The Containers and Cloud-Native Roadshow Dev Track

guides-m1-labs-infra.apps.cluster-beijing-5bc2.beijing-

Your Workshop Environment

The Workshop Environment You Are Using

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

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

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

How to complete this workshop

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

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

Good luck, and let's get started!

Getting Started with Cloud-Native Apps

In this module you'll work with an existing Java EE application for a retail webshop and migrate it and modernize it. The current version of the webshop is a Java EE application built for Oracle Weblogic Application Server and as part of a modernization strategy and to be able to automate releases and eventually apply advanced deployment scenarios (like Canary or Blue/Green deployments) there has been a decision to move this application to a container native environment from Red Hat called OpenShift.

During the first planning sprint you have investigated the possibility to deploy Oracle Weblogic to Red Hat OpenShift, but since Oracle Weblogic is limited in support and also not recommended to use in an orchestrated container native platform like OpenShift you have two options. Either you migrate to WebSphere Liberty Profile or you migrate to JBoss EAP.

Charateristics	Oracle Weblogic	JBoss EAP
Supported on OpenShift	Limited	Yes
Supports Java EE 8 Web Profile	Yes	Yes
Supports Java EE 8	Yes	Yes
Has an S2I build image	No	Yes
Has a container image that is supported by the vendor	No	Yes
Autoconfiguration of data sources and messaging	No	Yes
Autoconfiguration of SSO	No	Yes
Capable of handling transaction recovery in an ephemeral container	Unknown	Yes
Adapted for clustering in Kubernetes (session failover, etc)	No	Yes
Single vendor support	Yes	Yes
Support and maintenance cost	High	Included
Migration effort	High	Low

After the investigation the team agrees that JBoss EAP seems to be the much better choice with better support for running your application in OpenShift, but the last question mark on “Migration cost” is worrying. You decide to contact Red Hat to try to find out what the migration cost might actually be. Red Hat uses a tool called Red Hat Application Migration Toolkit (RHAMT) which help customers analyze their applications and report on the estimated effort to migrate and also provides detailed instructions on how to actually migrate the code.

What is 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

RHAMT examines application artifacts, including project source directories and application archives, then produces an HTML report that highlights areas needing changes. RHAMT can be used to migrate Java applications from previous versions of Red Hat JBoss Enterprise Application Platform or from other middleware, such as Oracle® WebLogic Server or IBM® WebSphere® Application Server.

How Does Red Hat Application Migration Toolkit Simplify Migration?

Red Hat Application Migration Toolkit looks for common resources and highlights technologies and known trouble spots when migrating applications. The goal is to provide a high-level view into the technologies used by the application and provide a detailed report organizations can use to estimate, document, and carry out migration of enterprise applications to Java EE and Red Hat JBoss Enterprise Application Platform.

> RHAMT is usually part of a much larger application migration and modernization program that involves well defined and repeatable phases over weeks or months and involves many people from a given business. Do not be fooled into thinking that every single migration is a simple affair and takes an hour or less! To learn more about Red Hat's philosophy and proven methodology, check out the RHAMT documentation and contact your local Red Hat representative when embarking on a real world migration and modernization strategy.

More RHAMT Resources

- Documentation
- Developer Homepage

What comes after migration?

You might wonder about developing **new** cloud-native apps once you complete your migration of existing apps via RHAMT.

Red Hat Runtimes provides a curated, best-of-breed collection of cloud native runtimes that let you match application requirements to the best runtime, and allows your organization to standardize on a set of runtimes to support the business. Fully supported runtimes include:

We will use several of these runtimes during the course of this workshop. After this workshop you can try these on your own using our self-paced, in-browser learning guides.

Let's get started!

Decide which Application Server to use in OpenShift

Lab1 - Decide which Application Server to use in OpenShift

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.

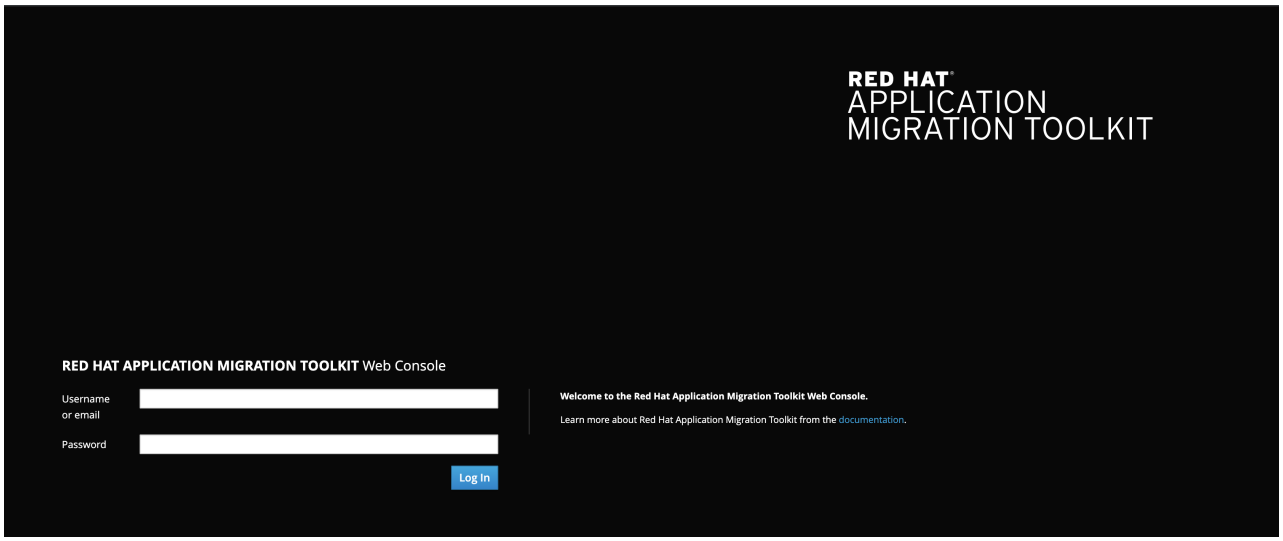
The Red Hat Application Migration Toolkit can be installed and used in a few different ways:

- `Web Console` - The web console for Red Hat Application Migration Toolkit is a web-based system that allows a team of users to assess and prioritize migration and modernization efforts for a large number of applications. It allows you to group applications into projects for analysis and provides numerous reports that highlight the results.
- `Command Line Interface` - The CLI is a command-line tool that allows users to assess and prioritize migration and modernization efforts for applications. It provides numerous reports that highlight the analysis results.
- `Eclipse Plugin` - The Eclipse plugin for Red Hat Application Migration Toolkit provides assistance directly in Eclipse and Red Hat JBoss Developer Studio for developers making changes for a migration or modernization effort. It analyzes your projects using RHAMT, marks migration issues in the source code, provides guidance to fix the issues, and offers automatic code replacement when possible.

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

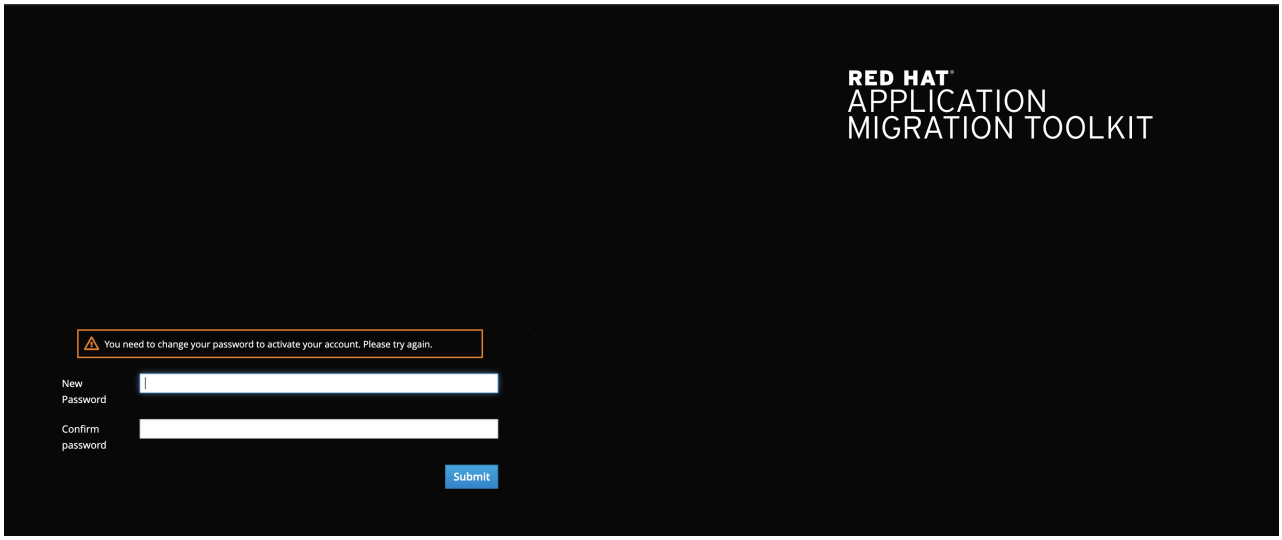
1. Login the RHAMT web console in OpenShift cluster

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

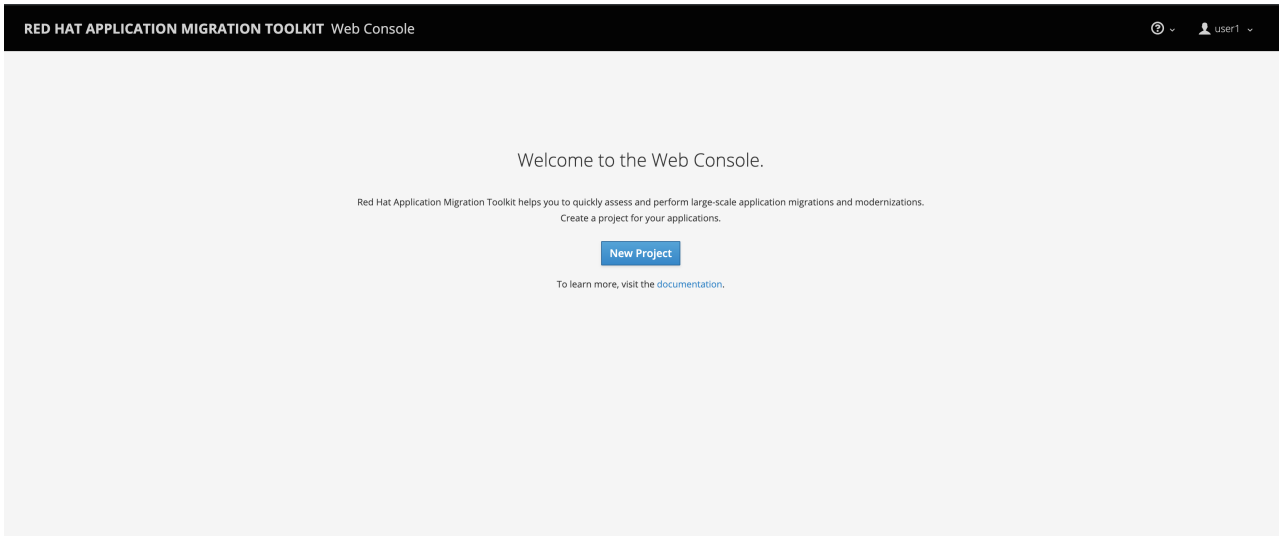


When you login the first time, you should be asked to change the password in order to comply with RH SSO policy.

> `NOTE` : You can use the current password to input a new password.



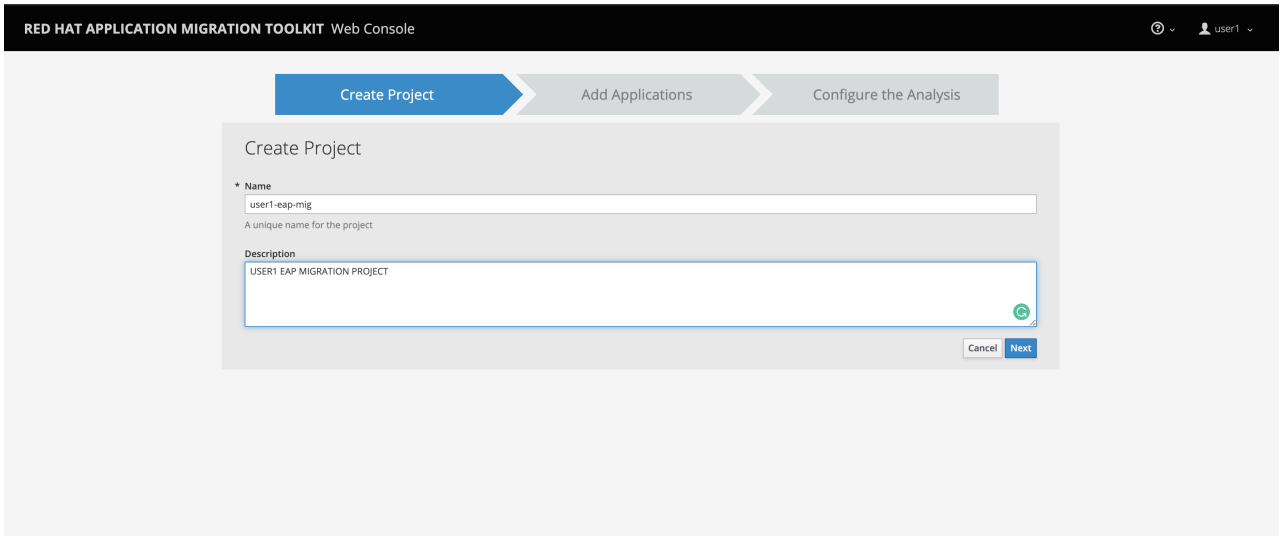
2. Create a new project



Input a name and description to create a project.

- Name: `userXX-eap-migration`
- Description: `USERXX EAP MIGRATION PROJECT`

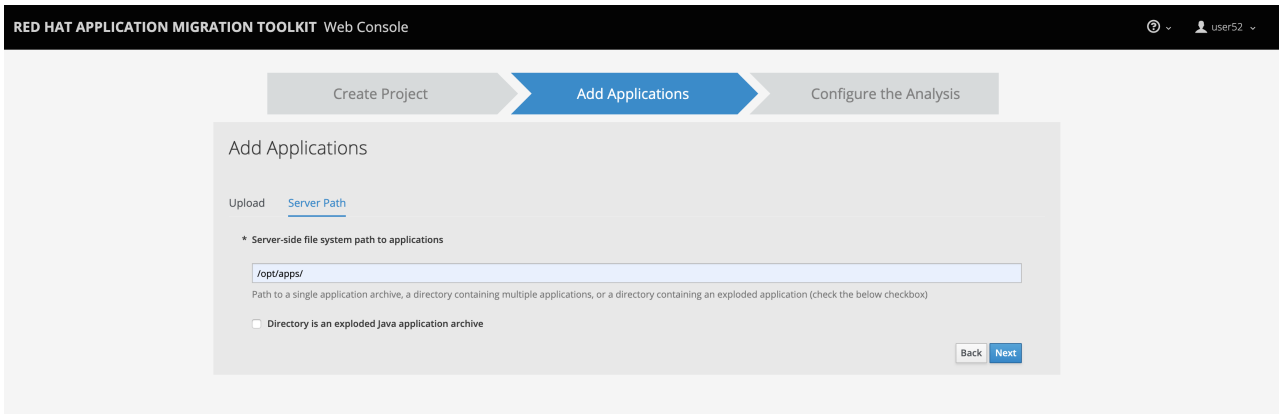
> `NOTE` : Add your username as prefix (i.e. `user1-eap-migration`) to distinguish each attendee's project.



3. Add the monolith application to the project

Select `Server Path` to analyze a monolithic application:

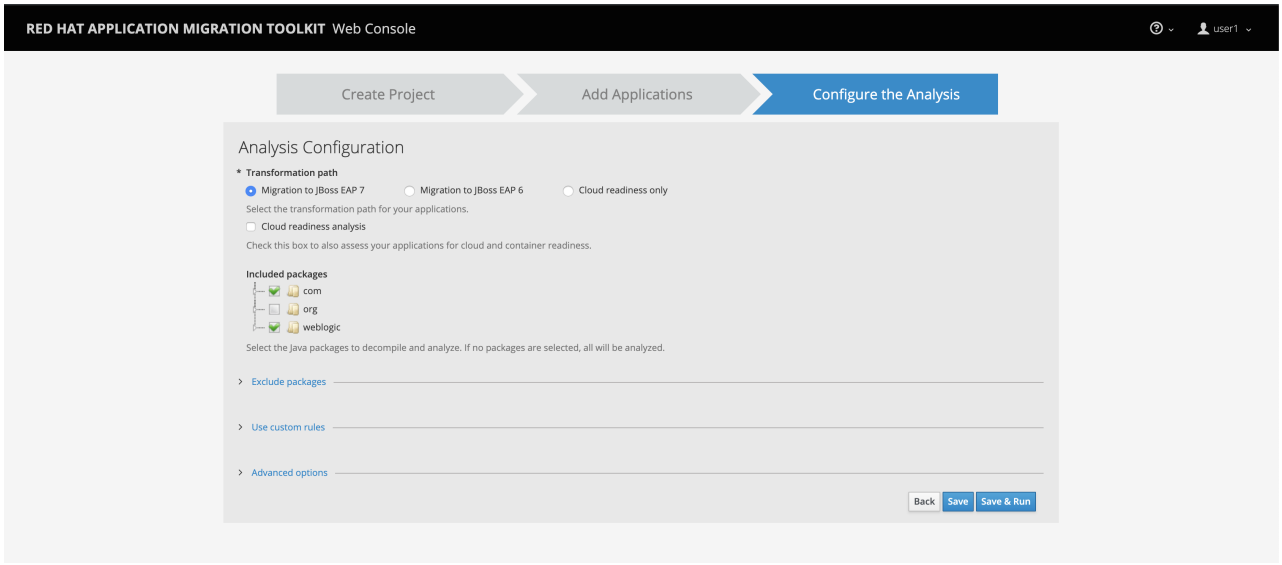
Server Path: `/opt/apps`



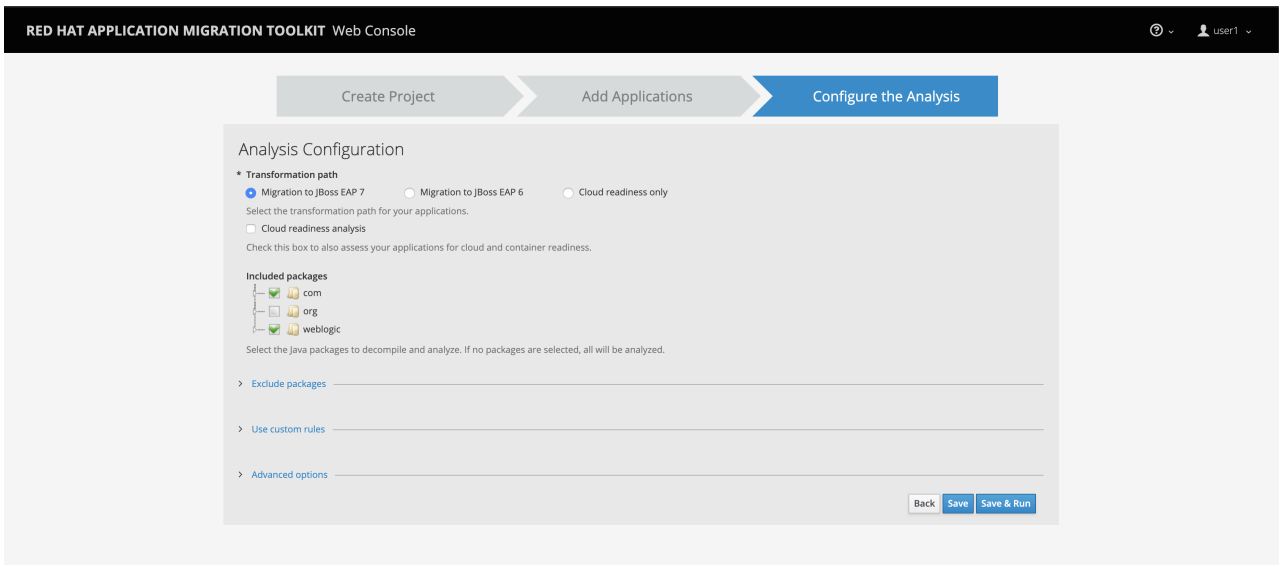
4. Select “Migration to JBoss EAP 7” in Transformation Path

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.

Check `com, weblogic` for Included packages.



Tip : This page may look just like an OpenShift console page, its not, its the Red Hat Application Migration Toolkit console - both project use PatternFly for a consistent web look and feel.

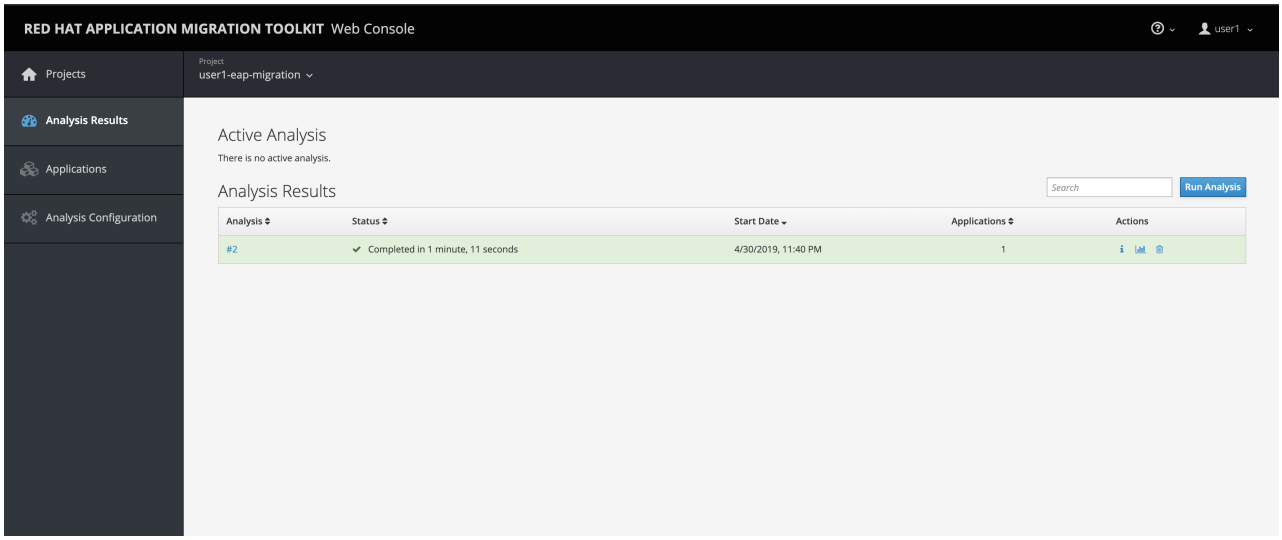


Click Save & Run

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

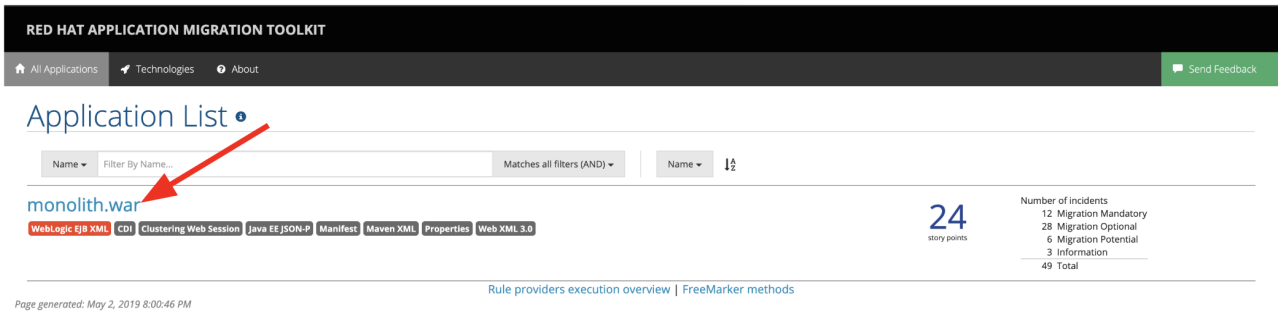
---

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



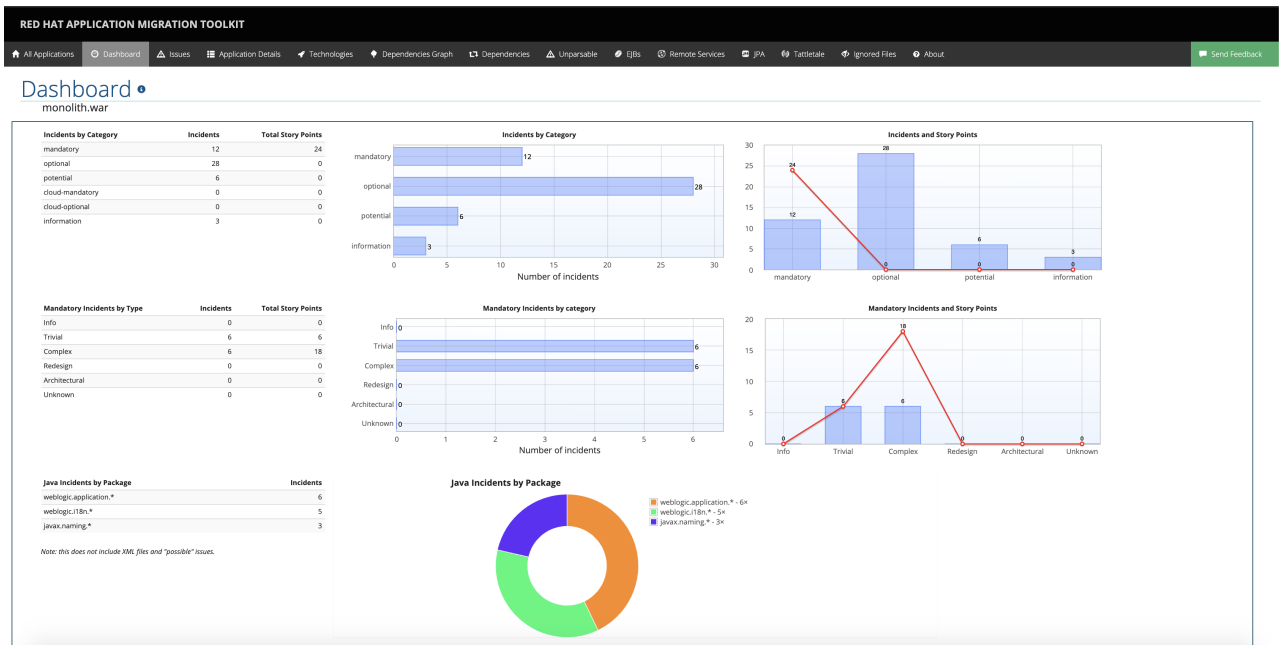
6. Review the report

You should see the landing page for 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.

There are several other sub-pages accessible by the menu near the top. Click on each one and observe the results for each of these pages:

- `All Applications` Provides a list of all applications scanned.
- `Dashboard` Provides an overview for a specific application.
- `Issues` Provides a concise summary of all issues that require attention.
- `Application Details` provides a detailed overview of all resources found within the application that may need attention during the migration.
- `Unparsable` shows all files that RHAMT could not parse in the expected format. For instance, a file with a .xml or .wsdl suffix is assumed to be an XML file. If the XML parser fails, the issue is reported here and also where the individual file is listed.
- `Dependencies` displays all Java-packaged dependencies found within the application.
- `Remote Services` Displays all remote services references that were found within the application.
- `EJBs` contains a list of EJBs found within the application.
- `JBPM` contains all of the JBPM-related resources that were discovered during analysis.
- `JPA` contains details on all JPA-related resources that were found in the application.
- `About` Describes the current version of RHAMT and provides helpful links for further assistance.

> Some of the above sections may not appear depending on what was detected in the project.

You also investigate the report to see if there are any complex migrations.

After analysing the results and the effort you can estimate the amount of time the effort to migrate your application. In this case you may estimate that given the 24 story points and that most of the changes are trivial the effort will be approximately a week, which is about the same time it would take to migration from Oracle Weblogic to WebSphere Liberty Profile. You may feel comfortable to claim `Low Migration effort` for this app when you present your recommendation to the team and the architectural board. As a result, let us assume today that your recommendation to migrate to JBoss EAP to deploy your application on OpenShift is approved.

For the first sprint you will focus on migrating the application to JBoss EAP, the DBA will migrate the database to a PostgresQL instance on OpenShift while the ops engineer setup a project for you in the OpenShift cluster.

Migrate to JBoss EAP

Lab2 - Migrate your application to JBoss EAP

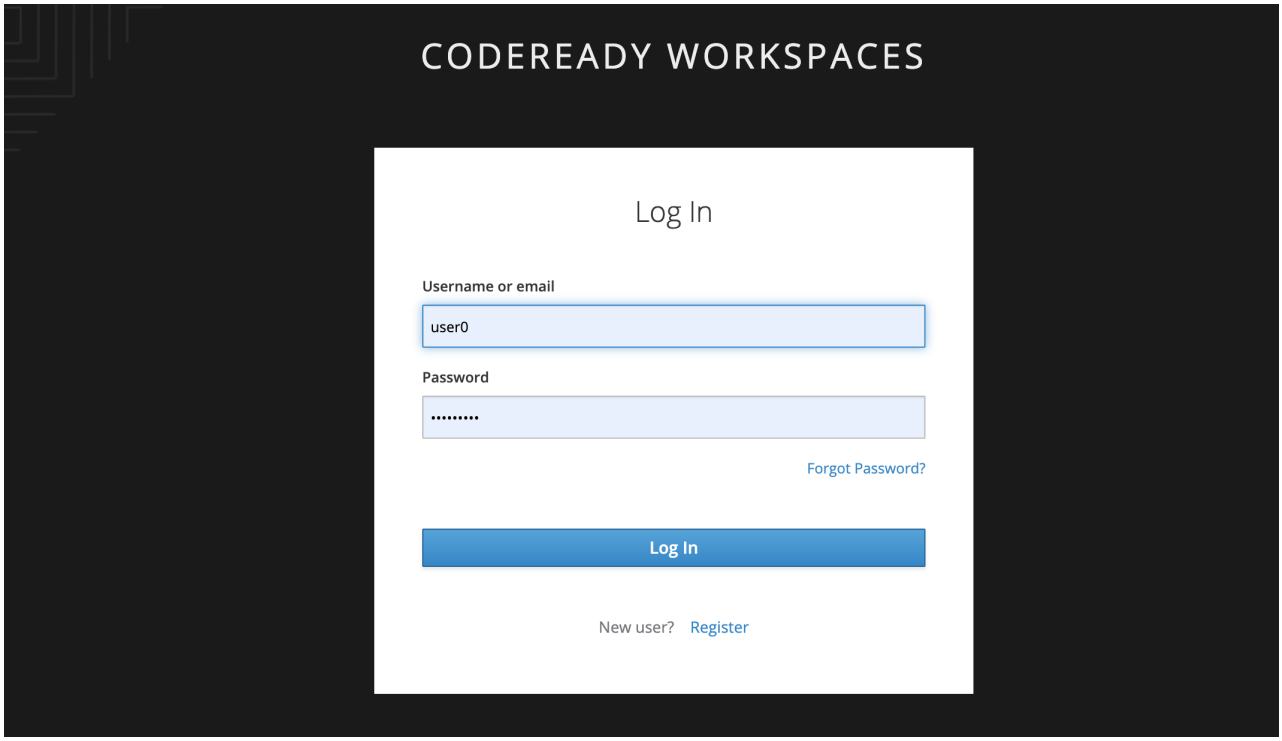
In this step you will migrate some Weblogic-specific code in the app to use standard Java EE interfaces.

1. Getting Ready for the labs

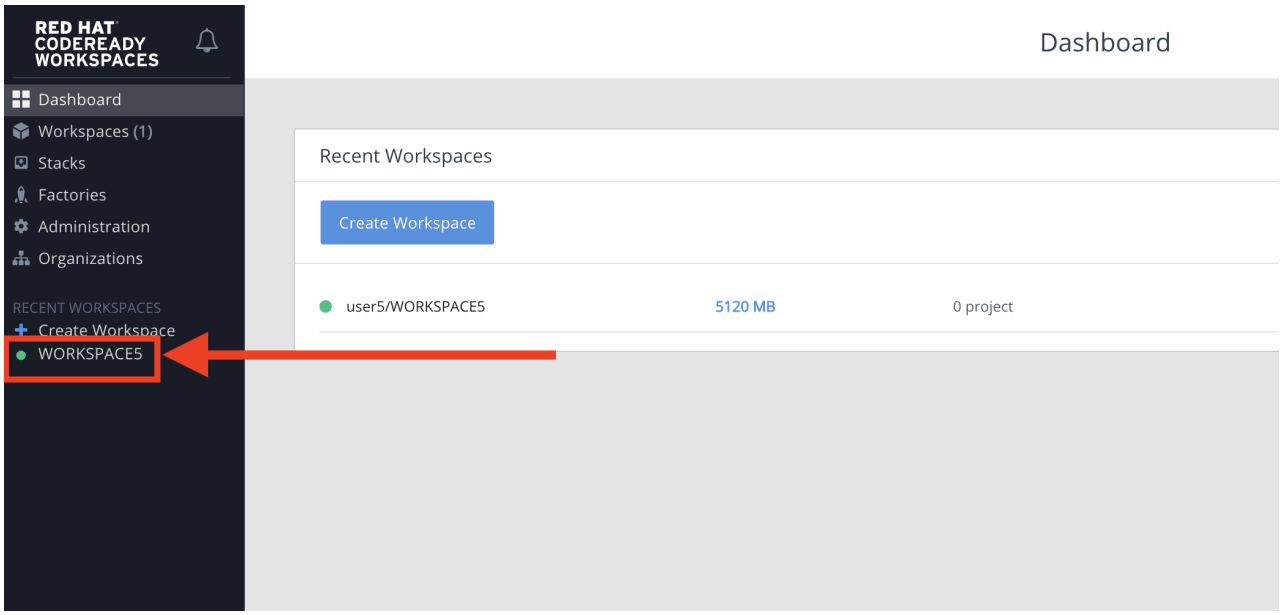
Access Your Development Environment

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

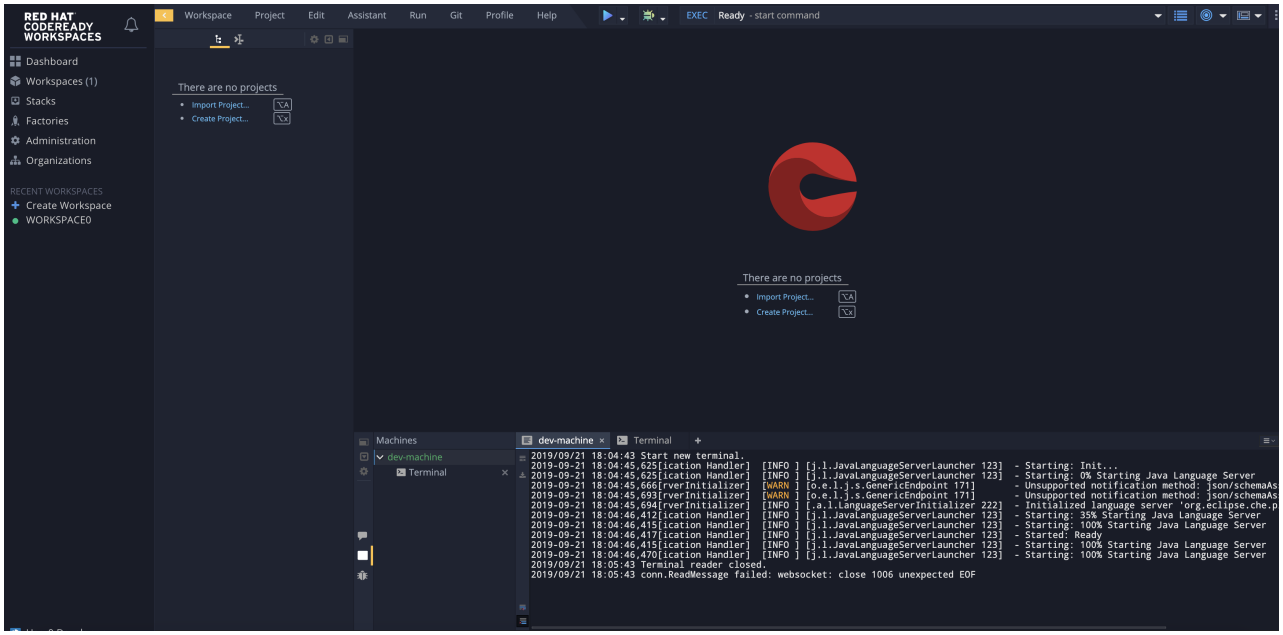
To get started, access the Che instance and log in using the username and password you've been assigned (e.g. `userXX/r3dh4t1!`):



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

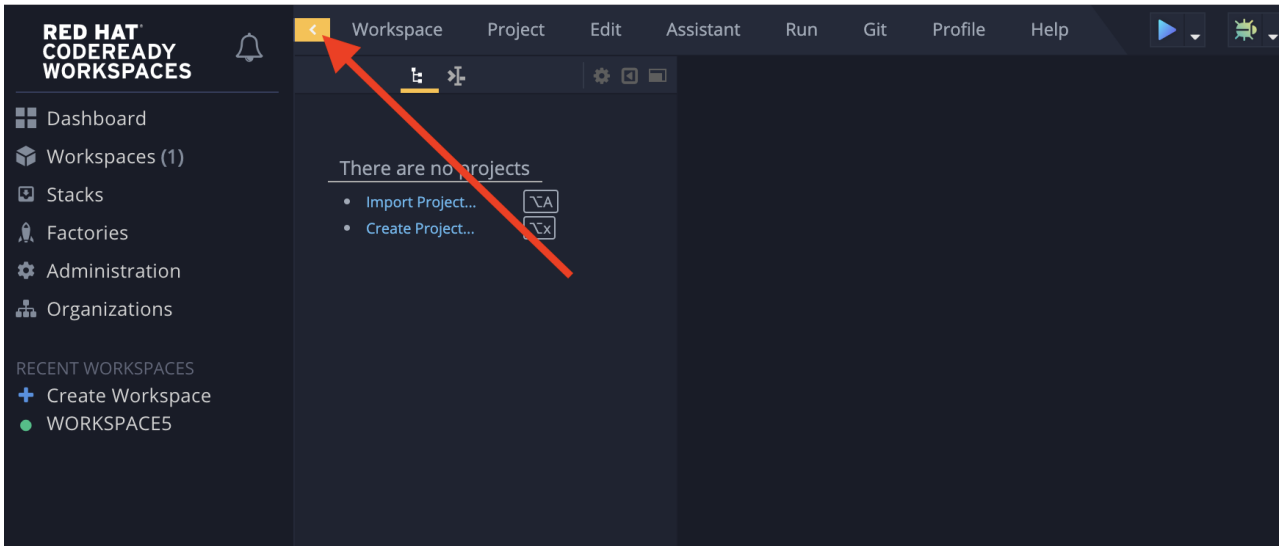


After a minute or two, you'll be placed in the workspace:



> **NOTE**:
>
> You may see random errors about websocket connections, plugins failing to load or other errors in the `dev-machine` window. You can ignore them as these are known issues that do not affect this workshop.

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

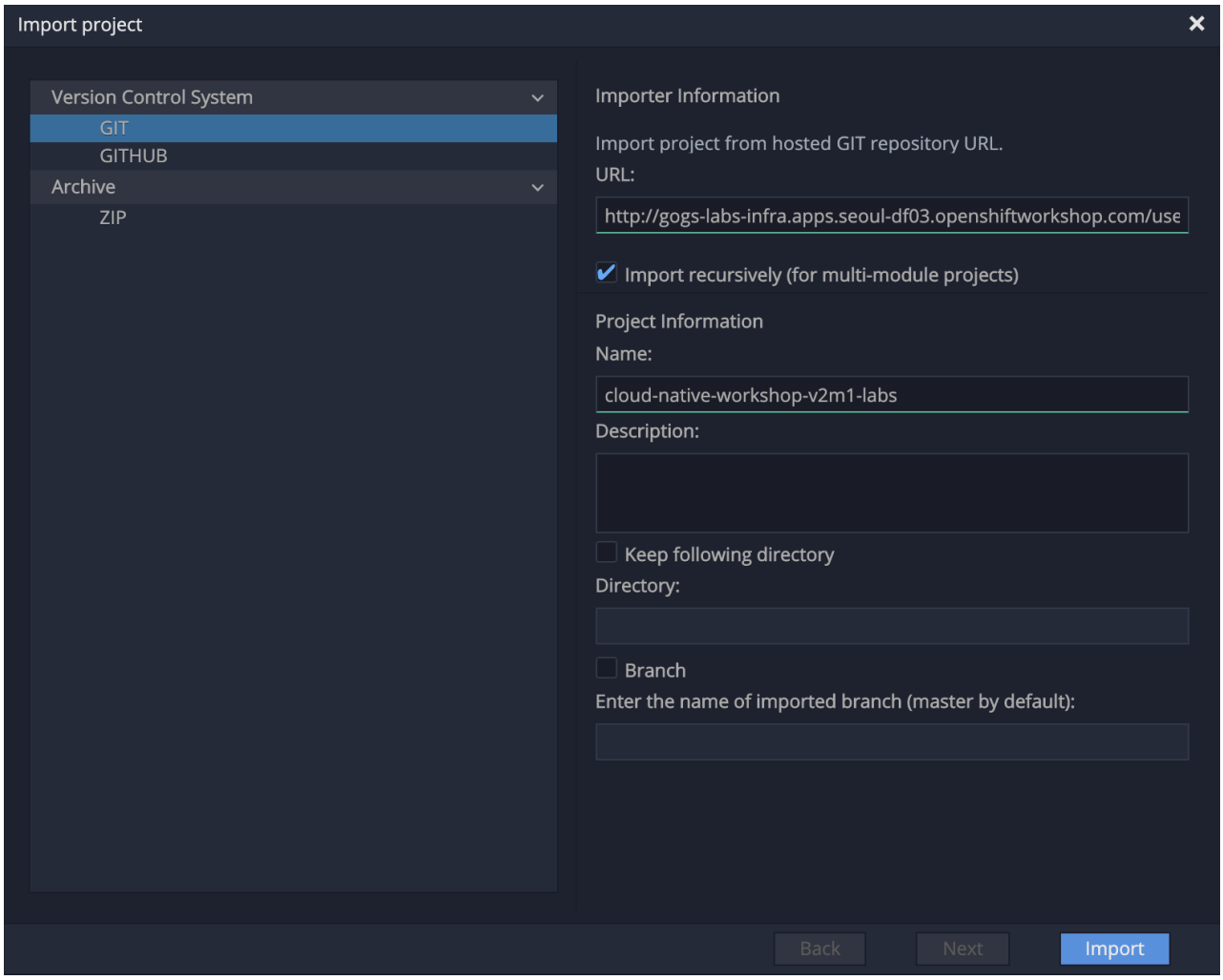


Users of Eclipse, IntelliJ IDEA or Visual Studio Code will see a familiar layout: a project/file browser on the left, a code editor on the right, and a terminal at the bottom. You'll use all of these during the course of this workshop, so keep this browser tab open throughout. **If things get weird,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

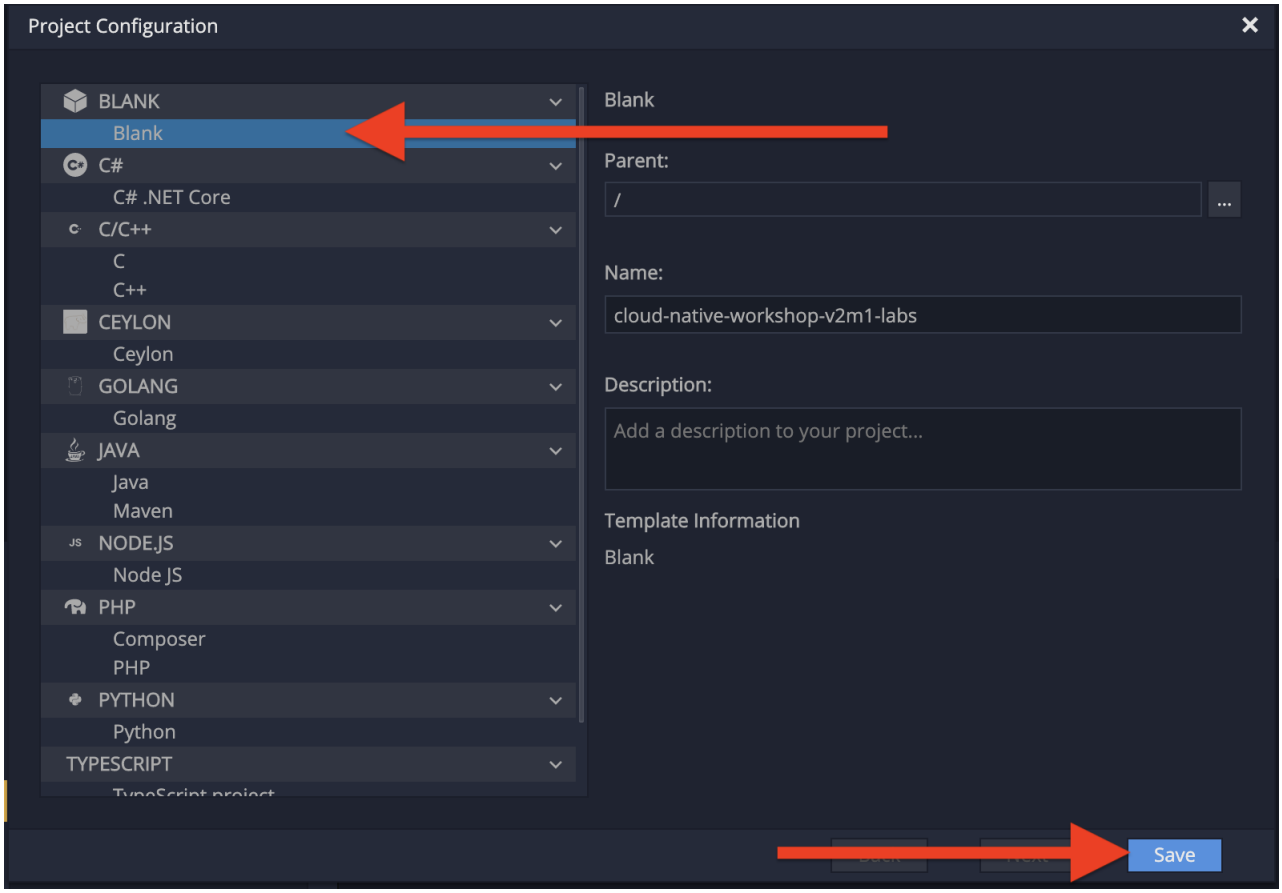
In the project explorer pane, click on the `Import Projects...` and enter the following:

- Version Control System: `GIT`

- URL: `http://gogs-labs-infra.apps.cluster-beijing-5bc2.beijing-5bc2.open.redhat.com/userXX/cloud-native-workshop-v2m1-labs.git` (IMPORTANT: replace userXX with your lab user)
- Check `Import recursively (for multi-module projects)`
- Name: `cloud-native-workshop-v2m1-labs`



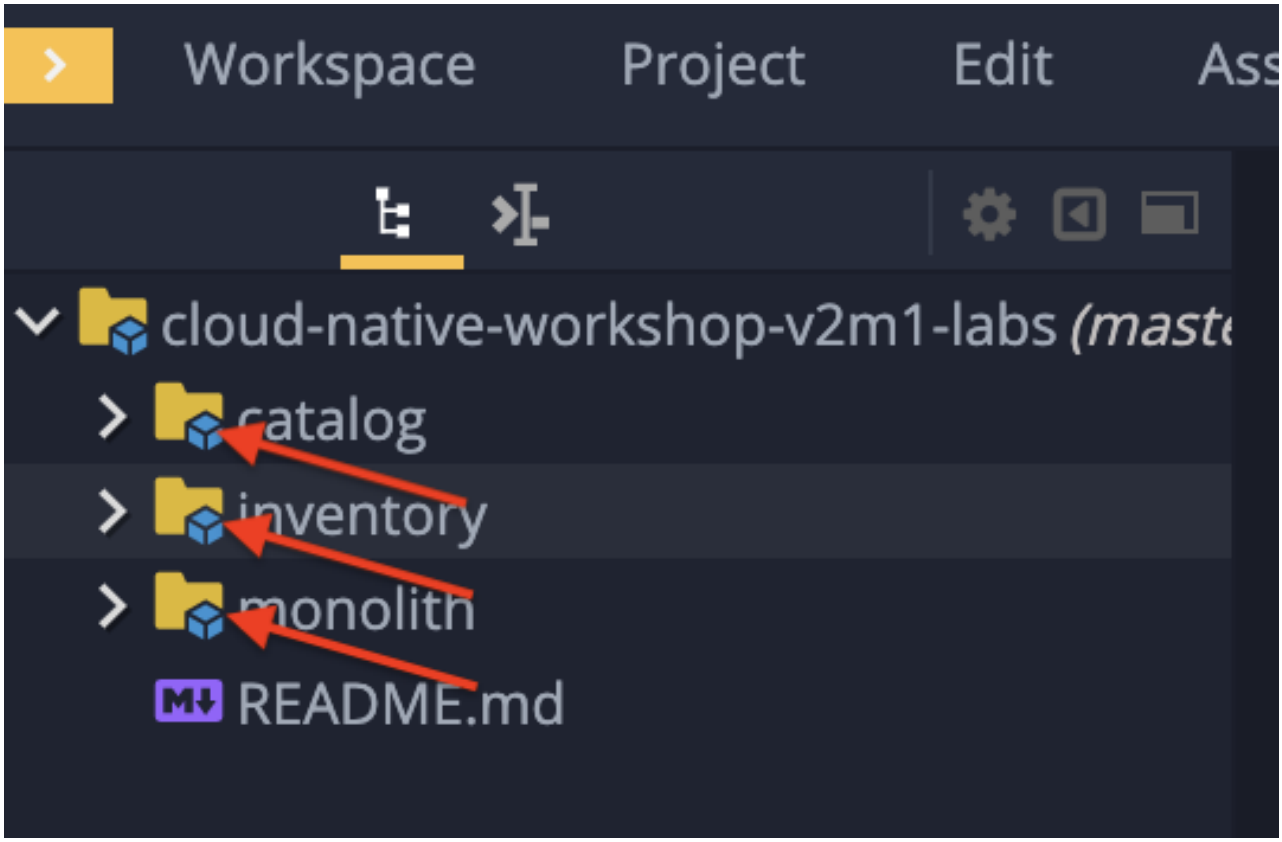
At the next screen, leave the project type set to `Blank` and click **Save**.



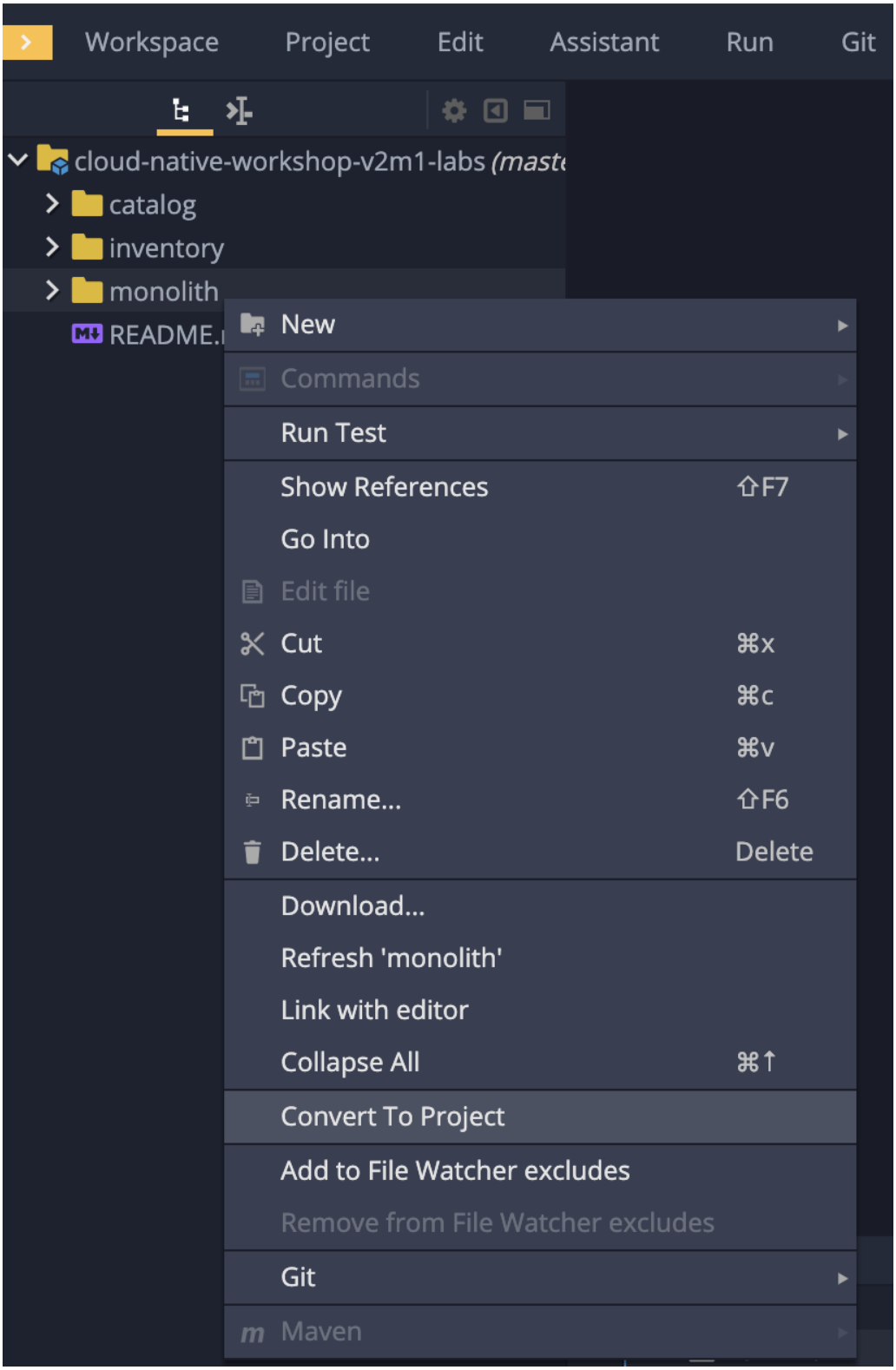
The project is imported into your workspace and is visible in the project explorer.

Convert Projects

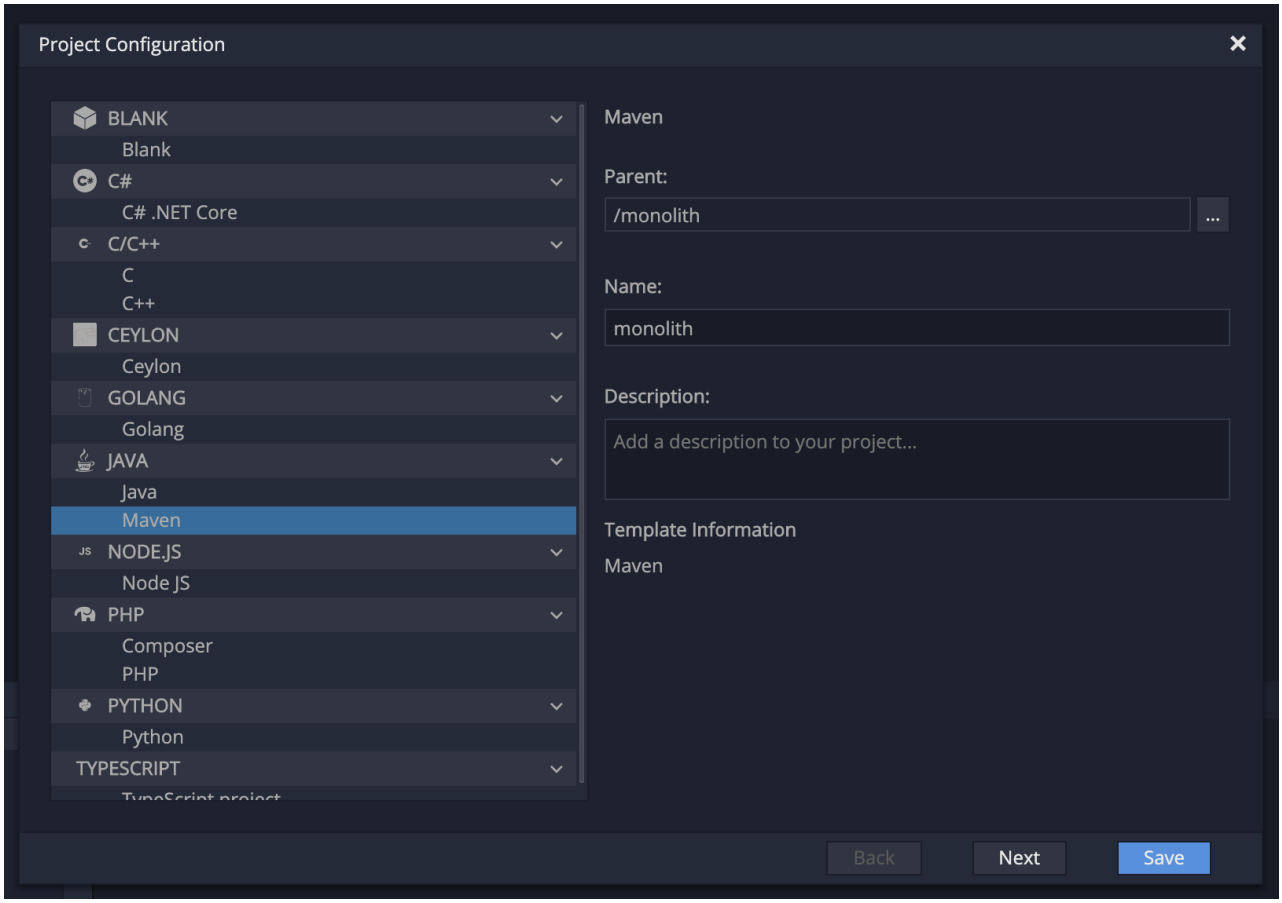
Expand the top-level project and look carefully at the icons next to each of the `monolith`, `catalog` and `inventory` directories. **Do you see a blue Maven icon as shown below?**



If you do **not** see these icons, then you'll need to right-click on each of the projects, and select "Convert to Project" and convert them to the *Maven* type project as shown below:

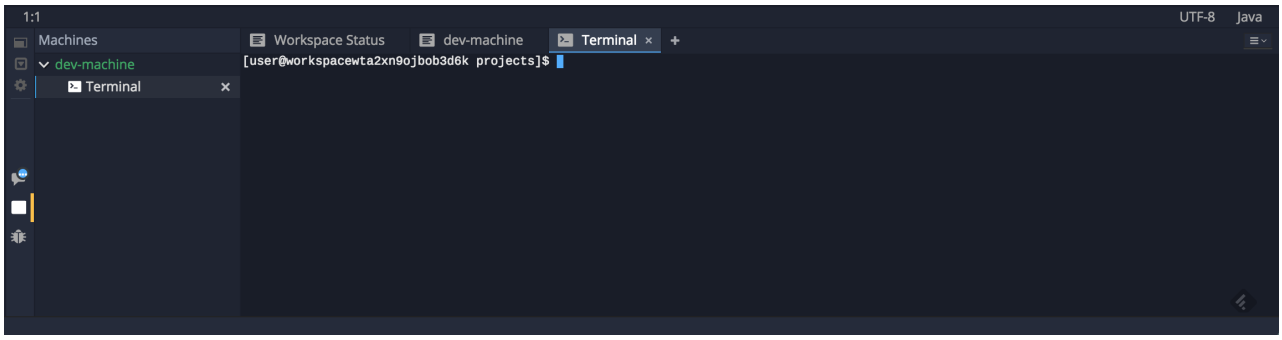


Choose **Maven** from the project configurations and then click on **Save**.



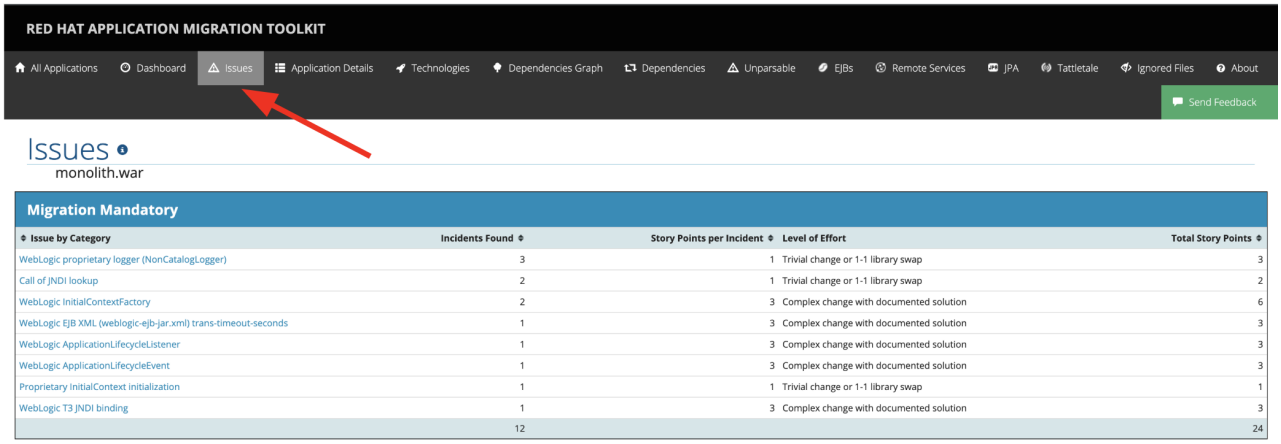
Be sure to do this for each of the `monolith`, `inventory` and `catalog` projects.

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



2. Review the issue related to `ApplicationLifecycleListener`

Open the Issues report in the RHAMT Console:



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 at Oracle WebLogic 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 JBoss Enterprise Application Platform, 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.

While the code in our startup and shutdown is very simple, in the real world this code may require additional thought as part of the migration. However, 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 this folder in the project tree navigation pane to the left side, and edit the source files under there.

Open the file `src/main/java/com/redhat/coolstore/utils/StartupListener.java`. Navigate the folder tree and double-click the source file to open it in the editing panel.

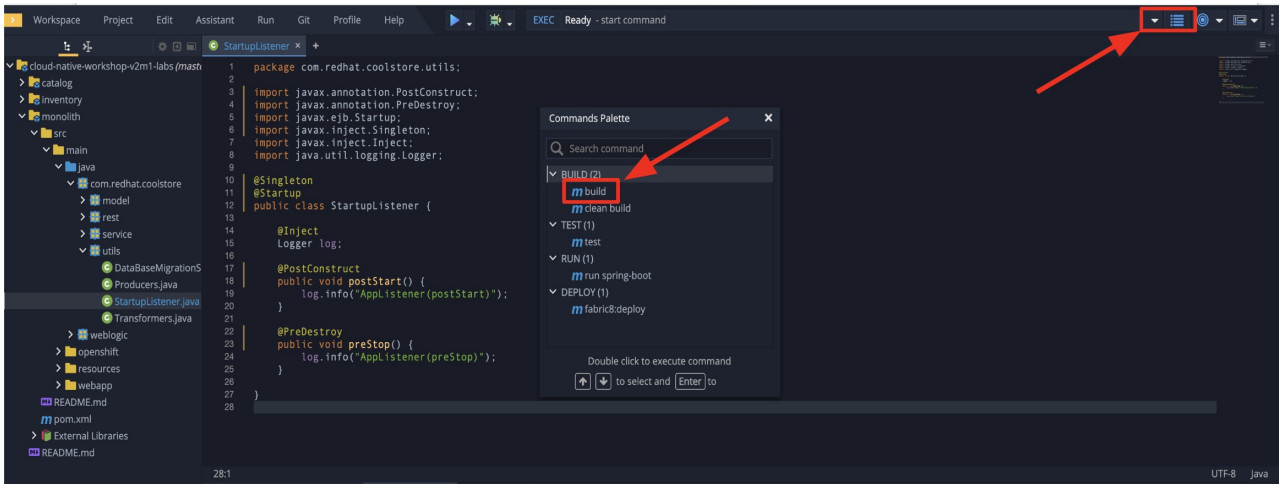
The first issue we will tackle is the one reporting the use of *Weblogic ApplicationLifecyleEvent* and *Weblogic LifecycleListener* in this file. Open the file to make these changes in the file. Replace the file so it is as follows:

```
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)");
    }

}
```

`Tip`: 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` (`CMD-Z` on Mac) or by using the `Edit -> Undo` menu option.

4. Test the build

Go to `Commands Palette` and dobule-click on `build` in CodeReady Workspaces:



If it 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.

In the next step, we will migrate some Weblogic-specific code in the app to use standard Java EE interfaces.

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

In this case we are using Weblogic's `NonCatalogLogger` which is a simplified logging framework that doesn't use localized message catalogs (hence the term *NonCatalog*).

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 if needed.

5. Make the changes

Navigate to the `Monolith Folder` and work on the source files under here.

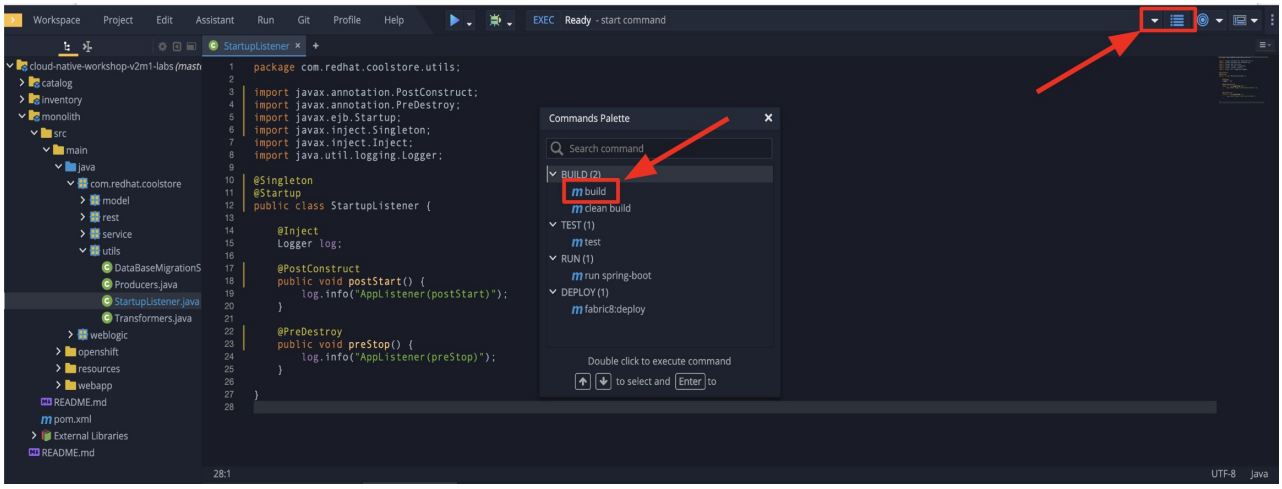
Open the `src/main/java/com/redhat/coolstore/service/OrderServiceMDB.java` file and replace its contents with:

```
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 using Maven to make sure you code still compiles via CodeReady Workspaces `BUILD` window:



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.

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 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, 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 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

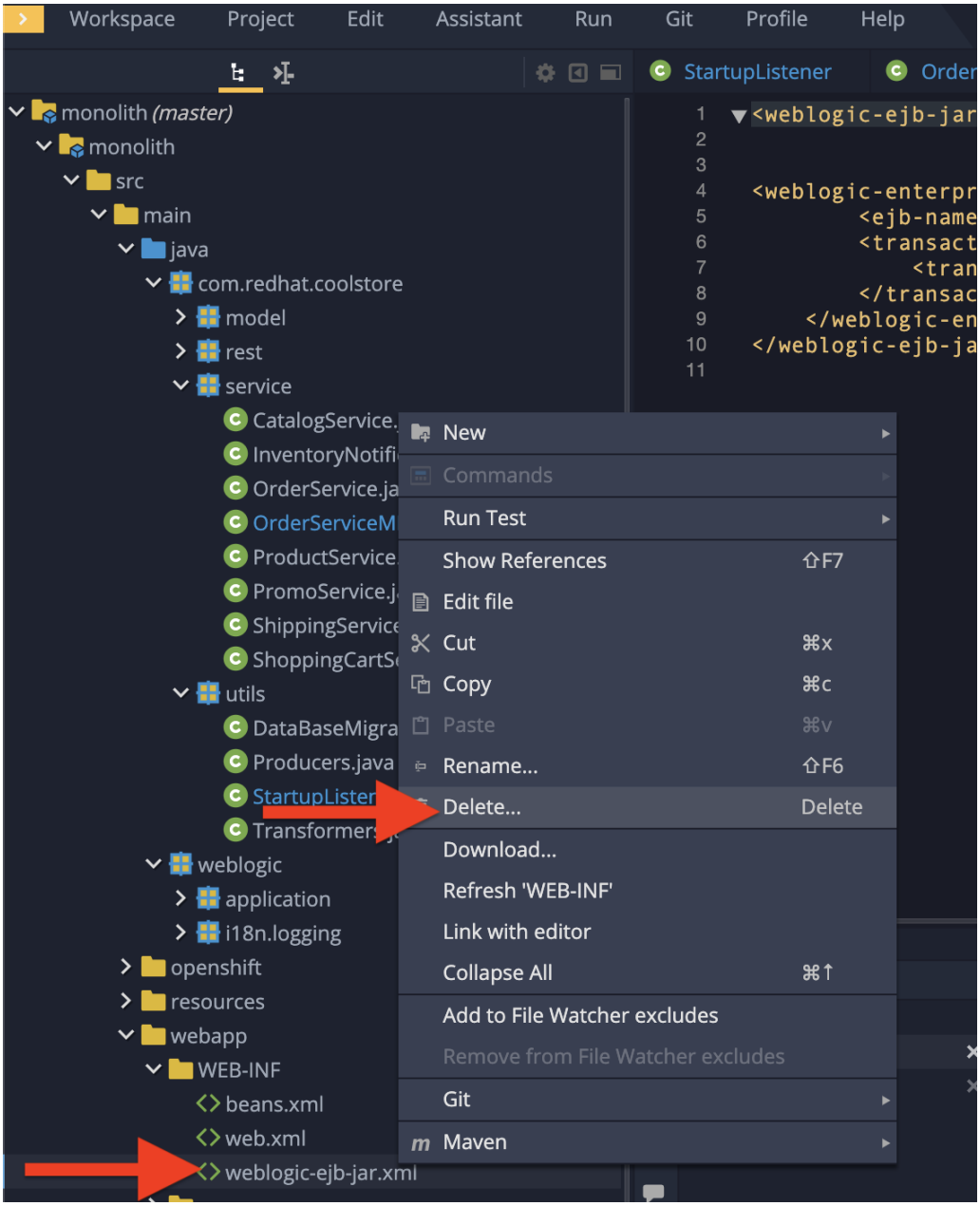
From the RHAMT Issues report, we will fix the remaining issues:

- `Call of JNDI lookup` - Our apps use a weblogic-specific JNDI 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.

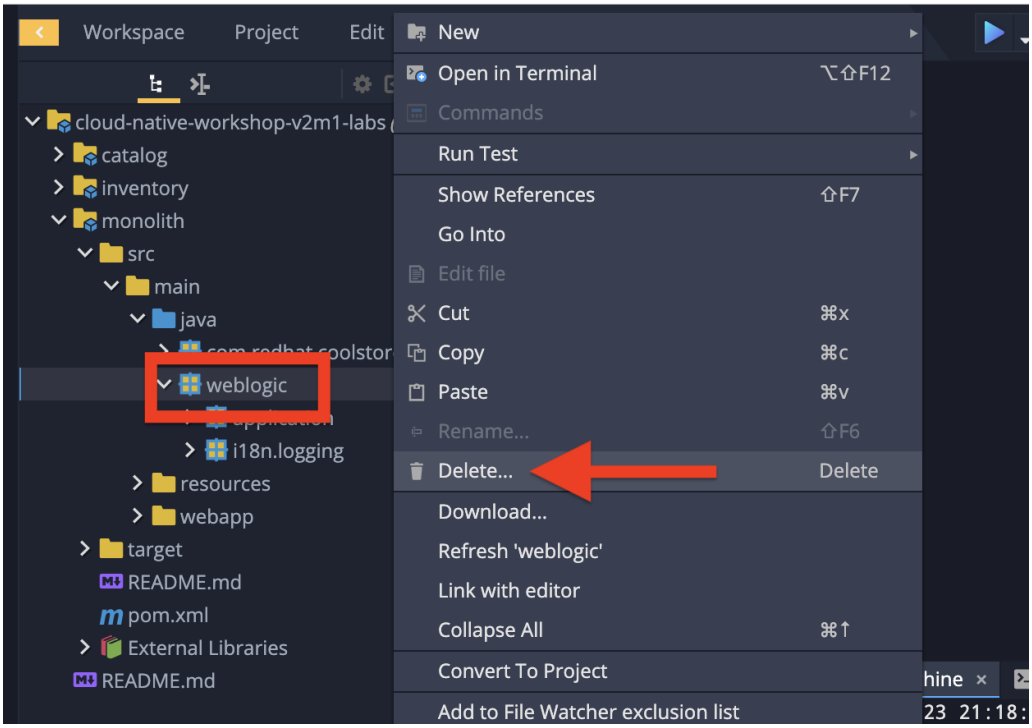
8. Remove the weblogic EJB Descriptors

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 on Eclipse Navigator:



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

Right-click on the `weblogic` folder and select **Delete** to delete the folder:



9. Fix the code

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

```
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
```

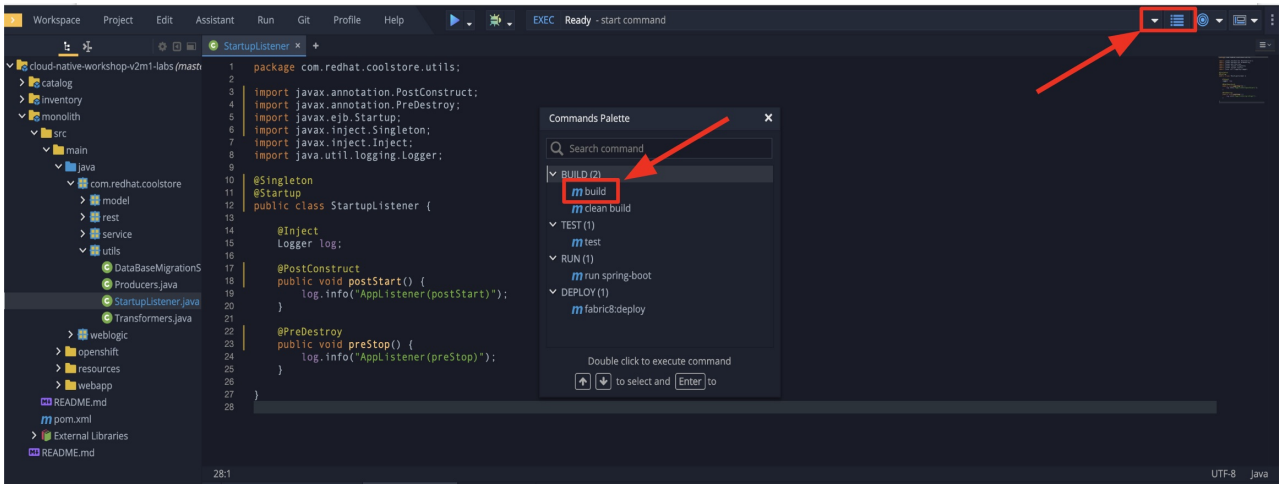
```
    private CatalogService catalogService;

    @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 and package the app using Maven to make sure you code still compiles via CodeReady Workspaces `BUILD` window:

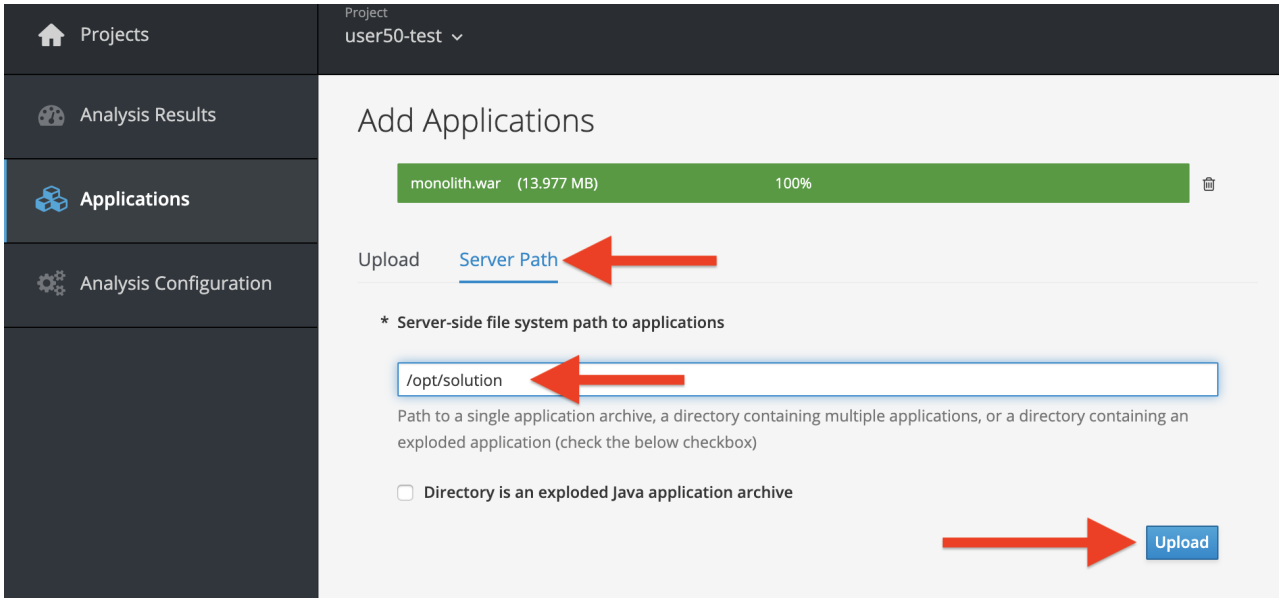


If builds successfully (you will see `BUILD SUCCESS`), then let's move on to the next issue! If it does not compile, verify you made all the changes correctly and try the build again.

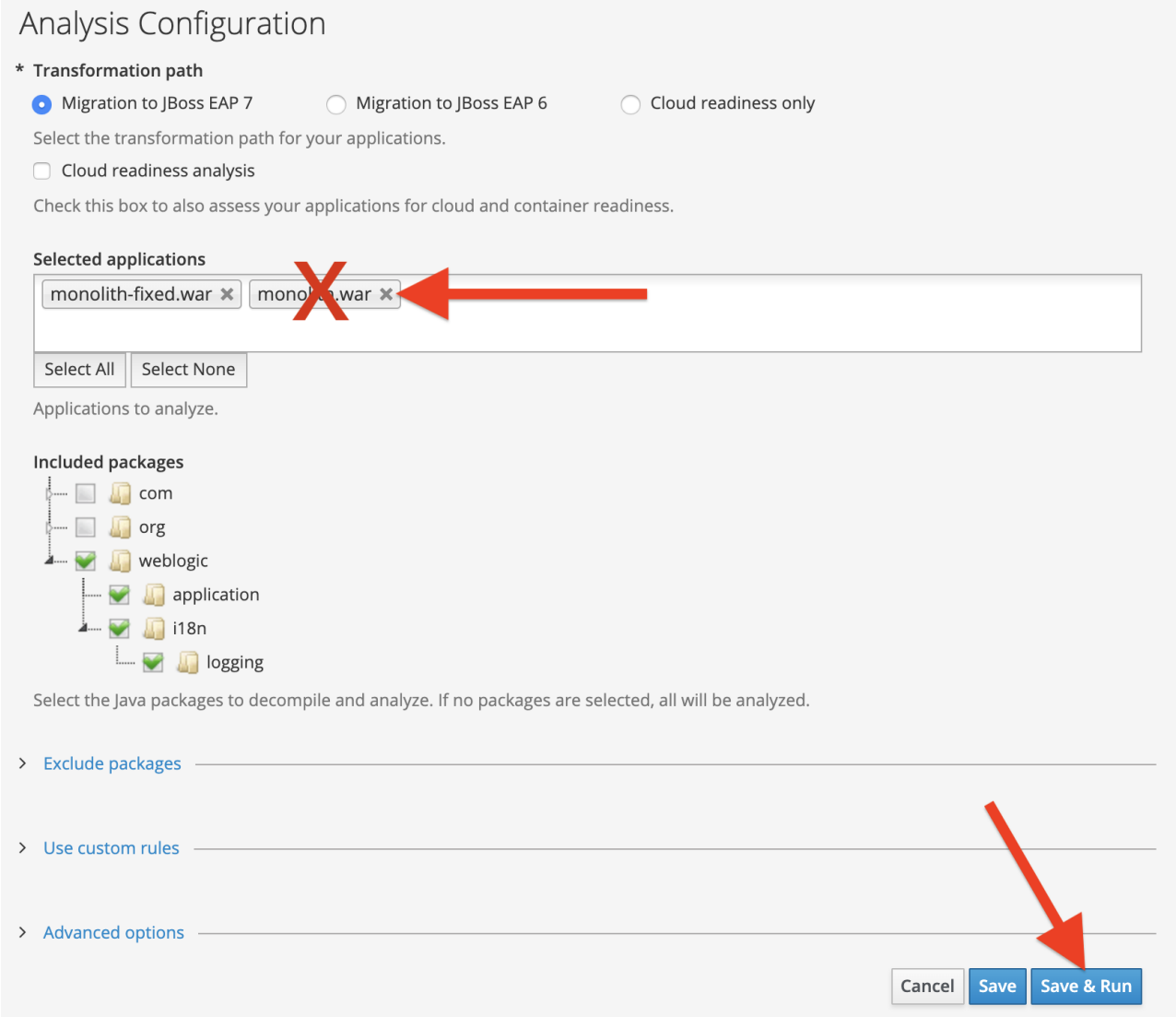
11. Re-run the RHAMT report

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

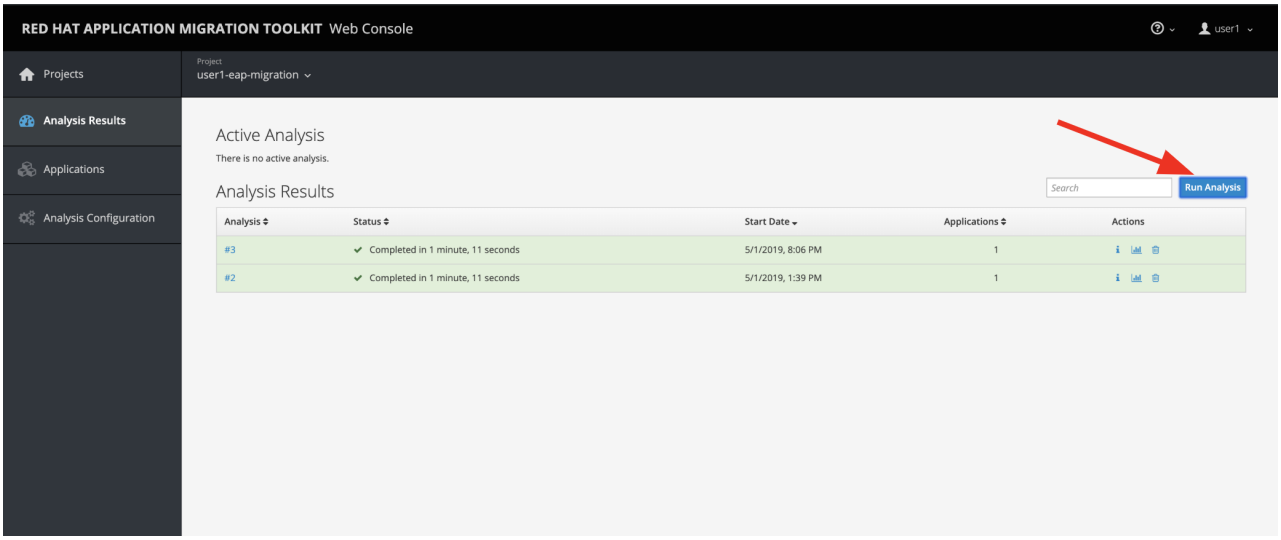
In the RHAMT Console, navigate to `Applications` on the left menu and click on `Add`. Enter the path to the fixed project at `/opt/solution` and click **Upload** to add the project:



Be sure to delete the old `monolith.war` to avoid analyzing it again:



and then click **Save and Run** to analyze the project:

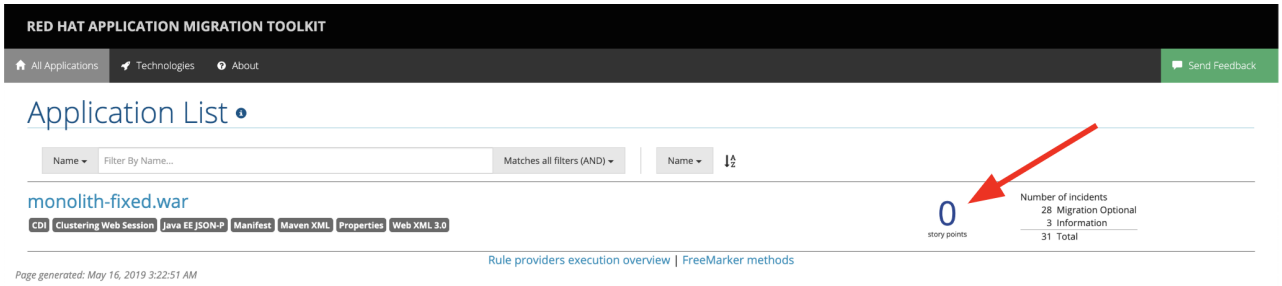


Depending on how many other students are running reports, your analysis might be *queued* for several minutes. If it is taking too long, feel free to skip the next section and proceed to step **13** and return back to the analysis later to confirm that you eliminated all the issues.

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

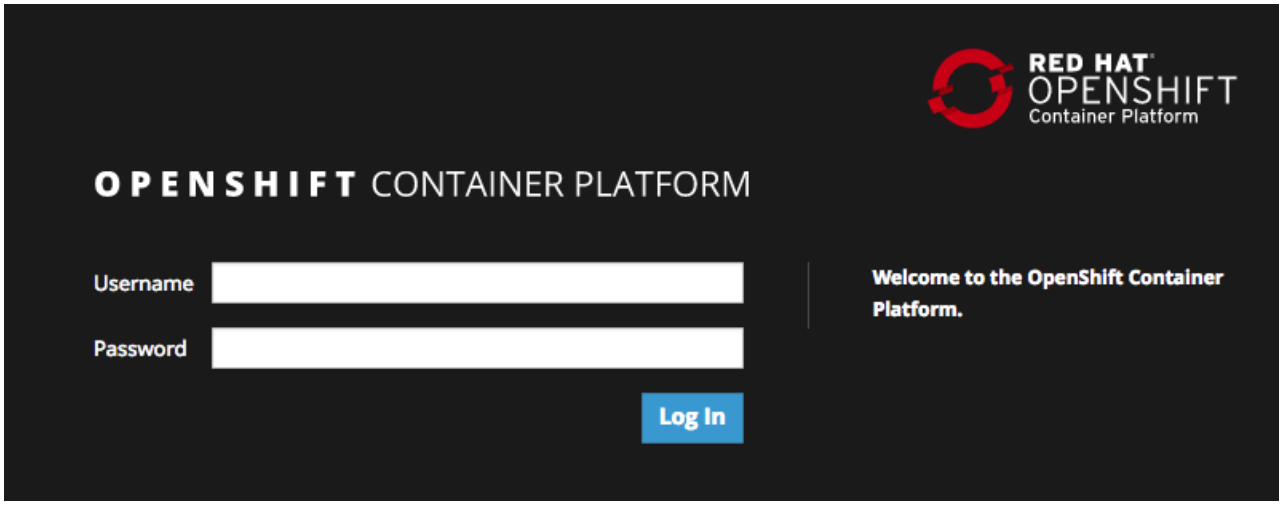
Open the `pom.xml` file.

At the `<!-- TODO: Add OpenShift profile here -->` we are going to add a the following configuration to the pom.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>deployments</outputDirectory>
          <warName>ROOT</warName>
        </configuration>
      </plugin>
    </plugins>
  </build>
</profile>
```

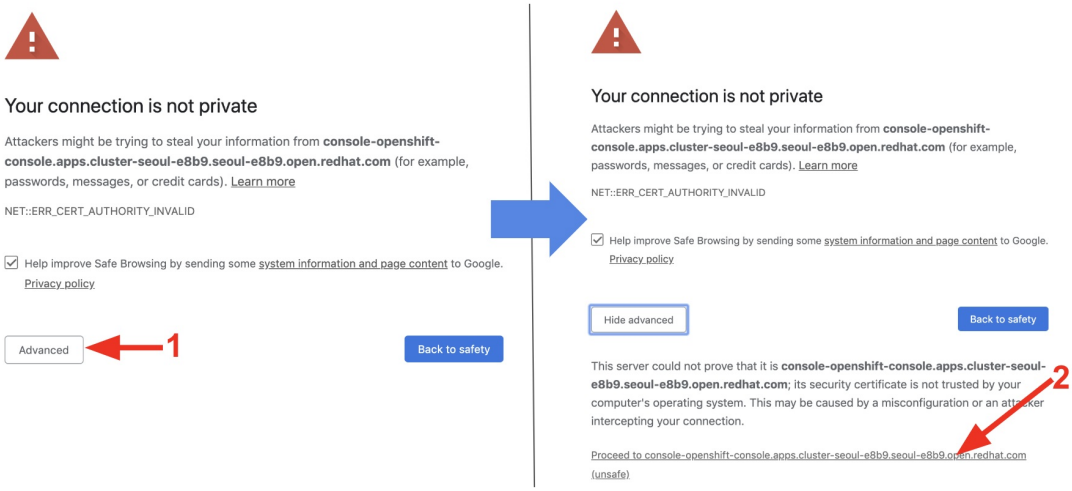
14. Create the OpenShift project

First, open a new brower with the OpenShift web console

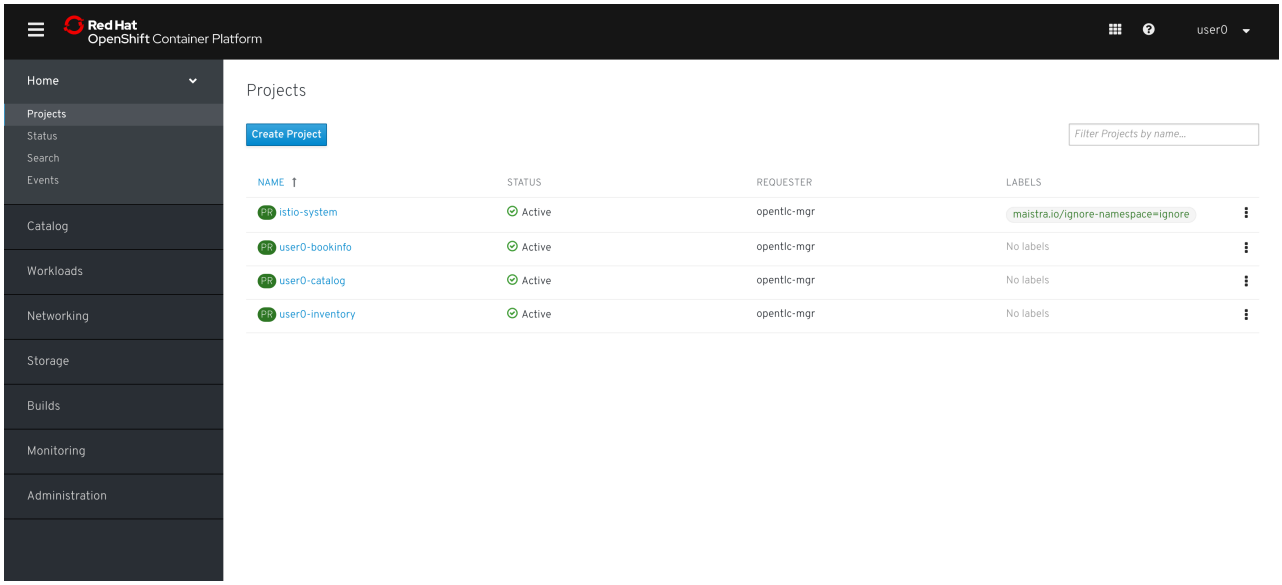


Login using:

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

> **NOTE**: Use of self-signed certificates
>
> When you access the OpenShift web console](https://console-openshift-console.apps.cluster-beijing-5bc2.beijing-5bc2.open.redhat.com) or other URLs via *HTTPS* protocol, you will see browser warnings like `Your > Connection is not secure` since this workshop uses self-signed certificates (which you should not do in production!). For example, if you're using **Chrome**, you will see the following screen.
>
> Click on `Advanced` then, you can access the HTTPS page when you click on `Proceed to...` !!!
>
> 
>
>
> Other browsers have similar procedures to accept the security exception.

You will see the OpenShift landing page:

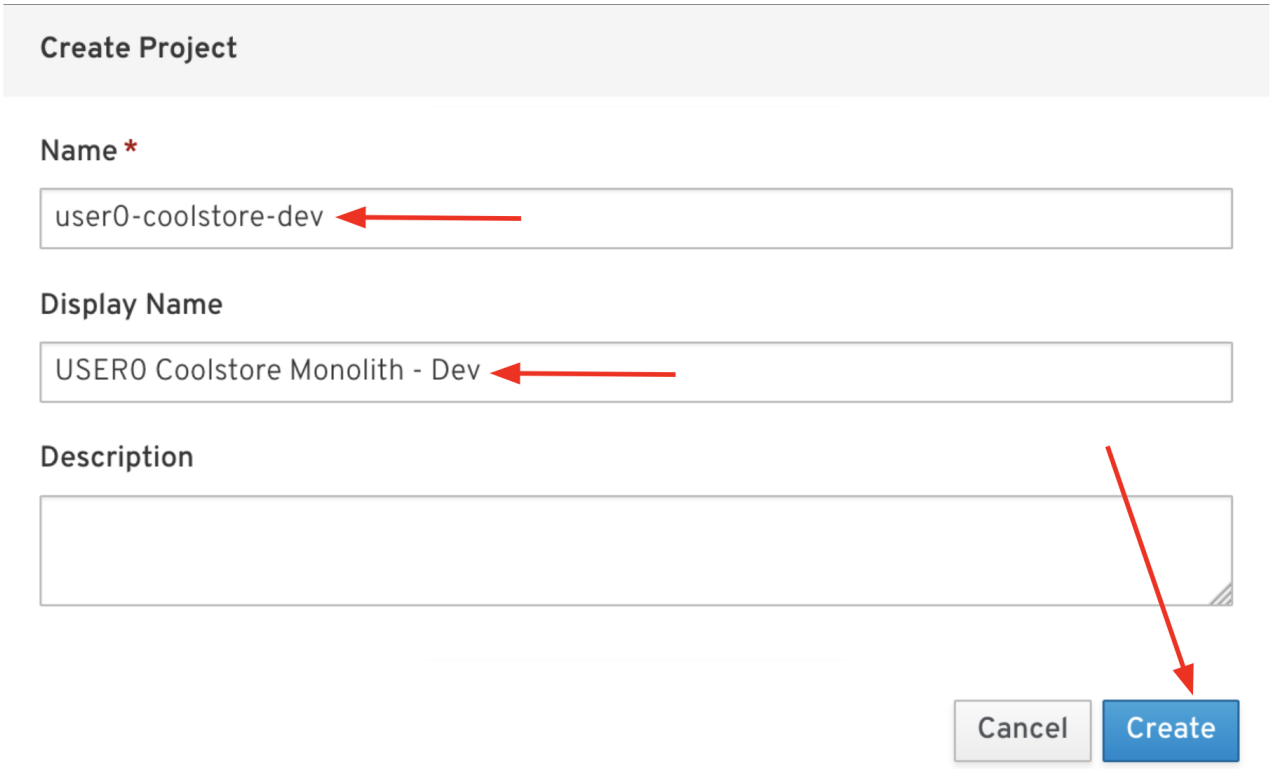


> The project displayed in the landing page depends on which labs you will run today. If you will develop `Service Mesh and Identity` then you will see pre-created projects as the above screeenshot.

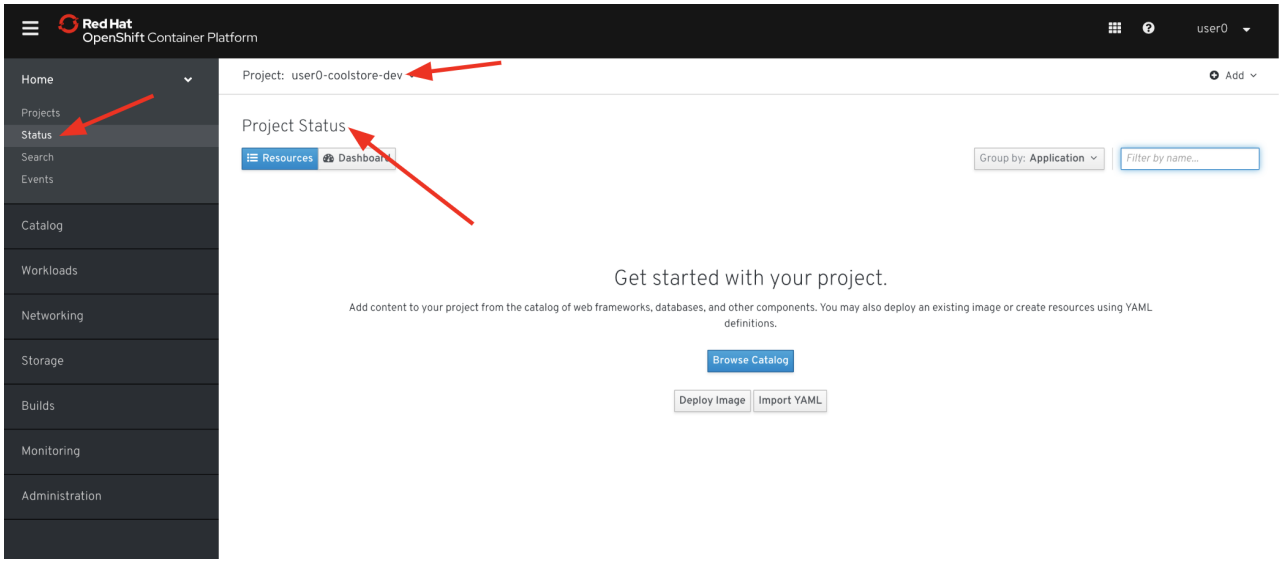
Click `Create Project`, fill in the fields, and click `Create`:

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

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



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



15. Deploy the monolith

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

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

Enter your username and password assigned to you:

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

You should see like:

```
Login successful.

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

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

Using project "default".
Welcome! See 'oc help' to get started.
```

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

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

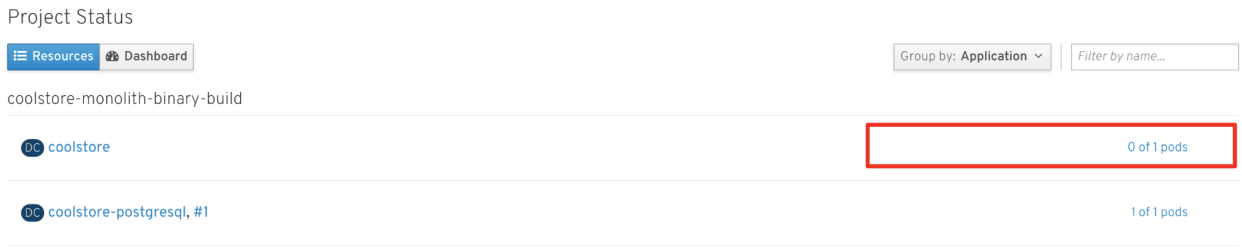
And finally deploy template:

```
oc new-app coolstore-monolith-binary-build
```

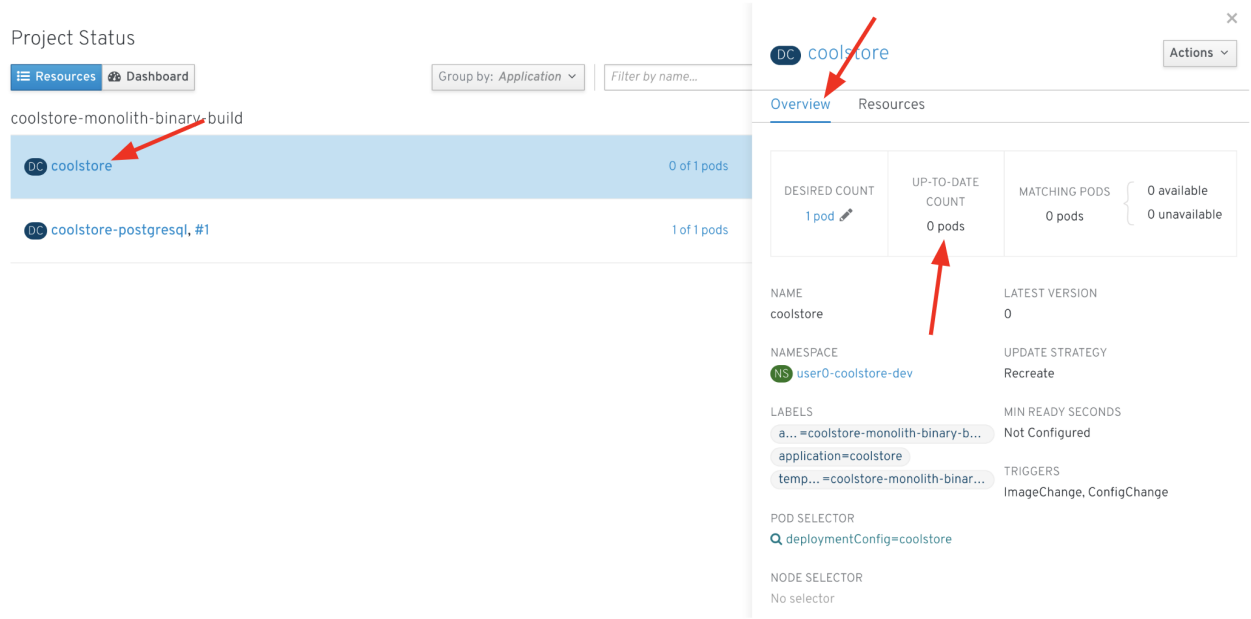
This will deploy both a PostgreSQL database and JBoss EAP, but it will not start a build for our application.

Then open up the `userXX-coolstore-dev` project status page at OpenShift web console

and verify the monolith template items are created:



You can see the components being deployed on the Project Status, but notice the `No running pod for Coolstore`. When you click on `coolstore DC` (Deployment Configs), you will see overview and resources.



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:

```
cd /projects/cloud-native-workshop-v2m1-labs/monolith/
```

```
mvn clean package -Popenshift
```

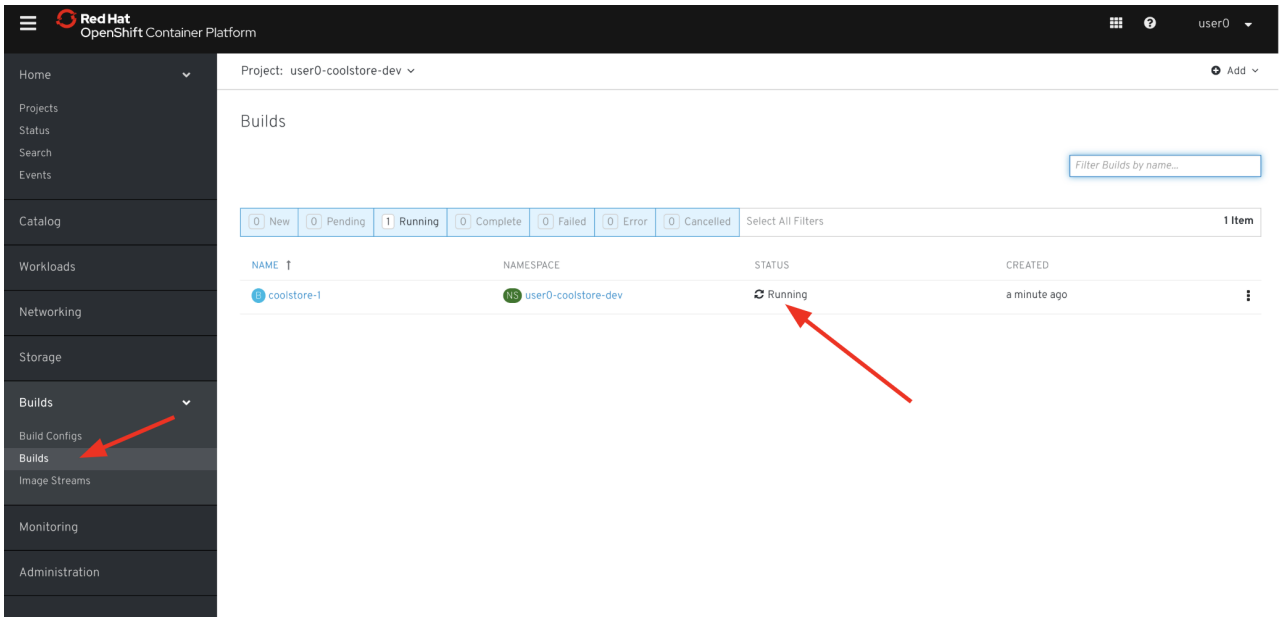
> NOTE : Make sure to run this mvn command at working directory(i.e monolith).

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=deployments/ROOT.war
```

When you navigate `Builds` menu, you will find out `coolstore-xx` is `running` in Status field:



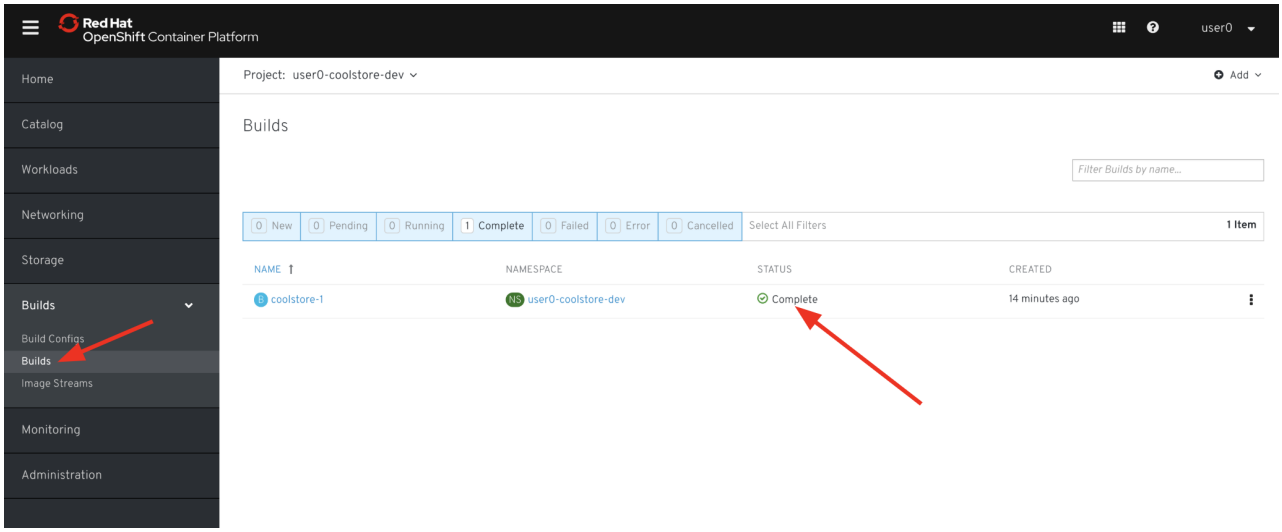
Wait for the build and deploy to complete:

```
oc rollout status -w dc/coolstore
```

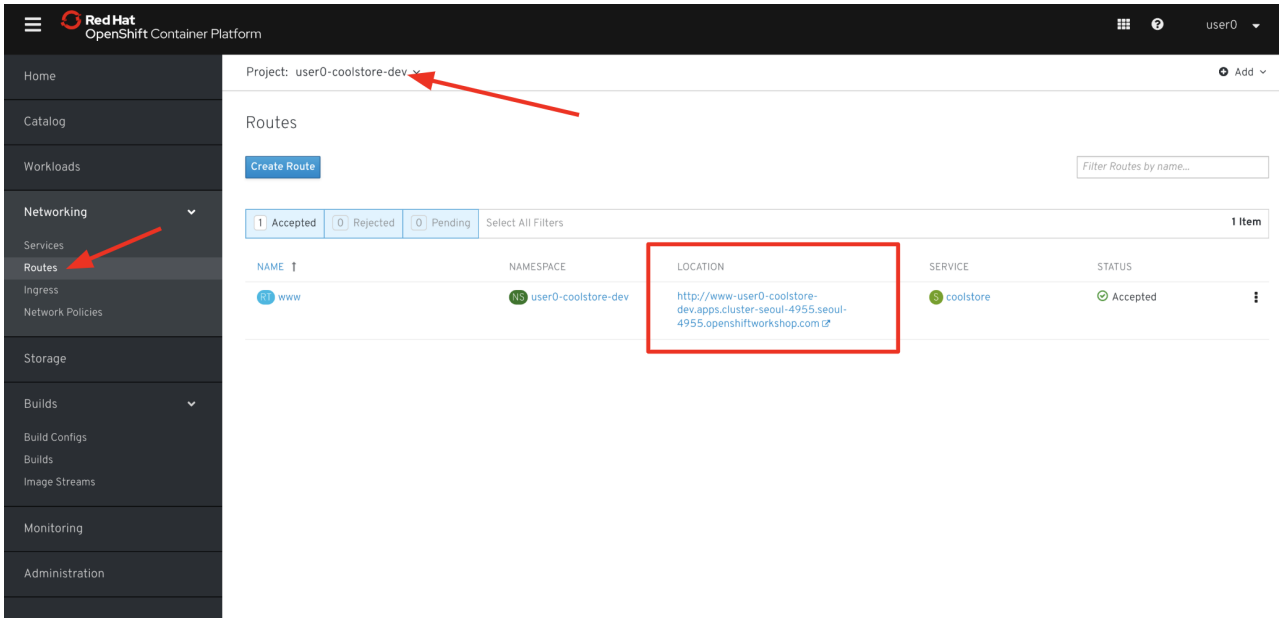
This command will be used often to wait for deployments to complete. Be sure it returns success when you use it! You should eventually see `replication controller "coolstore-1" successfully rolled out` .

> If the above command reports `Error from server (ServerTimeout)` then simply re-run the command until it reports success!

When it's done you should see the application deployed successfully.

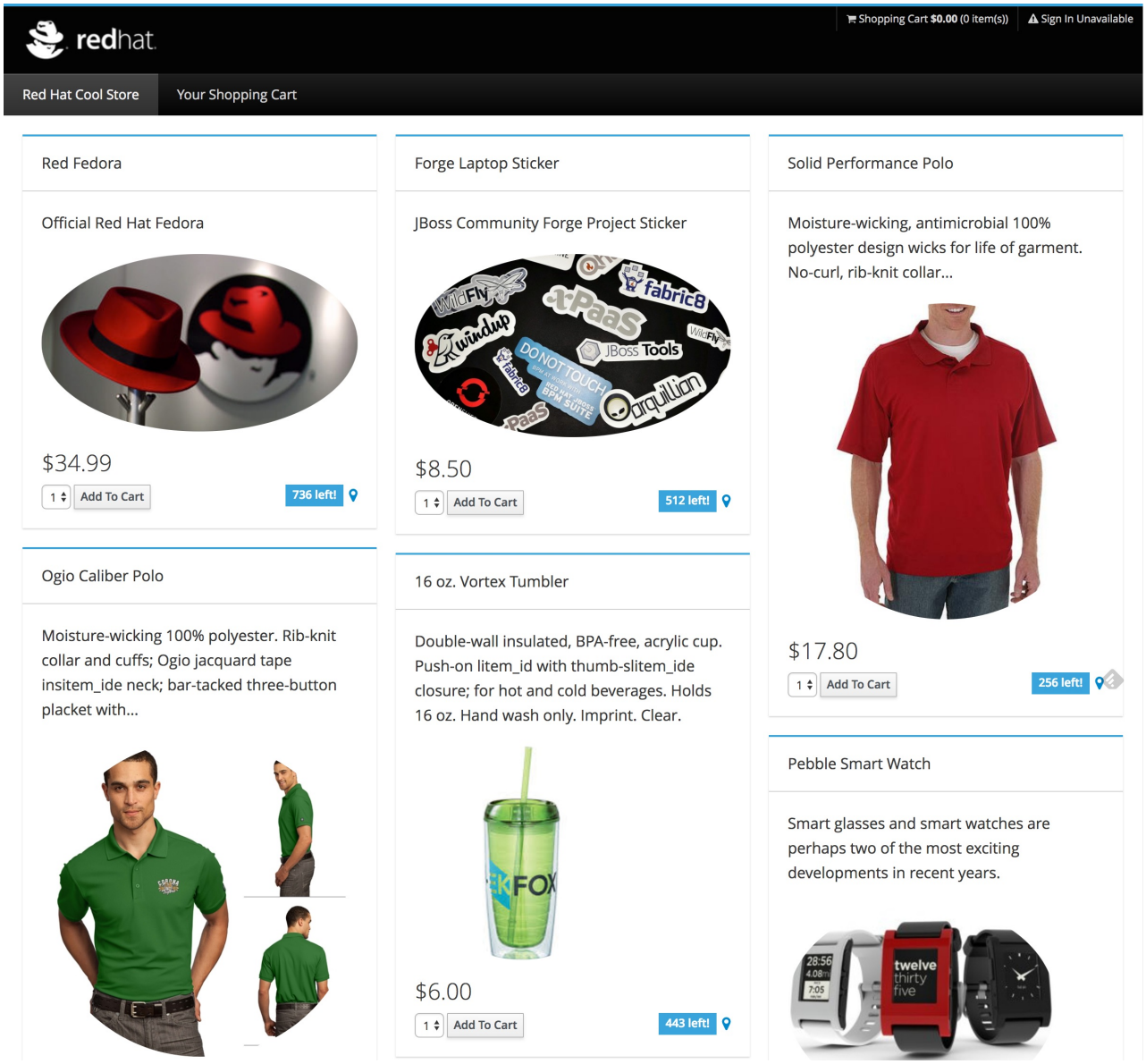


Test the application by clicking on the Route link at `Networking > Routes` on the left menu:



Congratulations!

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



In the next step you'll explore more of the developer features of OpenShift in preparation for moving the monolith to a microservices architecture later on. Let's go!

Summary

Now that you have migrating 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

Lab3 - Breaking the 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` Quarkus, which is designed to be container and developer friendly.

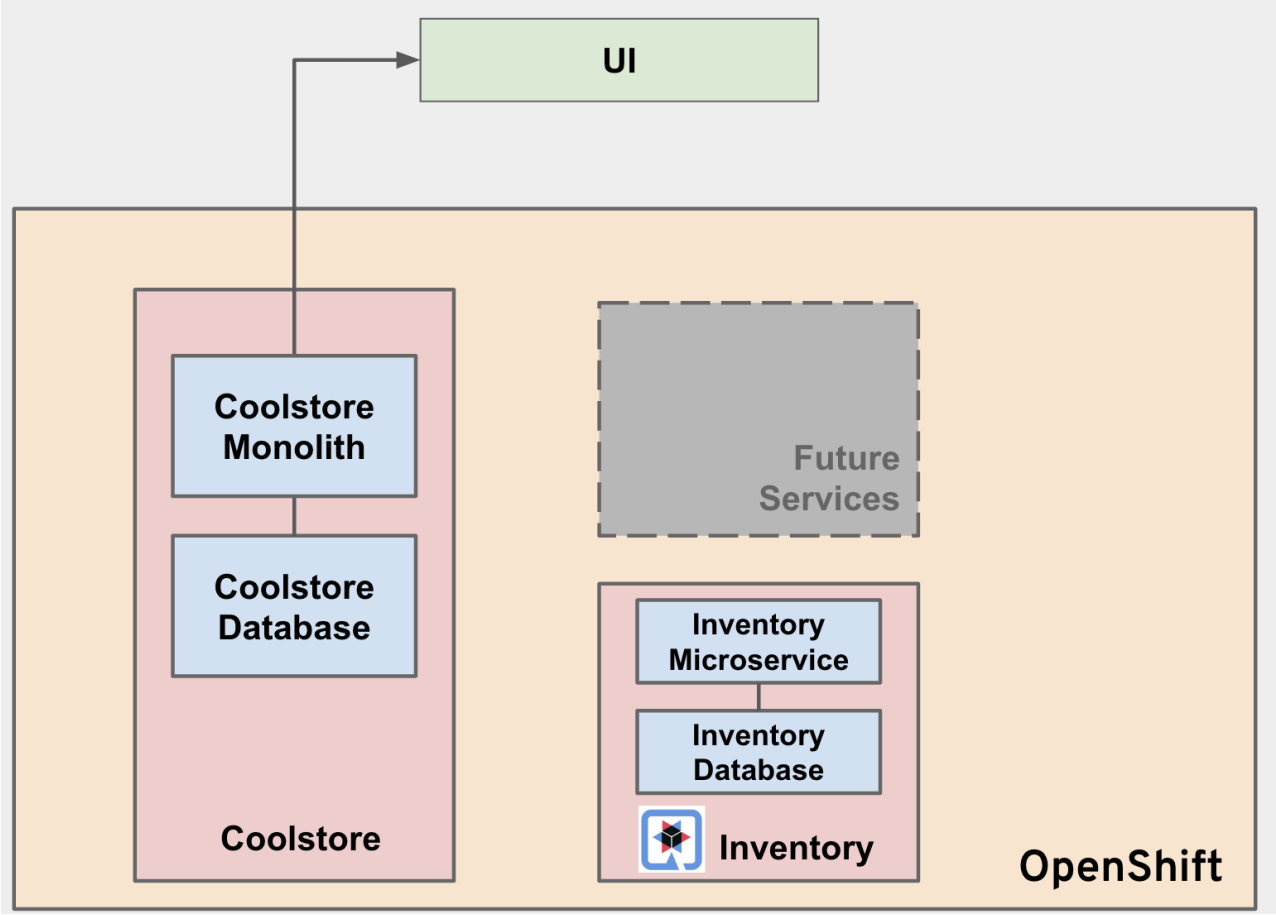
Quarkus is a *Kubernetes Native* Java stack, crafted from the best of breed Java libraries and standards. Amazingly fast boot time, incredibly low RSS memory (not just heap size!) offering near instant scale up and high density memory utilization in container orchestration platforms like Kubernetes.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

This will be one of the runtimes included in Red Hat Runtimes.

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:



What is Quarkus?

For years, the client-server architecture has been the de-facto standard to build applications. But a major shift happened. The one model rules them all age is over. A new range of applications and architecture styles has emerged and impacts how code is written and how applications are deployed and executed. HTTP microservices, reactive applications, message-driven microservices and serverless are now central players in modern systems.

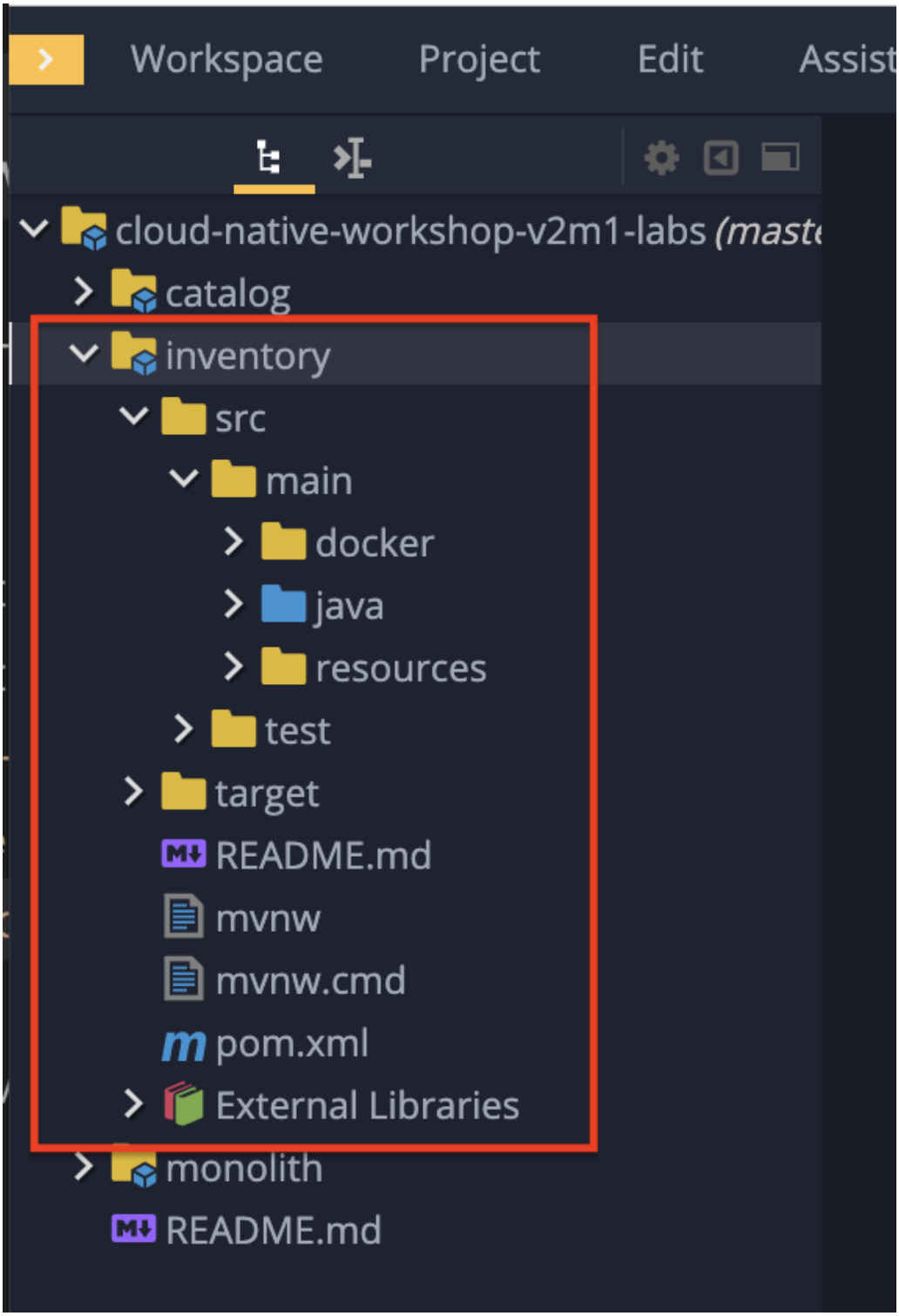
Quarkus offers 4 major benefits to build cloud-native, microservices, and serverless Java applicaitons:

- **Developer Joy** - Cohesive platform for optimized developer joy through unified configuration, Zero config with live reload in the blink of an eye, streamlined code for the 80% common usages with flexible for the 20%, and no hassle native executable generation.

- **Unifies Imperative and Reactive** - Inject the EventBus or the Vertx context for both Reactive and imperative development in the same application.
- **Functions as a Service and Serverless** - Superfast startup and low memory utilization. With Quarkus, you can embrace this new world without having to change your programming language.
- **Best of Breed Frameworks & Standards** - CodeReady Workspaces Vert.x, Hibernate, RESTEasy, Apache Camel, CodeReady Workspaces MicroProfile, Netty, Kubernetes, OpenShift, Jaeger, Prometheus, Apacke Kafka, Infinispan, and more.

1. Setup an Inventory proejct

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.

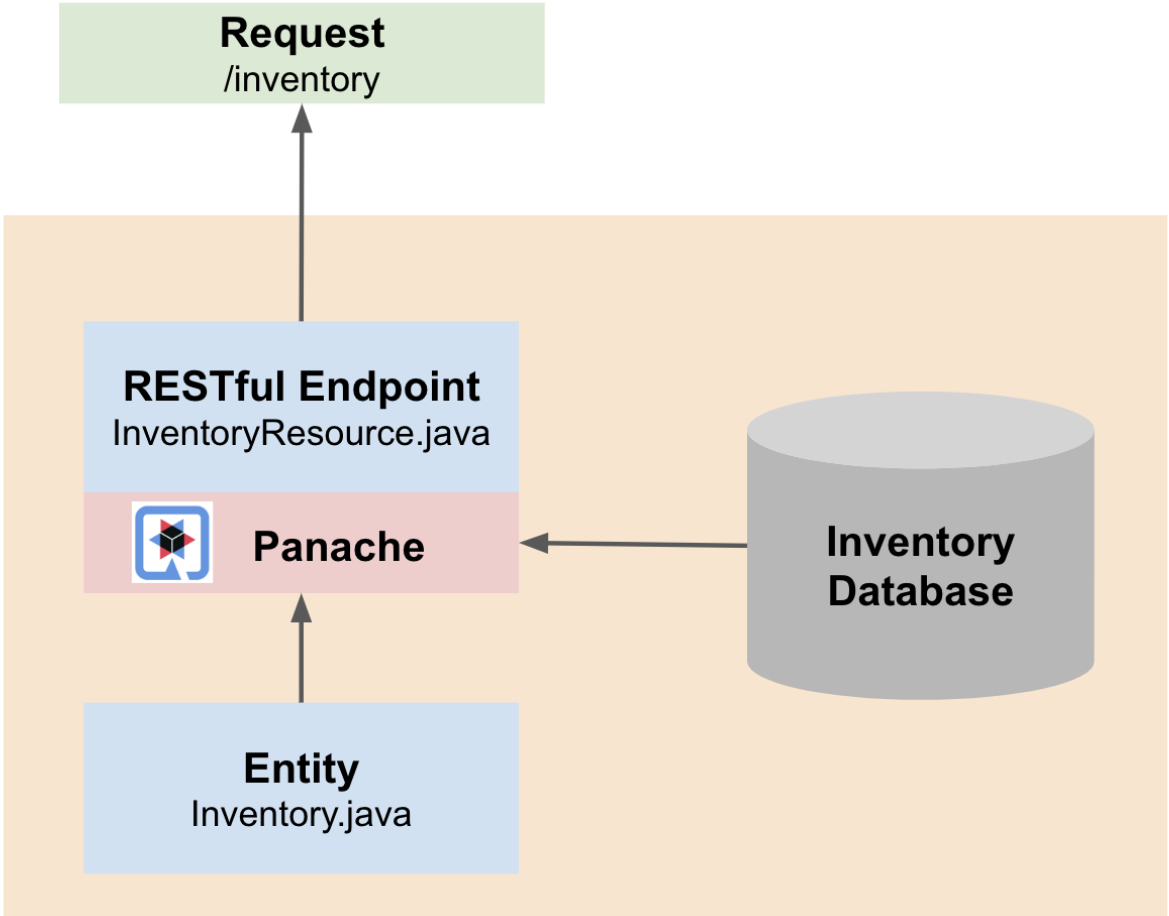
While the code is surprisingly simple, under the hood this is using:

- 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` (for testing) and we'll use the Quarkus Maven Plugin. Copy the following commands to add the *Hibernate ORM with Panache* extension via CodeReady Workspaces Terminal:

Go to `inventory' directory:

`cd /projects/cloud-native-workshop-v2m1-labs/inventory/`

`mvn quarkus:add-extension -Dextensions="hibernate-orm-panache"`

And then for local H2 database:

`mvn quarkus:add-extension -Dextensions="jdbc-h2"`

> NOTE: There are many more extensions for Quarkus for popular frameworks like Vert.x, Apache Camel, Infinispan, Spring DI compatibility (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!

Open up the empty **Inventory.java** file in *com.redhat.coolstore* package and paste the following code into it (identical to the monolith code):

```
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 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:

```
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.ext.ExceptionMapper;
import javax.ws.rs.ext.Provider;

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

@Path("/services/inventory")
@ApplicationScoped
@Produces("application/json")
@Consumes("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:

- `/inventory` that is accessible via *HTTP GET* which will return all known product Inventory entities as JSON
- `/inventory/<itemId>` that is accessible via *HTTP GET* at for example `/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**:

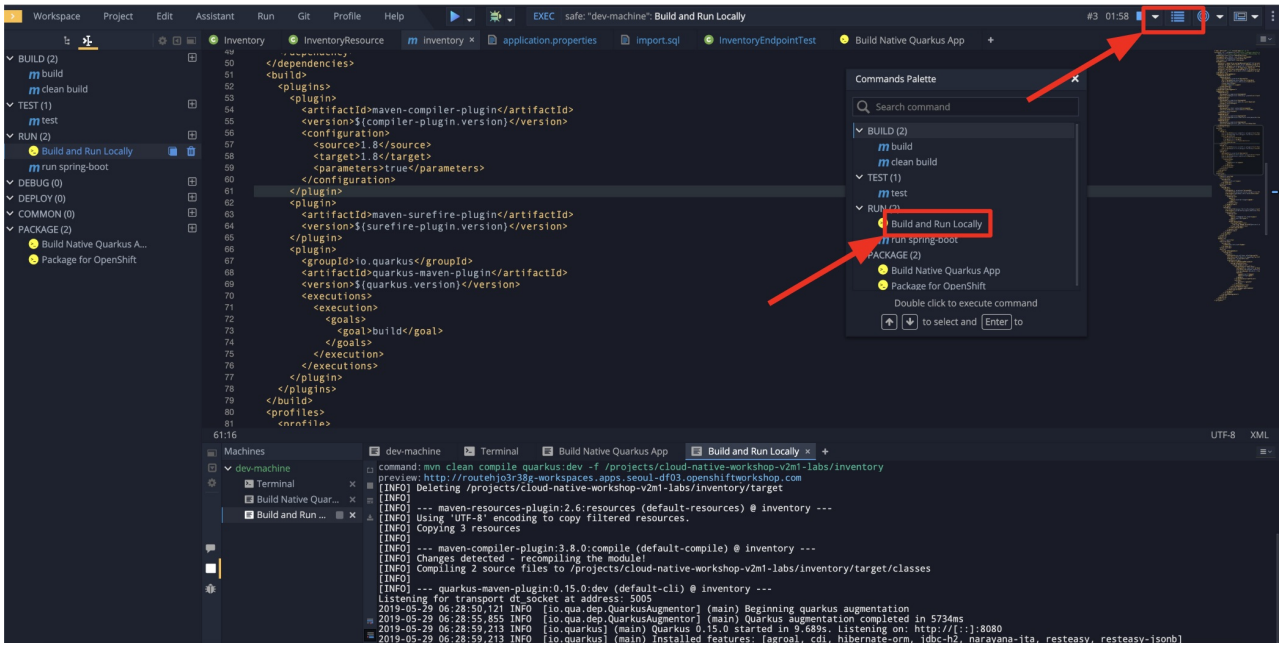
```
INSERT INTO INVENTORY (id, itemId, link, location, quantity) values (nextval('hibernate_sequence'), '329299', 'http://maps.google.com/?q=Raleigh', 'Raleigh', 736);
INSERT INTO INVENTORY (id, itemId, link, location, quantity) values (nextval('hibernate_sequence'), '329199', 'http://maps.google.com/?q=Boston', '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` :

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

7. Run Quarkus Inventory application

Now we are ready to run the inventory application. Click on **Commands Palette** then select **Build and Run Locally** in Run menu:



> This simply runs `mvn compile quarkus:dev` for you

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

```
2019-09-26 15:55:38,447 INFO  [io.quarkus] (main) Quarkus 0.22.0 started in 2.905s. Listening on: http://0.0.0.0:8080
2019-09-26 15:55:38,447 INFO  [io.quarkus] (main) Profile dev activated. Live Coding activated.
2019-09-26 15:55:38,447 INFO  [io.quarkus] (main) Installed features: [agroal, cdi, hibernate-orm, jdbc-h2, narayana-jta, resteasy, resteasy-jsonb]
```

Open a **new** CodeReady Workspaces Terminal and invoke the RESTful endpoint using the following CURL commands. The output looks like here:

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


```
[{"id":1,"itemId":"329299","link":"http://maps.google.com/?q=Raleigh","location":"Raleigh","quantity":736},{"id":2,"itemId":"329199","link":"http://maps.google.com/?q=Boston","location":"Boston","quantity":512},{"id":3,"itemId":"165613","link":"http://maps.google.com/?q=Seoul","location":"Seoul","quantity":256},{"id":4,"itemId":"165614","link":"http://maps.google.com/?q=Singapore","location":"Singapore","quantity":54},{"id":5,"itemId":"165954","link":"http://maps.google.com/?q=London","location":"London","quantity":87},{"id":6,"itemId":"444434","link":"http://maps.google.com/?q=NewYork","location":"NewYork","quantity":443},{"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}]
```


```
curl http://localhost:8080/services/inventory/329199 ; echo
```


```
[{"id":2,"itemId":"329199","link":"http://maps.google.com/?q=Boston","location":"Boston","quantity":512}]
```


Stop the application

Stop Quarkus development mode by closing the *Build and Run Locally* Terminal window.

8. Add Test Code and create a package

In this step, we will add unit tests so that we can test during `mvn package`. Open up the `src/test/java/com/redhat/coolstore/InventoryEndpointTest.java` file and replace the following code with the below code:

```
package com.redhat.coolstore;

import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.containsString;

import io.quarkus.test.junit.QuarkusTest;
import org.junit.jupiter.api.Test;

@QuarkusTest
public class InventoryEndpointTest {

    @Test
    public void testListAllInventory() {
        //List all, should have all 8 cities inventory the database has initially:
        given()
            .when().get("/services/inventory")
            .then()
            .statusCode(200)
            .body(
                containsString("Raleigh"),
                containsString("Boston"),
                containsString("Seoul"),
                containsString("Singapore"),
                containsString("London"),
                containsString("NewYork"),
                containsString("Paris"),
                containsString("Tokyo")
                );

        //List a certain item(ID:329299), Raleigh should be returned:
        given()
        .when().get("/services/inventory/329299")
        .then()
        .statusCode(200)
        .body(
            containsString("Raleigh")
        );
    }

}
```

Next we'll build an executable jar then deploy it to **OpenShift** soon. Use the following maven command via CodeReady Workspaces Terminal:

```
cd /projects/cloud-native-workshop-v2m1-labs/inventory
```

```
mvn clean package
```

If builds successfully (you will see **BUILD SUCCESS**), continue to the next step to deploy the application to OpenShift.

You can also run the Uber.jar to make sure if the inventory works. Use the following **Java command** then you see a similar output:

```
java -jar target/inventory-1.0-SNAPSHOT-runner.jar
```

```
2019-09-26 16:16:08,977 INFO  [io.quarkus] (main) Quarkus 0.23.1 started in 1.806s. Listening on: http://0.0.0.0:8080
2019-09-26 16:16:08,989 INFO  [io.quarkus] (main) Profile prod activated.
2019-09-26 16:16:08,989 INFO  [io.quarkus] (main) Installed features: [agroal, cdi, hibernate-orm, jdbc-h2, narayana-jta, resteasy, resteasy-jsonb]
```

Open a new CodeReady Workspaces Terminal and invoke the RESTful endpoint using the following CURL commands. The output looks like here:

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

```
[{"id":1,"itemId":"329299","link":"http://maps.google.com/?q=Raleigh","location":"Raleigh","quantity":736},{"id":2,"itemId":"329199","link":"http://maps.google.com/?q=Boston","location":"Boston","quantity":512},{"id":3,"itemId":"165613","link":"http://maps.google.com/?q=Seoul","location":"Seoul","quantity":256},{"id":4,"itemId":"165614","link":"http://maps.google.com/?q=Singapore","location":"Singapore","quantity":54},{"id":5,"itemId":"165954","link":"http://maps.google.com/?q=London","location":"London","quantity":87},{"id":6,"itemId":"444434","link":"http://maps.google.com/?q=NewYork","location":"NewYork","quantity":443},{"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}]
```

```
curl http://localhost:8080/services/inventory/329199 ; echo
```

```
[{"id":2,"itemId":"329199","link":"http://maps.google.com/?q=Boston","location":"Boston","quantity":512}]
```

Stop the application

Stop Quarkus development mode by closing the Terminal window in which you ran `java -jar`

You have now successfully created your first microservice using Quarkus and implemented a basic RESTful API on top of the Inventory database. Most of the code looks simpler than the monolith, demonstrating how easy it is to migrate existing monolithic Java EE application to microservices using `Quarkus` .

In next steps 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.

9. Create OpenShift Project

We have already deployed our coolstore monolith to OpenShift, but now we are working on re-architecting it to be microservices-based.

In this step, we will deploy our new Inventory microservice for our CoolStore application, so create 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.

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

```
cd /projects/cloud-native-workshop-v2m1-labs/inventory/
```

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

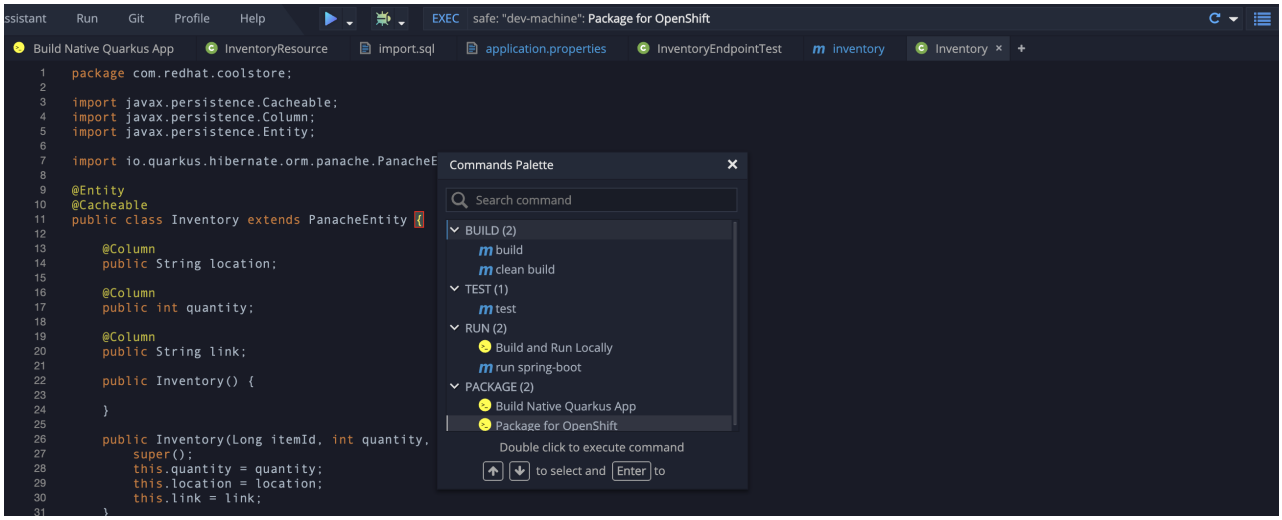
- **dev** - Activated when in development mode (i.e. **quarkus:dev**)
- **test** - Activated when running tests
- **prod** - The default profile when not running in development or test mode

There are two ways to set a custom profile, either via the `quarkus.profile` system property or the `QUARKUS_PROFILE` environment variable. If both are set the system property takes precedence. Note that it is not necessary to define the names of these profiles anywhere, all that is necessary is to create a config property with the profile name, and then set the current profile to that name.

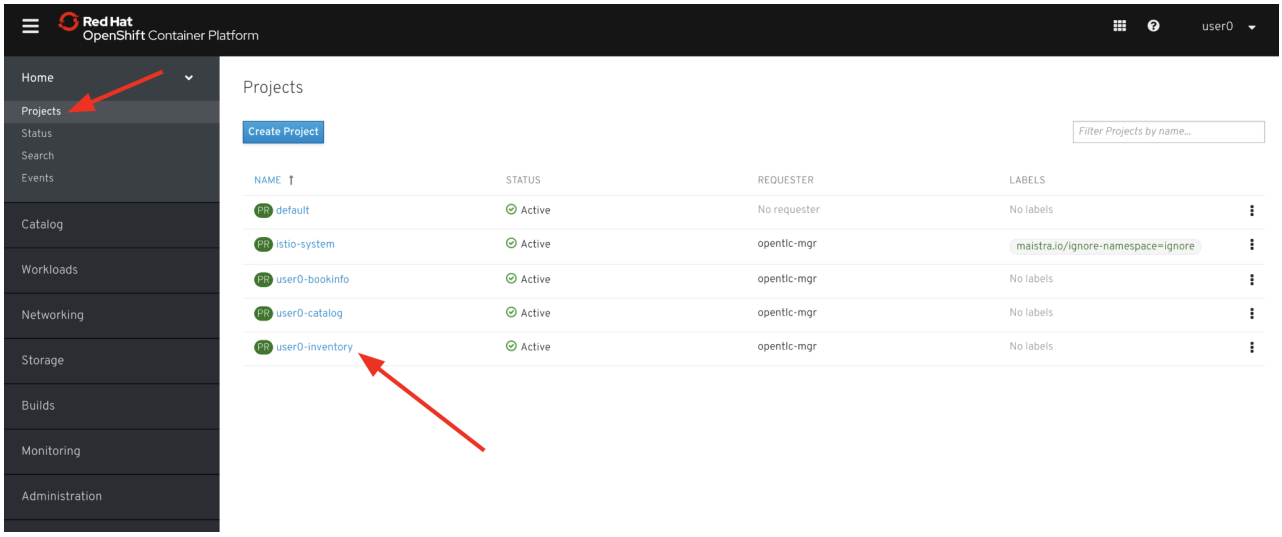
Let's add the following variables in *src/main/resources/application.properties*:

```
%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
```

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



In OpenShift, click on the name of the **userXX-inventory** project:



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

10. 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 by executing the following commands via CodeReady Workspaces Terminal:

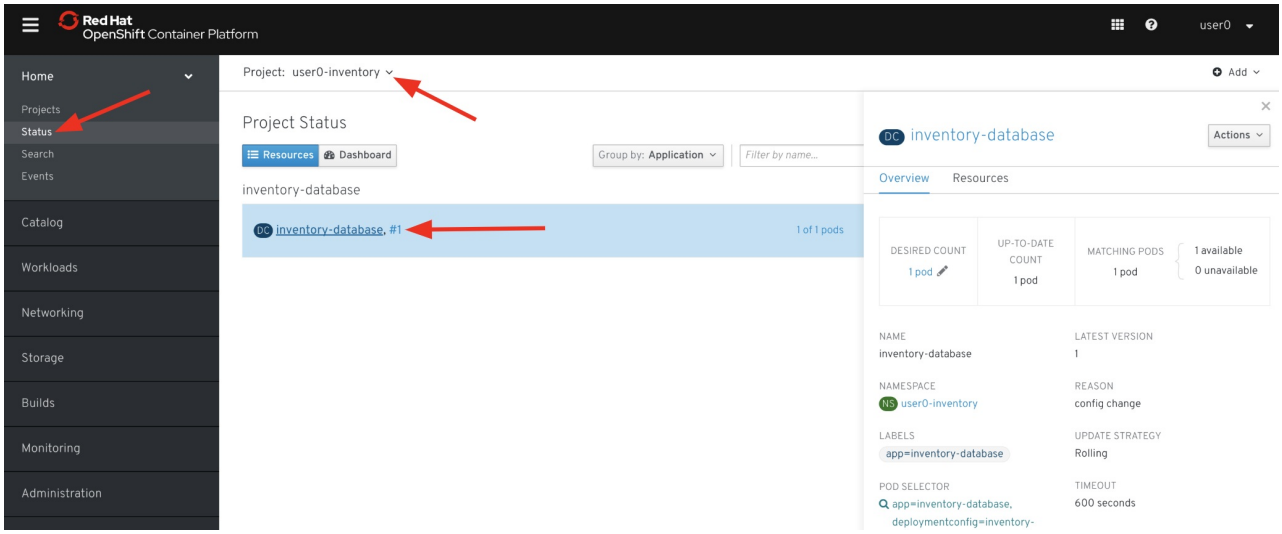
`oc project userXX-inventory`

Then run:

```
oc new-app -e POSTGRESQL_USER=inventory \
  -e POSTGRESQL_PASSWORD=mysecretpassword \
  -e POSTGRESQL_DATABASE=inventory openshift/postgresql:10 \
  --name=inventory-database
```

> NOTE: If you change the username and password you also need to update `src/main/resources/application.properties` which contains the credentials used when deploying to OpenShift.

This will deploy the database to our new project.



11. Build and Deploy

Red Hat OpenShift Application Runtimes includes a powerful maven plugin that can take an existing Quarkus application and generate the necessary Kubernetes configuration.

Build and deploy the project using the following command, which will use the maven plugin to deploy via CodeReady Workspaces Terminal:

`oc new-build registry.access.redhat.com/redhat-openjdk-18/openjdk18-openshift:1.5 --binary --name=inventory-quarkus -l app=inventory-quarkus`

This build uses the new Red Hat OpenJDK Container Image, providing foundational software needed to run Java applications, while staying at a reasonable size.

> NOTE: After a while, you may get logged out of OpenShift. If the above command fails due to permissions, you can repeat the OpenShift login process from earlier.

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

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

Once the build is done, we'll deploy it as an OpenShift application and override the Postgres URL to specify our production Postgres credentials:

```
oc new-app inventory-quarkus -e QUARKUS_PROFILE=prod
```

and expose your service to the world:

```
oc expose service inventory-quarkus
```

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

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

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

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

And now we can access using curl once again to find all inventories:

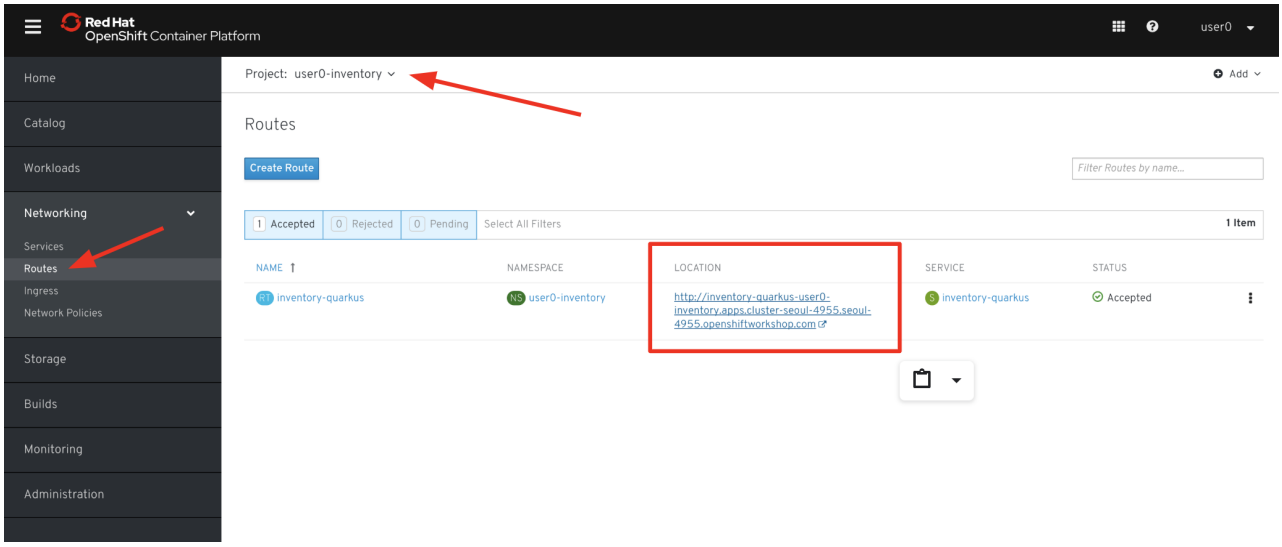
```
export URL="http://$(oc get route | grep inventory | awk '{print $2}')"
```

```
curl $URL/services/inventory ; echo
```

So now `Inventory` service is deployed to OpenShift. You can also see it in the Project Status in the OpenShift Console with its single replica running in 1 pod, along with the Postgres database pod.

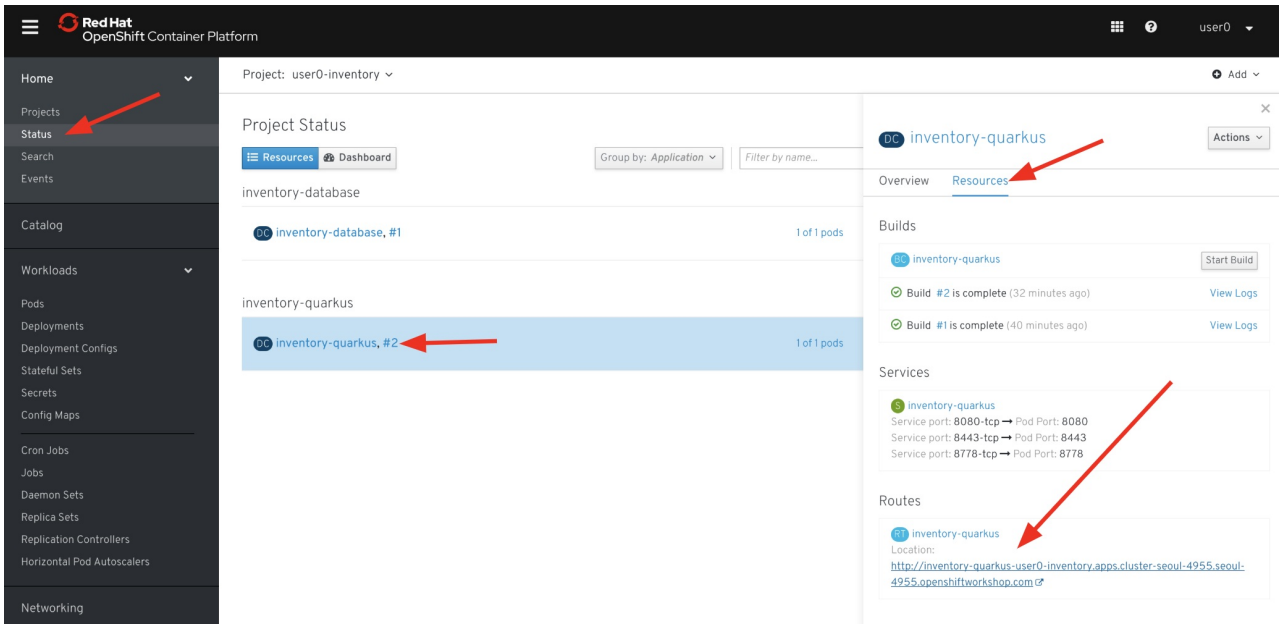
12. Access the application running on OpenShift

This sample project includes a simple UI that allows you to access the Inventory API. This is the same UI that you previously accessed outside of OpenShift which shows the CoolStore inventory. Click on the route URL at `Networking > Routes` to access the sample UI.



> NOTE: If you get a '404 Not Found' error, just reload the page a few times until the Inventory UI appears. This is due to a lack of health check which you are about to fix!

You can also access the application through the link on the `Project Status` page.



The UI will refresh the inventory table every 2 seconds, as before.

In the next steps you will enhance OpenShift's ability to manage the application lifecycle by implementing a *health check pattern*. By default, without health checks (or health *probes*) OpenShift considers services to be ready to accept service requests even before the application is truly ready or if the application is hung or otherwise unable to service requests. OpenShift must be *taught* how to recognize that our app is alive and ready to accept requests.

What is MicroProfile Health?

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

restarted. **Quarkus application** can utilize the MicroProfile Health specification through the *SmallRye Health extension*.

What is a Health Check?

A key requirement in any managed application container environment is the ability to determine when the application is in a ready state. Only when an application has reported as ready can the manager (in this case OpenShift) act on the next step of the deployment process. OpenShift makes use of various *probes* to determine the health of an application during its lifespan. A *readiness* probe is one of these mechanisms for validating application health and determines when an application has reached a point where it can begin to accept incoming traffic. At that point, the IP address for the pod is added to the list of endpoints backing the service and it can begin to receive requests. Otherwise traffic destined for the application could reach the application before it was fully operational resulting in error from the client perspective.

Once an application is running, there are no guarantees that it will continue to operate with full functionality. Numerous factors including out of memory errors or a hanging process can cause the application to enter an invalid state. While a *readiness* probe is only responsible for determining whether an application is in a state where it should begin to receive incoming traffic, a *liveness* probe is used to determine whether an application is still in an acceptable state. If the liveness probe fails, OpenShift will destroy the pod and replace it with a new one.

In our case we will implement the health check logic in a REST endpoint and let Quarkus publish that logic on the */health* endpoint for use with OpenShift.

13. Add Health Check Extension

We will add a Qurakus extension to the Inventory application for using **smallrye-health** and we'll use the Quarkus Maven Plugin. Copy the following commands to import the smallrye-health extension that implements the MicroProfile Health specification via CodeReady Workspaces Terminal:

Go to `inventory` directory and add the extension:

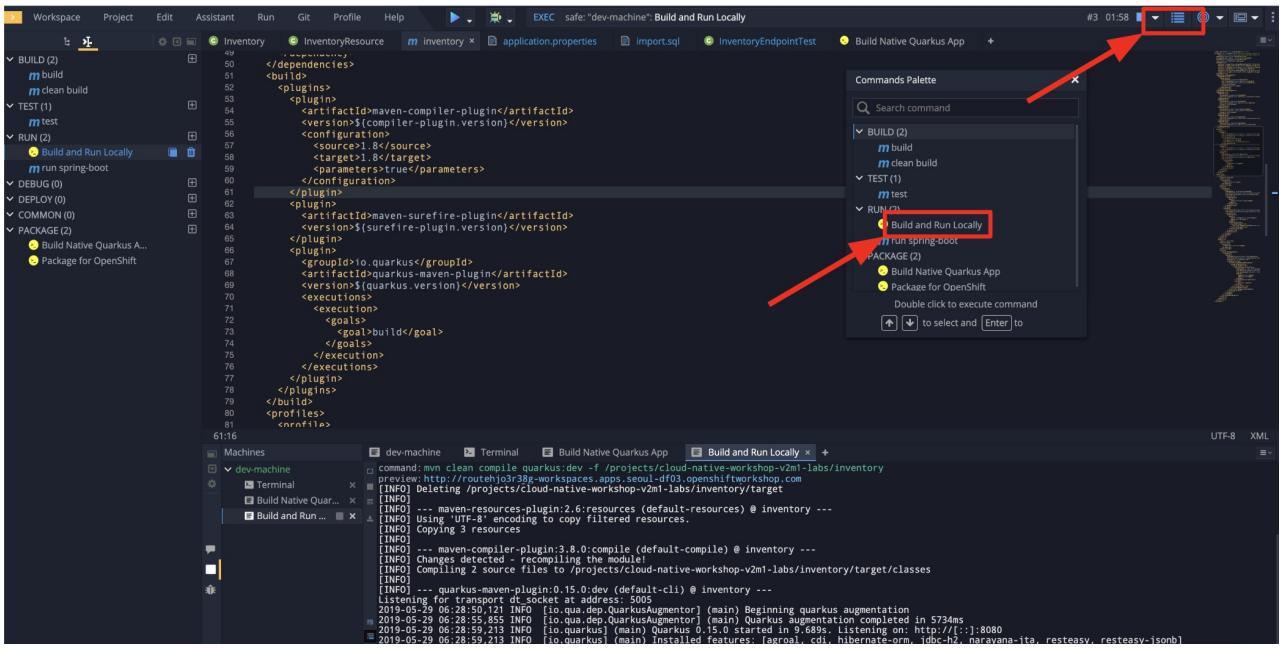
```
cd /projects/cloud-native-workshop-v2m1-labs/inventory/
```

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

14. Run the health check

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

Run the Inventory application via `mvn compile quarkus:dev` or click on **Build and Run Locally** in Commands Palette:



In a separate Terminal, access the `health check` endpoint using `curl http://localhost:8080/health` and the result should look like:

```
{
    "status": "UP",
    "checks": [
    ]
}
```

The health REST endpoint returns a simple JSON object with two fields:

- **status** - the overall result of all the health check procedures
- **checks** - an array of individual checks

The general *status* of the health check is computed as a logical AND of all the declared health check procedures. The *checks* array is empty as we have not specified any health check procedure yet so let's define some.

15. 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 the following logic will be put into a new Java class.

Replace the following codes with the exsiting entire codes:

```
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 Quarkus's **@Readiness** annotation, which directs Quarkus to expose this endpoint as a health check at */health/ready*.

> NOTE: You don't need to stop and re-run re-run the Inventory application because Quarkus will **reload the changes automatically**.

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

```
curl http://localhost:8080/health/ready
```

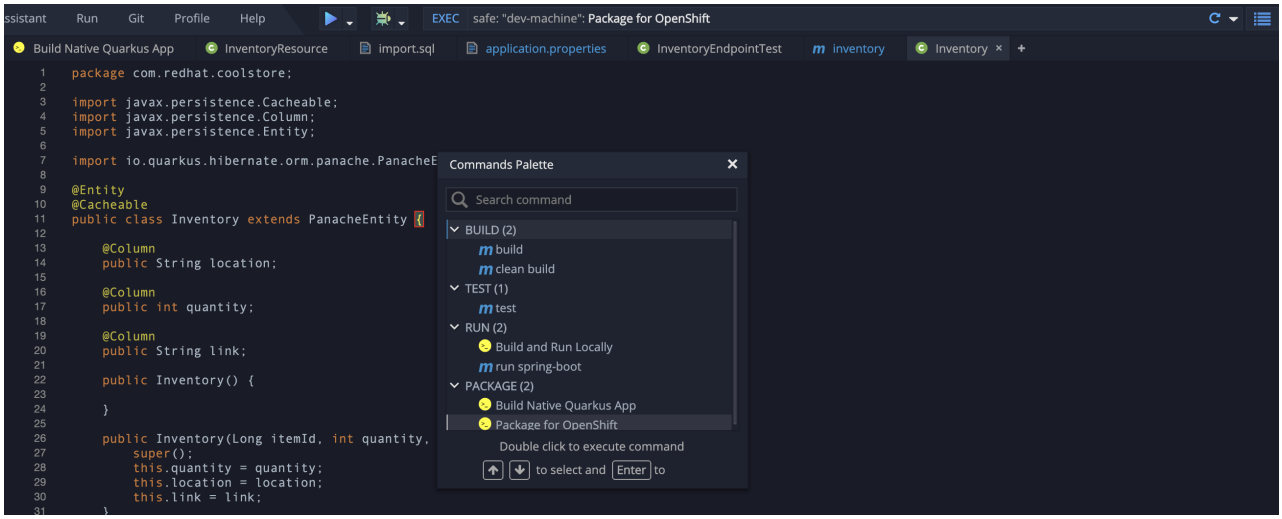
```
{
  "status": "UP",
   "checks": [
      {
         "name": "Success of Inventory Health Check!!!",
         "state": "UP"
      }
   ]
}
```

With our new health check in place, we'll need to build and deploy the updated application in the next step. Before move to the next step, be sure to close the termial where you're running Quarkus development mode.

`Tip` : You can define liveness probe using **@Liveness** annotation and the liveness check can be accessible at **/health/live** endpoint.

16. Re-Deploy to OpenShift

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



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

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

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

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

And wait for the result as below:

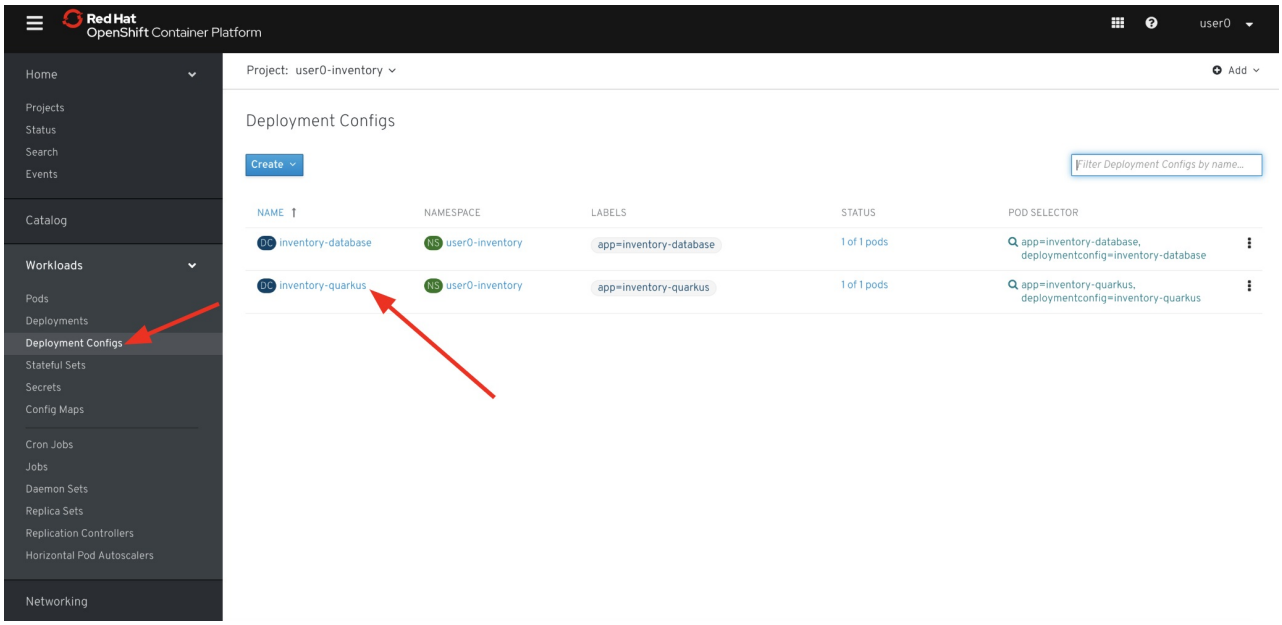
```
replication controller "inventory-quarkus-XX" successfully rolled out
```

Let's set *readiness probe* in OpenShift using *InventoryHealthCheck*. Run the follwing *oc set probe* command in CodeReady Workspaces:

```
oc set probe dc/inventory-quarkus --readiness --get-url=http://:8080/health/ready
```

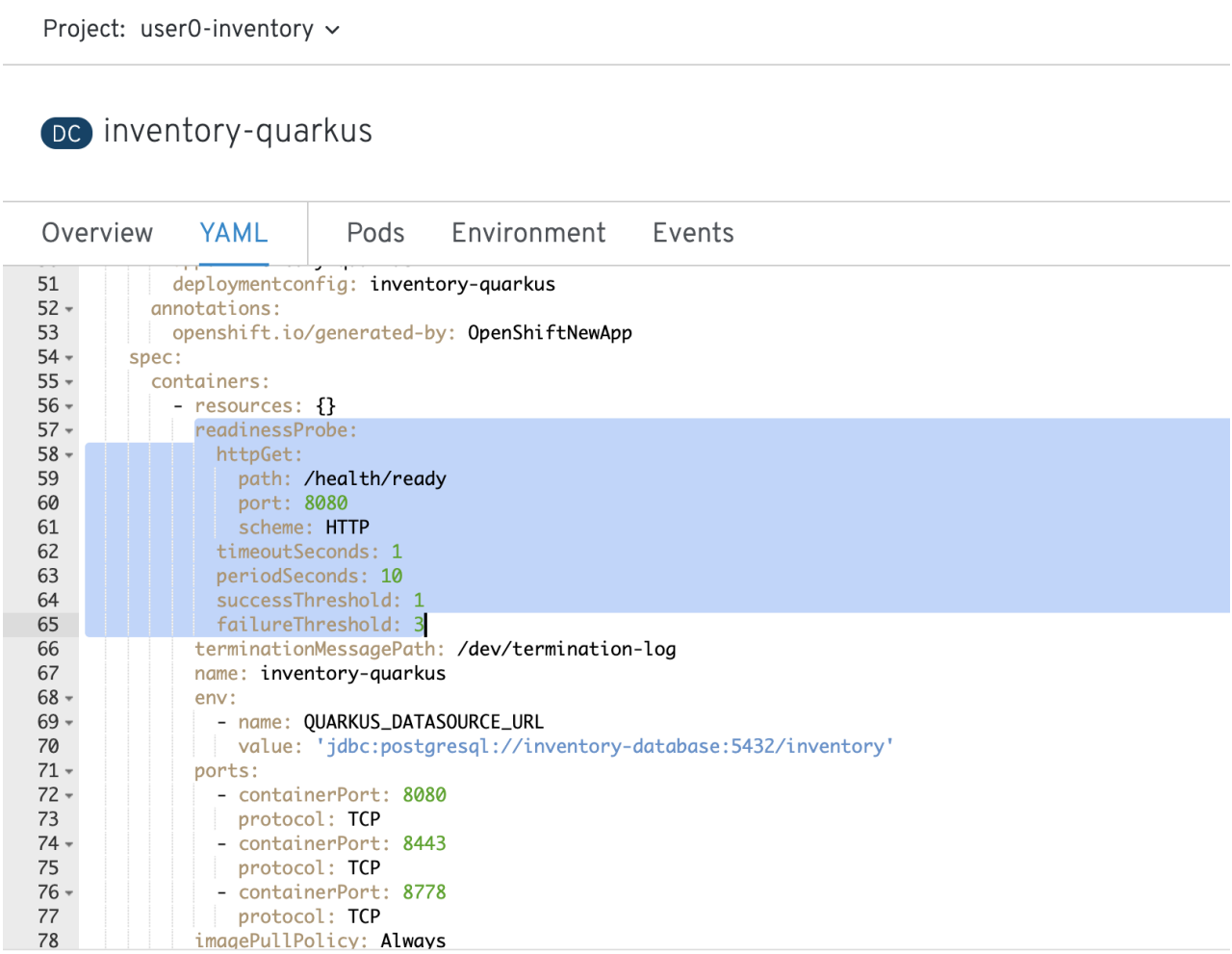
This will instruct OpenShift to use the `/health/ready` endpoint to continually check the health of the app.

Back on the OpenShift console, Navigate to *Deployment Configs* on the left menu then click on *inventory-quarkus*:



Click on *YAML* tab then you will see the following variables in *template.spec.containers.resources* path:

```
readinessProbe:
  httpGet:
    path: /health/ready
    port: 8080
    scheme: HTTP
  timeoutSeconds: 1
  periodSeconds: 10
  successThreshold: 1
  failureThreshold: 3
```



You should also be able to access the health check logic at the *inventory* endpoint via a web browser:

```
export URL="http://$(oc get route | grep inventory | awk '{print $2}')"
```

```
curl $URL/health/ready ; echo
```

You should see a JSON response like:

```
{
  "status": "UP",
  "checks": [
    {
      "name": "Success of Inventory Health Check!!!",
     "state": "UP"
    }
  ]
}
```

You can see the definition of the health check from the perspective of OpenShift via CodeReady Workspaces Terminal:

```
oc describe dc/inventory-quarkus | egrep 'Readiness|Liveness'
```

You should see:

> Readiness: http-get http://:8080/health/ready delay=0s timeout=1s period=10s #success=1 #failure=3

17. 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 3o seconds for it to be activated. Use the *oc* command to tune the probe to wait 30 seconds before starting to poll the probe:

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

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

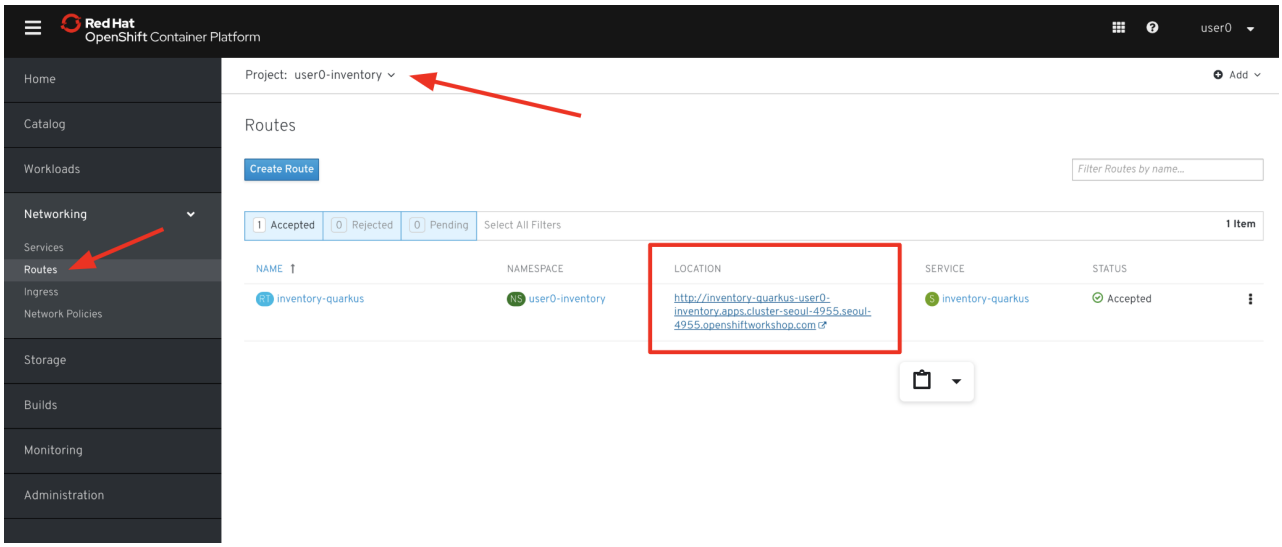
```
oc describe dc/inventory-quarkus | egrep 'Readiness|Liveness'
```

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

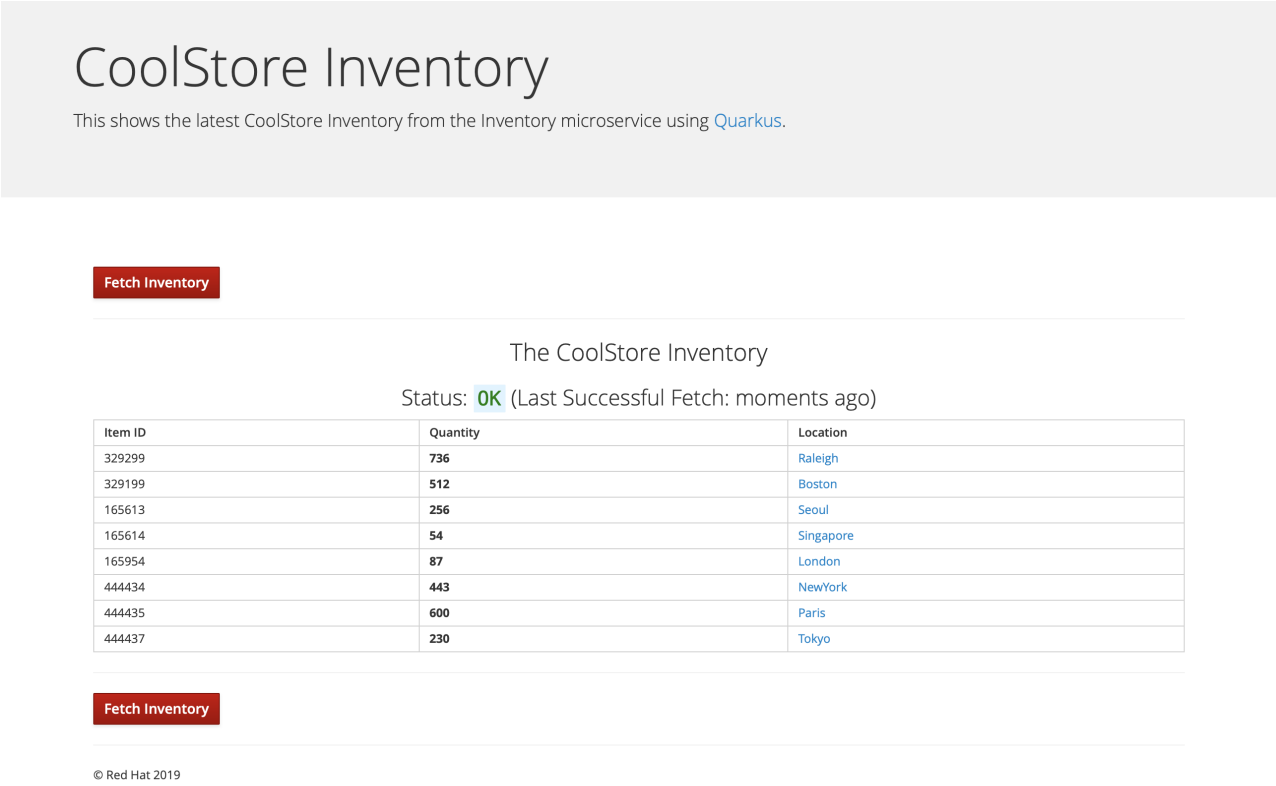
In the next step, we'll exercise the probe and watch as it fails and OpenShift recovers the application.

18. Exercise Health Check

From the project status page, click on the route link to open the sample application UI:



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



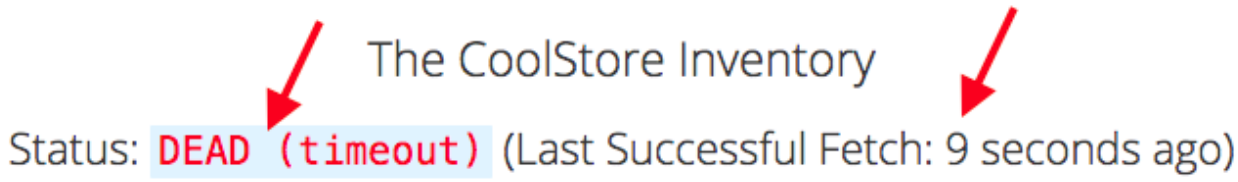
The app will begin polling the inventory as before and report success:

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

```
oc rsh dc/inventory-quarkus pkill java
```

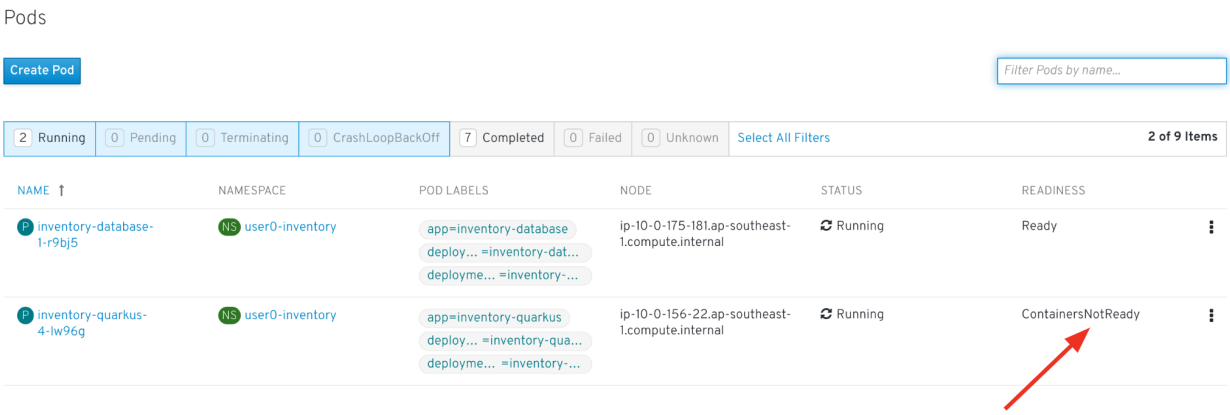
This will execute the Linux **pkill** command to stop the running Java process in the container.

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.



At this point, return to the OpenShift web console and click on the *Pods* on the left menu. Notice that the **ContainersNotReady** indicates the application is failing its *readiness*

probe:



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 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:

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.

Quarkus brings in a number of concepts and APIs from the Java EE community, so your existing Java EE skills can be re-used to bring your applications into the modern world of containers, microservices and cloud deployments.

Quarkus will be one of many components of Red Hat OpenShift Application Runtimes soon. **Stay tuned!!** In the next lab, you'll use Spring Boot, another popular framework, to implement additional microservices. Let's go!

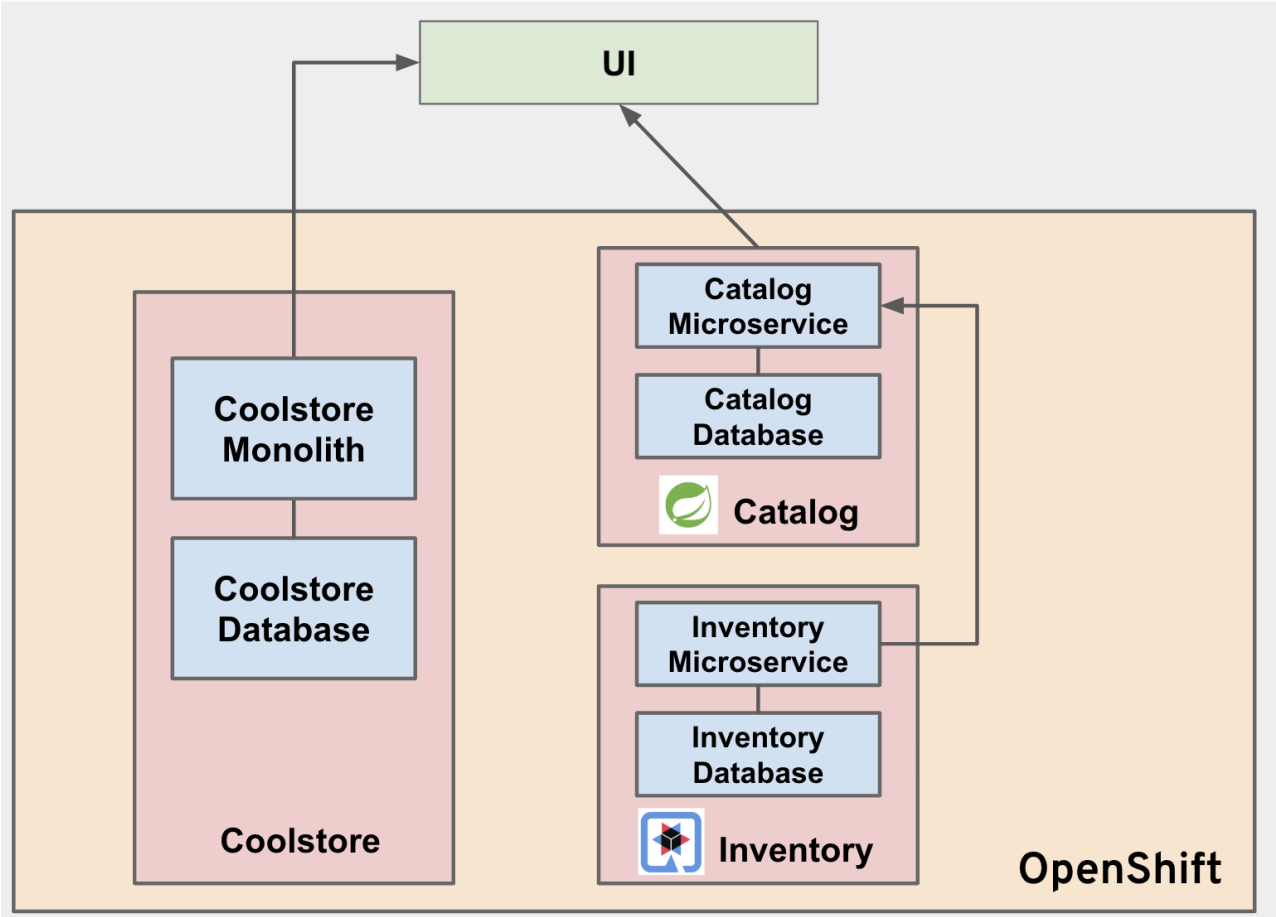
Break Monolith Apart - II

Lab4 - Breaking the 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 the Spring Framework. The reason for using Spring for this service is to introduce you to Spring Development, and how Red Hat Runtimes helps to make Spring development on Kubernetes easy. In real life, the reason for choosing Spring 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.

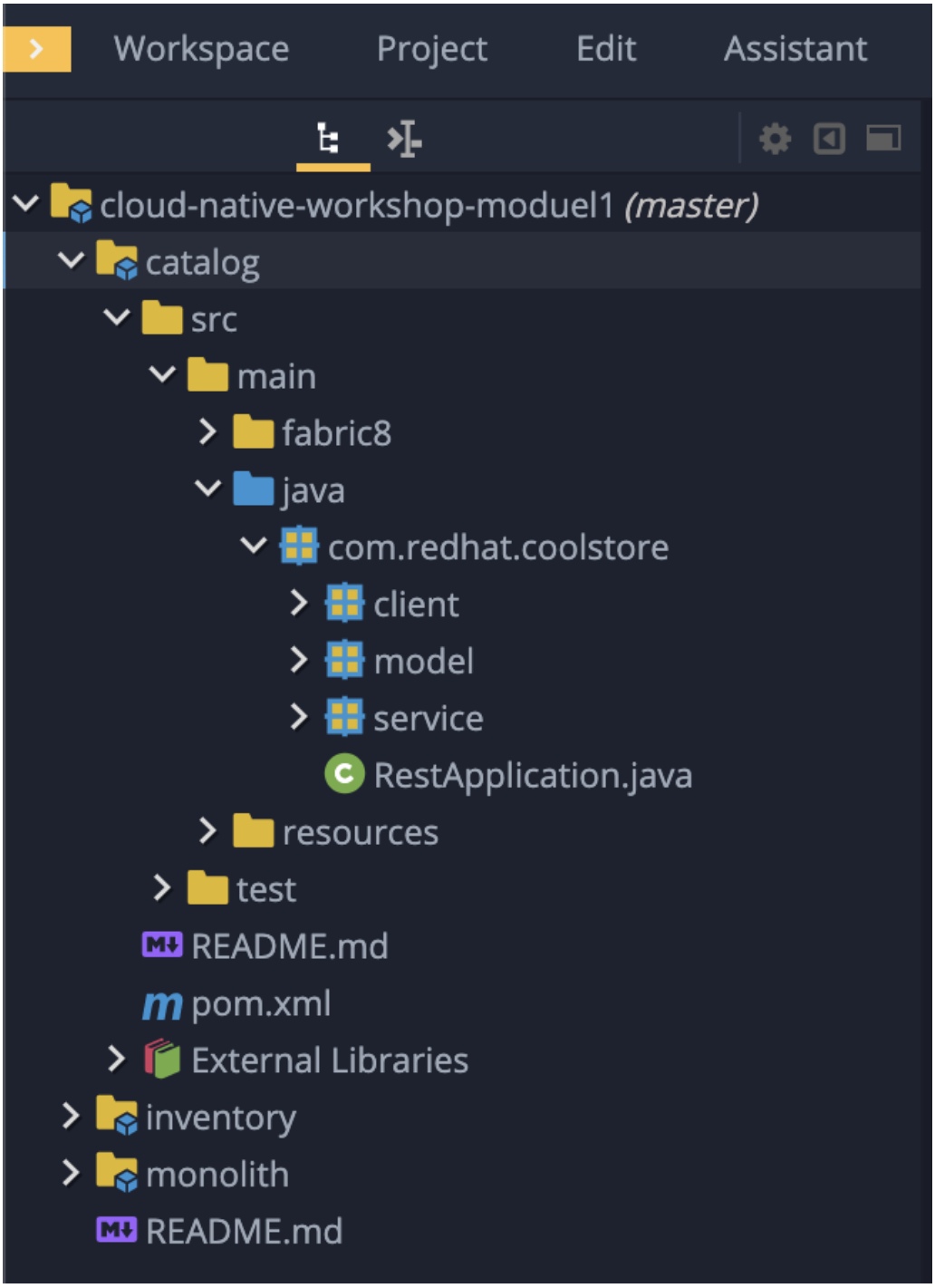
Aggregate microservices calls

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, right-click on *catalog* and then change a directory to catalog path on Terminal.



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.

> Click on the catalog folder in the project explorer and navigate to see its folders and files.

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 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 project.

```
<dependencyManagement>
<dependencies>
  <dependency>
    <groupId>me.snowdrop</groupId>
    <artifactId>spring-boot-bom</artifactId>
    <version>${spring-boot.bom.version}</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. Adding 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.

> NOTE: 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:

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

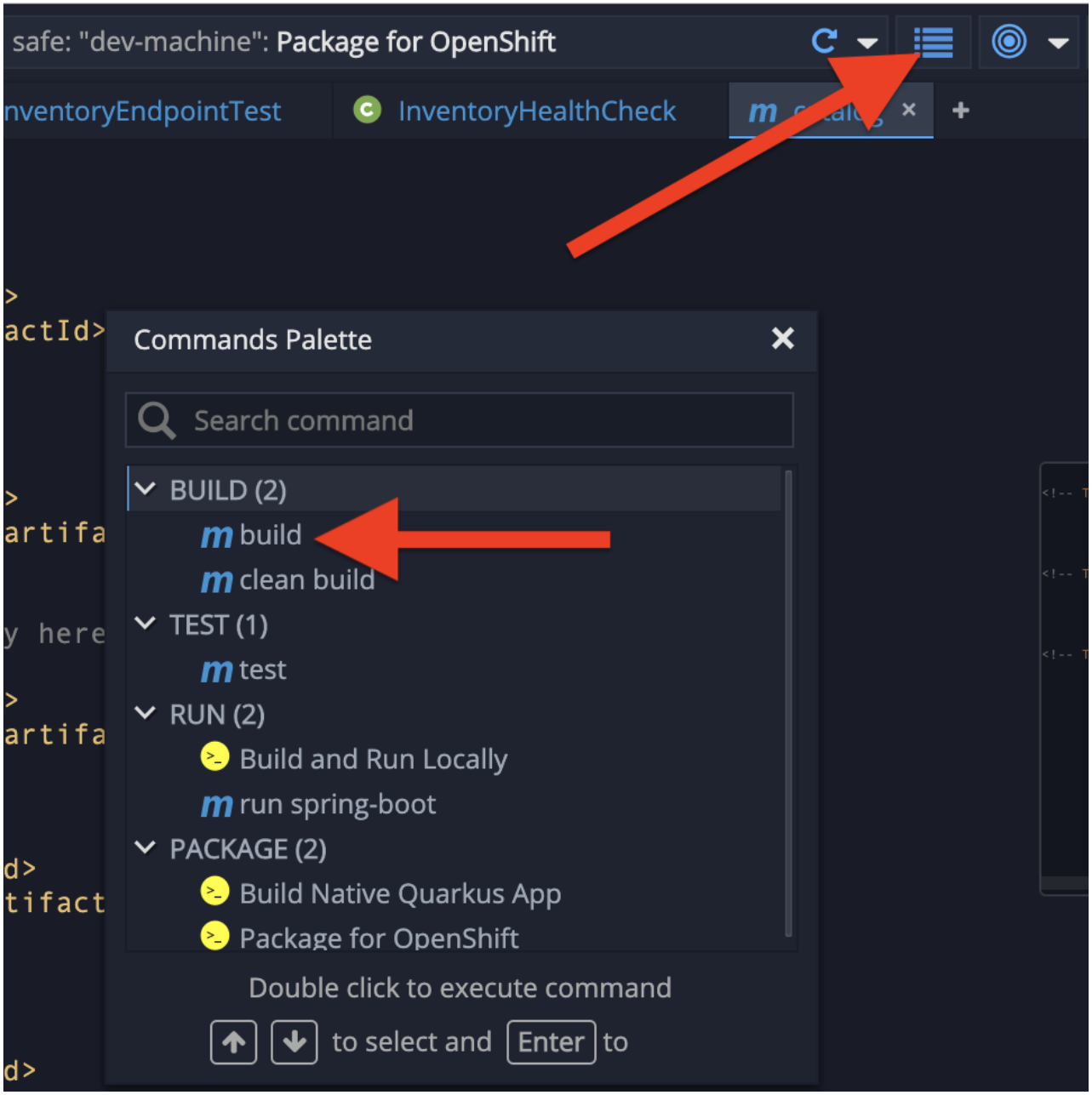
We will also make use of Java Persistance API (JPA) so we need to add the following to *pom.xml* at the `<!-- TODO: Add jdbc dependency here -->` marker:

```
<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:

```
<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>
```

Use the command palette and select ‘Build’ to build and package the app using Maven to make sure the changed code still compiles:



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

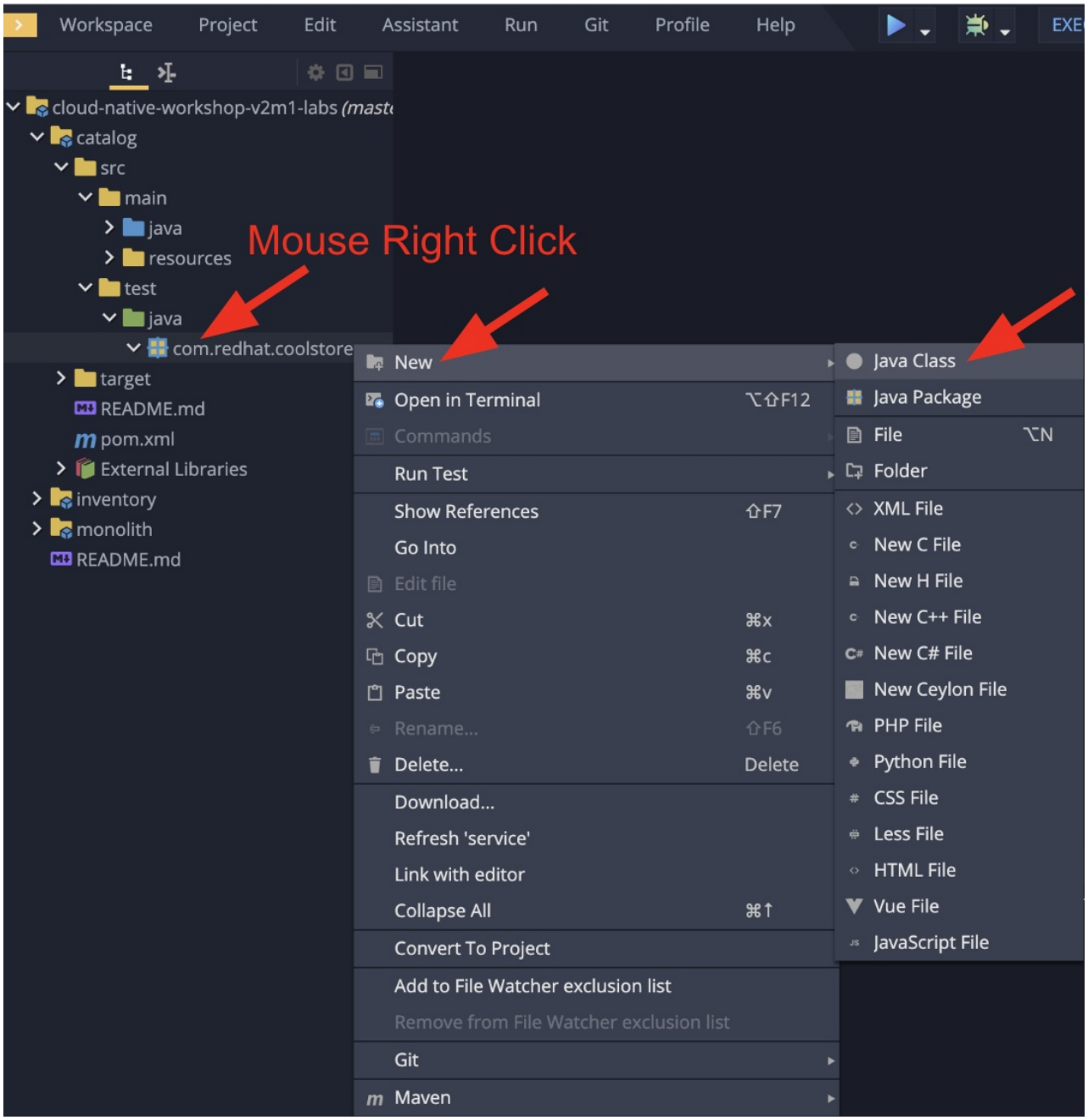
Now you've seen how to get started with Spring Boot development on Red Hat Runtimes.

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

4. Create Domain Objects

Before we create the database repository class to access the data it's good practice to create test cases for the different methods that we will use.

Right-click on the `src/test/java/com.redhat.coolstore.service` package, and select *New* > *Java Class*. Type `ProductRepositoryTest` into the dialog box and press **OK** which will create an empty class file.



Replace the content of this new file with the below code:

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

import java.util.List;
import java.util.stream.Collectors;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import static org.assertj.core.api.Assertions.assertThat;

import com.redhat.coolstore.model.Product;


@RunWith(SpringRunner.class)
@SpringBootTest
public class ProductRepositoryTest {

    //TODO: Insert Catalog Component here

    //TODO: Insert test_readOne here

    //TODO: Insert test_readAll here

}
```

Next, inject a handle to the future repository class which will provide access to the underlying data repository. It is injected with Spring's **@Autowired** annotation which locates, instantiates, and injects runtime instances of classes automatically, and manages their lifecycle (much like Java EE and it's CDI feature). Add these at the `<!-- TODO: Insert Catalog Component here -->` marker:

```
    @Autowired
    private ProductRepository repository;
```

The *ProductRepository* should provide a method called *findById(String id)* that returns a product and collect that from the database. We test this by querying for a product with id "444434" which should have name **Pebble Smart Watch**. The pre-loaded data comes from the *src/main/resources/schema.sql* file.

Add these at the `<!-- TODO: Insert test_readOne here -->` marker:

```
    @Test
    public void test_readOne() {
        Product product = repository.findById("444434");
        assertThat(product).isNotNull();
        assertThat(product.getName()).as("Verify product name").isEqualTo("Pebble Smart Watch");
        assertThat(product.getQuantity()).as("Quantity should be ZERO").isEqualTo(0);
    }
```

The *ProductRepository* should also provide a methods called *readAll()* that returns a list of all products in the catalog. We test this by making sure that the list contains a “Red Fedora”, “Forge Laptop Sticker” and “Oculus Rift”. Again, add these at the `<!-- TODO: Insert test_readAll here -->` marker:

```
@Test
public void test_readAll() {
   List<Product> productList = repository.readAll();
   assertThat(productList)
     .isNotNull()
     .isNotEmpty();
   List<String> names = productList.stream().map(Product::getName).collect(Collectors.toList());
   assertThat(names).contains("Red Fedora","Forge Laptop Sticker","Oculus Rift");
}
```

5. Implement the database repository

We are now ready to implement the database repository.

Use the same procedure to create a new Java Class in the `src/main/java/com/redhat/coolstore/service` package called `ProductRepository`. Replace the empty class with the following code:

```
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

}
```

> NOTE: 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:

```
@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:

```
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 methods that are used in the test. 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:

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

The *ProductRepositoryTest* also used another method called *findById(String id)* that should return a Product. The implementation of that method using the *JdbcTemplate* and *RowMapper* looks like this. Add these at the `<!-- TODO: Create a method for returning one product -->` marker:

```
public Product findById(String id) {
    return this.jdbcTemplate.queryForObject("SELECT * FROM catalog WHERE itemId = '" + id + "'",
rowMapper);
}
```

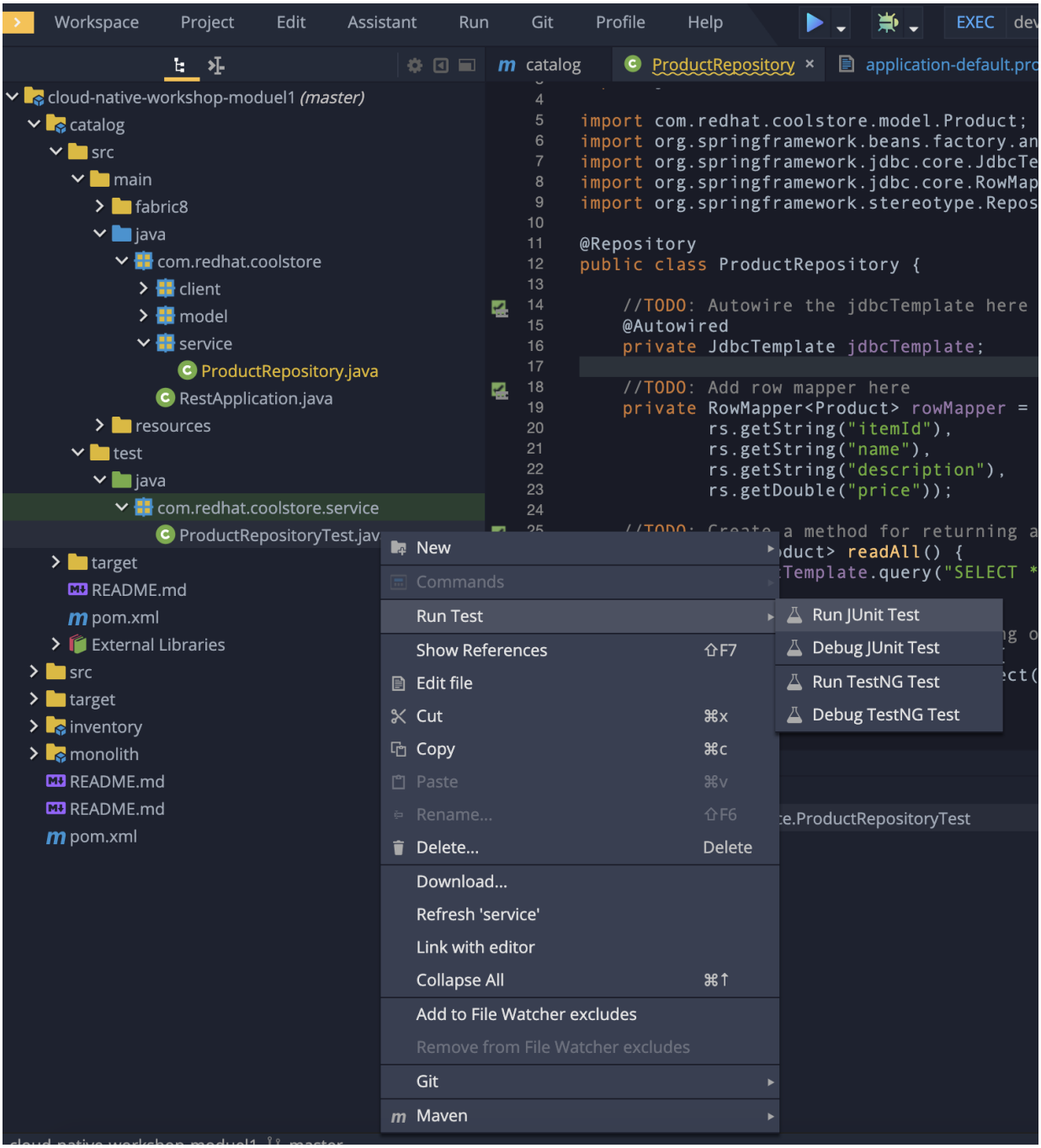
The *ProductRepository* should now have all the components, but we still need to tell spring how to connect to the database. For local development we will use the H2 in-memory database. When deploying this to OpenShift we are instead going to use the PostgreSQL database, which matches what we are using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

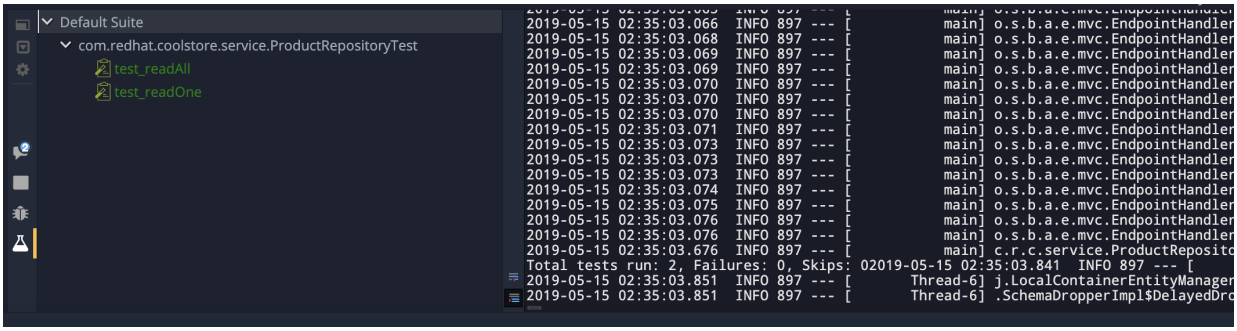
```
spring.datasource.url=jdbc:h2:mem:catalog;DB_CLOSE_ON_EXIT=FALSE
spring.datasource.username=sa
spring.datasource.password=sa
spring.datasource.driver-class-name=org.h2.Driver
```

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

Now we are ready to run the test to verify that everything works. Right-click on the `src/test/java/com/redhat/coolstore/service` package and select *Run Test* > *Run JUnit Test*.



The test should be successful and you should see green color *test_realAll*, *test_realOne* in Default Suite window.



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

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.

6. 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` in the `src/main/java/com/redhat/coolstore/service` package.

Replace the empty class with this code:

```
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 is 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. Let's again first start by creating a test case for our endpoint. We need two endpoints, one that exposes for GET calls to `/services/products` that will return all product in the catalog as JSON array, and the second one exposes GET calls to `/services/produc/{prodId}` which will return a single Product as a JSON Object. Let's again start by creating a test case.

Create the test case by creating a new class file called `CatalogEndpointTest` in the `src/test/java/com/redhat/coolstore/service` package.

Add the following code to the test case and make sure to *review* it without any codes change so that you understand how it works.

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

```
import com.redhat.coolstore.model.Inventory;
import com.redhat.coolstore.model.Product;
import io.specto.hoverfly.junit.rule.HoverflyRule;
import org.junit.ClassRule;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.HttpMethod;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.test.context.junit4.SpringRunner;

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

import static io.specto.hoverfly.junit.dsl.HttpBodyConverter.json;
import static io.specto.hoverfly.junit.dsl.ResponseCreators.success;
import static io.specto.hoverfly.junit.dsl.ResponseCreators.serverError;
import static io.specto.hoverfly.junit.dsl.matchers.HoverflyMatchers.startsWith;
import static org.assertj.core.api.Assertions.assertThat;
import static io.specto.hoverfly.junit.core.SimulationSource.dsl;
import static io.specto.hoverfly.junit.dsl.HoverflyDsl.service;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class CatalogEndpointTest {

    @Autowired
    private TestRestTemplate restTemplate;

    //TODO: Add ClassRule for HoverFly Inventory simulation

    @Test
    public void test_retriving_one_proudct() {
        ResponseEntity<Product> response
                = restTemplate.getForEntity("/services/product/329199", Product.class);
        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
        assertThat(response.getBody())
                .returns("329199",Product::getItemId)
                .returns("Forge Laptop Sticker",Product::getName)
    //TODO: Add check for Quantity
                .returns(8.50,Product::getPrice);
    }


    @Test
    public void check_that_endpoint_returns_a_correct_list() {

        ResponseEntity<List<Product>> rateResponse =
```

```
            restTemplate.exchange("/services/products",
                HttpMethod.GET, null, new ParameterizedTypeReference<List<Product>>() {
                });

        List<Product> productList = rateResponse.getBody();
        assertThat(productList).isNotNull();
        assertThat(productList).isNotEmpty();
        List<String> names = productList.stream().map(Product::getName).collect(Collectors.toList());
        assertThat(names).contains("Red Fedora","Forge Laptop Sticker","Oculus Rift");

        Product fedora = productList.stream().filter( p -> p.getItemId().equals("329299")).findAny().get();
        assertThat(fedora)
            .returns("329299",Product::getItemId)
            .returns("Red Fedora", Product::getName)
    //TODO: Add check for Quantity
            .returns(34.99,Product::getPrice);
    }

}
```

Now we are ready to implement the *CatalogEndpoint*.

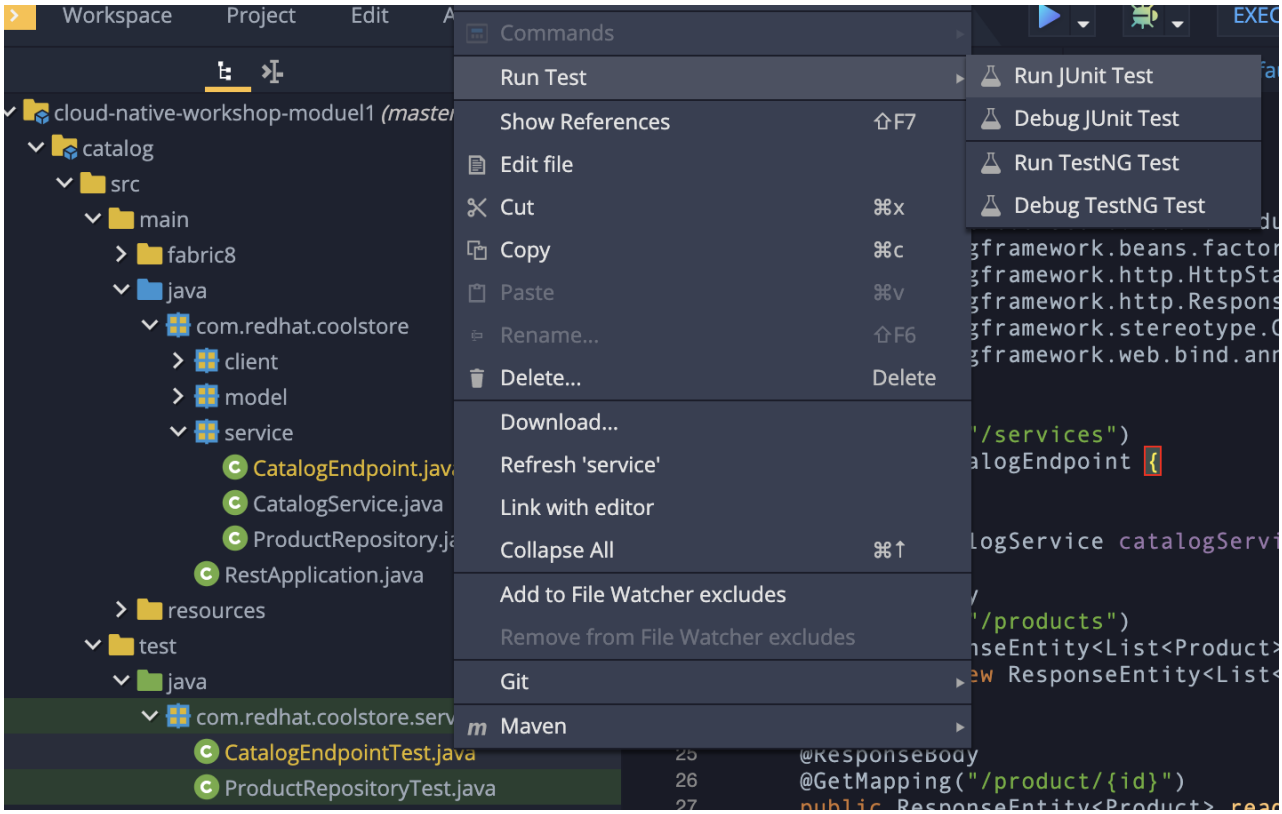
Start by creating a new class called `CatalogEndpoint` in the `src/main/java/com/redhat/coolstore/service` package.

Replace the contents with this code:

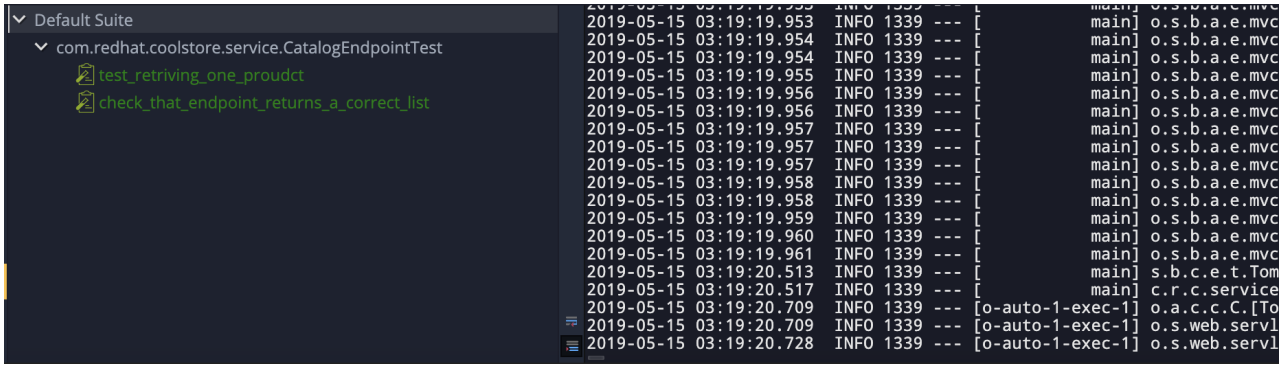
```
package com.redhat.coolstore.service;

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 default uses Jackson to serialize or map Java objects to JSON and vice versa. Because Jackson extends upon JAX-B and does can automatically parse simple Java structures and parse them into JSON and vice verse and since our `Product.java` is very simple and only contains basic attributes we do not need to tell Jackson how to parse between Product and JSON.

Now you can run the *CatalogEndpointTest* and verify that it works via **Run Junit Test**. Right-click on the `CatalogEndpointTest` and select *Run Test* > *Run JUnit Test*.



The test should be successful and you should see green color *test_retriving_one_proudct*, _check_that_endpoint_returns_a_correct_list<> in Default Suite window.



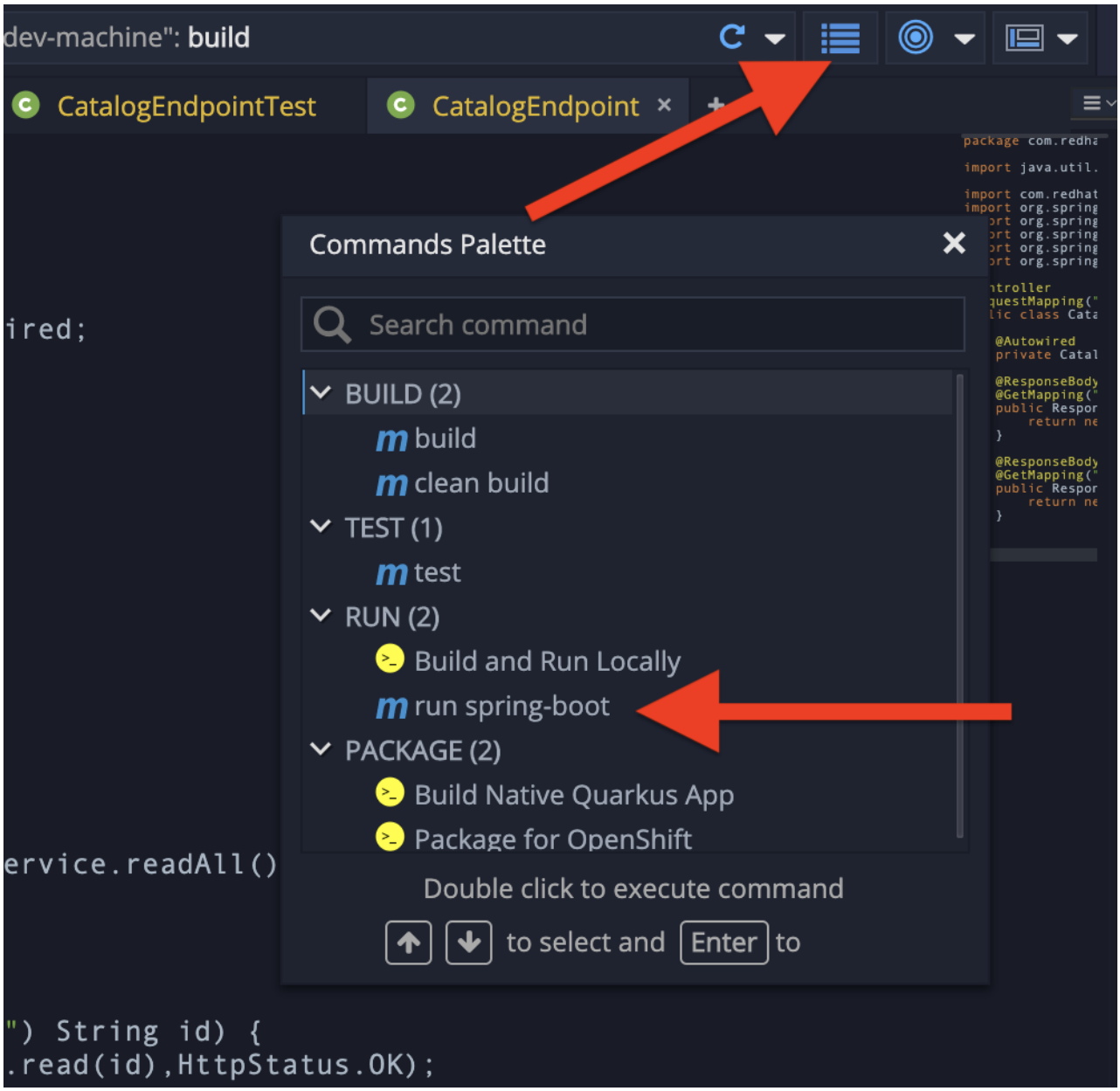
You can also run the following command via `CodeReady Workspaces Terminal` to verify the test cases.

```
cd /projects/cloud-native-workshop-v2m1-labs/catalog/
```

```
mvn verify -Dtest=CatalogEndpointTest
```

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

Start the application via CodeReady Workspaces **RUN** Menu:



Wait for the application to start. Then we can verify the endpoint by running the following command in Eclipse Terminal:

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

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

```
{
  "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 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.

> NOTE: Make sure to stop the service by closing `run spring-boot` tab window in CodeReady Workspace.

7. Get inventory data

So far our application has been kind of straight forward, but our monolith code for the catalog is also returning the inventory status. In the monolith since both the inventory data and catalog data are in the same database we used a `OneToOne` mapping in JPA like this:

```
@OneToOne(cascade = CascadeType.ALL,fetch=FetchType.EAGER)
@PrimaryKeyJoinColumn
private InventoryEntity inventory;
```

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:

I. 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 request to the server. If we have a slow internet connection this may cause issues.

II. 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 provides nice capabilities to do this and if you are interested to learn more about this, please checkout the Coolstore Microservices example: Here

III. 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 I or II here. Instead we are going to use option III and extend our Catalog to call the Inventory service.

8. Extending the test

In the Test-Driven Development style, let's first extend our test to test the Inventory functionality (which doesn't exist).

Open *src/test/java/com/redhat/coolstore/service/CatalogEndpointTest.java* again.

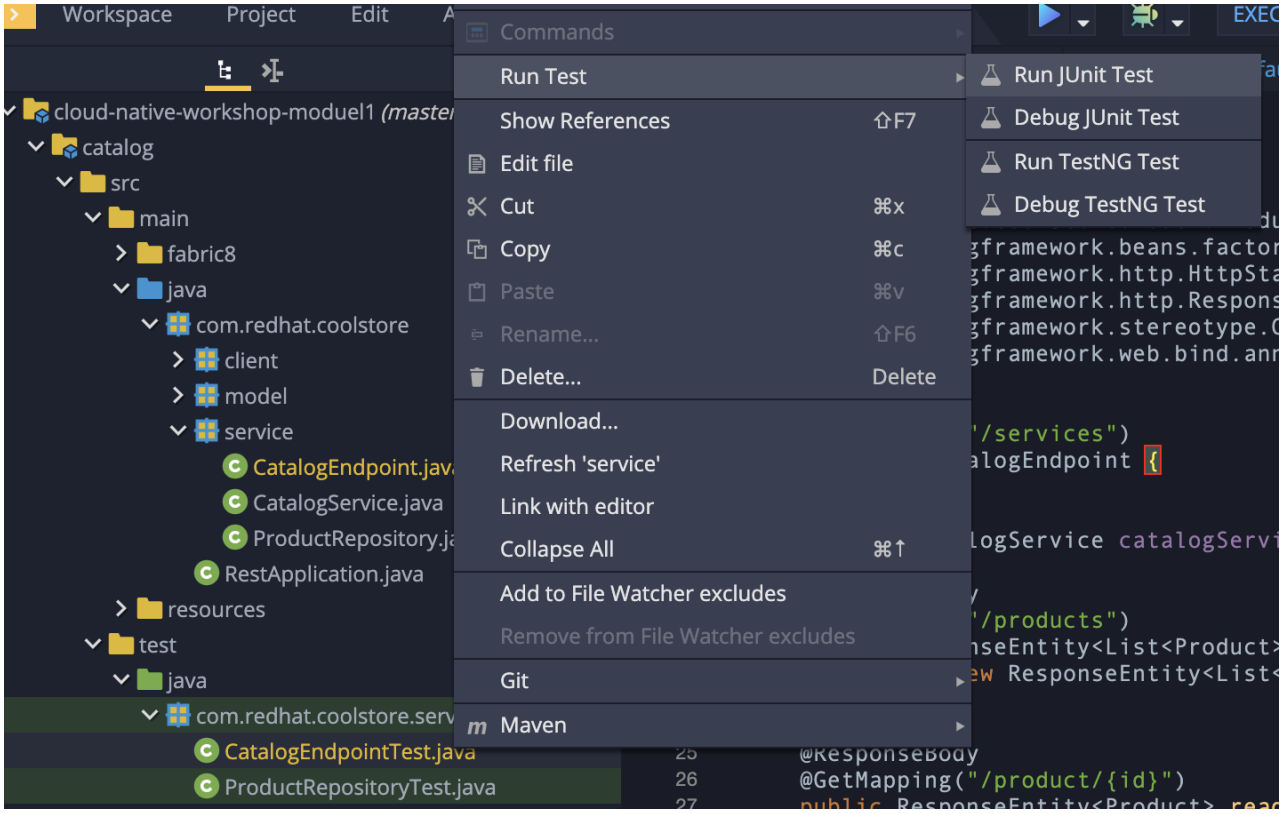
Now at the markers **//TODO: Add check for Quantity** add the following line:

```
.returns(9999,Product::getQuantity)
```

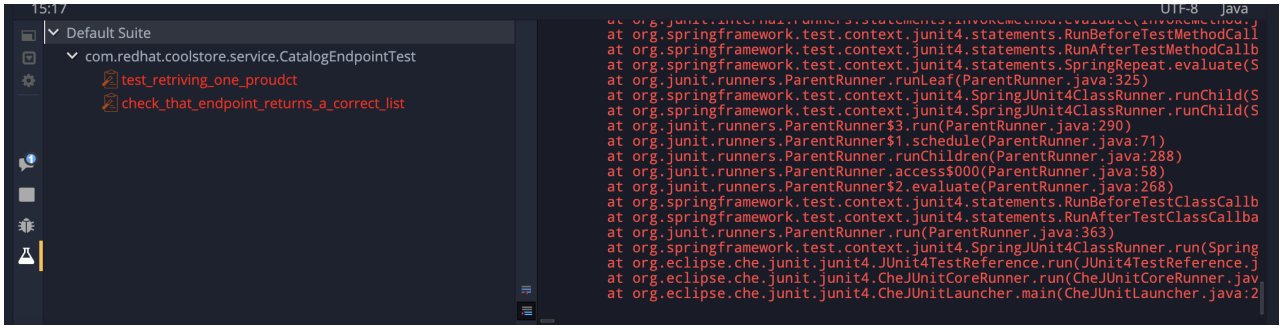
And add it to the second test as well at the remaining *//TODO: Add check for Quantity* marker:

```
.returns(9999,Product::getQuantity)
```

Now you can run the *CatalogEndpointTest* and verify that it **fails** via *Run Junit Test*:



The test *should fail* and you should see red color **test_retriving_one_proudct**, **check_that_endpoint_returns_a_correct_list** in Default Suite window.



The test fails because we are trying to call the Inventory service which is not runninmg.

We will soon implement the code to call the inventory service, but first we need a away to test this service without having to rely on the inventory services to be up and running. For that we are going to use an API Simulator called HoverFly and particular it's capability to simulate remote APIs. HoverFly is very convenient to use with Unit test and all we have to do is to add a **ClassRule** that will simulate all calls to inventory. Open the file to insert the code at the `//TODO: Add ClassRule for HoverFly Inventory simulation` marker in *CatalogEndpointTest* class:

```
@ClassRule
public static HoverflyRule hoverflyRule = HoverflyRule.inSimulationMode(dsl(
        service("inventory:8080")
//                .andDelay(2500, TimeUnit.MILLISECONDS).forMethod("GET")
                .get(startsWith("/services/inventory"))
//                .willReturn(serverError())
                .willReturn(success("[{\"itemId\":\"329199\",\"quantity\":9999}]", "application/json"))

));
```

This *ClassRule* means that if our tests are trying to call our inventory url, HoverFly will intercept this and respond with our hard coded response instead.

We will soon use the `// commented-out` lines, so keep them in there!

9. Implementing the Inventory Client

Since we now have a nice way to test our service-to-service interaction 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 integrate with Hystrix which gives us 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 todo is to 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` class in the `src/main/java/com/redhat/coolstore/client/` package in the project explorer.

Add the following code to the file:

```
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

}
```

There is one more thing that we need to do which is to tell Feign where the inventory service is running. Before that notice that we are setting the `@FeignClient(name="inventory")`.

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

Add these properties to it at the `#TODO: Configure netflix libraries` marker:

```
inventory.ribbon.listOfServers=inventory:8080
feign.hystrix.enabled=true
```

By setting *inventory.ribbon.listOfServers* we are hard coding the actual URL of the service to **inventory:8080**. 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. Later on we will update this value to reflect our URL when deploying to OpenShift.

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:

```
    @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:

```
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 **//import com.redhat.coolstore.client.InventoryClient** near the top

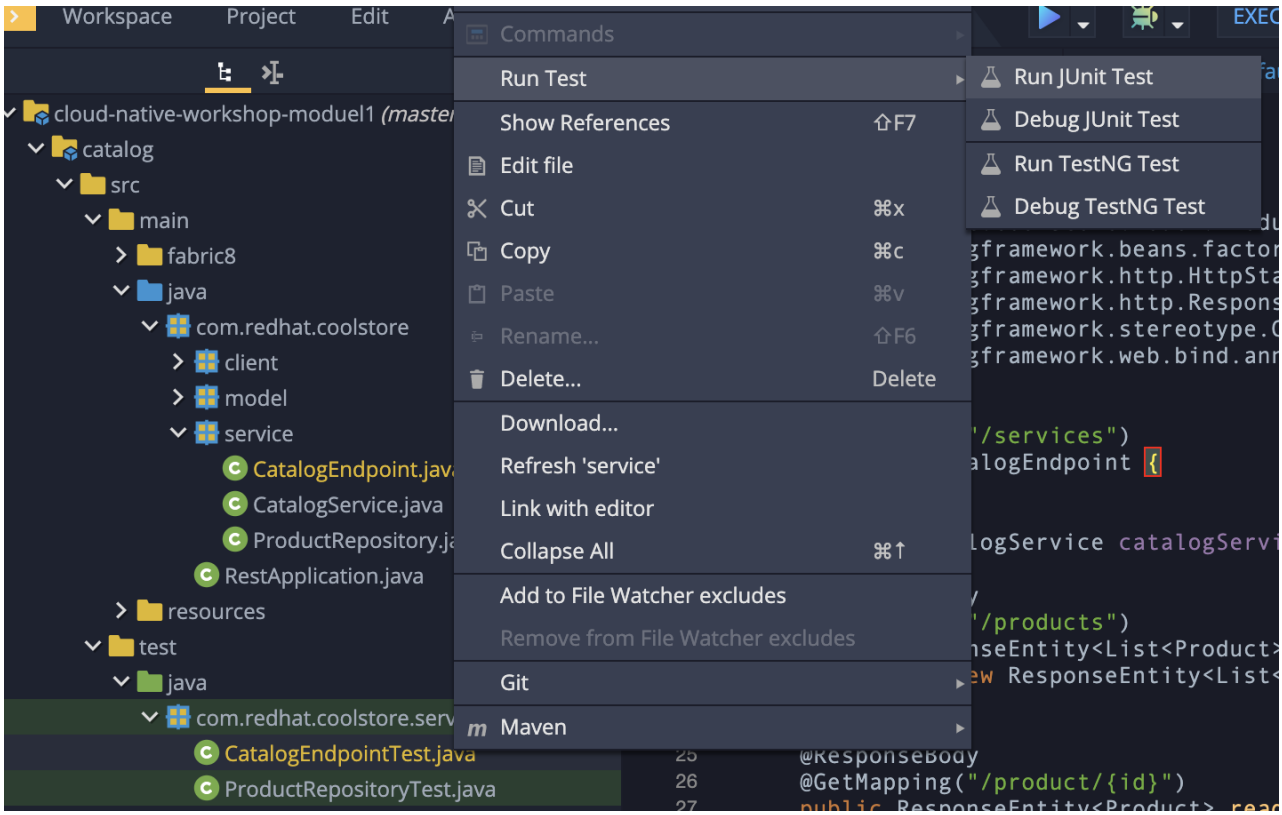
```
import com.redhat.coolstore.client.InventoryClient;
```

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

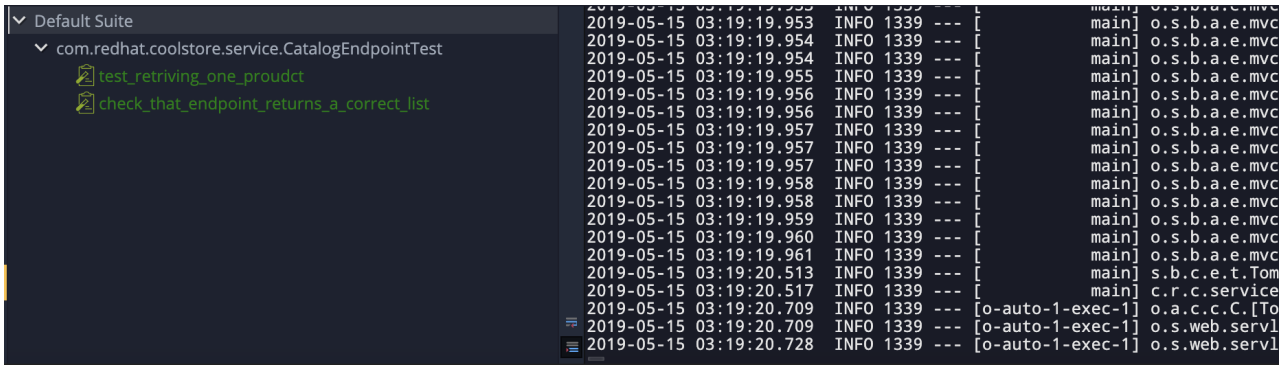
```
productList.forEach(p -> {
 JSONArray jsonArray = new JSONArray(this.inventoryClient.getInventoryStatus(p.getItemId()));
 List<String> quantity = IntStream.range(0, jsonArray.length())
   .mapToObj(index -> ((JSONObject)jsonArray.get(index))
   .optString("quantity")).collect(Collectors.toList());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

> NOTE: 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.

Now you can run the *CatalogEndpointTest* and verify that it works via **Run Junit Test**:



The test should be successful and you should see green color **test_retriving_one_proudct**, **check_that_endpoint_returns_a_correct_list** in Default Suite window.



So even if we don't have any inventory service running we can still run our test. However to actually run the service using `mvn spring-boot:run` we need to have an inventory service or the calls to `/services/products/` will fail. We will fix this in the next step.

Congratulations!

You now have the framework for retrieving products from the product catalog and enriching the data with inventory data from an external service. But what if that external inventory service does not respond? That's the topic for the next step.

10. 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 therefor 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.

Open: *src/main/java/com/redhat/coolstore/client/InventoryClient.java*

And paste this into it at the `//TODO: Add Fallback factory here` marker:

```
//TODO: Add Callback Factory Component
@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:

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

11. Test the Fallback

Now let's see if we can test the fallback. Optimally we should create a different test that fails the request and then verify the fallback value, however because we are limited in time we are just going to change our test so that it returns a server error and then verify that the test fails.

Open *src/test/java/com/redhat/coolstore/service/CatalogEndpointTest.java* and change the following lines:

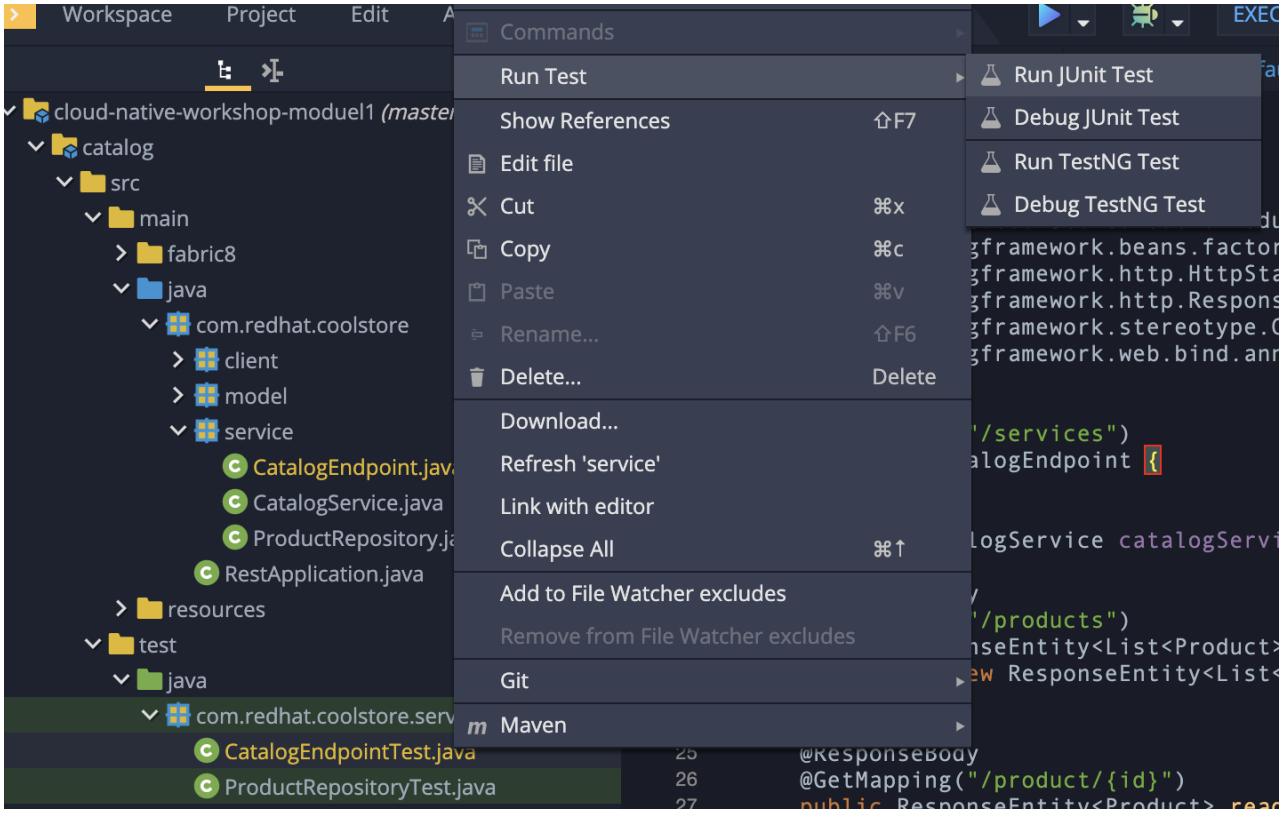
```
@ClassRule
public static HoverflyRule hoverflyRule = HoverflyRule.inSimulationMode(dsl(
        service("inventory:8080")
//                .andDelay(2500, TimeUnit.MILLISECONDS).forMethod("GET")
                .get(startsWith("/services/inventory"))
//                .willReturn(serverError())
                .willReturn(success("[{\"itemId\":\"329199\",\"quantity\":9999}]", "application/json"))

));
```

TO

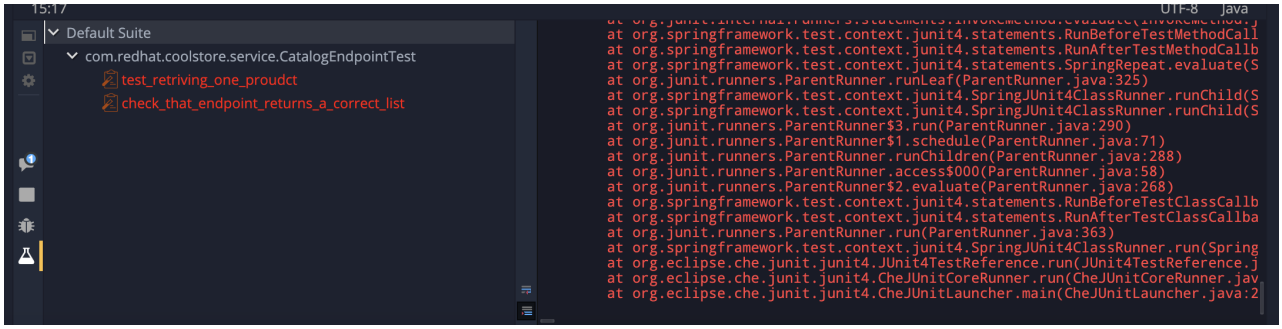
```
@ClassRule
public static HoverflyRule hoverflyRule = HoverflyRule.inSimulationMode(dsl(
        service("inventory:8080")
//                .andDelay(2500, TimeUnit.MILLISECONDS).forMethod("GET")
                .get(startsWith("/services/inventory"))
                .willReturn(serverError())
//                .willReturn(success("[{\"itemId\":\"329199\",\"quantity\":9999}]", "application/json"))

));
```

Notice that the Hoverfly Rule will now return `serverError` for all requests to inventory.

Now you can run the *CatalogEndpointTest* and verify that it **fails** via **Run Junit Test**:



The test *should fail* and you _should see red color `test_retriving_one_proudct_`, `_check_that_endpoint_returns_a_correct_list` in Default Suite window.



So since even if our inventory service fails we are still returning inventory quantity -1. The test fails because we are expecting the quantity to be 9999.

Change back the class rule by re-commenting out the *.willReturn(serverError())* line so that we don't fail the tests like this:

```
@ClassRule
public static HoverflyRule hoverflyRule = HoverflyRule.inSimulationMode(dsl(
        service("inventory:8080")
//                .andDelay(2500, TimeUnit.MILLISECONDS).forMethod("GET")
                .get(startsWith("/services/inventory"))
//                .willReturn(serverError())
                .willReturn(success("[{\"itemId\":\"329199\",\"quantity\":9999}]", "application/json"))

));
```

Make sure the test works again by re-running the `CatalogEndpointTest` JUnit Test.

12. 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 is hard to debug and pinpoint issues with. We should therefore also have sane defaults for our services. You can add defaults by adding it to the configuration.

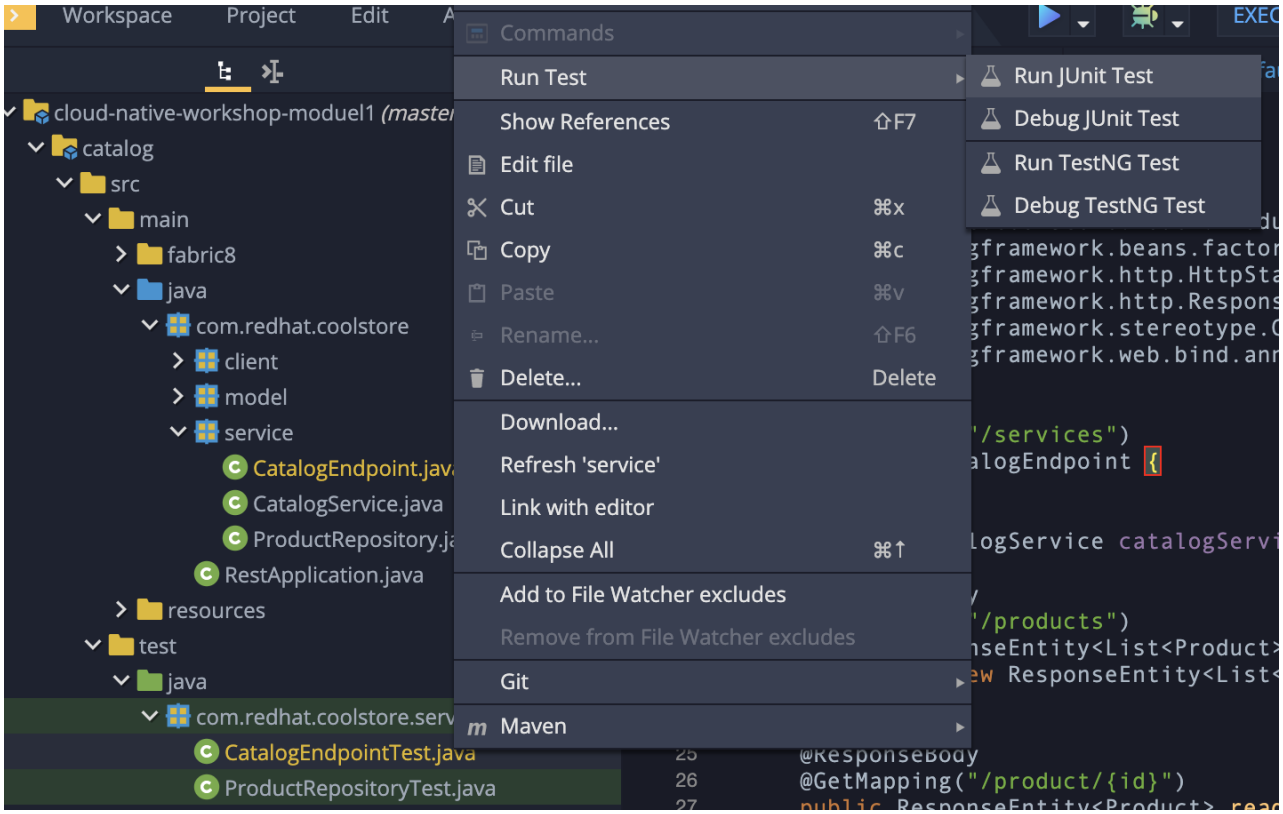
Open *src/main/resources/application-default.properties*

And add this line to it at the **#TODO: Set timeout to for inventory to 500ms** marker:

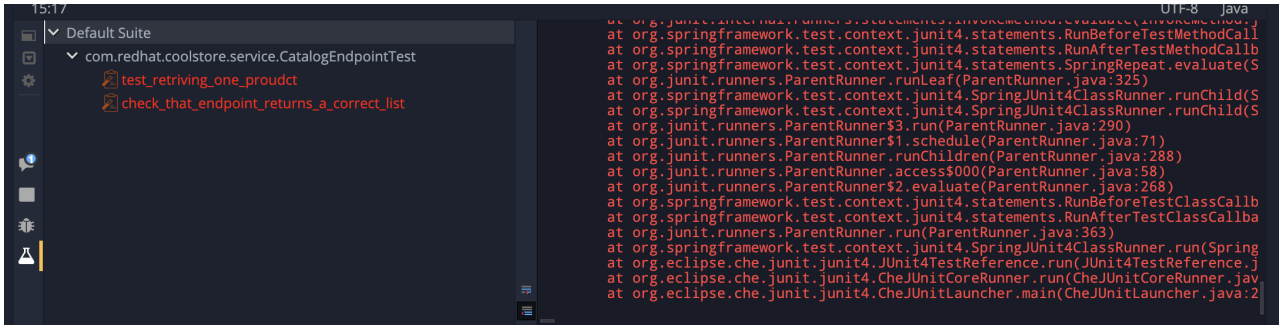
```
hystrix.command.inventory.execution.isolation.thread.timeoutInMilliseconds=500
```

Open *src/test/java/com/redhat/coolstore/service/CatalogEndpointTest.java* and un-comment the **.andDelay(2500, TimeUnit.MILLISECONDS).forMethod("GET")**

Now you can run the *CatalogEndpointTest* and verify that it **fails** via **Run Junit Test**:



The test *should fail* and you should see red color `test_retriving_one_proudct` , `check_that_endpoint_returns_a_correct_list` in Default Suite window.



This shows that the timeout works nicely. However, since we want our test to be successful **you should now comment out** `.andDelay(2500, TimeUnit.MILLISECONDS).forMethod("GET")` again and then verify that the test works by re-running the JUnit test.

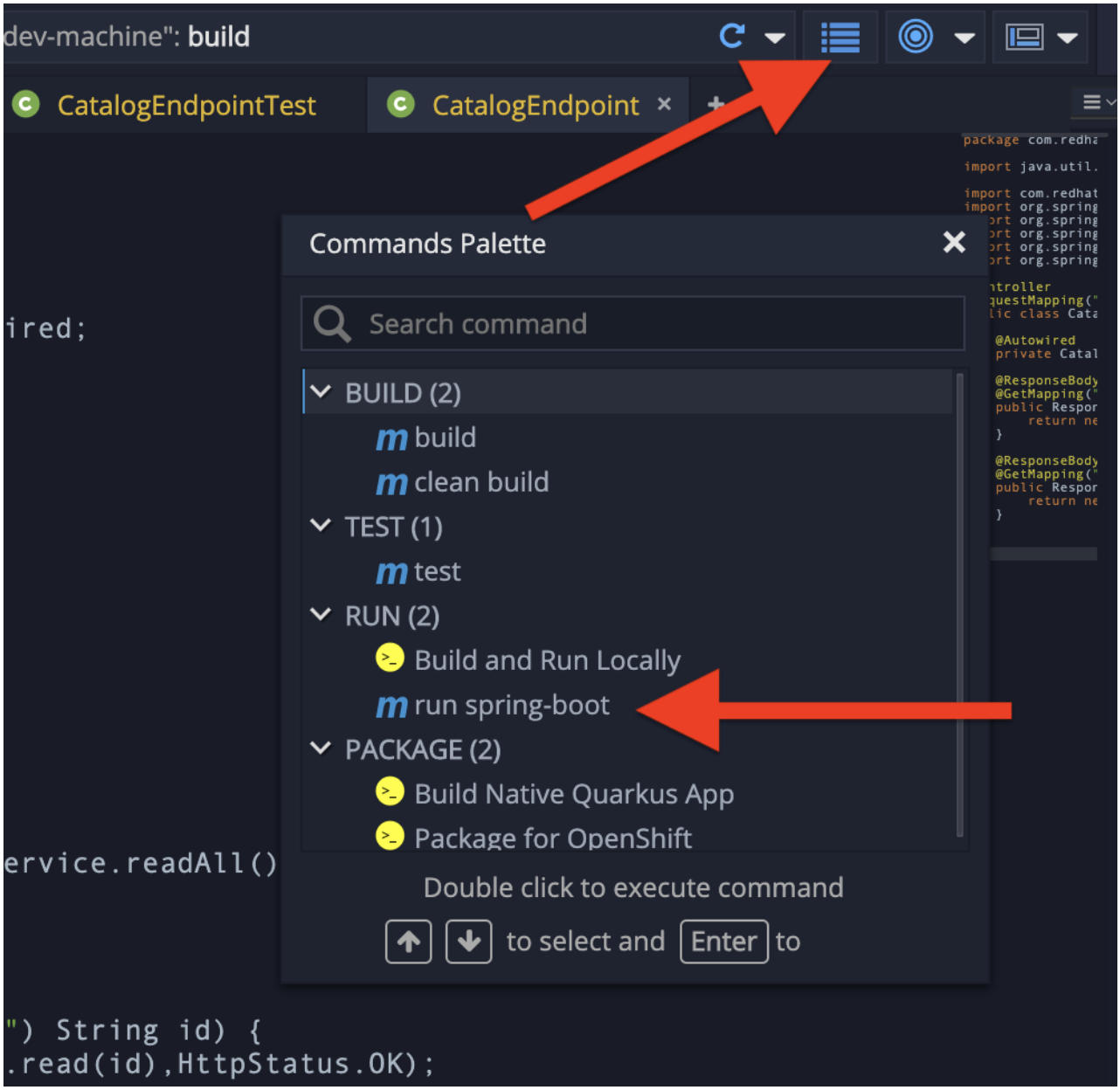
Congratulations!

You have now successfully executed the fourth step in this lab. In this step you've learned how to add Fallback logic to your class and how to add timeout to service calls. In the next step we now test our service locally before we deploy it to OpenShift.

13. Test Locally

As you have seen in previous steps, using the Spring Boot maven plugin (predefined in *pom.xml*), you can conveniently run the application locally and test the endpoint.

Start the application via CodeReady Workspaces **RUN** Menu:



Wait for the application to start. Then we can verify the endpoint by running the following command in Eclipse Terminal:

```
curl http://localhost:8081/services/product/329299 ; echo
```

You would see a JSON response like this:

```
{"itemId":"329299","name":"Red Fedora","desc":"Official Red Hat Fedora","price":34.99,"quantity":-1}%
```

> NOTE: Since we do not have an inventory service running locally the value for the quantity is -1, which matches the fallback value that we have configured.

The REST API returned a JSON object representing the inventory count for this product. Well done!

> NOTE: Make sure to stop the service by closing run spring-boot tab window in CodeReady Workspace.

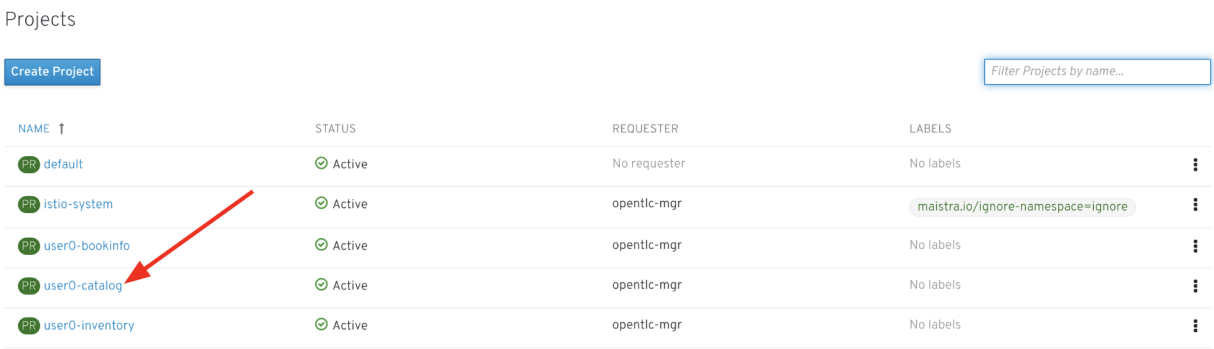
You have now successfully created your the Catalog service using Spring Boot and implemented basic REST API on top of the product catalog database. You have also learned how to deal with service failures.

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.

14. Create the OpenShift project

We have already deployed our coolstore monolith and inventory to OpenShift. In this step we will deploy our new Catalog microservice for our CoolStore application, so let's create a separate project to house it and keep it separate from our monolith and our other microservices.

Click on the name of the **userXX-catalog** project:



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

Next, we'll deploy your new microservice to OpenShift.

15. Deploy to OpenShift

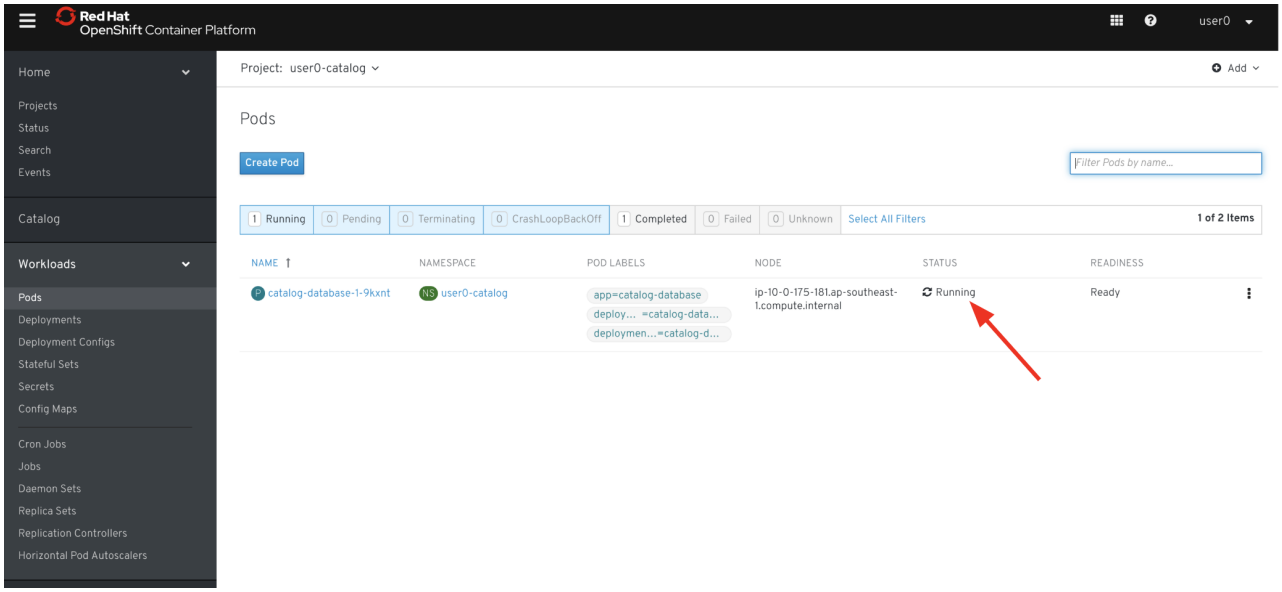
Now that you've logged into OpenShift, let's deploy our new catalog microservice:

Our production catalog microservice will use an external database (PostgreSQL) to house inventory data. First, deploy a new instance of PostgreSQL by executing via CodeReady Workspaces Terminal:

```
oc project userXX-catalog
```

```
oc new-app -e POSTGRESQL_USER=catalog \
        -e POSTGRESQL_PASSWORD=mysecretpassword \
        -e POSTGRESQL_DATABASE=catalog \
        openshift/postgresql:10 \
        --name=catalog-database
```

This will deploy the database to our new project.



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

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

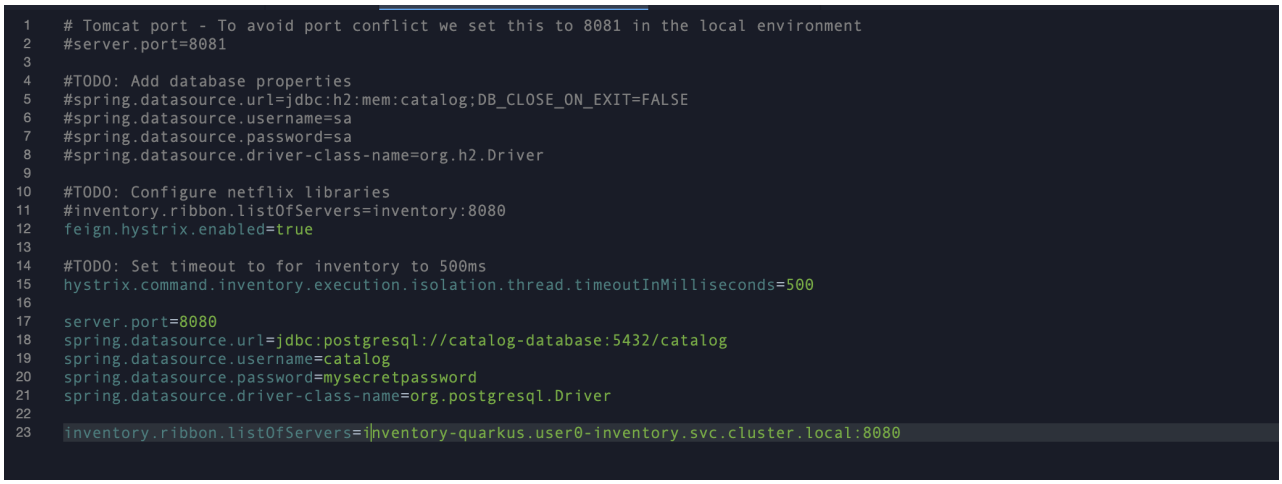
16. Update configuration

Open the file *src/main/resources/application-default.properties* in CodeReady Workspace.

Comment the local variables and add a remote variables. You can replace the whole contents with the following variables to the file:

> You have to replace **userXX** with your username in `inventory.ribbon.listOfServers=inventory-quarkus.userXX-inventory.svc.cluster.local:8080` .

```
# Tomcat port - To avoid port conflict we set this to 8081 in the local environment
#server.port=8081

#TODO: Add database 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

#TODO: Configure netflix libraries
#inventory.ribbon.listOfServers=inventory:8080
feign.hystrix.enabled=true

#TODO: Set timeout to for inventory to 500ms
hystrix.command.inventory.execution.isolation.thread.timeoutInMilliseconds=500

server.port=8080
spring.datasource.url=jdbc:postgresql://catalog-database:5432/catalog
spring.datasource.username=catalog
spring.datasource.password=mysecretpassword
spring.datasource.driver-class-name=org.postgresql.Driver

inventory.ribbon.listOfServers=inventory-quarkus.userXX-inventory.svc.cluster.local:8080
```



17. Build and Deploy

Build and deploy the project using the following command, which will use the maven plugin to deploy via CodeReady Workspaces Terminal:

```
cd /projects/cloud-native-workshop-v2m1-labs/catalog/
```

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

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

Then deploy the project using the following command, which will use the maven plugin to deploy via CodeReady Workspaces Terminal:

```
oc new-build registry.access.redhat.com/redhat-openjdk-18/openjdk18-openshift:1.5 --binary --name=catalog-springboot -l app=catalog-springboot
```

This build uses the new Red Hat OpenJDK Container Image, providing foundational software needed to run Java applications, while staying at a reasonable size.

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

```
oc start-build catalog-springboot --from-file=target/catalog-1.0.0-SNAPSHOT.jar --follow
```

Once the build is done, we'll deploy it as an OpenShift application:

```
oc new-app catalog-springboot
```

and expose your service to the world:

```
oc expose service catalog-springboot
```

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

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

Wait for that command to report replication controller "catalog-springboot-1" successfully rolled out before continuing.

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

And now we can access using curl once again to find a certain inventory:

```
export URL="http://$(oc get route | grep catalog | awk '{print $2}')"
```

```
curl $URL/services/product/329299 ; echo
```

The expected result data is here:

```
{"itemId":"329299","name":"Red Fedora","desc":"Official Red Hat Fedora","price":34.99,"quantity":736}
```

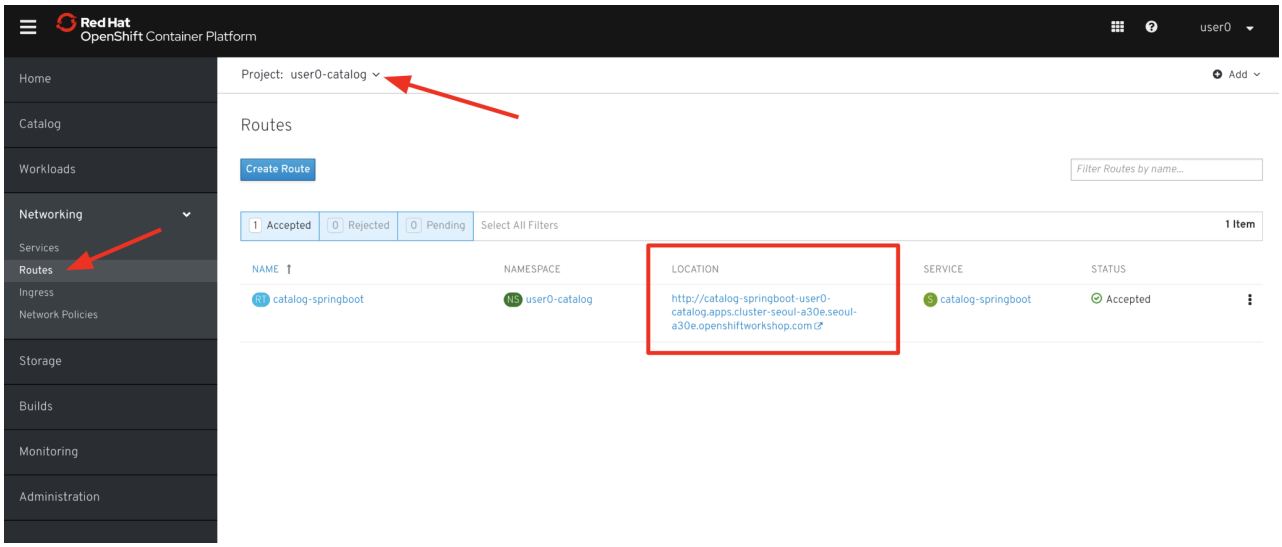
> **NOTE** if you do not get the expected output, make sure you replaced `userXX` in the `application-default.properties` file! If you forgot to do this, go back and make the change and re-build using the previous `mvn` command and re-deploy to OpenShift with the previous `oc start-build` command.

So now **Catalog** service is deployed to OpenShift. You can also see it in the Project Status in the OpenShift Console with running in 1 pod, along with the Postgres database pod.

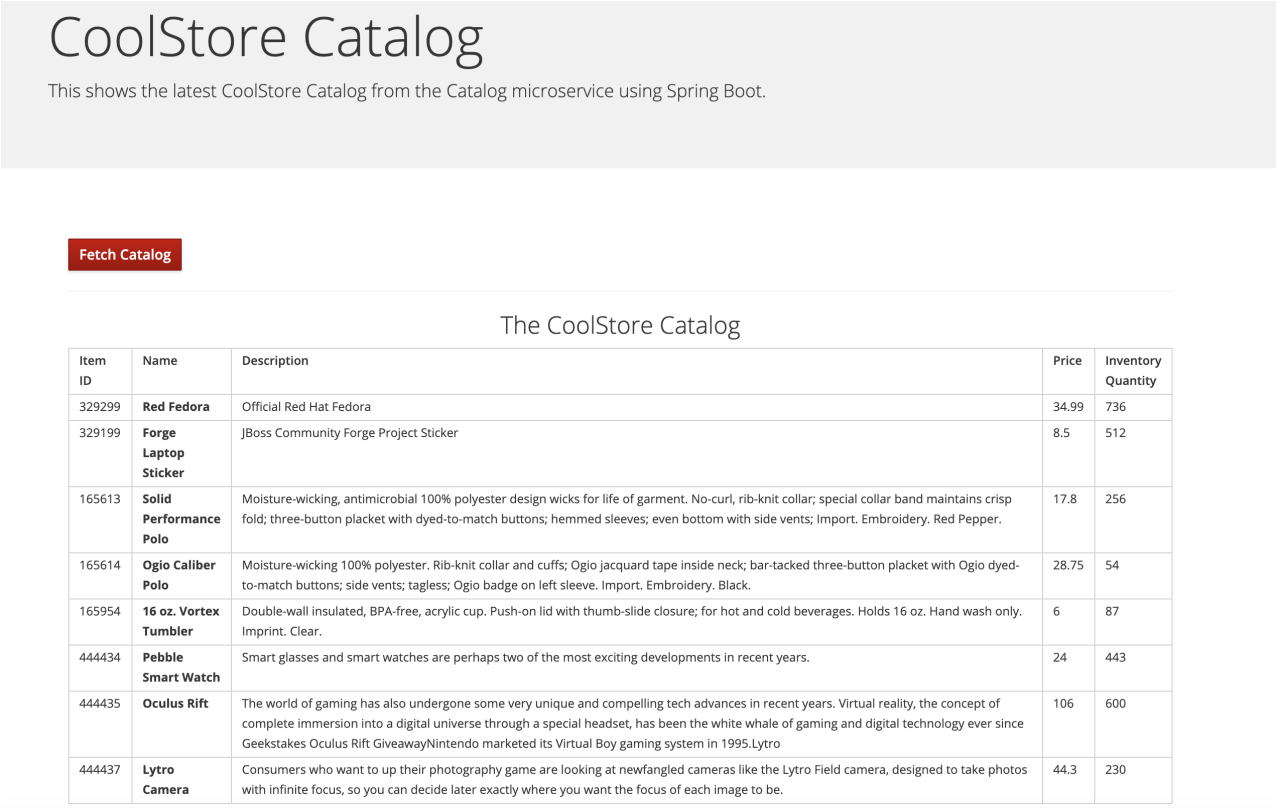
18. Access the application running on OpenShift

This sample project includes a simple UI that allows you to access the Inventory API. This is the same UI that you previously accessed outside of OpenShift which shows the CoolStore inventory. Click on the route URL at **Networking > Routes** in OpenShift web console to access the sample UI.

> You can also access the application through the link on Resources tab in the Project Status page.



The UI will refresh the catalog table every 2 seconds, as before.



Item ID	Name	Description	Price	Inventory Quantity
329299	**Red Fedora**	Official Red Hat Fedora	34.99	736
329199	**Forge Laptop Sticker**	JBoss Community Forge Project Sticker	8.5	512
165613	**Solid Performance Polo**	Moisture-wicking, antimicrobial 100% polyester design wicks for life of garment. No-curl, rib-knit collar; special collar band maintains crisp fold; three-button placket with dyed-to-match buttons; hemmed sleeves; even bottom with side vents; Import. Embroidery. Red Pepper.	17.8	256
165614	**Ogio Caliber Polo**	Moisture-wicking 100% polyester. Rib-knit collar and cuffs; Ogio jacquard tape inside neck; bar-tacked three-button placket with Ogio dyed-to-match buttons; side vents; tagless; Ogio badge on left sleeve. Import. Embroidery. Black.	28.75	54
165954	**16 oz. Vortex Tumbler**	Double-wall insulated, BPA-free, acrylic cup. Push-on lid with thumb-slide closure; for hot and cold beverages. Holds 16 oz. Hand wash only. Imprint. Clear.	6	87
444434	**Pebble Smart Watch**	Smart glasses and smart watches are perhaps two of the most exciting developments in recent years.	24	443
444435	**Oculus Rift**	The world of gaming has also undergone some very unique and compelling tech advances in recent years. Virtual reality, the concept of complete immersion into a digital universe through a special headset, has been the white whale of gaming and digital technology ever since Geekstakes Oculus Rift GiveawayNintendo marketed its Virtual Boy gaming system in 1995.Lytro	106	600
444437	**Lytro Camera**	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.	44.3	230

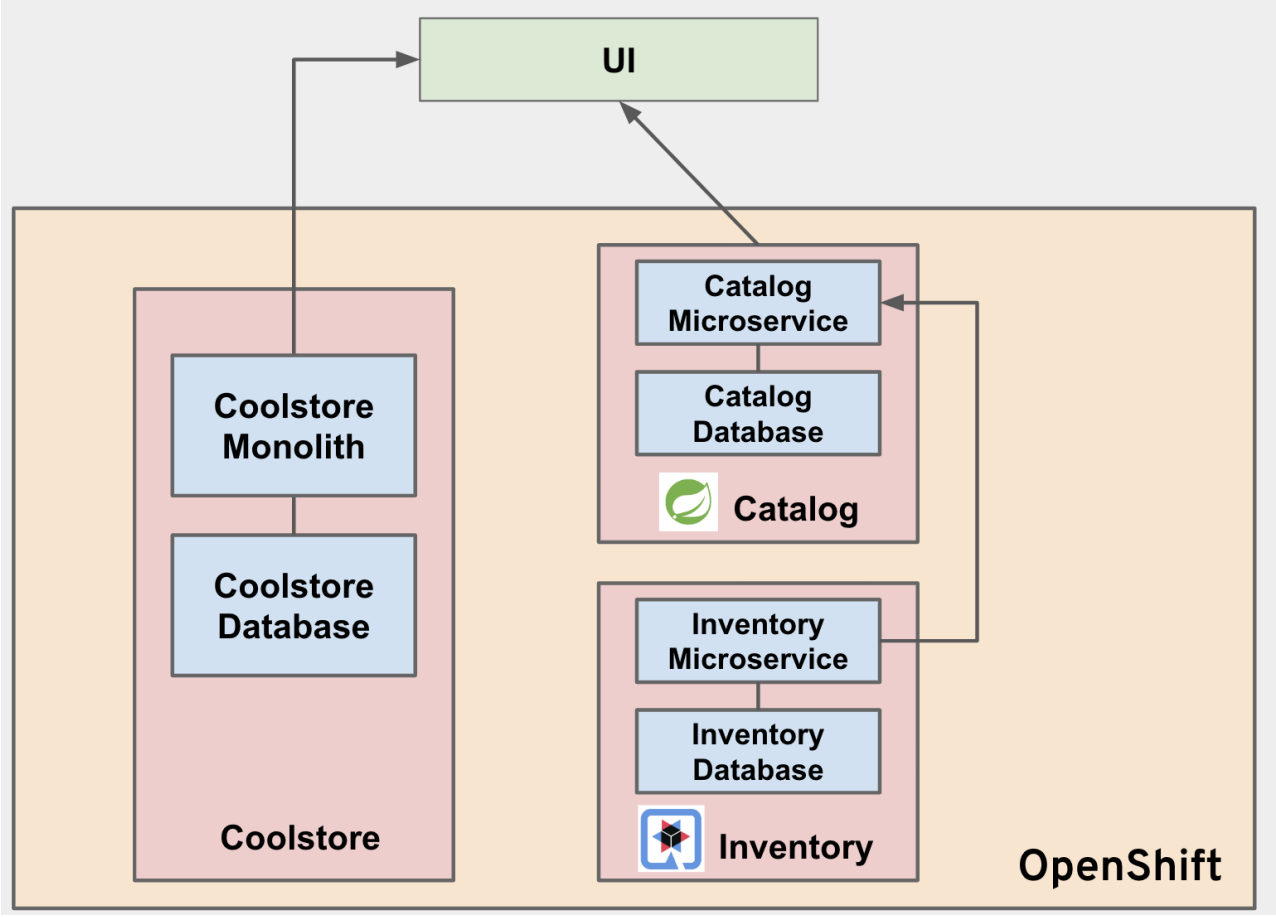
> NOTE: Since we previously have a inventory service running you should now see the actual quantity value and not the fallback value of -1.

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

19. Strangling the monolith

So far we haven't started strangling the monolith. To do this we are going to make use of routing capabilities in OpenShift. 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 *http:///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:

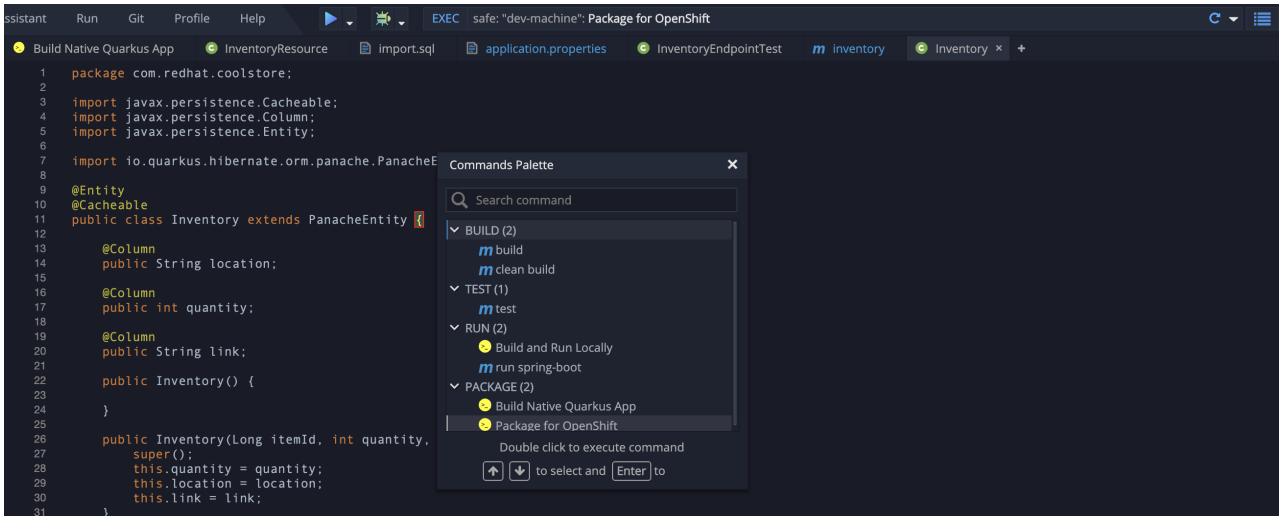


Flow the steps below to create **Cross-origin resource sharing (CORS)** based route. CORS is a mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served.

Create **CORSProvider** class in *src/main/java/com/redhat/coolstore* of **inventory** project to allow restricted resources on a *catalog* service. Copy the following all codes in the class:

```
package com.redhat.coolstore;

import org.jboss.resteasy.plugins.interceptors.CorsFilter;

import javax.ws.rs.core.Feature;
import javax.ws.rs.core.FeatureContext;
import javax.ws.rs.ext.Provider;

@Provider
public class CORSProvider implements Feature {
   @Override
   public boolean configure(FeatureContext context) {
      CorsFilter filter = new CorsFilter();
      filter.getAllowedOrigins().add("*");
      filter.setAllowedMethods("GET, POST, DELETE, OPTIONS, HEAD");
      filter.setAllowedHeaders("accept, content-type, origin");
      context.register(filter);
      return true;
   }
}
```

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



Restart and watch the build, which will take about a minute to complete. Replace your username with **userXX**:

```
cd /projects/cloud-native-workshop-v2m1-labs/inventory/
```

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

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. Add *@CrossOrigin* annotation on *CatalogEndpoint* class:

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

Repackage the project using the following command, which will use the maven plugin to deploy via CodeReady Workspaces Terminal:

```
cd /projects/cloud-native-workshop-v2m1-labs/catalog/
```

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

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

Restart and watch the build, which will take about a minute to complete. Replace your username with **userXX**:

```
cd /projects/cloud-native-workshop-v2m1-labs/catalog/
```

```
oc start-build catalog-springboot --from-file=target/catalog-1.0.0-SNAPSHOT.jar --follow -n userXX-catalog
```

Once the build is done, the catalog pod will be deployed automatically via DeploymentConfig Trigger in OpenShift.

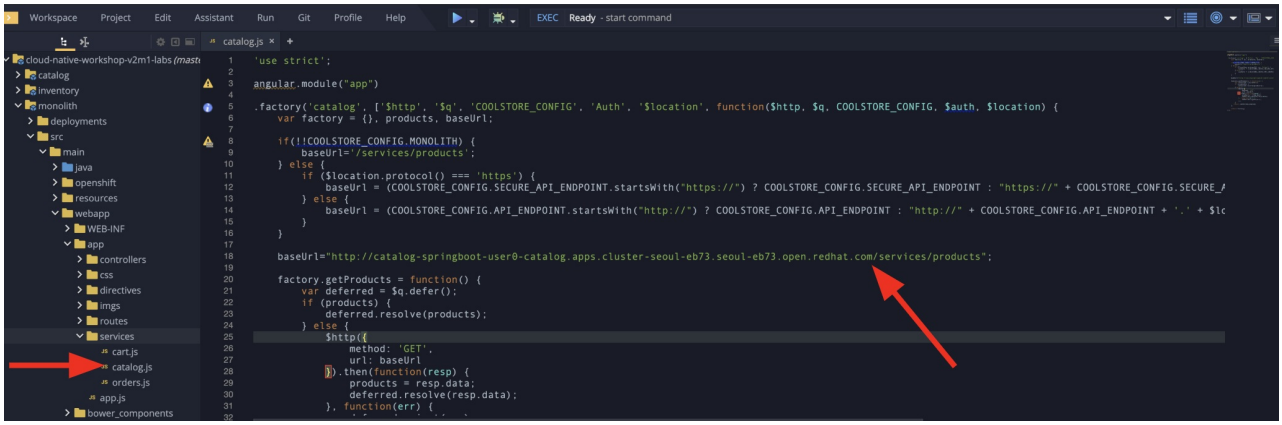
Let's update the catalog endpoint in monolith application. Copy the route URL of catalog service using following **oc** command in CodeReady Workspaces Terminal. Replace your username with **userXX**:

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

In the **monolith** project, open `catalog.js` in `src/main/webapp/app/services` and add a line as shown in the image to define the value of `baseUrl`.

```
baseUrl="YOUR_CATALOG_ROUTE_URL/services/products";
```

> Replace `YOUR_CATALOG_ROUTE_URL` with the URL emitted from the previous `echo` command



Rebuild the project in CodeReady Workspaces Terminal:

```
cd /projects/cloud-native-workshop-v2m1-labs/monolith/
```

```
mvn clean package -Popenshift
```

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

Restart and watch the build, which will take about a minute to complete. Replace your username with **userXX**:

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

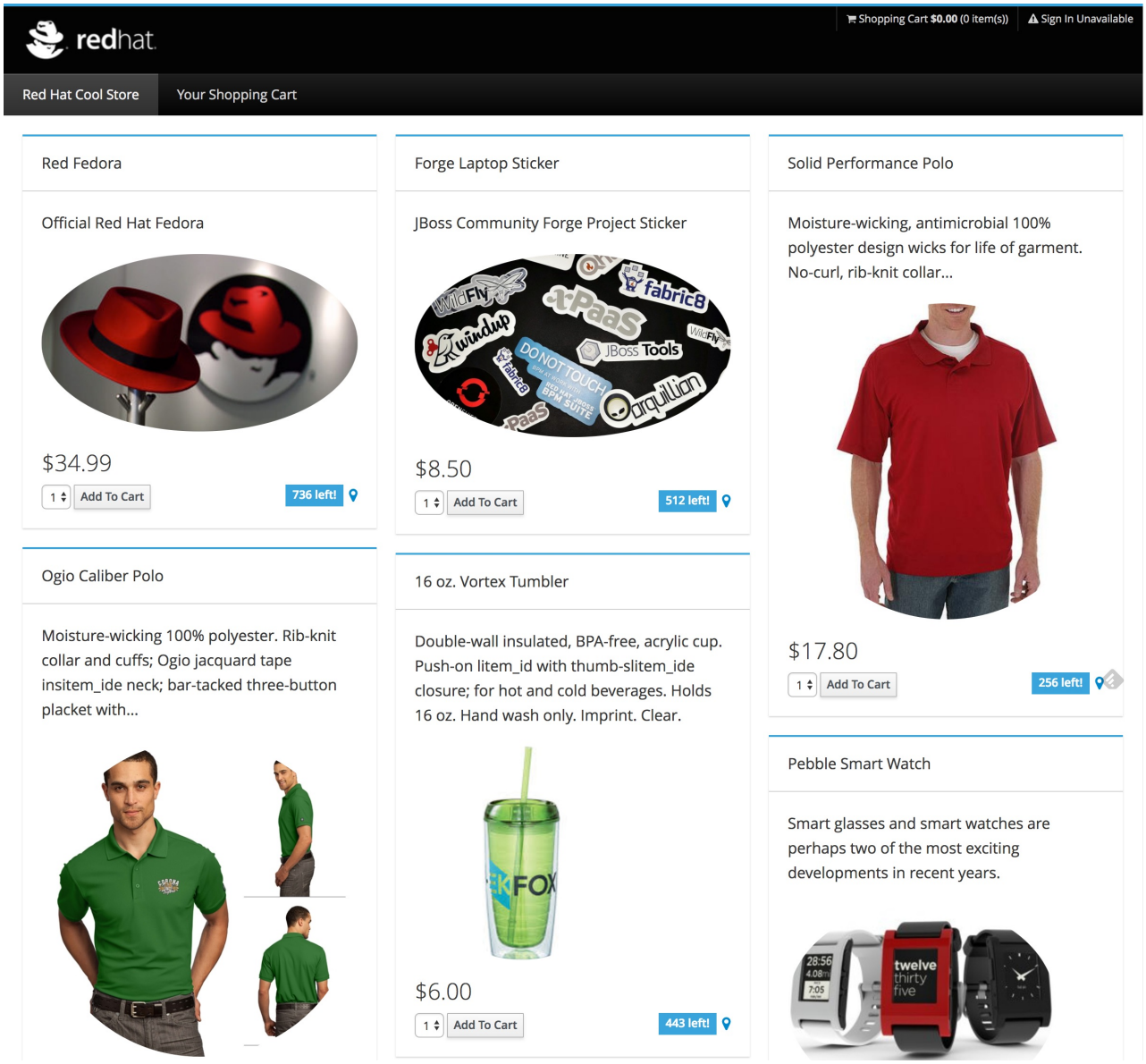
Once the build is done, the coolstore pod will be deployed automatically via DeploymentConfig Trigger in OpenShift. Ensure it's rolled out:

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

20. Test the UI

Open the monolith UI at by selecting the `userXX-coolstore-dev` project in the OpenShift web console, navigate to *Networking* > *Routes* and click on the link to the monolith UI.

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



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

> Note: 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 what Spring Boot and how it can be used together with OpenShift and OpenShift Kubernetes.

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.