+ -

▼

Step 1 of 9

Creating reactive microservices using MicroProfile Reactive Messaging

Learn how to write reactive Java microservices using MicroProfile Reactive Messaging.

What you'll learn

You will learn how to build reactive microservices that can send requests to other microservices, and asynchronously receive and process the responses. You will use an external messaging system to handle the asynchronous messages that are sent and received between the microservices as streams of events. MicroProfile Reactive Messaging makes it easy to write and configure your application to send, receive, and process the events efficiently.

Asynchronous messaging between microservices

Asynchronous communication between microservices can be used to build reactive and responsive applications. By decoupling the requests sent by a microservice from the responses that it receives, the microservice is not blocked from performing other tasks while waiting for the requested data to become available. Imagine asynchronous communication as a restaurant. A waiter might come to your table and take your order. While you are waiting for your food to be prepared, that waiter serves other tables and takes their orders too. When your food is ready, the waiter brings your food to the table and then continues to serve the other tables. If the waiter were to operate synchronously, they must take your order and then wait until they deliver your food before serving any other tables. In microservices, a request call from a REST client to another microservice can be time-consuming because the network might be slow, or the other service might be overwhelmed with requests and can't respond quickly. But in an asynchronous system, the microservice sends a request to another microservice and continues to send other calls and to receive and process other responses until it receives a response to the original request.

What is MicroProfile Reactive Messaging?

MicroProfile Reactive Messaging provides an easy way to asynchronously send, receive, and process messages that are received as continuous streams of events. You simply annotate application beans' methods and Open Liberty converts the annotated methods to reactive streams-compatible publishers, subscribers, and processors and connects them up to each other. MicroProfile Reactive Messaging provides a Connector API so that your methods can be connected to external messaging systems that produce and consume the streams of events, such as Apache Kafka.

The application in this guide consists of two microservices, **system** and **inventory**. Every 15 seconds, the **system** microservice calculates and publishes an event that contains its current average system load. The inventory microservice subscribes to that information so that it can keep an updated list of all the systems and their current system loads. The current inventory of systems can be accessed via the **/systems** REST endpoint. You'll create the **system** and **inventory** microservices using MicroProfile Reactive Messaging.

Getting started

The fastest way to work through this guide is to clone the Git repository and use the project that's inside:

```
git clone https://github.com/openliberty/guide-microprofile-reactive-messaging.git
```

Navigate into the guide's **start** directory which containers the starting project that you will build upon.

```
cd guide-microprofile-reactive-messaging/start
```

The **finish** directory contains the finished project that you will build.

Creating the Producer in the system microservice

The **system** microservice is the producer of the messages that are published to the Kafka messaging system as a stream of events. Every 15 seconds, the **system** microservice publishes an event that contains its calculation of the average system load (its CPU usage) for the last minute.

Create the **SystemService.java file**

```
touch system/src/main/java/io/openliberty/guides/system/SystemService.java
```

Open **SystemService.java** by navigating to

> [File -> Open]guide-microprofile-reactive-messaging/start/system/src/main/java/io/openliberty/guides/system/SystemService.java

Add the Java code:

```
package main.java.io.openliberty.guides.system;

import java.lang.management.ManagementFactory;
import java.lang.management.OperatingSystemMXBean;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.concurrent.TimeUnit;

import javax.enterprise.context.ApplicationScoped;

import org.eclipse.microprofile.reactive.messaging.Outgoing;
import org.reactivestreams.Publisher;

import io.openliberty.guides.models.SystemLoad;
```

```
import io.reactivex.rxjava3.core.Flowable;

@ApplicationScoped
public class SystemService {

    private static final OperatingSystemMXBean osMean =
            ManagementFactory.getOperatingSystemMXBean();
    private static String hostname = null;

    private static String getHostname() {
        if (hostname == null) {
            try {
                return InetAddress.getLocalHost().getHostName();
            } catch (UnknownHostException e) {
                return System.getenv("HOSTNAME");
            }
        }
        return hostname;
    }

    @Outgoing("systemLoad")
    public Publisher<SystemLoad> sendSystemLoad() {
        return Flowable.interval(15, TimeUnit.SECONDS)
                .map((interval -> new SystemLoad(getHostname(),
                        new Double(osMean.getSystemLoadAverage()))));
    }

}
```

The **SystemService** class contains a **Publisher** method that is called **sendSystemLoad()**, which calculates and returns the average system load. The **@Outgoing** annotation on the **sendSystemLoad()** method indicates that the method publishes its calculation as a message on a topic in the Kafka messaging system. The **Flowable.interval()** method from **rxJava** is used to set the frequency of how often the system service publishes the calculation to the event stream.

The messages are transported between the service and the Kafka messaging system through a channel called **systemLoad**. The name of the channel to use is set in the **@Outgoing("systemLoad")** annotation. Later in the guide, you will configure the service so that any messages sent by the **system** service through the **systemLoad** channel are published on a topic called **systemLoadTopic**.

Creating the consumer in the inventory microservice

The **inventory** microservice records in its inventory the average system load information that it received from potentially multiple instances of the **system** service.

Create **InventoryResource.java**

```
touch inventory/src/main/java/io/openliberty/guides/inventory/InventoryResource.java
```

> [File -> Open]guide-microprofile-reactive-messaging/start/inventory/src/main/java/io/openliberty/guides/inventory/InventoryResource.java

Add the Java code:

```
package io.openliberty.guides.inventory;

import java.util.List;
import java.util.Optional;
import java.util.Properties;
import java.util.logging.Logger;
import java.util.stream.Collectors;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

import org.eclipse.microprofile.reactive.messaging.Incoming;

import io.openliberty.guides.models.SystemLoad;

@ApplicationScoped
@Path("/inventory")
public class InventoryResource {

    private static Logger logger = Logger.getLogger(InventoryResource.class.getName());

    @Inject
    private InventoryManager manager;

    @GET
    @Path("/systems")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getSystems() {
```

```
        List<Properties> systems = manager.getSystems()
                .values()
                .stream()
                .collect(Collectors.toList());
        return Response
                .status(Response.Status.OK)
                .entity(systems)
                .build();
    }

    @GET
    @Path("/system/{hostId}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getSystem(@PathParam("hostId") String hostId) {
        Optional<Properties> system = manager.getSystem(hostId);
        if (system.isPresent()) {
            return Response
                    .status(Response.Status.OK)
                    .entity(system)
                    .build();
        }
        return Response
                .status(Response.Status.NOT_FOUND)
                .entity("hostId does not exist.")
                .build();
    }

    @DELETE
    @Produces(MediaType.APPLICATION_JSON)
    public Response resetSystems() {
        manager.resetSystems();
        return Response
                .status(Response.Status.OK)
                .build();
    }

    @Incoming("systemLoad")
    public void updateStatus(SystemLoad sl)  {
        String hostname = sl.hostname;
        if (manager.getSystem(hostname).isPresent()) {
            manager.updateCpuStatus(hostname, sl.loadAverage);
            logger.info("Host " + hostname + " was updated: " + sl);
        } else {
            manager.addSystem(hostname, sl.loadAverage);
            logger.info("Host " + hostname + " was added: " + sl);
        }
    }
}
```

The **inventory** microservice receives the message from the **system** microservice over the **@Incoming("systemLoad")** channel. The properties of this channel are defined in the **microprofile-config.properties** file. The **inventory** microservice is also a RESTful service that is served at the **/inventory** endpoint.

The **InventoryResource** class contains a method called **updateStatus()**, which receives the message that contains the average system load and updates its existing inventory of systems and their average system load. The **@Incoming("systemLoad")** annotation on the **updateStatus()** method indicates that the method retrieves the average system load information by connecting to the channel called **systemLoad**. Later in the guide, you will configure the service so that any messages sent by the **system** service through the **systemLoad** channel are retrieved from a topic called **systemLoadTopic**.

Configuring the MicroProfile Reactive Messaging connectors for Kafka

The **system** and **inventory** services exchange messages with the external messaging system through a channel. The MicroProfile Reactive Messaging Connector API makes it easy to connect each service to the channel. You just need to add configuration keys in a properties file for each of the services. These configuration keys define properties such as the name of the channel and the topic in the Kafka messaging system. Open Liberty includes the **liberty-kafka** connector for sending and receiving messages from Apache Kafka.

The system and inventory microservices each have a MicroProfile Config properties file to define the properties of their outgoing and incoming streams.

Create the **system microprofile-config.properties** in the **META-INF** directory

```
touch system/src/main/resources/META-INF/microprofile-config.properties
```

Open **microprofile-config.properties**

> [File -> Open]guide-microprofile-reactive-messaging/start/system/src/main/resources/META-INF/microprofile-config.properties

Add the **configuration properties**

```
# Liberty Kafka connector
mp.messaging.connector.liberty-kafka.bootstrap.servers=localhost:9093

# systemLoad stream
mp.messaging.outgoing.systemLoad.connector=liberty-kafka
mp.messaging.outgoing.systemLoad.topic=systemLoadTopic
mp.messaging.outgoing.systemLoad.key.serializer=org.apache.kafka.common.serialization.StringSerializer
mp.messaging.outgoing.systemLoad.value.serializer=io.openliberty.guides.models.SystemLoad$SystemLoadSerializer
```

The **mp.messaging.connector.liberty-kafka.bootstrap.servers** property configures the hostname and port for connecting to the Kafka server. The **system** microservice uses an outgoing connector to send messages through the **systemLoad** channel to the **systemLoadTopic** topic in the Kafka messaging-broker so that the **inventory** microservices can consume the messages. The **key.serializer** and **value.serializer** properties characterize how to serialize the messages. The **SystemLoadSerializer** class implements the logic for turning a **SystemLoad** object into JSON and is configured as the **value.serializer**.

The **inventory** microservices uses a similar **microprofile-config.properties** configuration to define its required incoming stream. Create the inventory **microprofile-config.properties** file.

```
touch inventory/src/main/resources/META-INF/microprofile-config.properties
```

Open **microprofile-config.properties**

> [File -> Open]guide-microprofile-reactive-messaging/start/inventory/src/main/resources/META-INF/microprofile-config.properties

```
# Liberty Kafka connector
mp.messaging.connector.liberty-kafka.bootstrap.servers=localhost:9093

# systemLoad stream
mp.messaging.incoming.systemLoad.connector=liberty-kafka
mp.messaging.incoming.systemLoad.topic=systemLoadTopic
mp.messaging.incoming.systemLoad.key.deserializer=org.apache.kafka.common.serialization.StringDeserializer
mp.messaging.incoming.systemLoad.value.deserializer=io.openliberty.guides.models.SystemLoad$SystemLoadDeserializer
mp.messaging.incoming.systemLoad.group.id=system-load-status
```

The **inventory** microservice uses an incoming connector to receive messages through the **systemLoad** channel. The messages were published by the system microservice to the **systemLoadTopic** in the Kafka message broker. The **key.deserializer** and **value.deserializer** properties define how to deserialize the messages. The **SystemLoadDeserializer** class implements the logic for turning JSON into a **SystemLoad** object and is configured as the **value.deserializer**. The **group.id** property defines a unique name for the consumer group. A consumer group is a collection of consumers who share a common identifier for the group. You can also view a consumer group as the various machines that ingest from the Kafka topics. All of these properties are required by the Apache Kafka Producer Configs and Apache Kafka Consumer Configs.

Configuring the server

To use MicroProfile Reactive Messaging, you must enable the feature in the **server.xml** file for each service.

To run the services, the Open Liberty server on which each service runs needs to be correctly configured. Relevant features, including the MicroProfile Reactive Messaging feature, must be enabled for the **system** and **inventory** services.

Create the **server.xml** file

```
touch system/src/main/liberty/config/server.xml
```

Open the **server.xml** file

> [File -> Open] guide-microprofile-reactive-messaging/start/system/src/main/liberty/config/server.xml

Add the contents for the **server.xml**.

```
<server description="System Service">

  <featureManager>
    <feature>cdi-2.0</feature>
    <feature>concurrent-1.0</feature>
    <feature>jsonb-1.0</feature>
    <feature>mpHealth-2.1</feature>
    <feature>mpConfig-1.3</feature>
    <!-- tag::featureMP[] -->
    <feature>mpReactiveMessaging-1.0</feature>
    <!-- end::featureMP[] -->
  </featureManager>

  <variable name="default.http.port" defaultValue="9083"/>
  <variable name="default.https.port" defaultValue="9446"/>

  <httpEndpoint host="*" httpPort="${default.http.port}"
      httpsPort="${default.https.port}" id="defaultHttpEndpoint"/>

  <webApplication location="system.war" contextRoot="/"/>
</server>
```

The **server.xml** file is already configured for the **inventory** microservice.

Building and running the application

Build the **system** and **inventory** microservices using Maven and then run them in Docker containers.

Create the **pom.xml** file

```
touch system/pom.xml
```

Open the **pom.xml** file

> [File -> Open] guide-microprofile-reactive-messaging/start/system/pom.xml

The **pom.xml** file lists the **microprofile-reactive-messaging-api**, **kafka-clients**, and **rxjava** dependencies.

The **microprofile-reactive-messaging-api** dependency is needed to enable the use of MicroProfile Reactive Messaging API. The **kafka-clients** dependency is added because the application needs a Kafka client to connect to the Kafka broker. The **rxjava** dependency is used for creating events at regular intervals.

Add the **maven** dependencies, **properties**, **plugins**, **packaging method**, **name** and the **version**.

```
<?xml version='1.0' encoding='utf-8'?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>io.openliberty.guides</groupId>
    <artifactId>system</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <liberty.var.default.http.port>9083</liberty.var.default.http.port>
        <liberty.var.default.https.port>9446</liberty.var.default.https.port>
    </properties>

    <dependencies>
        <dependency>
            <groupId>jakarta.platform</groupId>
            <artifactId>jakarta.jakartaee-api</artifactId>
            <version>8.0.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.eclipse.microprofile</groupId>
            <artifactId>microprofile</artifactId>
            <version>3.3</version>
            <type>pom</type>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.eclipse.microprofile.reactive.messaging</groupId>
            <artifactId>microprofile-reactive-messaging-api</artifactId>
            <version>1.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>io.openliberty.guides</groupId>
            <artifactId>models</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-clients</artifactId>
            <version>2.4.0</version>
        </dependency>
        <dependency>
            <groupId>io.reactivex.rxjava3</groupId>
            <artifactId>rxjava</artifactId>
            <version>3.0.0</version>
        </dependency>
        <dependency>
            <groupId>org.microshed</groupId>
            <artifactId>microshed-testing-liberty</artifactId>
            <version>0.9</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.testcontainers</groupId>
            <artifactId>kafka</artifactId>
            <version>1.12.5</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>5.6.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <finalName>${project.artifactId}</finalName>
        <plugins>
```

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <version>3.2.3</version>
                <configuration>
                    <packagingExcludes>pom.xml</packagingExcludes>
                </configuration>
            </plugin>

            <plugin>
                <groupId>io.openliberty.tools</groupId>
                <artifactId>liberty-maven-plugin</artifactId>
                <version>3.2.1</version>
            </plugin>

            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.22.2</version>
            </plugin>

            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-failsafe-plugin</artifactId>
                <version>2.22.2</version>
                <executions>
                    <execution>
                        <id>integration-test</id>
                        <goals>
                            <goal>integration-test</goal>
                        </goals>
                        <configuration>
                            <trimStackTrace>false</trimStackTrace>
                        </configuration>
                    </execution>
                    <execution>
                        <id>verify</id>
                        <goals>
                            <goal>verify</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

The **Dockerfiles** are already provided for use.

To build the application, run the Maven **install** and **package** goals from the command line:

```
mvn -pl models install
```

Package the application:

```
mvn package
```

Update to the latest **open-liberty** Docker image.

```
docker pull open-liberty
```

Containerize the microservices:

Build the **system** docker image

```
docker build -t system:1.0-SNAPSHOT system/.
```

Build the **inventory** docker image

```
docker build -t inventory:1.0-SNAPSHOT inventory/.
```

Next, use the provided script to start the application in Docker containers. The script creates a network for the containers to communicate with each other. It also creates containers for Kafka, Zookeeper, and the microservices in the project. For simplicity, the script starts one instance of the system service.

Start the application

```
./scripts/startContainers.sh
```

Testing the application

After the application is up and running, you can access the application by making a GET request to the **/systems** endpoint of the **inventory** service.

To access the **inventory** microservice, use the **inventory/systems** URL, and you see the CPU **systemLoad** property for all the systems. Be aware that it will take a couple of minutes for all of the services to start up so you may have to wait to receive the expected output.

```
curl http://localhost:9085/inventory/systems
```

The output should contain the **hostname** and **systemLoad**

```
{
   "hostname":"30bec2b63a96",
   "systemLoad":2.25927734375
}
```

You can revisit the **/inventory/systems** URL after a while, and you will notice the **CPU systemLoad** property for all the systems have changed.

```
curl http://localhost:9085/inventory/systems
```

Use the **hostId** URL to see the CPU systemLoad property for one particular system.

In this example the **hostId = 30bec2b63a96**

```
curl http://localhost:9085/inventory/system/HOST_ID
```

Example URL = curl http://localhost:9085/inventory/system/30bec2b63a96

Tearing down the environment

Finally, use the following script to stop the application:

```
./scripts/stopContainers.sh
```

Deploying the application on OpenShift

Pushing your images

You can push the Docker images for the **system** and **inventory** services to your provided container registry so you can deploy these on OpenShift. Find the name of your namespace with the following command:

```
bx cr namespace-list
```

You'll see an output similar to the following:

```
Listing namespaces for account 'QuickLabs - IBM Skills Network' in registry 'us.icr.io'...

Namespace
sn-labs-johndoe
```

In this case, the namespace is **sn-labs-johndoe**. Store your namespace in a variable:

```
NAMESPACE_NAME=YOUR_NAME
```

E.g. **NAMESPACE_NAME=sn-labs-johndoe**

Verify that the variable contains your namespace name correctly:

```
echo $NAMESPACE_NAME
```

Login to the given registry with the following command:

```
bx cr login
```

Run the following commands to push the inventory images to your container registry namespace:

```
docker tag inventory:1.0-SNAPSHOT us.icr.io/$NAMESPACE_NAME/inventory-reactive:1.0-SNAPSHOT

docker push us.icr.io/$NAMESPACE_NAME/inventory-reactive:1.0-SNAPSHOT
```

Do the same for the system images:

```
docker tag system:1.0-SNAPSHOT us.icr.io/$NAMESPACE_NAME/system-reactive:1.0-SNAPSHOT

docker push us.icr.io/$NAMESPACE_NAME/system-reactive:1.0-SNAPSHOT
```

Deploy the application

Create the configuration file:

```
touch openshift.yaml
```

> [File -> Open] guide-microprofile-reactive-messaging/start/openshift.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: system-deployment
  labels:
```

```
    app: system
spec:
  selector:
    matchLabels:
      app: system
  template:
    metadata:
      labels:
        app: system
    spec:
      containers:
      - name: system-container
        image: us.icr.io/$NAMESPACE_NAME/system-reactive:1.0-SNAPSHOT
        env:
        - name: MP_MESSAGING_CONNECTOR_LIBERTY_KAFKA_BOOTSTRAP_SERVERS
          value: "kafka-service:9092"
        ports:
        - containerPort: 9083
      imagePullSecrets:
      - name: icr
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: inventory-deployment
  labels:
    app: inventory
spec:
  selector:
    matchLabels:
      app: inventory
  template:
    metadata:
      labels:
        app: inventory
    spec:
      containers:
      - name: inventory-container
        image: us.icr.io/$NAMESPACE_NAME/inventory-reactive:1.0-SNAPSHOT
        env:
        - name: MP_MESSAGING_CONNECTOR_LIBERTY_KAFKA_BOOTSTRAP_SERVERS
          value: "kafka-service:9092"
        ports:
        - containerPort: 9085
      imagePullSecrets:
      - name: icr
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kafka-deployment
  labels:
    app: kafka
spec:
  selector:
    matchLabels:
      app: kafka
  template:
    metadata:
      labels:
        app: kafka
    spec:
      containers:
      - name: kafka-container
        image: bitnami/kafka:2
        env:
        - name: KAFKA_CFG_ZOOKEEPER_CONNECT
          value: "zookeeper-service:2181"
        - name: ALLOW_PLAINTEXT_LISTENER
          value: "yes"
        - name: KAFKA_ADVERTISED_HOST_NAME
          value: "kafka-service"
        - name: KAFKA_ADVERTISED_PORT
          value: "9092"
        - name: KAFKA_CFG_ADVERTISED_LISTENERS
          value: PLAINTEXT://kafka-service:9092
        ports:
        - containerPort: 9092
      imagePullSecrets:
      - name: icr
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: zookeeper-deployment
```

```
  labels:
    app: zookeeper
spec:
  selector:
    matchLabels:
      app: zookeeper
  template:
    metadata:
      labels:
        app: zookeeper
    spec:
      containers:
      - name: zookeeper-container
        image: bitnami/zookeeper:3
        env:
        - name: ALLOW_ANONYMOUS_LOGIN
          value: "yes"
        ports:
        - containerPort: 2182
      imagePullSecrets:
      - name: icr
---
apiVersion: v1
kind: Service
metadata:
  name: system-service
spec:
  type: NodePort
  selector:
    app: system
  ports:
  - protocol: TCP
    port: 9083
    targetPort: 9083
---
apiVersion: v1
kind: Service
metadata:
  name: kafka-service
spec:
  type: NodePort
  selector:
    app: kafka
  ports:
  - protocol: TCP
    port: 9092
    targetPort: 9092
---
apiVersion: v1
kind: Service
metadata:
  name: inventory-service
spec:
  type: NodePort
  selector:
    app: inventory
  ports:
  - protocol: TCP
    port: 9085
    targetPort: 9085
---
apiVersion: v1
kind: Service
metadata:
  name: zookeeper-service
spec:
  type: NodePort
  selector:
    app: zookeeper
  ports:
  - protocol: TCP
    port: 2181
    targetPort: 2181
---
apiVersion: v1
kind: Route
metadata:
  name: inventory-route
spec:
  to:
    kind: Service
    name: inventory-service
```

Correct the image names so they are pulled from your namespace using the following command:

```
sed -i 's=$NAMESPACE_NAME='"$NAMESPACE_NAME"'=g' openshift.yaml
```

Run the following command to deploy the resources as defined in openshift.yaml:

```
oc apply -f openshift.yaml
```

Access your services

Run the following command to retrieve your URL:

```
oc get routes
```

Your app URL will be something like this: inventory-route-sn-labs-.sn-labs-user-sandbox-pr-a45631dc5778dc6371c67d206ba9ae5c-0000.tor01.containers.appdomain.cloud

Append "**/inventory/systems**" after the URL and you should see an output with the **hostname** and **systemLoad**. Be aware that it will take a couple of minutes for all of the services to start up so you may have to wait to receive the expected output.

Troubleshooting (optional)

If your application on OpenShift is not running as expected, you can run the following commands to view the logs of pods.

```
oc get pods
```

The above command should output four pods like the following :

```
NAME                                     READY   STATUS    RESTARTS   AGE
inventory-deployment-7bdd56d97d-htp2s    1/1     Running   0          7m11s
kafka-deployment-7c65bf9898-kwwcg        1/1     Running   0          7m11s
system-deployment-59df68697c-7lxtz       1/1     Running   0          7m12s
zookeeper-deployment-6b6d546fff-4zx4p    1/1     Running   0          7m10s
```

The following command will display the logs from your application pod and the output should give you information on what might be wrong.

```
oc logs -f inventory-deployment-7bdd56d97d-htp2s
```

Note: The name of your application pod might be different.

Cleaning up

Let's clean up the resources we just created. You can execute the following commands:

```
oc delete -f openshift.yaml
```

Well done

Congratulations, you now have experienced building a reactive microservice-based application using MicroProfile Reactive messaging, and experienced sending and receiving messaging between microservices within this application using Apache Kafka. You have also deployed the application on OpenShift.

CONTINUE