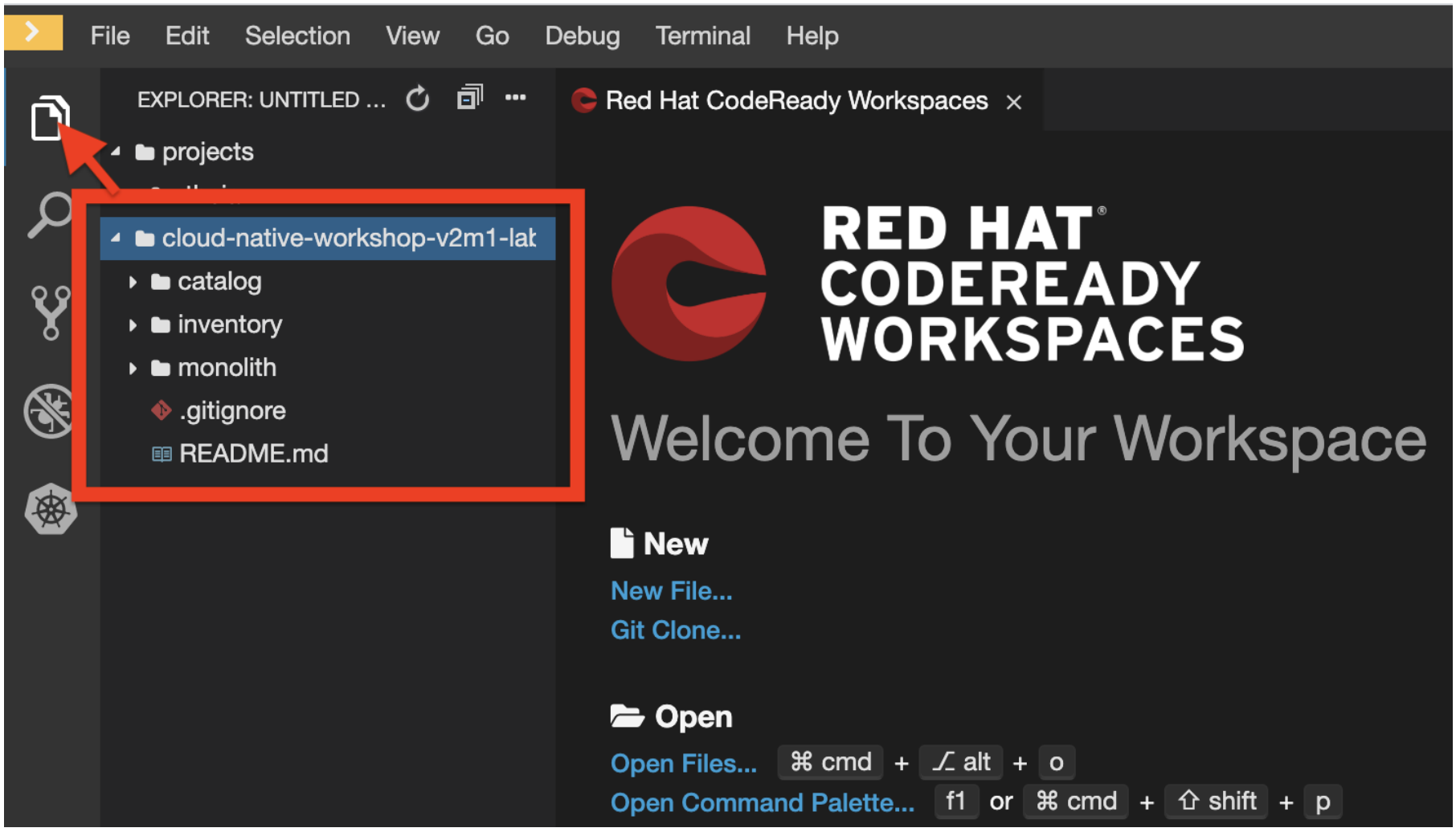


Step through the prompts, using the following value for **Repository URL**. If you use **FireFox**, it may end up pasting extra spaces at the end, so just press backspace after pasting:

```NONE
https://github.com/RedHat-Middleware-Workshops/cloud-native-workshop-v2m1-labs.git
```



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

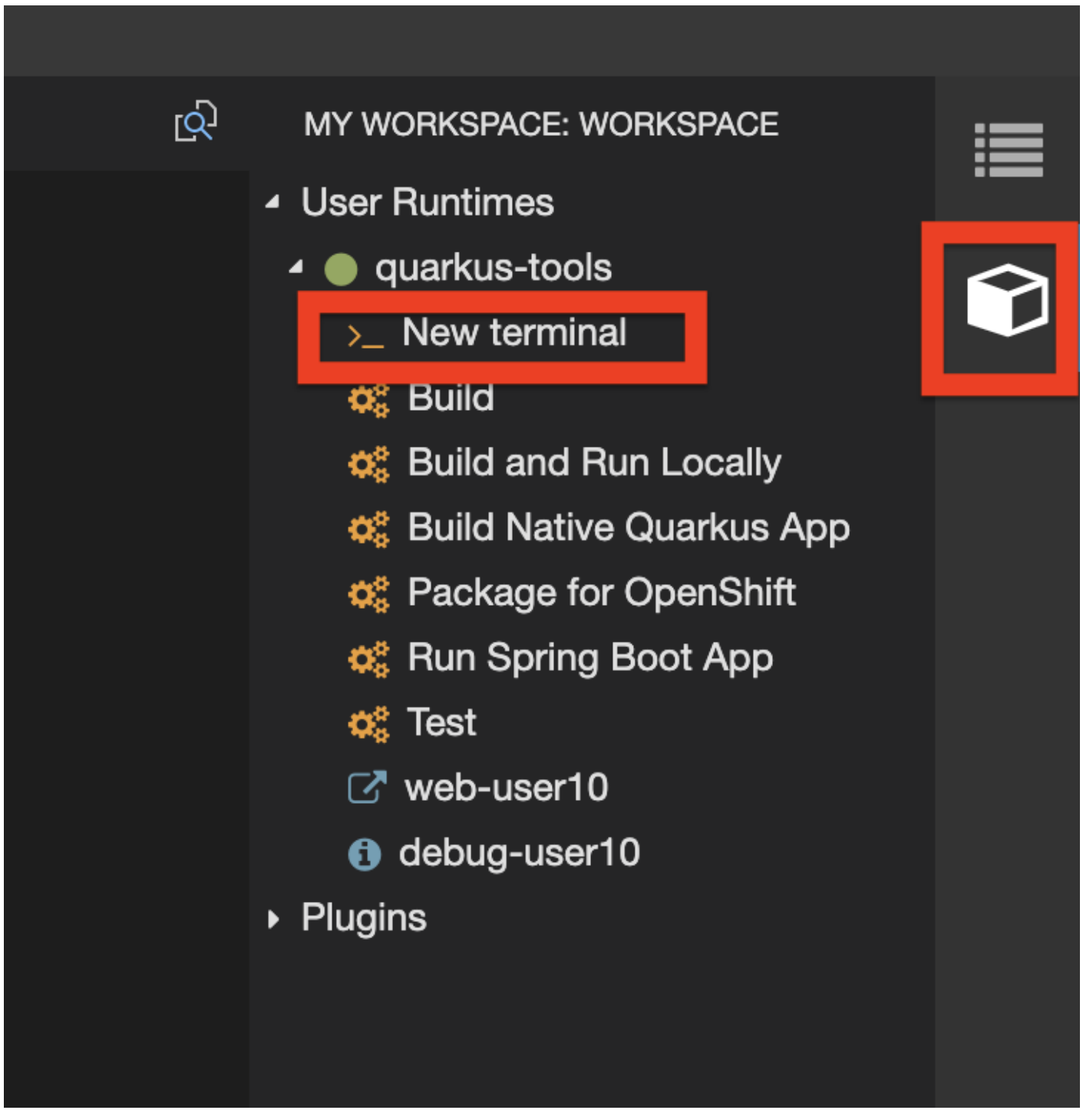


IMPORTANT: Check out proper Git branch

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

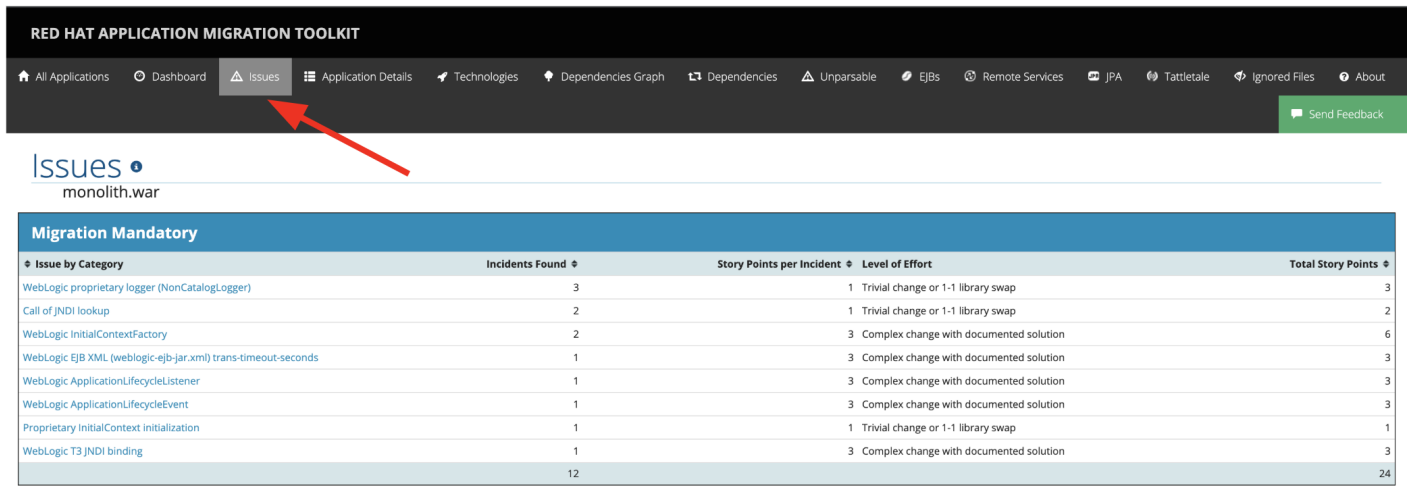
```sh
cd $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m1-labs && git checkout ocp-4.4
```

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:



2. Fix the issue with `ApplicationLifecycleListener`

Open the Issues report in the RHAMT Console (http://rhamt-web-console-labs-infra.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com):



RHAMT provides helpful links to understand the issue deeper and offer guidance for the migration.

The WebLogic `ApplicationLifecycleListener` abstract class is used to perform functions or schedule jobs in Oracle WebLogic, like server start and stop. In this case we have code in the `postStart` and `preStop` methods which are executed after Weblogic starts up and before it shuts down, respectively.

In Jakarta EE, there is no equivalent to intercept these events, but you can get equivalent functionality using a *Singleton EJB* with standard annotations, as suggested in the issue in the RHAMT report.

We will use the `@Startup` annotation to tell the container to initialize the singleton session bean at application start. We will similarly use the `@PostConstruct` and `@PreDestroy` annotations to specify the methods to invoke at the start and end of the application lifecyle achieving the same result but without using proprietary interfaces.

Using this method makes the code much more portable.

3. Fix the ApplicationLifecycleListener issues

To begin we are fixing the issues under the Monolith application. Navigate to the `cloud-native-workshop-v2m1-labs` folder in the project tree, then open the file `monolith/src/main/java/com/redhat/coolstore/utils/StartupListener.java` by clicking on it.

Replace the file content with:

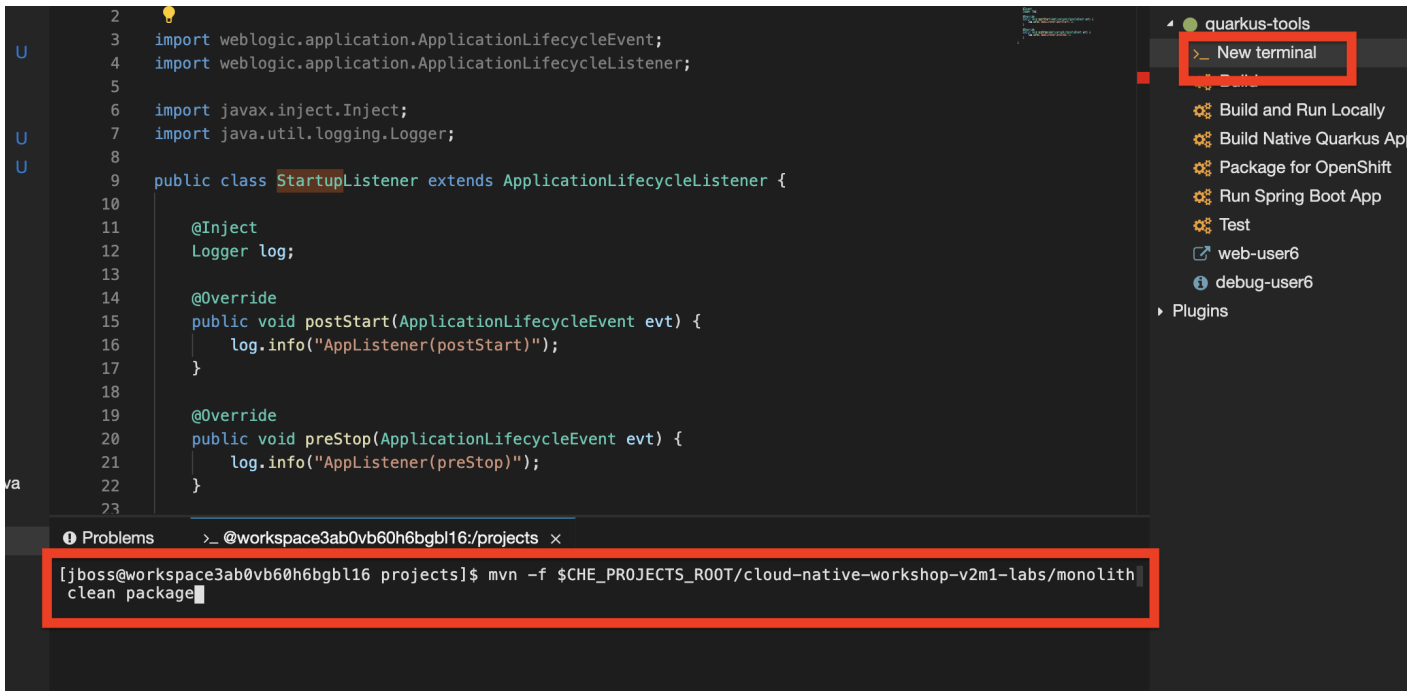
```java
package com.redhat.coolstore.utils;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.Startup;
import javax.inject.Singleton;
import javax.inject.Inject;
import java.util.logging.Logger;

@Singleton
@Startup
public class StartupListener {

    @Inject
    Logger log;

    @PostConstruct
    public void postStart() {
        log.info("AppListener(postStart)");
    }

    @PreDestroy
    public void preStop() {
        log.info("AppListener(preStop)");
    }

}
```

Where is the Save button? CodeReady workspaces will autosave your changes, that is why you can't find a SAVE button - no more losing code because you forgot to save. You can undo with `CTRL-Z` (or `CMD-Z` on a Mac) or by using the `Edit → Undo` menu option.

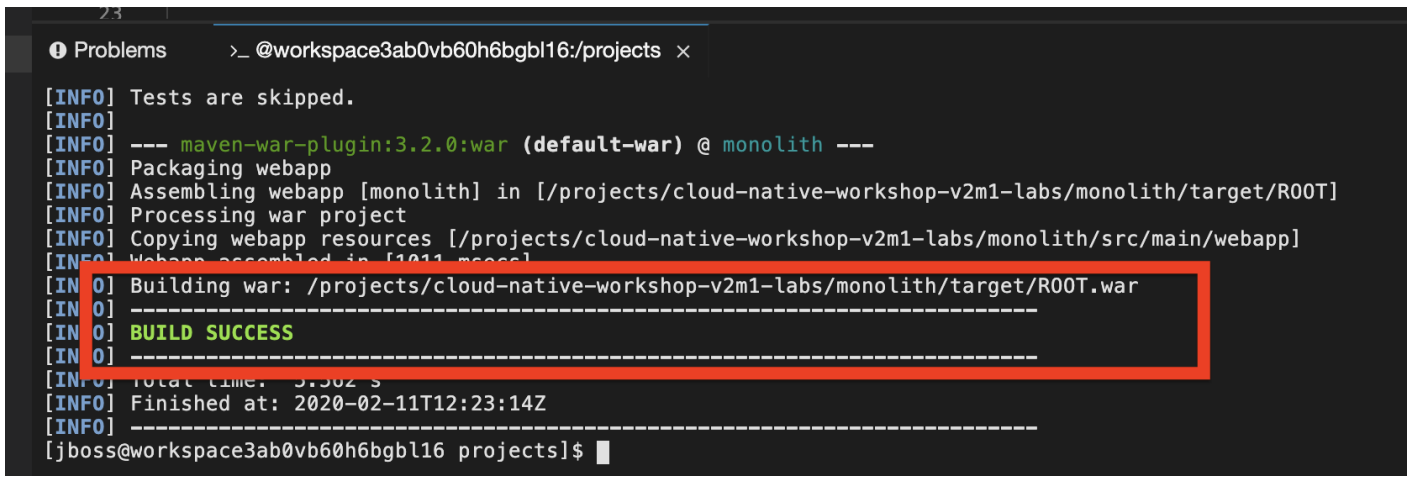
4. Test the build

Open a new Terminal window under the `quarkus-tools` container (on the right). In the terminal, issue the following command to test the build:

```sh
mvn -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m1-labs/monolith clean package
```

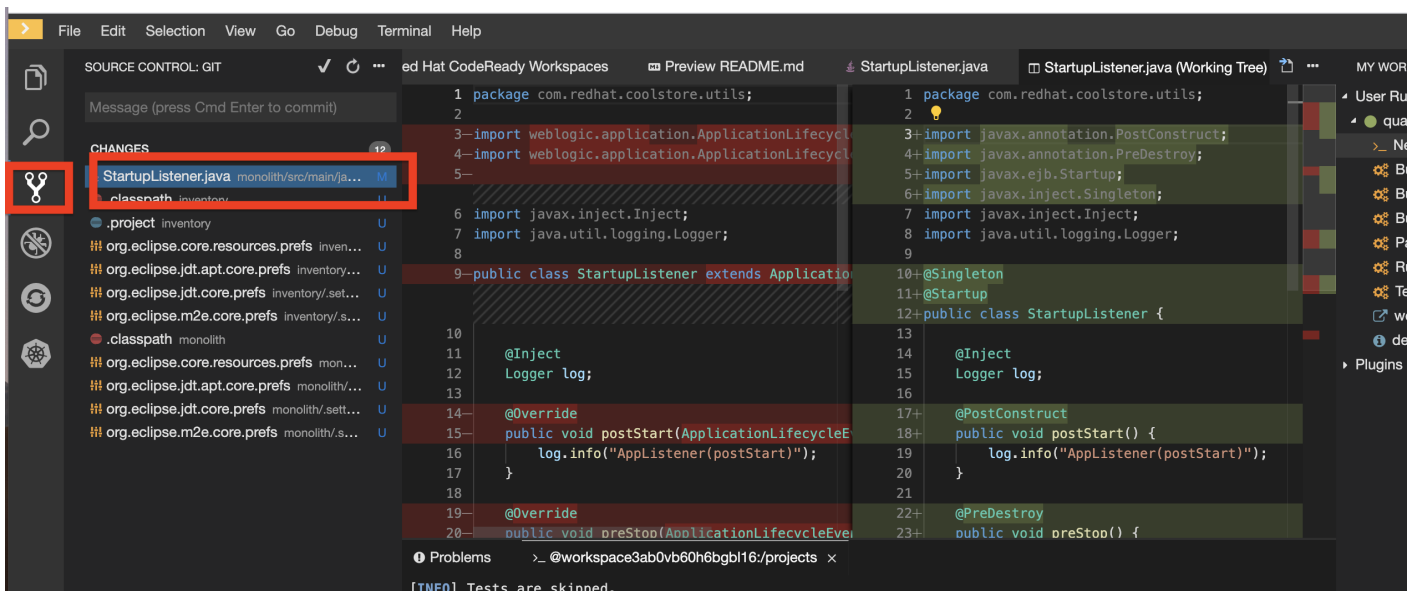


If it builds successfully (you will see `BUILD SUCCESS`), let's move on to the next issue! If it does not compile, verify you made all the changes correctly and try the build again.



View the diffs

You can review the changes you've made. On the left, click on the *Version Control* icon, which shows a list of the changed files. Double-click on `StartupListener.java` to view the differences you've made:



CodeReady keeps track (using Git) of the changes you make, and you can use version control to check in, update, and compare files as you change them.

For now, go back to the *Explorer* tree and lets fix the remaining issues.

5. Fix the logger issues

Some of our application makes use of Weblogic-specific logging methods like the `NonCatalogLogger`, which offer features related to logging of internationalized content, and client-server logging.

The WebLogic `NonCatalogLogger` is not supported on JBoss EAP (or any other Java EE platform), and should be migrated to a supported logging framework, such as the JDK Logger or JBoss Logging.

We will use the standard Java Logging framework, a much more portable framework. The framework also supports internationalization (https://docs.oracle.com/javase/8/docs/technotes/guides/logging/overview.html#a1.17) if needed.

In the same `monolith` directory, open the `src/main/java/com/redhat/coolstore/service/OrderServiceMDB.java` file and replace its contents with:

JAVA
```java
package com.redhat.coolstore.service;

import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.inject.Inject;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

import com.redhat.coolstore.model.Order;
import com.redhat.coolstore.utils.Transformers;

import java.util.logging.Logger;

@MessageDriven(name = "OrderServiceMDB", activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup", propertyValue = "topic/orders"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Topic"),
    @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-acknowledge")})
public class OrderServiceMDB implements MessageListener {

    @Inject
    OrderService orderService;

    @Inject
    CatalogService catalogService;

    private Logger log = Logger.getLogger(OrderServiceMDB.class.getName());

    @Override
    public void onMessage(Message rcvMessage) {
        TextMessage msg = null;
        try {
                if (rcvMessage instanceof TextMessage) {
                        msg = (TextMessage) rcvMessage;
                        String orderStr = msg.getBody(String.class);
                        log.info("Received order: " + orderStr);
                        Order order = Transformers.jsonToOrder(orderStr);
                        log.info("Order object is " + order);
                        orderService.save(order);
                        order.getItemList().forEach(orderItem -> {
                            catalogService.updateInventoryItems(orderItem.getProductId(), orderItem.getQuantity());
                        });
                }
        } catch (JMSException e) {
            throw new RuntimeException(e);
        }
    }

}
```

That one was pretty easy.

6. Test the build

Build and package the app again just as before:

SH
```sh
mvn -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m1-labs/monolith clean package
```

If builds successfully (you will see `BUILD SUCCESS`), then let's move on to the next issue! If it does not compile, verify you made all the changes correctly and try the build again.

Fix issues with MDBs

In this final step we will again migrate some Weblogic-specific code in the app to use standard Java EE interfaces, and one JBoss-specific interface.

Our application uses JMS (https://en.wikipedia.org/wiki/Java_Message_Service) to communicate. Each time an order is placed in the application, a JMS message is sent to a JMS Topic, which is then consumed by listeners (subscribers) to that topic to process the order using Message-driven beans (https://docs.oracle.com/javaee/6/tutorial/doc/gipko.html), a form of Enterprise JavaBeans (EJBs) that allow Java EE applications to process messages asynchronously.

In this case, `InventoryNotificationMDB` is subscribed to and listening for messages from `ShoppingCartService`. When an order comes through the `ShoppingCartService`, a message is placed on the JMS Topic. At that point, the `InventoryNotificationMDB` receives a message and if the inventory service is below a pre-defined threshold, sends a message to the log indicating that the supplier of the product needs to be notified.

Unfortunately this MDB was written a while ago and makes use of weblogic-proprietary interfaces to configure and operate the MDB. RHAMT has flagged this and reported it using a number of issues.

JBoss EAP provides an even more efficient and declarative way to configure and manage the lifecycle of MDBs. In this case, we can use annotations to provide the necessary initialization and configuration logic and settings. We will use the `@MessageDriven` and `@ActivationConfigProperty` annotations, along with the `MessageListener` interfaces to provide the same functionality as from Weblogic.

Much of Weblogic's interfaces for EJB components like MDBs reside in Weblogic descriptor XML files. Open `src/main/webapp/WEB-INF/weblogic-ejb-jar.xml` to see one of these descriptors. There are many different configuration possibilities for EJBs and MDBs in this file, but luckily our application only uses one of them, namely it configures `<trans-timeout-seconds>` to 30, which means that if a given transaction within an MDB operation takes too long to complete (over 30 seconds), then the transaction is rolled back and exceptions are thrown. This interface is Weblogic-specific so we'll need to find an equivalent in JBoss.

You should be aware that this type of migration is more involved than the previous steps, and in real world applications it will rarely be as simple as changing one line at a time for a migration. Consult the RHAMT documentation (https://access.redhat.com/documentation/en/red-hat-application-migration-toolkit) for more detail on Red Hat's Application Migration strategies or contact your local Red Hat representative to learn more about how Red Hat can help you on your migration path.

7. Review the issues

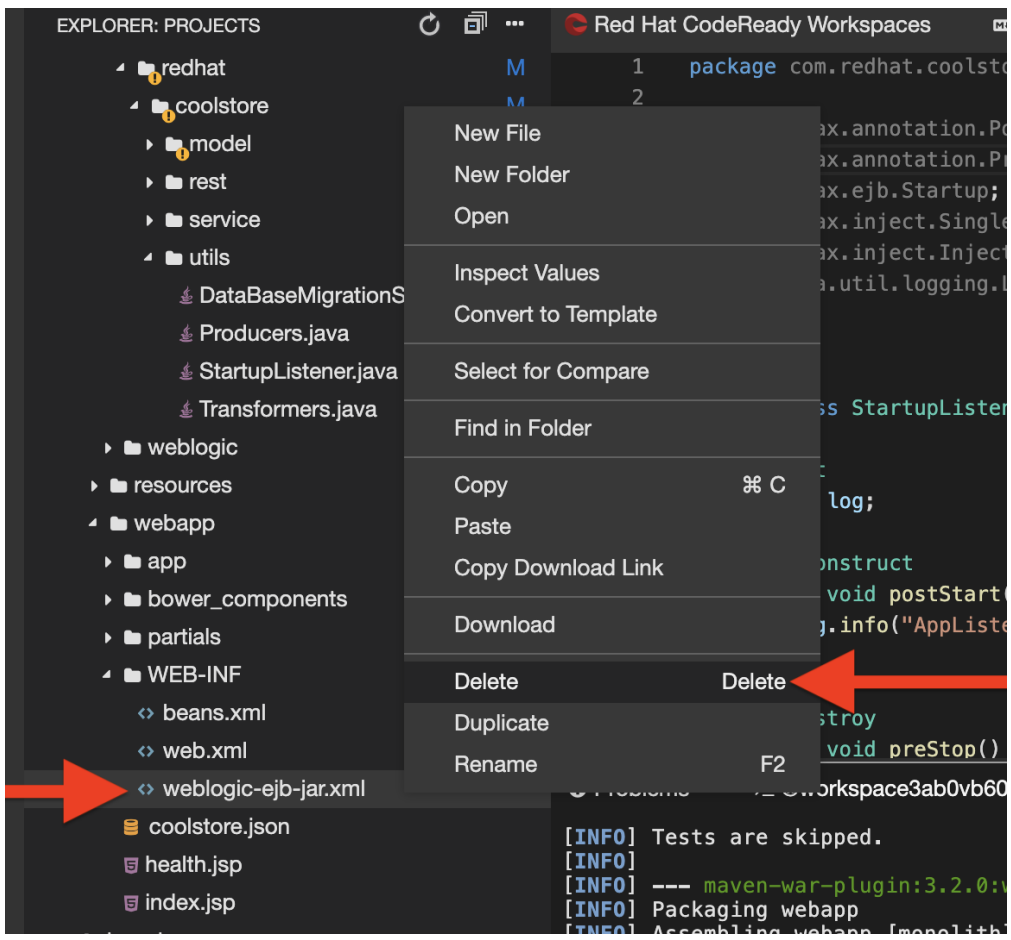
From the RHAMT Issues report, we will fix the remaining issues:

- `Call of JNDI lookup` - Our apps use a weblogic-specific JNDI (https://en.wikipedia.org/wiki/Java_Naming_and_Directory_Interface) lookup scheme.
- `Proprietary InitialContext initialization` - Weblogic has a very different lookup mechanism for InitialContext objects
- `WebLogic InitialContextFactory` - This is related to the above, essentially a Weblogic proprietary mechanism
- `WebLogic T3 JNDI binding` - The way EJBs communicate in Weblogic is over T2, a proprietary implementation of Weblogic.

All of the above interfaces have equivalents in JBoss, however they are greatly simplified and overkill for our application which uses JBoss EAP's internal message queue implementation provided by Apache ActiveMQ Artemis (https://activemq.apache.org/artemis/).

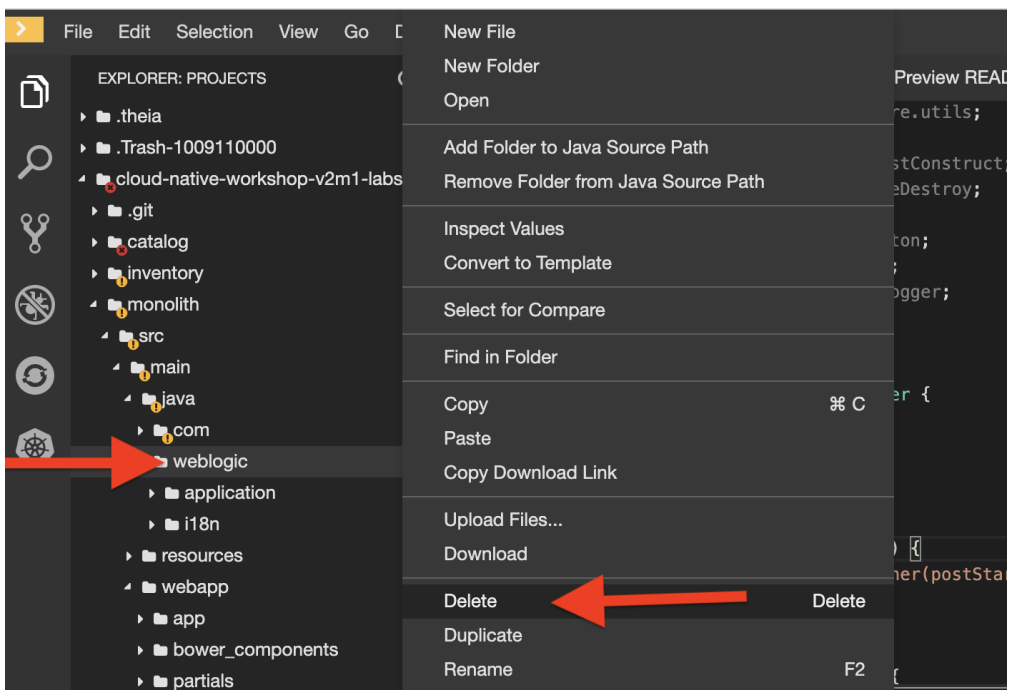
8. Remove the weblogic EJB Descriptors

The first step is to remove the unneeded `weblogic-ejb-jar.xml` file. This file is proprietary to Weblogic and not recognized or processed by JBoss EAP. Delete the file by right-clicking on the `src/main/webapp/WEB-INF/weblogic-ejb-jar.xml` file and choose **Delete**, and click **OK**.



While we're at it, let's remove the stub weblogic implementation classes added as part of the scenario.

Right-click on the `src/main/java/weblogic` folder and select **Delete** to delete the folder:



9. Fix the code

Open the `monolith/src/main/java/com/redhat/coolstore/service/InventoryNotificationMDB.java` file and replace its contents with:

```java
package com.redhat.coolstore.service;

import com.redhat.coolstore.model.Order;
import com.redhat.coolstore.utils.Transformers;

import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.inject.Inject;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;
import java.util.logging.Logger;

@MessageDriven(name = "InventoryNotificationMDB", activationConfig = {
        @ActivationConfigProperty(propertyName = "destinationLookup", propertyValue = "topic/orders"),
        @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Topic"),
        @ActivationConfigProperty(propertyName = "transactionTimeout", propertyValue = "30"),
        @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-acknowledge")})
public class InventoryNotificationMDB implements MessageListener {

    private static final int LOW_THRESHOLD = 50;

    @Inject
    private CatalogService catalogService;

    @Inject
    private Logger log;

    public void onMessage(Message rcvMessage) {
        TextMessage msg;
        {
            try {
                if (rcvMessage instanceof TextMessage) {
                    msg = (TextMessage) rcvMessage;
                    String orderStr = msg.getBody(String.class);
                    Order order = Transformers.jsonToOrder(orderStr);
                    order.getItemList().forEach(orderItem -> {
                        int old_quantity = catalogService.getCatalogItemById(orderItem.getProductId()).getInventory().getQuantity();
                        int new_quantity = old_quantity - orderItem.getQuantity();
                        if (new_quantity < LOW_THRESHOLD) {
                            log.warning("Inventory for item " + orderItem.getProductId() + " is below threshold (" + LOW_THRESHOLD + "), contact
supplier!");
                        }
                    });
                }


            } catch (JMSException jmse) {
                System.err.println("An exception occurred: " + jmse.getMessage());
            }
        }
    }
}
```

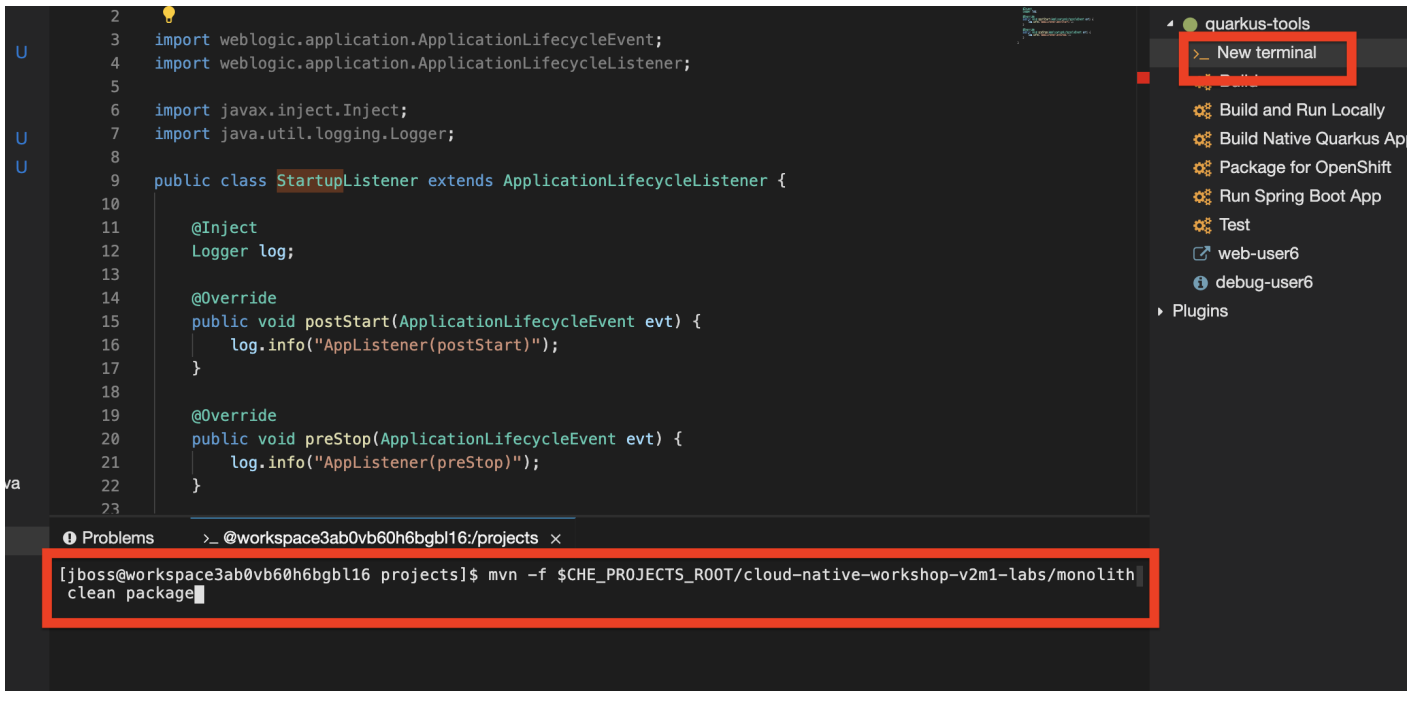
Remember the `<trans-timeout-seconds>` setting from the `weblogic-ejb-jar.xml` file? This is now set as an `@ActivationConfigProperty` in the new code. There are pros and cons to using annotations vs. XML descriptors and care should be taken to consider the needs of the application.

Your MDB should now be properly migrated to JBoss EAP.

10. Test the build

Build once again:

```sh
mvn -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m1-labs/monolith clean package
```

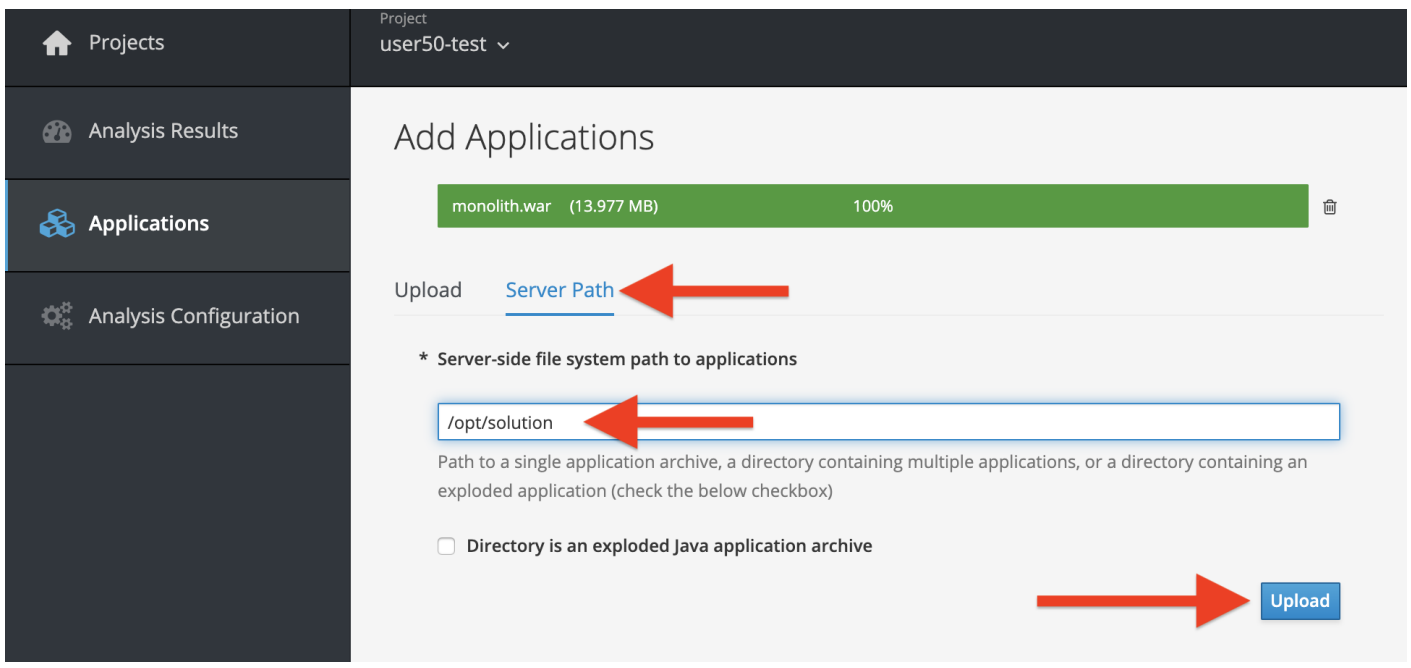


If builds successfully (you will see `BUILD SUCCESS`). If it does not compile, verify you made all the changes correctly and try the build again.

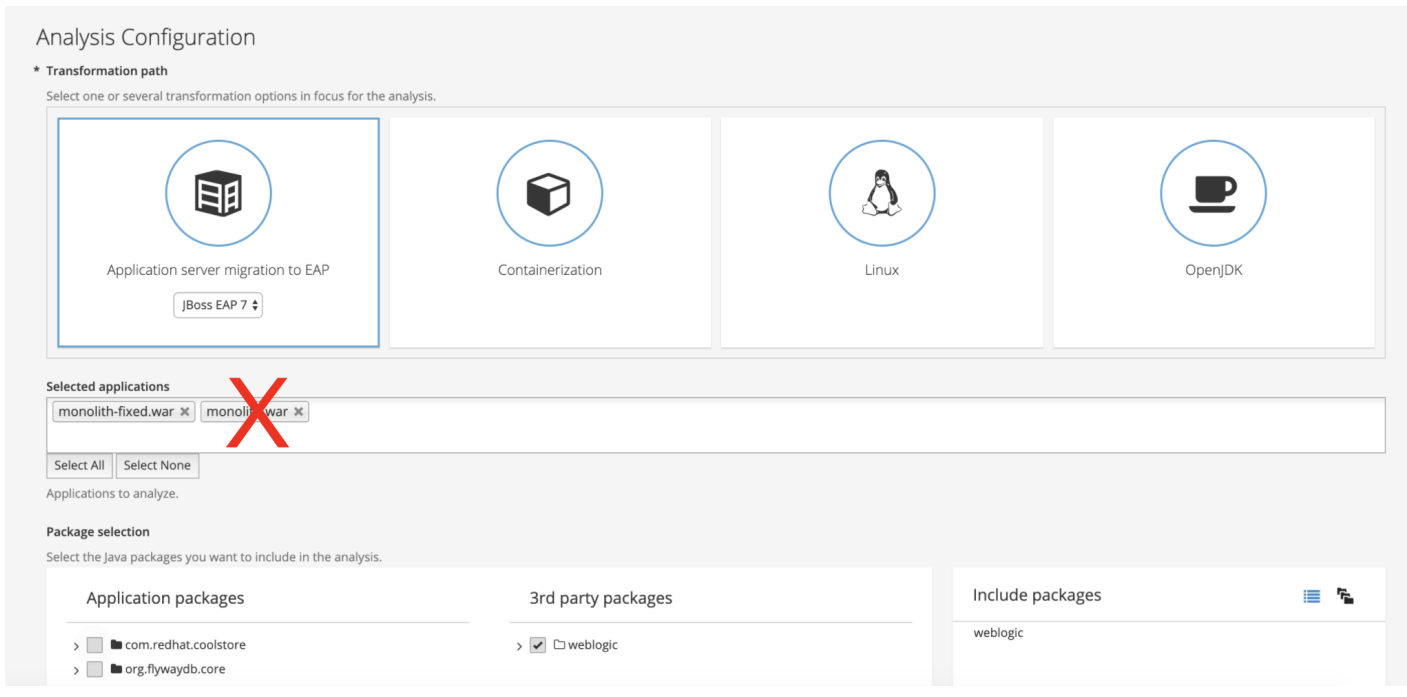
11. Re-run the RHAMT report

In this step we will re-run the RHAMT report to verify our migration was successful.

In the RHAMT Console (http://rhamt-web-console-labs-infra.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com), navigate to `Applications` on the left menu and click on **Add**. Go the **Server Path** tab and enter the path to the fixed project at `/opt/solution` and click **Upload** to add the project:



Be sure to delete the old `monolith.war` to avoid analyzing it again and then click **Save and Run** to analyze the project:

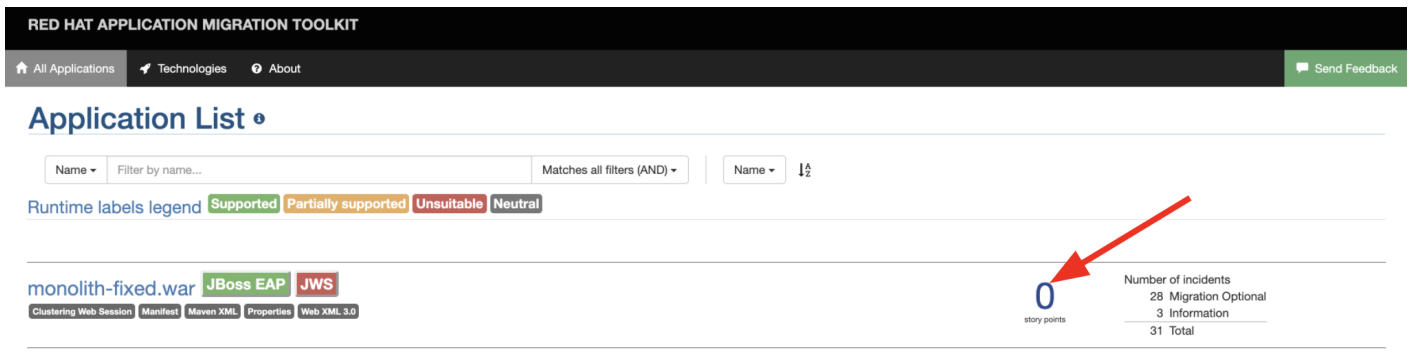


Depending on how many other students are running reports, your analysis might be *queued* for several minutes. If it is taking too long, feel free to skip the next section and proceed to step **13** and return back to the analysis later to confirm that you eliminated all the issues.

12. View the results

Click on the lastet result to go to the report web page and verify that it now reports 0 Story Points:

You have successfully migrated this app to JBoss EAP, congratulations!



Now that we've migrated the app, let's deploy it and test it out and start to explore some of the features that JBoss EAP plus Red Hat OpenShift bring to the table.

13. Add an OpenShift profile

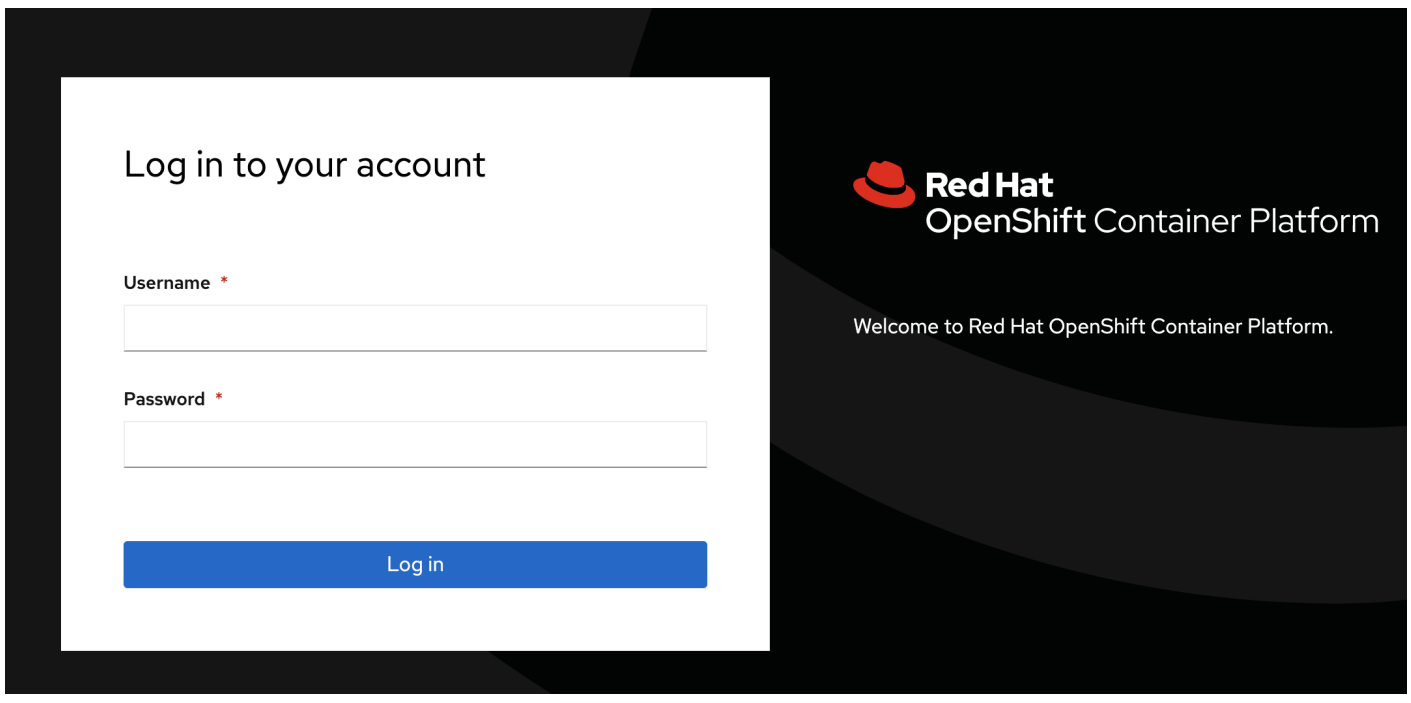
Open the `monolith/pom.xml` file.

At the `<!-- TODO: Add OpenShift profile here -->` we are going to add a the following configuration to the pom.xml

```xml
        <profile>
          <id>openshift</id>
          <build>
              <plugins>
                  <plugin>
                      <artifactId>maven-war-plugin</artifactId>
                      <version>2.6</version>
                      <configuration>
                          <webResources>
                              <resource>
                                  <directory>${basedir}/src/main/webapp/WEB-INF</directory>
                                  <filtering>true</filtering>
                                  <targetPath>WEB-INF</targetPath>
                              </resource>
                          </webResources>
                          <outputDirectory>${basedir}/deployments</outputDirectory>
                          <warName>ROOT</warName>
                      </configuration>
                  </plugin>
              </plugins>
          </build>
        </profile>
```

14. Create the OpenShift project

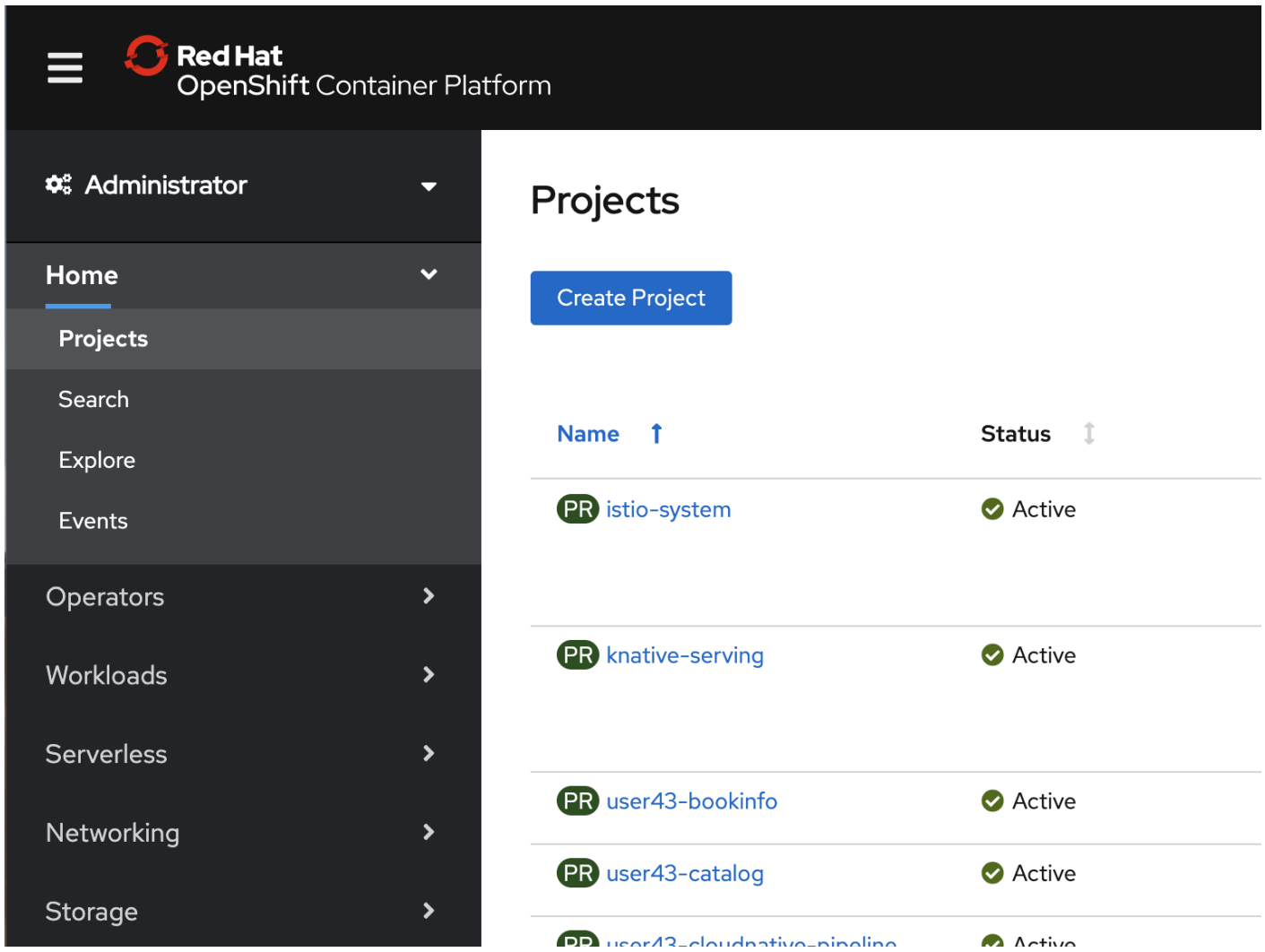
First, open a new brower with the OpenShift web console (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com):

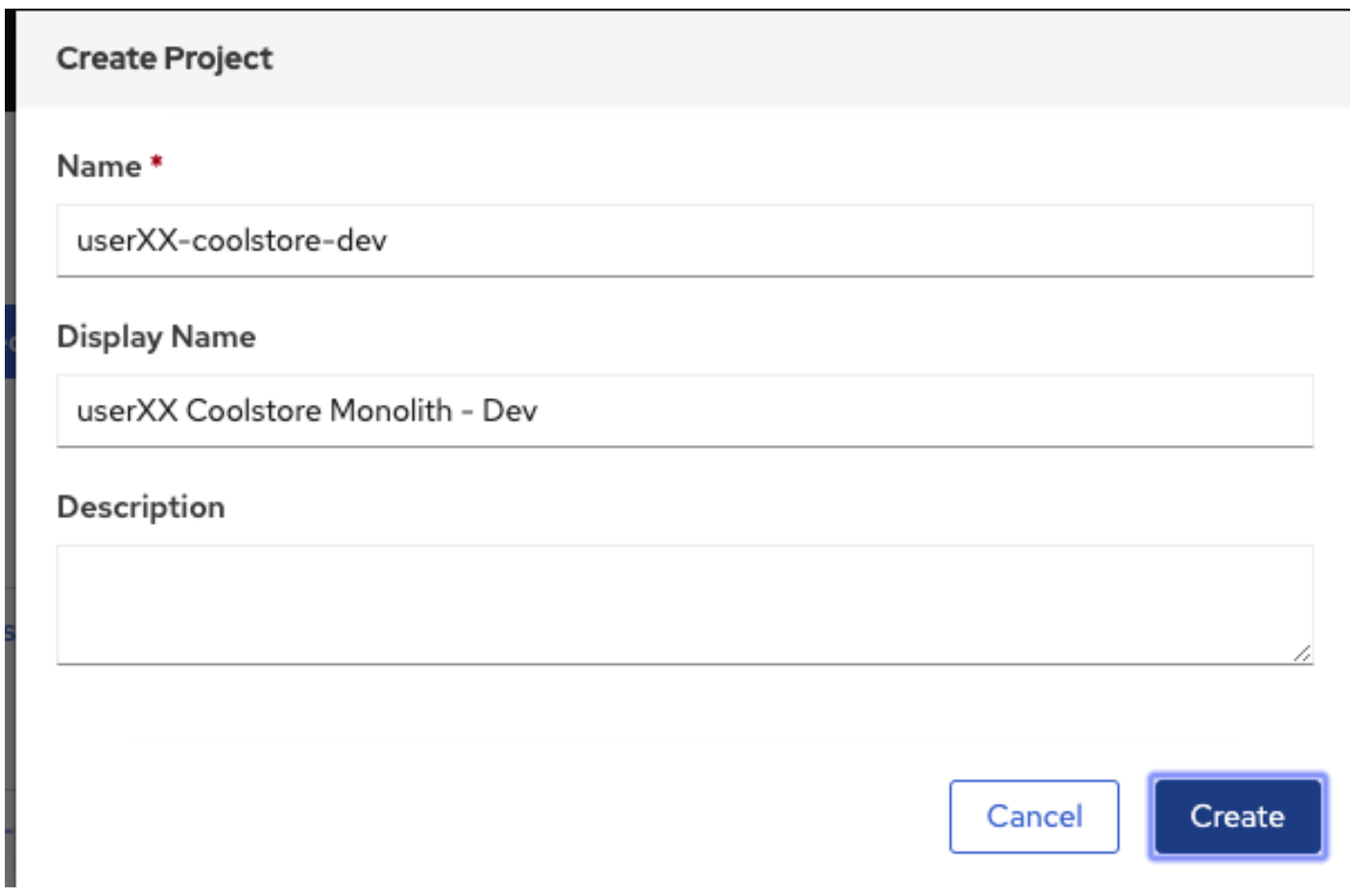


Login using your credentials:

- Username: `user16`
- Password: `r3dh4t1!`

You will see a list of projects to which you have access:





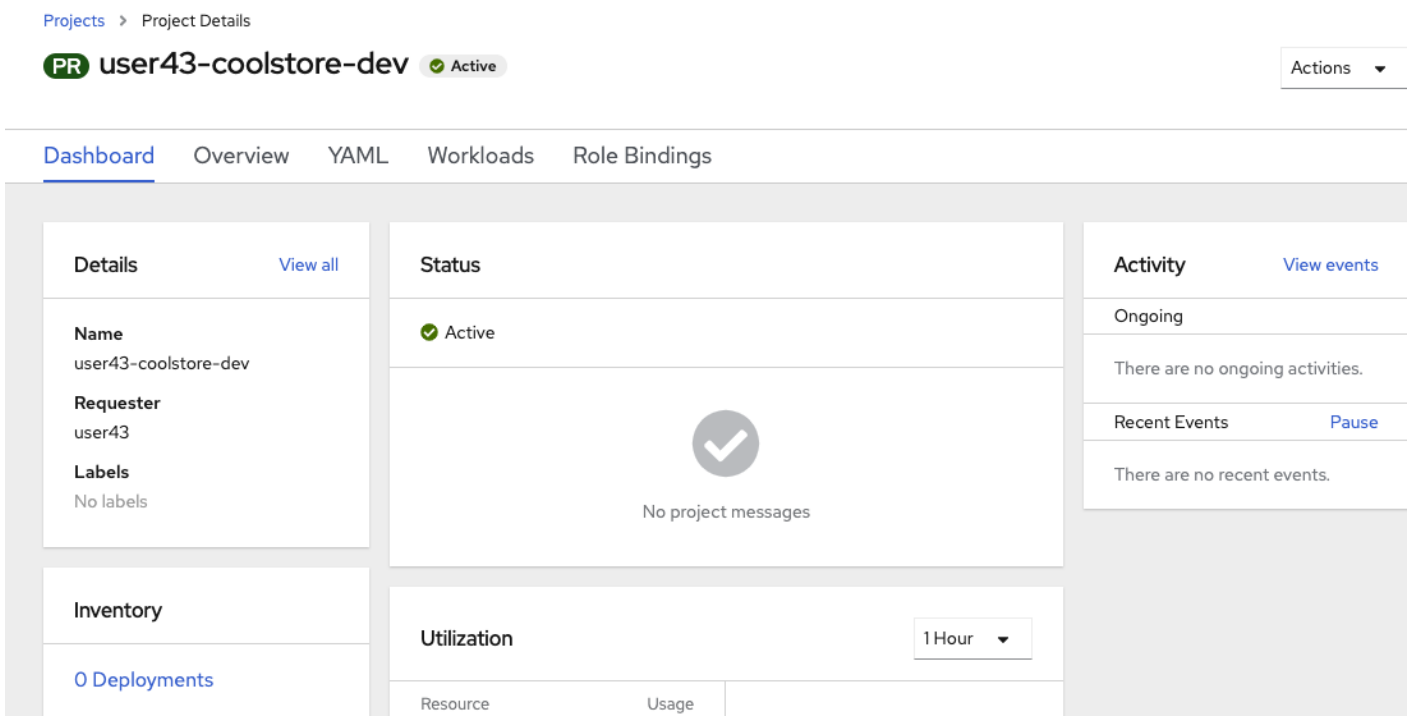
The project displayed in the landing page depends on which labs you will run today.

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

- Name: `user16-coolstore-dev`
- Display Name: `user16 Coolstore Monolith - Dev`
- Description: *leave this field empty*

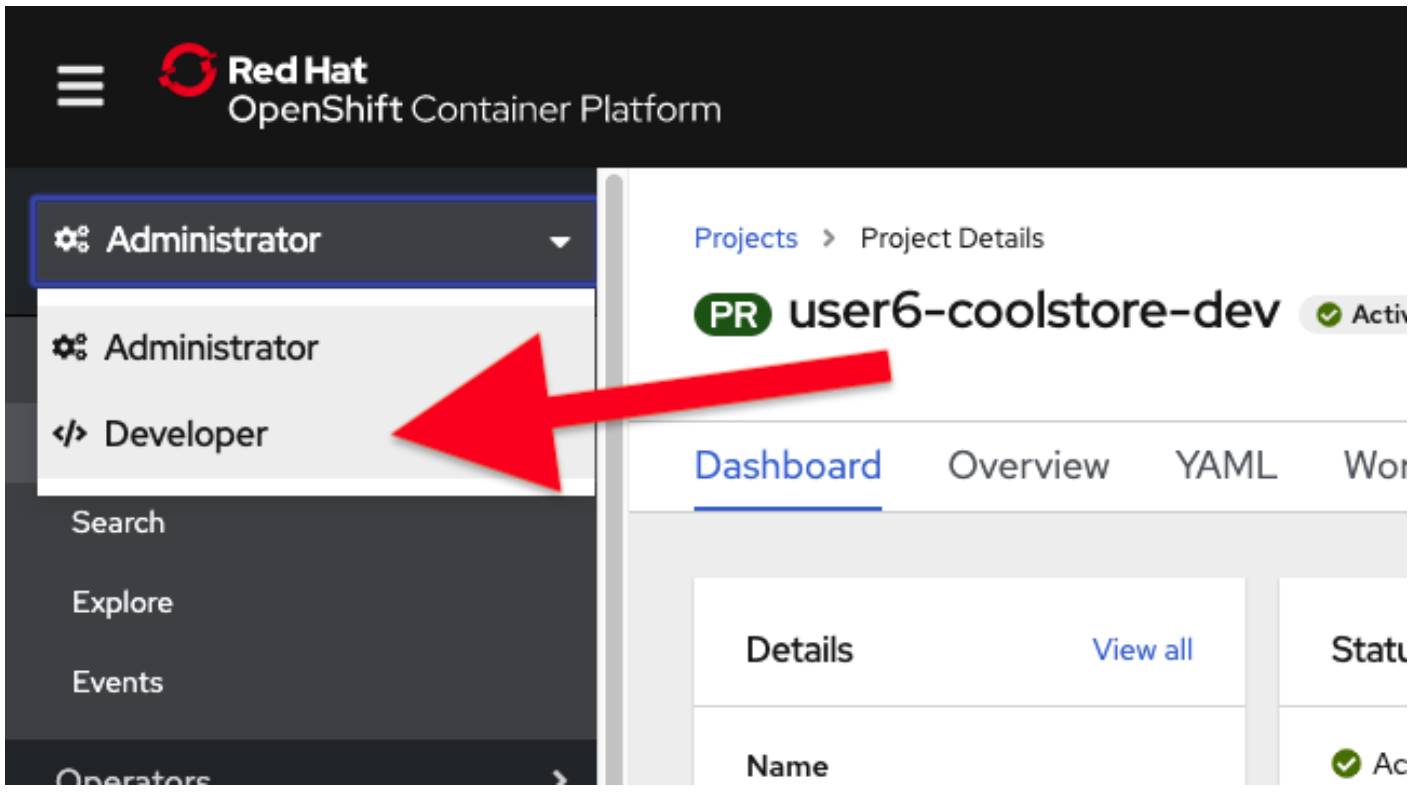
YOU **MUST** USE `user16-coolstore-dev` AS THE PROJECT NAME, as this name is referenced later on and you will experience failures if you do not name it `user16-coolstore-dev` !

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



Switch to Developer Perspective

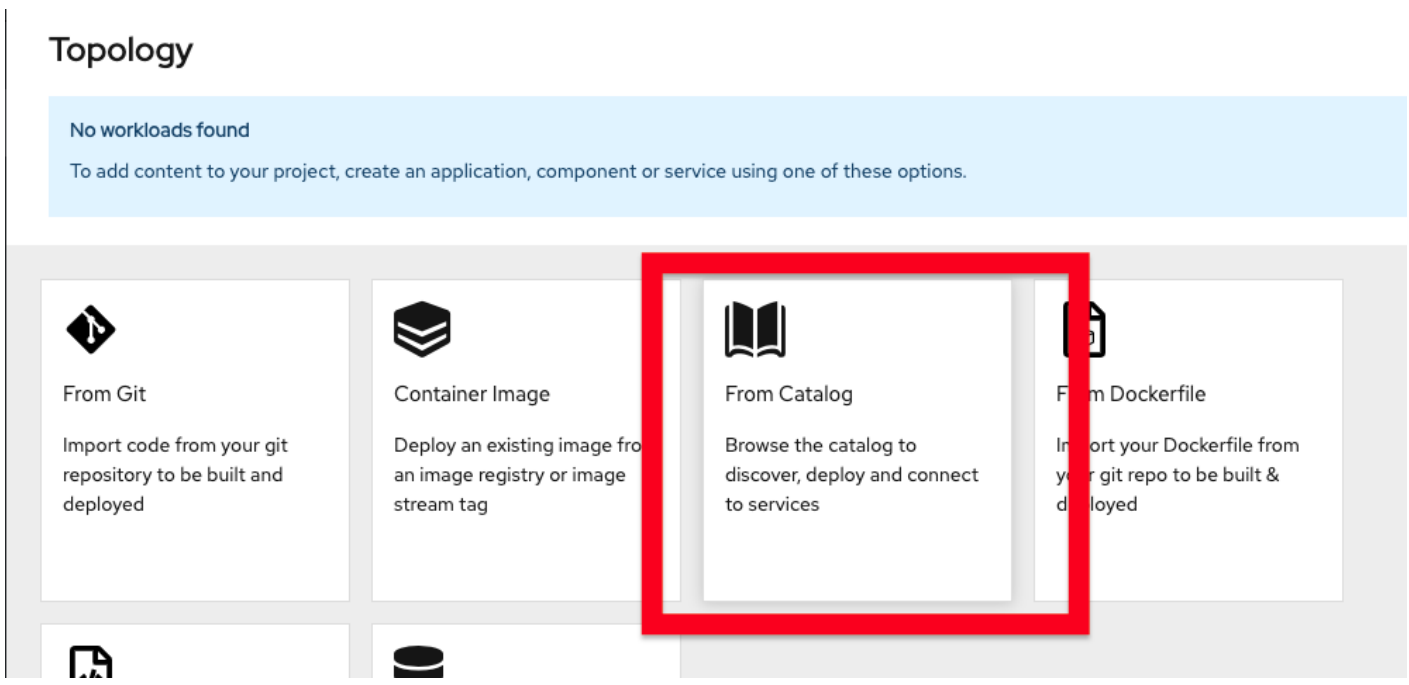
OpenShift 4 provides both an *Administrator* and *Developer* view in its console. Switch to the *Developer Perspective* using the dropdown on the left:



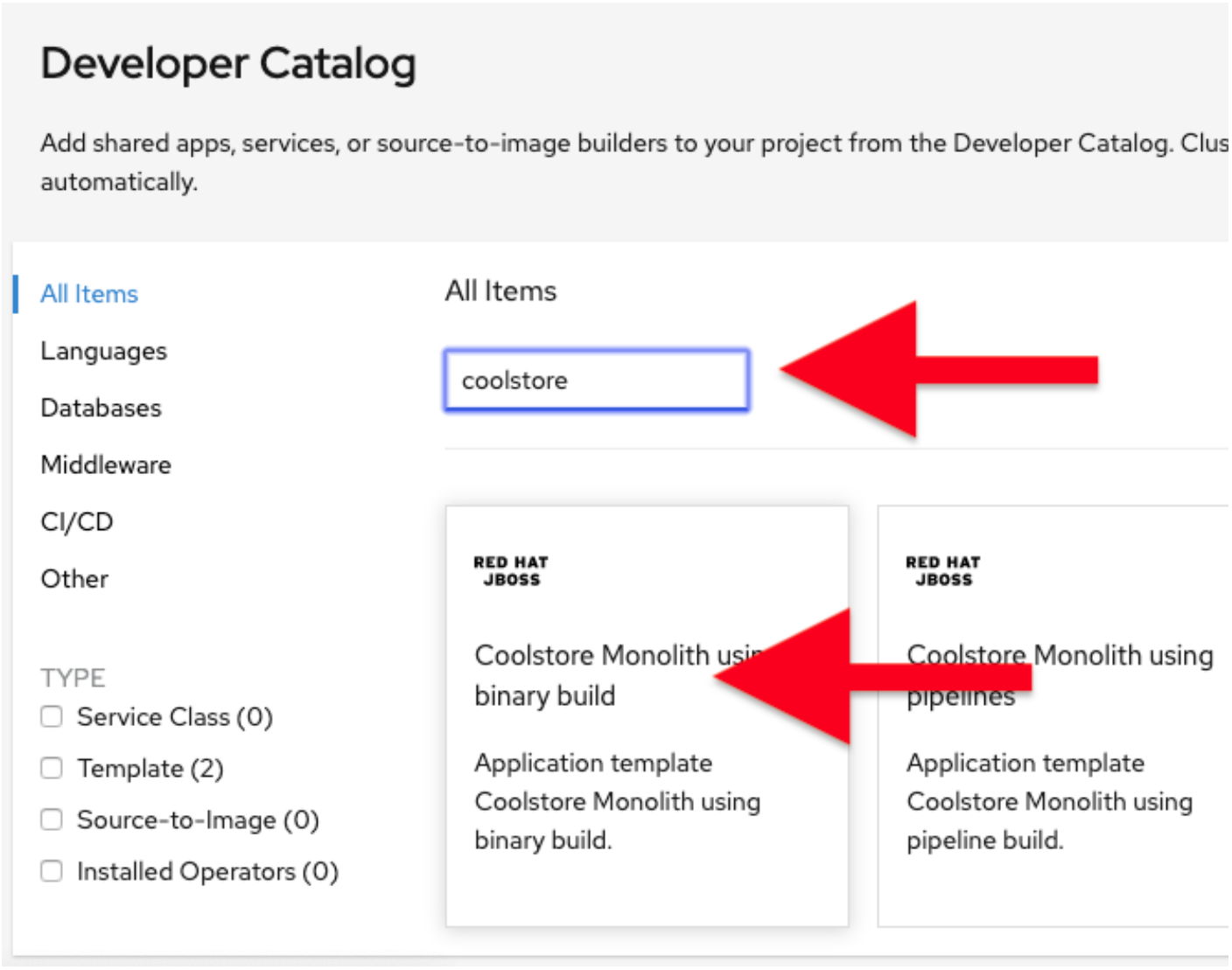
This provides a developer-centric view of applications deployed to the project. Since we have nothing deployed yet, you are presented with a set of ways to deploy applications.

15. Deploy the monolith

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

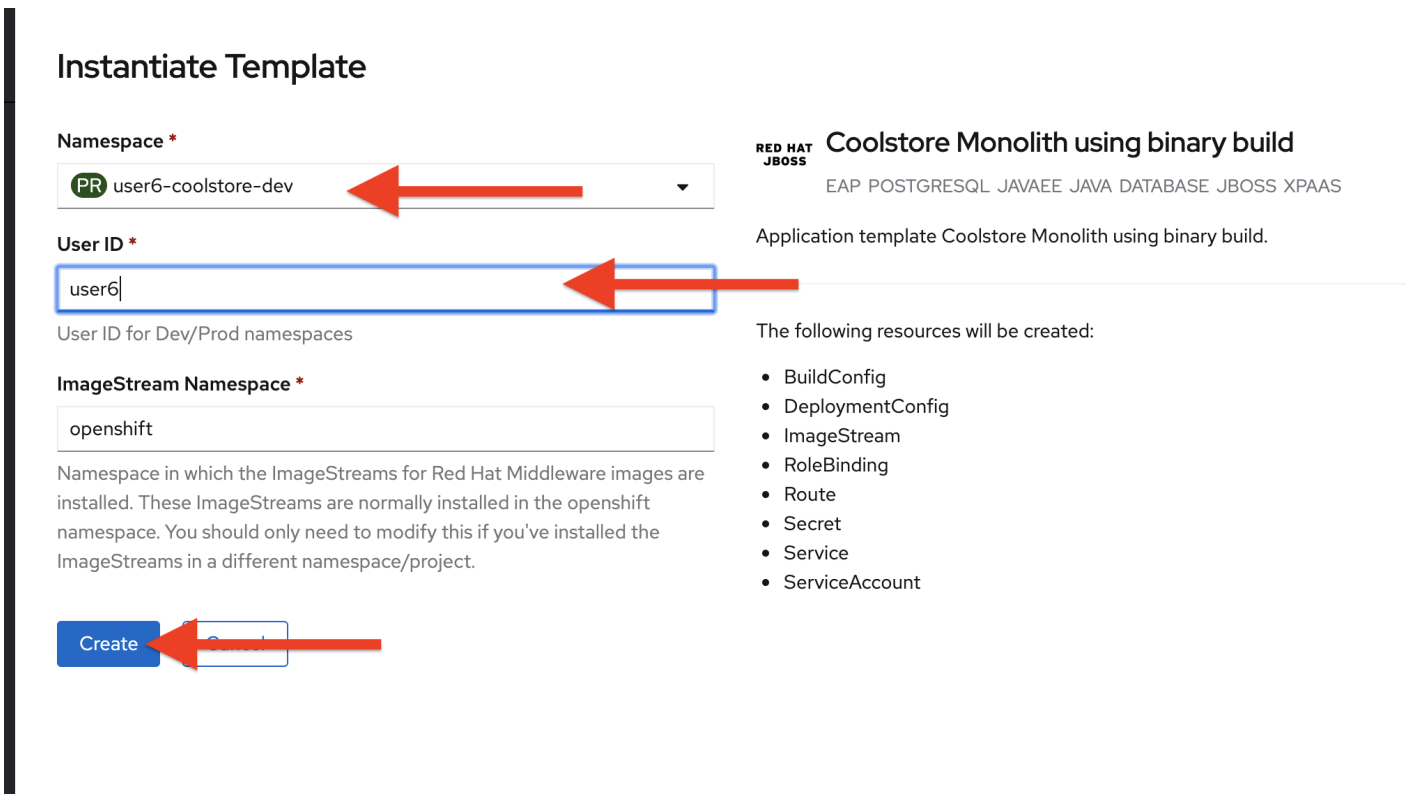


In the search box, type in `coolstore` and choose *Coolstore Monolith using binary build* and then click **Instantiate Template**. If you don't see the *coolstore* template, make sure to uncheck **Operator Backed** in *Type*.



Fill in the following fields:

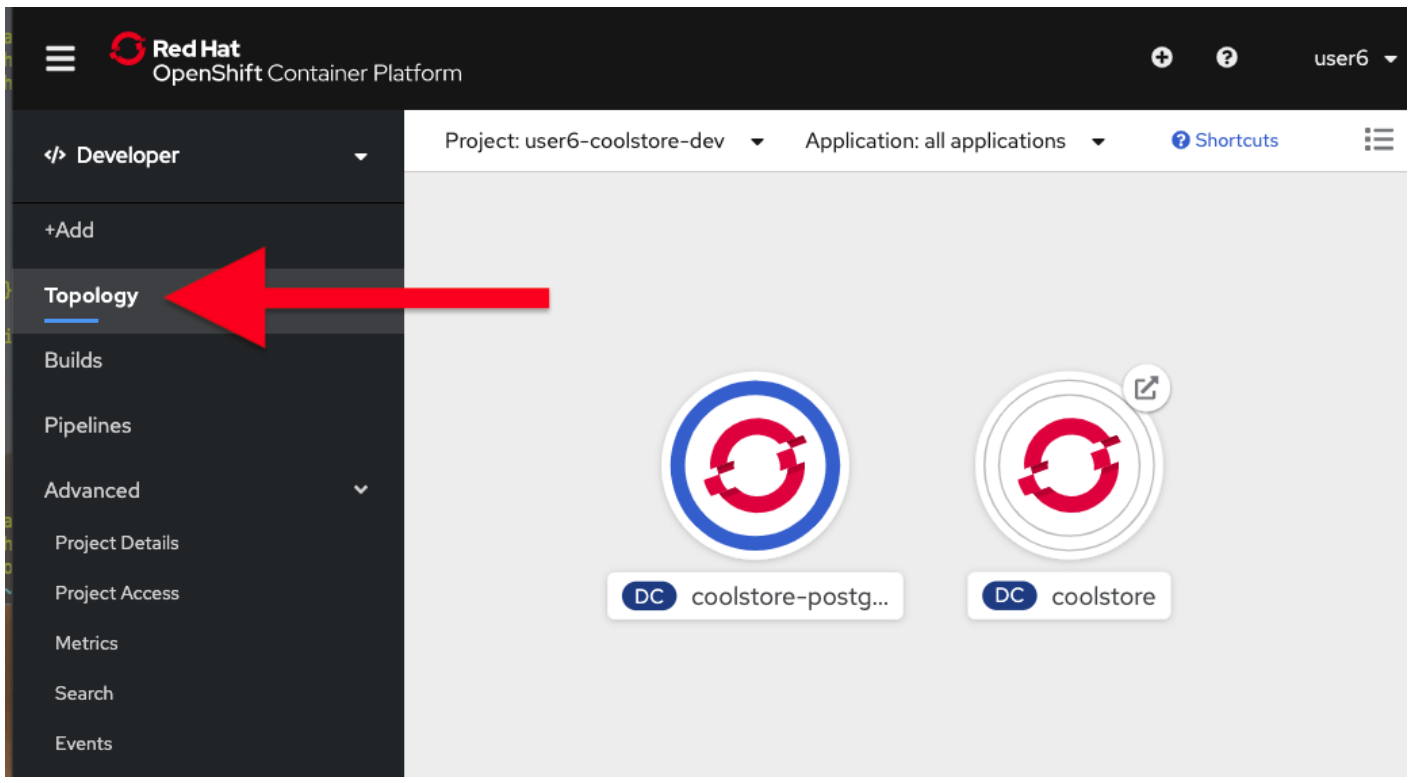
- **Namespace**: `user16-coolstore-dev` (this should already be selected)
- **User ID**: `user16`



Leave other values the same and click **Create**.

Go to the *Topology* view 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.

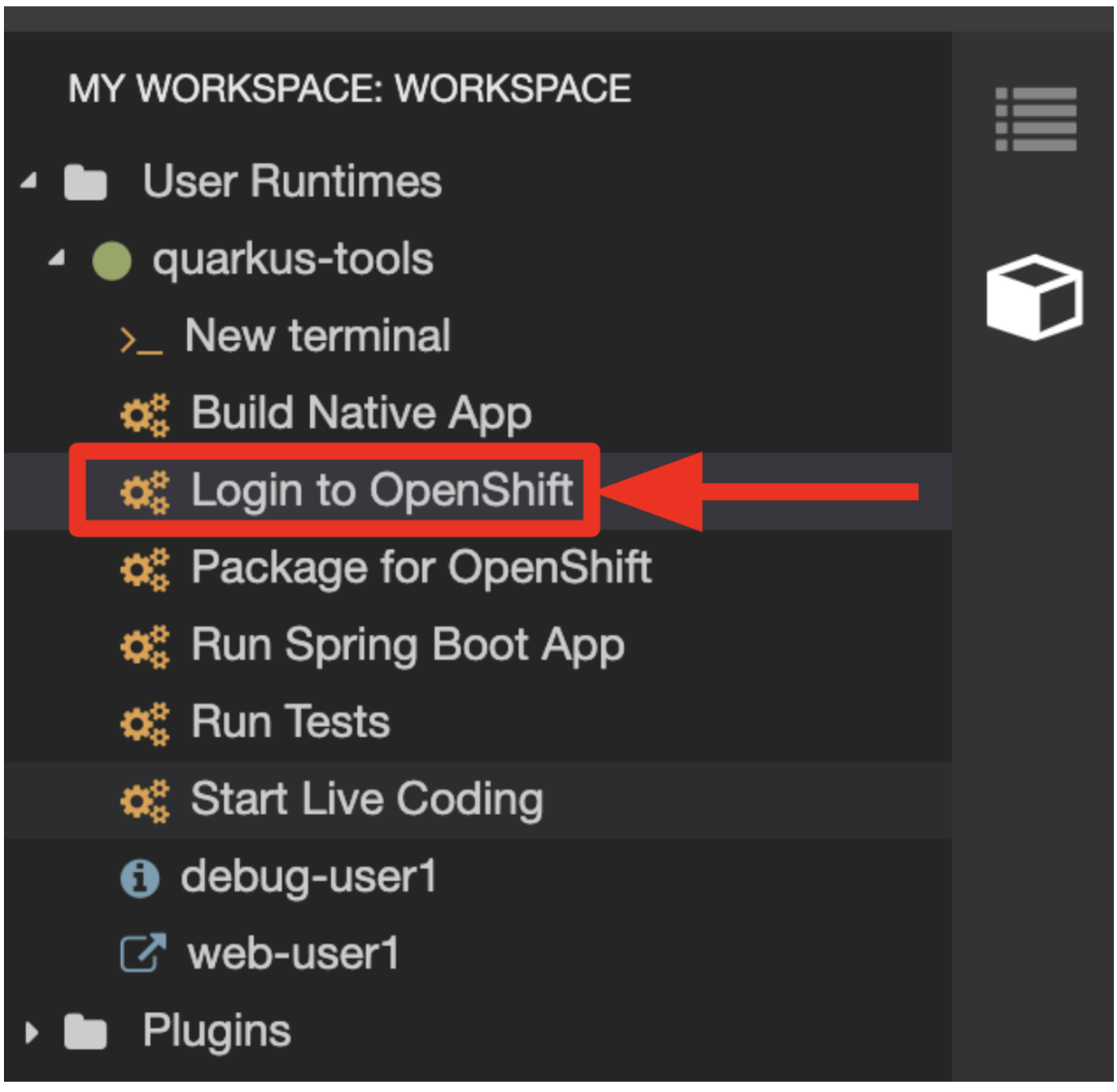


You can see the *postgres* database running (with a dark blue circle), and the coolstore monolith, which has not yet been deployed or started.

Deploy monolith using 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: 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):

```
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.
```

NONE

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!

Switch to the developer project you created earlier via CodeReady Workspaces Terminal window:

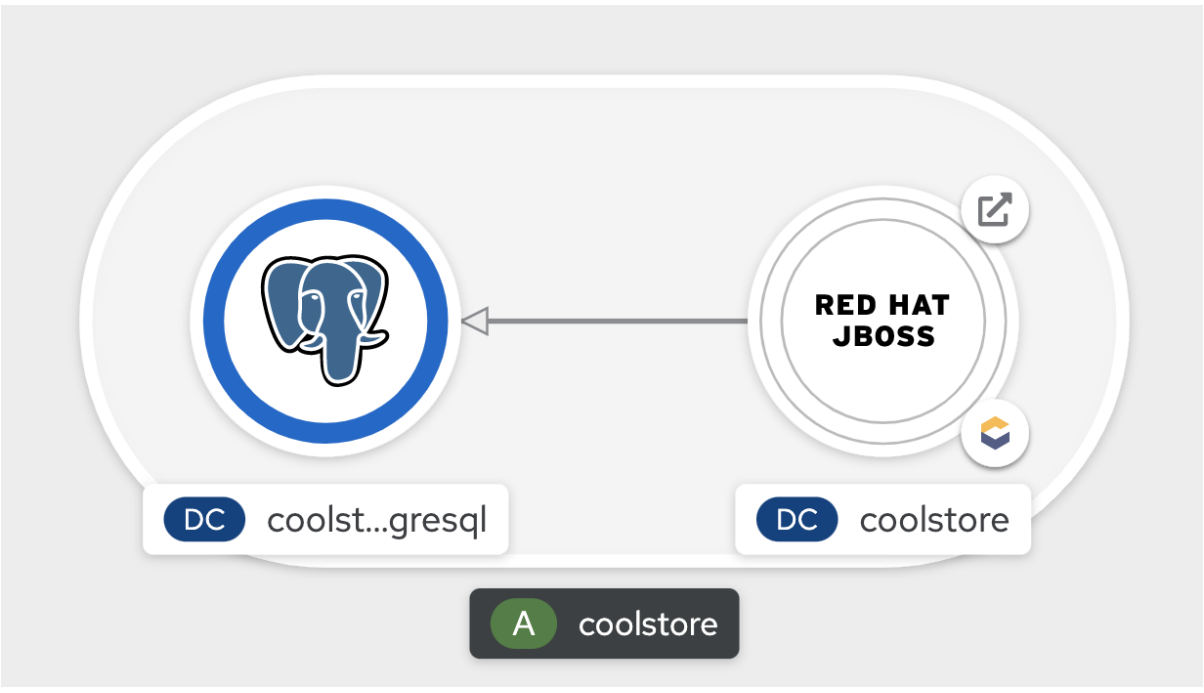
```
oc project user16-coolstore-dev
```

SH

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

```
oc label dc/coolstore-postgresql app.openshift.io/runtime=postgresql --overwrite && \
oc label dc/coolstore app.openshift.io/runtime=jboss --overwrite && \
oc label dc/coolstore-postgresql app.kubernetes.io/part-of=coolstore --overwrite && \
oc label dc/coolstore app.kubernetes.io/part-of=coolstore --overwrite && \
oc annotate dc/coolstore app.openshift.io/connects-to=coolstore-postgresql --overwrite && \
oc annotate dc/coolstore app.openshift.io/vcs-uri=https://github.com/RedHat-Middleware-Workshops/cloud-native-workshop-v2m1-labs.git --overwrite
&& \
oc annotate dc/coolstore app.openshift.io/vcs-ref=ocp-4.4 --overwrite
```

SH



You have not yet deployed the container image built in previous steps, but you'll do that next.

16. Deploy application using Binary build

In this development project we have selected to use a process called *binary builds*, which means that instead of pointing to a public Git Repository and have the S2I (Source-to-Image) build process download, build, and then create a container image for us we are going to build locally and just upload the artifact (e.g. the `.war` file). The binary deployment will speed up the build process significantly.

First, build the project once more using the `openshift` Maven profile, which will create a suitable binary for use with OpenShift (this is not a container image yet, but just the `.war` file). We will do this with the `oc` command line.

Build the project via CodeReady Workspaces Terminal window:

```
mvn clean package -Popenshift -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m1-labs/monolith
```

SH

Wait for the build to finish and the `BUILD SUCCESS` message!

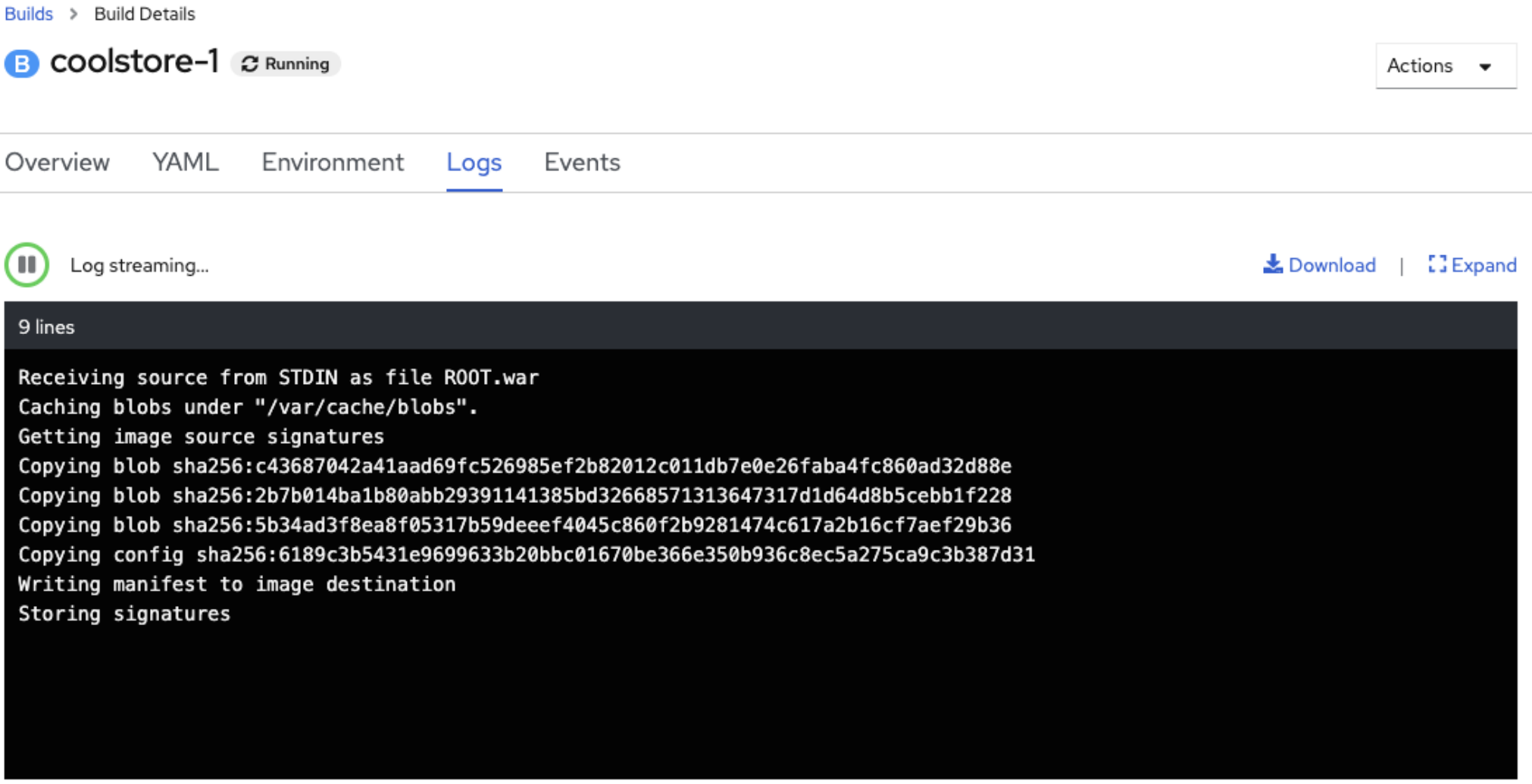
And finally, start the build process that will take the `.war` file and combine it with JBoss EAP and produce a Linux container image which will be automatically deployed into the project, thanks to the *DeploymentConfig* object created from the template:

```
oc start-build coolstore --from-file $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m1-labs/monolith/deployments/ROOT.war
```

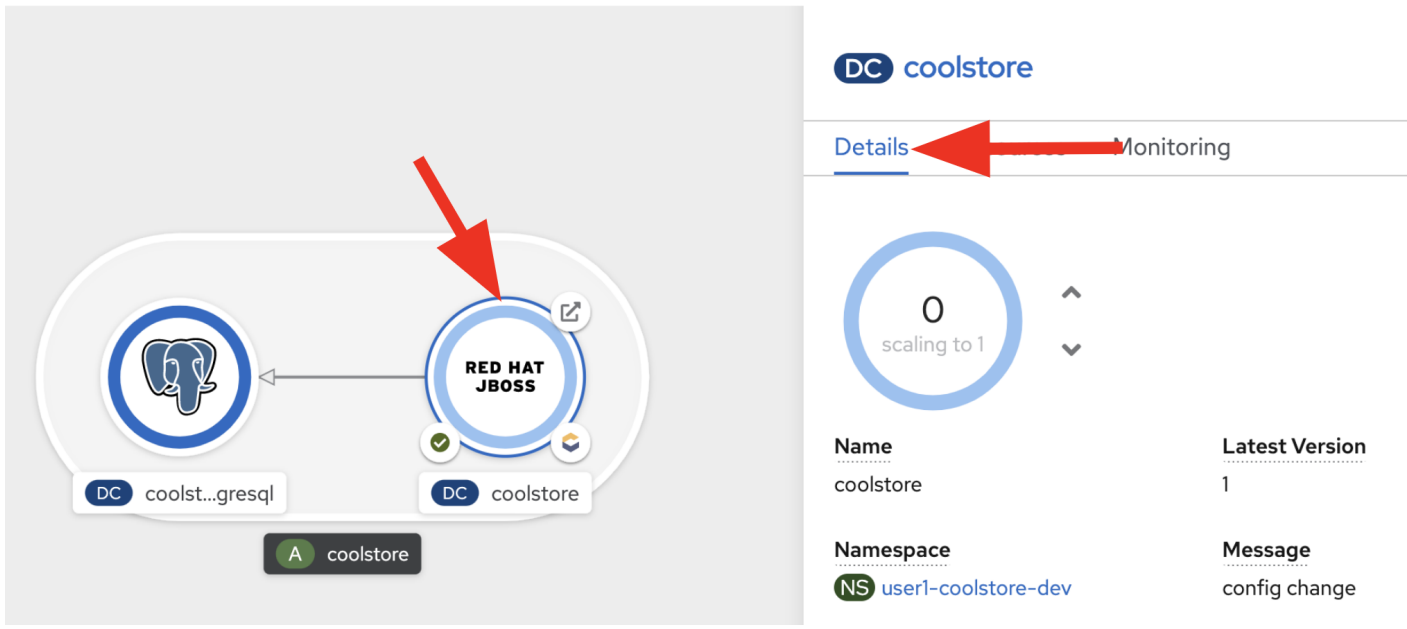
SH

Back in the topology view, you should see the monolith being built:

Click on the building icon to see the build log:



Return to the Topology view, and click on the main icon and view the *Overview*:

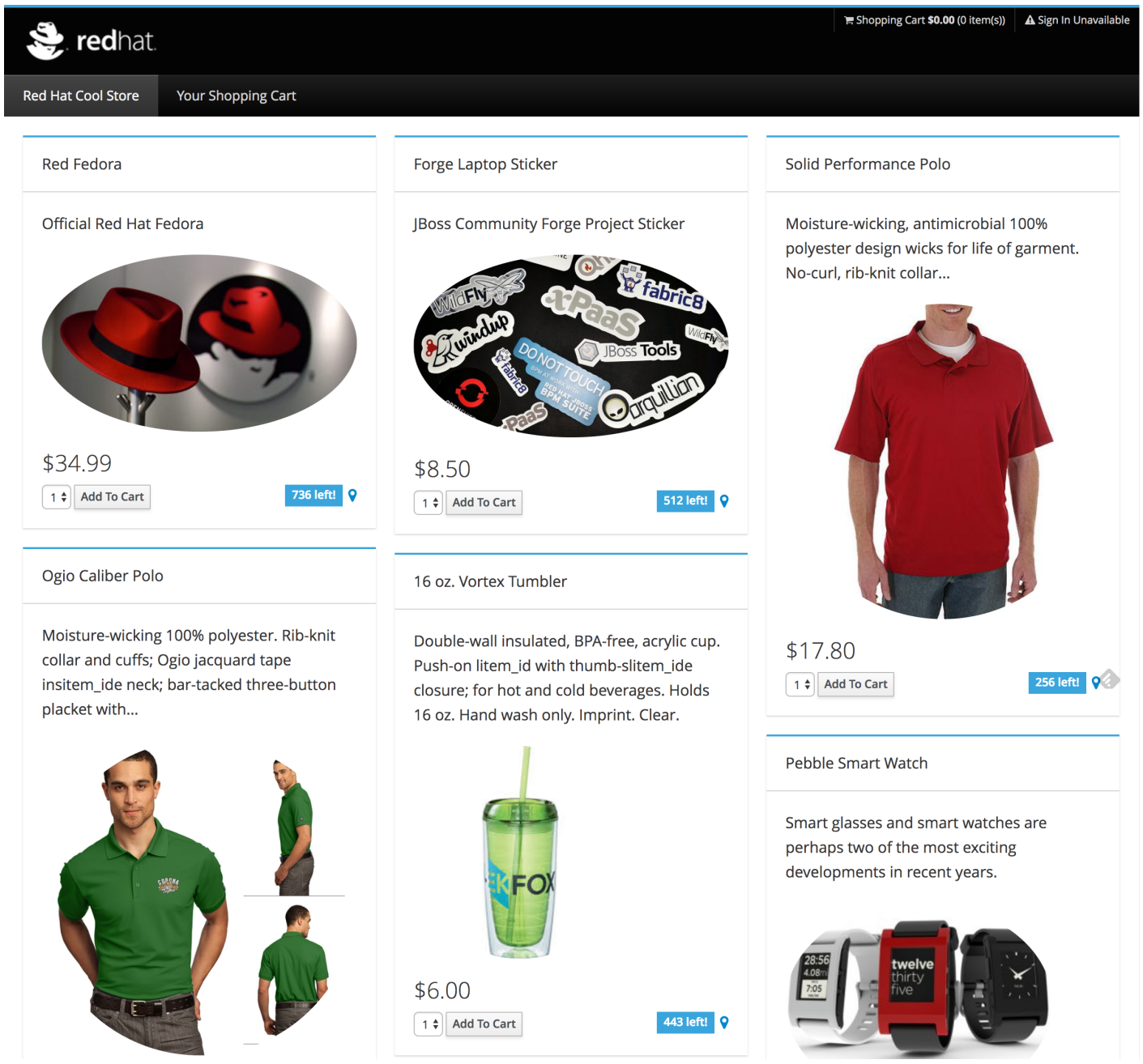


Wait for the deployment to complete and the dark blue circle:

Test the application by clicking on the Route link:

Congratulations!

Now you are using the same application that we built locally on OpenShift. That wasn't too hard right?



Summary

Now that you have migrated an existing Java EE app to the cloud with JBoss and OpenShift, you are ready to start modernizing the application by breaking the monolith into smaller microservices in incremental steps, and employing modern techniques to ensure the application runs well in a distributed and containerized environment.

Break Monolith Apart - I

In the previous labs you learned how to take an existing monolithic Java EE application to the cloud with JBoss EAP and OpenShift, and you got a glimpse into the power of OpenShift for existing applications.

You will now begin the process of modernizing the application by breaking the application into multiple microservices using different technologies, with the eventual goal of re-architecting the entire application as a set of distributed microservices. Later on we'll explore how you can better manage and monitor the application after it is re-architected.

In this lab you will learn more about *Supersonic, Subatomic Java* with Quarkus (https://quarkus.io/), which is designed to be container and developer friendly.

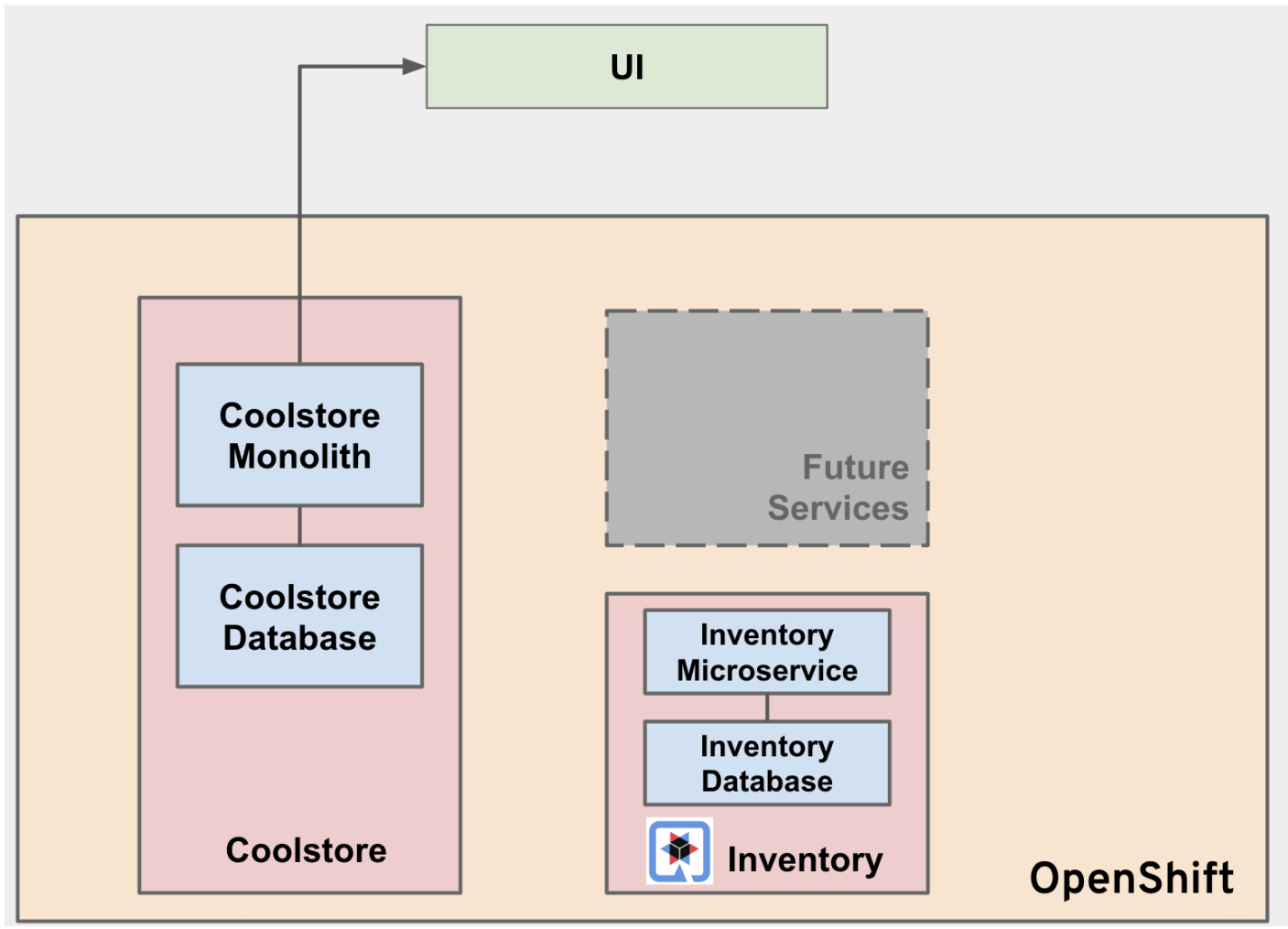
Quarkus is a *Kubernetes Native* Java stack, crafted from the best of breed Java libraries and standards. Amazingly fast boot time, incredibly low RSS memory (not just heap size!) offering near instant scale up and high density memory utilization in container orchestration platforms like Kubernetes. Quarkus uses a technique called compile time boot.

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).

Goals of this lab

You will implement one component of the monolith as a Quarkus microservice and modify it to address microservice concerns, understand its structure, deploy it to OpenShift and exercise the interfaces between Quarkus apps, microservices, and OpenShift/Kubernetes.

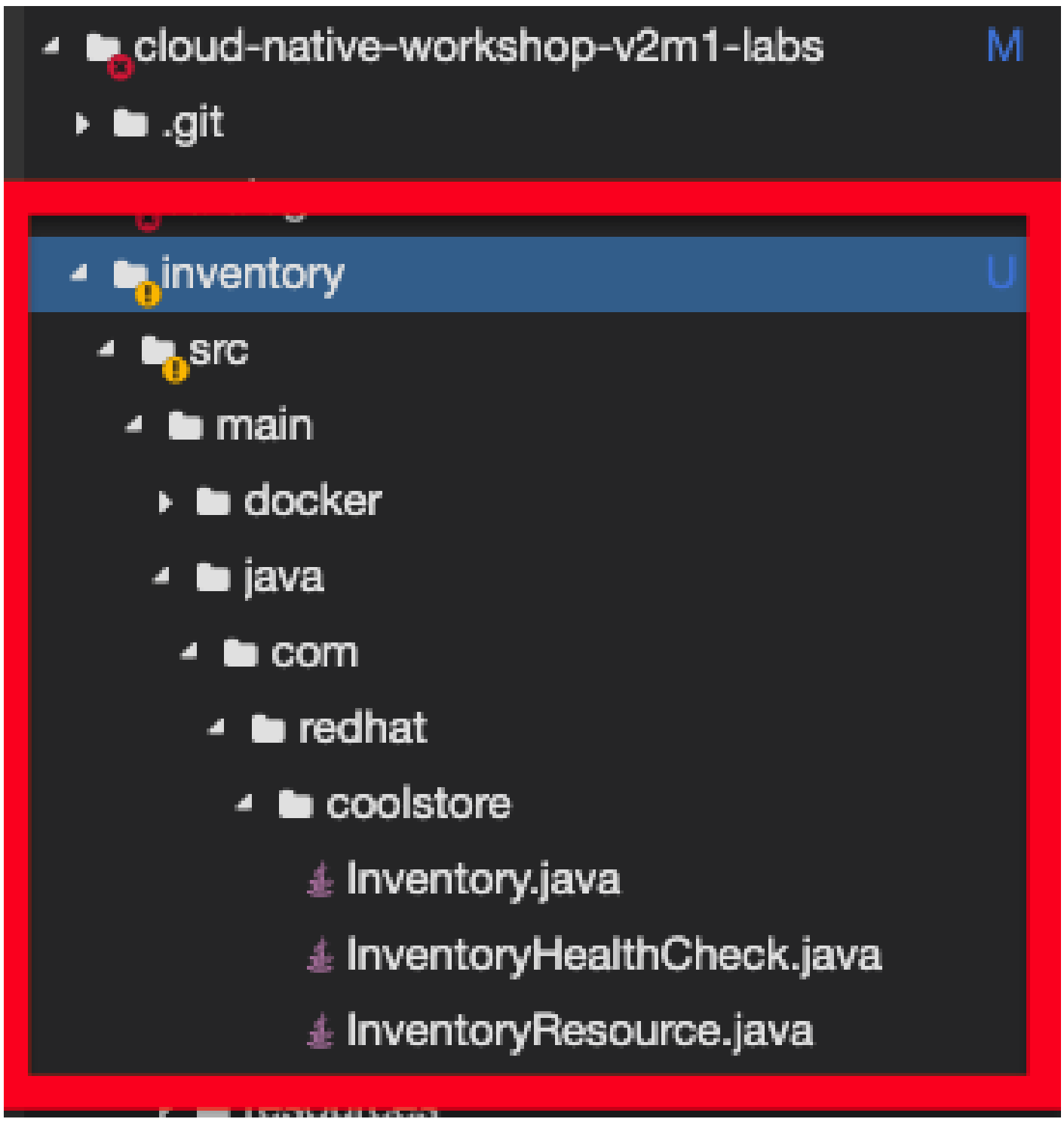
The goal is to deploy this new microservice alongside the existing monolith, and then later on we'll tie them together. But after this lab, you should end up with something like:



1. Setup an Inventory proejct

Run the following commands to set up your environment for this lab and start in the right directory:

In the project explorer, expand the **inventory** project.



2. Examine the Maven project structure

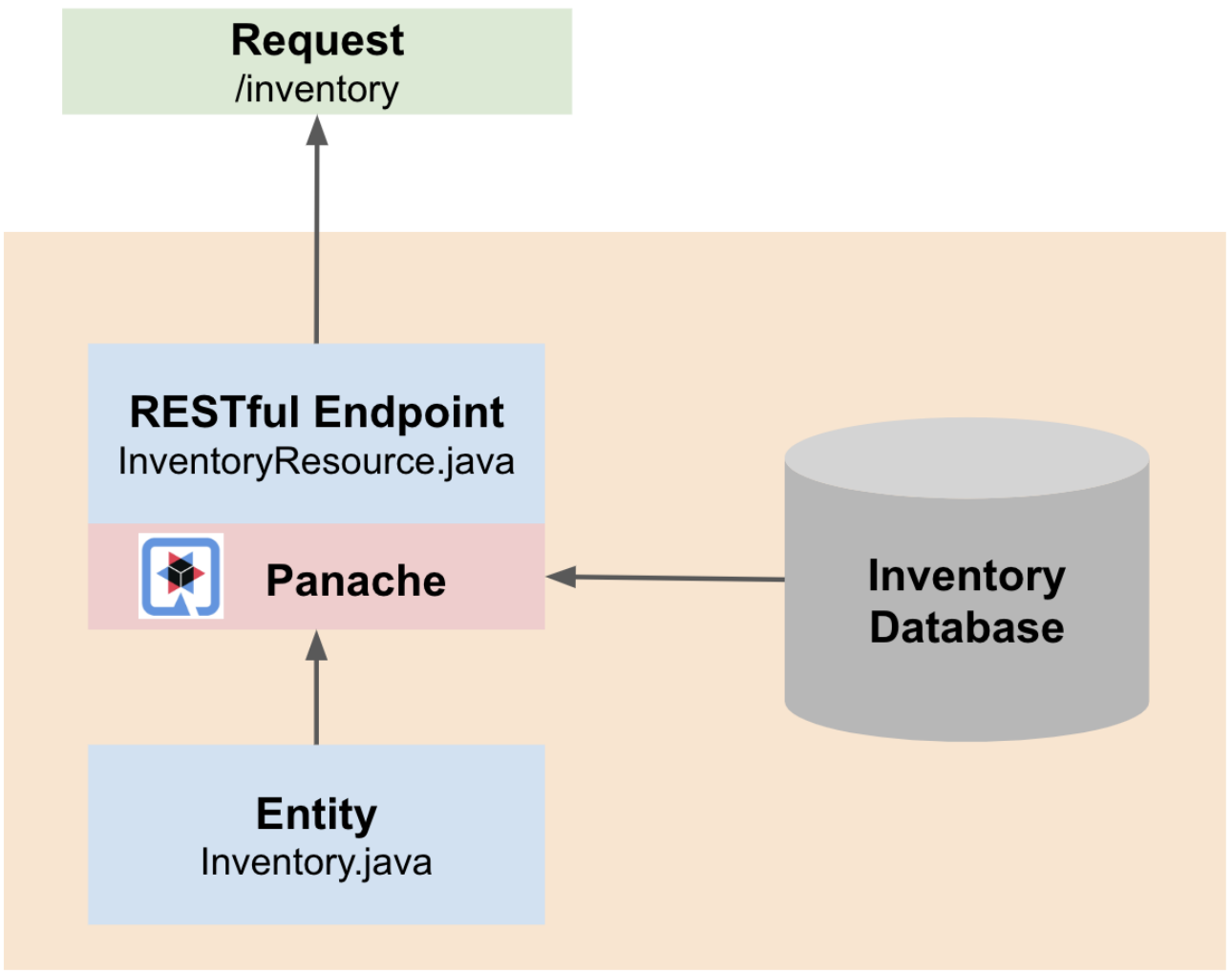
The sample Quarkus project shows a minimal CRUD service exposing a couple of endpoints over REST, with a front-end based on Angular so you can play with it from your browser.

While the code is surprisingly simple, under the hood this is using:

- RESTEasy to expose the REST endpoints
- Hibernate ORM with Panache to perform CRUD operations on the database
- A PostgreSQL database; see below to run one via Linux Container
- Some example `Dockerfile`s to generate new images for JVM and Native mode compilation

`Hibernate ORM` is the de facto JPA implementation and offers you the full breadth of an Object Relational Mapper. It makes complex mappings possible, but it does not make simple and common mappings trivial. Hibernate ORM with Panache focuses on making your entities trivial and fun to write in Quarkus.

Now let's write some code and create a domain model, service interface and a RESTful endpoint to access inventory:



3. Add Quarkus Extensions

We will add Quarkus extensions to the Inventory application for using *Panache* (a simplified way to access data via Hibernate ORM), a database with Postgres (in production) and *H2* (in-memory database for testing). We'll also add the ability to add health probes (which we'll use later on) using the MicroProfile Health extension. Run the following commands to add the extensions using CodeReady Terminal:

```sh
mvn quarkus:add-extension -Dextensions="hibernate-orm-panache, jdbc-h2, health" -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m1-labs/inventory
```

you will see:

✅ Adding extension io.quarkus:quarkus-hibernate-orm-panache ✅ Adding extension io.quarkus:quarkus-jdbc-h2 ✅ Adding extension io.quarkus:quarkus-smallrye-health

And this adds the extensions to `pom.xml`.

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 (e.g. `@Autowired`), and more.

4. Create Inventory Entity

With our skeleton project in place, let's get to work defining the business logic.

The first step is to define the model (entity) of an Inventory object. Since Quarkus uses Hibernate ORM Panache, we can re-use the same model definition from our monolithic application - no need to re-write or re-architect!

Under the `inventory` directory, open up the empty **Inventory.java** file in *com.redhat.coolstore* package and paste the following code into it (identical to the monolith code):

```java
package com.redhat.coolstore;

import javax.persistence.Cacheable;
import javax.persistence.Entity;

import io.quarkus.hibernate.orm.panache.PanacheEntity;

@Entity
@Cacheable
public class Inventory extends PanacheEntity {

    public String itemId;
    public String location;
    public int quantity;
    public String link;

    public Inventory() {

    }

}
```

By extending `PanacheEntity` in your entities, you will get an ID field that is auto-generated. If you require a custom ID strategy, you can extend `PanacheEntityBase` instead and handle the ID yourself.

By using Use public fields, there is no need for functionless getters and setters (those that simply get or set the field). You simply refer to fields like Inventory.location without the need to write a Inventory.getLocation() implementation. Panache will auto-generate any getters and setters you do not write, or you can develop your own getters/setters that do more than get/set, which will be called when the field is accessed directly.

The `PanacheEntity` superclass comes with lots of super useful static methods and you can add your own in your derived entity class. Much like traditional object-oriented programming it's natural and recommended to place custom queries as close to the entity as possible, ideally within the entity definition itself. Users can just start using your entity Inventory by typing Inventory, and get completion for all the operations in a single place.

When an entity is annotated with `@Cacheable`, all its field values are cached except for collections and relations to other entities. This means the entity can be loaded quicker without querying the database for frequently-accessed, but rarely-changing data.

5. Define the RESTful endpoint of Inventory

In this step we will mirror the abstraction of a *service* so that we can inject the Inventory *service* into various places (like a RESTful resource endpoint) in the future. This is the same approach that our monolith uses, so we can re-use this idea again. Open up the empty **InventoryResource.java** class in the *com.redhat.coolstore* package.

Add this code to it:

```java
package com.redhat.coolstore;

import java.util.List;
import java.util.stream.Collectors;

import javax.enterprise.context.ApplicationScoped;
import javax.json.Json;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.ext.ExceptionMapper;
import javax.ws.rs.ext.Provider;

import org.jboss.resteasy.annotations.jaxrs.PathParam;

@Path("/services/inventory")
@ApplicationScoped
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class InventoryResource {

    @GET
    public List<Inventory> getAll() {
        return Inventory.listAll();
    }

    @GET
    @Path("/{itemId}")
    public List<Inventory> getAvailability(@PathParam String itemId) {
        return Inventory.<Inventory>streamAll()
        .filter(p -> p.itemId.equals(itemId))
        .collect(Collectors.toList());
    }

    @Provider
    public static class ErrorMapper implements ExceptionMapper<Exception> {

        @Override
        public Response toResponse(Exception exception) {
            int code = 500;
            if (exception instanceof WebApplicationException) {
                code = ((WebApplicationException) exception).getResponse().getStatus();
            }
            return Response.status(code)
                    .entity(Json.createObjectBuilder().add("error", exception.getMessage()).add("code", code).build())
                    .build();
        }

    }
}
```

The above REST services defines two endpoints:

- `/services/inventory` that is accessible via *HTTP GET* which will return all known product Inventory entities as JSON
- `/services/inventory/<itemId>` that is accessible via *HTTP GET* at for example `services/inventory/329199` with the last path parameter being the ID for which we want inventory status.

6. Add inventory data

Let's add inventory data to the database so we can test things out. Open up the `src/main/resources/import.sql` file and copy the following SQL statements to **import.sql**:

```sql
INSERT INTO INVENTORY (id, itemId, link, location, quantity) values (nextval('hibernate_sequence'), '329299', 'http://maps.google.com/?q=Raleigh',
'Raleigh', 736);
INSERT INTO INVENTORY (id, itemId, link, location, quantity) values (nextval('hibernate_sequence'), '329199', 'http://maps.google.com/?q=Boston',
'Boston', 512);
INSERT INTO INVENTORY (id, itemId, link, location, quantity) values (nextval('hibernate_sequence'), '165613', 'http://maps.google.com/?q=Seoul',
'Seoul', 256);
INSERT INTO INVENTORY (id, itemId, link, location, quantity) values (nextval('hibernate_sequence'), '165614', 'http://maps.google.com/?
q=Singapore', 'Singapore', 54);
INSERT INTO INVENTORY (id, itemId, link, location, quantity) values (nextval('hibernate_sequence'), '165954', 'http://maps.google.com/?q=London',
'London', 87);
INSERT INTO INVENTORY (id, itemId, link, location, quantity) values (nextval('hibernate_sequence'), '444434', 'http://maps.google.com/?q=NewYork',
'NewYork', 443);
INSERT INTO INVENTORY (id, itemId, link, location, quantity) values (nextval('hibernate_sequence'), '444435', 'http://maps.google.com/?q=Paris',
'Paris', 600);
INSERT INTO INVENTORY (id, itemId, link, location, quantity) values (nextval('hibernate_sequence'), '444437', 'http://maps.google.com/?q=Tokyo',
'Tokyo', 230);
```

In Development, we will configure to use local in-memory H2 database for local testing. Add these lines to `src/main/resources/application.properties`:

```PROPERTIES
%dev.quarkus.datasource.url=jdbc:h2:file://projects/database.db
%dev.quarkus.datasource.driver=org.h2.Driver
%dev.quarkus.datasource.username=inventory
%dev.quarkus.datasource.password=mysecretpassword
%dev.quarkus.datasource.max-size=8
%dev.quarkus.datasource.min-size=2
%dev.quarkus.hibernate-orm.database.generation=drop-and-create
%dev.quarkus.hibernate-orm.log.sql=false
```

7. Run Quarkus Inventory application

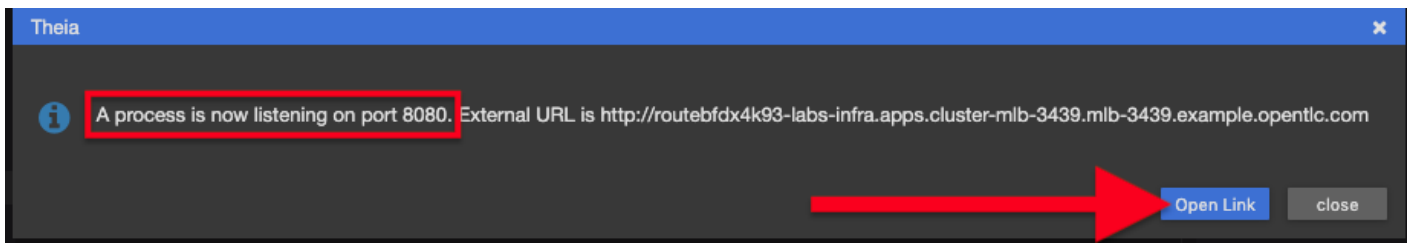
In the Terminal, run the project in *Live Coding* mode:

```SH
mvn clean compile quarkus:dev -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m1-labs/inventory
```

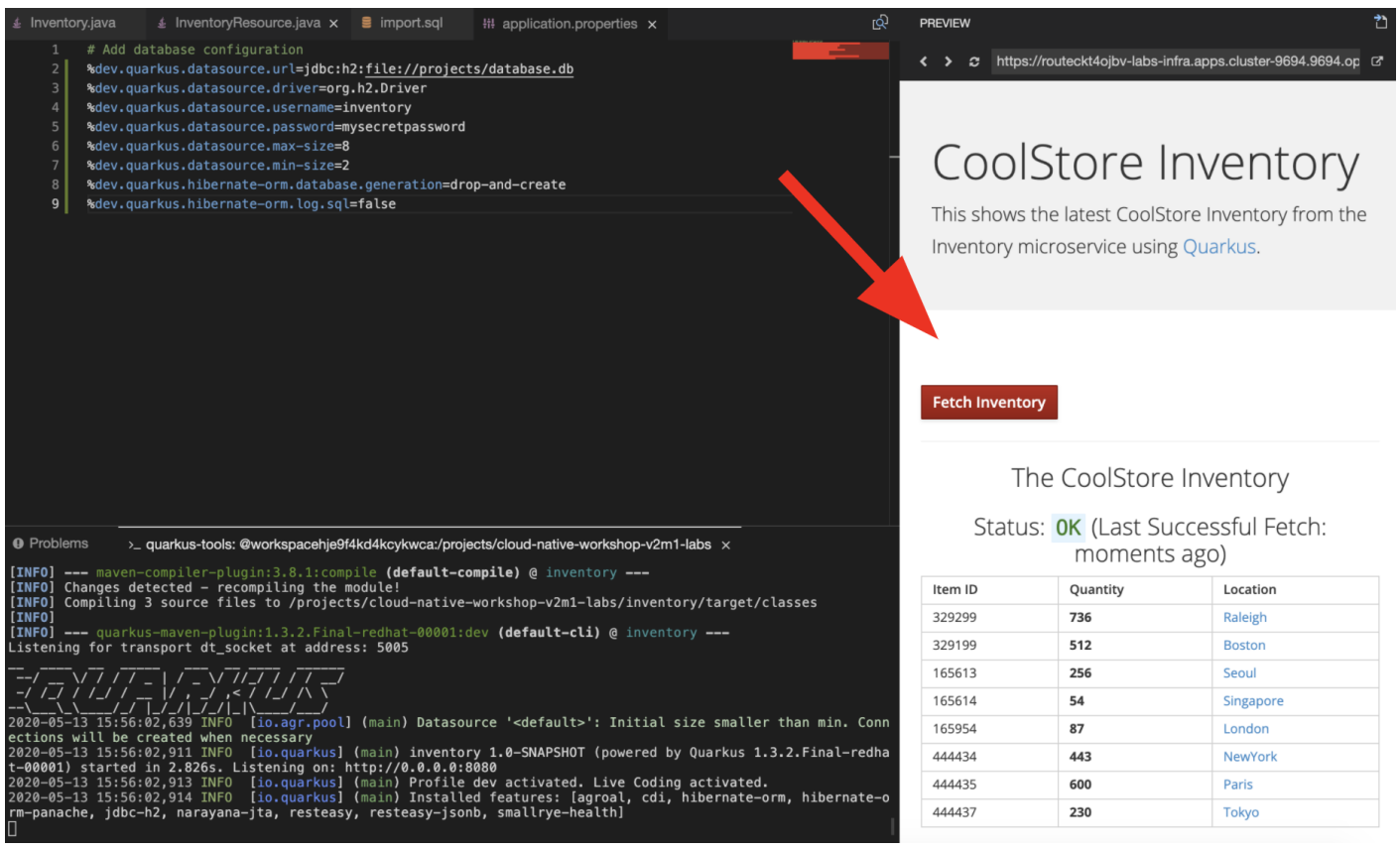
You should see a bunch of log output that ends with:

```CONSOLE
2020-03-19 14:41:17,171 INFO  [io.agr.pool] (main) Datasource '<default>': Initial size smaller than min. Connections will be created when
necessary
2020-03-19 14:41:17,454 INFO  [io.quarkus] (main) inventory 1.0-SNAPSHOT (running on Quarkus xx.xx.xx) started in 3.353s. Listening on:
http://0.0.0.0:8080
2020-03-19 14:41:17,457 INFO  [io.quarkus] (main) Profile dev activated. Live Coding activated.
2020-03-19 14:41:17,457 INFO  [io.quarkus] (main) Installed features: [agroal, cdi, hibernate-orm, hibernate-orm-panache, jdbc-h2, narayana-jta,
resteasy, resteasy-jsonb, smallrye-health]
```

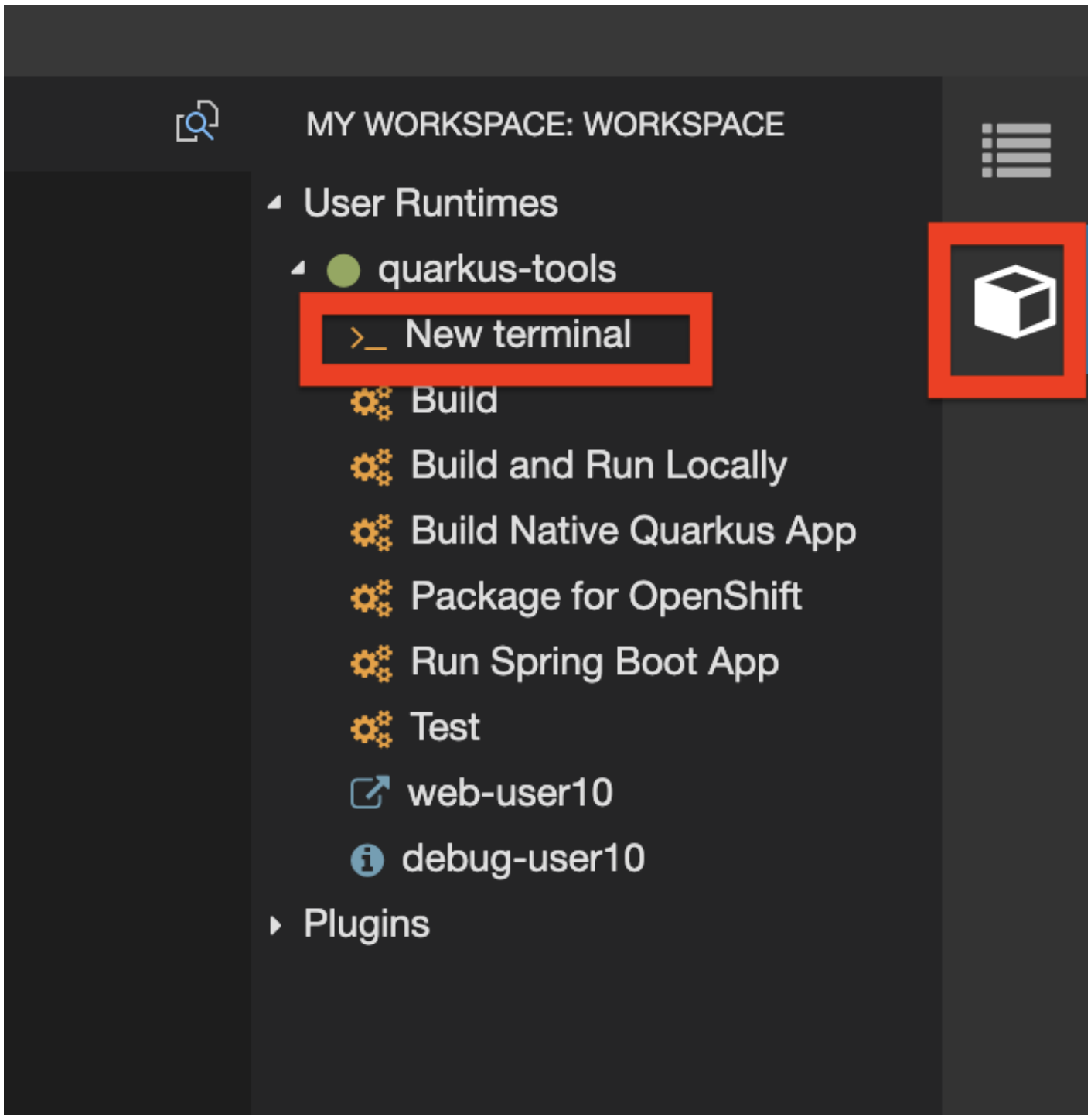
CodeReady will also detect that the Quarkus app opens port `5005` (for debugging) and `8080` (for web requests). Do not open port 5005, but when prompted, open the port `8080`, which opens a small web browser in CodeReady:



You should see the inventory web frontend directly in CodeReady (you may need to click the *reload* icon):



Open a **new** CodeReady Workspaces Terminal:



and invoke the RESTful endpoint using the following CURL commands.

```sh
curl http://localhost:8080/services/inventory | jq
```

The output looks like:

```json
...
{
  "id": 7,
  "itemId": "444435",
  "link": "http://maps.google.com/?q=Paris",
  "location": "Paris",
  "quantity": 600
},
{
  "id": 8,
  "itemId": "444437",
  "link": "http://maps.google.com/?q=Tokyo",
  "location": "Tokyo",
  "quantity": 230
}
```

8. Add health probe

What is MicroProfile Health?

MicroProfile Health allows applications to provide information about their state to external viewers which is typically useful in cloud environments like OpenShift where automated processes must be able to determine whether the application should be discarded or restarted.

Run the health check

When you imported the *health extension* earlier, the `/health` endpoint is automatically exposed directly that can be used to run the health check procedures.

Our application is still running, so you can exercise the default (no-op) health check with this command in a separate Terminal:

```sh
curl -s http://localhost:8080/health | jq
```

The output shows:

```
JSON
{
  "status": "UP",
  "checks": [
    {
      "name": "Database connections health check",
      "status": "UP"
    }
  ]
}
```

The general *outcome* of the health check is computed as a logical AND of all the declared health check procedures. Quarkus extensions can also provide default health checks out of the box, which is why you see the `Database connections health check` above, since we are using a database extension.

9. Create your first health check

Next, let's fill in the class by creating a new RESTful endpoint which will be used by OpenShift to probe our services. Open empty Java class: `src/main/java/com/redhat/coolstore/InventoryHealthCheck.java` and add the following code:

```
JAVA
package com.redhat.coolstore;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;

import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;
import org.eclipse.microprofile.health.Readiness;

@Readiness
@ApplicationScoped
public class InventoryHealthCheck implements HealthCheck {

    @Inject
    private InventoryResource inventoryResource;

    @Override
    public HealthCheckResponse call() {

        if (inventoryResource.getAll() != null) {
            return HealthCheckResponse.named("Success of Inventory Health Check!!!").up().build();
        } else {
            return HealthCheckResponse.named("Failure of Inventory Health Check!!!").down().build();
        }
    }
}
```

The `call()` method exposes an HTTP GET endpoint which will return the status of the service. The logic of this check does a simple query to the underlying database to ensure the connection to it is stable and available. The method is also annotated with MicroProfile's `@Readiness` annotation, which directs Quarkus to expose this endpoint as a health check at `/health/ready`.

You don't need to stop and re-run re-run the Inventory application because Quarkus will **reload the changes automatically** via the *Live Coding* feature.

Access the health endpoint again using *curl* and the result looks like:

```
SH
curl -s http://localhost:8080/health | jq
```

The result should be:

```
JSON
{
  "status": "UP",
  "checks": [
    {
      "name": "Success of Inventory Health Check!!!",
      "status": "UP"
    },
    {
      "name": "Database connections health check",
      "status": "UP"
    }
  ]
}
```

You now see the default health check, along with your new Inventory health check.

You can define separate readiness and liveness probes using `@Liveness` and `@Readiness` annotations and access them separately at `/health/live` and `/health/ready`.

Cleanup

Stop the Quarkus app by typing `CTRL-C` in the Terminal in which the app runs.

10. Create OpenShift Project

In this step, we will deploy our new Inventory microservice for our CoolStore application in a separate project to house it and keep it separate from our monolith and our other microservices we will create later on.

Before going to OpenShift console, we will repackage the Quarkus application for adding a PostgreSQL extension because our Inventory service will connect to PostgeSQL database in production on OpenShift.

Quarkus also offers the ability to automatically generate OpenShift resources based on sane default and user supplied configuration. The OpenShift extension is actually a wrapper extension that brings together the kubernetes (https://quarkus.io/guides/deploying-to-kubernetes) and container-image-s2i (https://quarkus.io/guides/container-image#s2i) extensions with defaults so that it's easier for the user to get started with Quarkus on OpenShift.

Add *quarkus-jdbc-postgresql* and *openshift* extension via CodeReady Workspaces Terminal:

```
mvn quarkus:add-extension -Dextensions="jdbc-postgresql,openshift" -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m1-labs/inventory
```
SH

you will see:

✅ Adding extension io.quarkus:quarkus-openshift ✅ Adding extension io.quarkus:quarkus-jdbc-postgresql

Quarkus supports the notion of *configuration profiles*. These allows you to have multiple configurations in the same file and select between then via a *profile name*.

By default Quarkus has three profiles, although it is possible to use as many as you like. The default profiles are:

- `dev` - Activated when in development mode (i.e. **quarkus:dev**)
- `test` - Activated when running tests
- `prod` - The default profile when not running in development or test mode

Let's `add` the following variables in *src/main/resources/application.properties*:

```
%prod.quarkus.datasource.url=jdbc:postgresql://inventory-database:5432/inventory
%prod.quarkus.datasource.driver=org.postgresql.Driver
%prod.quarkus.datasource.username=inventory
%prod.quarkus.datasource.password=mysecretpassword
%prod.quarkus.datasource.max-size=8
%prod.quarkus.datasource.min-size=2
%prod.quarkus.hibernate-orm.database.generation=drop-and-create
%prod.quarkus.hibernate-orm.sql-load-script=import.sql
%prod.quarkus.hibernate-orm.log.sql=true
%prod.quarkus.s2i.base-jvm-image=registry.access.redhat.com/ubi8/openjdk-11

%prod.quarkus.kubernetes-client.trust-certs=true 1
%prod.quarkus.container-image.build=true 2
%prod.quarkus.kubernetes.deploy=true 3
%prod.quarkus.kubernetes.deployment-target=openshift 4
%prod.quarkus.openshift.expose=true 5
%prod.quarkus.openshift.labels.app.openshift.io/runtime=quarkus 6
```
SHELL

1. We are using self-signed certs in this simple example, so this simply says to the extension to trust them.
2. Instructs the extension to build a container image
3. Instructs the extension to deploy to OpenShift after the container image is built
4. Instructs the extension to generate and create the OpenShift resources (like `DeploymentConfig` and `Service`) after building the container
5. Instructs the extension to generate an OpenShift `Route`.
6. Adds a nice-looking icon to the app when viewing the OpenShift Developer Toplogy

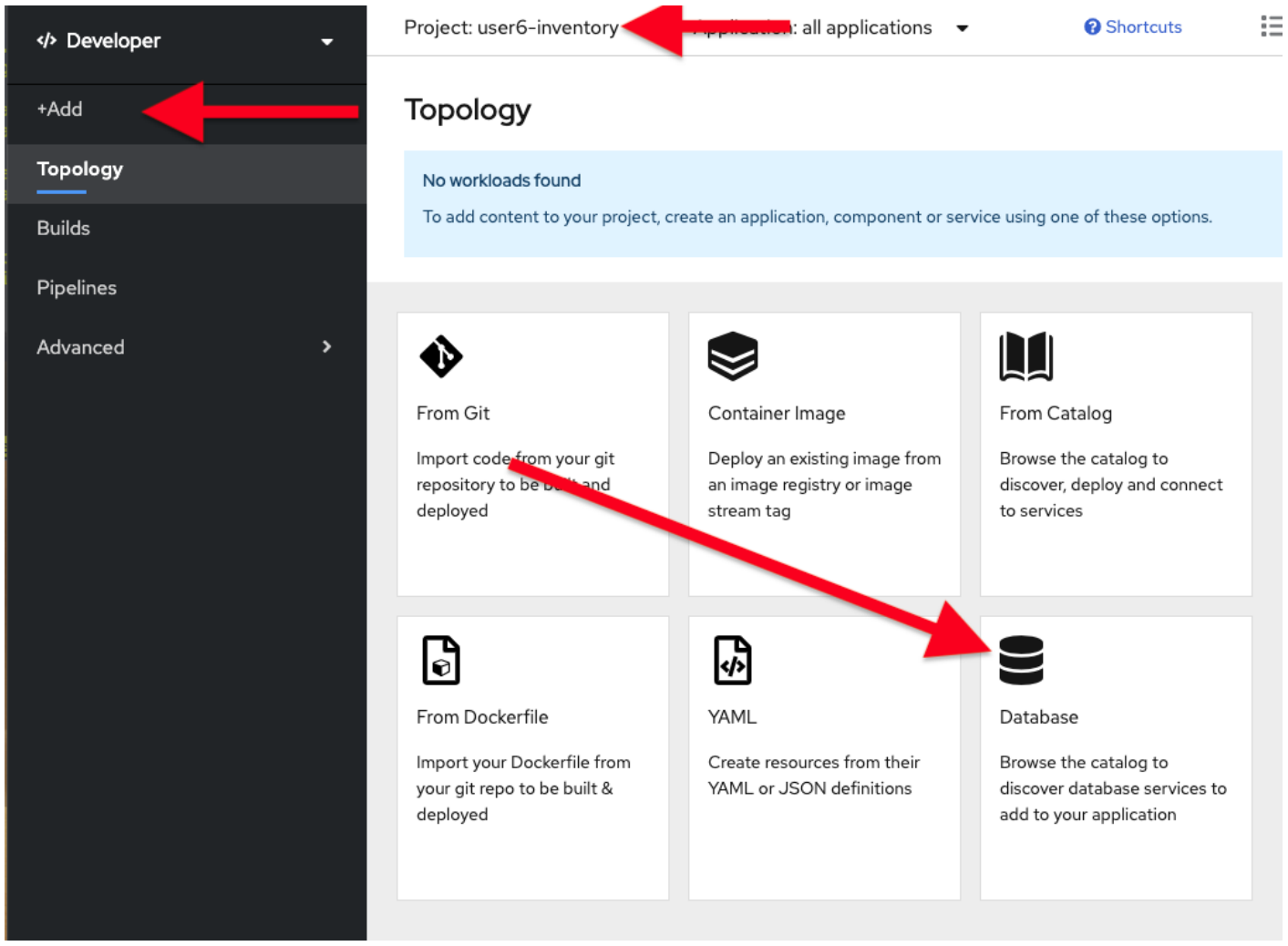
In OpenShift, ensure you're in the *Developer* perspective and then choose the `user16-inventory` project which has already been created for you.

There's nothing there yet, but that's about to change.

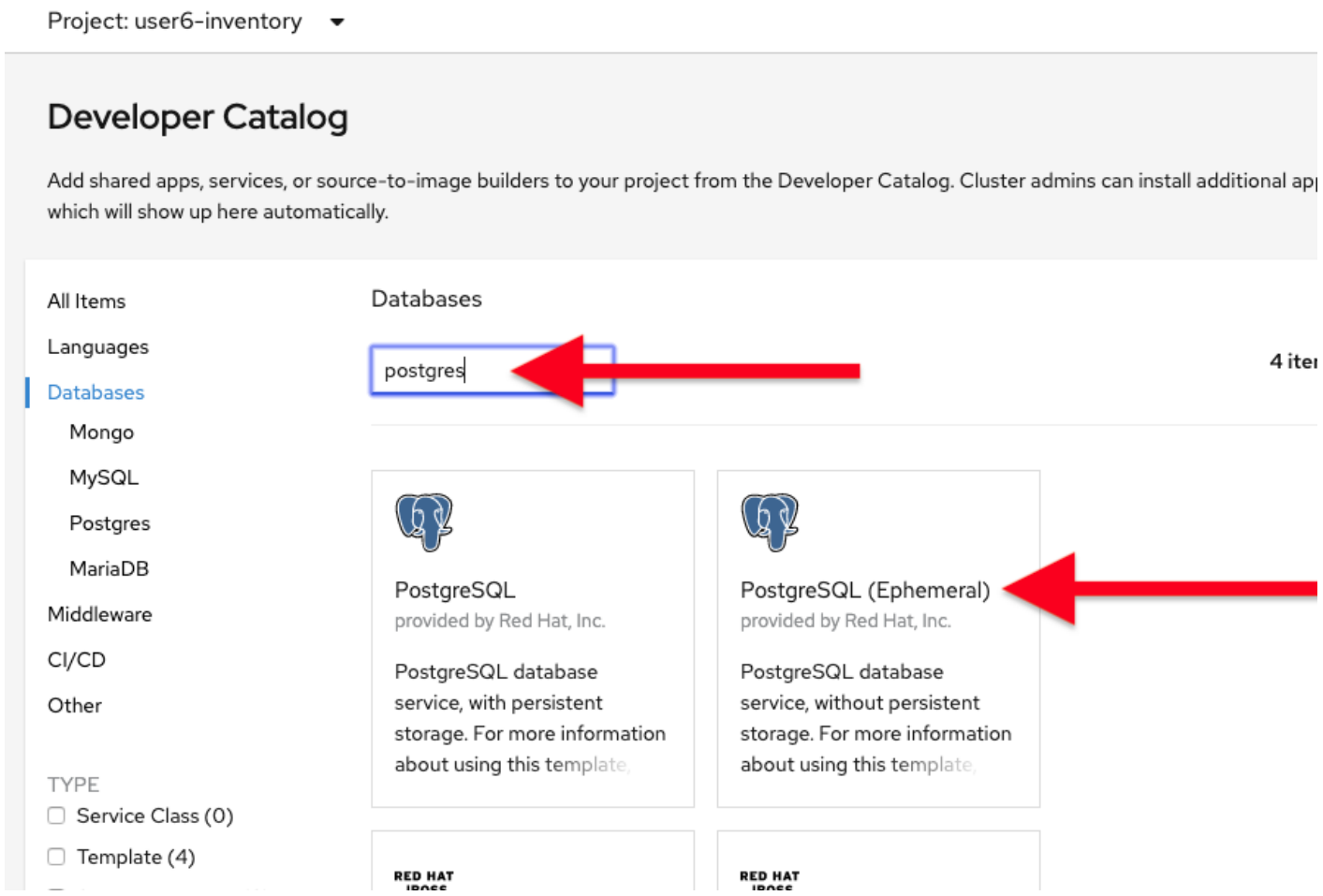
11. Deploy to OpenShift

Let's deploy our new inventory microservice to OpenShift!

Our production inventory microservice will use an external database (PostgreSQL) to house inventory data. First, deploy a new instance of PostgreSQL. Click **+Add** on the left, on the *Database* box on the project overview:

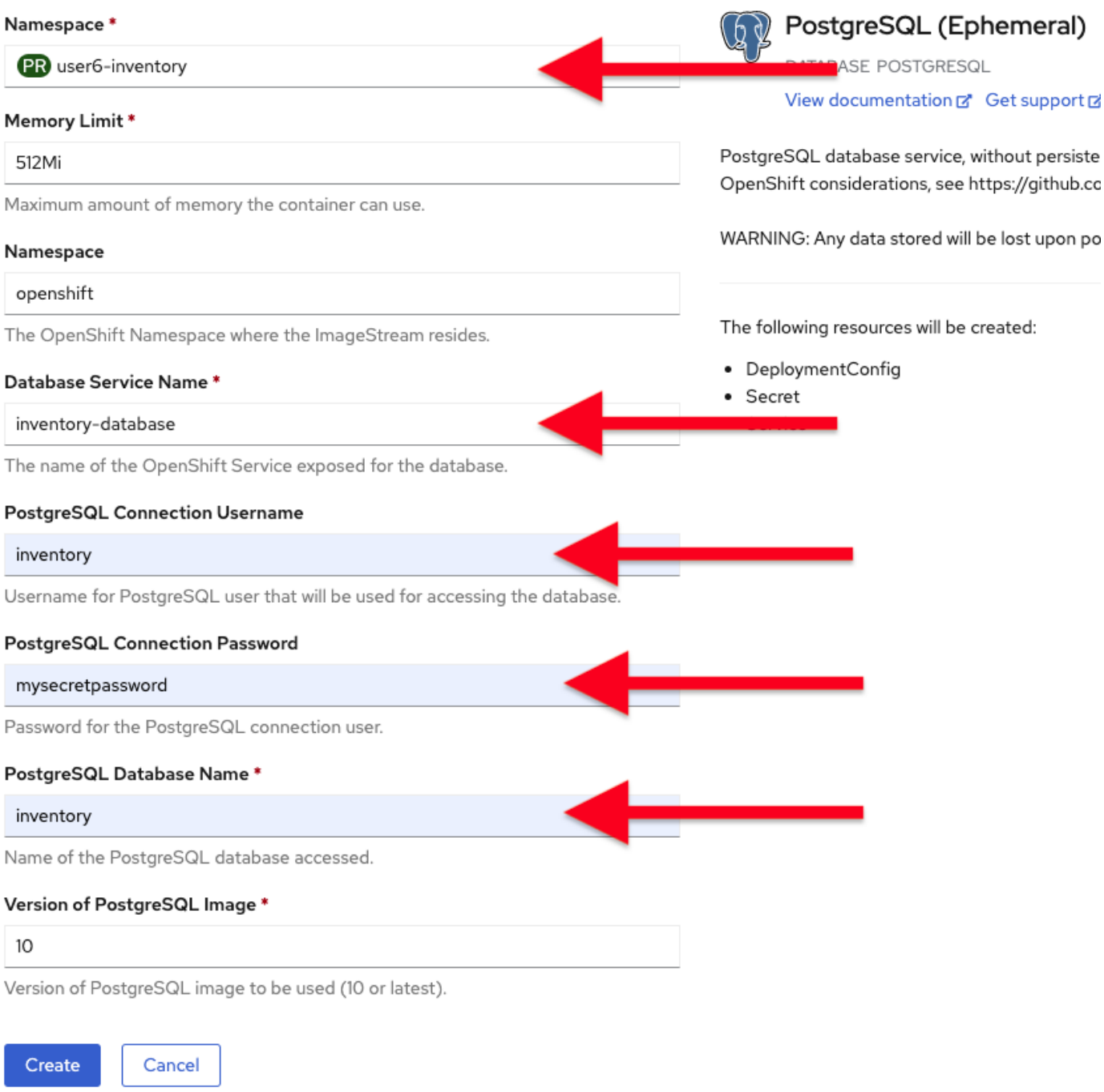


Type in `postgres` in the search box, and click on the **PostgreSQL (ephemeral)**:



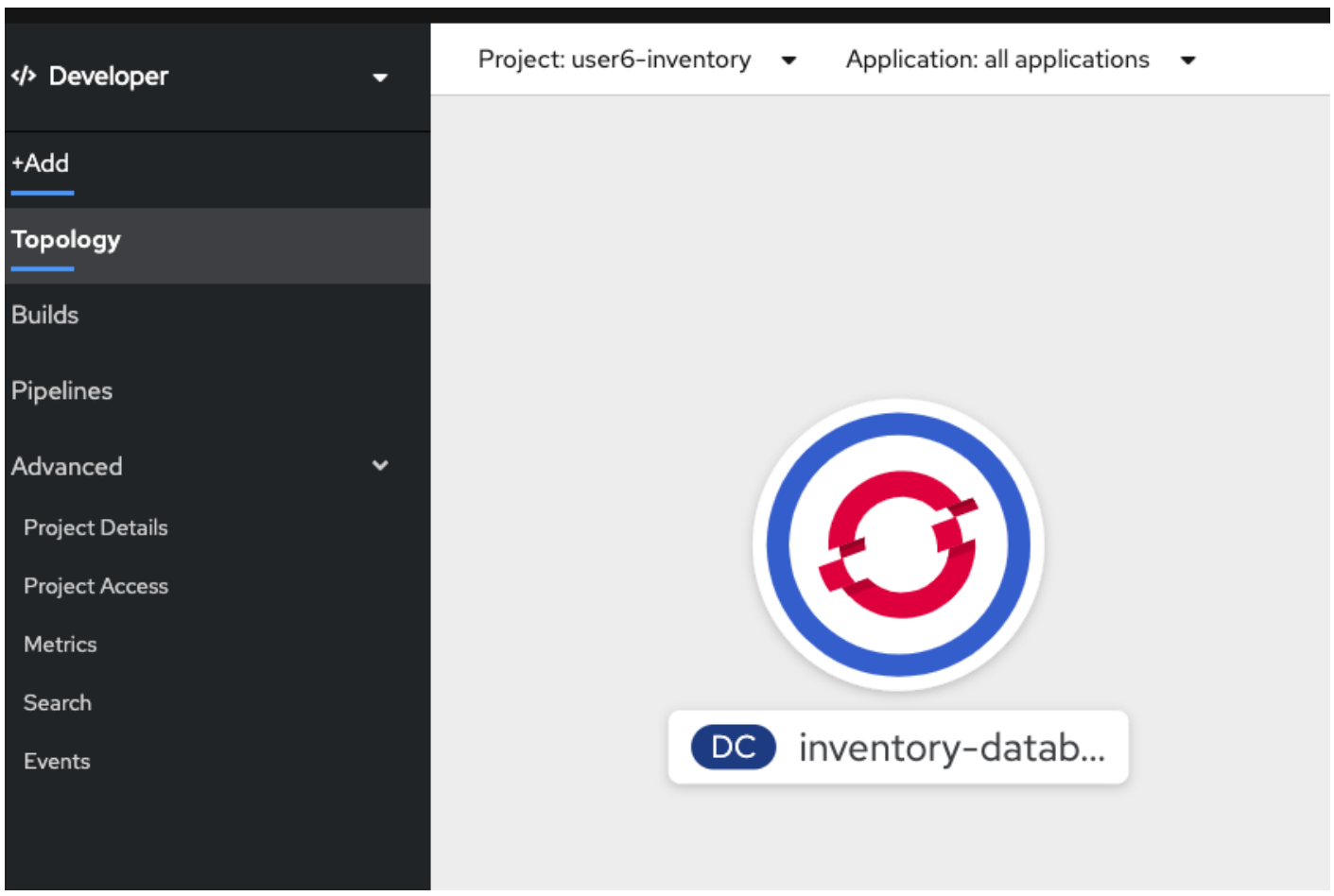
Click on **Instantiate Template** and fill in the following fields, leaving the others as their default values:

- **Namespace**: *choose* `user16-inventory` *for the first Namespace. Leave the second one as `openshift`*
- **Database Service Name**: `inventory-database`
- **PostgreSQL Connection Username**: `inventory`
- **PostgreSQL Connection Password**: `mysecretpassword`
- **PostgreSQL Database Name**: `inventory`



Click **Create**.

This will deploy the database to our new project. Click on the Topology View (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-inventory) to see it:



12. Deploy to OpenShift

Now let's deploy the application itself. Run the following command which will build and deploy using the OpenShift extension:

```sh
oc project user16-inventory && \
mvn clean package -DskipTests -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m1-labs/inventory
```

The output should end with `BUILD SUCCESS`.

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

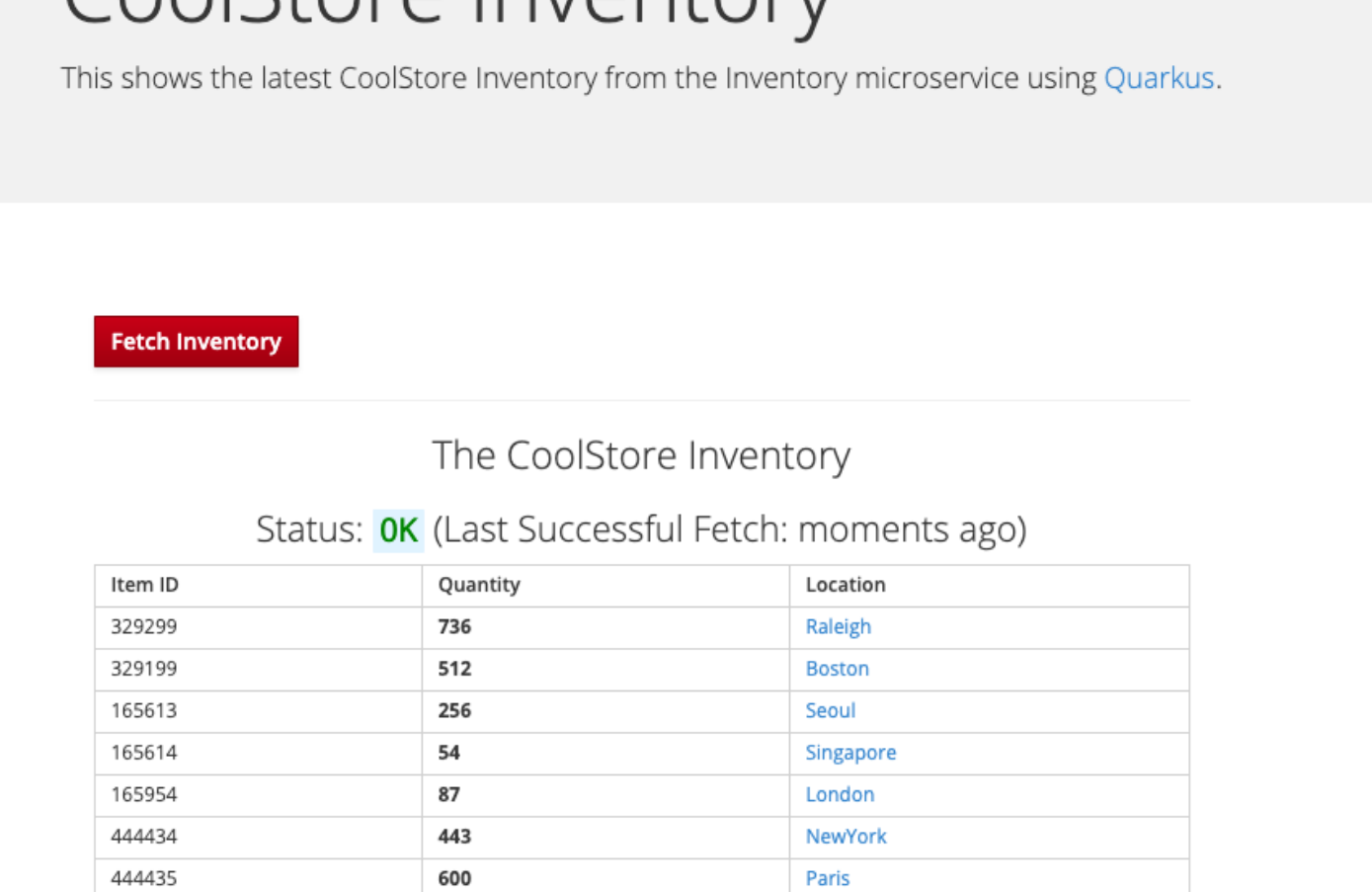
```sh
oc rollout status -w dc/inventory
```

Wait for that command to report **replication controller *inventory-1* successfully rolled out** before continuing.

And label the items with proper icons:

```sh
oc label dc/inventory-database app.openshift.io/runtime=postgresql --overwrite && \
oc label dc/inventory app.kubernetes.io/part-of=inventory --overwrite && \
oc label dc/inventory-database app.kubernetes.io/part-of=inventory --overwrite && \
oc annotate dc/inventory app.openshift.io/connects-to=inventory-database --overwrite && \
oc annotate dc/inventory app.openshift.io/vcs-uri=https://github.com/RedHat-Middleware-Workshops/cloud-native-workshop-v2m1-labs.git --overwrite
&& \
oc annotate dc/inventory app.openshift.io/vcs-ref=ocp-4.4 --overwrite
```

Back on the Topology View (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-inventory), make sure it's done deploying (dark blue circle):



Click on the Route icon above (the arrow) to access our inventory running on OpenShift:

The UI will refresh the inventory table every 2 seconds.

You should also be able to access the health check logic at the *inventory* endpoint using *curl* in the Terminal:

```SH
curl $(oc get route inventory -o jsonpath="{.spec.host}")/health/ready | jq
```

You should the same JSON response:

```JSON
{
  "status": "UP",
  "checks": [
    {
      "name": "Database connections health check",
      "status": "UP"
    },
    {
      "name": "Success of Inventory Health Check!!!",
      "status": "UP"
    }
  ]
}
```

13. Adjust probe timeout

The various timeout values for the probes can be configured in many ways. Let's tune the *readiness probe* initial delay so that we have to wait 30 seconds for it to be activated. Use the *oc* command to tune the probe to wait 30 seconds before starting to poll the probe:

```SH
oc set probe dc/inventory --readiness --initial-delay-seconds=30
```

And verify it's been changed (look at the *delay=* value for the Readiness probe) via CodeReady Workspaces Terminal:

```SH
oc describe dc/inventory | egrep 'Readiness|Liveness'
```

The result:

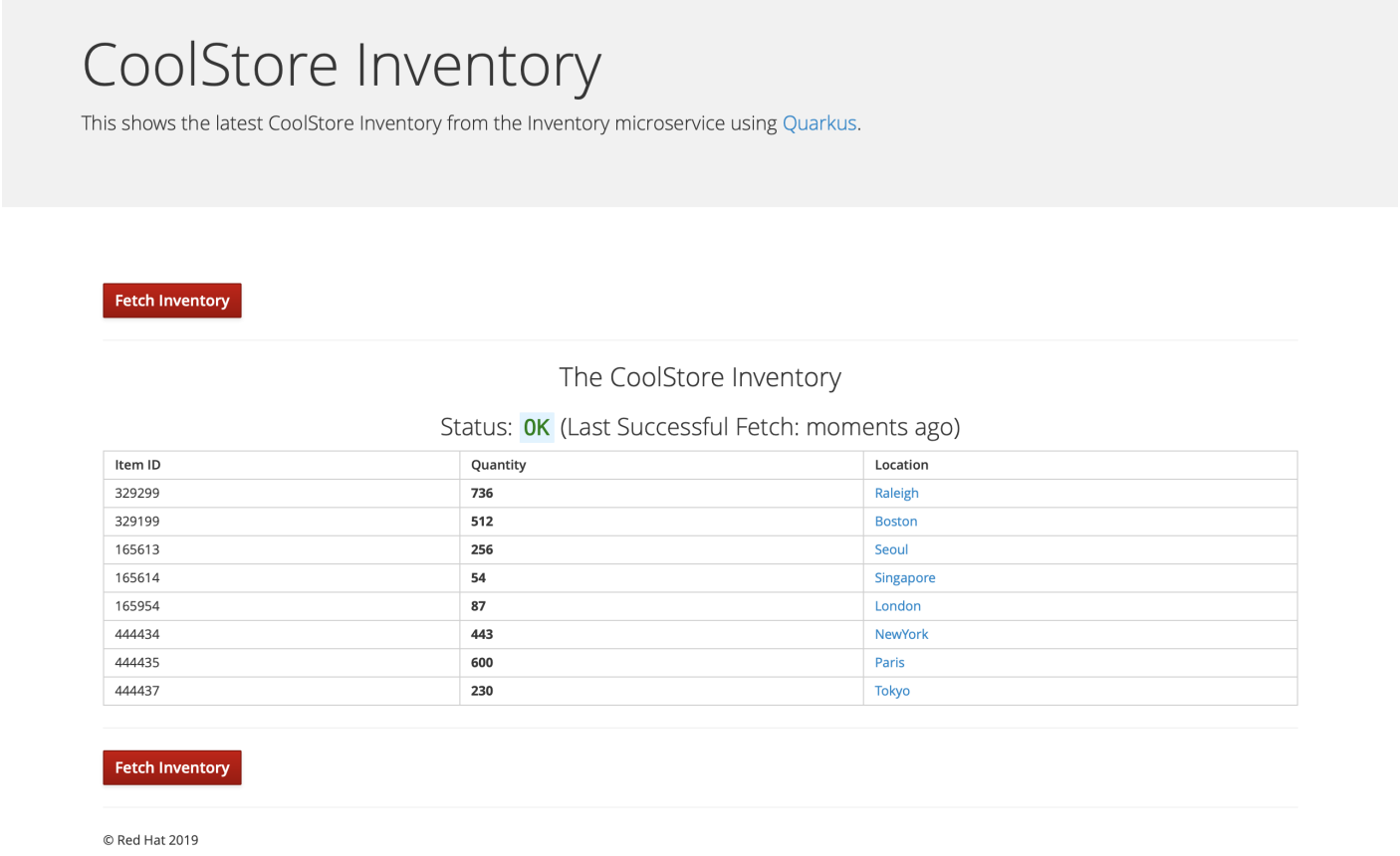
```CONSOLE
    Readiness:          http-get http://:8080/health/ready delay=30s timeout=1s period=10s #success=1 #failure=3
```

In the next step, we'll exercise the probe and watch as it fails and OpenShift recovers the application.

14. Exercise Health Check

Open the Inventory UI (http://inventory-user16-inventory.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com).

This will open up the sample application UI in a new browser tab:



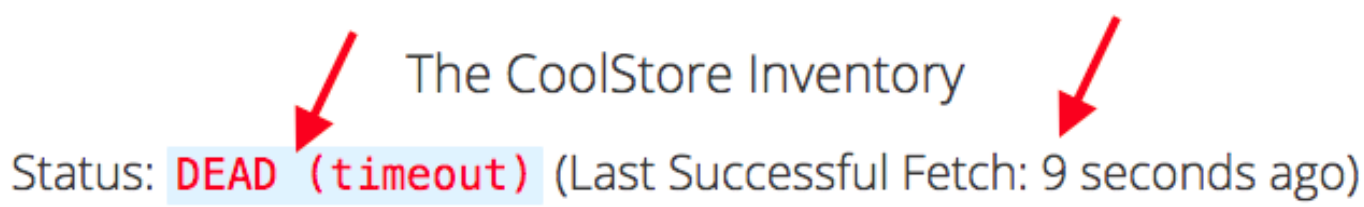
The app will begin polling the inventory as before and report success:

Now you will corrupt the service and cause its health check to start failing. To simulate the app crashing, let's kill the underlying service so it stops responding. Execute via CodeReady Workspaces Terminal:

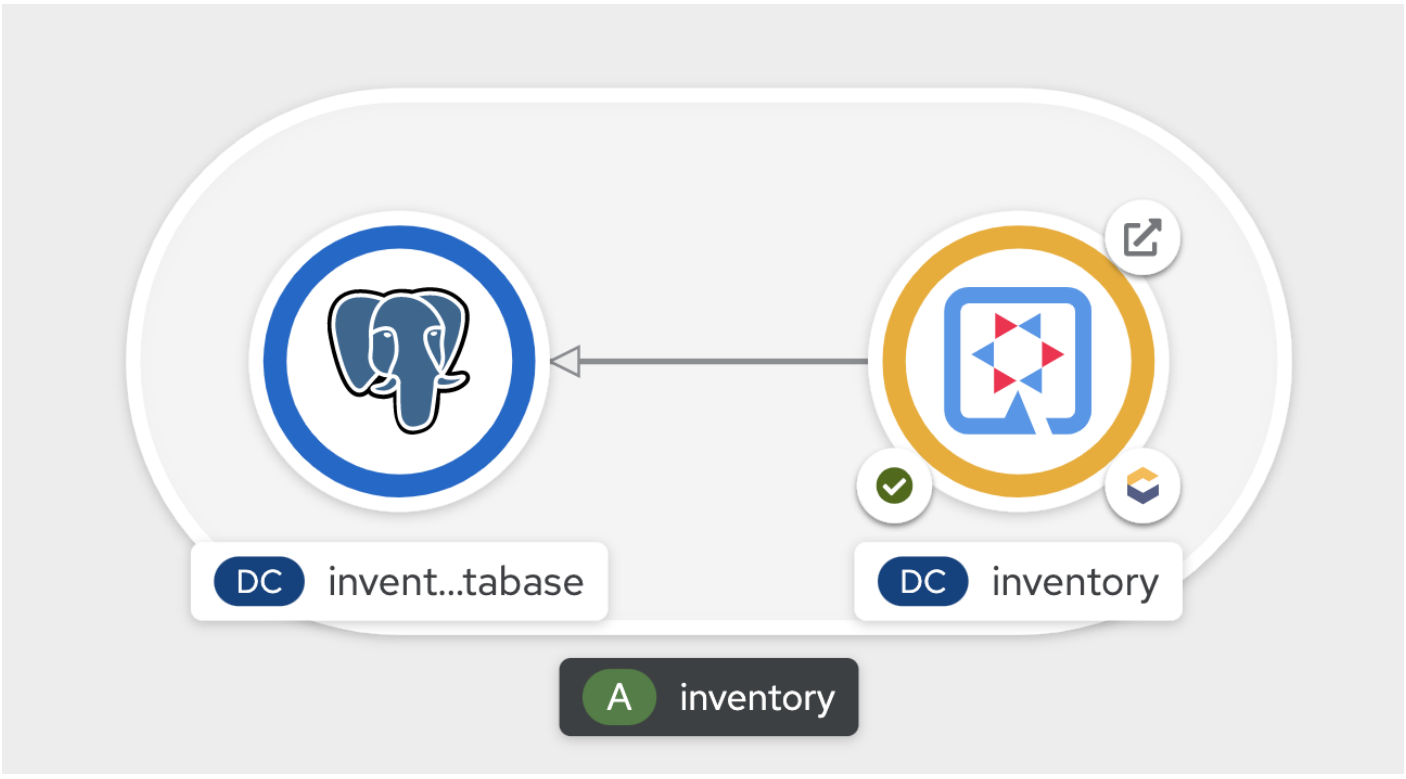
```sh
oc rsh dc/inventory kill 1
```

This will execute the Linux **pkill** command to stop the running Java process in the container.

Check out the application sample UI page and notice it is now failing to access the inventory data, and the *Last Successful Fetch* counter starts increasing, indicating that the UI cannot access inventory. This could have been caused by an overloaded server, a bug in the code, or any other reason that could make the application unhealthy.



Back in the Topology View (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-inventory), you will see the pod is failing (light blue or yellow warning circle):



After too many healthcheck probe failures, OpenShift will forcibly kill the pod and container running the service, and spin up a new one to take its place. Once this occurs, the light blue or yellow warning circle should return to dark blue. This should take about 30 seconds.

Return to the same sample app UI (without reloading the page) and notice that the UI has automatically re-connected to the new service and successfully accessed the inventory once again:

Let's set the probe back to more appropriate values:

```sh
oc set probe dc/inventory --readiness --initial-delay-seconds=5 --period-seconds=5 --failure-threshold=15
```

Summary

You learned a bit more about what Quarkus is, and how it can be used to create modern Java microservice-oriented applications.

You created a new Inventory microservice representing functionality previously implmented in the monolithic CoolStore application. For now this new microservice is completely disconnected from our monolith and is not very useful on its own. In future steps you will link this and other microservices into the monolith to begin the process of strangling the monolith (https://www.martinfowler.com/bliki/StranglerApplication.html).

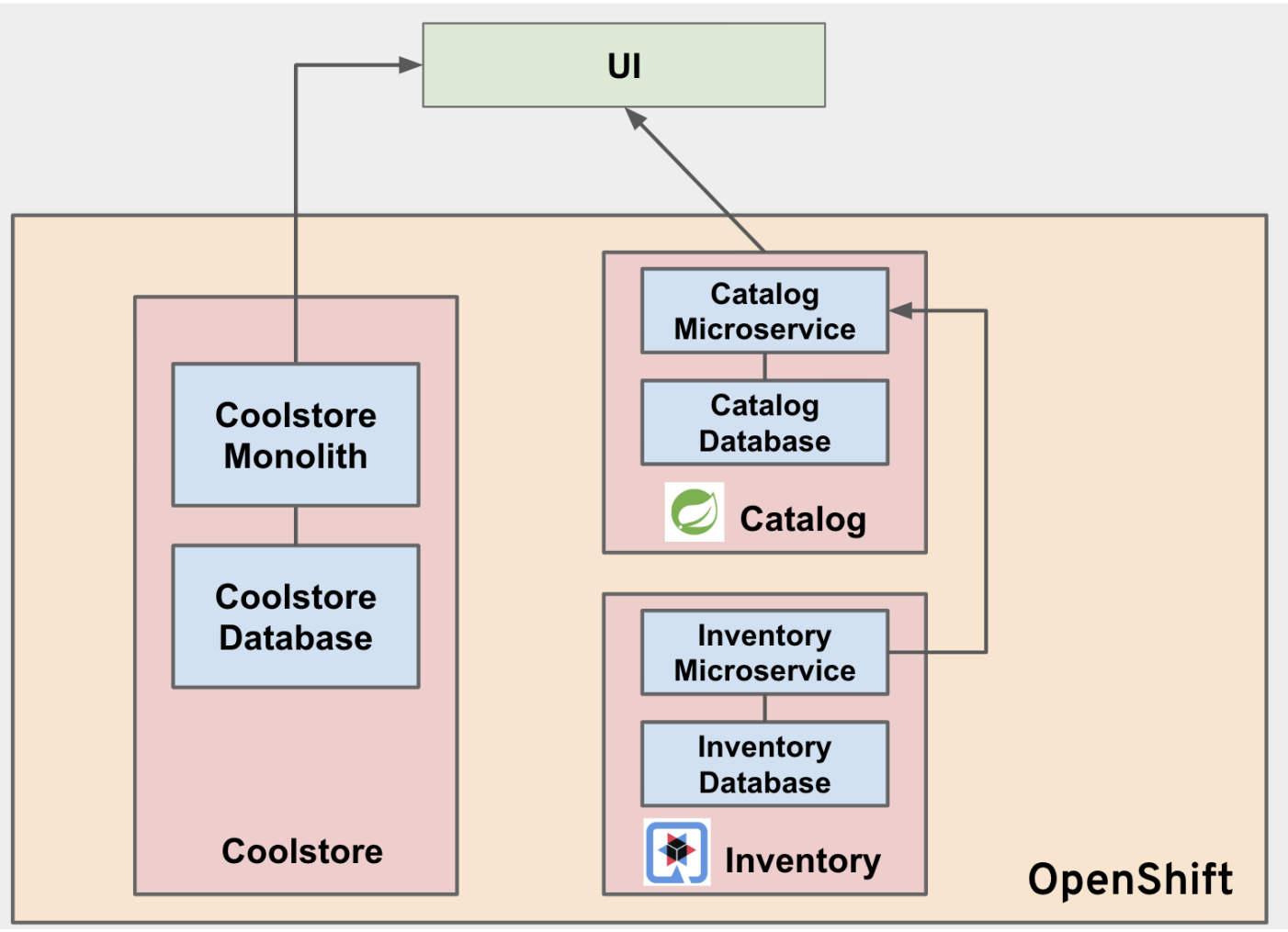
In the next lab, you'll use Spring Boot, another popular framework, to implement additional microservices. Let's go!

Break Monolith Apart - II

In the previous lab, you learned how to take an existing monolithic app and refactor a single *inventory* service using Quarkus. The previous lab resulted in you creating an inventory service, but so far we haven't started *strangling* the monolith. That is because the inventory service is never called directly by the UI. It's a backend service that is only used only by other backend services. In this lab, you will create the catalog service and the catalog service will call the inventory service. When you are ready, you will change the route to tie the UI calls to new service.

To implement this, we are going to use Red Hat support for Spring Boot (https://access.redhat.com/products/spring-boot). The reason for using Spring for this service is to introduce you to Spring Boot development, and how Red Hat Runtimes (https://www.redhat.com/en/products/runtimes) helps to make Spring development on Kubernetes easy. In real life, the reason for choosing Spring Boot vs. others mostly depends on personal preferences, like existing knowledge, etc. At the core Spring and Java EE are very similar.

The goal is to produce something like:



What is Spring Framework?

Spring is one of the most popular Java Frameworks and offers an alternative to the Java EE programming model. Spring is also very popular for building applications based on microservices architectures. Spring Boot is a popular tool in the Spring ecosystem that helps with organizing and using 3rd-party libraries together with Spring and also provides a mechanism for boot strapping embeddable runtimes, like Apache Tomcat. Bootable applications (sometimes also called *fat jars*) fits the container model very well since in a container platform like OpenShift responsibilities like starting, stopping and monitoring applications are then handled by the container platform instead of an Application Server.

*Red Hat& offers support and maintenance over stated time periods for the major versions of *Spring Boot*. Learn more (https://access.redhat.com/documentation/en-us/red_hat_support_for_spring_boot).

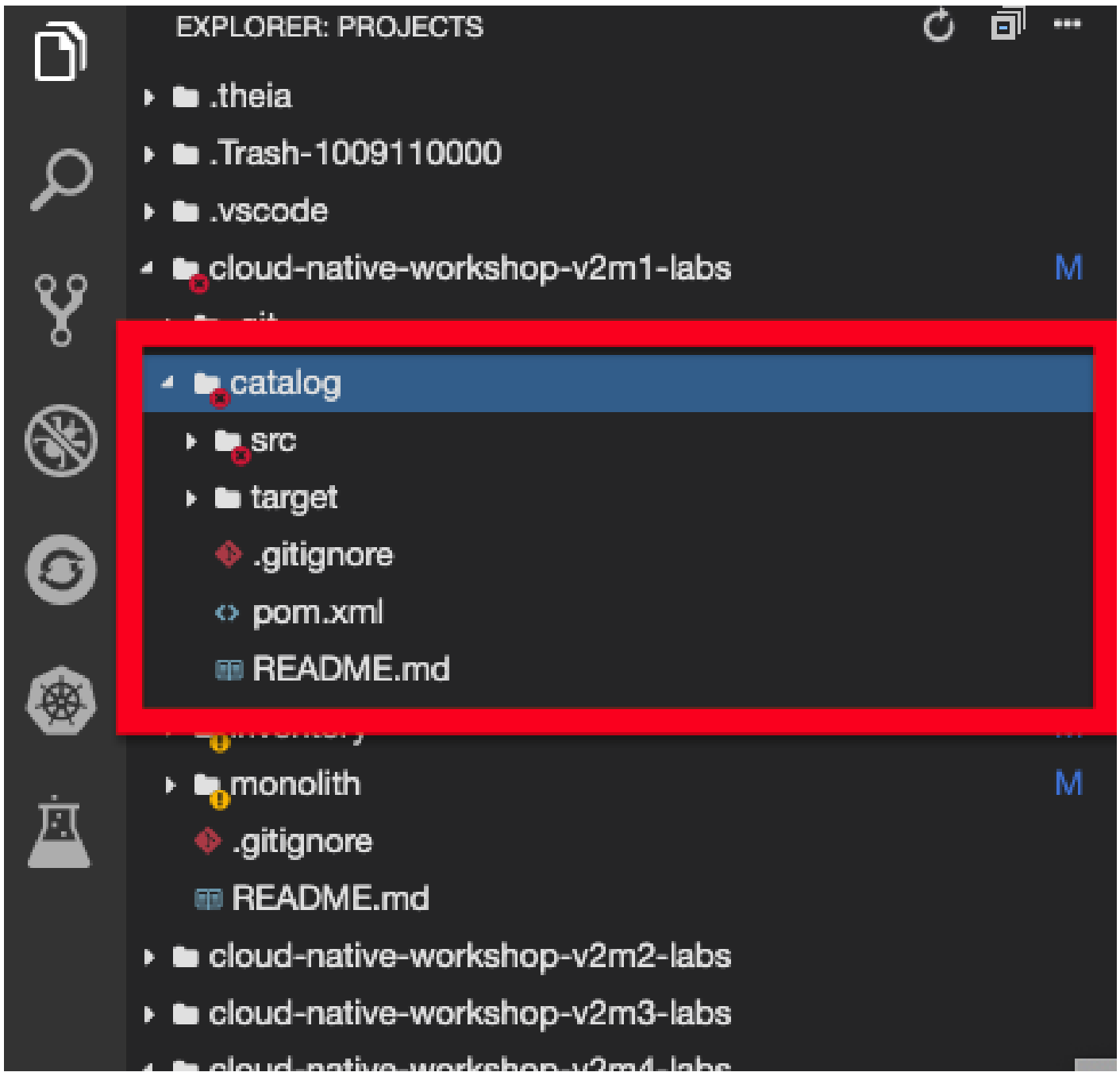
Aggregate microservices calls

Another thing you will learn in this lab is one of the techniques to aggregate services using service-to-service calls. Other possible solutions would be to use a microservices gateway or combine services using client-side logic.

1. Setup a Catalog project

Run the following commands to set up your environment for this lab and start in the right directory:

In the project explorer, expand the *catalog* project (ignore the red error icons on some files - we'll get to those shortly)



2. Examine the Maven project structure

The sample project shows the components of a basic Spring Boot project laid out in different subdirectories according to Maven best practices.

As you can see, there are some files that we have prepared for you in the project. Under *src/main/resources/static/index.html* we have for example prepared a simple html-based UI file for you. This matches very well what you would get if you generated an empty project from the Spring Initializr (https://start.spring.io) web page.

One file that differs slightly is the `pom.xml` . Please open the and examine it a bit closer (but do not change anything at this time)

As you review the content, you will notice that there are a lot of *TODO* comments. **Do not remove them!** These comments are used as a marker and without them, you will not be able to finish this lab.

Notice that we are not using the default BOM (Bill of material) that Spring Boot projects typically use. Instead, we are using a BOM provided by Red Hat as part of the Snowdrop (http://snowdrop.me/) project.

```xml
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>me.snowdrop</groupId>
            <artifactId>spring-boot-bom</artifactId>
            <version>2.1.13.Final-redhat-00001</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

We use this bill of material to make sure that we are using the version of for example Apache Tomcat that Red Hat supports.

3. Add web (Apache Tomcat) to the application

Our application will be a web application, so we need to use a servlet container like Apache Tomcat or Undertow. Since Red Hat offers support for Apache Tomcat (e.g., security patches, bug fixes, etc.), we will use it.

Undertow is another an open source project that is maintained by Red Hat and therefore Red Hat plans to add support for Undertow shortly.

Because of the Red Hat BOM and access to the Red Hat maven repositories all we need to do to enable the supported Apache Tomcat as servlet container is to add the following dependency to your *pom.xml*. Add these lines at the `<!-- TODO: Add web (tomcat) dependency here -->` marker:

```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

We will also make use of Java Persistence API (JPA) so we need to add the following to *pom.xml* at the `<!-- TODO: Add jdbc dependency here -->` marker:

```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
```

We will go ahead and add a bunch of other dependencies while we have the pom.xml open. These will be explained later. Add these at the `<!-- TODO: Add actuator, feign and hystrix dependency here -->` marker:

```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
</dependency>
```

This should also make the red error icons disappear! Now, build the project to make sure everything compiles so far:

```sh
mvn -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m1-labs/catalog clean package
```

If it builds successfully (you will see **BUILD SUCCESS**), you have now successfully executed the first step in this lab.

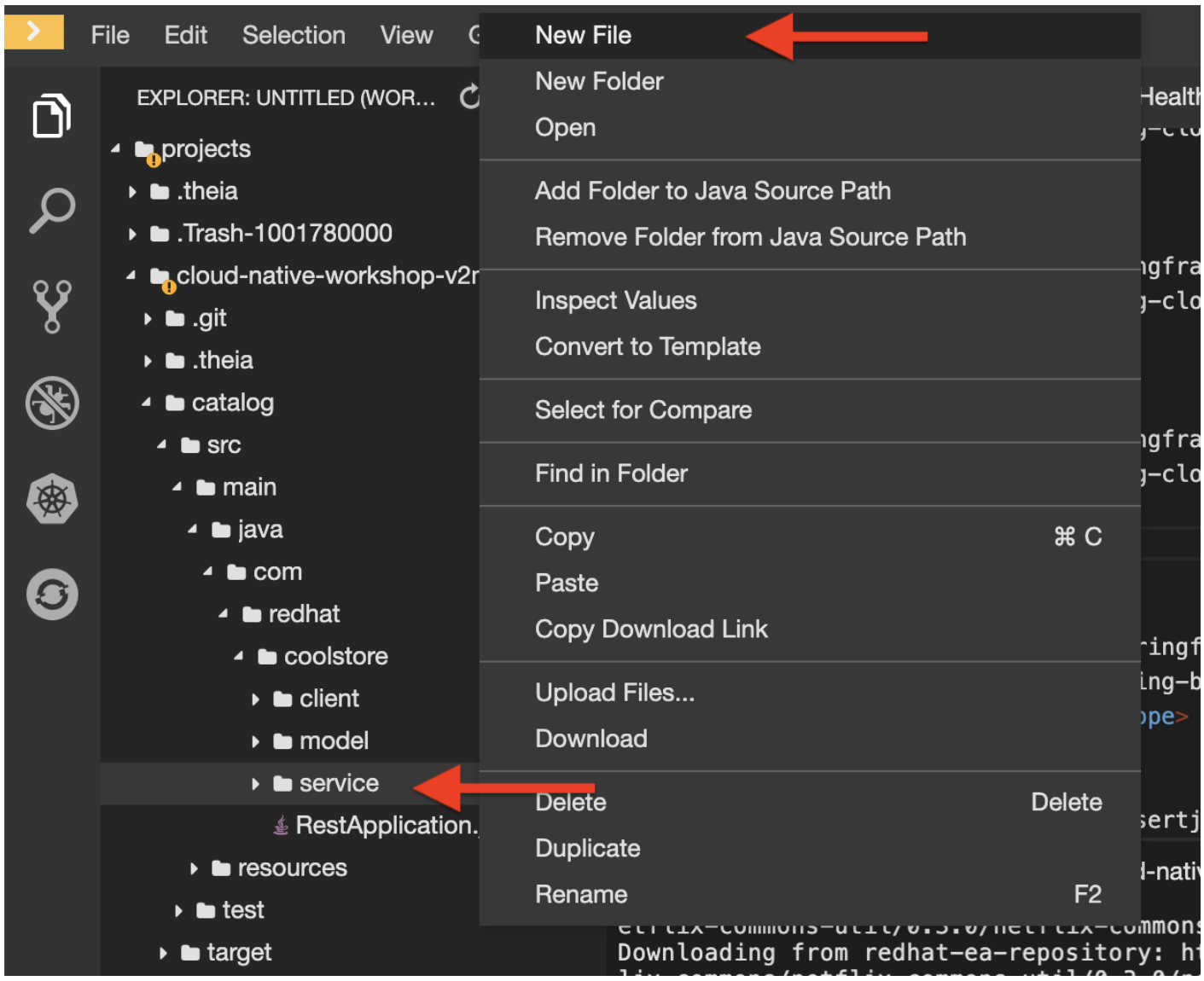
Now you've seen how to get started with Spring Boot development on Red Hat Runtimes.

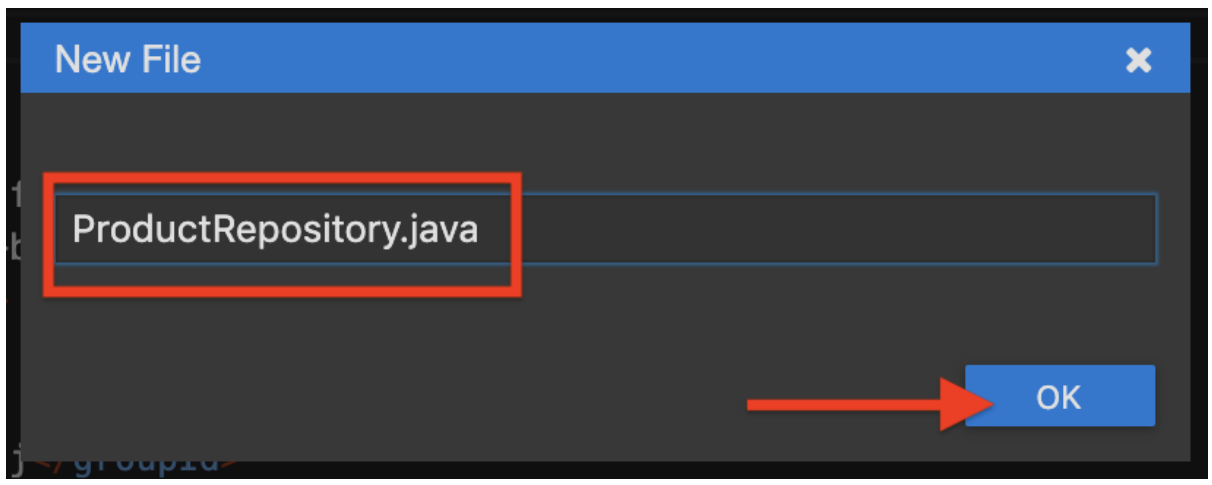
In next step of this lab, we will add the logic to be able to read data from the database.

4. Create Domain Objects

We are now ready to implement the database repository.

In the catalog project, right-click on the `src/main/java/com/redhat/coolstore/service` directory and select **New File**. Name the file `ProductRepository.java` :





In the file, add this code:

```java
package com.redhat.coolstore.service;

import java.util.List;

import com.redhat.coolstore.model.Product;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

@Repository
public class ProductRepository {

    //TODO: Autowire the jdbcTemplate here

    //TODO: Add row mapper here

    //TODO: Create a method for returning all products

    //TODO: Create a method for returning one product

}
```

This class is annotated with `@Repository`. This is a feature of Spring that makes it possible to avoid a lot of boiler plate code and only write the implementation details for this data repository. It also makes it very easy to switch to another data storage, like a NoSQL database.

Spring Data provides a convenient way for us to access data without having to write a lot of boiler plate code. One way to do that is to use a *JdbcTemplate*. First we need to autowire that as a member to *ProductRepository*. Add these at the `TODO: Autowire the jdbcTemplate here` marker:

```java
    @Autowired
    private JdbcTemplate jdbcTemplate;
```

The *JdbcTemplate* require that we provide a *RowMapper* so that it can map between rows in the query to Java Objects. We are going to define the *RowMapper* like this. Add these at the `<!-- TODO: Add row mapper here -->` marker:

```java
private RowMapper<Product> rowMapper = (rs, rowNum) -> new Product(
        rs.getString("itemId"),
        rs.getString("name"),
        rs.getString("description"),
        rs.getDouble("price"));
```

Now we are ready to create the business methods. Let's start with the `readAll()`. It should return a `List<Product>` and then we can write the query as `SELECT * FROM catalog` and use the rowMapper to map that into `Product` objects. Add these at the `<!-- TODO: Create a method for returning all products -->` marker:

```java
public List<Product> readAll() {
    return this.jdbcTemplate.query("SELECT * FROM catalog", rowMapper);
}
```

We also need a way to find a single product element. Add these at the `<!-- TODO: Create a method for returning one product -->` marker:

```java
public Product findById(String id) {
    return this.jdbcTemplate.queryForObject("SELECT * FROM catalog WHERE itemId = ?", new Object[]{id}, rowMapper);
}
```

The *ProductRepository* should now have all the components, but we still need to tell spring how to connect to the database. For local development we will use the H2 in-memory database. Later, when deploying this to OpenShift we will use the PostgreSQL database, which matches what we are using in production.

The Spring Framework has a lot of sane defaults that can always seem magical sometimes, but basically all we have to do to setup the database driver is to provide some configuration values. Open `src/main/resources/application-default.properties` and add the following properties where the comment says `#TODO: Add database properties`.

```properties
spring.datasource.url=jdbc:h2:mem:catalog;DB_CLOSE_ON_EXIT=FALSE
spring.datasource.username=sa
spring.datasource.password=sa
spring.datasource.driver-class-name=org.h2.Driver
```

The Spring Data framework will automatically see if there is a `schema.sql` in the class path and run that when initializing.

Now you've seen how to use Spring Data to collect data from the database and how to use a local H2 database for development and testing.

In next step of this lab, we will add the logic to expose the database content from REST endpoints using JSON format.

5. Create Catalog Service

Now you are going to create a service class. Later on the service class will be the one that controls the interaction with the inventory service, but for now it's basically just a wrapper of the repository class.

Again, create a new class `CatalogService.java` in the `src/main/java/com/redhat/coolstore/service` package.

Replace the empty class with this code:

```java
package com.redhat.coolstore.service;

import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

//import com.redhat.coolstore.client.InventoryClient;
import com.redhat.coolstore.model.Product;

import org.json.JSONArray;
import org.json.JSONObject;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class CatalogService {

    @Autowired
    private ProductRepository repository;

    //TODO: Autowire Inventory Client

    public Product read(String id) {
        Product product = repository.findById(id);
        //TODO: Update the quantity for the product by calling the Inventory service
        return product;
    }

    public List<Product> readAll() {
        List<Product> productList = repository.readAll();
        //TODO: Update the quantity for the products by calling the Inventory service
        return productList;
    }

}
```

As you can see there are a number of `TODO` in the code, and later we will use these placeholders to add logic for calling the Inventory Client to get the quantity.

Now we are ready to create the endpoints that will expose REST service.

Start by creating a new class called `CatalogEndpoint.java` in the `src/main/java/com/redhat/coolstore/service` package.

Replace the contents with this code:

```java
package com.redhat.coolstore.service;

import java.util.List;
import com.redhat.coolstore.model.Product;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/services")
public class CatalogEndpoint {

    private final CatalogService catalogService;

    public CatalogEndpoint(CatalogService catalogService) {
      this.catalogService = catalogService;
    }

    @GetMapping("/products")
    public List<Product> readAll() {
      return this.catalogService.readAll();
    }

    @GetMapping("/product/{id}")
    public Product read(@PathVariable("id") String id) {
      return this.catalogService.read(id);
    }

}
```

The Spring MVC Framework by default uses *Jackson* to serialize or map Java objects to JSON and vice-versa. Jackson extends upon JAX-B and can automatically parse simple Java structures and parse them into JSON and vice versa. Our `Product.java` pre-created class is very simple and only contains basic attributes we do not need to tell Jackson how to parse between Product and JSON.

Since we now have endpoints that return the catalog we can also start the service and load the default page again, which should now return the products.

Start the application via the CodeReady Workspaces Terminal using the following command:

```sh
mvn clean spring-boot:run -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m1-labs/catalog
```

If you get a popup about port `8081` being available, simply dismiss it with the `X` button.

Wait for the application to start and get the `Started RestApplication in 4.025 seconds (JVM running for 4.361)` log message. Then, verify the endpoint by running the following command in a separate Terminal:

```sh
curl http://localhost:8081/services/products | jq
```

You should get a full JSON array consisting of all the products:

```json
  {
    "itemId": "329299",
    "name": "Red Fedora",
    "desc": "Official Red Hat Fedora",
    "price": 34.99,
    "quantity": 0
  },
  { ... }
```

You have now successfully executed the third step in this lab.

Now you've seen how to create a REST application in Spring MVC and create a simple application that returns product.

In the next step, we will also call another service to enrich the endpoint response with inventory status.

Make sure to stop the service by clicking `CTRL-C` in the terminal that the app is running in!

6. Get inventory data

When redesigning our application to Microservices using domain driven design we have identified that Inventory and Product Catalog are two separate domains. However our current UI expects to retrieve data from both the Catalog Service and Inventory service in a singe request.

Service interaction

Our problem is that the user interface requires data from two services when calling the REST service on `/services/products`. There are multiple ways to solve this like:

1. Client Side integration - We could extend our UI to first call `/services/products` and then for each product item call `/services/inventory/{prodId}` to get the inventory status and then combine the result in the web browser. This would be the least intrusive method, but it also means that if we have 100 of products the client will make 101 requests to the server. If we have a slow internet connection this may cause issues.

2. Microservices Gateway - Creating a gateway in front of the *Catalog Service* that first calls the Catalog Service and then based on the response calls the inventory is another option. This way we can avoid lots of calls from the client to the server. Apache Camel (http://camel.apache.org) provides nice capabilities to do this and if you are interested to learn more about this, please checkout the Coolstore Microservices example: Here (http://github.com/jbossdemocentral/coolstore-microservice)

3. Service-to-Service - Depending on use-case and preferences another solution would be to do service-to-service calls instead. In our case means that the Catalog Service would call the Inventory service using REST to retrieve the inventory status and include that in the response.

There are no right or wrong answers here, but since this is a workshop on application modernization using Red Hat Runtimes we will not choose option 1 or 2 here. Instead we are going to use option 3 and extend our Catalog to call the Inventory service.

7. Implementing the Inventory Client

We can now create the client that calls the Inventory. Netflix has provided some nice extensions to the Spring Framework that are mostly captured in the Spring Cloud project, however Spring Cloud is mainly focused on Pivotal Cloud Foundry and because of that Red Hat and others have contributed Spring Cloud Kubernetes to the Spring Cloud project, which enables the same functionallity for Kubernetes based platforms like OpenShift.

The inventory client will use a Netflix project called *Feign*, which provides a nice way to avoid having to write boilerplate code. Feign also integrates with Hystrix which gives us the capability to Circuit Break calls that don't work. We will discuss this more later, but let's start with the implementation of the Inventory Client. Using Feign all we have to do is create a interface that details which parameters and return type we expect, annotate it with `@RequestMapping` and provide some details and then annotate the interface with `@Feign` and provide it with a name.

Create the `InventoryClient.java` class in the `src/main/java/com/redhat/coolstore/client/` package in the project explorer.

Add the following code to the file:

```java
package com.redhat.coolstore.client;

import feign.hystrix.FallbackFactory;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.http.MediaType;
import org.springframework.stereotype.Component;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@FeignClient(name="inventory")
public interface InventoryClient {

    @RequestMapping(method = RequestMethod.GET, value = "/services/inventory/{itemId}", consumes = {MediaType.APPLICATION_JSON_VALUE})
    String getInventoryStatus(@PathVariable("itemId") String itemId);

    //TODO: Add Fallback factory here

}
```

Open the `src/main/resources/application-default.properties` file. Add these properties to it at the `#TODO: Configure netflix libraries` marker:

```java
inventory.ribbon.listOfServers=inventory.user16-inventory.svc.cluster.local:8080
feign.hystrix.enabled=true
```

By setting `inventory.ribbon.listOfServers` we are hard coding the actual URL of the service to `user16-inventory.svc.cluster.local:8080` (which will point to our inventory microservice we created in the last lab). If we had multiple servers we could also add those using a comma. However using Kubernetes there is no need to have multiple endpoints listed here since Kubernetes has a concept of *Services* that will internally route between multiple instances of the same service.

Now that we have a client we can make use of it in our *CatalogService*.

Open *src/main/java/com/redhat/coolstore/service/CatalogService.java*

And autowire (e.g. inject) the client into it by inserting this at the `//TODO: Autowire Inventory Client` marker:

```java
    @Autowired
    private InventoryClient inventoryClient;
```

Next, update the `read(String id)` method at the comment `//TODO: Update the quantity for the product by calling the Inventory service` add the following:

```java
        JSONArray jsonArray = new JSONArray(inventoryClient.getInventoryStatus(product.getItemId()));
        List<String> quantity = IntStream.range(0, jsonArray.length())
            .mapToObj(index -> ((JSONObject)jsonArray.get(index))
            .optString("quantity")).collect(Collectors.toList());
        product.setQuantity(Integer.parseInt(quantity.get(0)));
```

Also, don't forget to add the import statement by un-commenting the import statement for InventoryClient near the top

```java
import com.redhat.coolstore.client.InventoryClient;
```

Also in the *readAll()* method replace the comment `//TODO: Update the quantity for the products by calling the Inventory service` with the following:

```java
        productList.forEach(p -> {
          JSONArray jsonArray = new JSONArray(this.inventoryClient.getInventoryStatus(p.getItemId()));
          List<String> quantity = IntStream.range(0, jsonArray.length())
            .mapToObj(index -> ((JSONObject)jsonArray.get(index))
            .optString("quantity")).collect(Collectors.toList());
          p.setQuantity(Integer.parseInt(quantity.get(0)));
        });
```

Class `JSONArray` is an ordered sequence of values. Its external text form is a string wrapped in square brackets with commas separating the values. The internal form is an object having get and opt methods for accessing the values by index, and element methods for adding or replacing values.

8. Create a fallback for inventory

In the previous step we added a client to call the Inventory service. Services calling services is a common practice in Microservices Architecture, but as we add more and more services the likelihood of a problem increases dramatically. Even if each service has 99.9% update, if we have 100 of services our estimated up time will only be ~90%. We therefore need to plan for failures to happen and our application logic has to consider that dependent services are not responding.

In the previous step we used the Feign client from the Netflix cloud native libraries to avoid having to write boilerplate code for doing a REST call. However Feign also have another good property which is that we easily create fallback logic. In this case we will use static inner class since we want the logic for the fallback to be part of the Client and not in a separate class.

In the `InventoryClient`, add the following code at the `//TODO: Add Fallback factory here` marker:

```java
@Component
class InventoryClientFallbackFactory implements FallbackFactory<InventoryClient> {
  @Override
  public InventoryClient create(Throwable cause) {
    return itemId -> "[{'quantity':-1}]";
  }
}
```

After creating the fallback factory all we have todo is to tell Feign to use that fallback in case of an issue, by adding the fallbackFactory property to the `@FeignClient` annotation. and replace the existing `@FeignClient(name="inventory")` line with this line:

```java
@FeignClient(name="inventory",fallbackFactory = InventoryClient.InventoryClientFallbackFactory.class)
```

9. Slow running services

Having fallbacks is good but that also requires that we can correctly detect when a dependent services isn't responding correctly. Besides from not responding a service can also respond slowly causing our services to also respond slow. This can lead to cascading issues that are hard to debug and pinpoint issues with. We should therefore also have sane defaults for our services. You can add defaults by adding them to the configuration.

Open `src/main/resources/application-default.properties`

And add this line to it at the `#TODO: Set timeout to for inventory` marker:

```
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=20000
```

Let's re-test our app locally. Re-build and re-run the app:

```sh
mvn clean spring-boot:run -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m1-labs/catalog
```

Then, access the product catalog again in a separate terminal:

```sh
curl http://localhost:8081/services/products | jq
```

You will see:

```json
  {
    "itemId": "444437",
    "name": "Lytro Camera",
    "desc": "Consumers who want to up their photography game are looking at newfangled cameras like the Lytro Field camera, designed to take photos with infinite focus, so you can decide later exactly where you want the focus of each image to be.",
    "price": 44.3,
    "quantity": 230
  }
```

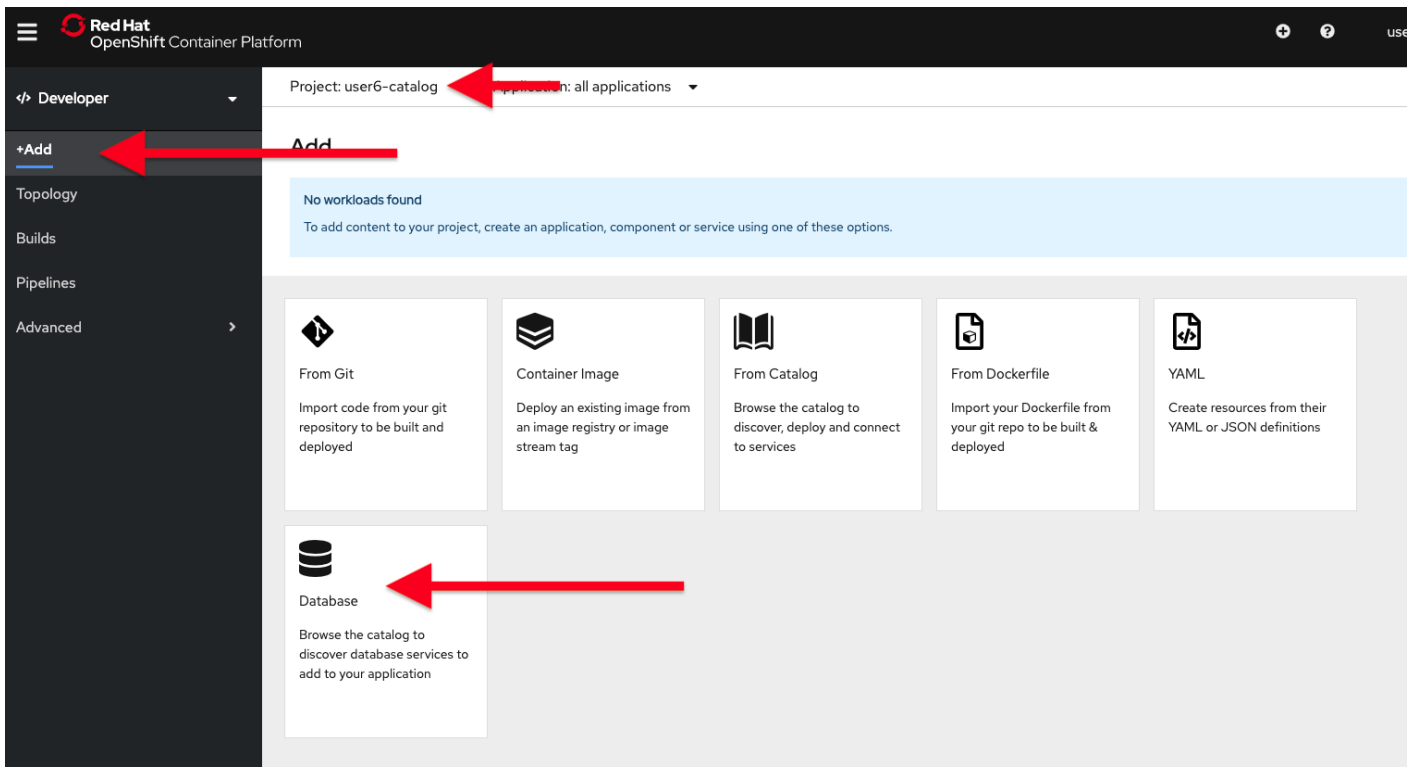
Notice the `"quantity": 230` - because CodeReady Workspaces runs in our OpenShift cluster, our value for `inventory.ribbon.listOfServers` we set earlier is completely valid!

Congratulations! You now have the framework for retrieving products from the product catalog and enriching the data with inventory data from an external service. In next step of this lab we will deploy our application to OpenShift Container Platform and then start adding additional features to take care of various aspects of cloud native microservice development.

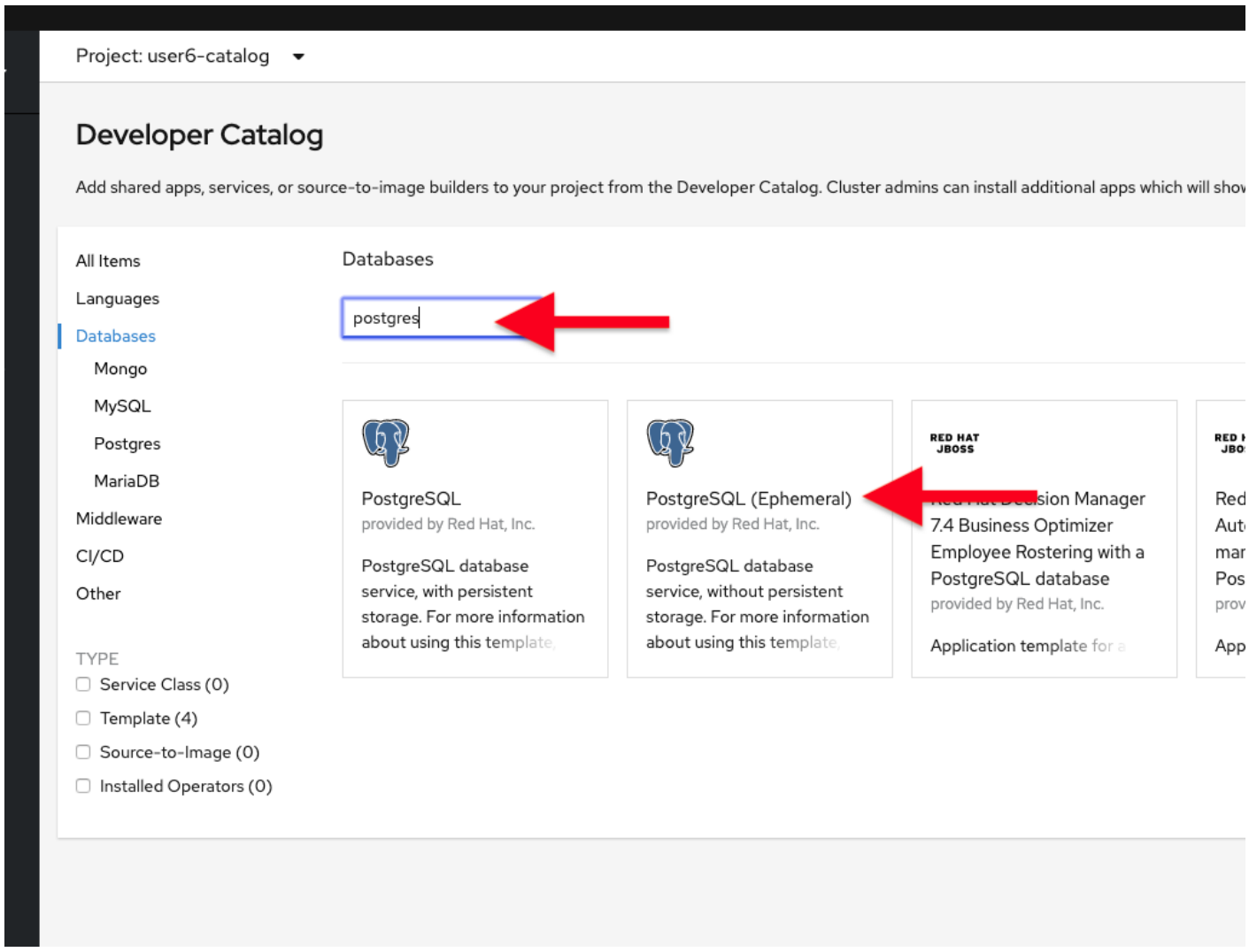
10. Add Database OpenShift

Our production catalog microservice will use an external database (PostgreSQL) to house inventory data. We've created an `user16-catalog` project for you. Visit the Topology View for user16-catalog project (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-catalog).

Click **+Add** on the left, on the *Database* box on the project overview:

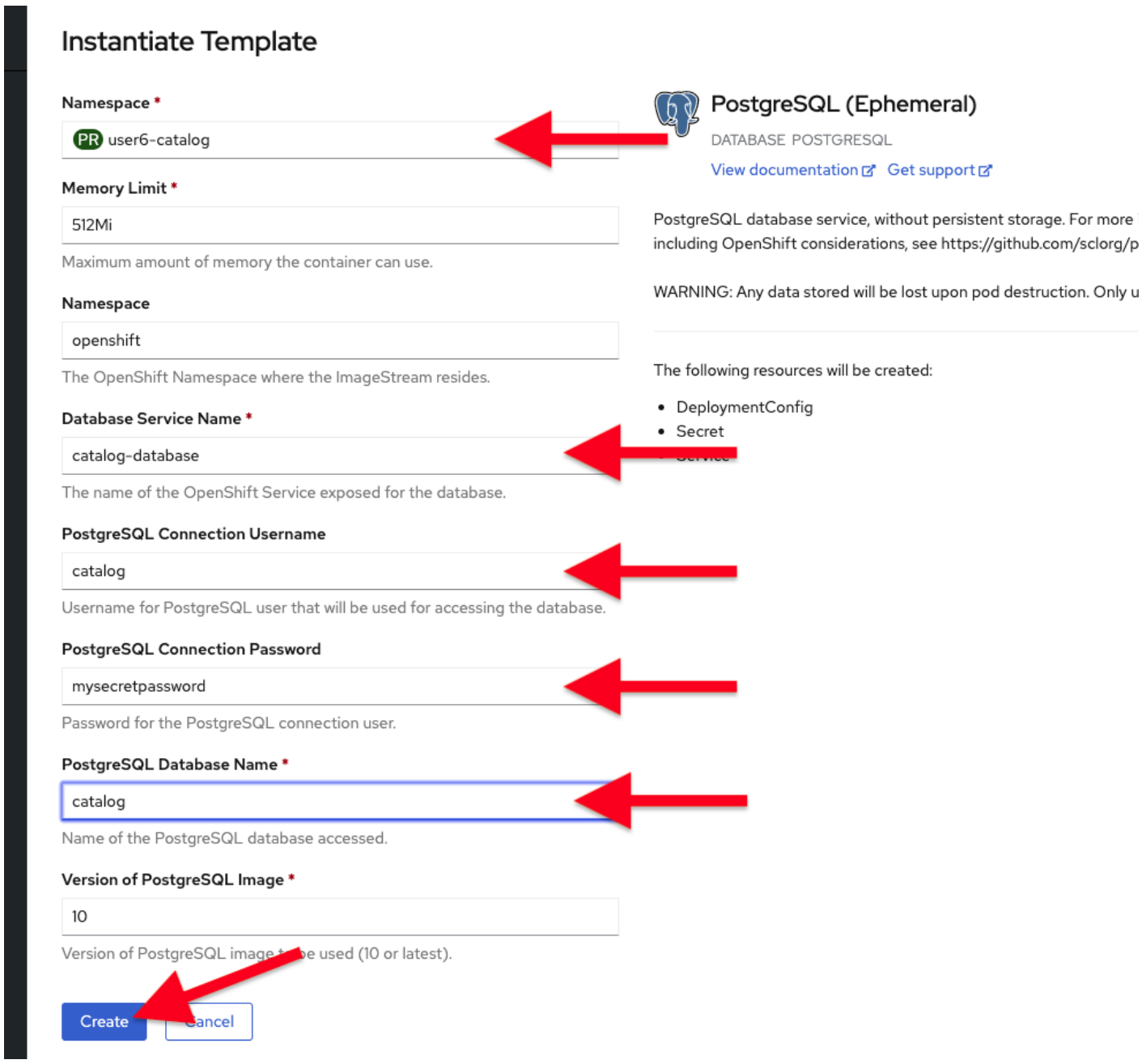


Type in `postgres` in the search box, and click on the **PostgreSQL (ephemeral)**:



Click on **Instantiate Template** and fill in the following fields, leaving the others as their default values:

- **Namespace**: *choose* `user16-catalog` *for the first Namespace. Leave the second one as `openshift`*
- **Database Service Name**: `catalog-database`
- **PostgreSQL Connection Username**: `catalog`
- **PostgreSQL Connection Password**: `mysecretpassword`
- **PostgreSQL Database Name**: `catalog`



This will deploy the database to our catalog project. Click on the Topology View (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-catalog) to see it:

11. Add production configuration

Create a new file in the `src/main/resources` called `application-openshift.properties` in CodeReady Workspaces.

Be sure your new file is in the same directory alongside the existing `application-default.properties`!

Add the following content to this file:

```properties
# Production
server.port=8080
spring.datasource.url=jdbc:postgresql://catalog-database:5432/catalog
spring.datasource.username=catalog
spring.datasource.password=mysecretpassword
spring.datasource.initialization-mode=always
spring.datasource.initialize=true
spring.datasource.schema=classpath:/schema.sql
spring.datasource.continue-on-error=true

feign.hystrix.enabled=true
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=20000
inventory.ribbon.listOfServers=inventory.user16-inventory.svc.cluster.local:8080
```

We'll use this file by specifying a Spring *profile* when we deploy to OpenShift.

12. Build and Deploy

If you still have the local app running, stop it by typing `CTRL-C` in its Terminal.

Build and deploy the project using the following command in a Terminal:

```sh
mvn clean install spring-boot:repackage -DskipTests -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m1-labs/catalog
```

You should see a **BUILD SUCCESS** at the end of the build output.

Then deploy the project using the following command in the CodeReady Workspaces Terminal:

```sh
oc project user16-catalog && \
oc new-build registry.access.redhat.com/ubi8/openjdk-11 --binary --name=catalog-springboot -l app=catalog-springboot
```

And then start and watch the build, which will take about a minute to complete:

```sh
oc start-build catalog-springboot --from-file $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m1-labs/catalog/target/catalog-1.0.0-SNAPSHOT.jar --follow
```

Once the build is done, we'll deploy it as an OpenShift application and override the spring profile to use our *production* values.

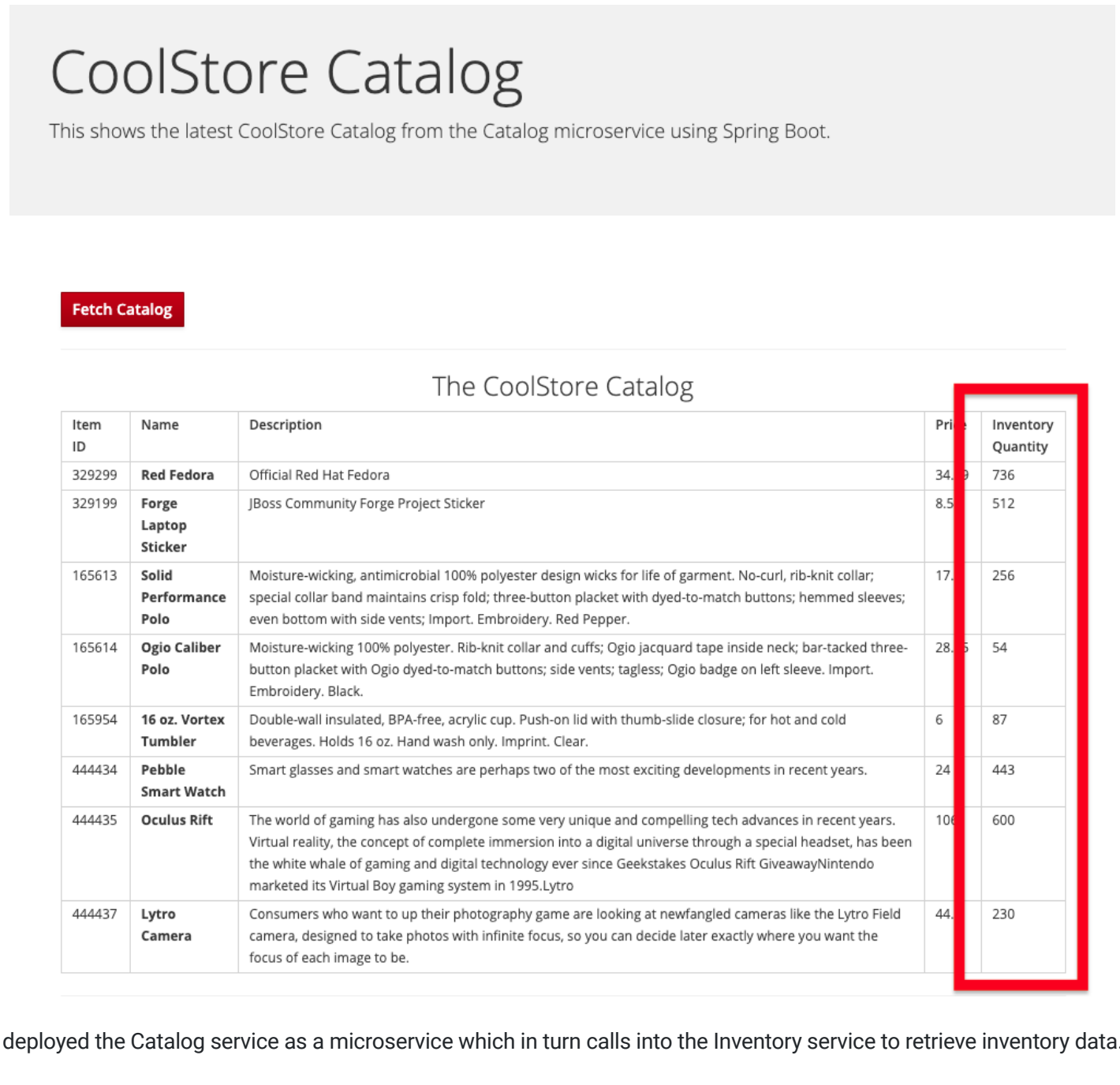
```sh
oc new-app catalog-springboot -e JAVA_OPTS_APPEND='-Dspring.profiles.active=openshift'
```

and run this to expose your service to the world and add a health check:

```sh
oc expose service catalog-springboot && oc set probe dc/catalog-springboot  --readiness --get-url=http://:8080 --initial-delay-seconds=5 --period-seconds=5 --failure-threshold=15
```

Finally, make sure it's actually done rolling out. Visit the Topology View (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-catalog) for the catalog, and ensure you get the blue circles!

And then access the Catalog Web frontend (http://catalog-springboot-user16-catalog.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com) and ensure you get the expected inventory quantity (and not `-1`):

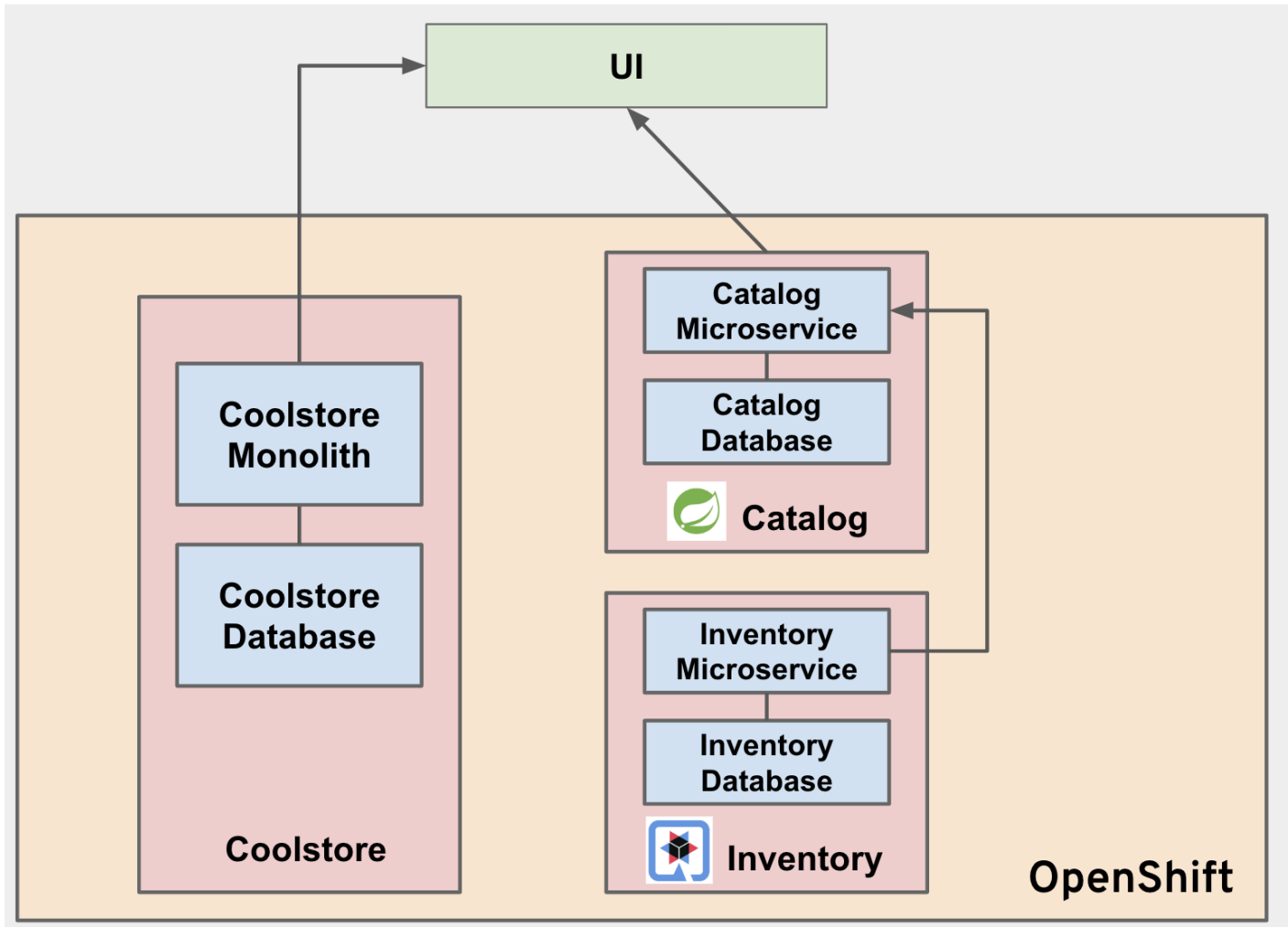


Congratulations! You have deployed the Catalog service as a microservice which in turn calls into the Inventory service to retrieve inventory data.

13. Strangling the monolith

So far we haven't started strangling the monolith (https://www.martinfowler.com/bliki/StranglerApplication.html). Each external request coming into OpenShift (unless using ingress, which we are not) will pass through a route. In our monolith the web page uses client side REST calls to load different parts of pages.

For the home page the product list is loaded via a REST call to `/services/products`. At the moment calls to that URL will still hit product catalog in the monolith. Now we will route these calls to our newly created catalog services instead and end up with something like:



Follow the steps below to create a **Cross-origin resource sharing (CORS)** based route. CORS is a mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served.

Quarkus comes with a *CORS filter* which implements the *javax.servlet.Filter* interface and intercepts all incoming HTTP requests. It can be enabled in the Quarkus configuration file. Add the following line in the `inventory` project (our Quarkus app created earlier) in the `src/main/resources/application.properties` file:

```PROPERTIES
%prod.quarkus.http.cors=true
```

Rebuild and redeploy the **inventory** application using this command (which will again use the OpenShift Quarkus extension to deploy):

```SH
oc project user16-inventory && \
mvn clean package -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m1-labs/inventory -DskipTests
```

This will take about a minute to complete. Once the build is done, the inventory pod will be deployed automatically via DeploymentConfig Trigger in OpenShift.

Open `CatalogEndpoint` class in `src/main/java/com/redhat/coolstore/service` of `catalog` project to allow restricted resources on a *product* page of the monolith application. Replace the class-level annotations with:

```JAVA
@CrossOrigin
@RestController
@RequestMapping("/services")
```

We simply added the `@CrossOrigin` annotation.

Rebuild and re-deploy the **catalog** service using the following commands:

```SH
mvn clean install spring-boot:repackage -DskipTests -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m1-labs/catalog && \
oc start-build -n user16-catalog catalog-springboot --from-file $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m1-labs/catalog/target/catalog-1.0.0-SNAPSHOT.jar --follow
```

This will take about a minute to complete. Once the build is done, the catalog pod will be deployed automatically via DeploymentConfig Trigger in OpenShift.

Let's update the catalog endpoint in monolith application. In the **monolith** 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`, just before the `factory.getProducts = function()` line:

```JAVASCRIPT
baseUrl='http://catalog-springboot-user16-catalog.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/services/products';
```

Rebuild and re-deploy the **monolith** project in CodeReady Workspaces Terminal:

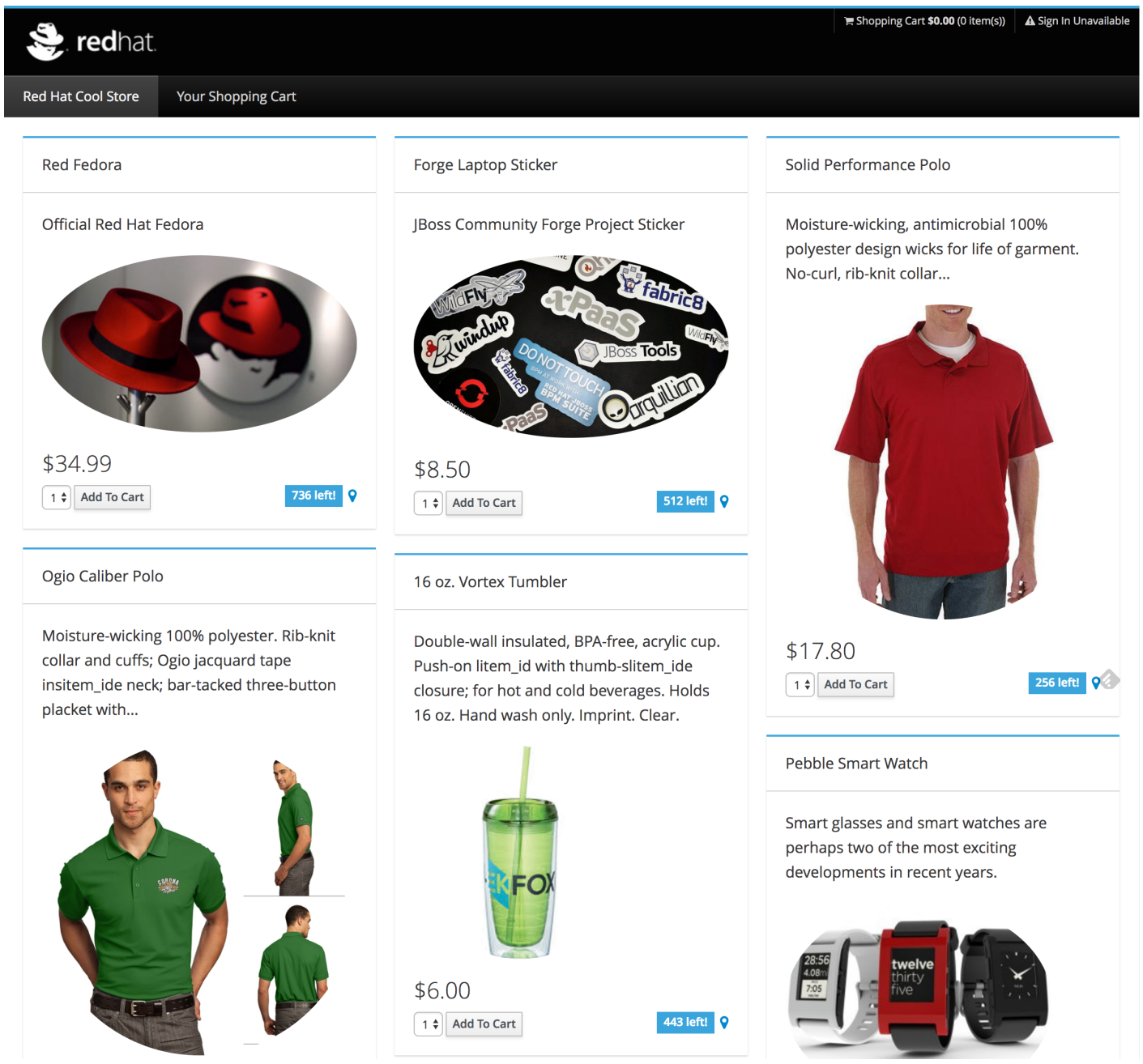
```sh
mvn clean package -DskipTests -Popenshift -f $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m1-labs/monolith && \
oc start-build -n user16-coolstore-dev coolstore --from-file $CHE_PROJECTS_ROOT/cloud-native-workshop-v2m1-labs/monolith/deployments/ROOT.war --follow
```

Once the build is done, the coolstore pod will be deployed automatically via DeploymentConfig Trigger in OpenShift. Ensure it's rolled out by visiting the Monolith Topology (https://console-openshift-console.apps.cluster-bbva-0efa.bbva-0efa.open.redhat.com/topology/ns/user16-coolstore-dev) and wait for the blue circles!

14. Test the UI

Open the monolith UI by clicking the route URL icon (the arrow to the upper right of the blue circle for the coolstore monolith)

Observe that the new catalog is being used along with the monolith:



The screen will look the same, with proper inventory, but notice that the earlier product *Atari 2600 Joystick* is now gone, as it has been removed in our new catalog microservice.

If the web page is still same then you should clean cookies and caches in your web browser.

Congratulations!

You have now successfully begun to *strangle* the monolith. Part of the monolith's functionality (Inventory and Catalog) are now implemented as microservices.

Summary

In this lab you learned a bit more about developing with Spring Boot and how it can be used together with OpenShift.

You created a new product catalog microservice representing functionality previously implemented in the monolithic CoolStore application. This new service also communicates with the inventory service to retrieve the inventory status for each product.
