The Containers and Cloud-Native Roadshow Dev Track

guides-m4-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!

Cloud-Native Application Architectures

Cloud Native Application Architectures

Developing applications that are reactive, imperative, event driven, and polyglot are all requirements of modern application architecture. While cloud infrastructure and container Kubernetes solutions, such as Red Hat OpenStack Platform and Red Hat OpenShift, provide a robust infrastructure foundation for distributed environments, similar seamless application services require building applications that take full advantage of such an infrastructure.

Moreover, applications require hybrid and multi cloud architecture to thrive in the digital economy. A unified cloud native application environment has become essential to developers because it enables higher productivity and innovation with a full, cohesive development platform. An application environment is equally critical to operations because of the rapid change and high demands for scalability, agility, and reliability.

In this module, we will learn what cloud-native architecture we need to design for running containerized applications on DevOps/Cloud Native platform in scale and speed. Then we will also develop cloud native applications based on architecture patterns such as high-performing cache, event-driven/reactive, and serverless using Red Hat Runtimes, Red Hat CodeReady Workspaces and Red Hat OpenShift Container Platform.

Capabilities of a Cloud Native Application Architectures?

The benefits of cloud native application architectures enable speed of development and deployment, flexibility, quality, and reliability. More importantly, it allows developers to integrate the applications with the latest open source technologies without a steep learning curve. While there are many ways to build and architect cloud native applications following are some great ingredients for consideration:

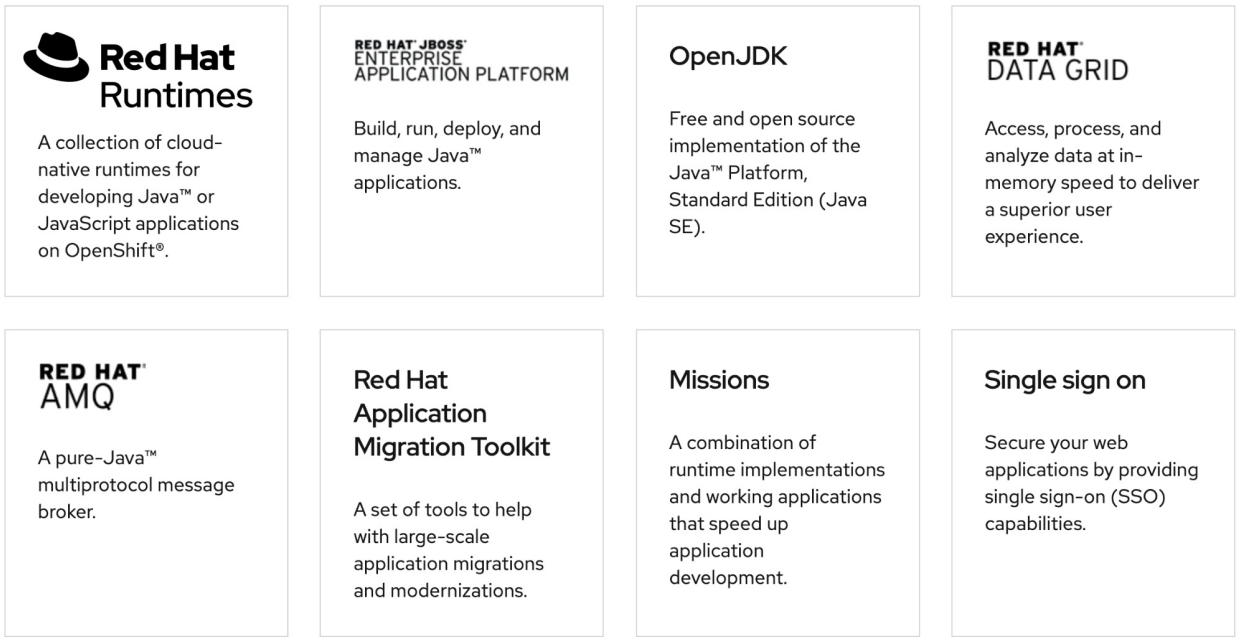
- *Runtimes* - More likely to be written in the container first or/and Kubernetes native language, which means runtimes such as Java, Node.js, Go, Python, and Ruby, etc.
- *Security* - Deploying and maintaining applications in a multi cloud, hybrid cloud application environment, security becomes of utmost importance and should be part of the environment.
- *Observability* - The ability to observer applications and their behavior in the cloud. Tools that can give realtime metrics, and more information about the use e.g., Prometheus, Grafana, Kiali, etc.

- *Efficiency* - Focused on tiny memory footprint, small artifact size, and fast booting time to make portable applications across hybrid/multi-cloud platforms. Primarily, it will leverage an expected spice in production via rapid scaling with consuming minimal computing resources.
- *Interoperability* - Easy to integrate cloud native apps with the latest open source technologies such as Infinispan, MicroProfile, Hibernate, Apache Kafka, Jaeger, Prometheus, and more for building standard runtimes architecture.
- *DevOps/DevSecOps* - Designed for continuous deployment to production in line with the minimum viable product (MVP), with security as part of the tooling together with development, automating testing, and collaboration.

How to build Cloud Native applications and architecture with Red Hat?

`Red Hat Runtimes` is a recommended set of products, tools, and components to develop and maintain cloud-native applications. It provides lightweight runtimes and frameworks for highly-distributed cloud environments such as microservices, with in-memory caching for fast data access, and messaging for quick data transfer supporting existing applications.

Red Hat Runtimes products and components:



- *Red Hat® JBoss® Enterprise Application Platform 7 (JBoss EAP)* is the market-leading opensource platform for modern Java applications deployed in any environment. JBoss EAP's architecture is lightweight, modular, and cloud ready. Based on the opensource WildFly app server project, the platform offers powerful management and automation for higher developer productivity.

- *A set of cloud-native runtimes* are Spring Boot with Tomcat, Reactive Vert.x, Javascript Node.Js, MicroProfile Throntail.(Quarkus is coming soon!)
- *Red Hat OpenJDK* is an opensource implementation of the Java Platform SE (Standard Edition) supported and maintained by the OpenJDK community. OpenJDK is the default Java development and runtime in Red Hat Enterprise Linux.
- *Red Hat Data Grid* is an opensource in-memory distributed data management system designed for scalability and fast access to large volumes of data. More than just a distributed caching solution, it also offers additional functionality such as map/reduce, querying, processing for streaming data, and transaction capabilities.
- *Red Hat AMQ (Broker)* is a pure-Java multiprotocol message broker that offers specialized queueing behaviors, message persistence, and manageability.
- *Red Hat Application Migration Toolkit* provides a set of utilities for easing the process of taking customers' proprietary or outdated middleware platforms to state-of-the-art lightweight, modular, and cloud-ready middleware application infrastructure is making teams more productive and ready for the future.
- *Missions and Boosters* are a combination of runtime implementations and working applications that accelerate application development. Missions are working applications that showcase different fundamental pieces of building cloudnative applications and services. A booster is the implementation of a mission in a specific runtime.
- *Red Hat Single Sign-On* based on the Keycloak project, Red Hat sso enables customers to secure web applications byproviding Web single sign-on) capabilities based on popular standards such as SAML 2.0, OpenID Connect and OAuth 2.0.The RH-sso server can act as a SAML or OpenID Connect-based identity provider, mediating your enterprise user directoryor 3rd-party SSO provider for identity information with your applications via standards-based tokens.

Red Hat Runtimes also provide integrated and optimized products and components to deliver modern applications, whether the goal is to keep existing applications or create new ones. Applications Runtimes enable developers to containerize applications with a microservices architecture, improve data access speed via in-memory data caching, enhance application performance with messaging, or adapt cloud-native application development using modern development patterns and technologies.

Additionally, we have also chosen to use Quarkus for most of the applications in the labs. Read on to learn more about Quarkus.

> NOTE: At the time of writing this guide, Quarkus is still a community project and is not part of any of the Red Hat Middleware products.

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.

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

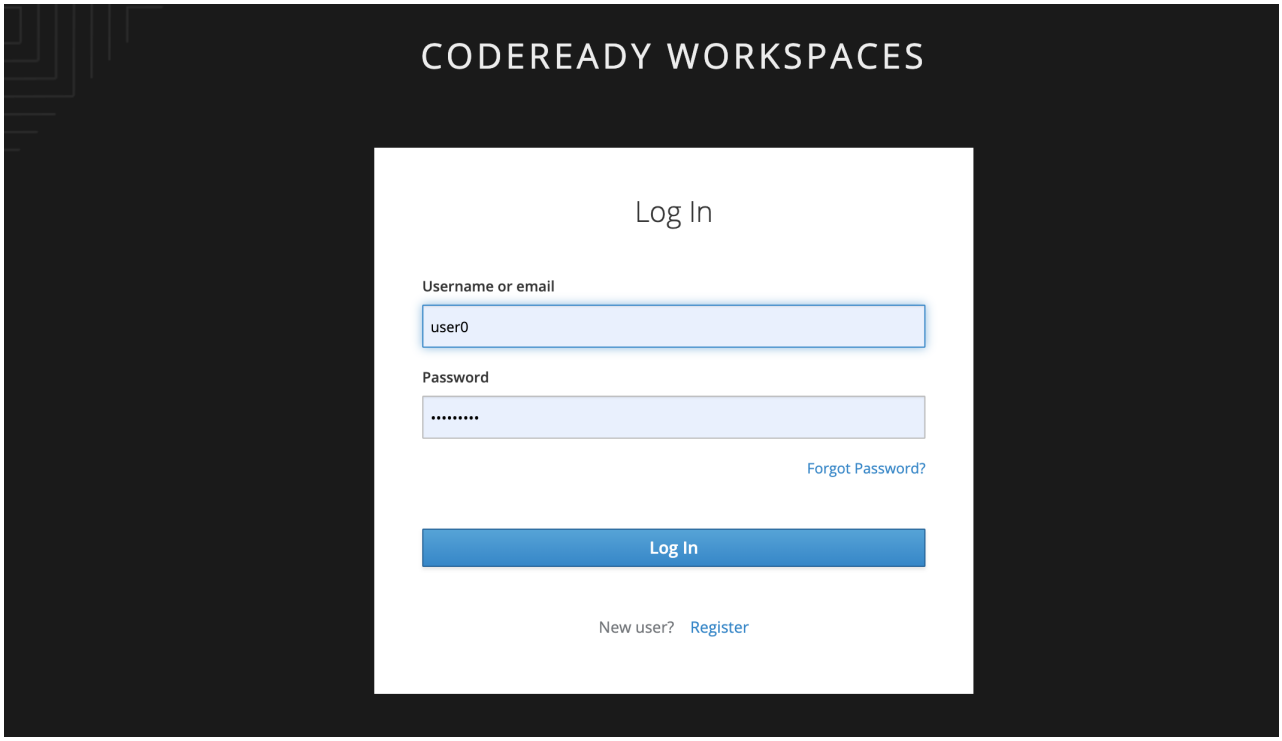
- *Developer Joy* - Cohesive platform for optimized developer joy through unified configuration, Zero config with live reload in the blink of an eye, streamlined code for the 80% common usages with flexible for the 20%, and no hassle native executable generation.
- *Unifies Imperative and Reactive* -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

Getting Ready for the labs

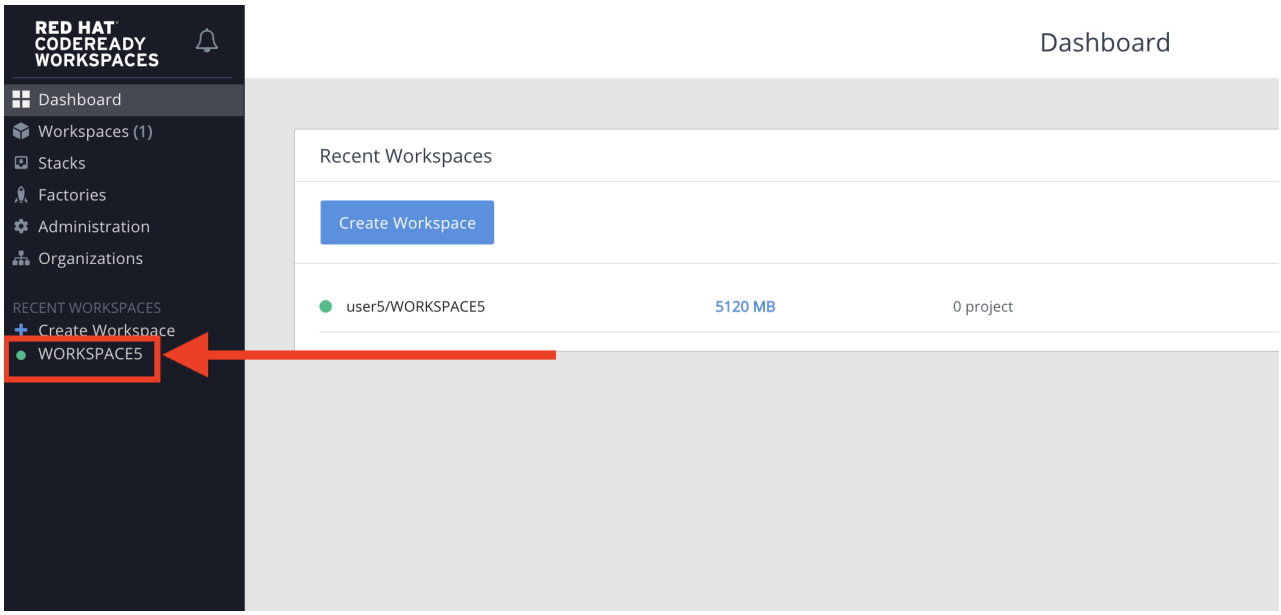
If this is the first module you are doing today

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

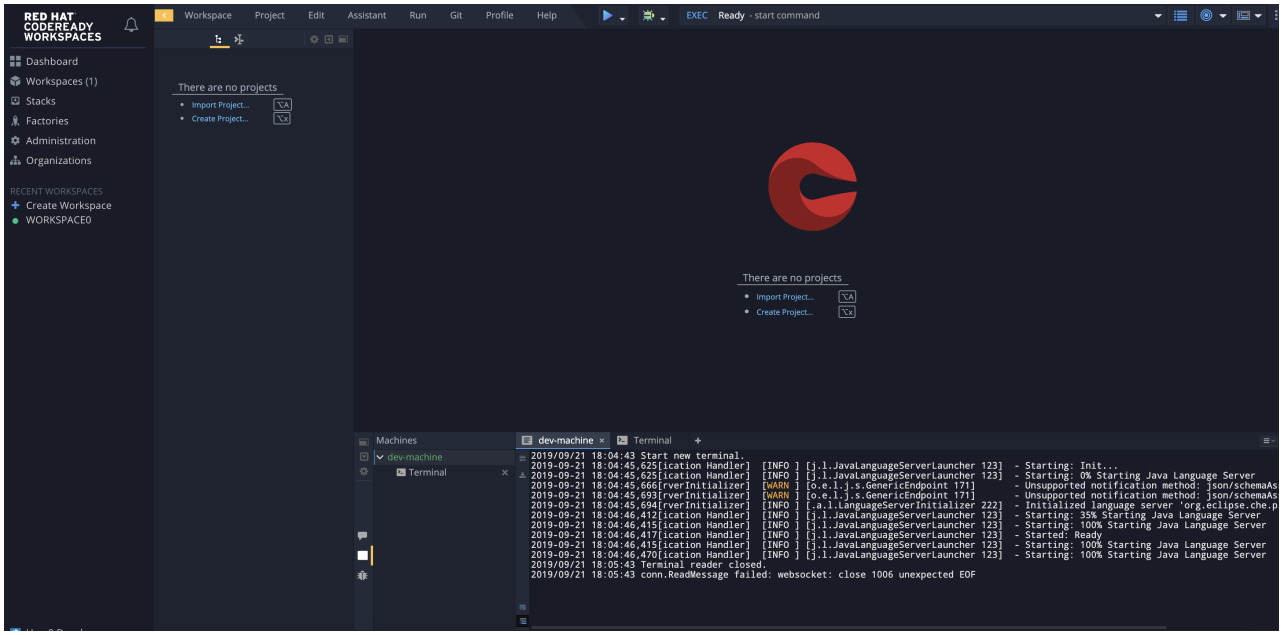
To get started, access the Che instance and log in using the username and password you've been assigned (e.g. `userXX/r3dh4t1!`):



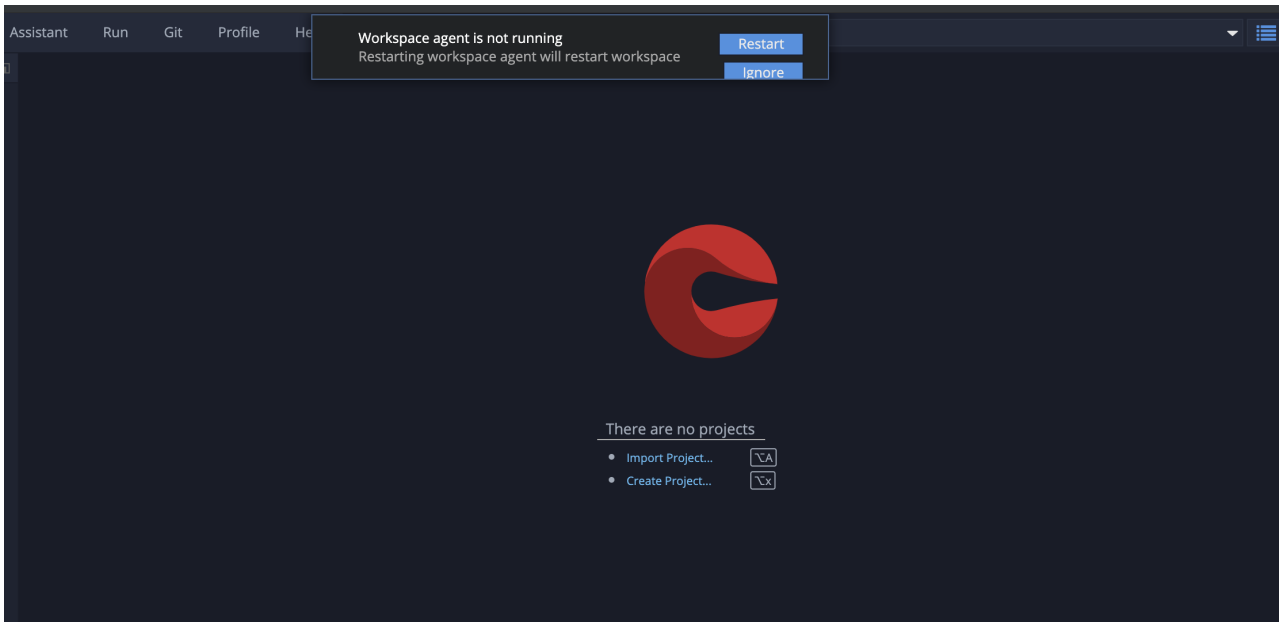
Once you log in, you'll be placed on your personal dashboard. We've pre-created workspaces for you to use. Click on the name of the pre-created workspace on the left, as shown below (the name will be different depending on your assigned number). You can also click on the name of the workspace in the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



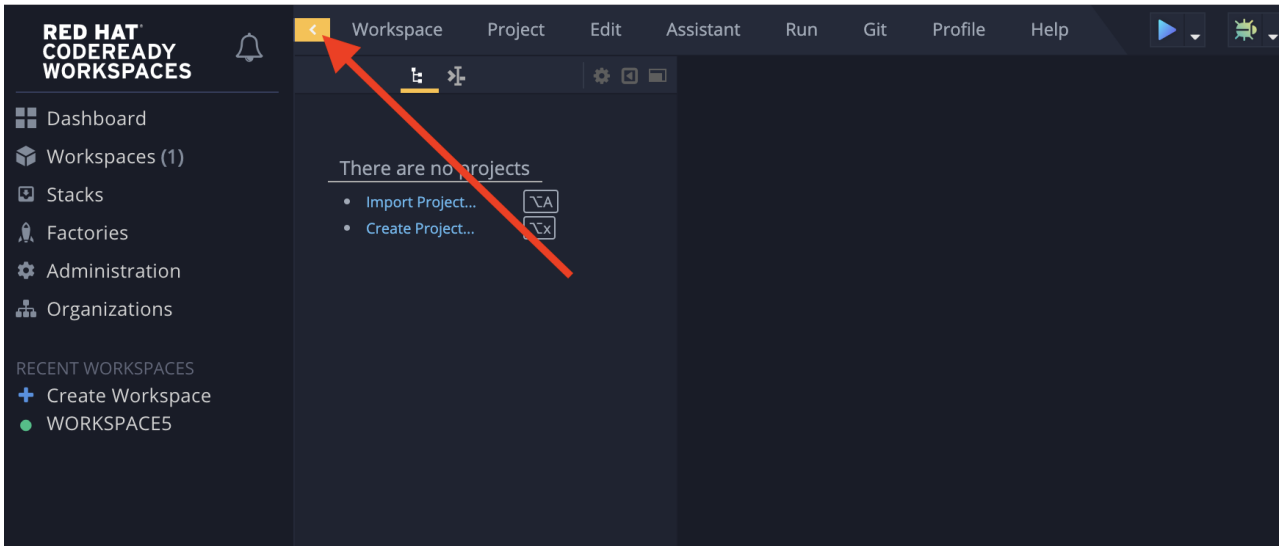
After a minute or two, you'll be placed in the workspace:



You might see **Workspace agent is not running** in the popup message, click on `Restart` :



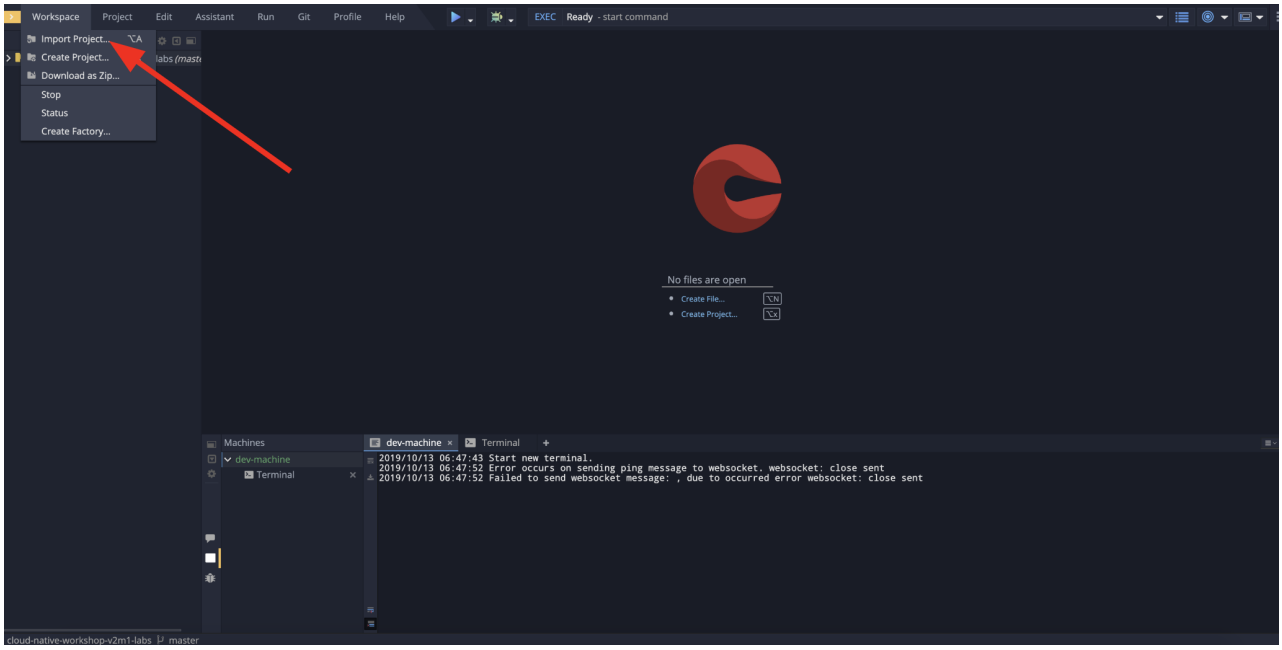
To gain extra screen space, click on the yellow arrow to hide the left menu (you won't need it):



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

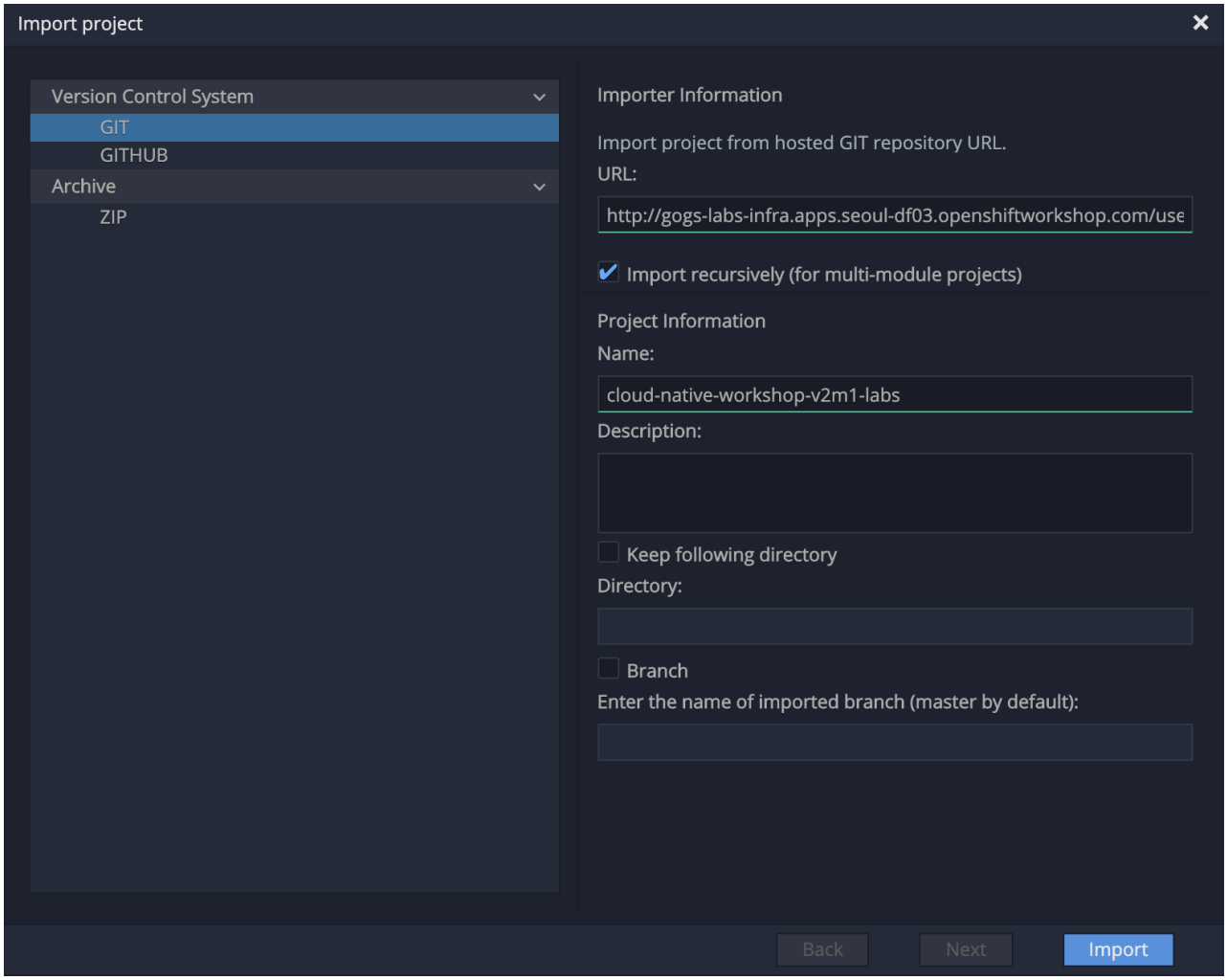
Import Projects

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



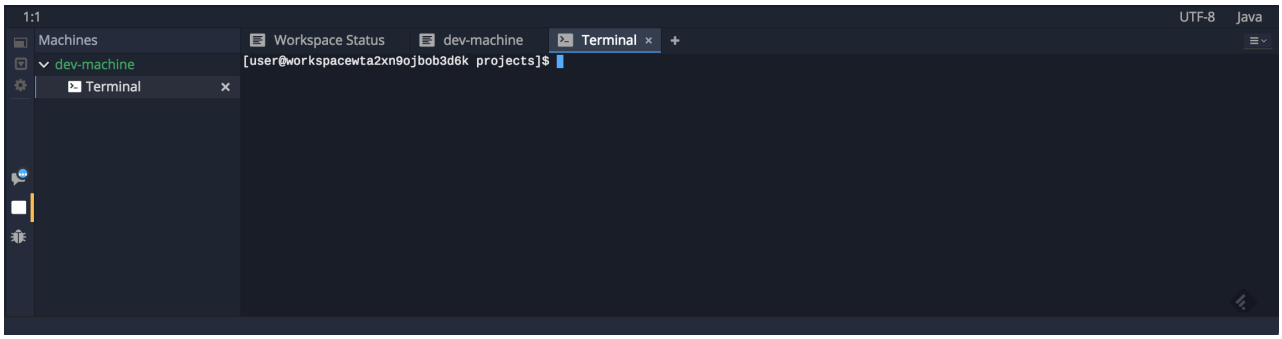
- Version Control System: `GIT`
- URL: `http://gogs-labs-infra.apps.cluster-beijing-5bc2.beijing-5bc2.open.redhat.com/userXX/cloud-native-workshop-v2m4-labs.git` (IMPORTANT: replace userXX with your lab user)
- Check `Import recursively (for multi-module projects)`
- Name: `cloud-native-workshop-v2m4-labs`

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



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

> 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.



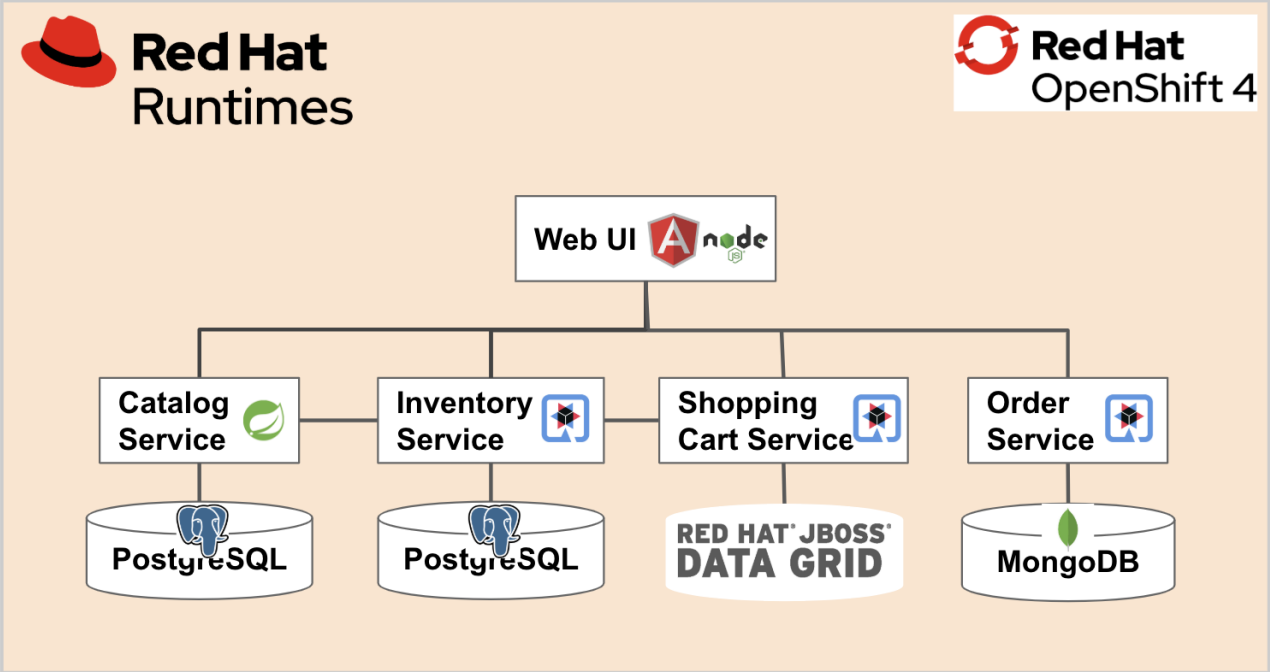
Creating High-performing Cacheable Service

Lab1 - Creating High-performing Cacheable Service

In this lab, we'll develop 5 microservices into the cloud-native appliation architecture. These cloud-native applications will have transactions with multiple datasources such as `PostgreSQL` and `MongoDB`. Especially, we will learn how to configure datasources easily using `Quarkus Extensions`. In the end, we will optimize `data transaction performance` of the shopping cart service thru integrating with a `Cache(Data Grid) server` to increase end users'(customers) satification. And there's more fun facts how easy it is to deploy applications on OpenShift 4 via `oc` command line tool.

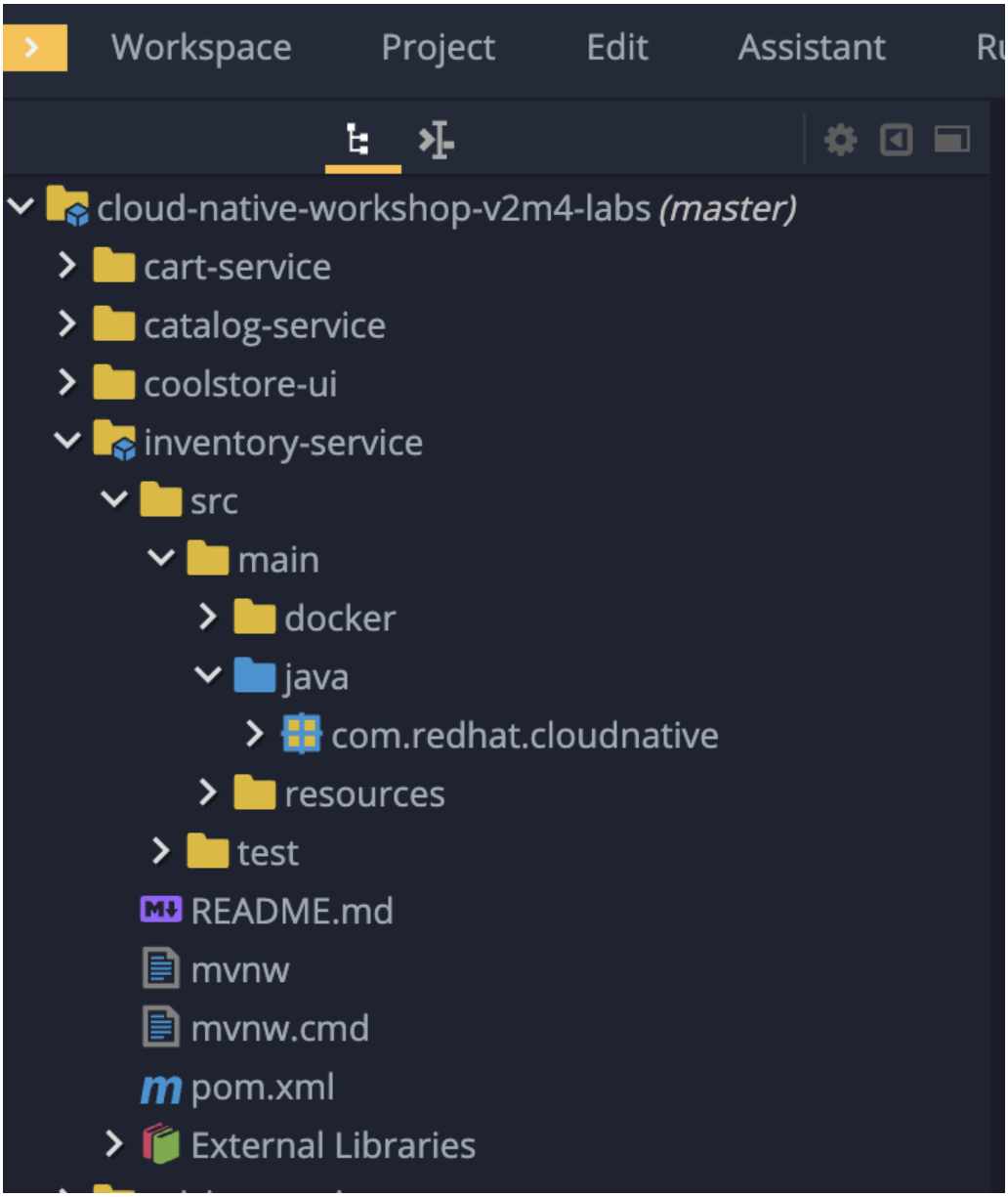
Goals of this lab

The goal is to develop advanced cloud-native applications on `Red Hat Runtimes` and deploy them on `OpenShift 4` including `single sign-on access management` and `distributed cache manageemnt`. After this lab, you should end up with something like:



1. Deploying Inventory Service

`Inventory Service` serves inventory and availability data for retail products. Lets's go through quickly how the inventory service works and built on `Quarkus` Java runtimes. Go to `Project Explorer` in `CodeReady Workspaces` Web IDE and expand `inventory-service` directory.



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

- `RESTEasy` to expose the REST endpoints
- `Hibernate ORM` with Panache to perform the CRUD operations on the database
- `Maven` Java project structure

`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.

When you open `Inventory.java` in `src/main/java/com/redhat/cloudnative/` as below, you will understand how easy to create a domain model using Quarkus extension(Hibernate ORM with Panache).

```
@Entity
@Cacheable
public class Inventory extends PanacheEntity {

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

    public Inventory() {

    }

}
```

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

- By using Use public fields, there is no need for functionless getters and setters (those that simply get or set the field). You simply refer to fields like Inventory.location without the need to write a Inventory.geLocation() 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.

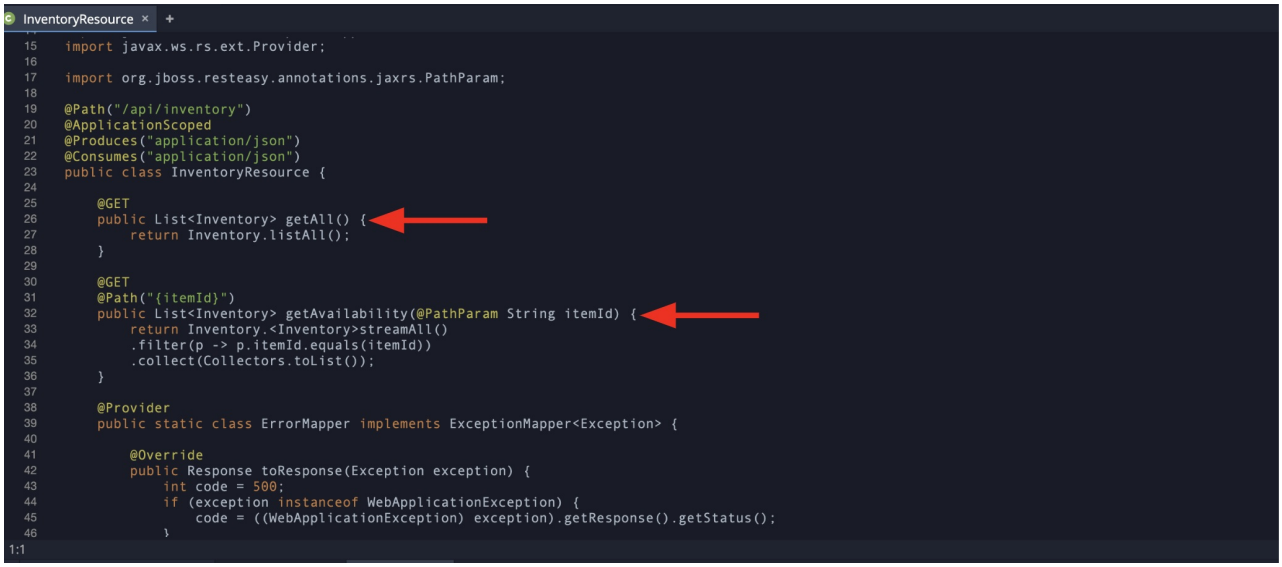
The `PanacheEntity` superclass comes with lots of super useful static methods and you can add your own in your derived entity class, and much like traditional object-oriented programming it's natural and recommended to place custom queries as close to the entity as possible, ideally within the entity definition itself. Users can just start using your entity Inventory by typing Inventory, and getting 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 without querying the database, but be careful as it implies the loaded entity might not reflect recent changes in the database.

Next, let's find out how `inventory service` exposes `RESTful APIs` on Quarkus. Open `InventoryResource.java` in `src/main/java/com/redhat/cloudnative/` and you will see the following code sniffet.

The REST services defines two endpoints:

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



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

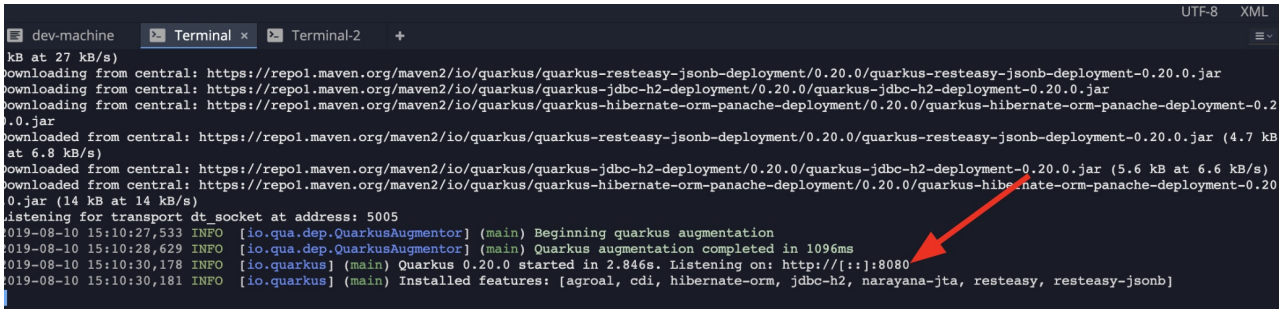
```
quarkus.datasource.url=jdbc:h2:file://projects/database.db
quarkus.datasource.driver=org.h2.Driver
quarkus.datasource.username=inventory
quarkus.datasource.password=mysecretpassword
quarkus.datasource.max-size=8
quarkus.datasource.min-size=2
quarkus.hibernate-orm.database.generation=drop-and-create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

Let's run the inventory application locally using maven plugin command via CodeReady Workspaces Terminal:

```
cd /projects/cloud-native-workshop-v2m4-labs/inventory-service/
```

```
mvn compile quarkus:dev
```

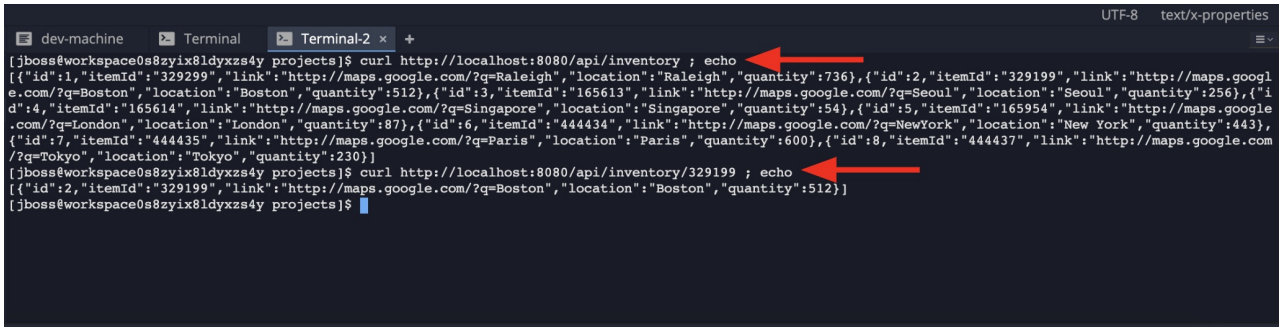
You should see a bunch of log output that ends with:



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/api/inventory ; echo
```

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



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

In production, the inventory service will connect to PostgeSQL on OpenShift cluster.

We will use Quarkus extension to add PostgreSQL JDBC Driver. Go back to CodeReady Workspaces Terminal and run the following maven plugin:

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

Package the applicaiton via running the following maven plugin in CodeReady WorkspacesTerminal:

```
mvn clean package -DskipTests
```

> NOTE : You should SKIP the Unit test because you don't have PostgreSQL database in local environment.

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
```

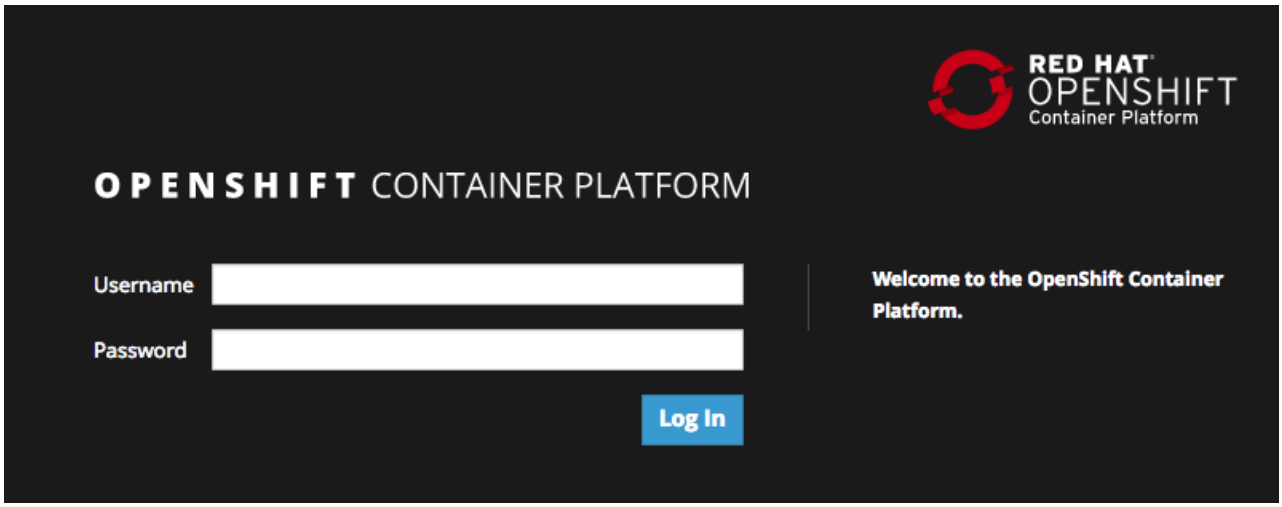
Enter your username and password assigned to you:

- Username: userXX
- Password: r3dh4t1!

You should see like:

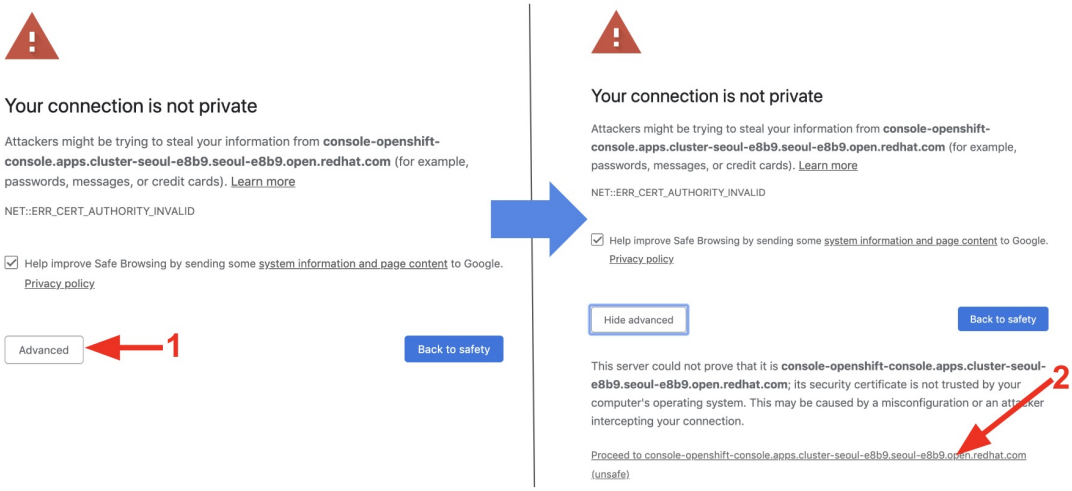
```
Login successful.

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.
```

First, open a new brower with the OpenShift web console

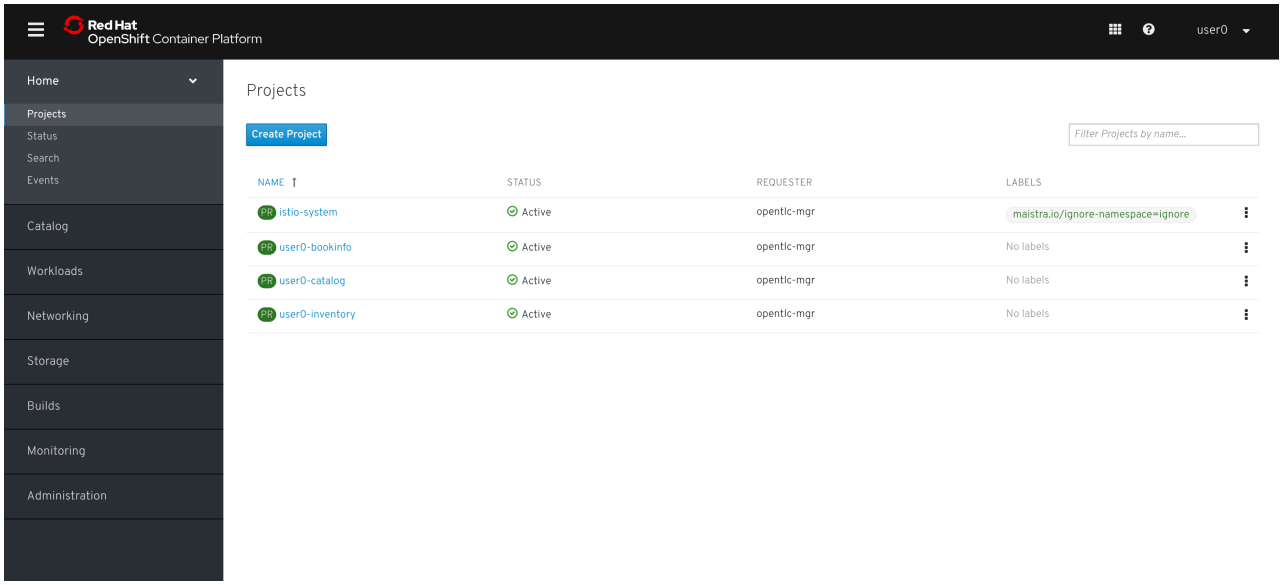


Login using:

- 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.

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:

Create a new project in OpenShift Cluster. You need to replace `userXX` with your username:

```
oc project userXX-cloudnativeapps
```

Deploy PostgreSQL to the project:

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

Build the image using on OpenShift:

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

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

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

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

Deploy it as an OpenShift application after the build is done and override the Postgres URL to specify our production Postgres credentials:

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

Create the route

```
oc expose svc/inventory
```

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

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

Wait for that command to report replication controller `inventory-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 all inventories:

Get the route URL

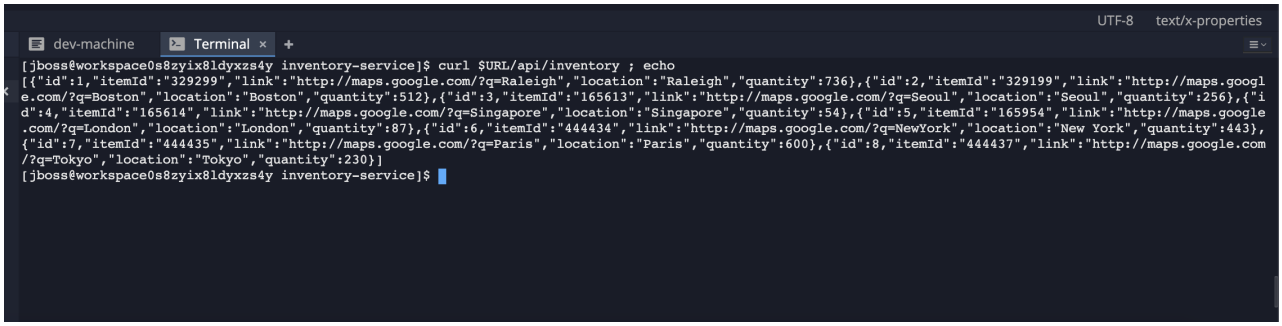
```
export URL="http://$(oc get route | grep inventory | awk '{print $2}')"
```

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

You will see the following result:

> **NOTE** It may take a few tries to get the below result as the pod spins up. Keep trying until you get the below output!

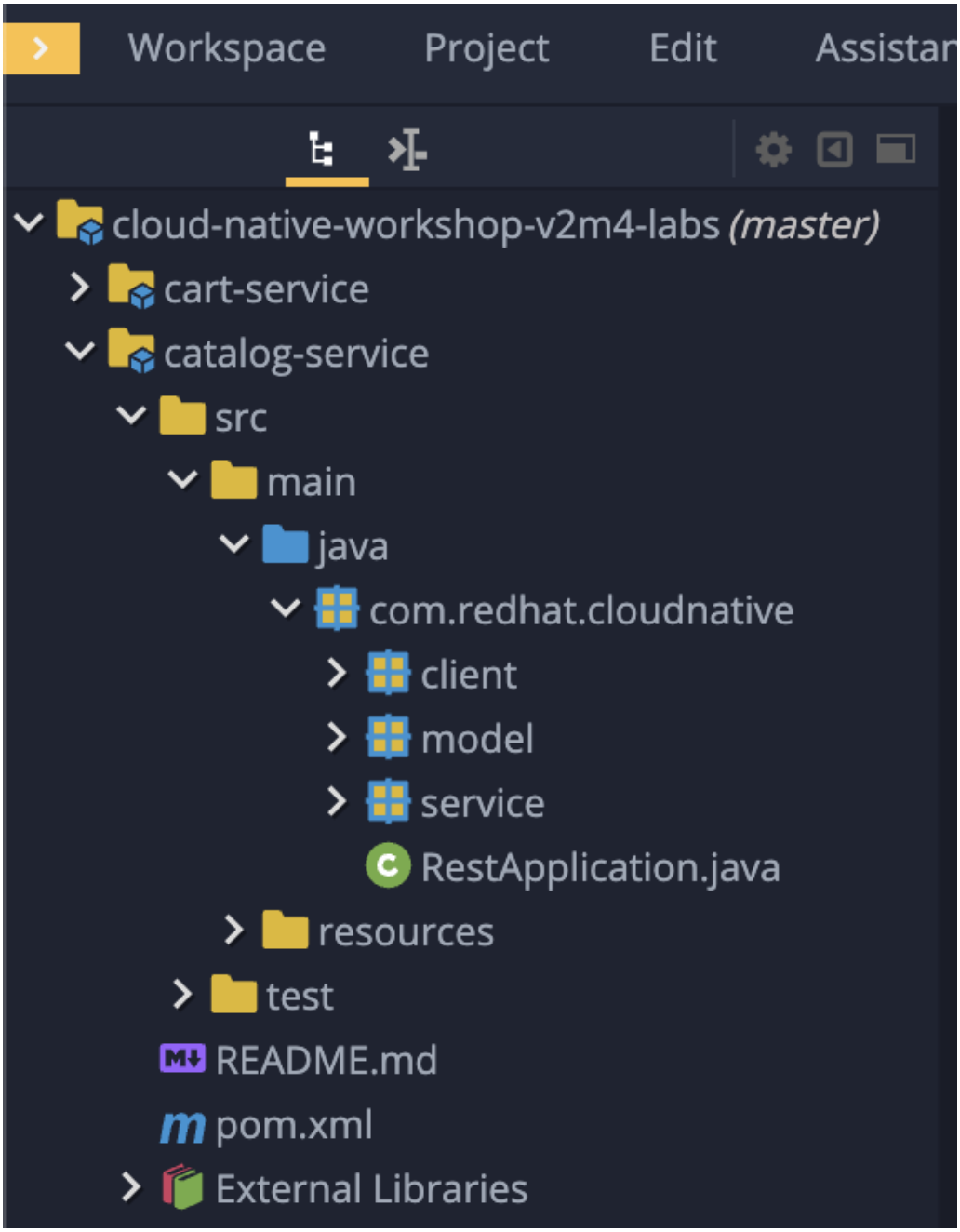

```
[{"id":1,"itemId":"329299","link":"http://maps.google.com/?
q=Raleigh","location":"Raleigh","quantity":736},
{"id":2,"itemId":"329199","link":"http://maps.google.com/?q=Bost
on","location":"Boston","quantity":512},{"id":3,"itemId":"165613","link":"http://maps.google.com/?
q=Seoul","location":"Seoul","quantity":256},{"id":4,"itemId":"165614","li
nk":"http://maps.google.com/?q=Singapore","location":"Singapore","quantity":54},
{"id":5,"itemId":"165954","link":"http://maps.google.com/?q=London","location":"London","qu
antity":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}]
```

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.

2. Deploying Catalog Service

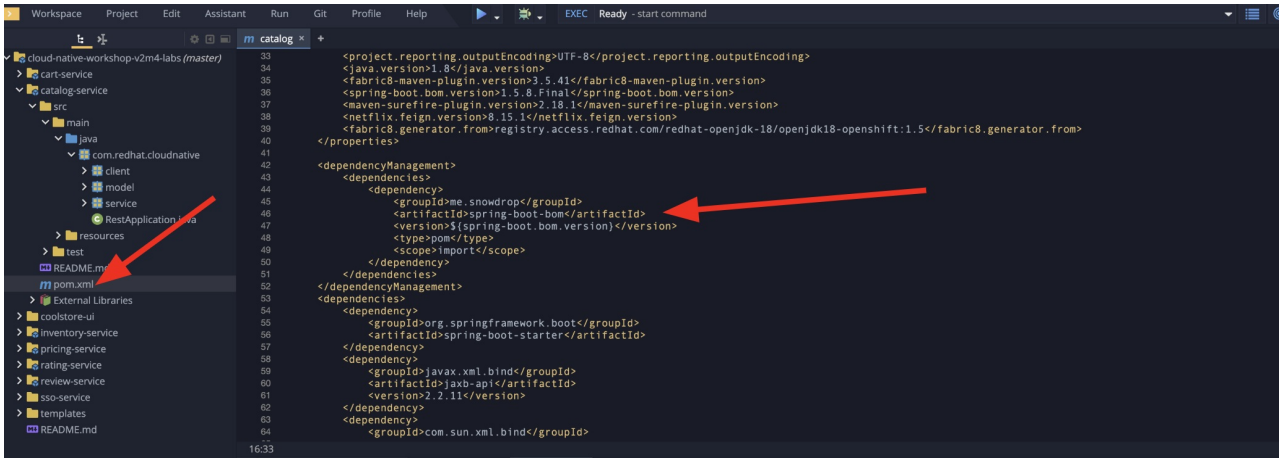
`Catalog Service` serves products and prices for retail products. Lets's go through quickly how the catalog service works and built on `Spring Boot` Java runtimes. Go to `Project Explorer` in `CodeReady Workspaces` Web IDE and expand `catalog-service` directory.



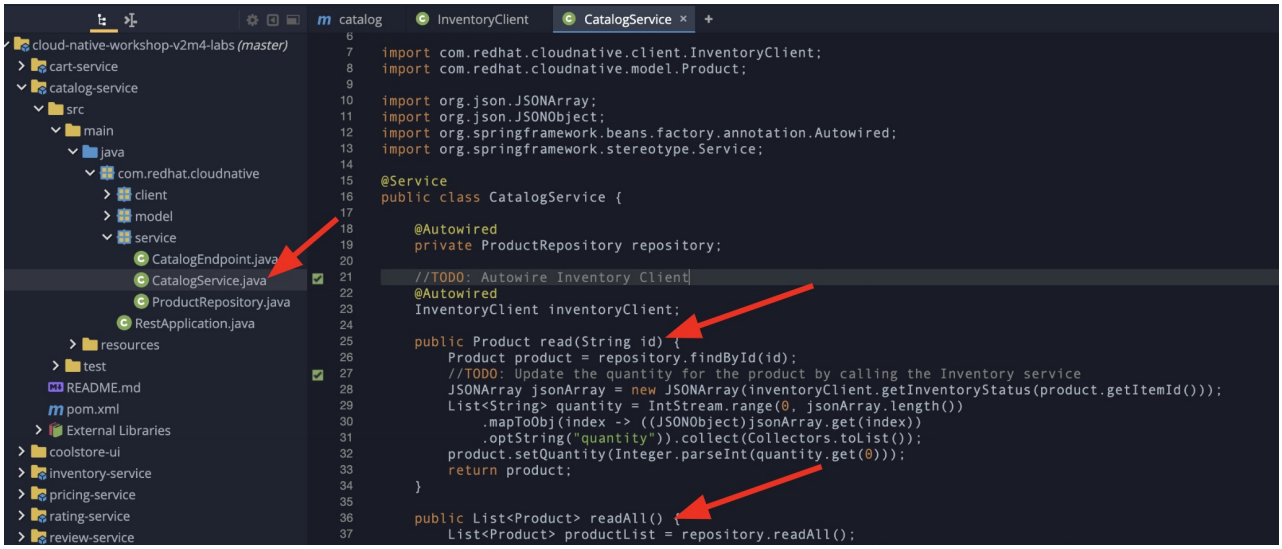
First of all, we won't implement the catalog application to retrieve data because of all funtions are already built when we imported this project from Git server. There're a few interesting things what we need to take a look at this Spring Boot application before we will deploy it to OpenShift cluster.

This catalog service is 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>
```



Also, catalog service calls the inventory service that we deployed earlier using REST to retrieve the inventory status and include that in the response. Open `CatalogService.java` in `src/main/java/com/redhat/coolstore/service` directory via Project Explorer and how `read()` and `readAll()` method work:



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-v2m4-labs/catalog-service/
```

```
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.

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:

Make sure if the current project is `userXX-cloudnativeapps` .

Deploy PostgreSQL to the project:

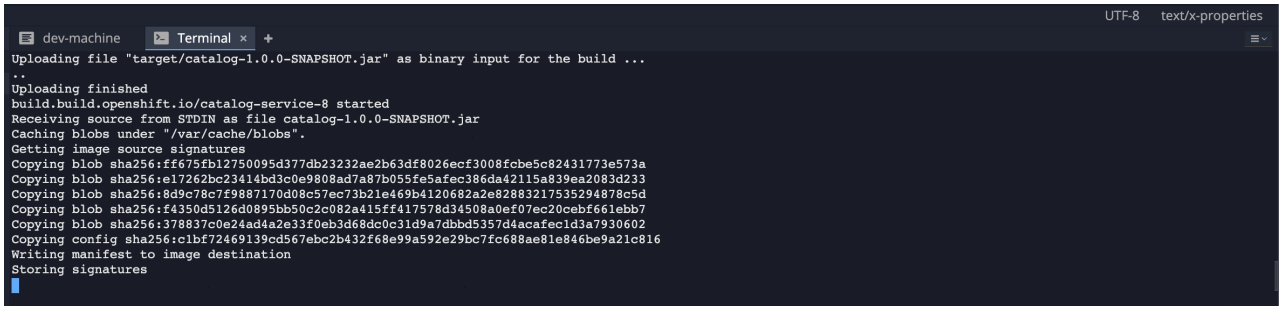
```
oc new-app -e POSTGRESQL_USER=catalog \
   -e POSTGRESQL_PASSWORD=mysecretpassword \
   -e POSTGRESQL_DATABASE=catalog \
   openshift/postgresql:10 \
   --name=catalog-database
```

Build the image using on OpenShift:

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

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

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



Deploy it as an OpenShift application after the build is done and override the Postgres URL to specify our production Postgres credentials:

`oc new-app catalog`

Create the route

`oc expose service catalog`

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

`oc rollout status -w dc/catalog`

Wait for that command to report replication controller `catalog-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:

Get the route URL

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

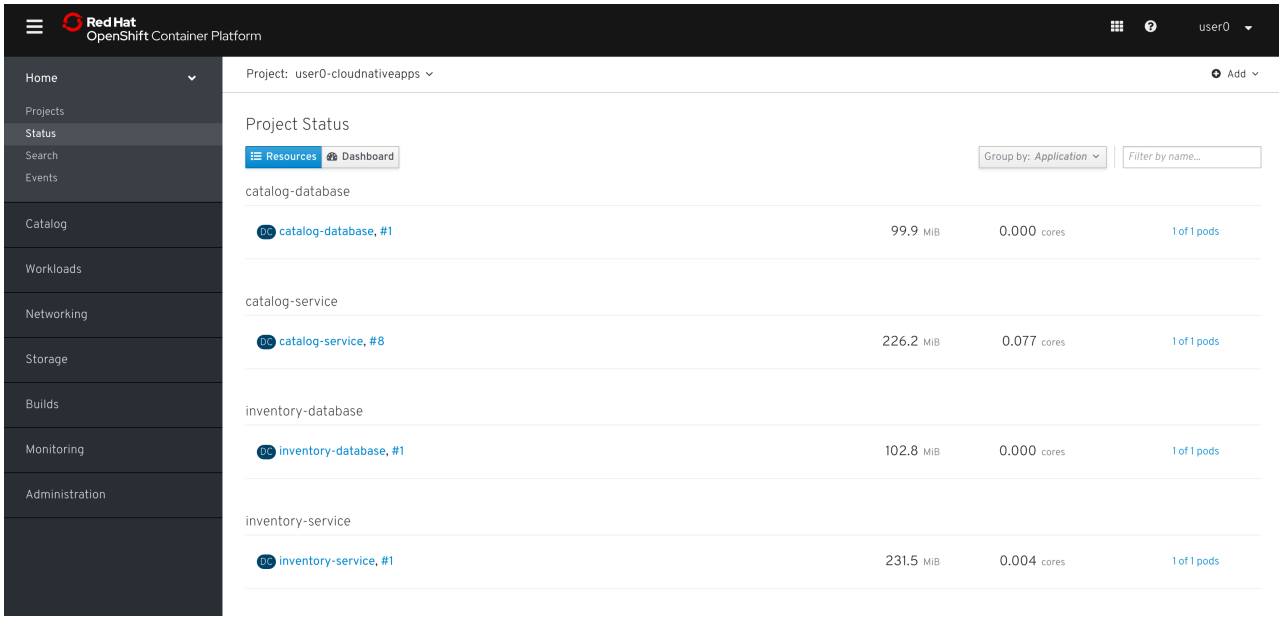
```
curl $URL/api/product/329299 ; echo
```

You will see the following result:

> **NOTE** It may take a few tries to get the below result as the pod spins up. Keep trying until you get the below output! Also, you may get `quantity:0` for the first few times as the link to the inventory service is established.

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

So now `Catalog` service is deployed to OpenShift. You can also see it in the Project Status in the OpenShift Console with running 4 pods such as catalog, catalog-database, inventory, and inventory-database.



3. Developing and Deploying Shopping Cart Service

By now, you have deployed some of the essential elements for the Coolstore application. However, an online shop without a cart means no checkout experience. In this section, we are going to implement the Shopping Cart; in our Microservice world, we are going to call it the `cart service` and our java artifact/repo is called the `cart-service`.

Let's get started!

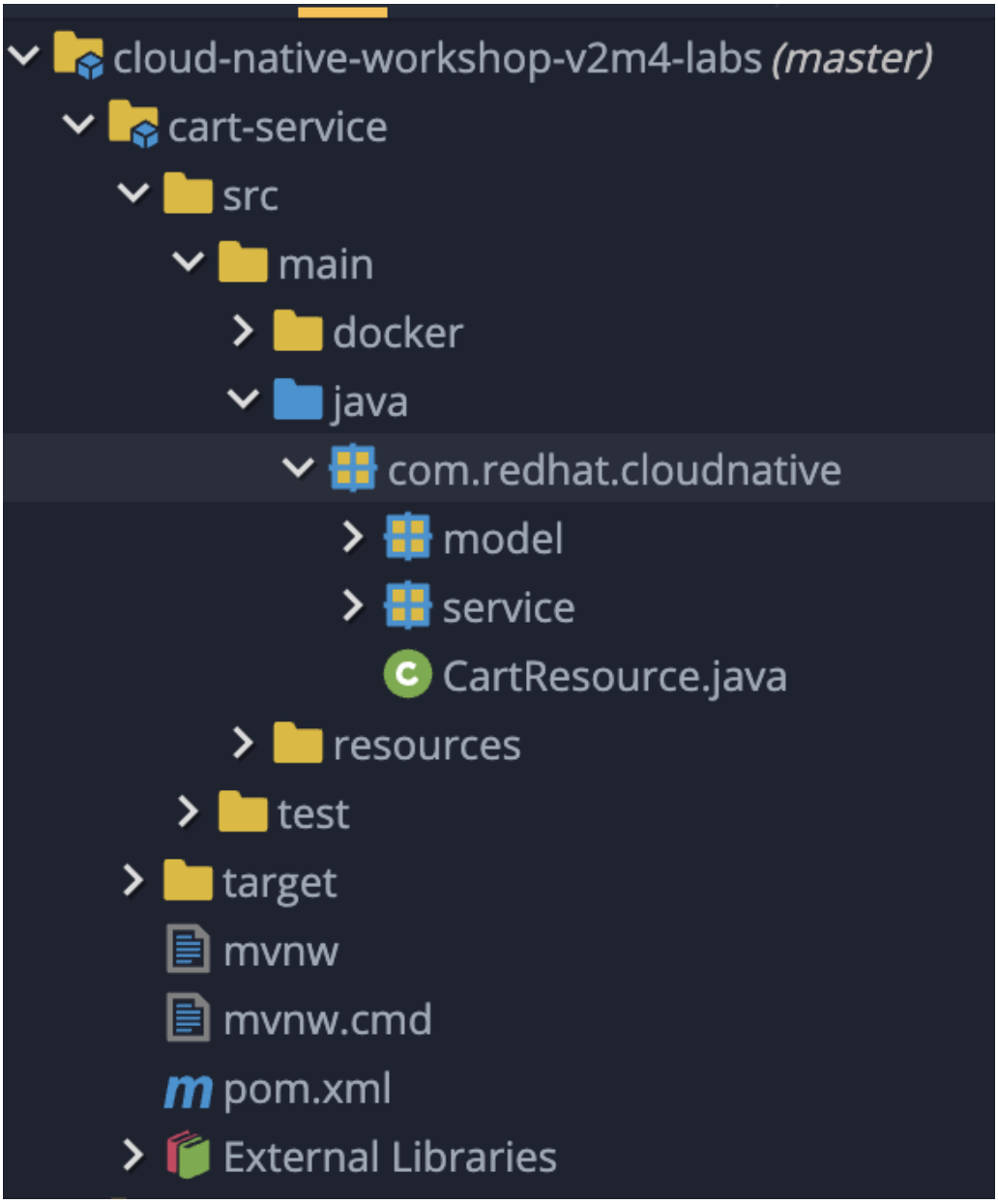
In a nutshell, the Cart service is RESTful and built with Quarkus using the Red Hat's Distributed `Data Grid` technology.

What are the building blocks of the Shopping cart a.k.a cart-service?

It uses a Red Hat's Distributed `Data Grid` technology to store all shopping carts and assigns a unique id to them. It uses the `Quarkus Infinispan client` to do this. The Shopping cart makes a call via the Quarkus Rest client to fetch all items in the Catalog. In the end, Shopping cart also throws out a `Kafka` message to the topic Orders, when checking out. For that, we use the `Quarkus Kafka client` in the next lab. Last and perhaps worth mentioning the `REST+Swagger UI` also part of the REST API support in `Quarkus`.

What is a `Shopping Cart` in our context? A Shopping cart has a list of Shopping Items. Quantity of a product in the Items list `Discount` and promotional details. We will see these in more details when we look at our model.

For this lab, we are using the code ready workspaces, make sure you have the following project open in your workspace. Lets's go through quickly how the cart service works and built on `Quarkus` Java runtimes. Go to `Project Explorer` in `CodeReady Workspaces` Web IDE and expand `cart-service` directory.



Adding a distributed cache to our cart-service

We are going to use the Red hat Distributed `Data Grid` for caching all the users' carts.

`Red Hat® Distributed Data Grid` is an in-memory, distributed, NoSQL datastore solution. Your applications can access, process, and analyze data at in-memory speed to deliver a superior user experience with features and benefits as below:

- `Act faster` - Quickly access your data through fast, low-latency data processing using memory (RAM) and distributed parallel execution.
- `Scale quickly` - Achieve linear scalability with data partitioning and distribution across cluster nodes.
- `Always available` - Gain high availability through data replication across cluster nodes.

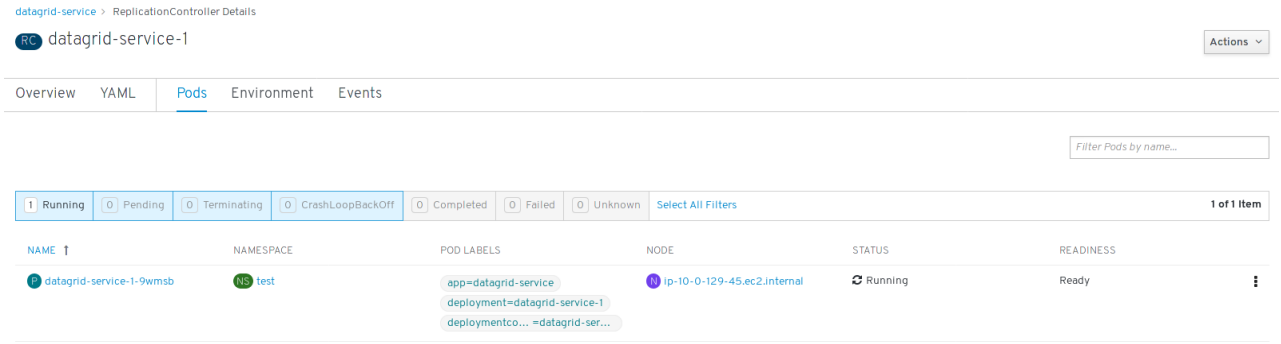
- `Fault tolerance` - Attain fault tolerance and recover from disaster through cross-data center geo-replication and clustering.
- `More productivity` - Gain development flexibly and higher productivity with a highly versatile, functionally rich NoSQL data store.
- `Protect data` - Obtain comprehensive data security with encryption and role-based access.

Lets create a simple version of the cache service in our cluster. Open the Terminal in your CodeReady workspace and run the following command:

```
oc new-app jboss/infinispan-server:10.0.0.Beta3 --name=datagrid-service
```

> `NOTE` : This will create a single instance of infinispan server the community version of the DataGrid. At the time of writing this guide, the infinspan client for Quarkus does not work with DataGrid, and Quarkus itself is also a community project.

Once deployed you should see the newly created `datagrid-service` in your project dashboard as follows:



Now that our cache service a.k.a datagrid-service is deployed. We want to ensure that everything in our cart is persisted in this blazing fast cache. It will help us when we have a few million users per second on a black Friday.

Following is what we need to do:

- Model our data
- Choose how we store the data
- Create a marshaller for our data
- Inject our cache connection into the service

We have made this choice easier for you. The default serialization is done using a library based on `protobuf` . We need to define the protobuf schema and a marshaller for each user type(s).

Let's take a look at our `cart.proto` file in `META-INF`:

```
package coolstore;

message ShoppingCart {
  required double cartItemTotal = 1;
  required double cartItemPromoSavings = 2;
  required double shippingTotal = 3;
  required double shippingPromoSavings = 4;
  required double cartTotal = 5;
  required string cartId = 6;

  repeated ShoppingCartItem shoppingCartItemList = 7;
}

message ShoppingCartItem {
  required double price = 1;
  required int32 quantity = 2;
  required double promoSavings = 3;
  required Product product = 4;
}

// TODO ADD Product
message Promotion {
  required string itemId = 1;
  required double percentOff = 2;
}
```

- So our ShoppingCart has ShoppingCartItem
- ShoppingCartItem has Product

But we havent defined the `Product` yet. Lets go ahead and do that. Add this code to the `//TODO ADD Product` marker:

```
message Product {
  required string itemId = 1;
  required string name = 2;
  required string desc = 3;
  required double price = 4;
}
```

`Great!`, now we have the Product defined in our proto model. We should also ensure that this model also exists as `POJO` (Plain Old Java Object), that way our `REST Endpoint`, or `Cache` will be able to directly serialize and desrialize the data.

Lets open up our `Product.java` in package model:

```
    private String itemId;
    private String name;
    private String desc;
    private double price;
```

Notice that the entities match our proto file. The rest or Getters and Setters, so we can read and write data into them.

Lets go ahead and create a `Marshaller` for our Product class which will do exactly what we intend, read and write to our cache.

Create a new Java class called `ProductMarshaller.java` in `com.redhat.cloudnative.model` and copy the below code into the file:

```
package com.redhat.cloudnative.model;

import com.redhat.cloudnative.model.Product;
import org.infinispan.protostream.MessageMarshaller;

import java.io.IOException;

public class ProductMarshaller implements MessageMarshaller<Product> {

    /**
     * Proto file specimen
     * message Product {
     * required string itemId = 1;
     * required string name = 2;
     * required string desc = 3;
     * required double price = 4;
     * }
     */

    @Override
    public Product readFrom(ProtoStreamReader reader) throws IOException {
        String itemId = reader.readString("itemId");
        String name = reader.readString("name");
        String desc = reader.readString("desc");
        double price = reader.readDouble("price");

        return new Product(itemId, name, desc, price);
    }

    @Override
    public void writeTo(ProtoStreamWriter writer, Product product) throws IOException {
        writer.writeString("itemId", product.getItemId());
        writer.writeString("name", product.getName());
        writer.writeString("desc", product.getDesc());
        writer.writeDouble("price", product.getPrice());
    }

    @Override
    public Class<? extends Product> getJavaClass() {
        return Product.class;
    }

    @Override
    public String getTypeName() {
        return "coolstore.Product";
    }

}
```

So now we have the capability to read from a `ProtoStream` and `Write` to it. And this will be done directly into our cache. We have already created the other model classes and mashallers, feel free to look around.

Now its time to configure our `RemoteCache`, since its not embedded into our service.

Open the `Producers.java` file in the `com.redhat.cloudnative` directory/package.

We use the producer to ensure our RemoteCache gets instantiated. We create methods called getCache and getConfigBuilder

- getConfigBuilder: sets up the basic cache config
- getCache, sets up our marshallers and proto files
- other config properties are injected at runtime

Add this code below the `// TODO Add getCache` and `// TODO add getConfigBuilder` marker:

```
@Produces
RemoteCache<String, ShoppingCart> getCache() throws IOException {

   RemoteCacheManager manager = new RemoteCacheManager(getConfigBuilder().build());

   SerializationContext serCtx = ProtoStreamMarshaller.getSerializationContext(manager);
   FileDescriptorSource fds = new FileDescriptorSource();
   fds.addProtoFiles("META-INF/cart.proto");
   serCtx.registerProtoFiles(fds);
   serCtx.registerMarshaller(new ShoppingCartMarshaller());
   serCtx.registerMarshaller(new ShoppingCartItemMarshaller());
   serCtx.registerMarshaller(new ProductMarshaller());
   serCtx.registerMarshaller(new PromotionMarhsaller());
   return manager.getCache();
}

protected ConfigurationBuilder getConfigBuilder() {
   ConfigurationBuilder cfg = null;
   cfg = new ConfigurationBuilder().addServer()
         .host(dgHost)
         .port(dgPort)
         .marshaller(new ProtoStreamMarshaller())
         .clientIntelligence(ClientIntelligence.BASIC);

   return cfg;

}
```

`Perfect`, now we have all the building blocks ready to use the cache. Lets start using our cache.

Next we need to make sure we will inject our cache in our service. Open `com.redhat.cloudnative.service.ShoppingCartServiceImpl` and add this at the `// TODO Inject RemoteCache` marker:

```
@Inject
@Remote("default")
RemoteCache<String, ShoppingCart> carts;
```

Building `cart-service` REST API with Quarkus

The cart is quite simple; All the information from the browser i.e., via our `Angular App` is via `JSON` at the `/api/cart` endpoint:

- `GET` request `/{cartId}` gets the items in the cart, or creates a new unique ID if one is not present
- `POST` to `/{cartId}/{itemId}/{quantity}` will add items to the cart
- `DELETE` to `/{cartId}/{itemId}/{quantity}` will remove items from the cart. * And finally a `POST` to `/checkout/{cartId}` will remove the items and invoke the checkout procedure

Let's take a look at how we do this with Quarkus. In our `cart-service` project and in our main package i.e., `com.redhat.cloudnative` is the `CartResource`. Let's take a look at the getCart method.

At the `// TODO ADD getCart method` marker, add this method:

```
public ShoppingCart getCart(@PathParam("cartId") String cartId) {
   return shoppingCartService.getShoppingCart(cartId);
}
```

The code above is using the `ShoppingCartService`, which is injected into the `CartResource` via the Dependency Injection. The `ShoppingCartService` take a `cartId` as a parameter and returns the associated ShoppingCart. So that's perfect, however, for our Endpoint i.e., `CartResource` to respond, we need to define a couple of things:

- The type of HTTPRequest
- The type of data it can receive
- The path it resolves too

Add the following code on top of the `getCart` method

```
@GET
@Produces(MediaType.TEXT_PLAIN)
@Path("/{cartId}")
@Operation(summary = "get the contents of cart by cartId")
```

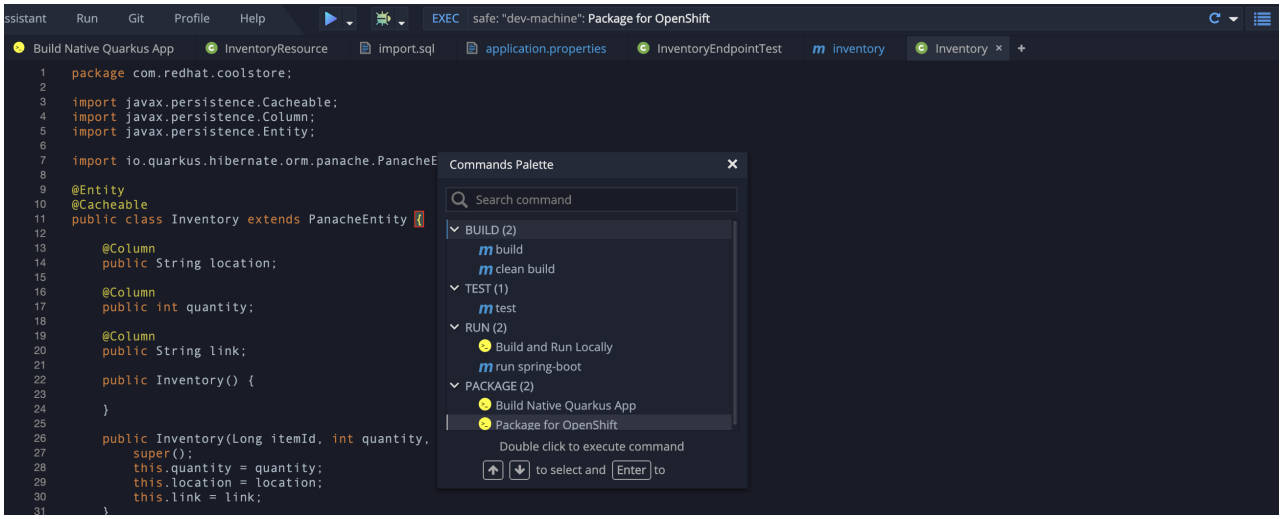
We have now successfully stated that the method adheres to a GET request and accepts data in `plain text`. The path would be `/api/cart/{cartId}` finally, we add the `@Operation` annotation for some documentation, which is important for other developers using our service.

Take this opportunity to look at some of the other methods. You will find `@POST` and `@DELETE` and also the paths they adhere too. This is how we can construct a simple Endpoint for our application.

> **NOTE** There are other `// TODO` markers and commented-out code we will use later. Leave them alone for now.

Package and Deploy the cart-service

Package the cart application via clicking on `Package for OpenShift` in `Commands Palette` :



Or run the following maven plugin in CodeReady WorkspacesTerminal:

```
cd /projects/cloud-native-workshop-v2m4-labs/cart-service/
```

```
mvn clean package -DskipTests
```

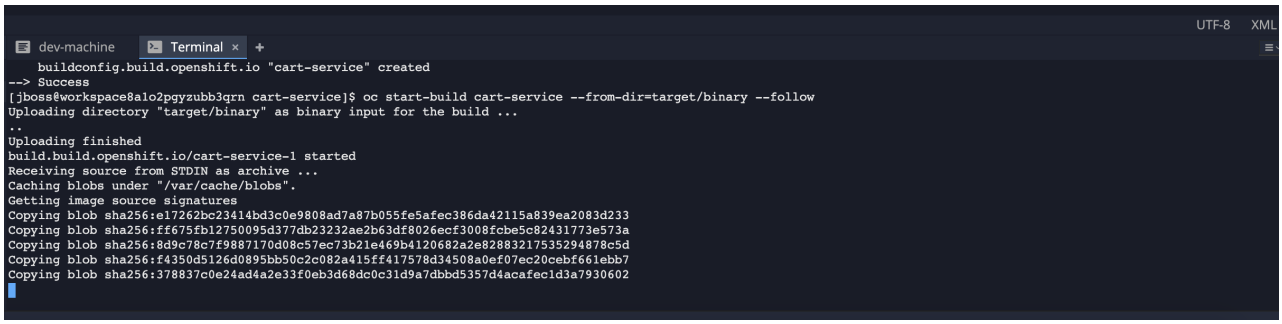
Build the image using on OpenShift:

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

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

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

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



Deploy it as an OpenShift application after the build is done:

```
oc new-app cart
```

Create the route

```
oc expose svc/cart
```

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

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

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

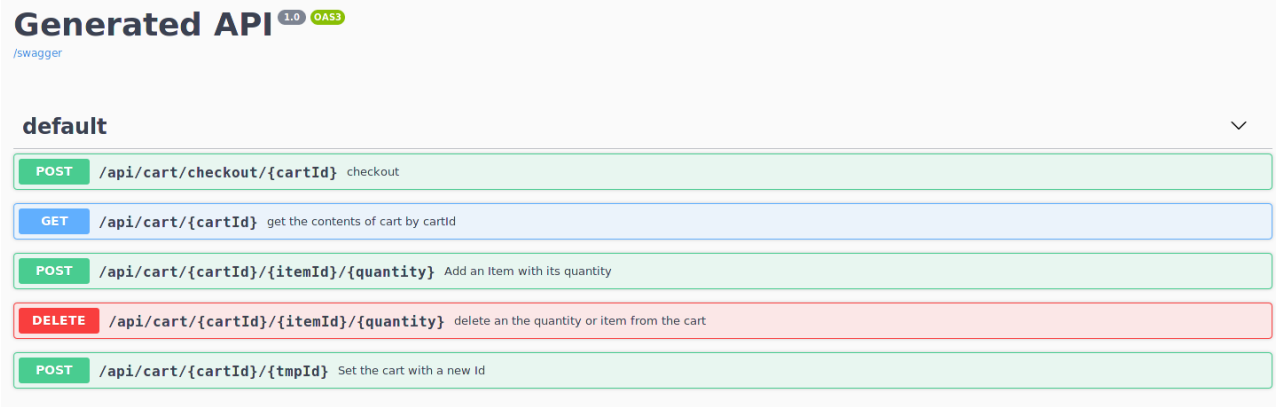
> NOTE: Even if the rollout command reports success the application may not be ready yet and the reason for that is that we currently don`t have any liveness check configured.

With the app deployed, we can check out the API page that Quarkus generates.

Run this command in the CodeReady Terminal to discover the URL to the app:

```
echo http://$(oc get route cart -o=go-template --template='{{ .spec.host }}')/swagger-ui
```

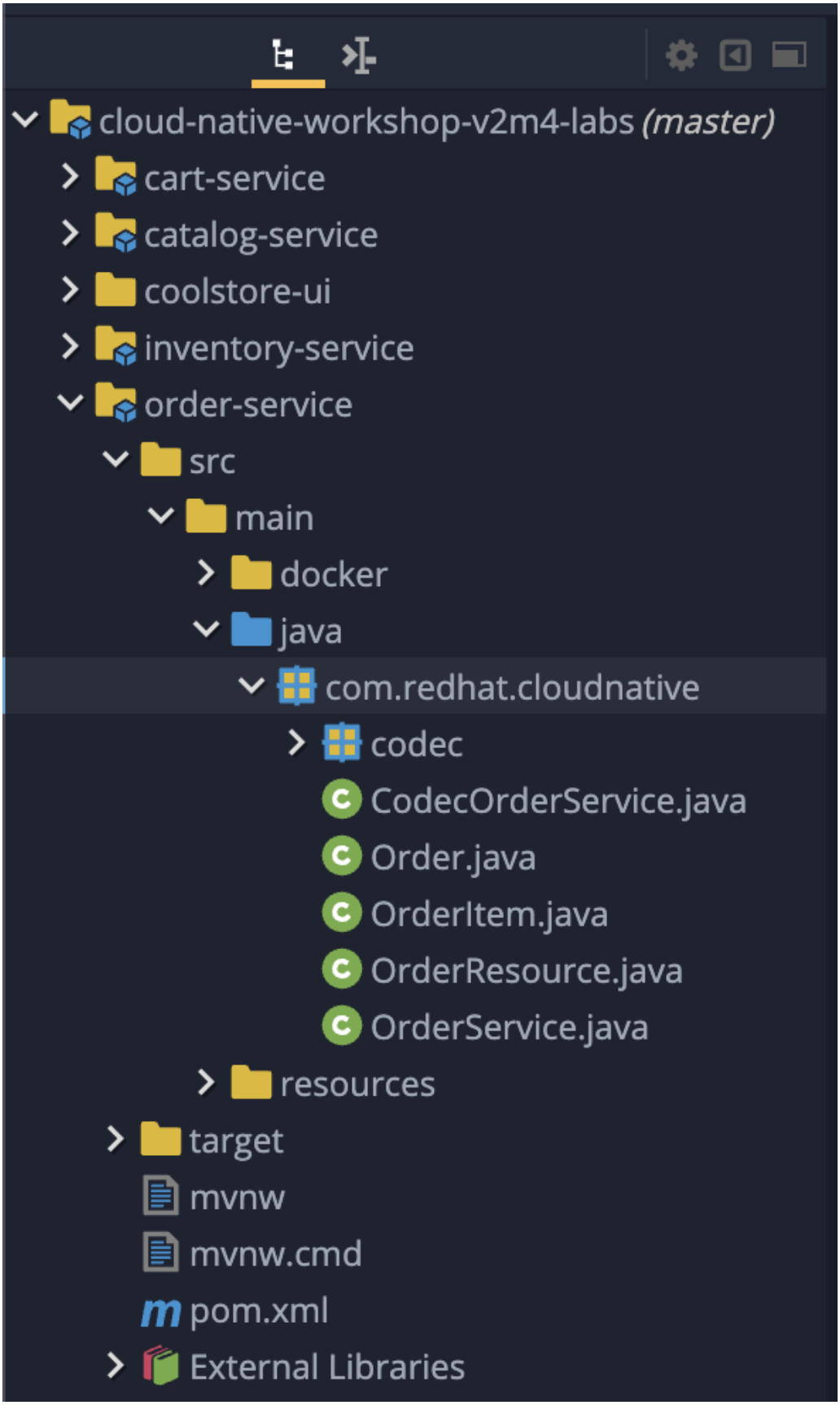
Open this URL in your browser!



Notice that the documentation after the methods, this is an excellent way for other service developers to know what you intend to do with each service method. You can try to invoke the methods and see the output from the service. Hence an excellent way to test quickly as well.

4. Developing and Deploying Order Service

`Order Service` manages all orders when customers checkout items in the shopping cart. Lets's go through quickly how the order service get `REST` services to use the `MongoDB` database with `Quarkus` Java runtimes. Go to `Project Explorer` in `CodeReady Workspaces` Web IDE and expand `order-service` directory.



The application built in `Quarkus` is quite simple: the user can add elements in a list using `RESTful APIs` and the list is updated. All the information between the client and the server are formatted as `JSON`. The elements are stored in `MongoDB`.

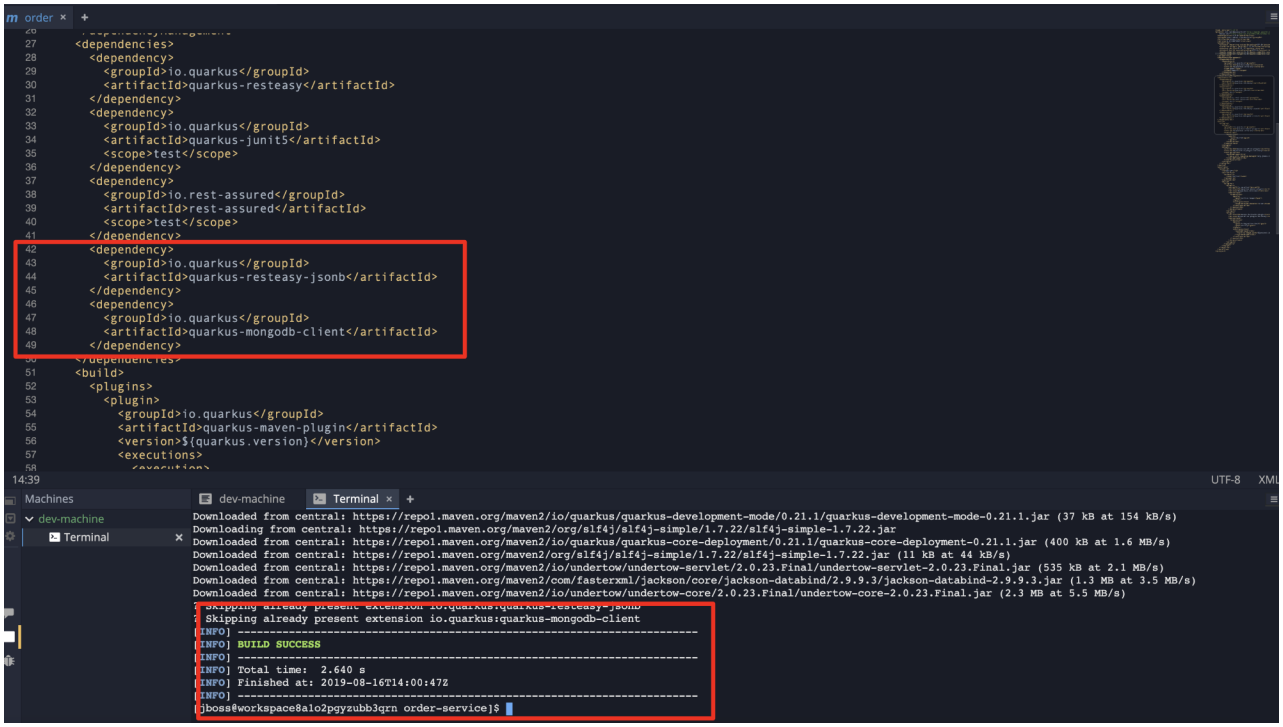
Adding Maven Dependencies using Quarkus Extensions

Execute the following command via CodeReady Workspaces Terminal:

```
cd /projects/cloud-native-workshop-v2m4-labs/order-service/
```

```
mvn quarkus:add-extension -Dextensions="resteasy-jsonb,mongodb-client"
```

This command generates a Maven structure importing the RESTEasy/JAX-RS, JSON-B and MongoDB Client extensions. After this, the quarkus-mongodb-client extension has been added to your `pom.xml` .



Creating Order Service using JSON REST service

First, let's have a look at the `Order` bean in `src/main/java/com/redhat/cloudnative/` as follows:

```java
package com.redhat.cloudnative;

public class Order {

    private String orderId;
    private String name;
    private String total;
    private String ccNumber;
    private String ccExp;
    private String billingAddress;
    private String status;

    public Order() {
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }

    public String getOrderId() {
        return orderId;
    }
```

Nothing fancy. One important thing to note is that having a default constructor is required by the `JSON serialization layer` .

Now, open a `com.redhat.cloudnative.OrderService` that will be the business layer of our application and `store/load` the orders from the mongoDB database. Add the following Java codes at each market.

`// TODO: Inject MongoClient here` marker:

```
@Inject MongoClient mongoClient;
```

`// TODO: Add a while loop to make an order lists using MongoCursor here` marker in `list()` method:

```
MongoCursor<Document> cursor = getCollection().find().iterator();

try {
   while (cursor.hasNext()) {
      Document document = cursor.next();
      Order order = new Order();
      order.setOrderId(document.getString("orderId"));
      order.setName(document.getString("name"));
      order.setTotal(document.getString("total"));
      order.setCcNumber(document.getString("ccNumber"));
      order.setCcExp(document.getString("ccExp"));
      order.setBillingAddress(document.getString("billingAddress"));
      order.setStatus(document.getString("status"));
      list.add(order);
   }
} finally {
   cursor.close();
}
```

`// TODO: Add to create a Document based order here` marker in `add(Order order)` method:

```
Document document = new Document()
      .append("orderId", order.getOrderId())
      .append("name", order.getName())
      .append("total", order.getTotal())
      .append("ccNumber", order.getCcNumber())
      .append("ccExp", order.getCcExp())
      .append("billingAddress", order.getBillingAddress())
      .append("status", order.getStatus());
getCollection().insertOne(document);
```

Now, edit the `com.redhat.cloudnative.OrderResource` class as follows in each marker:

`// TODO: Add JAX-RS annotations here` marker:

```
@Path("/api/orders")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
```

`// TODO: Inject OrderService here` marker:

```
@Inject OrderService orderService;
```

`// TODO: Add list(), add(), updateStatus() methods here` marker:

```
    @GET
    public List<Order> list() {
        return orderService.list();
    }

    @POST
    public List<Order> add(Order order) {
        orderService.add(order);
        return list();
    }

    @GET
    @Path("/{orderId}/{status}")
    public List<Order> updateStatus(@PathParam("orderId") String orderId, @PathParam("status")
String status) {
        orderService.updateStatus(orderId, status);
        return list();
    }
```

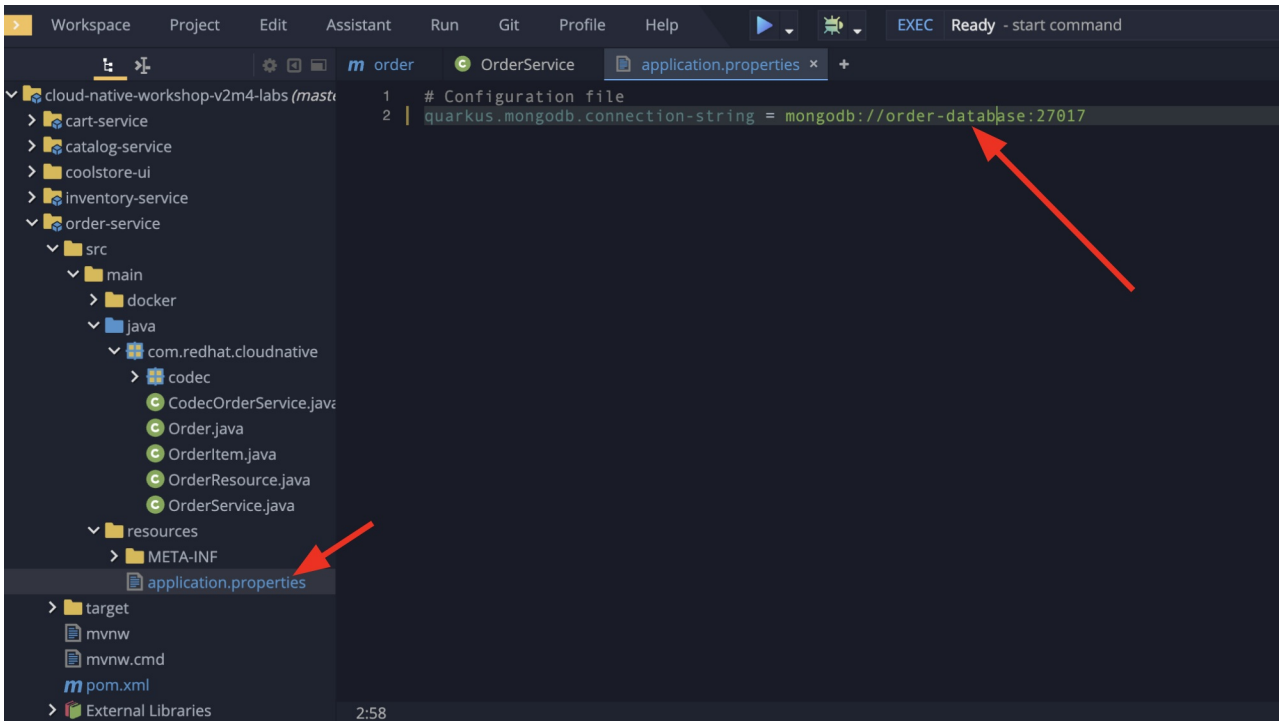
The implementation is pretty straightforward and you just need to define your endpoints using the `JAX-RS annotations` and use the `OrderService` to list/add new orders.

Configuring the MongoDB database

The main property to configure is the URL to access to `MongoDB,` almost all configuration can be included in the connection URI so we advise you to do so, you can find more information in the MongoDB documentation

Open `application.properties` in `src/main/resources/` and add the following configuration:

```
quarkus.mongodb.connection-string = mongodb://order-database:27017
```



Simplifying MongoDB Client usage using BSON codec

By using a Bson `Codec`, the MongoDB Client will take care of the transformation of your domain object to/from a MongoDB `Document` automatically.

First you need to create a Bson `Codec` that will tell Bson how to transform your entity to/from a MongoDB `Document`. Here we use a `CollectibleCodec` as our object is retrievable from the database (it has a MongoDB identifier), if not we would have used a `Codec` instead. More information in the codec documentation.

Edit the `com.redhat.cloudnative.codec.OrderCodec` class as follows:

`// TODO: Add Encode & Decode contexts here` marker:

```
@Override
public void encode(BsonWriter writer, Order Order, EncoderContext encoderContext) {
    Document doc = new Document();
    doc.put("orderId", Order.getOrderId());
    doc.put("name", Order.getName());
    doc.put("total", Order.getTotal());
    doc.put("ccNumber", Order.getCcNumber());
    doc.put("ccExp", Order.getCcExp());
    doc.put("billingAddress", Order.getBillingAddress());
    doc.put("status", Order.getStatus());
    documentCodec.encode(writer, doc, encoderContext);
}

@Override
public Class<Order> getEncoderClass() {
    return Order.class;
}

@Override
public Order generateIdIfAbsentFromDocument(Order document) {
    if (!documentHasId(document)) {
        document.setOrderId(UUID.randomUUID().toString());
    }
    return document;
}

@Override
public boolean documentHasId(Order document) {
    return document.getOrderId() != null;
}

@Override
public BsonValue getDocumentId(Order document) {
    return new BsonString(document.getOrderId());
}

@Override
public Order decode(BsonReader reader, DecoderContext decoderContext) {
    Document document = documentCodec.decode(reader, decoderContext);
    Order order = new Order();
    if (document.getString("orderId") != null) {
        order.setOrderId(document.getString("orderId"));
    }
    order.setName(document.getString("name"));
    order.setTotal(document.getString("total"));
    order.setCcNumber(document.getString("ccNumber"));
    order.setCcExp(document.getString("ccExp"));
    order.setBillingAddress(document.getString("billingAddress"));
    order.setStatus(document.getString("status"));
    return order;
}
```

Then you need to create a `CodecProvider` to link this `Codec` to the Order class.

Edit the `com.redhat.cloudnative.codec.OrderCodecProvider` class as follows:

`// TODO: Add Codec get method here` marker:

```
@Override
public <T> Codec<T> get(Class<T> clazz, CodecRegistry registry) {
    if (clazz == Order.class) {
        return (Codec<T>) new OrderCodec();
    }
    return null;
}
```

`Quarkus` will register the `CodecProvider` for you.

Finally, when getting the `MongoCollection` from the database you can use directly the `Order` class instead of the `Document` one, the codec will automatically map the `Document` to/from your `Order` class.

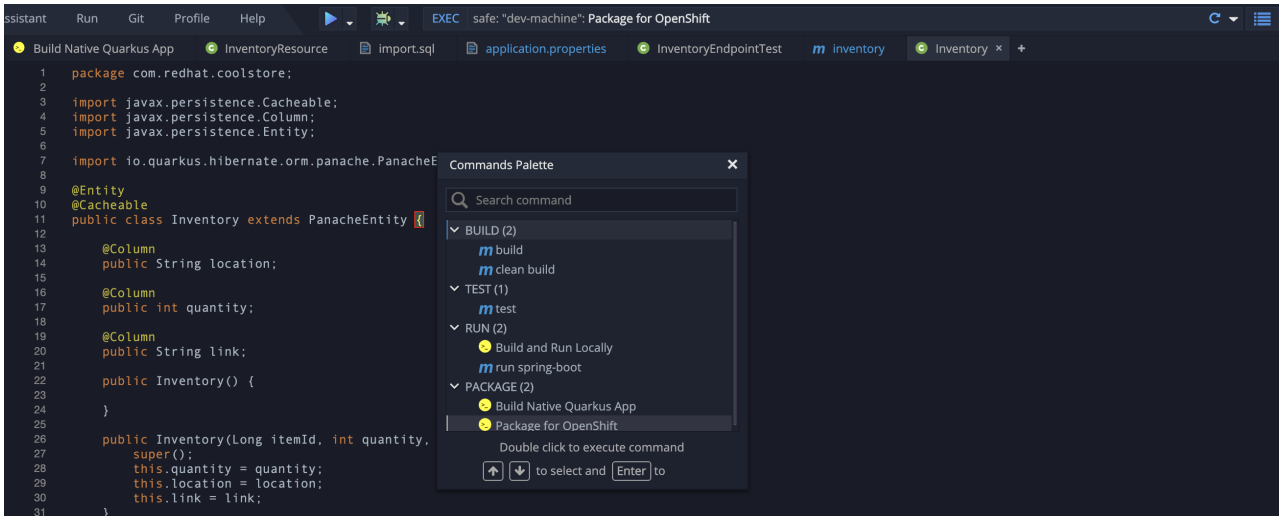
Edit the `com.redhat.cloudnative.CodecOrderService` class as follows:

`// TODO: Add MongoCollection method here` marker:

```
private MongoCollection<Order> getCollection(){
    return mongoClient.getDatabase("order").getCollection("order", Order.class);
}
```

Building and Deploying Application to OpenShift

Package the cart application via clicking on `Package for OpenShift` in `Commands Palette` :



Or run the following maven plugin in CodeReady WorkspacesTerminal:

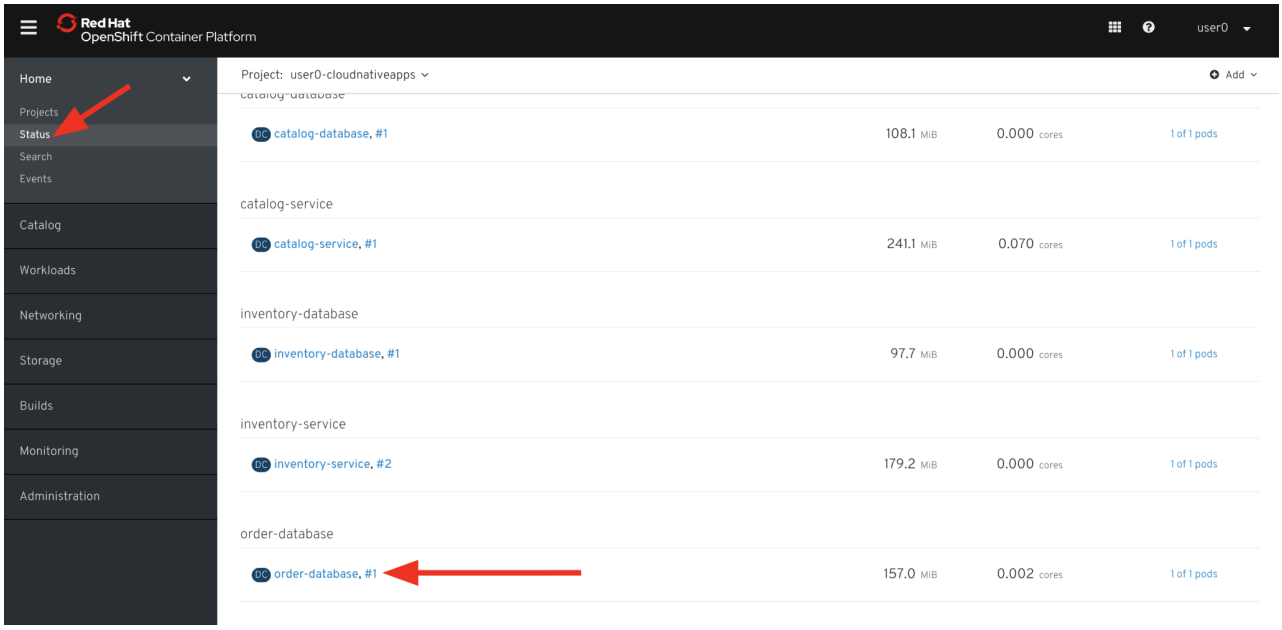
`mvn clean package -DskipTests`

Deploying Order service with MongoDB to OpenShift

Run the following `oc` command to deploy a `MongoDB` to OpenShift via CodeReady Workspaces Terminal:

```
oc new-app --docker-image mongo:4.0 --name=order-database
```

Once the MongoDB is deployed successfully, it will be showd in `Project Status`.



Build the image using on OpenShift:

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

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

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

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

Deploy it as an OpenShift application after the build is done:

```
oc new-app order
```

Create the route

```
oc expose svc/order
```

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

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

Wait for that command to report replication controller `order-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 all inventories:

Get the route URL

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

```
curl $URL/api/orders ; echo
```

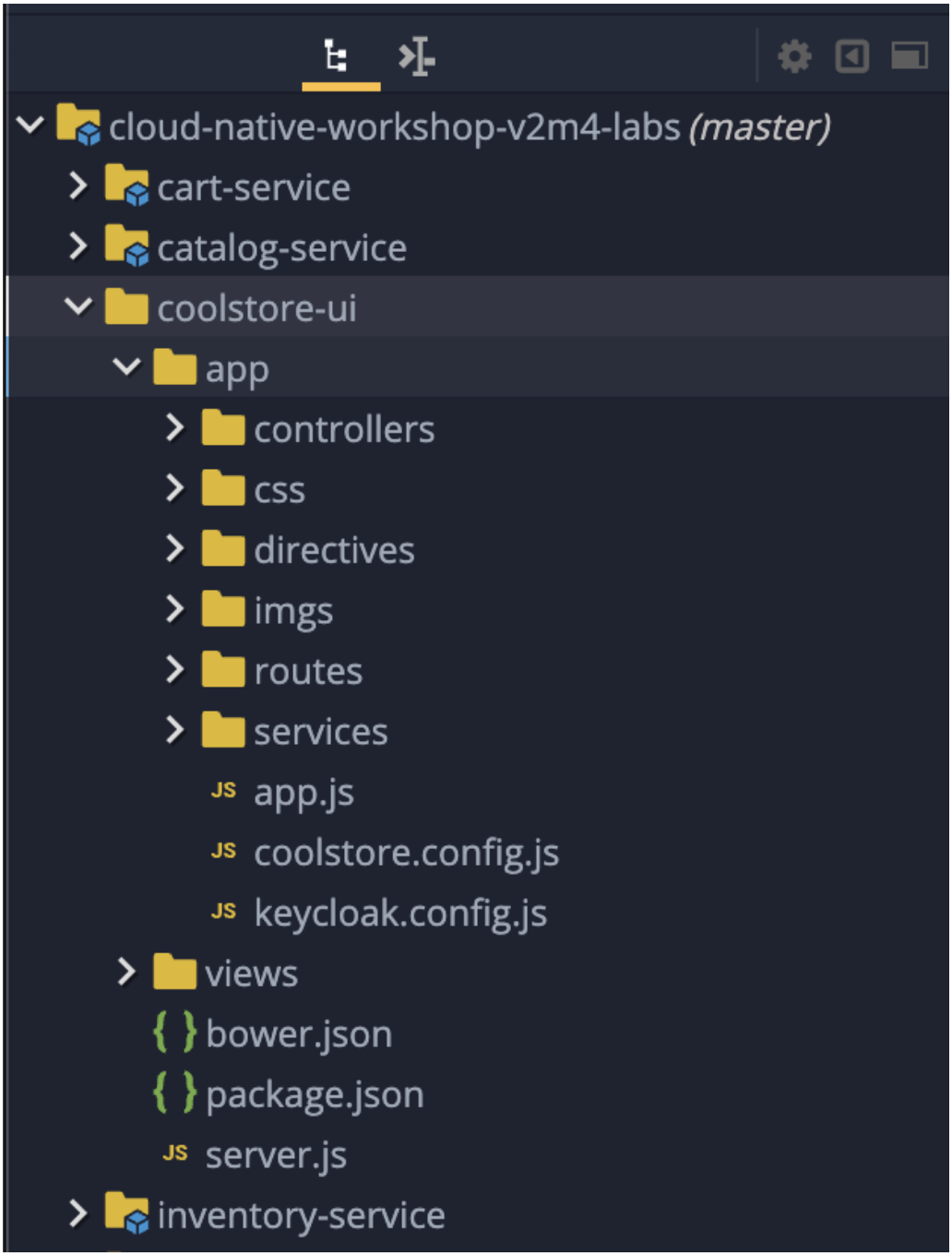
You will see empty result because you didn't add any shopping items yet:

```
[]
```

5. Deploying WEB-UI Service

`WEB-UI Service` serves a frontend based on AngularJS and PatternFly running in a Node.js container. Red Hat OpenShift Application Runtimes includes `Node.js` support in enterprise prouction environment.

Lets's go through quickly how the frontend service works and built on `Node.js` runtimes. Go to `Project Explorer` in `CodeReady Workspaces` Web IDE and expand `coolstore-ui` directory.



You will see javascripts for specific cloud-native services such as cart, catatlog, and order service as above.

Now, we will deploy a presentation layer to OpenShift cluster using `Nodeshift` command line tool, a programmable API that you can use to deploy Node.js projects to `OpenShift`.

Install the `Nodeshift` tool via CodeReady Workspaces Terminal:

```
cd /projects/cloud-native-workshop-v2m4-labs/coolstore-ui/
```

```
npm install --save-dev nodeshift
```

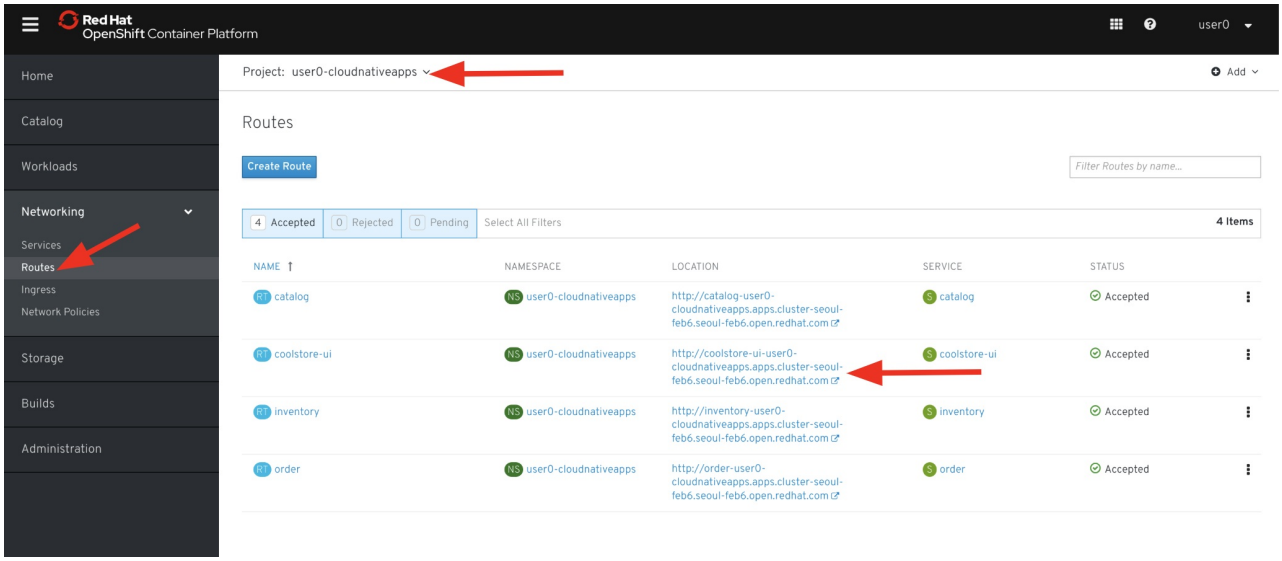
Deploy the web-ui service using `Nodeshift` via CodeReady Workspaces Terminal and it will take a couple of minutes to complete the web-ui application deployment:

```
npm run nodeshift
```

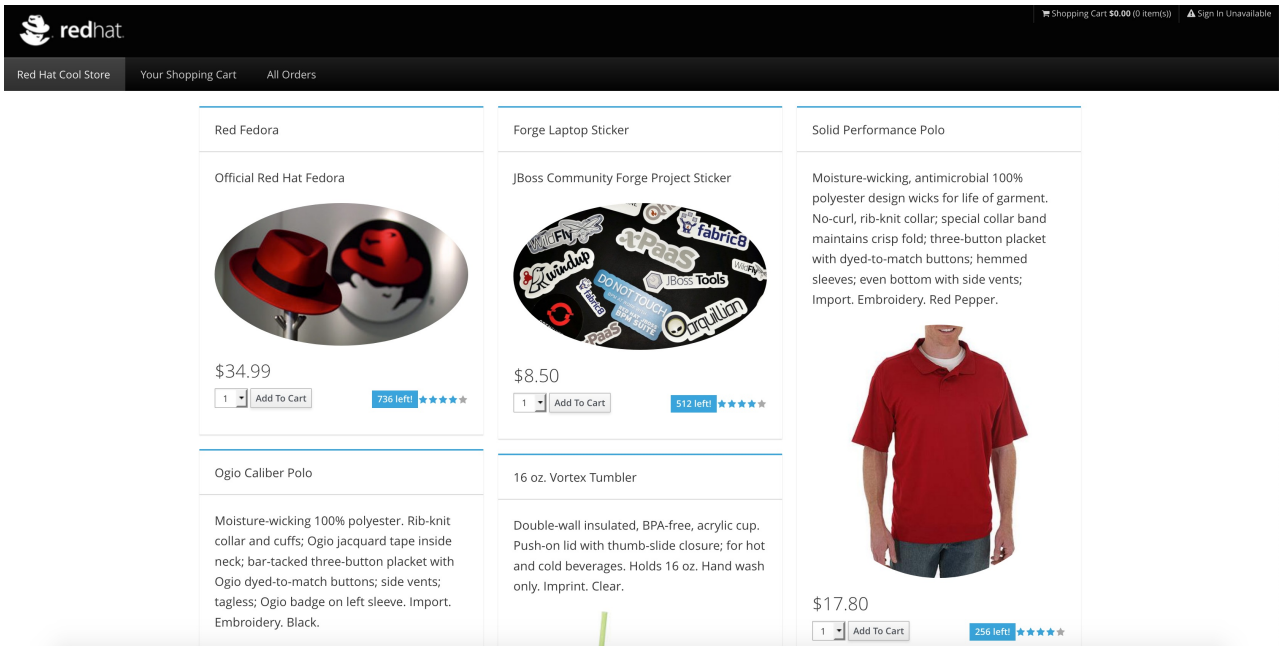
Create the route

`oc expose svc/coolstore-ui`

Go to `Networking > Routes` in OpenShift web console and click on the route URL of `coolstore-ui` :



You will see the prouct page of `Red Hat Cool Store` as below:



Summary

In this scenario we developed five microservices with `REST API` exposure to communicate with the other microservices. We also used a variety of application runtimes such as `Quarkus` , `Spring Boot` , and `NodeJS` to compile, package, and containerize applications which is a major capability of the advanced cloud-native architecture.

To deploy the cloud-native applications with multiple datasources on `OpenShift` cluster, `Quarkus` provides an easy way to connect multiple datasources and obtain a reference to those datasources such as `PostgreSQL` and `MongoDB` in code.

In the end, we optimized `data transaction performance` of the shopping cart service thru integrating with a `JBoss Data Grid` to increase end users'(customers) satification. `Congratulations!`

Creating Event-Driven Service

Lab2 - Creating Event-Driven/Reactive Services

Traditional microservice architecture is typically composed of many individual services with different functions. Each application service has many clients that need to communicate with the service for fetching data. It will become more complex to handle data streams because everything can be a stream of data such as end-user clicks, RESTful APIs, IoT devices generating data. And complexity rises when these services are running on hybrid or multi-cloud infrastructure.

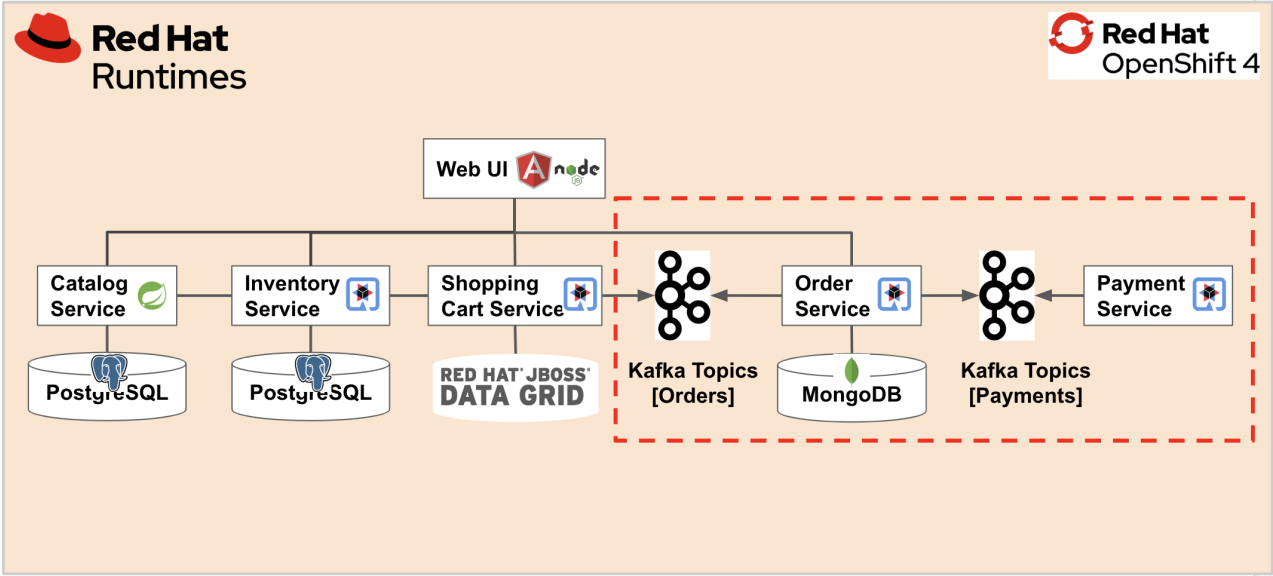
In an event-driven architecture, we can treat data streams as *events* using reactive programming and distributed messaging. *Reactive programming* is an asynchronous programming paradigm concerned with data streams and the propagation of change. In the previous lab, we developed Inventory, Catalog, Shopping Cart and Order services with obvious interactions.

In this lab, we'll change our shopping cart and order implementation and add a payment service as an Event-Driven/Reactive application in our cloud-native appliation architecture. These cloud-native applications will use AMQ Streams (based on Apache Kafka) as a messaging/streaming backbome. AMQ Streams makes it easy to run Apache Kafka on OpenShift with key features:

- Designed for horizontal scalability
- Message ordering guarantee at the partition level
- Message rewind/replay - *Long term* storage allows the reconstruction of an application state by replaying the messages

Goals of this lab

The goal is to develop advanced cloud-native applications on **Red Hat Runtimes** and deploy them on **OpenShift 4** including **AMQ Streams** for distributed messaging capabilities. After this lab, you should end up with something like:



Scalability is one of the flagship features of Apache Kafka. It is achieved by partitioning the data and distributing them across multiple brokers. Such data sharding also has a big impact on how clients connect and use the broker. This is especially visible when Kafka is running within a platform like Kubernetes but is accessed from outside of that platform.

Strimzi is an open source project that provides container images and operators for running Apache Kafka.

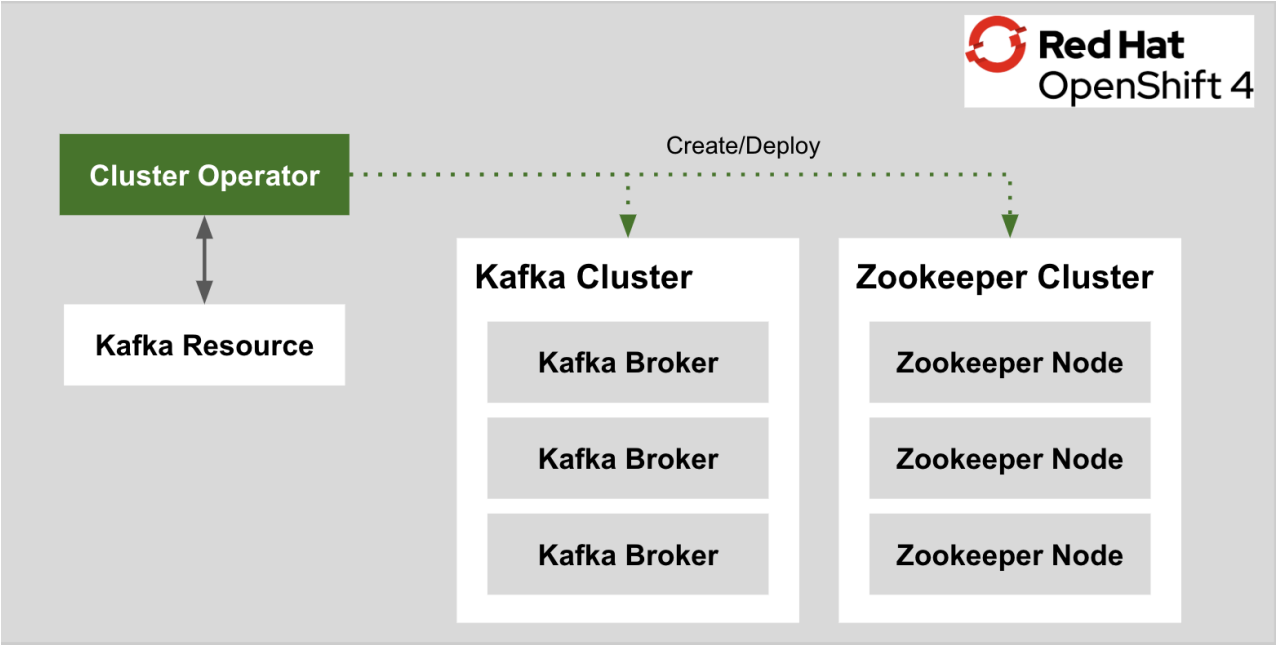
In this lab, we will use productized and supported versions of the Strimzi and Apache Kafka projects through Red Hat AMQ.

1. Create a Kafka Cluster and Topics

AMQ Streams is already installed using the following *Operators* so you don't need to install it in this lab:

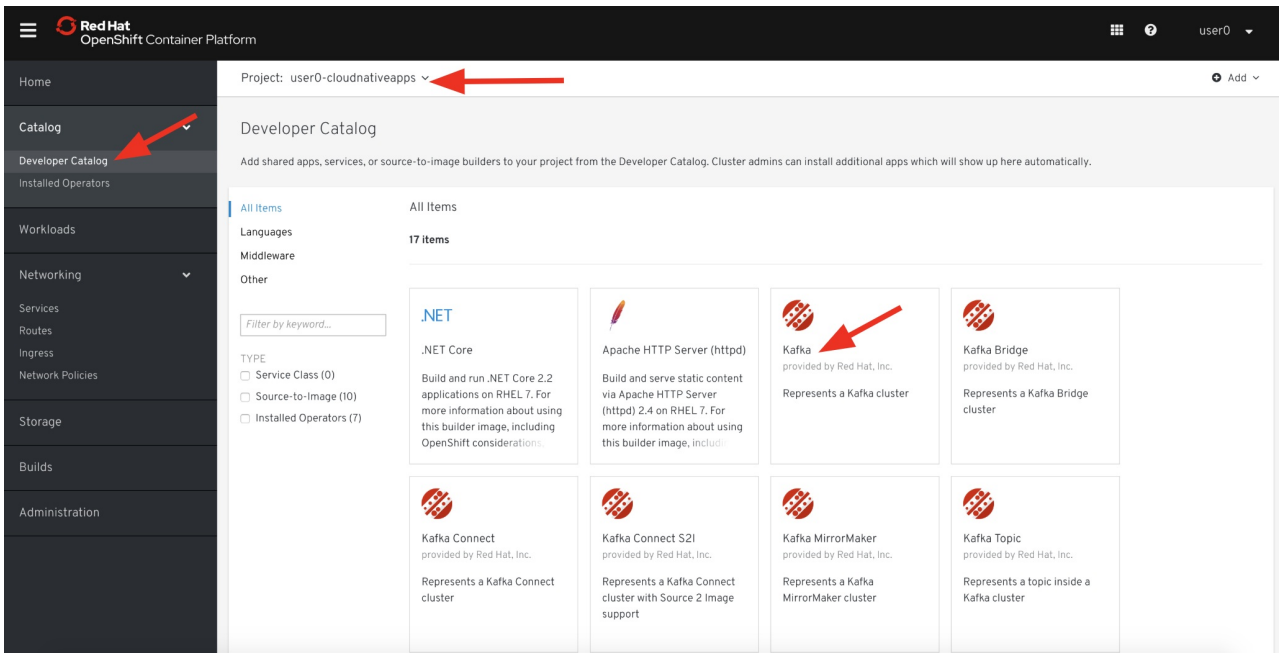
- `Kafka Operator` - Responsible for deploying and managing Apache Kafka clusters within an OpenShift cluster.
- `Topic Operator` - Responsible for managing Kafka topics within a Kafka cluster running within an OpenShift cluster.
- `User Operator` - Responsible for managing Kafka users within a Kafka cluster running within an OpenShift cluster.

The basic architecture of operators in AMQ is seen below:



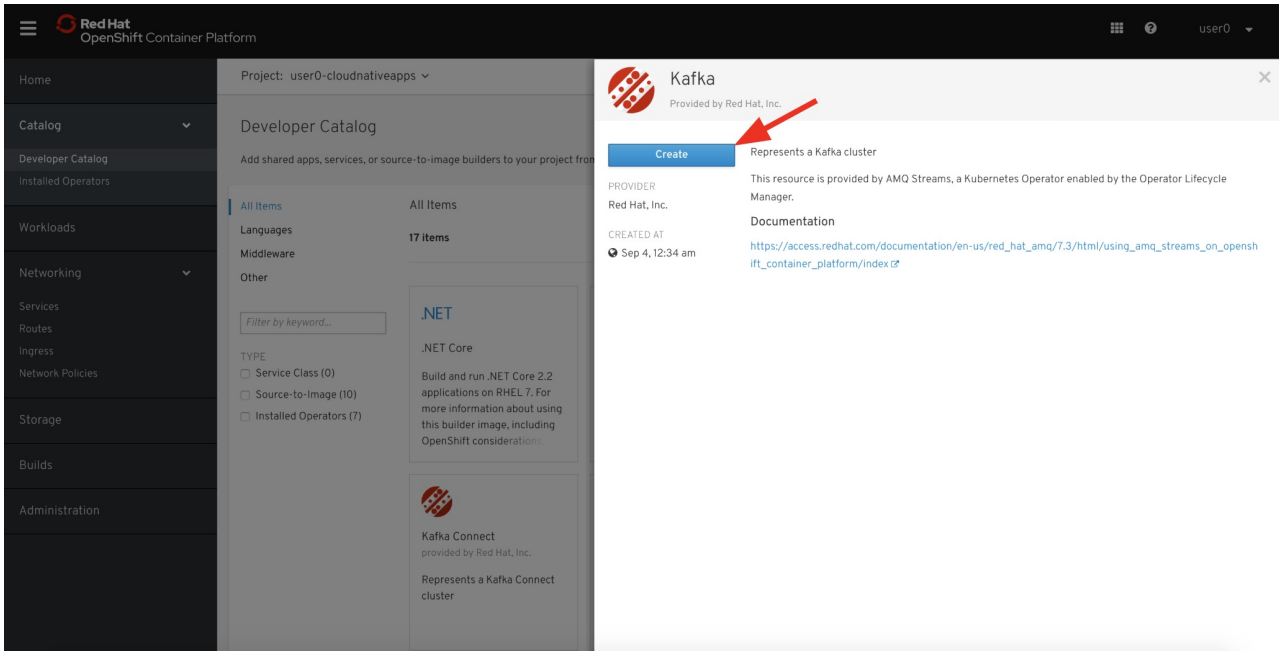
Creating a `Kafka cluster` in `userXX-cloudnativeapp` project

Navigate to *Catalog* > *Developer Catalog* in the left menu. In the search box, type in 'kafka' and Click on the `Kafka` box.

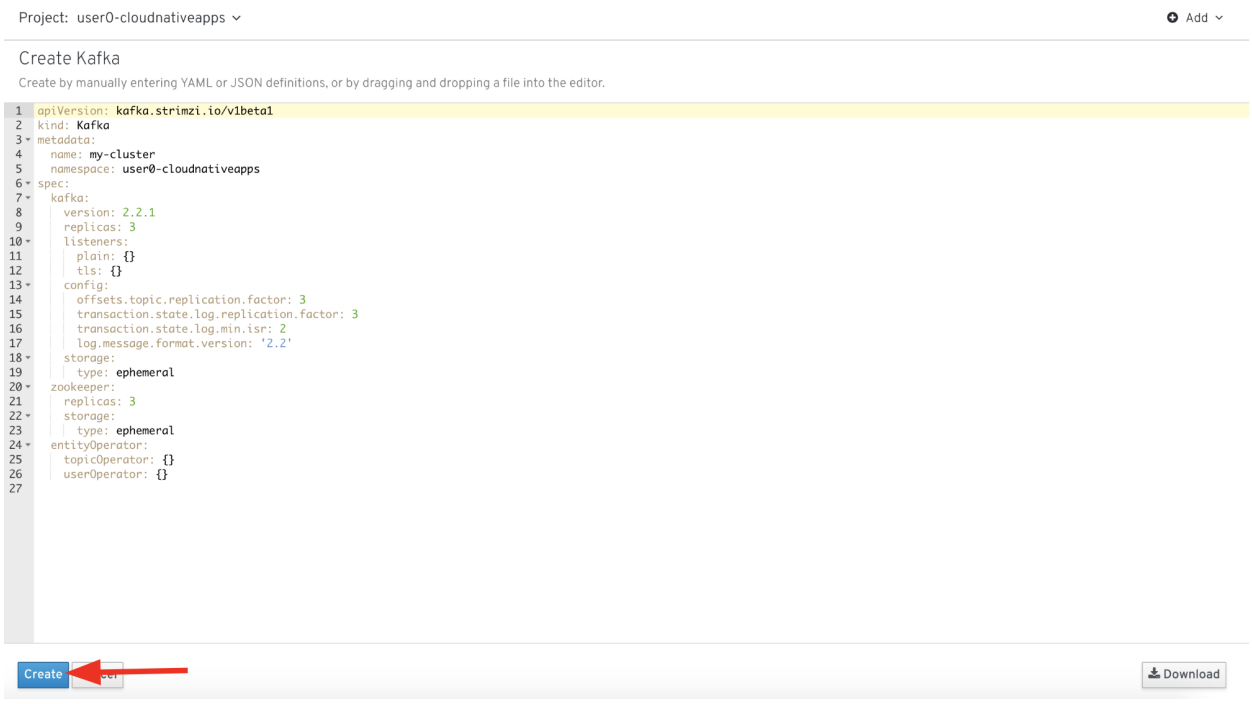


Click on **Create** to represent a Kafka cluster using AMQ Streams Operator.

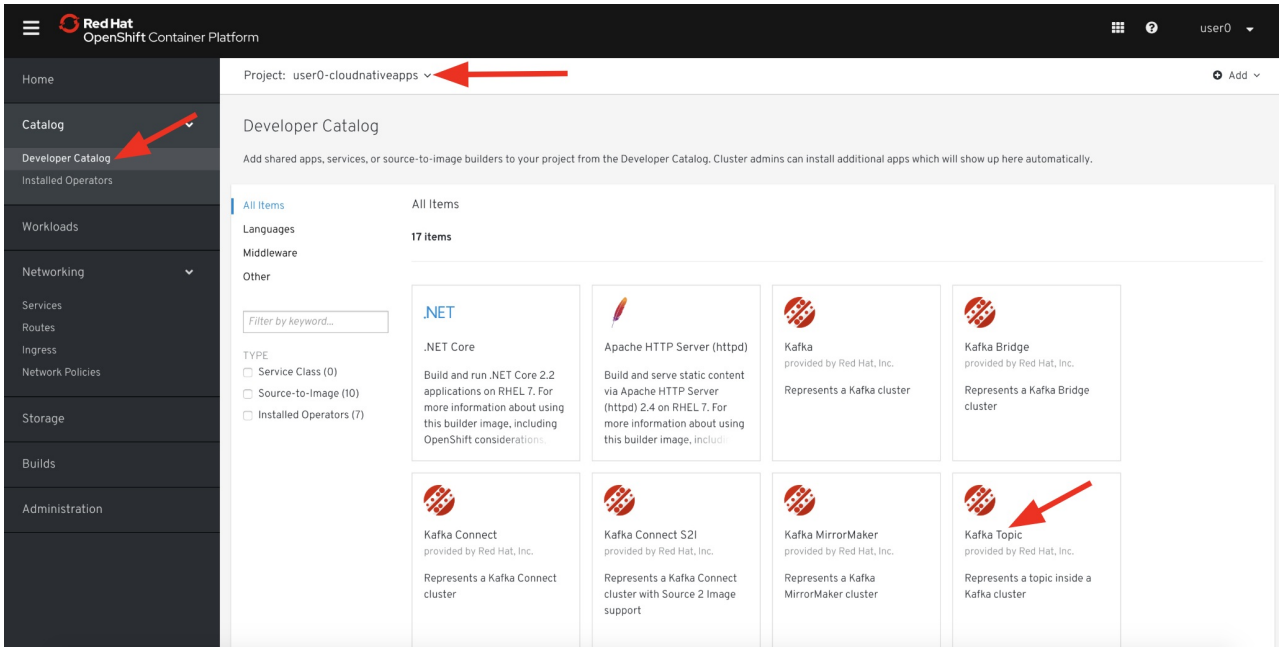
> **WARNING**
>
> Be sure you are in your `userXX-cloudnativeapp` project in the drop-down menu at the top. If you are in any other project, and try to create things, it will fail with permission denied!



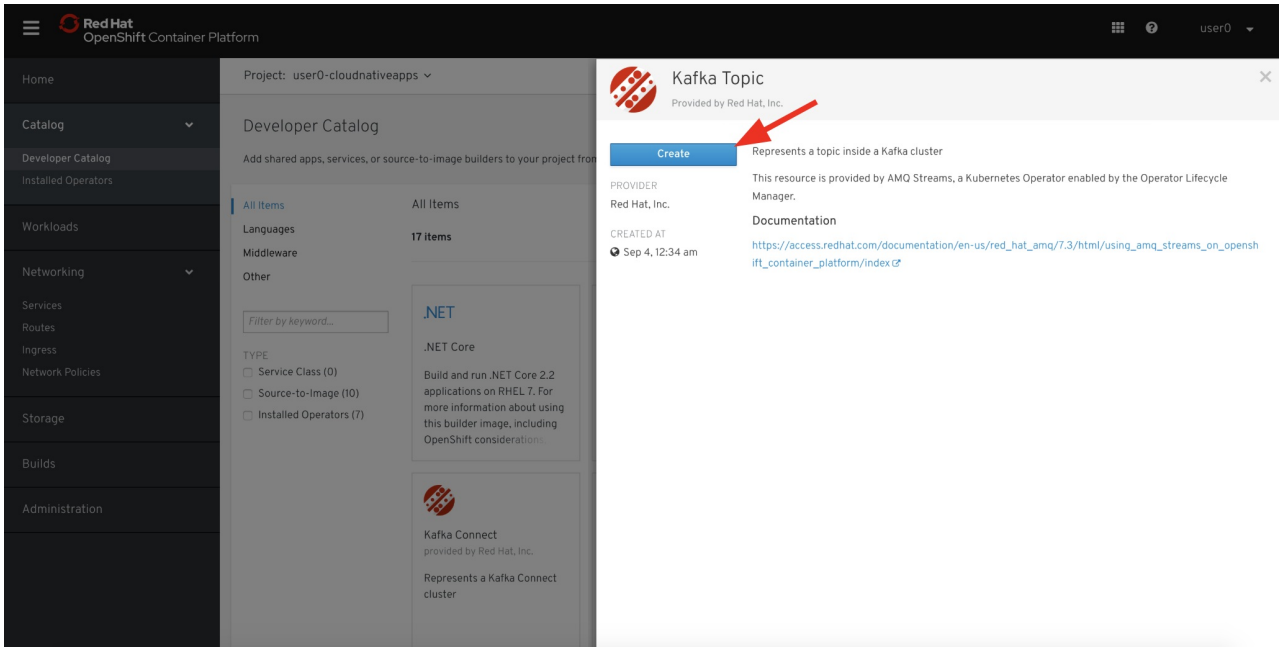
You will enter YAML editor that defines a `Kafka` Cluster. Keep the all values as-is then click on **Create** on the bottom.



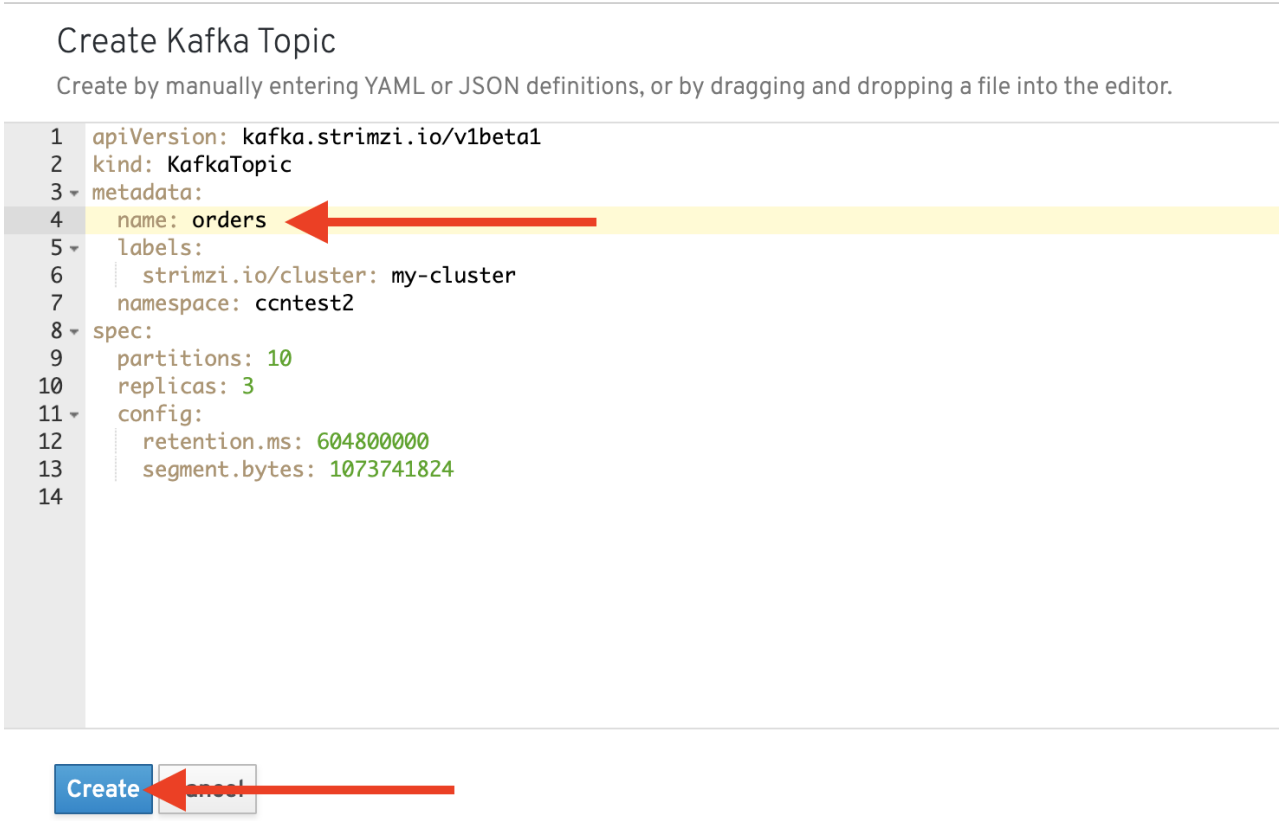
Next, we will create Kafka *Topic*. Return to *Catalog* > *Developer Catalog* and type in `kafka` but this time click on the **Kafka Topic** box.



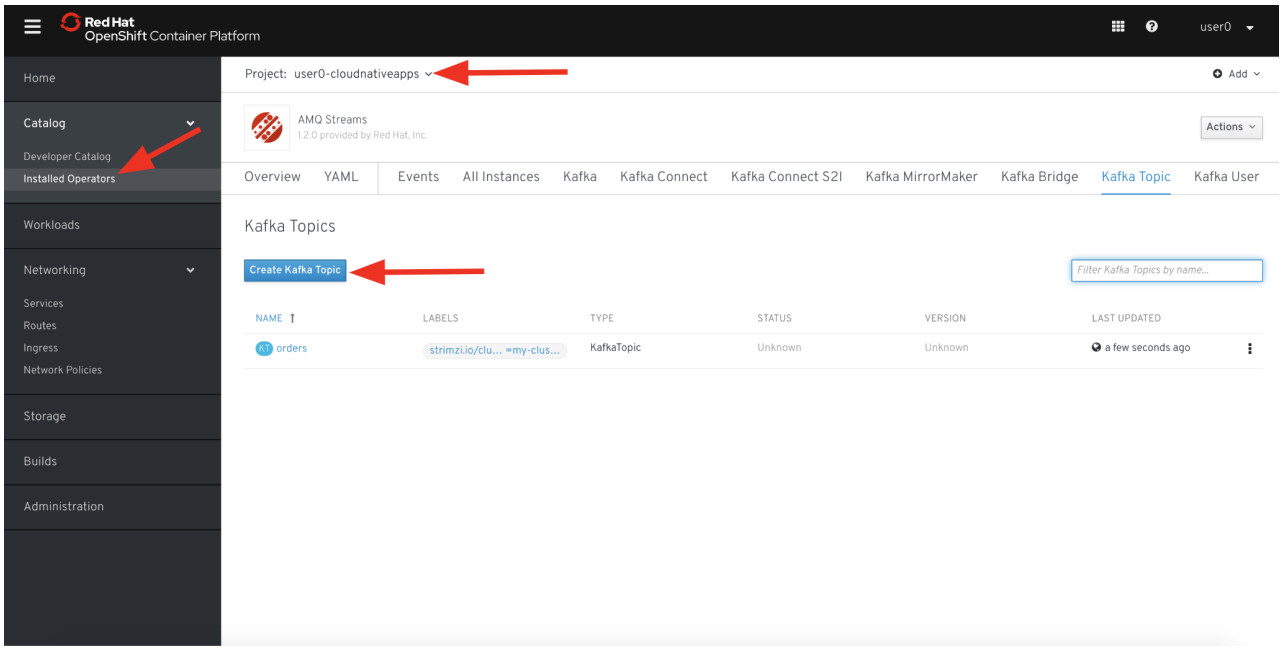
Click on **Create** to create a topic inside the Kafka cluster.



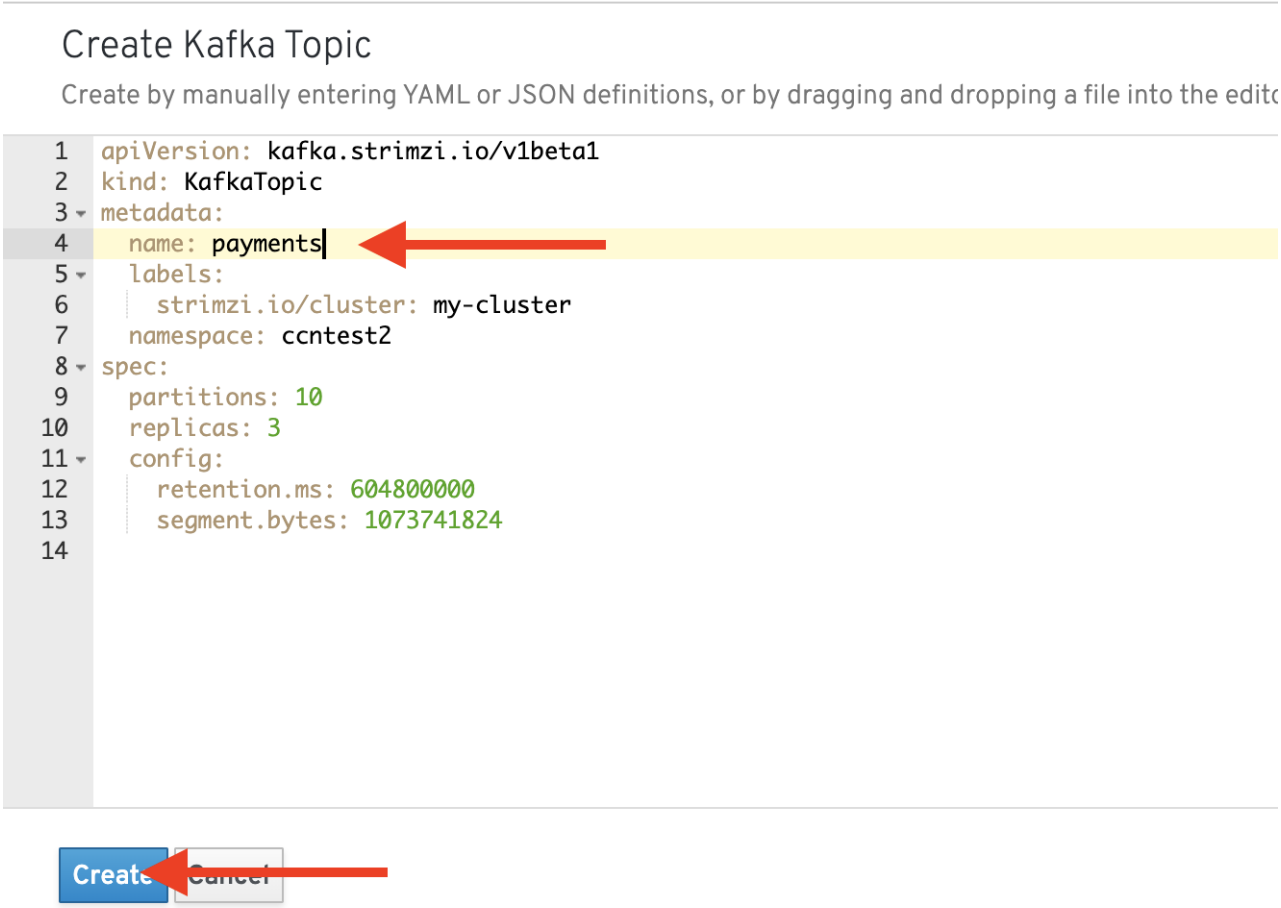
You will enter YAML editor that defines a `KafkaTopic` object. Change the name to `orders` as shown then click on **Create** on the bottom.



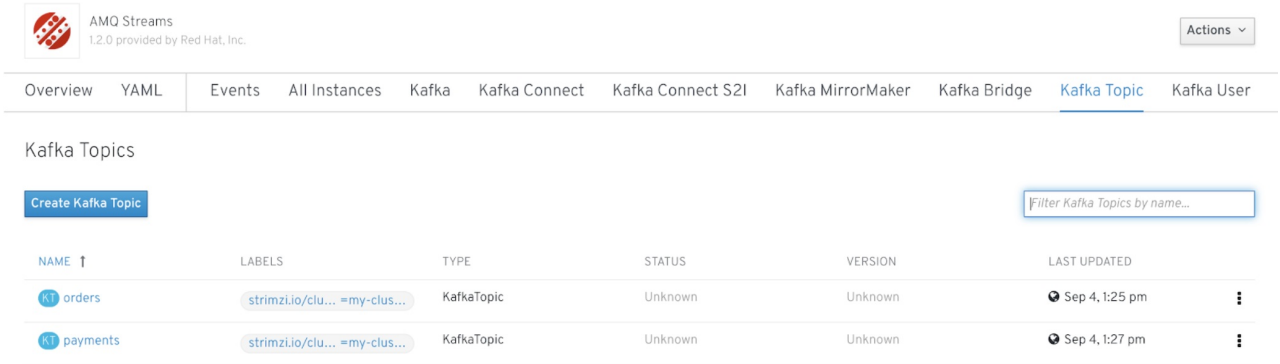
Create another topic using the same process as above, but called `payments`



Change the name to `payments` then click on **Create** on the bottom.



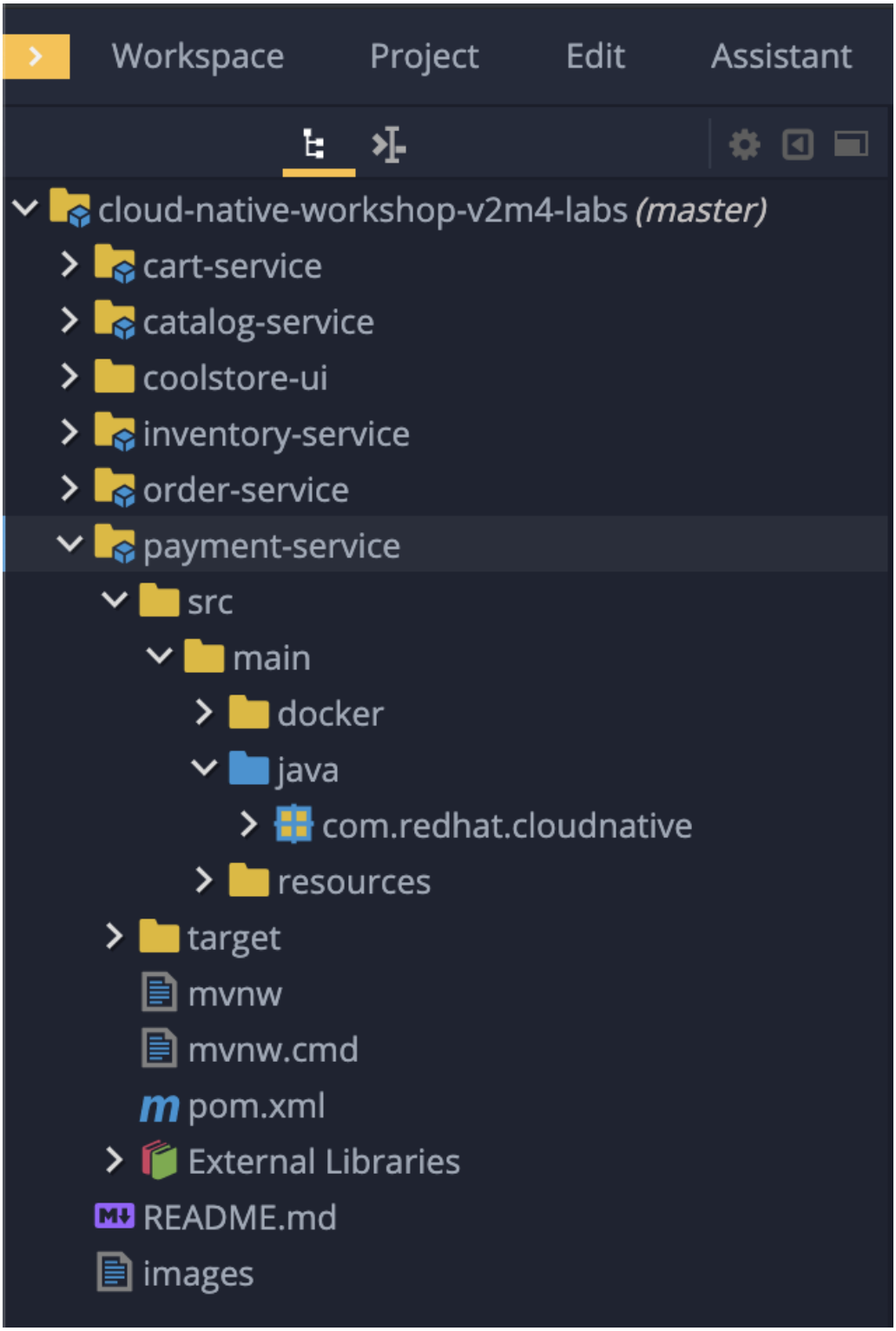
Well done! You now have a running Kafka cluster with two Kafka Topics called `payments` and `orders`.



2. Develop and Deploy Payment Service

Our **Payment Service** will offer online services for accepting electronic payments by a variety of payment methods including credit card or bank-based payments when orders are checked out in shopping cart. It doesn't really do anything but will represent a payment microservice that will "process" online shopping orders as they are posted to our services.

In CodeReady Workspaces, navigate to the `payment-service` directory.



In this step, we will learn how our Quarkus-based payment service can use Kafka to receive order events and *react* with payment events.

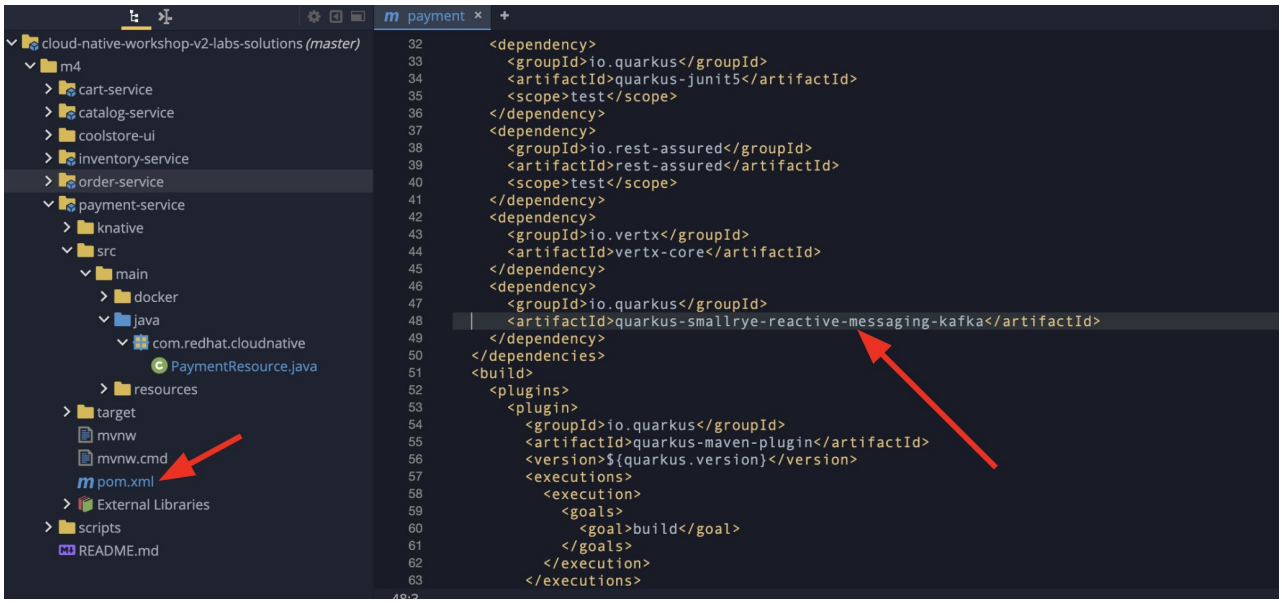
Adding Maven Dependencies using Quarkus Extensions

Execute the following command via CodeReady Workspaces *Terminal*:

```
cd /projects/cloud-native-workshop-v2m4-labs/payment-service/
```

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

This command imports the Kafka extensions for Quarkus applications and provides all the necessary capabilities to integrate with Kafka clusters. Confirm your `pom.xml` looks as below, with the new dependencies:



Writing the application

Let's start by adding fields to access configuration using `@ConfigProperty` and a `Producer` field which will be used to send messages. We'll also add a `log` field so we can see debug messages later on.

> Add this code to the `PaymentResource.java` file (in the `src/main/java/com/redhat/cloudnative` directory) at the `// TODO: Add Messaging ConfigProperty here` marker:

```
@ConfigProperty(name = "mp.messaging.outgoing.payments.bootstrap.servers")
public String bootstrapServers;

@ConfigProperty(name = "mp.messaging.outgoing.payments.topic")
public String paymentsTopic;

@ConfigProperty(name = "mp.messaging.outgoing.payments.value.serializer")
public String paymentsTopicValueSerializer;

@ConfigProperty(name = "mp.messaging.outgoing.payments.key.serializer")
public String paymentsTopicKeySerializer;

private Producer<String, String> producer;

public static final Logger log = LoggerFactory.getLogger(PaymentResource.class);
```

Next, we need a method to handle incoming events, which in this lab will be coming directly from Kafka, but later will come through as HTTP POST events.

> Add this code at the `// TODO: Add handleCloudEvent method here` marker:

```
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.TEXT_PLAIN)
    public void handleCloudEvent(String cloudEventJson) {
        String orderId = "unknown";
        String paymentId = "" + ((int)(Math.floor(Math.random() * 100000)));

        try {
            log.info("received event: " + cloudEventJson);
            JsonObject event = new JsonObject(cloudEventJson);
            orderId = event.getString("orderId");
            String total = event.getString("total");
            JsonObject ccDetails = event.getJsonObject("creditCard");
            String name = event.getString("name");

            // fake processing time
            Thread.sleep(5000);
            if (!ccDetails.getString("number").startsWith("4")) {
                fail(orderId, paymentId, "Invalid Credit Card: " + ccDetails.getString("number"));
            }
             pass(orderId, paymentId, "Payment of " + total + " succeeded for " + name + " CC details: " +
ccDetails.toString());
        } catch (Exception ex) {
             fail(orderId, paymentId, "Unknown error: " + ex.getMessage() + " for payment: " +
cloudEventJson);
        }
    }
```

> Note that the `Thread.sleep(5000);` will cause credit card "processing" to take 5 seconds, to simulate a real world processing time.

Now we need to implement the `pass()` and `fail()` methods referenced above. These methods will send messages to Kafka using our `producer` field.

Add the following code to the `// TODO: Add pass method here` marker:

```
    private void pass(String orderId, String paymentId, String remarks) {

        JsonObject payload = new JsonObject();
        payload.put("orderId", orderId);
        payload.put("paymentId", paymentId);
        payload.put("remarks", remarks);
        payload.put("status", "COMPLETED");
        log.info("Sending payment success: " + payload.toString());
        producer.send(new ProducerRecord<String, String>(paymentsTopic, payload.toString()));
    }
```

Add this code to the `// TODO: Add fail method here` marker:

```
private void fail(String orderId, String paymentId, String remarks) {
    JsonObject payload = new JsonObject();
    payload.put("orderId", orderId);
    payload.put("paymentId", paymentId);
    payload.put("remarks", remarks);
    payload.put("status", "FAILED");
    log.info("Sending payment failure: " + payload.toString());
    producer.send(new ProducerRecord<String, String>(paymentsTopic, payload.toString()));
}
```

Next, add a method that will receive events from Kafka. We will use the MicroProfile reactive messaging API `@Incoming` annotation to do this.

Add this code to the `// TODO: Add consumer method here` marker:

```
@Incoming("orders")
public CompletionStage<Void> onMessage(KafkaMessage<String, String> message)
      throws IOException {

    log.info("Kafka message with value = {} arrived", message.getPayload());
    handleCloudEvent(message.getPayload());
    return message.ack();
}
```

And finally, we need a method to initialize the Kafka producer (the consumer will be initialized automatically via Quarkus Kafka extension). We will use the Quarkus `StartupEvent` Lifecycle listener API, with the `@Observes` annotation to mark this method as one that should run when the app starts:

Add this code to the `// TODO: Add init method here` marker:

```
public void init(@Observes StartupEvent ev) {
    Properties props = new Properties();

    props.put("bootstrap.servers", bootstrapServers);
    props.put("value.serializer", paymentsTopicValueSerializer);
    props.put("key.serializer", paymentsTopicKeySerializer);
    producer = new KafkaProducer<String, String>(props);
}
```

This method will consume Kafka streams from the `orders` topic and call our `handleCloudEvent()` method. Later on we'll delete this method and use Knative Events to handle the incoming stream. But for now we'll use this method to listen to the topic.

Configuring the application

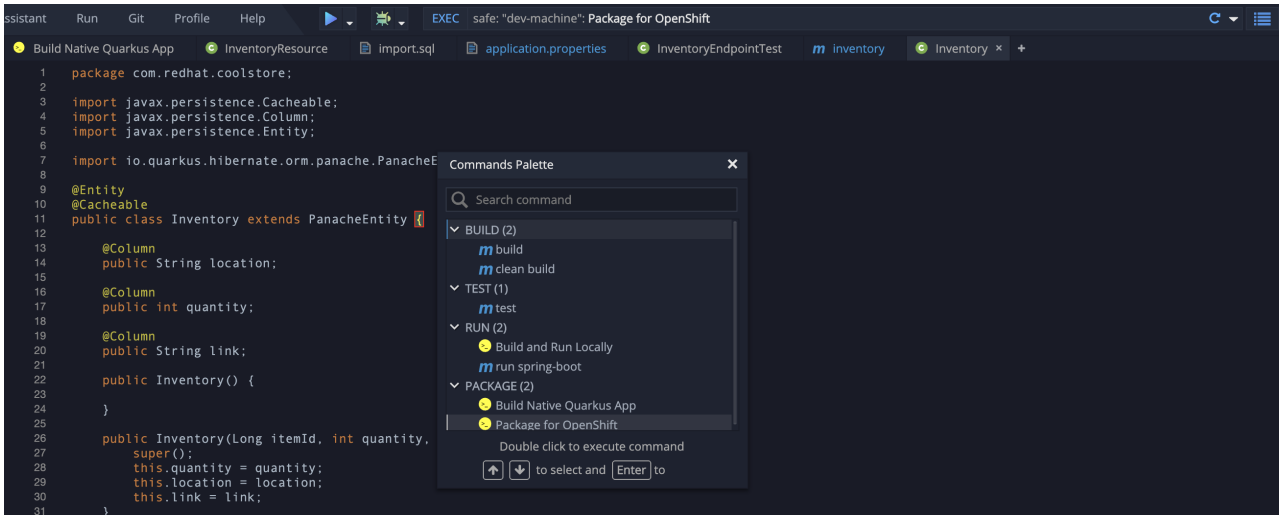
Quarkus and its extensions are configured by an `application.properties` file. Open this file (it is in the `src/main/resources` directory).

Add these values to the file:

```
# Outgoing stream
mp.messaging.outgoing.payments.bootstrap.servers=my-cluster-kafka-bootstrap:9092
mp.messaging.outgoing.payments.connector=smallrye-kafka
mp.messaging.outgoing.payments.topic=payments
mp.messaging.outgoing.payments.value.serializer=org.apache.kafka.common.serialization.StringSerial

mp.messaging.outgoing.payments.key.serializer=org.apache.kafka.common.serialization.StringSerializ


# Incoming stream (unneeded when using Knative events)
mp.messaging.incoming.orders.connector=smallrye-kafka
mp.messaging.incoming.orders.value.deserializer=org.apache.kafka.common.serialization.StringDeseri

mp.messaging.incoming.orders.key.deserializer=org.apache.kafka.common.serialization.StringDeserial

mp.messaging.incoming.orders.bootstrap.servers=my-cluster-kafka-bootstrap:9092
mp.messaging.incoming.orders.group.id=payment-order-service
mp.messaging.incoming.orders.auto.offset.reset=earliest
mp.messaging.incoming.orders.enable.auto.commit=true
mp.messaging.incoming.orders.request.timeout.ms=30000
```

Deploying Payment service to OpenShift

Package the payment application by clicking on **Package for OpenShift** in the Commands Palette`:



Or run the following command in a CodeReady Workspaces *Terminal*:

```
mvn clean package -DskipTests
```

This will build an executable JAR file in the `target/` directory.

To deploy this to OpenShift, define a new build in our project:

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

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

Force update the OpenJDK image tags just in case they haven't been imported yet:

```
oc import-image openjdk18-openshift --all
```

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

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

Deploy it as an OpenShift application after the build is done:

```
oc new-app payment
```

Create the route

```
oc expose svc/payment
```

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

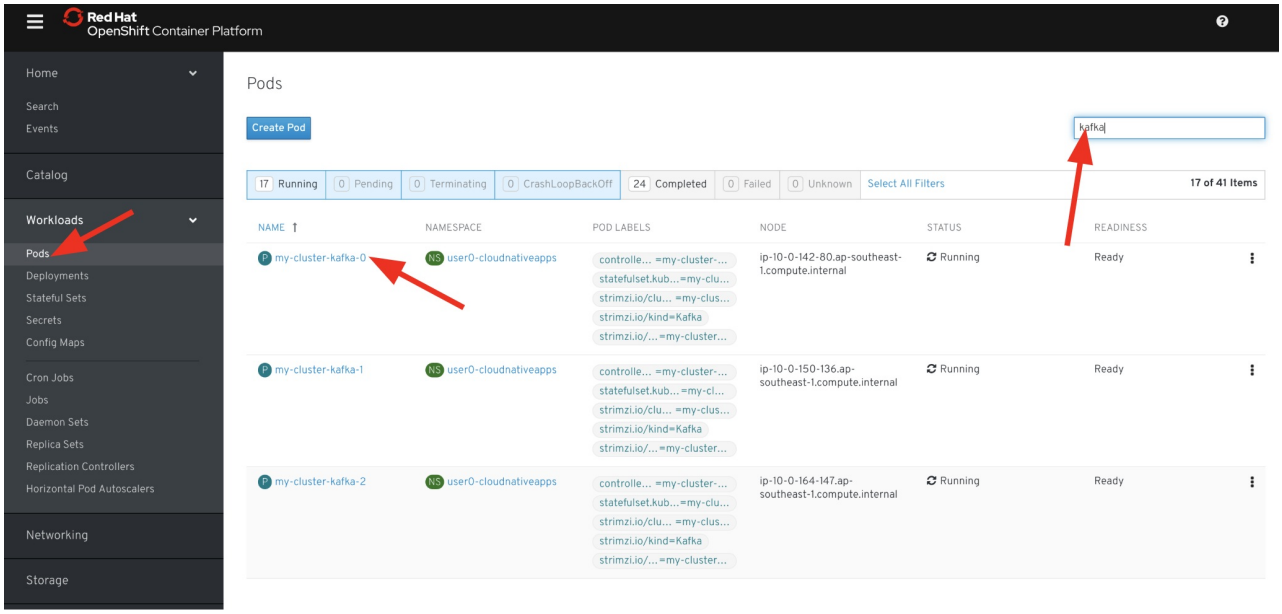
```
oc rollout status -w dc/payment
```

Wait for that command to report `replication controller payment-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.

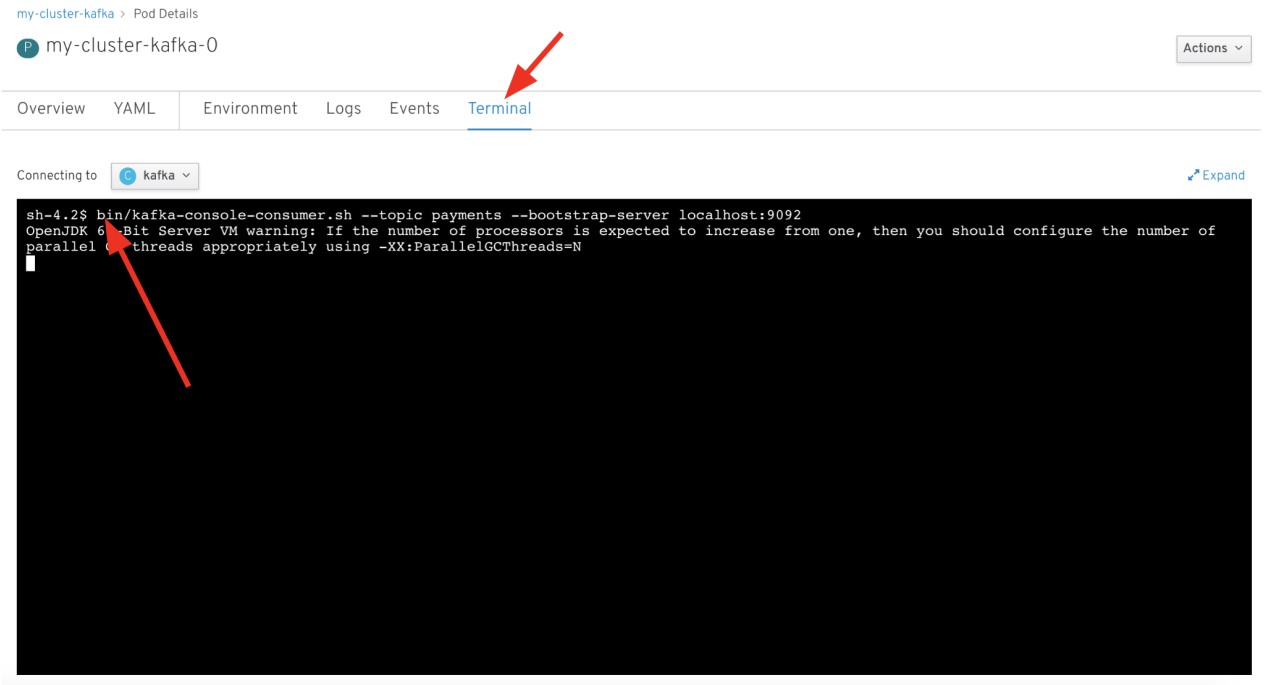
Testing the Application

Go to *Workloads* > *Pods* on the left menu then search `cluster-kafka` pods. Click on the `my-cluster-kafka-0` pod:



We will watch the Kafka topic via a CLI to confirm the messages are being sent/received in Kafka. Click on the *Terminal* tab in OpenShift (not in CodeReady!) then execute the following command:

```
bin/kafka-console-consumer.sh --topic payments --bootstrap-server localhost:9092
```



Keep this tab open to act as a debugger for Kafka messages.

Let's produce a new topic message using `curl` command in CodeReady Workspaces *Terminal*:

First, fetch the URL of our new payment service and store it in an environment variable:

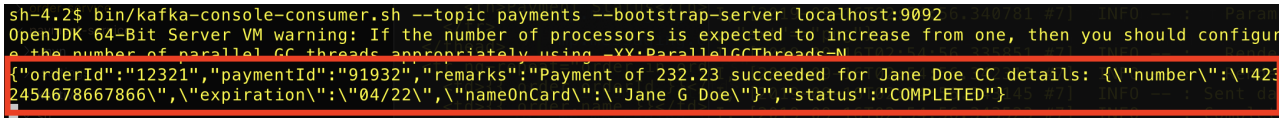
```
export URL="http://$(oc get route | grep payment | awk '{print $2}')"
```

Then execute this to HTTP POST a message to our payment service with an example order:

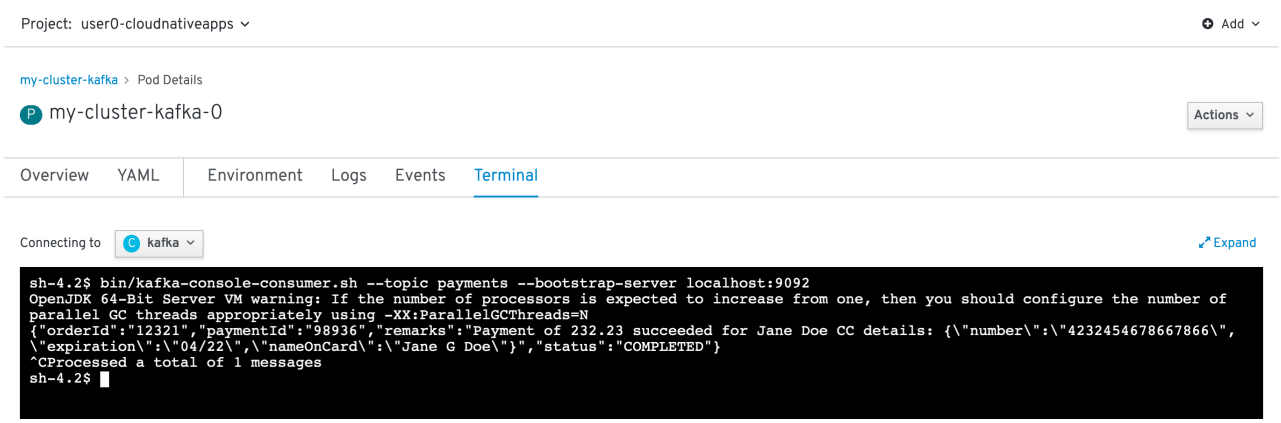
```
curl -i -H 'Content-Type: application/json' -X POST -d'{"orderId": "12321","total": "232.23", "creditCard": {"number": "4232454678667866","expiration": "04/22","nameOnCard": "Jane G Doe"}, "billingAddress": "123 Anystreet, Pueblo, CO 32213", "name": "Jane Doe"}' $URL
```

The payment service will recieve this *order* and produce a *payment* result on the Kafka *payment* topic. You will see the following result in `Pod Terminal` :

```
{"orderId":"12321","paymentId":"25658","remarks":"Payment of 232.23 succeeded for Jane Doe CC details: {\"number\":\"4232454678667866\",\"expiration\":\"04/22\",\"nameOnCard\":\"Jane G Doe\"}","status":"COMPLETED"}
```



Before moving to the next step, stop the Kafka consumer console via `CTRL + C` in Terminal:



3. Adding Kafka Client to Cart Service

By now we have added several microservices to operate on our retail shopping data. Quite often, other services or functions would need the data we are working with. e.g. once a user checks out, there are other services like an *Order Service* and our *Payment Service* that will need this information, and would most likely want to process further. So we will integrate our Cart service with Kafka so that it can send an order message when a shopper checks out.

To do that open the `cart-service/src/main/java/com/redhat/cloudnative/CartResource.java` file in CodeReady.

Adding Maven Dependencies using Quarkus Extensions

Execute the following command via CodeReady Workspaces *Terminal*:

```
cd /projects/cloud-native-workshop-v2m4-labs/cart-service/
```

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

This will add the Kafka extension and APIs to our Cart service app.

Like our Payment service, add this code to the `// TODO: Add annotation of orders messaging configuration here` marker inside the `CartResource` class inside the `com.redhat.cloudnative` package:

```
@ConfigProperty(name = "mp.messaging.outgoing.orders.bootstrap.servers")
public String bootstrapServers;

@ConfigProperty(name = "mp.messaging.outgoing.orders.topic")
public String ordersTopic;

@ConfigProperty(name = "mp.messaging.outgoing.orders.value.serializer")
public String ordersTopicValueSerializer;

@ConfigProperty(name = "mp.messaging.outgoing.orders.key.serializer")
public String ordersTopicKeySerializer;

private Producer<String, String> producer;
```

Next, un-comment (or add if they are missing) the following `import` statements:

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
```

The init method as it denotes creates the Kafka configuration, we have externalized this configuration and injected the variables as properties on the class.

Replace the empty `init()` method with this code:

```
public void init(@Observes StartupEvent ev) {
    Properties props = new Properties();

    props.put("bootstrap.servers", bootstrapServers);
    props.put("value.serializer", ordersTopicValueSerializer);
    props.put("key.serializer", ordersTopicKeySerializer);
    producer = new KafkaProducer<String, String>(props);
}
```

The `sendOrder()` method is quite simple, it takes the Order POJO as a param and serializes that into JSON to send over the Kafka topic.

Replace the empty `sendOrder()` method with this code:

```
private void sendOrder(Order order, String cartId) {
    order.setTotal(shoppingCartService.getShoppingCart(cartId).getCartTotal() + "");
    producer.send(new ProducerRecord<String, String>(ordersTopic, Json.encode(order)));
    log.info("Sent message: " + Json.encode(order));
}
```

Now that we have those methods, lets add a call to our `sendOrder()` method when we are checking out. Replace the code for `checkout()` with this code:

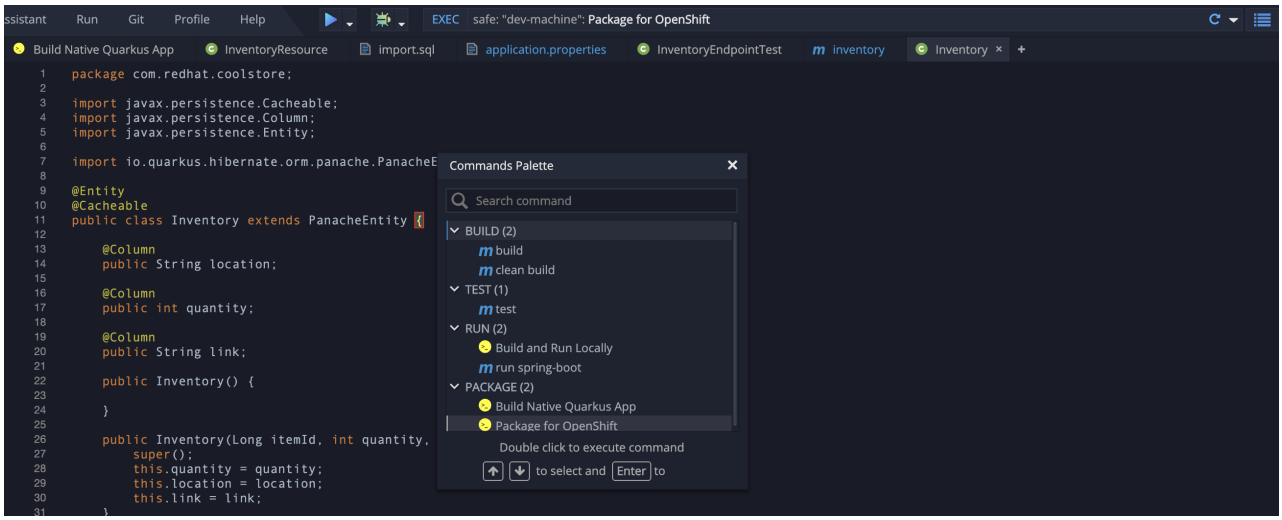
```
@POST
@Path("/checkout/{cartId}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@Operation(summary = "checkout")
public ShoppingCart checkout(@PathParam("cartId") String cartId, Order order) {
   sendOrder(order, cartId);
   return shoppingCartService.checkout(cartId);
}
```

Almost there! Next let's add the configuration to our `application.properties` file (in the `src/main/resources` of the `cart-service` project):

```
mp.messaging.outgoing.orders.bootstrap.servers=my-cluster-kafka-bootstrap:9092
mp.messaging.outgoing.orders.connector=smallrye-kafka
mp.messaging.outgoing.orders.topic=orders
mp.messaging.outgoing.orders.value.serializer=org.apache.kafka.common.serialization.StringSerializer

mp.messaging.outgoing.orders.key.serializer=org.apache.kafka.common.serialization.StringSerializer
```

Re-Deploying Cart service to OpenShift

Package the cart application via clicking on `Package for OpenShift` in `Commands Palette` :

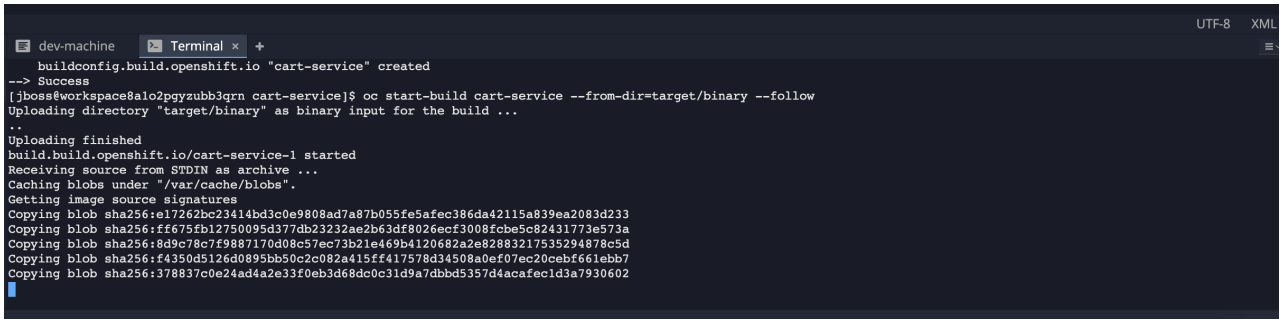


Or run the following maven plugin in CodeReady Workspaces *Terminal*:

```
mvn clean package -DskipTests
```

Rebuild a container image based the cart artifact that we just packaged, which will take about minutes to complete:

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



The cart service will be redeployed automatically via OpenShift Deployment triggers after it completes to build.

4. Adding Kafka Client to Order Service

Like the `payments` service, our `order` service will listen for orders being placed, but will not process payments - instead the order service will merely record the orders and their states for eventual display in the UI. Let's add this capability to the order service.

Adding Maven Dependencies using Quarkus Extensions

Execute the following command via CodeReady Workspaces Terminal:

```
cd /projects/cloud-native-workshop-v2m4-labs/order-service/
```

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

This command generates a Maven project, importing the Kafka extensions for Quarkus applications and provides all the necessary capabilities to integrate with the Kafka clusters and subscribe `payments` topic and `orders` topic. Let's confirm your `pom.xml` as below:

Creating Orders and Payments Consumer in Order Service

In the `order-service` project, Create a new Java class, `KafkaOrders.java` in `src/main/java/com/redhat/cloudnative` to consume messages from the Kafka `orders` and `payments` topic. Copy the following entire code into `KafkaOrders.java` .

```
package com.redhat.cloudnative;

import io.smallrye.reactive.messaging.kafka.KafkaMessage;
import org.eclipse.microprofile.reactive.messaging.Incoming;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import javax.enterprise.context.ApplicationScoped;

import java.io.IOException;
import java.util.concurrent.CompletionStage;

import javax.inject.Inject;
import io.vertx.core.json.JsonObject;

@ApplicationScoped
public class KafkaOrders {

    private static final Logger LOG = LoggerFactory.getLogger(KafkaOrders.class);

    @Inject
    OrderService orderService;

    @Incoming("orders")
    public CompletionStage<Void> onMessage(KafkaMessage<String, String> message)
            throws IOException {

        LOG.info("Kafka order message with value = {} arrived", message.getPayload());

        JsonObject orders = new JsonObject(message.getPayload());
        Order order = new Order();
        order.setOrderId(orders.getString("orderId"));
        order.setName(orders.getString("name"));
        order.setTotal(orders.getString("total"));
        order.setCcNumber(orders.getJsonObject("creditCard").getString("number"));
        order.setCcExp(orders.getJsonObject("creditCard").getString("expiration"));
        order.setBillingAddress(orders.getString("billingAddress"));
        order.setStatus("PROCESSING");
        orderService.add(order);

        return message.ack();
    }

    @Incoming("payments")
    public CompletionStage<Void> onMessagePayments(KafkaMessage<String, String> message)
            throws IOException {

        LOG.info("Kafka payment message with value = {} arrived", message.getPayload());

        JsonObject payments = new JsonObject(message.getPayload());
        orderService.updateStatus(payments.getString("orderId"), payments.getString("status"));

        return message.ack();
    }
```

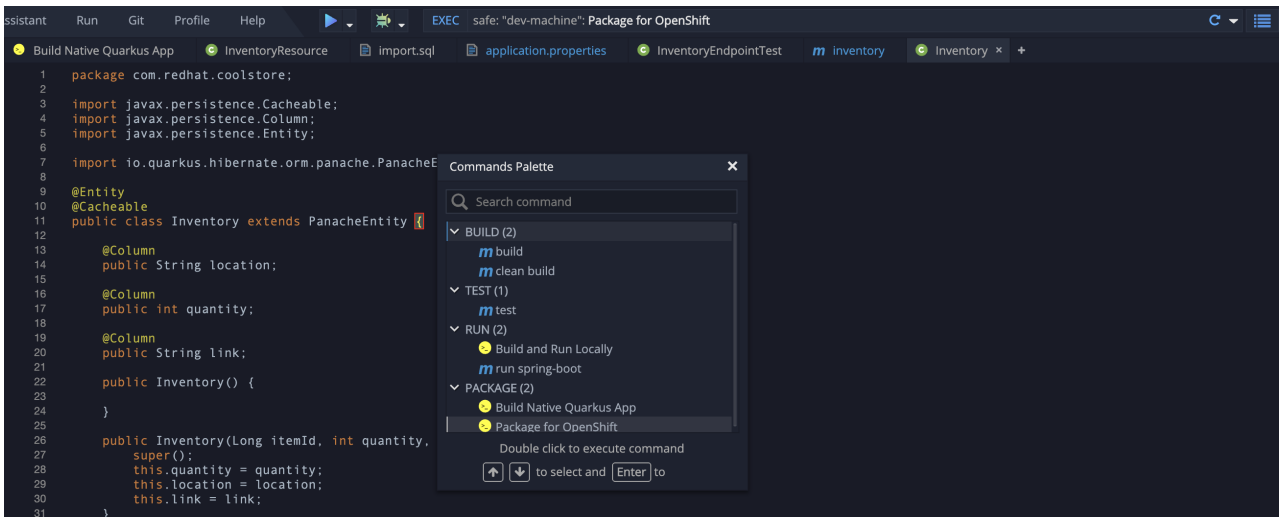
```
}
```

Almost there; Next lets add the configuration to our `src/main/resources/application.properties` file in the `order-service` project:

```
# Incoming payment topic messages
mp.messaging.incoming.payments.connector=smallrye-kafka
mp.messaging.incoming.payments.value.deserializer=org.apache.kafka.common.serialization.StringDe

mp.messaging.incoming.payments.key.deserializer=org.apache.kafka.common.serialization.StringDese

mp.messaging.incoming.payments.bootstrap.servers=my-cluster-kafka-bootstrap:9092
mp.messaging.incoming.payments.group.id=order-service
mp.messaging.incoming.payments.auto.offset.reset=earliest
mp.messaging.incoming.payments.enable.auto.commit=true
mp.messaging.incoming.payments.request.timeout.ms=30000

# Enable CORS requests from browsers
quarkus.http.cors=true

# Incoming order topic messages
mp.messaging.incoming.orders.connector=smallrye-kafka
mp.messaging.incoming.orders.value.deserializer=org.apache.kafka.common.serialization.StringDeseri

mp.messaging.incoming.orders.key.deserializer=org.apache.kafka.common.serialization.StringDeserial

mp.messaging.incoming.orders.bootstrap.servers=my-cluster-kafka-bootstrap:9092
mp.messaging.incoming.orders.group.id=order-service
mp.messaging.incoming.orders.auto.offset.reset=earliest
mp.messaging.incoming.orders.enable.auto.commit=true
mp.messaging.incoming.orders.request.timeout.ms=30000
```

Re-Deploying Order service to OpenShift

Package the order application via clicking on `Package for OpenShift` in `Commands Palette`:



Or run the following maven plugin in CodeReady Workspaces *Terminal*:

```
mvn clean package -DskipTests
```

Rebuild a container image based the cart artifact that we just packaged, which will take about minutes to complete:

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

The order service will be redeployed automatically via OpenShift Deployment triggers after it completes to build.

Let's confirm if the all services works correctly using `Kafka` messaging via coolstore GUI test.

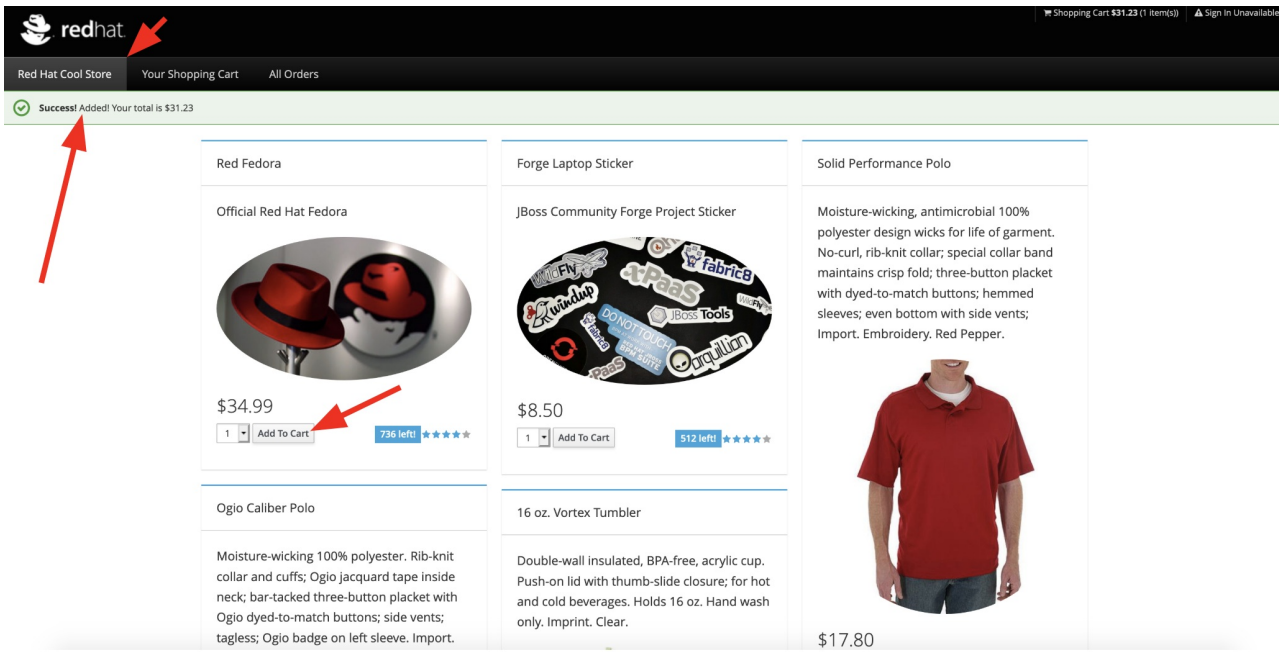
5. End to End Functional Testing

Let's go shopping! Open the Web UI in your browser. To get the URL to the Web UI, run this command in CodeReady *Terminal*:

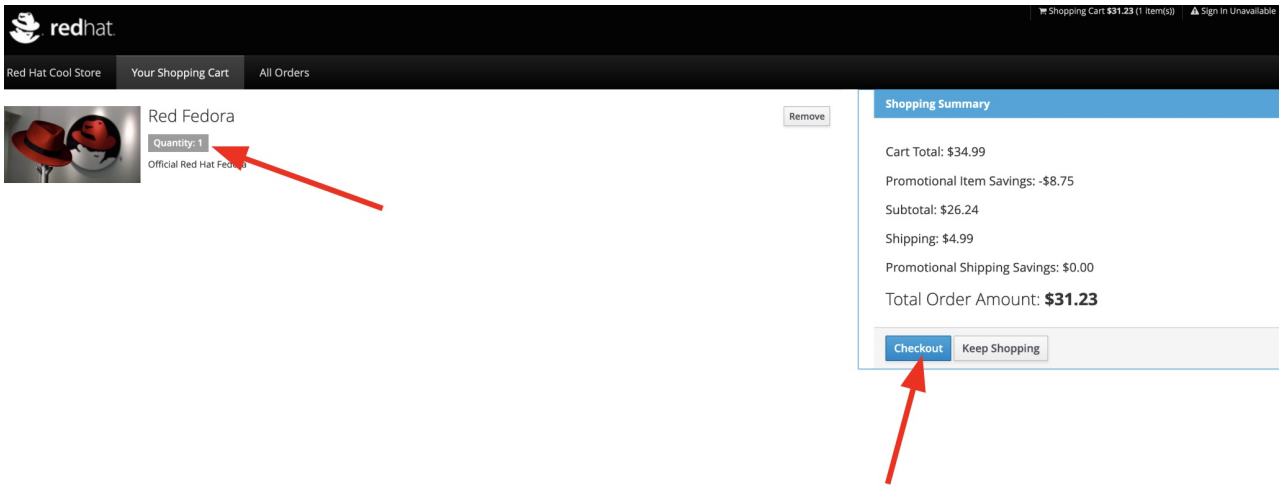
```
oc get route | grep coolstore-ui | awk '{print $2}'
```

Add some cool items to your shopping cart in the following shopping scenarios:

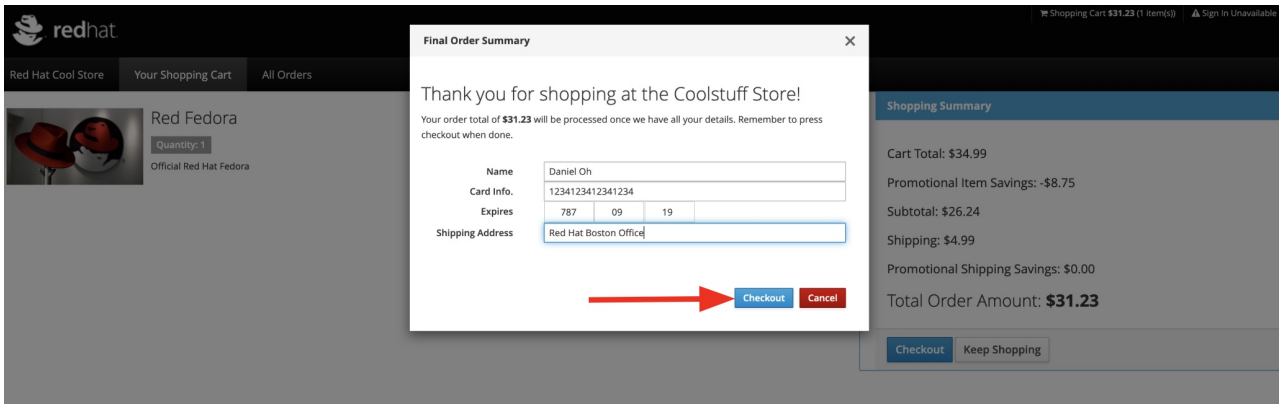
1) Add a *Red Hat Fedora* to your cart by click on **Add to Cart**. You will see the `Success! Added!` message under the top munu.



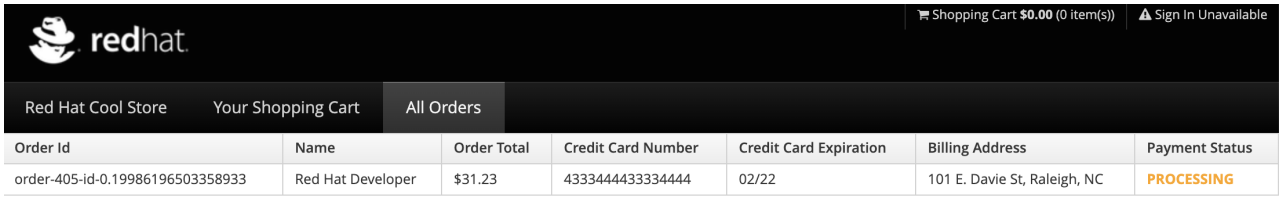
2) Go to the **Your Shopping Cart** tab and click on the **Checkout** button . Input the credit card information. The Card Info should be 16 digits and begin with the digit `4` . For example `4123987754646678` .



3) Input your Credit Card information to pay for the items:

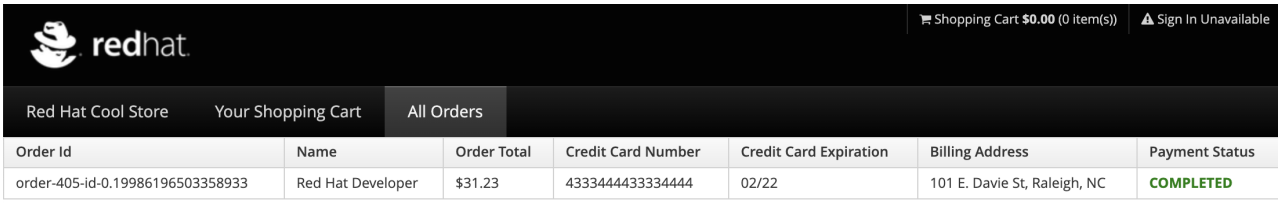


4) Confirm the *Payment Status* of the your shopping items in the **All Orders** tab. It should be `Processing` .



5) After a few moments, reload the **All Orders** page to confirm that the Payment Status changed to `COMPLETED` or `FAILED` .

> `Note` : If the status is still `Processing` , the order service is processing incoming Kafka messages and storing them in MongoDB. Please reload the page a few times more.



Summary

In this scenario we developed an *Event-Driven/Reactive* cloud-native appliction to deal with data streams from the shopping cart service to the order service and payment service using *Apache Kafka*).

We also used Quarkus and its *Kafka extension* to integrate the app with Kafka. `AMQ Streams` , a fully supported Kafka solution from Red Hat, enables you to create Apache Kafka clusters very easily via OpenShift developer catalog.

In the end, we now have message-driven microservices for implementing reactive systems, where all the components interact using asynchronous messages passing. Most importantly, **Quarkus** is perfectly suited to implement event-driven microservices and reactive systems. Congratulations!

Evolving Serverless Service

Lab3 - Evolving services with Serverless

In our cloud-native application architecture, We now have multiple microservices in a *reactive* system. However, it's not necessary our applications and services be up and running 24 hours day. They only need to be running *on-demand*, when something needs to use the service. This is one of the reasons why *serverless* architectures have gained popularity.

- **Serverless** is often used interchangeably with the term *FaaS* (Functions-as-a-Service). But serverless doesn't mean that there is no server. In fact, there *are* servers - a public cloud provider provides the servers that deploy, run, and manage your application.

- **Serverless computing** is an emerging category that represents a shift in the way developers build and deliver software systems. Abstracting application infrastructure away from the code can greatly simplify the development process while introducing new cost and efficiency benefits. Serverless computing and FaaS will play an important role in helping to define the next era of enterprise IT, along with cloud-native services and the hybrid cloud.

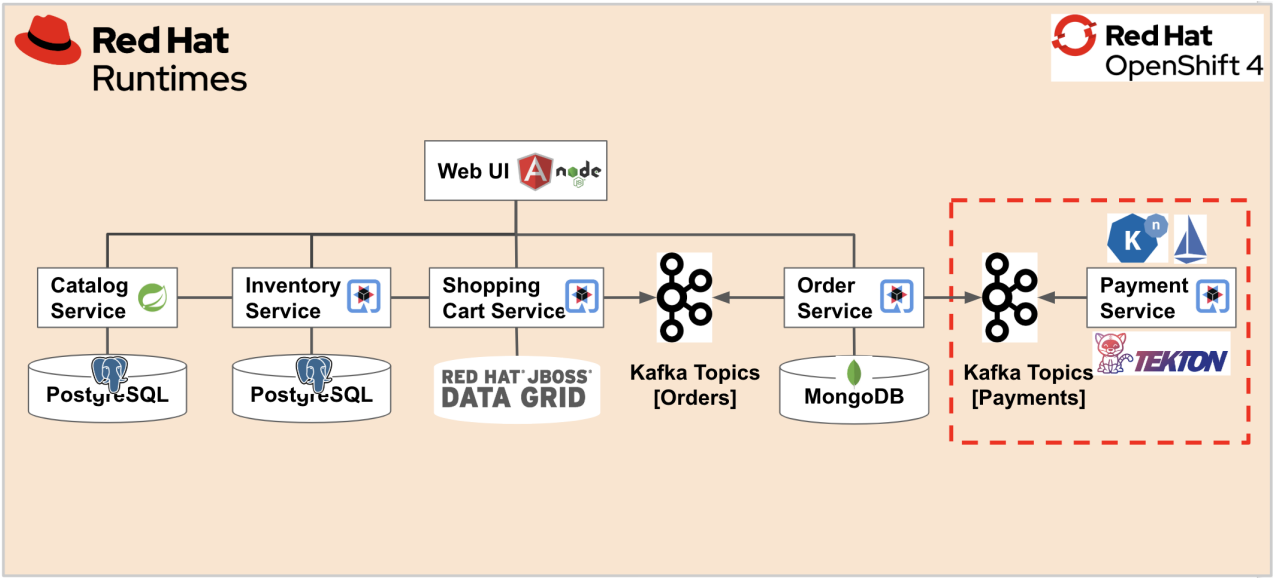
- **Serverless platforms** provide APIs that allow users to run code snippets (functions, also called *actions*) and return the results of each function. Serverless platforms also provide endpoints to allow the developer to retrieve function results. These endpoints can be used as inputs for other functions, thereby providing a sequence (or chain) of related functions.

The severless application enables DevOps teams to enjoy benefits like:

- Optimizing computing resources(i.e CPU, Memory)
- Autoscaling
- Simplifying CI/CD pipeline

Goals of this lab

The goal is to develop serverless applications on **Red Hat Runtimes** and deploy them on **OpenShift 4** using **OpenShift Serverless** (Based on the Knative project) with a cloud-native, continuous integration and delivery (CI/CD) Pipelines using `Tekton`. After this lab, you should end up with something like:



In this lab, we'll deploy the Payment Service as a Quarkus-based serverless application using Knative Serving, Istio, and Tekton Pipelines.

The Knative Kafka Event *source* enables *Knative Eventing* integration with Apache Kafka. When a message is produced in Apache Kafka, the Apache Kafka Event Source will consume the produced message and post that message to the corresponding event *sink*.

What is Knative?

Knative extends Kubernetes to provide components for building, deploying, and managing serverless applications. Build serverless applications that run wherever you need them—on-premise or on any cloud—with Knative and OpenShift.

Knative components leverage best practices from real-world Kubernetes deployments:

- Servinguses Kubernetes and Istio to rapidly deploy, network, and automatically scale serverless workloads.
- Eventing is common infrastructure for consuming and producing events to stimulate applications.

What is a Service Mesh?

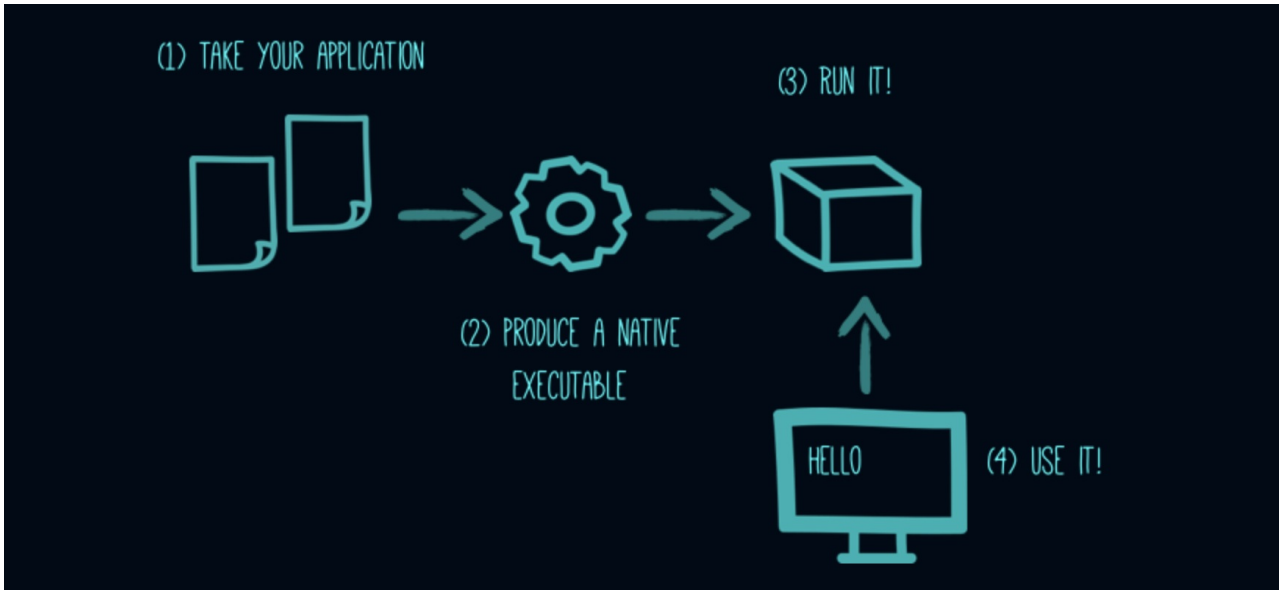
Service Mesh provides traffic monitoring, access control, discovery, security, resiliency, and other useful things to a group of services. Istio does all that, but it doesn't require any changes to the code of any of those services. To make the magic happen, Istio deploys a proxy (called a *sidecar*) next to each service. All of the traffic meant for a service goes to the proxy, which uses policies to decide how, when, or if that traffic should go on to the service. Istio also enables sophisticated DevOps techniques such as canary deployments, circuit breakers, fault injection, and more.

Istio also moves operational aspects away from code development and into the domain of operations. Why should a developer be burdened with circuit breakers and fault injections and should they respond to them? Yes, but for handling and/or creating them? Take that out of your code and let your code focus on the underlying business domain.

1. Building a Native Executable

Let's now produce a native executable for an example Quarkus application. It improves the startup time of the application, and produces a minimal disk and memory footprint. The executable would have everything to run the application including the `JVM` (shrunk to be just enough to run the application), and the application. This is accomplished using GraalVM.

`GraalVM` is a universal virtual machine for compiling and running applications written in JavaScript, Python, Ruby, R, JVM-based languages like Java, Scala, Groovy, Kotlin, Clojure, and LLVM-based languages such as C and C++. It includes ahead-of-time compilation, aggressive dead code elimination, and optimal packaging as native binaries that moves a lot of startup logic to build-time, thereby reducing startup time and memory resource requirements significantly.



`GraalVM` is already installed for you. Inspect the value of the `GRAALVM_HOME` variable in the CodeReady Workspaces *Terminal* with:

```
echo $GRAALVM_HOME
```

In this step, we will learn how to compile the application to a native executable and run the native image on local machine.

Compiling a native image takes longer than a regular JAR file (bytecode) compilation. However, this compilation time is only incurred once, as opposed to every time the application starts, which is the case with other approaches for building and executing JARs.

Build Native Image and Run it Locally

Let's find out why Quarkus calls itself *SuperSonic Subatomic Subatomic Java*. Let's build a sample app. In CodeReady Terminal, run this command:

```
mkdir /tmp/hello && cd /tmp/hello && \
mvn io.quarkus:quarkus-maven-plugin:0.21.2:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=getting-started \
    -DclassName="org.acme.quickstart.GreetingResource" \
    -Dpath="/hello"
```

This will create a simple Quarkus app in the `/tmp/hello` directory.

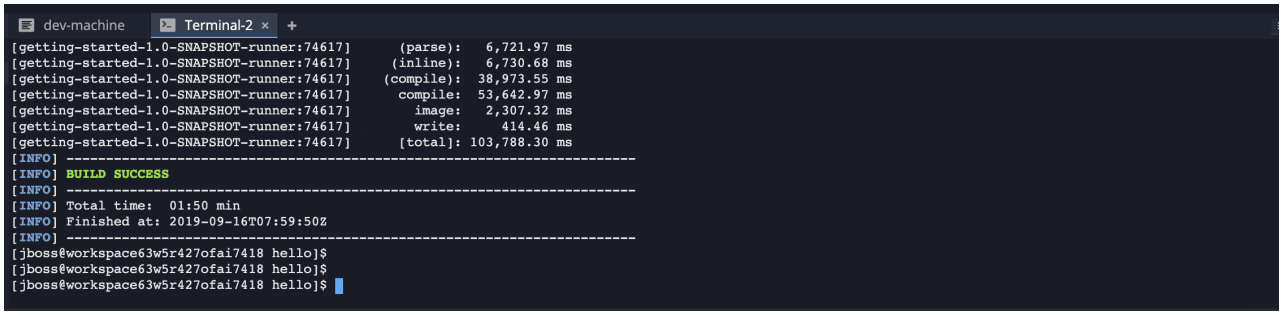
Next, create a `native executable` with this command:

`mvn -f /tmp/hello clean package -Pnative -DskipTests`

> This may take a minute or two to run. One of the benefits of Quarkus is amazingly fast startup time, at the expense of a longer build time to optimize and remove dead code, process annotations, etc. This is only incurred once, at build time rather than *every* startup!

> NOTE: You can safely ignore any warnings like `Warning: RecomputeFieldValue.FieldOffset automatic substitution failed`. These are harmless and will be removed in future releases of Quarkus.

> NOTE: Since we are on Linux in this environment, and the OS that will eventually run our application is also Linux, we can use our local OS to build the native Quarkus app. If you need to build native Linux binaries when on other OS's like Windows or Mac OS X, you'll need to have Docker installed and then use `mvn clean package -Pnative -Dnative-image.docker-build=true -DskipTests=true`.



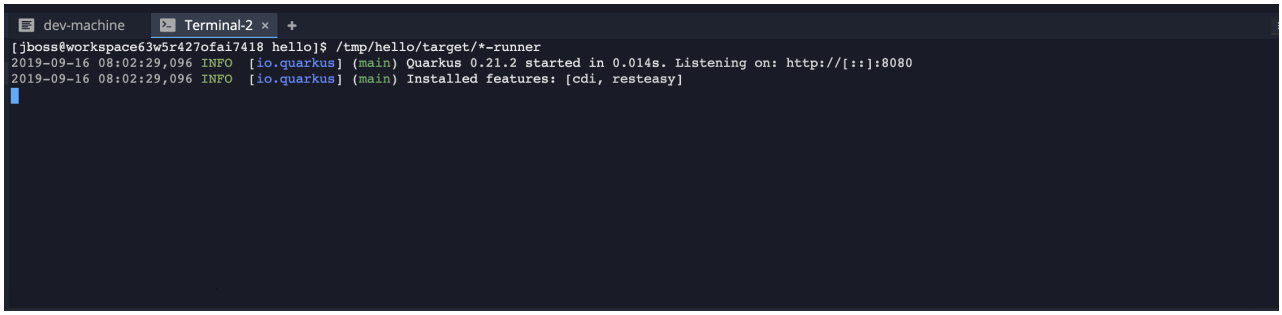
Since our environment here is Linux, you can just run it. In the CodeReady Workspaces Terminal, run:

`/tmp/hello/target/*-runner`

Notice the amazingly fast startup time:

```
2019-09-16 08:02:29,096 INFO  [io.quarkus] (main) Quarkus 0.21.2 started in 0.014s. Listening on: http://[::]:8080
2019-09-16 08:02:29,096 INFO  [io.quarkus] (main) Installed features: [cdi, resteasy]
```

That's *14 milliseconds* to start up.

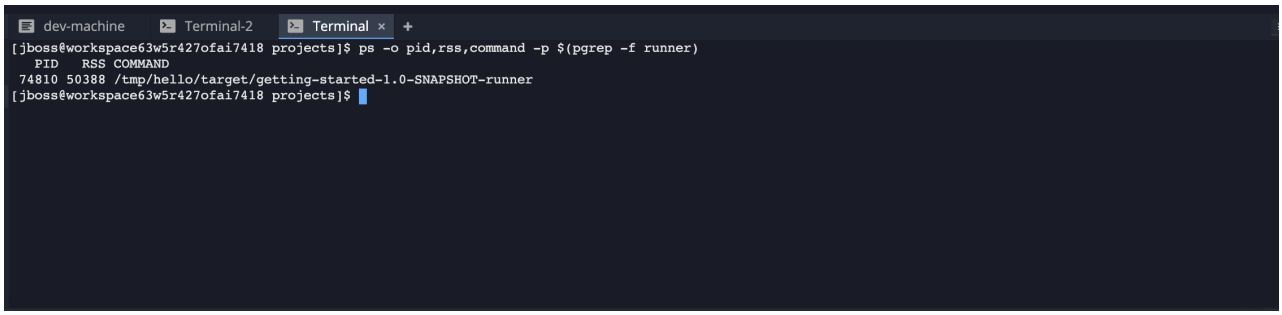


And extremely low memory usage as reported by the Linux `ps` utility. While the app is running, open another Terminal (click the `+` button on the terminal tabs line) and run:

```
ps -o pid,rss,command -p $(pgrep -f runner)
```

You should see something like:

```
  PID   RSS COMMAND
74810 50388 /tmp/hello/target/getting-started-1.0-SNAPSHOT-runner
```



This shows that our process is taking around `50 MB` of memory (Resident Set Size, or RSS). Pretty compact!

> NOTE: The RSS and memory usage of any app, including Quarkus, will vary depending your specific environment, and will rise as the application experiences load.

Make sure the app works. In a new CodeReady Workspaces Terminal run:

```
curl -i http://localhost:8080/hello; echo
```

You should see the return:

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: text/plain;charset=UTF-8
Content-Length: 5
Date: Mon, 16 Sep 2019 03:35:40 GMT

hello
```

`Congratuations!` You've now built a Java application as a native executable JAR and a Linux native binary. We'll explore the benefits of native binaries later in when we start deploying to Kubernetes.

Before moving to the next step, go to the first Terminal tab and press `CTRL+C` to stop our native app (or close the Terminal window).

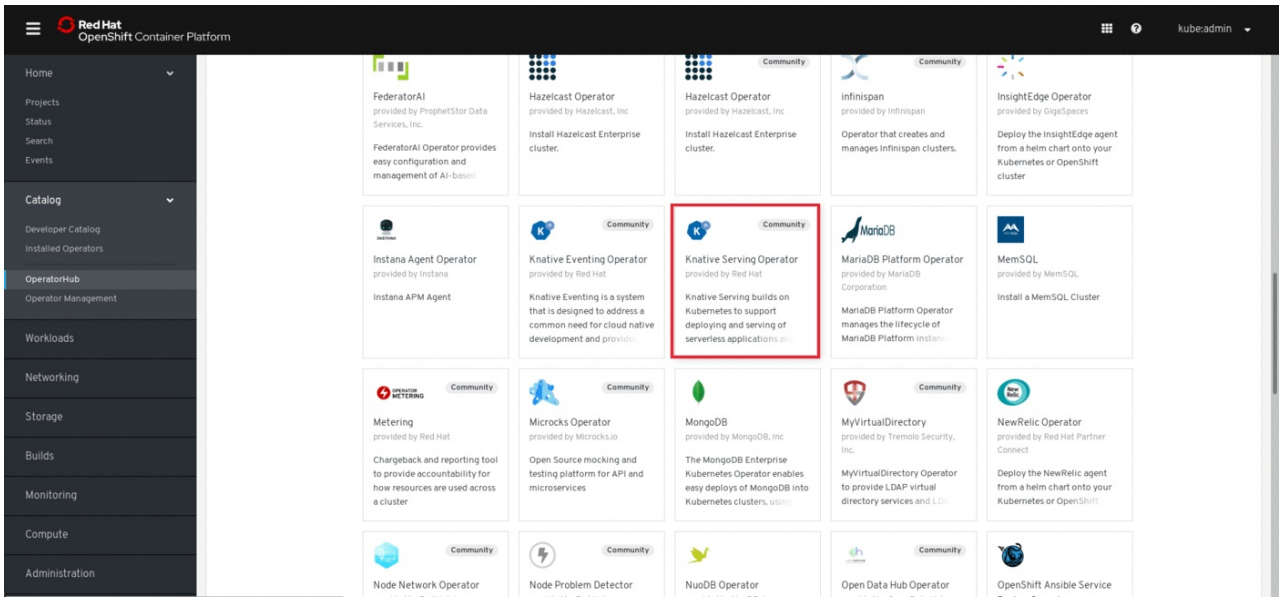
2. Delete old payment service

Knative Serving builds on Kubernetes and Istio to support deploying and serving of serverless applications and functions. *Serving* is easy to get started with and scales to support advanced scenarios.

The Knative Serving project provides middleware primitives that enable:

- Rapid deployment of serverless containers
- Automatic scaling up and down to zero
- Routing and network programming for Istio components
- Point-in-time snapshots of deployed code and configurations

In the lab, *Knative Serving* is already installed on your OpenShift cluster but if you want to install Knative Serving on your own OpenShift cluster, you can play with Installing the Knative Serving Operator as below:



First, we need to delete existing `BuildConfig` as it is based an excutable Jar that we deployed it in lab 2.

```
oc delete bc/payment
```

We also will delete our existing payment *deployment* and *route* since Knative will handle deploying the payment service and routing traffic to its managed pod when needed. Delete the existing payment deployment and its associated route and service with:

```
oc delete dc/payment route/payment svc/payment
```

3. Enable Knative Eventing integration with Apache Kafka Event

Knative Eventing is a system that is designed to address a common need for cloud native development and provides composable primitives to enable `late-binding` event sources and event consumers with below goals:

- Services are loosely coupled during development and deployed independently.
- Producer can generate events before a consumer is listening, and a consumer can express an interest in an event or class of events that is not yet being produced.
- Services can be connected to create new applications without modifying producer or consumer, and with the ability to select a specific subset of events from a particular producer.

The *Apache Kafka Event source* enables Knative Eventing integration with Apache Kafka. When a message is produced to Apache Kafka, the Event Source will consume the produced message and post that message to the corresponding event sink.

Remove direct Knative integration code

Currently our Payment service directly binds to Kafka to listen for events. Now that we have Knative eventing integration, we no longer need this code. Open the `PaymentResource.java` file (in `payment-service/src/main/java/com/redhat/cloudnative` directory).

Delete (or comment out) the `onMessage()` method:

```
//    @Incoming("orders")
//    public CompletionStage<Void> onMessage(KafkaMessage<String, String> message)
//            throws IOException {
//
//        log.info("Kafka message with value = {} arrived", message.getPayload());
//        handleCloudEvent(message.getPayload());
//        return message.ack();
//    }
```

And delete the configuration for the incoming stream. In `application.properties`, delete (or comment out) the following lines for the *Incoming* stream:

```
# Incoming stream (unneeded when using Knative events)
# mp.messaging.incoming.orders.connector=smallrye-kafka
#
mp.messaging.incoming.orders.value.deserializer=org.apache.kafka.common.serialization.StringDeseri

#
mp.messaging.incoming.orders.key.deserializer=org.apache.kafka.common.serialization.StringDeserial

# mp.messaging.incoming.orders.bootstrap.servers=my-cluster-kafka-bootstrap:9092
# mp.messaging.incoming.orders.group.id=payment-order-service
# mp.messaging.incoming.orders.auto.offset.reset=earliest
# mp.messaging.incoming.orders.enable.auto.commit=true
# mp.messaging.incoming.orders.request.timeout.ms=30000
```

Rebuild and re-deploy new Payment service

Open the `payment-service/pom.xml` in the editor, then in the CodeReady command palette, Choose `Build Native Quarkus App`. This will re-build our native executable in the `target/` directory.

Or you can run the commands directly:

```
cd /projects/cloud-native-workshop-v2m4-labs/payment-service/
```

```
mvn clean package -Pnative -DskipTests
```

This will execute `mvn clean package -Pnative` behind the scenes. The `-Pnative` argument selects the native maven profile which invokes the Graal compiler.

We've deleted our old build configuration that took a JAR file. We need a new build configuration that can take our new native compiled Quarkus app. Create a new build config with this command:

```
oc new-build quay.io/quarkus/ubi-quarkus-native-binary-s2i:19.2.0 --binary --name=payment -l app=payment
```

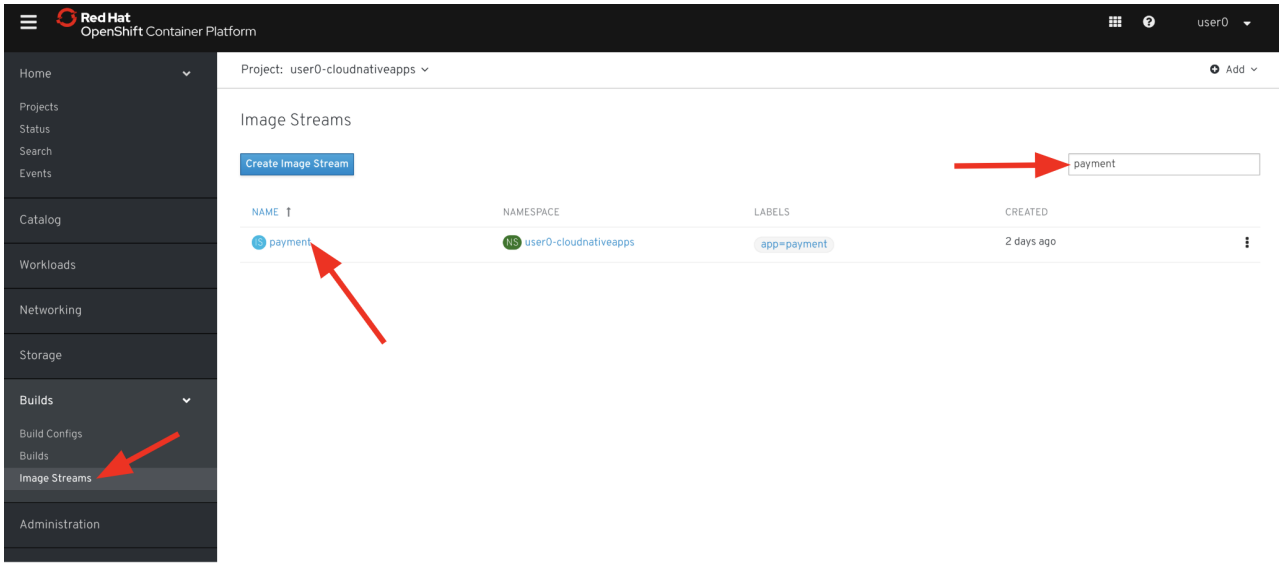
You should get a `--> Success message` at the end.

Mext, start and watch the build, which will take about 3-4 minutes to complete:

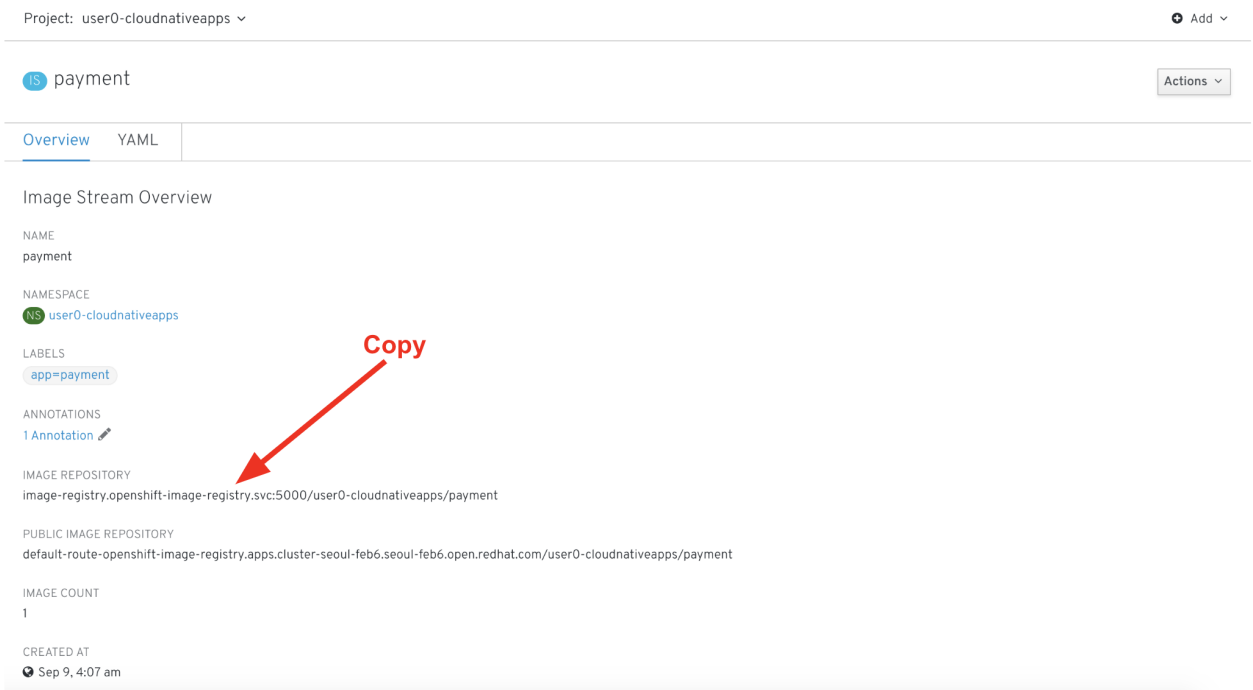
```
oc start-build payment --from-file target/*-runner --follow
```

This step will combine the native binary with a base OS image, create a new container image, and push it to an internal image registry.

Once that's done, go to *Builds* > *Image Streams* on the left menu then input `payment` to show the payment imagestream. Click on `payment` imagestream:



In the *Overview* tab, copy the `IMAGE REPOSITORY` value shown and then open the `payment-service/knative/knative-serving-service.yaml` file and update the `image:` line with this value.



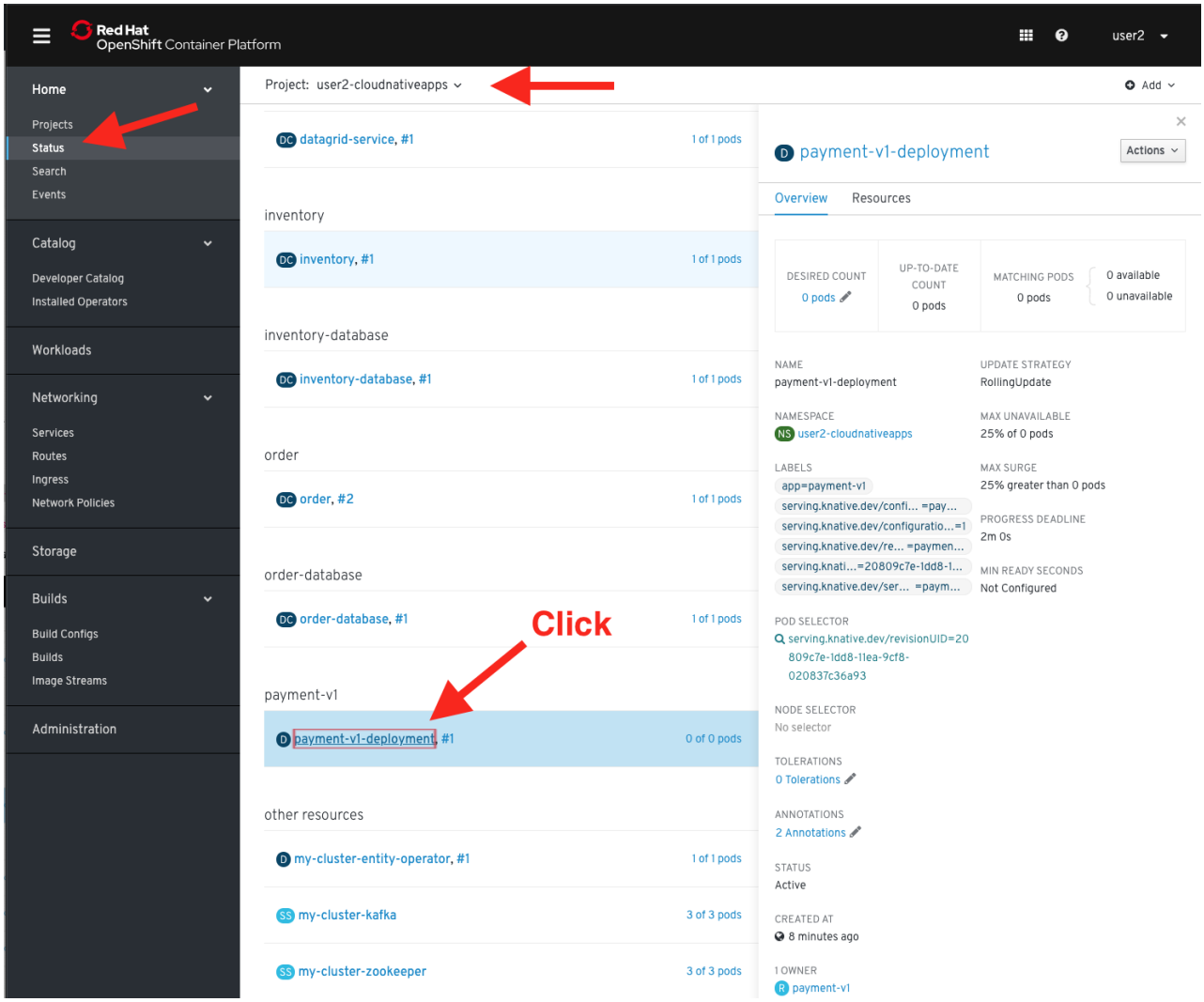

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: payment
spec:
  template:
    metadata:
      name: payment
      annotations:
        # disable istio-proxy injection
        sidecar.istio.io/inject: "false"
    spec:
      containers:
        # Replace Project name userXX-cloudnativeapps with project in which payment is deployed
      - image: YOUR_IMAGE_SERVICE_URL:latest
```

The service can then be deployed using the following command via CodeReady Workspaces Terminal:

`oc apply -f /projects/cloud-native-workshop-v2m4-labs/payment-service/knative/knative-serving-service.yaml`

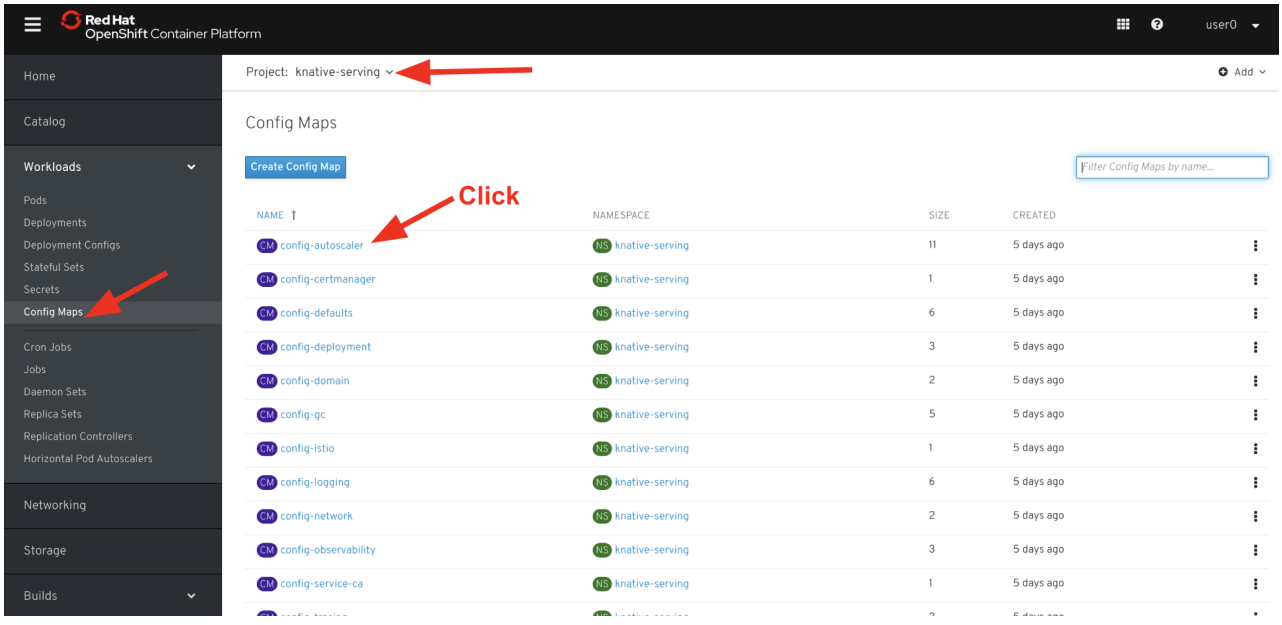
After successful creation of the service we should see a Kubernetes Deployment named similar to `payment-v1-deployment` available.

Go to *Home* > *Status* on the left menu and click on **payment-v1-deployment**. You will confirm 1 pod is *available*.



In the lab environment, *Knative Serving* and *Knative Eventing* components are already installed. Select the `knative-serving` project in the project drop-down selector, then go to `Workloads > Config Maps` in the left menu.

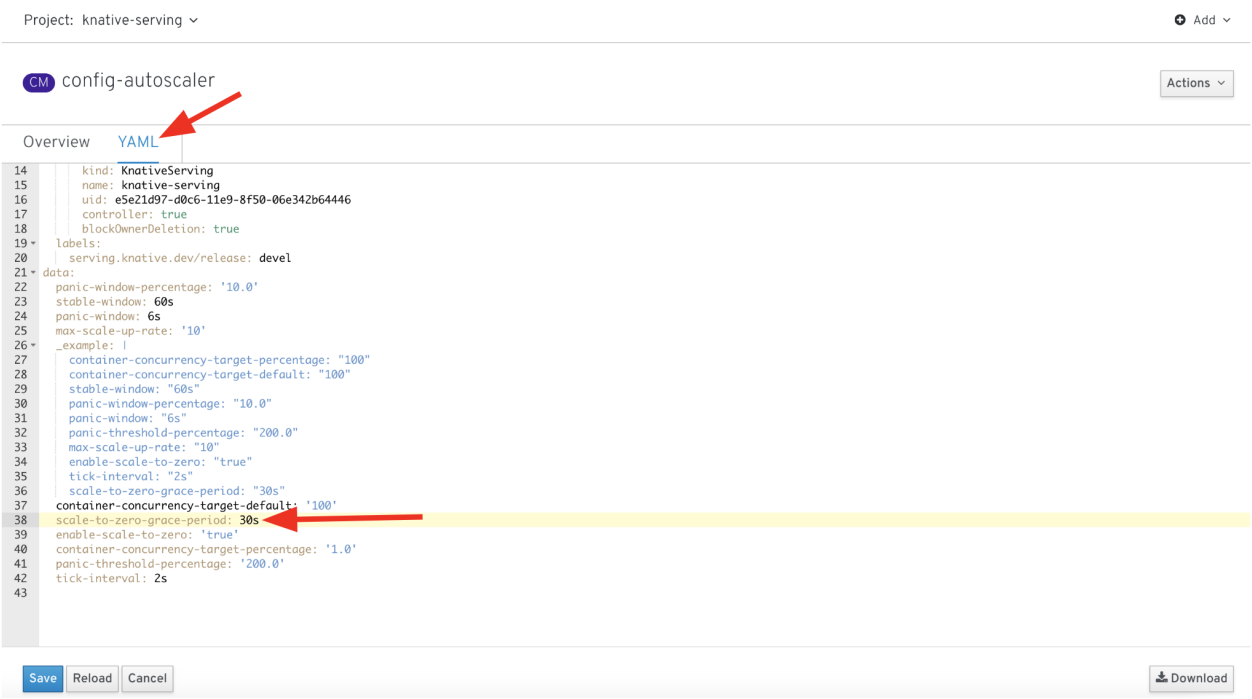
Click on **config-autoscaler**.



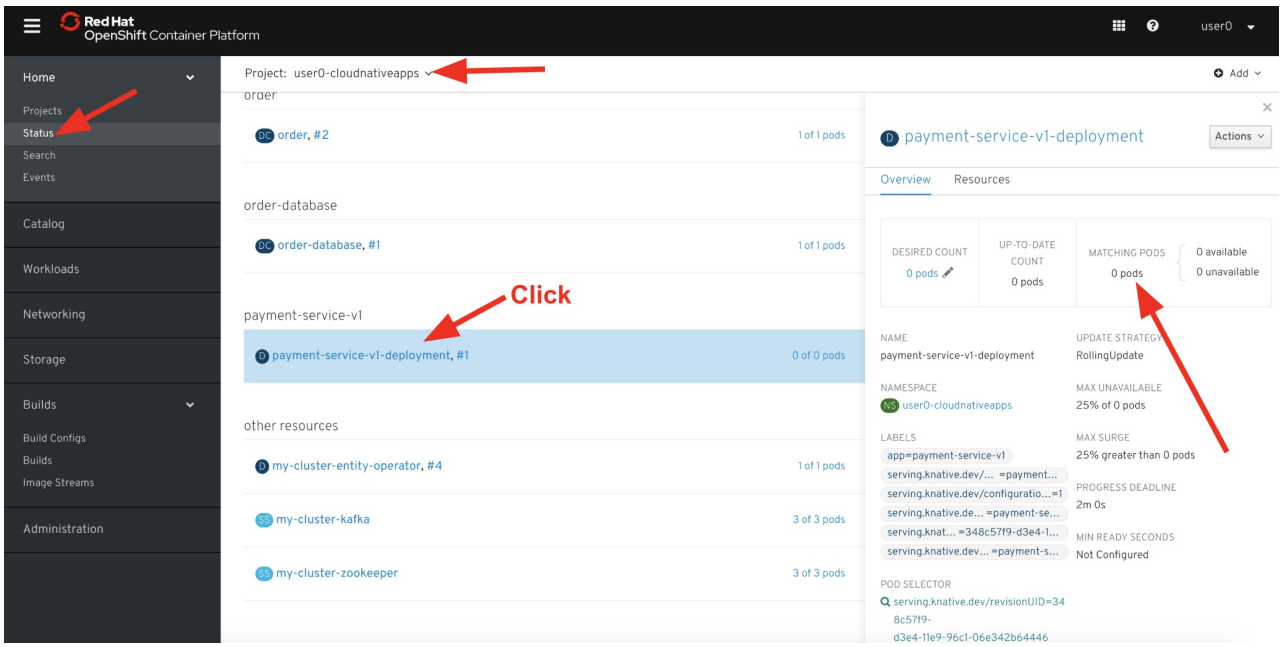
Once you click on **config-autoscaler**, click on the **YAML** tab to show the source code to the config map. Here you will see the details on how Knative autoscaling feature is specified.

As default, Knative will automatically scale services down to zero instances when the service(i.e. payment) has no request after 30 seconds:

```
scale-to-zero-grace-period: 30s
```



In the meantime, it probably took at least 30 seconds so select your `userXX-cloudnativeapps` project using the drop-down at the top and then go back to *Home* > *Status* on the left menu and click on **payment-v1-deployment**. You will see 0 pods are available:



You can't access the serverless service using traditional routing (e.g. `oc get route`). Knative maintains its own routing table for managed services. You can list the routes that knative knows of with:

```
oc get rt
```

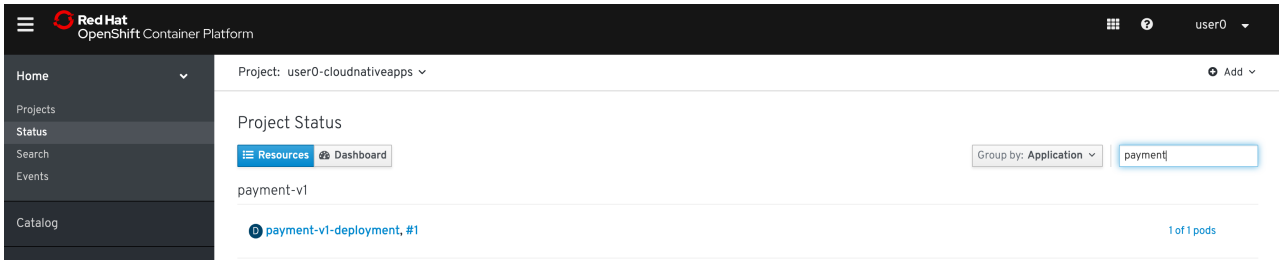
```
NAME      URL                                          READY   REASON
payment   http://payment.userXX-cloudnativeapps.[subdomain]   True
```

If you send traffic to this endpoint it will trigger the autoscaler to scale the app up. Trigger the app:

```
export SVC_URL=$(oc get rt payment -o template='{{ .status.url }}')
```

```
curl -i -H 'Content-Type: application/json' -d '{"foo": "bar"}' $SVC_URL
```

This will send some dummy data to the `payment` service, but more importantly it triggered knative to spin up the pod again automatically, and will shut it down 30 seconds later.



`Congratulations!` You've now deployed the payment service as a Quarkus native image, served with *Knative Serving*, quicker than traditional Java applications. This is not the end of Knative capabilites so we will now see how the payment service will scale up *magically* in the following exercises.

Create KafkaSource to enable Knative Eventing

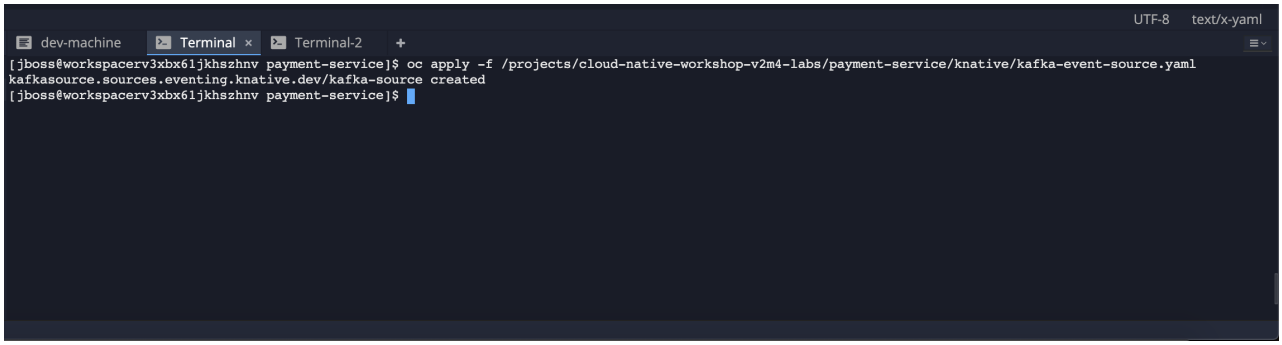
In this lab, Knative Eventing is already installed but if you want to install it in your own OpenShift cluster then you can install it via the *Knative Eventing Operator* in OpenShift web console.

Open `knative/kafka-event-source.yaml` (in the *payment-service* project) to define a *KafkaSource* to integrate with the Knative Eventing. Copy the following YAML code to this file:

```
apiVersion: sources.eventing.knative.dev/v1alpha1
kind: KafkaSource
metadata:
  name: kafka-source
spec:
  consumerGroup: payment-consumer-group
  bootstrapServers: my-cluster-kafka-bootstrap:9092
  topics: orders
  sink:
    apiVersion: serving.knative.dev/v1alpha1
    kind: Service
    name: payment
```

The object can then be deployed using the following command via CodeReady Workspaces Terminal:

`oc apply -f /projects/cloud-native-workshop-v2m4-labs/payment-service/knative/kafka-event-source.yaml`



You can also see a new pod spun up which will manage the connection between Kafka and our **payments** service:

`oc get pods -l knative-eventing-source-name=kafka-source`

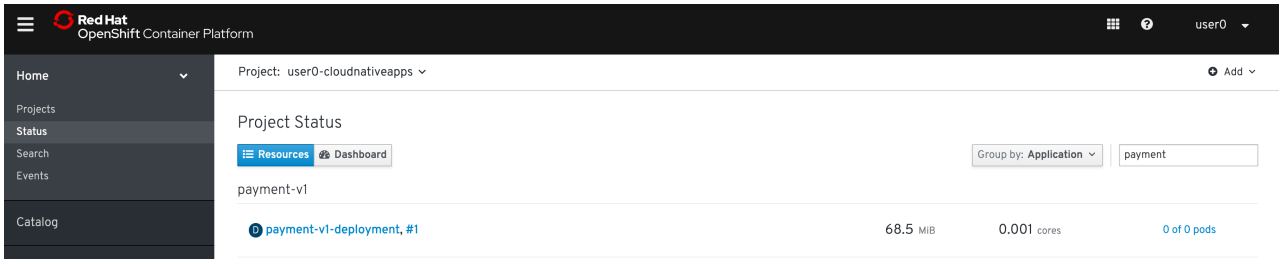
```
NAME                            READY   STATUS    RESTARTS   AGE
kafka-source-jktp9-b6b4c6797-4rspk   2/2     Running   1          21s
```

Great job!

Let's make sure if the payment service works properly with Knative features via Coolstore Web UI.

4. End to End Functional Testing

Before getting started, we need to make sure if *payment service* is scaled down to *zero* again in *Project Status*:

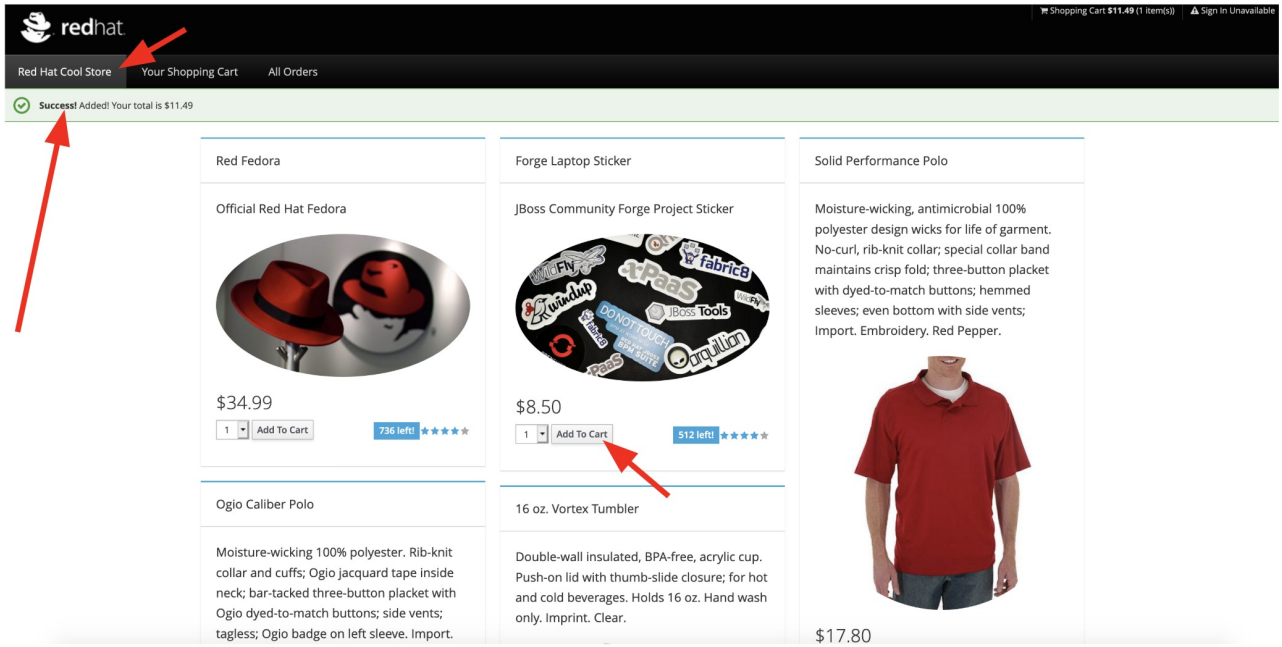


Let's go shopping! Open the Web UI in your browser. To get the URL to the Web UI, run this command in CodeReady *Terminal*:

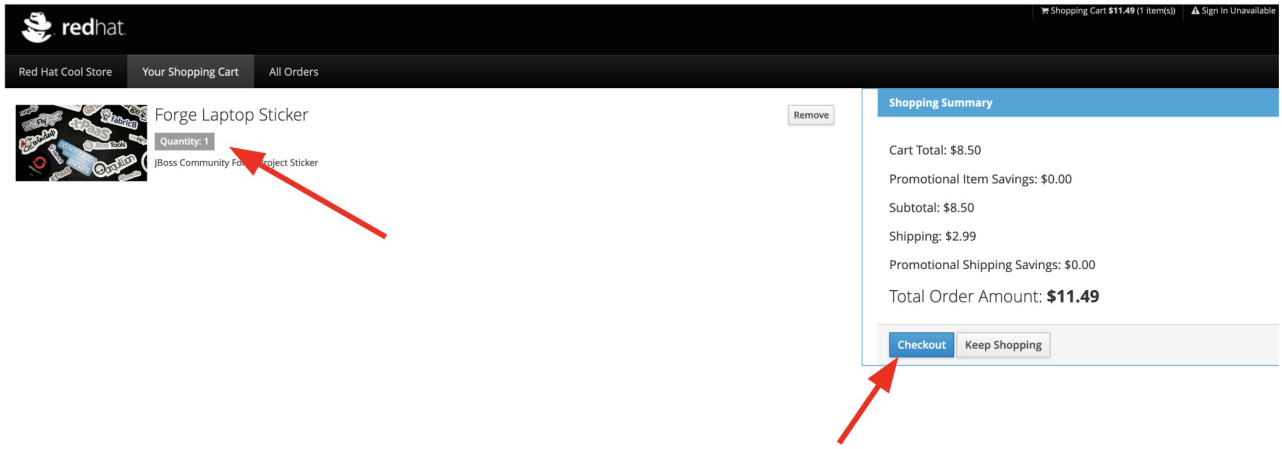
```
oc get route | grep coolstore-ui | awk '{print $2}'
```

Add some cool items to your shopping cart in the following shopping scenarios:

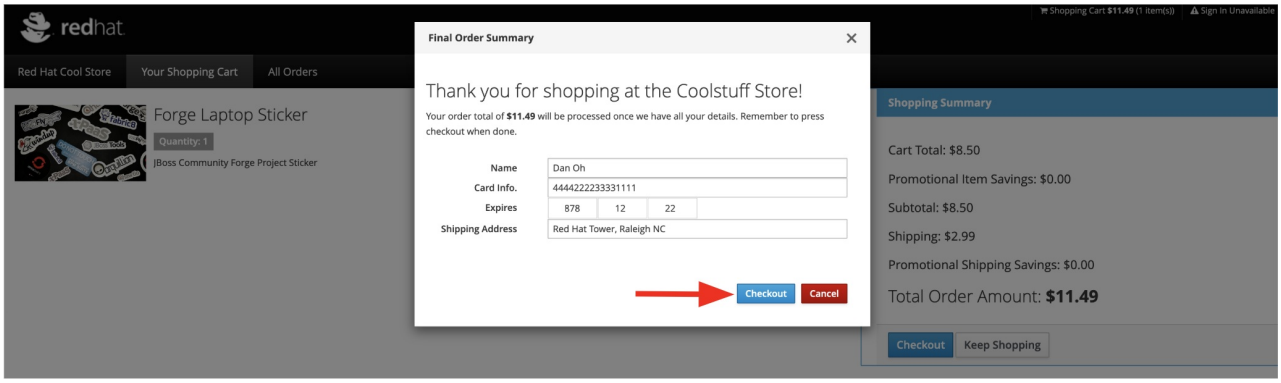
1) Add a *Froge Laptop Sticker* to your cart by click on **Add to Cart**. You will see the `Success! Added!` message under the top munu.



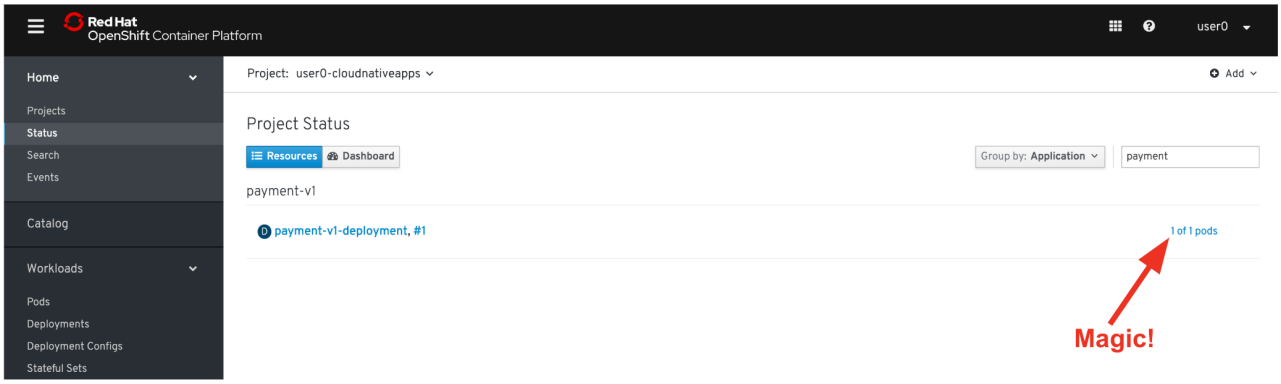
2) Go to the **Your Shopping Cart** tab and click on the **Checkout** button . Input the credit card information. The Card Info should be 16 digits and begin with the digit `4` . For example `4123987754646678` .



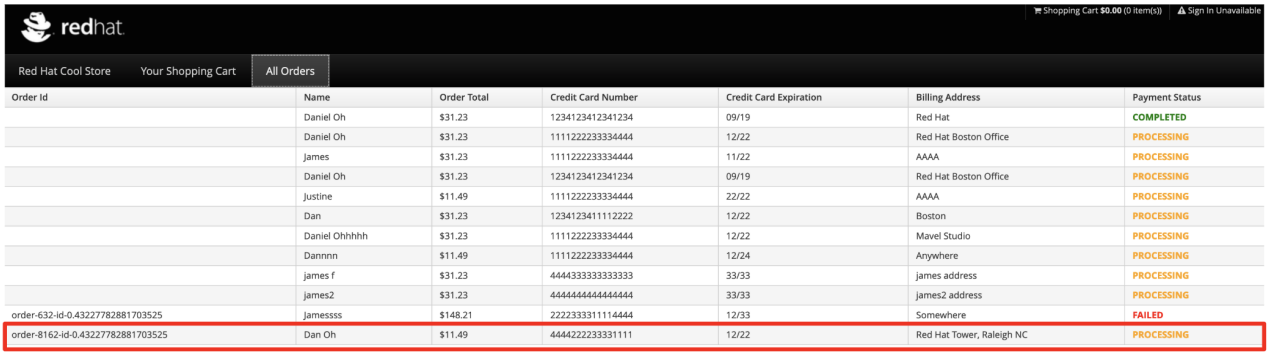
3) Input your Credit Card information to pay for the items:



4) Let's find out how *Kafka Event* enables *Knative Eventing*. Go back to *Project Status* in OpenShift web console then confirm if *payment service* is up automatically. It's `MAGIC!!`

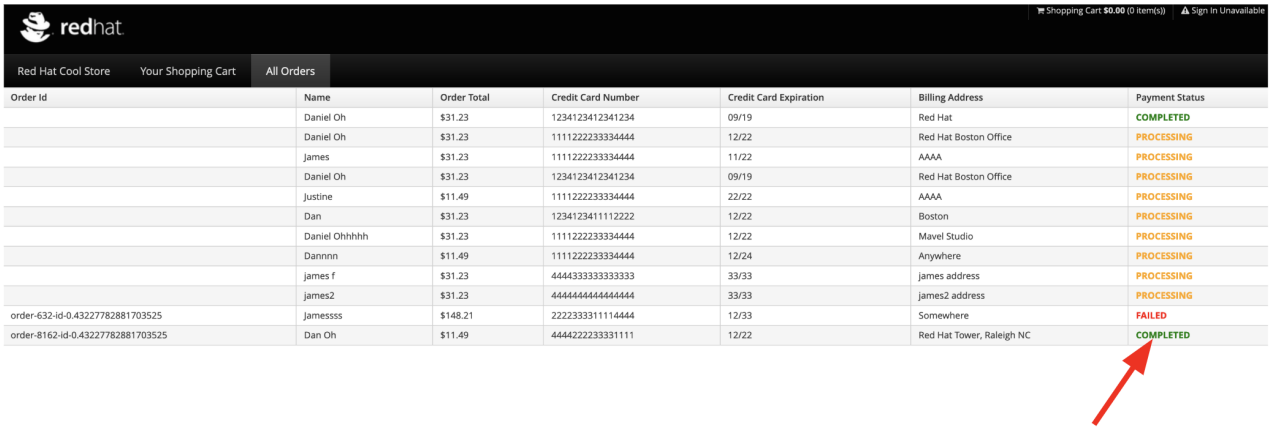


5) Confirm the *Payment Status* of the your shopping items in the **All Orders** tab. It should be `Processing` .



5) After a few moments, reload the **All Orders** page to confirm that the Payment Status changed to `COMPLETED` or `FAILED` .

> `Note` : If the status is still `Processing` , the order service is processing incoming Kafka messages and store thme in MongoDB. Please reload the page a few times more.



This is the same result as before, but using Knative eventing to make a more powerful event-driven system that can scale with demand.

5. Creating Cloud-Native CI/CD Pipelines using Tekton

What is the Cloud-Native CI/CD Pipelines?

There're lots of open source CI/CD tools to build, test, deploy, and manage cloud-native applications/microservices: from on-premise to private, public, and hybrid cloud. Each tool provides different features to integrate with existing platforms/systems. This sometimes makes it more complex for DevOps teams to be able to create the CI/CD pipelines and maintain them on Kubernetes clusters. The *cloud-native CI/CD pipeline* should be defined and executed in the Kubernetes native way. For example, the pipeline can be specified as Kubernetes resources using YAML format.

OpenShift Pipelines provides a cloud-native, continuous integration and delivery (CI/CD) solution for building pipelines using Tekton.

Tekton is a flexible, Kubernetes-native, open-source CI/CD framework that enables automating deployments across multiple platforms (Kubernetes, serverless, VMs, etc) by abstracting away the underlying details.

OpenShift Pipelines features:

- Standard CI/CD pipeline definition based on Tekton
- Build images with Kubernetes tools such as S2I, Buildah, Buildpacks, Kaniko, etc
- Deploy applications to multiple platforms such as Kubernetes, serverless and VMs
- Easy to extend and integrate with existing tools
- Scale pipelines on-demand
- Portable across any Kubernetes platform
- Designed for microservices and decentralized teams
- Integrated with the OpenShift Developer Console

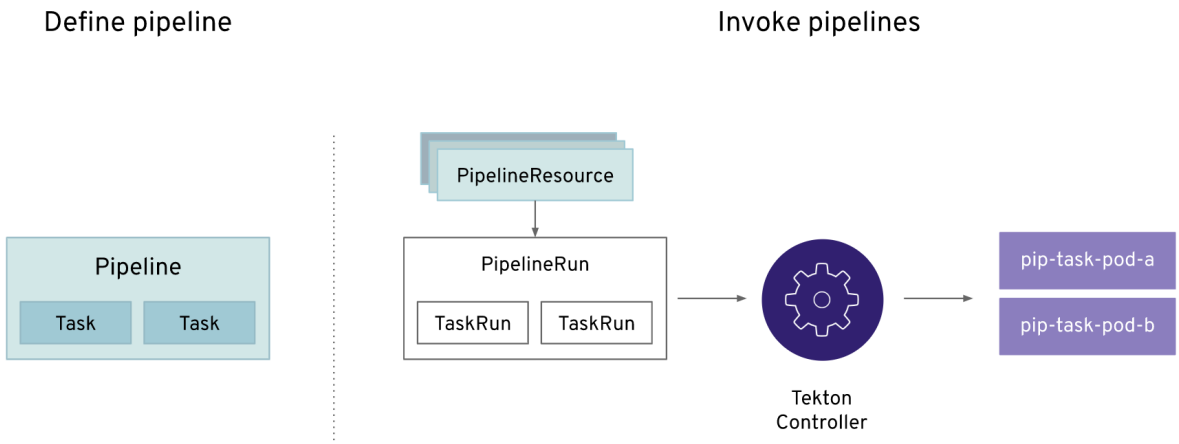
> In the lab, OpenShift Pipelines is already installed on OpenShift cluster but if you want to install OpenShift Pipelines on your own OpenShift cluster, OpenShift Pipelines is provided as an add-on on top of OpenShift that can be installed via an operator available in the OpenShift OperatorHub.

What is Tekton?

Tekton defines a number of Kubernetes custom resources as building blocks in order to standardize pipeline concepts and provide a terminology that is consistent across CI/CD solutions. These custom resources are an extension of the Kubernetes API that let users create and interact with these objects using the OpenShift CLI (`oc`), `kubectl` , and other Kubernetes tools.

The custom resources needed to define a pipeline are listed below:

- `Task` : a reusable, loosely coupled number of steps that perform a specific task (e.g. building a container image)
- `Pipeline` : the definition of the pipeline and the tasks that it should perform
- `PipelineResource` : inputs (e.g. git repository) and outputs (e.g. image registry) to and out of a pipeline or task
- `TaskRun` : the execution and result (i.e. success or failure) of running an instance of task
- `PipelineRun` : the execution and result (i.e. success or failure) of running a pipeline



For further details on pipeline concepts, refer to the Tekton documentation that provides an excellent guide for understanding various parameters and attributes available for defining pipelines.

In this lab, we will walk you through pipeline concepts and how to create and run a CI/CD pipeline for building and deploying serverless applications on `Knative` on OpenShift.

Deploy Sample Application

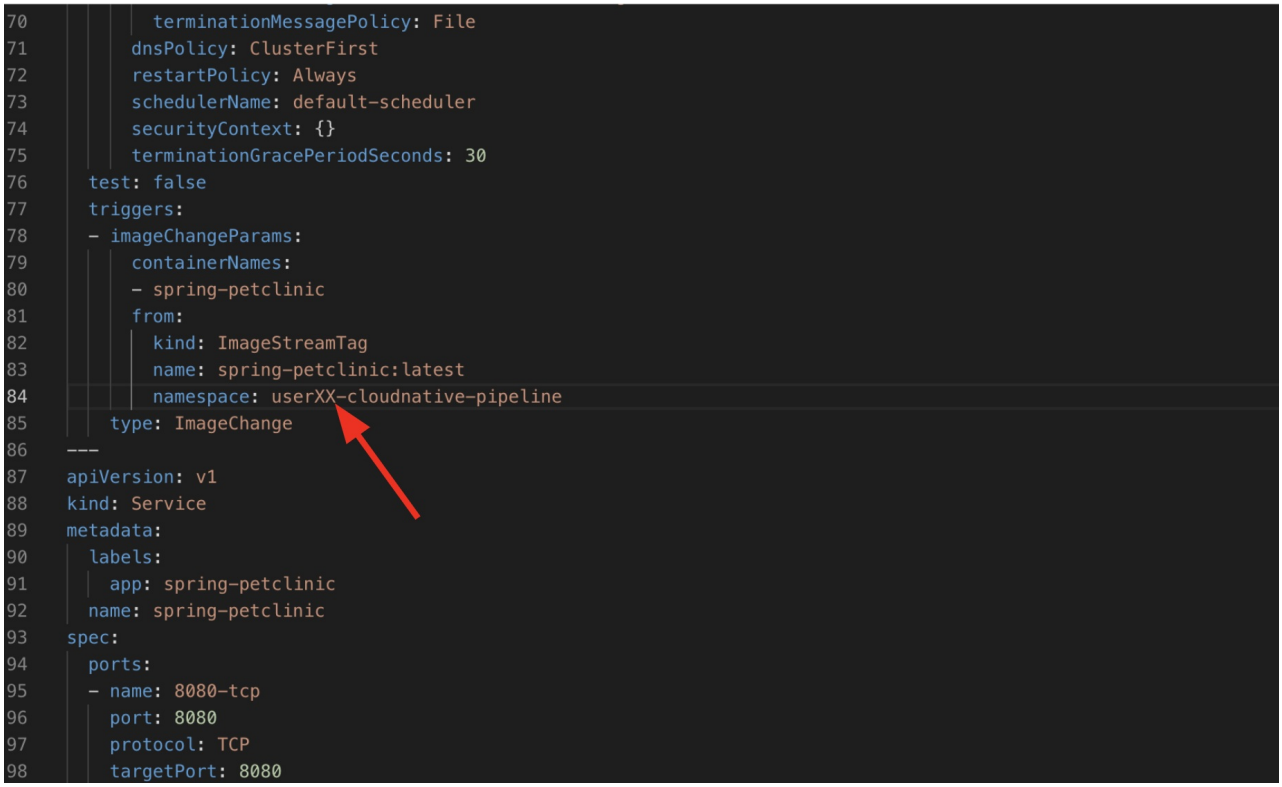
Change to your developer project for the sample application that you will be using in this lab using this command: (replace `userXX` with your username):

```
oc project userXX-cloudnative-pipeline
```

You will use the Spring PetClinic sample application during this tutorial, which is a simple Spring Boot application.

Create the Kubernetes objects for deploying the PetClinic app on OpenShift. The deployment will not complete since there are no container images built for the PetClinic application yet. That you will do in the following sections through a CI/CD pipeline.

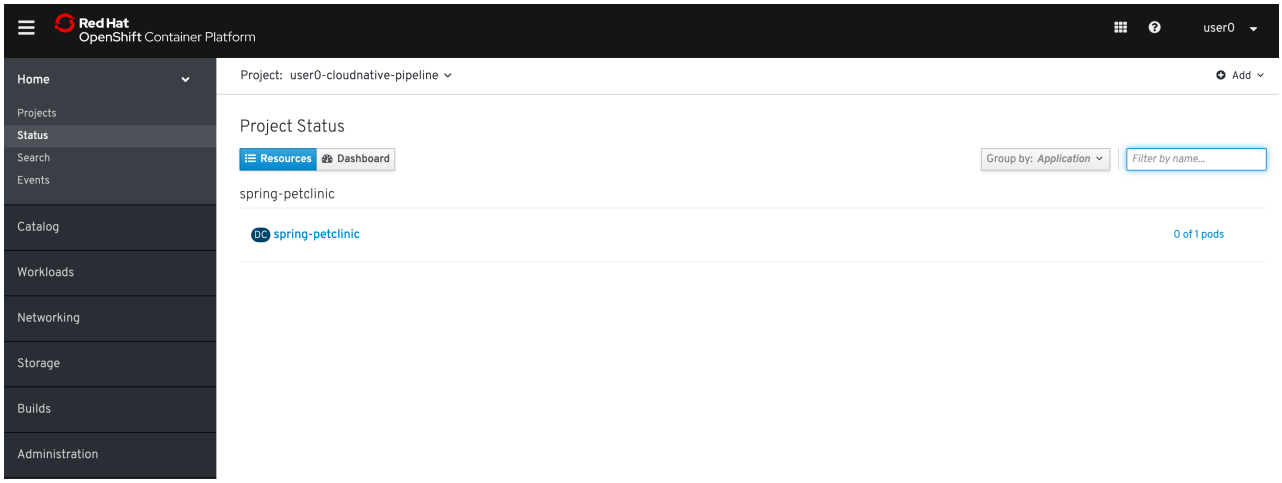
Replace `userXX-cloudnative-pipeline` with your username in **knative/pipeline/petclinic.yaml**:



Then create the object in Kubernetes:

```
oc create -f knative/pipeline/petclinic.yaml
```

You should be able to see the deployment in the OpenShift web console.



Install Tasks

`Tasks` consist of a number of steps that are executed sequentially. Each `task` is executed in a separate container within the same pod. They can also have inputs and outputs in order to interact with other tasks in the pipeline.

Here is an example of a Maven task for building a Maven-based Java application:

```
apiVersion: tekton.dev/v1alpha1
kind: Task
metadata:
  name: maven-build
spec:
  inputs:
    resources:
    - name: workspace-git
      targetPath: /
      type: git
  steps:
  - name: build
    image: maven:3.6.0-jdk-8-slim
    command:
    - /usr/bin/mvn
    args:
    - install
```

When a `task` starts running, it starts a pod and runs each `step` sequentially in a separate container on the same pod. This task happens to have a single step, but tasks can have multiple steps, and, since they run within the same pod, they have access to the same volumes in order to cache files, access configmaps, secrets, etc. `Tasks` can also receive inputs (e.g., a git repository) and outputs (e.g., an image in a registry) in order to interact with each other.

Note that only the requirement for a git repository is declared on the task and not a specific git repository to be used. That allows `tasks` to be reusable for multiple pipelines and purposes. You can find more examples of reusable `tasks` in the Tekton Catalog and OpenShift Catalog repositories.

Install the `openshift-client` and `s2i-java` tasks from the catalog repository using `oc` or `kubectl` , which you will need for creating a pipeline in the next section:

Create the following Tekton tasks which will be used in the `Pipelines` :

```
oc create -f knative/pipeline/openshift-client-task.yaml
```

```
oc create -f knative/pipeline/s2i-java-8-task.yaml
```

Let's confirm if the **tasks** are installed properly using Tekton CLI that already installed in CodeReady Workspaces.

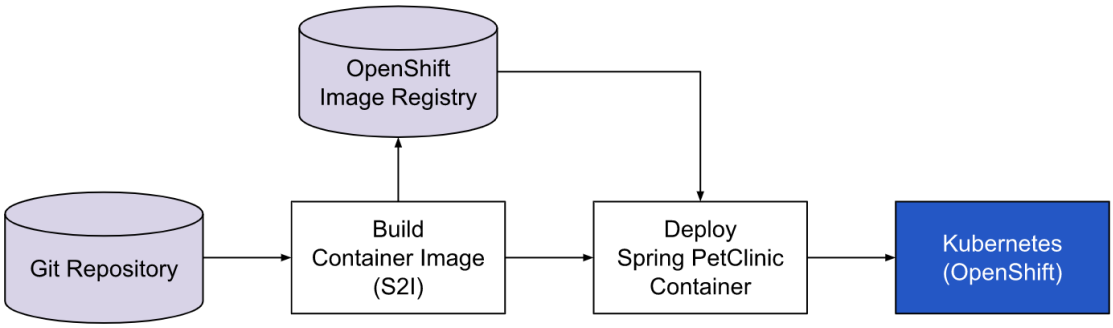
```
tkn task list
```

```
openshift-client   7 seconds ago
s2i-java-8         3 seconds ago
```

Create Pipeline

A pipeline defines a number of tasks that should be executed and how they interact with each other via their inputs and outputs.

In this lab, we will create a pipeline that takes the source code of PetClinic application from GitHub and then builds and deploys it on OpenShift using Source-to-Image (S2I).



Here is the YAML file that represents the above pipeline:

```
apiVersion: tekton.dev/v1alpha1
kind: Pipeline
metadata:
  name: petclinic-deploy-pipeline
spec:
  resources:
  - name: app-git
    type: git
  - name: app-image
    type: image
  tasks:
  - name: build
    taskRef:
      name: s2i-java-8
    params:
      - name: TLSVERIFY
        value: "false"
    resources:
      inputs:
      - name: source
        resource: app-git
      outputs:
      - name: image
        resource: app-image
  - name: deploy
    taskRef:
      name: openshift-client
    runAfter:
      - build
    params:
    - name: ARGS
      value:
        - rollout
        - latest
        - spring-petclinic
```

This pipeline performs the following:

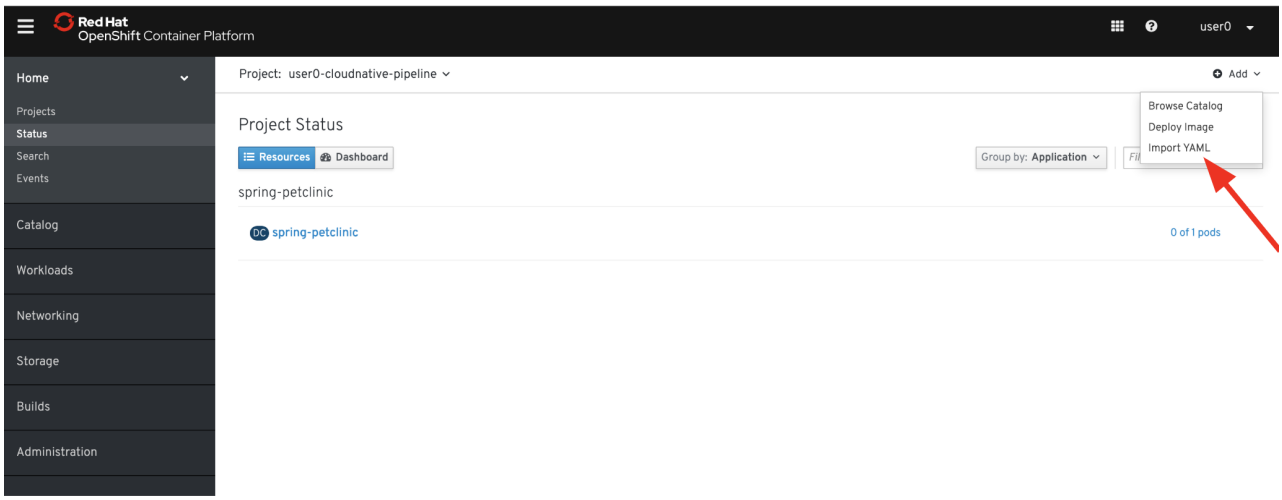
- Clones the source code of the application from a Git repository (`app-git` resource)
- Builds the container image using the `s2i-java-8` task that generates a `Dockerfile` for the application and uses Buildah to build the image
- The application image is pushed to an image registry (`app-image` resource)
- The new application image is deployed on OpenShift using the `openshift-cli`

You might have noticed that there are no references to the PetClinic Git repository and its image in the registry. That's because `Pipelines` in Tekton are designed to be generic and re-usable across environments and stages through the application's lifecycle. `Pipelines` abstract away the specifics of the Git source repository and image to be

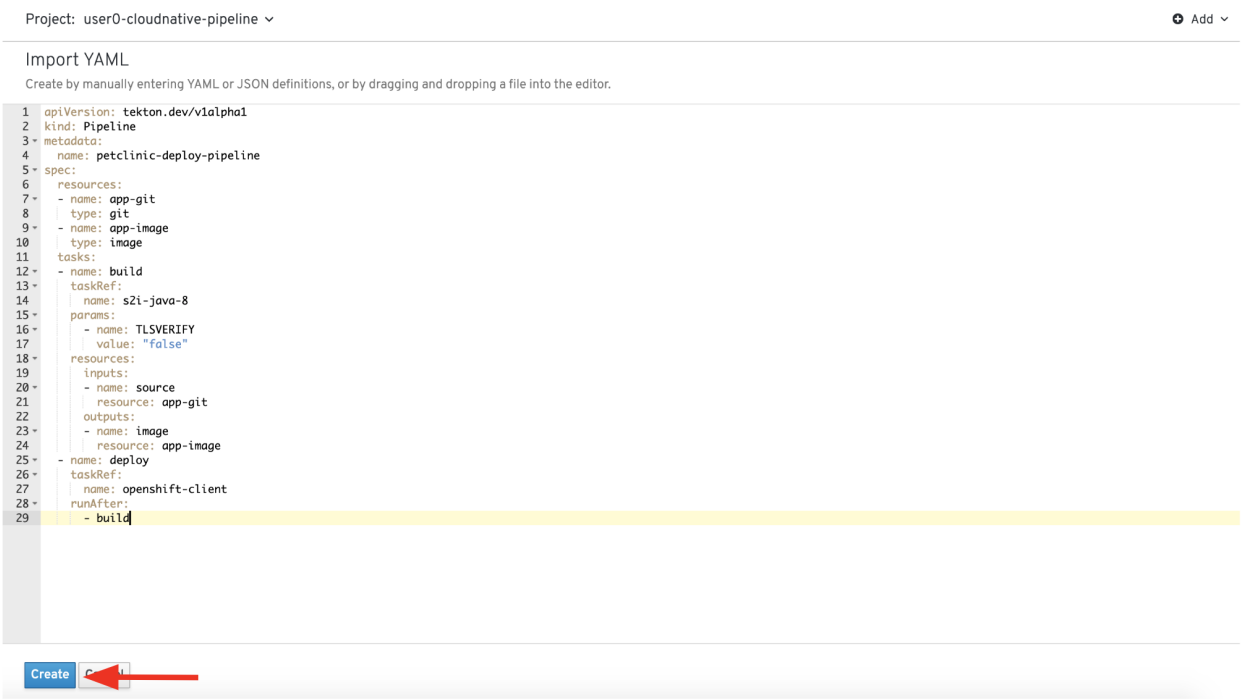
produced as `resources`. When triggering a pipeline, you can provide different Git repositories and image registries to be used during pipeline execution. Be patient! You will do that in a little bit in the next section.

The execution order of `tasks` is determined by dependencies that are defined between the tasks via `inputs` and `outputs` as well as explicit orders that are defined via `runAfter`.

In the OpenShift web console, you can click on *Add* > *Import YAML* at the top right of the screen while you are in the `userXX-cloudnative-pipeline` project.



Paste the YAML into the textfield, and click on `Create`.



Check the list of pipelines you have created in CodeReady Workspaces Terminal:

```
tkn pipeline ls
```

```
NAME                    AGE              LAST RUN   STARTED   DURATION   STATUS
petclinic-deploy-pipeline  8 seconds ago  ---        ---       ---        ---
```

Trigger Pipeline

Now that the pipeline is created, you can trigger it to execute the tasks specified in the pipeline. Triggering pipelines is an area that is under development and in the next release it will be possible to be done via the OpenShift web console and Tekton CLI. In this tutorial, you will trigger the pipeline through creating the Kubernetes objects (the hard way!) in order to learn the mechanics of triggering.

First, you should create a number of `PipelineResources` that contain the specifics of the Git repository and image registry to be used in the pipeline during execution. Expectedly, these are also reusable across multiple pipelines.

The following `PipelineResource` defines the Git repository and reference for the PetClinic application. Create the following pipeline resources via the OpenShift web console via `Add → Import YAML` :

```
apiVersion: tekton.dev/v1alpha1
kind: PipelineResource
metadata:
  name: petclinic-git
spec:
  type: git
  params:
  - name: url
    value: https://github.com/spring-projects/spring-petclinic
```

And the following defines the OpenShift internal registry for the PetClinic image to be pushed to. Create the following pipeline resources via the OpenShift web console via `Add → Import YAML` . Replace your username with `userXX` :

```
apiVersion: tekton.dev/v1alpha1
kind: PipelineResource
metadata:
  name: petclinic-image
spec:
  type: image
  params:
  - name: url
    value: image-registry.openshift-image-registry.svc:5000/userXX-cloudnative-pipeline/spring-petclinic
```

Create the above pipeline resources via the OpenShift web console via `Add → Import YAML` .

You can see the list of resources created in CodeReady Workspaces Terminal:

`tkn resource ls`

```
NAME            TYPE    DETAILS
petclinic-git   git     url: https://github.com/spring-projects/spring-petclinic
petclinic-image image   url: image-registry.openshift-image-registry.svc:5000/userXX-cloudnative-pipeline/spring-petclinic
```

A `PipelineRun` is how you can start a pipeline and tie it to the Git and image resources that should be used for this specific invocation. You can start the pipeline in CodeReady Workspaces Terminal:

```
tkn pipeline start petclinic-deploy-pipeline \
    -r app-git=petclinic-git \
    -r app-image=petclinic-image \
    -s pipeline
```

The result looks like:

```
Pipelinerun started: petclinic-deploy-pipeline-run-97kdv
```

The `-r` flag specifies the PipelineResources that should be provided to the pipeline and the `-s` flag specifies the service account to be used for running the pipeline.

As soon as you started the `petclinic-deploy-pipeline pipeline`, a pipelinerun is instantiated and pods are created to execute the tasks that are defined in the pipeline.

`tkn pipeline list`

```
NAME                    AGE            LAST RUN                          STARTED         DURATION   STATUS
petclinic-deploy-pipeline   21 seconds ago   petclinic-deploy-pipeline-run-97kdv   11 seconds ago   ---
Running
```

Check out the logs of the pipeline as it runs using the `tkn pipeline logs` command which interactively allows you to pick the pipelinerun of your interest and inspect the logs:

`tkn pipeline logs -f`

```
? Select pipeline : petclinic-deploy-pipeline
? Select pipelinerun : petclinic-deploy-pipeline-run-97kdv started 39 seconds ago

...
[build : push] Copying config
sha256:6c2be43b49deee05b0dee97bd23dab0dcfd9b1b6352fd085f833f62e7d106ae8
[build : push] Writing manifest to image destination
[build : push] Copying config
sha256:6c2be43b49deee05b0dee97bd23dab0dcfd9b1b6352fd085f833f62e7d106ae8
[build : push] Writing manifest to image destination
...
[build : image-digest-exporter-bj6dr] 2019/09/17 05:06:09 Image digest exporter output: []
[deploy : oc] deploymentconfig.apps.openshift.io/spring-petclinic rolled out
```

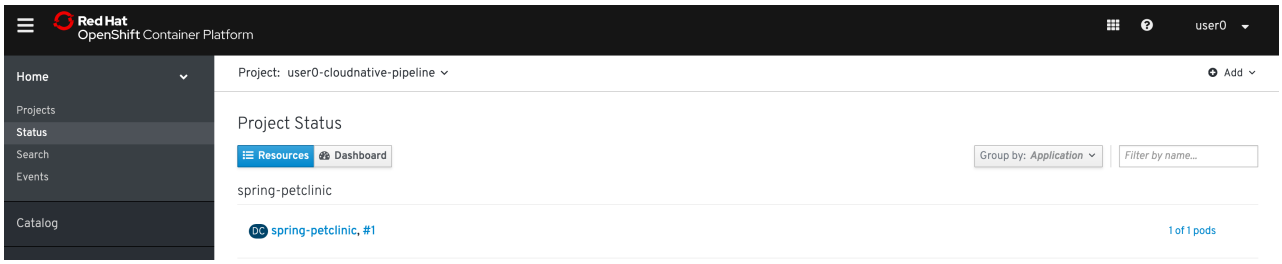
> `Note` : The build log(*ImageResource petclinic-image doesn't have an index.json file*) doesn't mean an error but it's vailation check. Even if you're failed, **Pipeline Build** will continue.

After a few minutes, the pipeline should finish successfully.

`tkn pipeline list`

```
NAME                      AGE              LAST RUN                          STARTED          DURATION   STATUS
petclinic-deploy-pipeline   7 minutes ago   petclinic-deploy-pipeline-run-97kdv   5 minutes ago   4 minutes   Succeeded
```

Looking back at the project, you should see that the PetClinic image is successfully built and deployed.



Summary

In this module, we learned how to develop cloud-native applications using multiple Java runtimes (Quarkus and Spring Boot), Javascript (Node.js) and different datasources (i.e. PostgreSQL, MongoDB) to handle a variety of business use cases which implement real-time *request/response* communication using REST APIs, high performing cacheable services using **JBoss Data Grid**, event-driven/reactive shopping cart service using Apache Kafka in **Red Hat AMQ Streams**, and in the end, we treated the payment service as a `Serverless` application using `Knative` with Serving, Eventing, and Pipeline(Tekton).

Red Hat Runtimes enables enterprise developers to design the advanced cloud-native architecture and develop, build, deploy the cloud-native application on hybrid cloud on the **Red Hat OpenShift Container Platform**. Congratulations!

Additional Resources:
