Contents

SCENARIO 3: Transforming an existing monolith (Part 1)

- Purpose: Showing developers and architects how Red Hat jumpstarts modernization
- Difficulty: `intermediate`
- Time: `45 minutes`

Intro

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

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

In this scenario you will learn more about WildFly Swarm, one of the runtimes included in Red Hat OpenShift Application Runtimes. WildFly Swarm is a great place to start since our application is a Java EE application, and your skills as a Java EE developer will naturally translate to the world of WildFly Swarm.

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

Goals of this scenario

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

What is WildFly Swarm?

Java EE applications are traditionally created as an **ear** or **war** archive including all dependencies and deployed in an application server. Multiple Java EE applications can and were typically deployed in the same application server. This model is well understood in the development teams and has been used over the past several years.

WildFly Swarm offers an innovative approach to packaging and running Java EE applications by packaging them with just enough of the Java EE server runtime to be able to run them directly on the JVM using **java -jar** For more details on various approaches to packaging Java applications, read this blog post.

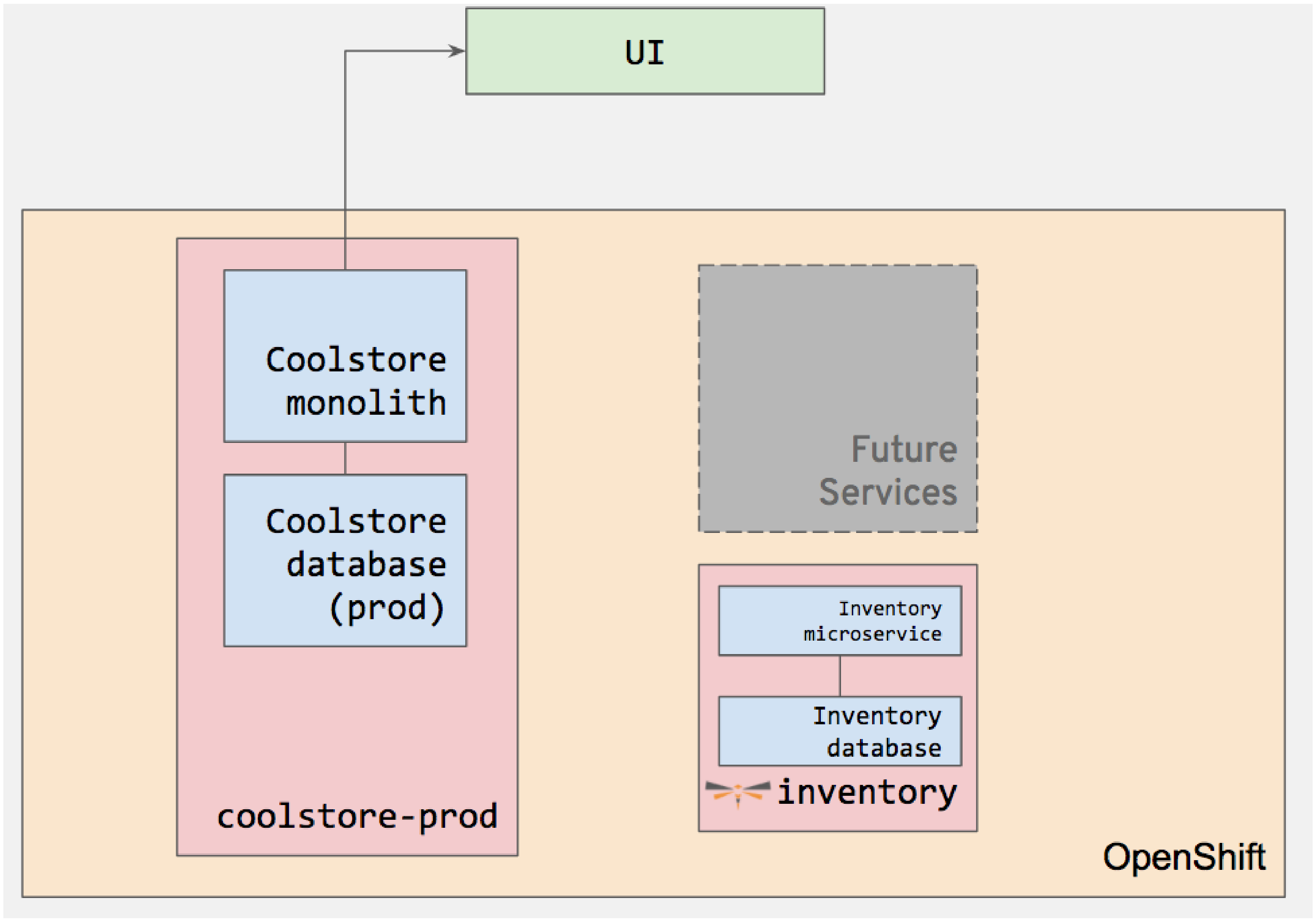


Figure 1: Logo

Figure 2: Logo

WildFly Swarm is based on WildFly and it's compatible with Eclipse MicroProfile, which is a community effort to standardized the subset of Java EE standards such as JAX-RS, CDI and JSON-P that are useful for building microservices applications.

Since WildFly Swarm is based on Java EE standards, it significantly simplifies refactoring existing Java EE application to microservices and allows much of existing code-base to be reused in the new services.

Setup for Exercise

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

```
#!/usr/bin/env bash

cd ${HOME}/projects/inventory
git pull --quiet
```

Examine the sample project

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

1. Examine the Maven project structure.

> Click on the `tree` command below to automatically copy it into the terminal and execute it

```
tree
```

This is a minimal Java EE project with support for JAX-RS for building RESTful services and JPA for connecting to a database. JAX-RS is one of Java EE standards that uses Java annotations to simplify the development of RESTful web services. Java Persistence API (JPA) is another Java EE standard that provides Java developers with an object/relational mapping facility for managing relational data in Java applications.

This project currently contains no code other than the main class for exposing a single RESTful application defined in `src/main/java/com/redhat/coolstore/rest/RestApplication.java`.

Run the Maven build to make sure the skeleton project builds successfully. You should get a **BUILD SUCCESS** message in the logs, otherwise the build has failed.

> Make sure to run the **package** Maven goal and not **install**. The latter would download a lot more dependencies and do things you don't need yet!

```
mvn clean package
```

You should see a **BUILD SUCCESS** in the logs.

Once built, the resulting *jar* is located in the **target** directory:

```
ls target/*.jar
```

The listed jar archive, **inventory-1.0.0-SNAPSHOT-swarm.jar** , is an uber-jar with all the dependencies required packaged in the *jar* to enable running the application with **java -jar**. WildFly Swarm also creates a *war* packaging as a standard Java EE web app that could be deployed to any Java EE app server (for example, JBoss EAP, or its upstream WildFly project).

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

Create Inventory Domain

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

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

Create a new Java class named `Inventory.java` in `com.redhat.coolstore.model` package with the following code, identical to the monolith code (click **Copy To Editor** to create the class):

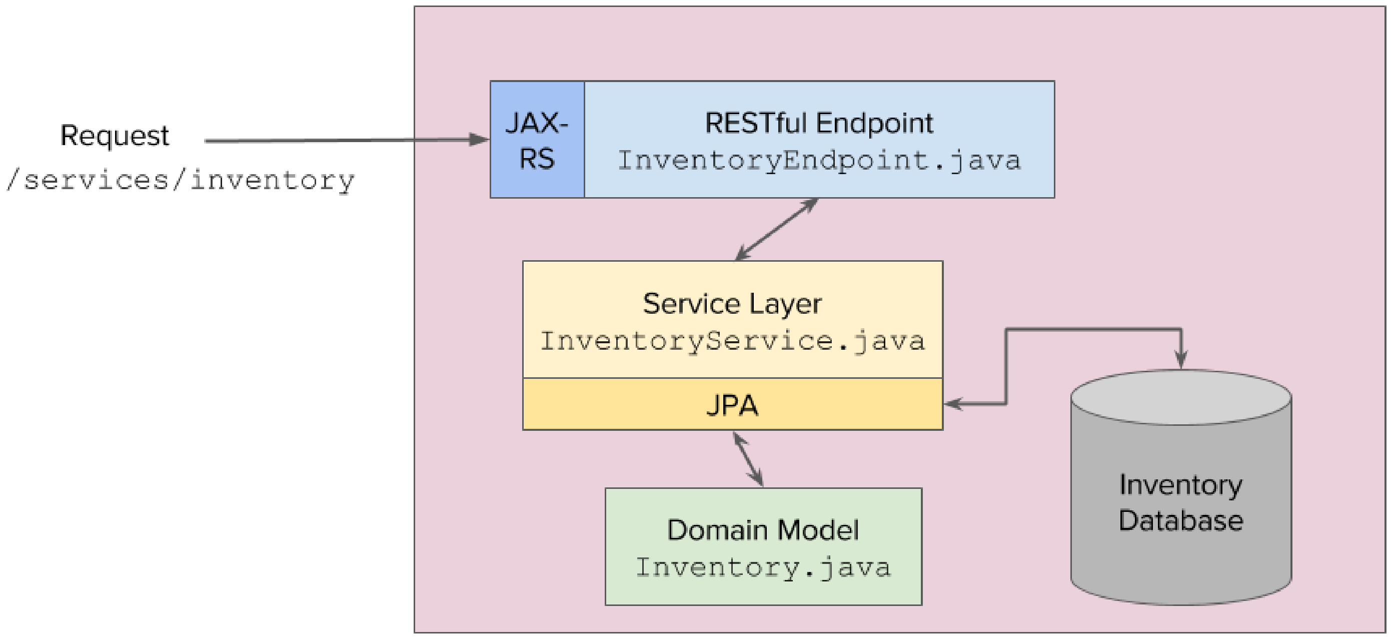


Figure 3: Inventory RESTful Service

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

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.UniqueConstraint;
import java.io.Serializable;

@Entity
@Table(name = "INVENTORY", uniqueConstraints = @UniqueConstraint(columnNames = "itemId"))
public class Inventory implements Serializable {

    private static final long serialVersionUID = -7304814269819778382L;

    @Id
    private String itemId;


    private String location;


    private int quantity;


    private String link;

    public Inventory() {

    }

    public Inventory(String itemId, int quantity, String location, String link) {
        super();
        this.itemId = itemId;
        this.quantity = quantity;
        this.location = location;
        this.link = link;
    }

    public String getItemId() {
        return itemId;
    }

    public void setItemId(String itemId) {
        this.itemId = itemId;
    }

    public String getLocation() {
        return location;
    }

    public void setLocation(String location) {
        this.location = location;
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

    public String getLink() {
        return link;
    }

    public void setLink(String link) {
        this.link = link;
    }

    @Override
    public String toString() {
        return "Inventory [itemId=" + itemId + ", availability=" + quantity + "/" + location + " lin
    }
}
```

Review the **Inventory** domain model and note the JPA annotations on this class. **@Entity** marks the class as a JPA entity, **@Table** customizes the table creation process by defining a table name and database constraint and **@Id** marks the primary key for the table.

WildFly Swarm configuration is done to a large extent through detecting the intent of the developer and automatically adding the required dependencies configurations to make sure it can get out of the way and developers can be productive with their code rather than Googling for configuration snippets. As an example, configuration database access with JPA is done by adding the JPA *fraction* and a database driver to the `pom.xml`, and then configuring the database connection details in `src/main/resources/project-stages.yml`.

Examine `src/main/resources/META-INF/persistence.xml` to see the JPA datasource configuration for this project. Also note that the configurations uses `src/main/resources/META-INF/load.sql` to import initial data into the database.

Examine `src/main/resources/project-stages.yml` to see the database connection details. An in-memory H2 database is used in this scenario for local development and in the following steps will be replaced with a PostgreSQL database with credentials coming from an OpenShift *secret*. Be patient! More on that later.

Build and package the Inventory service using Maven to make sure you code compiles:

```
mvn clean package
```

If builds successfully, continue to the next step to create a new service.

Create Inventory Service

In this step we will mirror the abstraction of a *service* so that we can inject the Inventory *service* into various places (like a RESTful resource endpoint) in the future. This is the same approach that our monolith uses, so we can re-use this idea again. Create an **InventoryService** class in the `com.redhat.coolstore.service` package by clicking **Copy To Editor** with the below code:

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


import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

import com.redhat.coolstore.model.Inventory;

import java.util.Collection;
import java.util.List;

@Stateless
public class InventoryService {

    @PersistenceContext
    private EntityManager em;

    public InventoryService() {

    }

    public boolean isAlive() {
        return em.createQuery("select 1 from Inventory i")
                .setMaxResults(1)
                .getResultList().size() == 1;
    }
    public Inventory getInventory(String itemId) {
        return em.find(Inventory.class, itemId);
    }

    public List<Inventory> getAllInventory() {
        Query query = em.createQuery("SELECT i FROM Inventory i");
        return query.getResultList();
    }
}
```

Review the **InventoryService** class and note the EJB and JPA annotations on this class:

- **@Stateless** marks the class as a *Stateless EJB*, and its name suggests, means that instances of the class do not maintain state, which means they can be created and destroyed at will by the management system, and be re-used by multiple clients without instantiating multiple copies of the bean. Because they can support multiple clients, stateless EJBs can offer better scalability for applications that require large numbers of clients.
- **@PersistenceContext** objects are created by the Java EE server based on the JPA definition in `persistence.xml` that we examined earlier, so to use it at runtime it is injected by this annotation and can be used to issue queries against the underlying database backing the **Inventory** entities.

This service class exposes a few APIs that we'll use later:

- **isAlive()** - A simple health check to determine if this service class is ready to accept requests. We will use this later on when defining OpenShift health checks.
- **getInventory()** and **getAllInventory()** are APIs used to query for one or all of the stored **Inventory* entities. We'll use this later on when implementing a RESTful endpoint.

Re-Build and package the Inventory service using Maven to make sure your code compiles:

```
mvn clean package
```

You should see a **BUILD SUCCESS** in the build logs. If builds successfully, continue to the next step to create a new RESTful endpoint that uses this service.

Create RESTful Endpoints

WildFly Swarm uses JAX-RS standard for building REST services. Create a new Java class named `InventoryEndpoint.java` in `com.redhat.coolstore.rest` package with the following content by clicking on *Copy to Editor*:

```
package com.redhat.coolstore.rest;

import java.io.Serializable;
import java.util.List;

import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
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 com.redhat.coolstore.model.Inventory;
import com.redhat.coolstore.service.InventoryService;

@RequestScoped
@Path("/inventory")
public class InventoryEndpoint implements Serializable {

    private static final long serialVersionUID = -7227732980791688773L;

    @Inject
    private InventoryService inventoryService;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Inventory> getAll() {
        return inventoryService.getAllInventory();
    }

    @GET
    @Path("{itemId}")
    @Produces(MediaType.APPLICATION_JSON)
    public Inventory getAvailability(@PathParam("itemId") String itemId) {
        return inventoryService.getInventory(itemId);
    }

}
```

The above REST services defines two endpoints:

- `/services/inventory` that is accessible via **HTTP GET** which will return all known product Inventory entities as JSON
- `/services/inventory/<id>` that is accessible via **HTTP GET** at for example **/services/inventory/329299** with the last path parameter being the product id which we want to check its inventory status.

The code also injects our new **InventoryService** using the [CDI @Inject](https://docs.oracle.com/javaee/7/tutorial/partcdi.h annotation, which gives us a runtime handle to the service we defined in the previous steps that we can use to query the database when the RESTful APIs are invoked.

Build and package the Inventory service again using Maven:

```
mvn clean package
```

You should see a **BUILD SUCCESS** in the build logs.

Test Locally

Using the WildFly Swarm maven plugin (predefined in `pom.xml`), you can conveniently run the application locally and test the endpoint.

```
mvn wildfly-swarm:run
```

> As an uber-jar, it could also be run with `java -jar target/inventory-1.0-SNAPSHOT-swarm.jar` but you don't need to do this now

Once the application is done initializing you should see:

```
INFO  [org.wildfly.swarm] (main) WFSWARM99999: WildFly Swarm is Ready
```

Running locally using `wildfly-swarm:run` will use an in-memory database with default credentials. In a production application you will use an external source for credentials using an OpenShift *secret* in later steps, but for now this will work for development and testing.

3. Test the application

To test the running application, click on the **Local Web Browser** tab in the console frame of this browser window. This will open another tab or window of your browser pointing to port 8080 on your client.

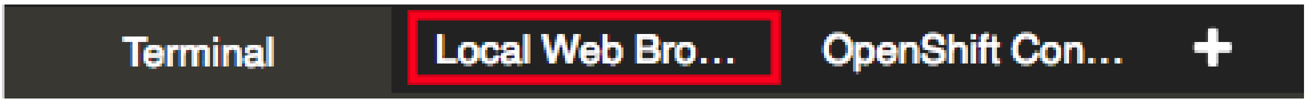


Figure 4: Local Web Browser Tab

> or use this at

`http://localhost:8080` link.

You should now see a html page that looks like this

This is a simple webpage that will access the inventory *every 2 seconds* and refresh the table of product inventories.

You can also click the **Fetch Inventory** button to force it to refresh at any time.

To see the raw JSON output using `curl`, you can open an new terminal window by clicking on the plus (+) icon on the terminal toolbar and then choose **Open New Terminal**. You can also click on the following command to automatically open a new terminal and run the test:

```
curl http://localhost:8080/services/inventory/329299 ; echo
```

You would see a JSON response like this:

```
{"itemId":"329299","location":"Raleigh","quantity":736,"link":"http://maps.google.com/?q=Raleigh"}
```

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

4. Stop the application

Before moving on, click in the first terminal window where WildFly Swarm is running and then press `CTRL-C` to stop the running application! (or click this command to issue a `CTRL-C` for you: `clear`)

You should see something like:

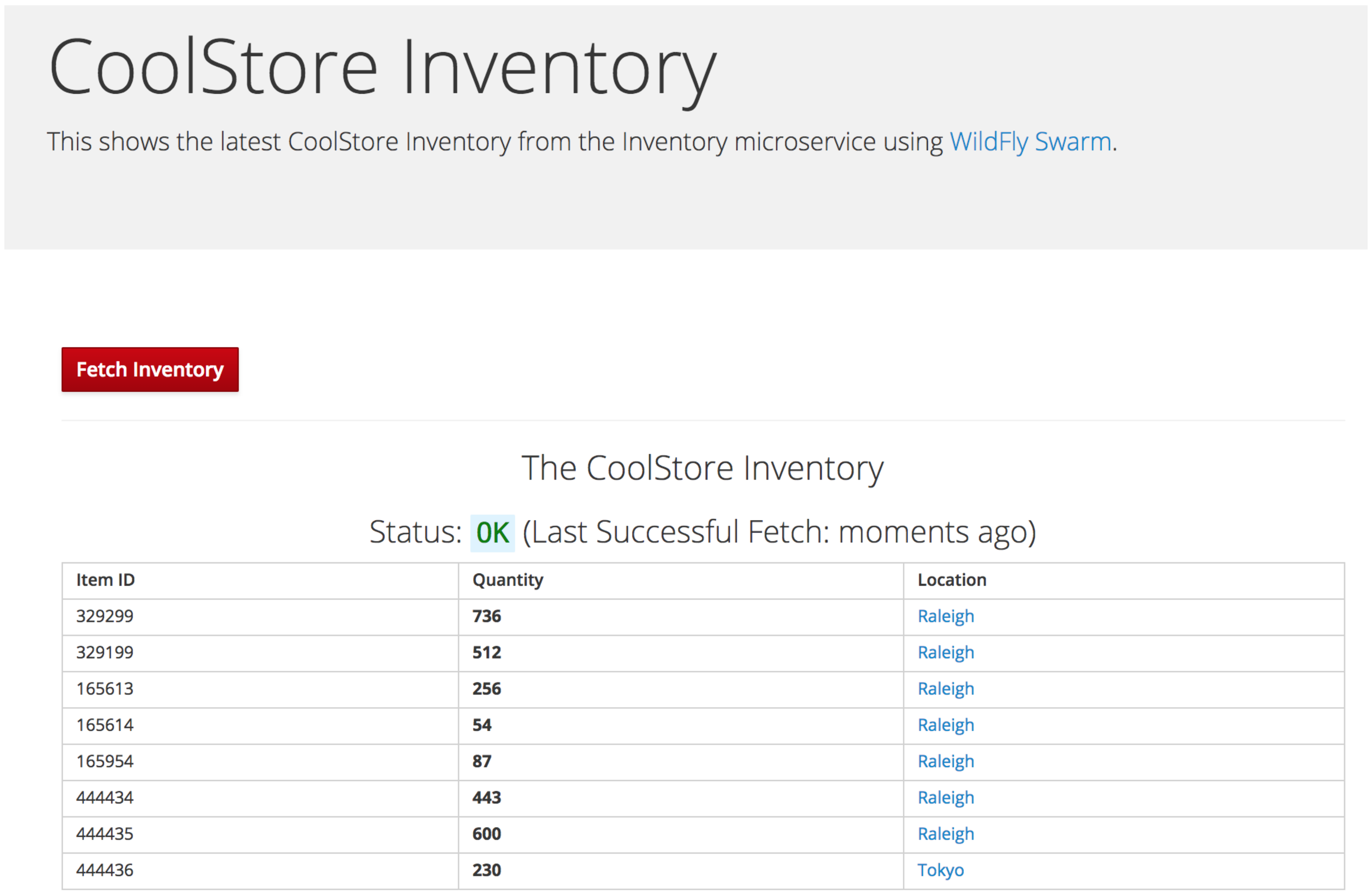
```
WFLYSRV0028: Stopped deployment inventory-1.0.0-SNAPSHOT.war (runtime-name: inventory-1.0.0-SNAPSHOT
```

This indicates the application is stopped.

Congratulations

You have now successfully created your first microservice using WildFly Swarm and implemented a basic RESTful API on top of the Inventory database. Most of the code is the same as was found in the monolith, demonstrating how easy it is to migrate existing monolithic Java EE applications to microservices using WildFly Swarm.

In next steps of this scenario we will deploy our application to OpenShift Container Platform and then start adding additional features to take care of various aspects of cloud native microservice development.


CoolStore Inventory

This shows the latest CoolStore Inventory from the Inventory microservice using WildFly Swarm.

The CoolStore Inventory

Status: OK (Last Successful Fetch: moments ago)

Item ID	Quantity	Location
329299	**736**	Raleigh
329199	**512**	Raleigh
165613	**256**	Raleigh
165614	**54**	Raleigh
165954	**87**	Raleigh
444434	**443**	Raleigh
444435	**600**	Raleigh
444436	**230**	Tokyo

Figure 5: App

Create OpenShift Project

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

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

1. Create project

Create a new project for the *inventory* service:

```
oc new-project inventory --display-name="CoolStore Inventory Microservice Application"
```

3. Open the OpenShift Web Console

You should be familiar with the OpenShift Web Console by now! Click on the “OpenShift Console” tab:

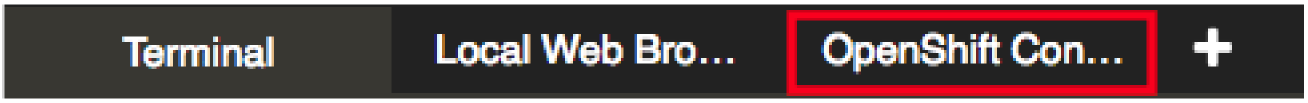
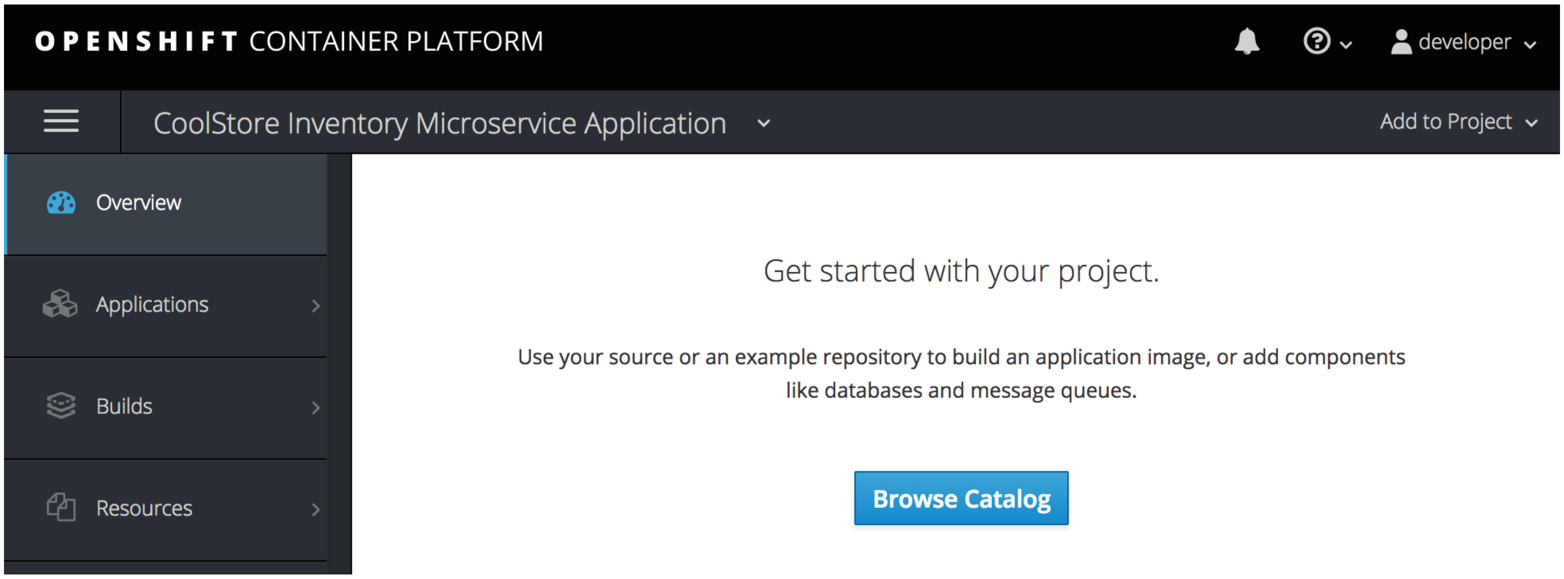


Figure 6: OpenShift Console Tab

And navigate to the new *inventory* project overview page (or use this quick link at

`https://$OPENSHIFT_MASTER/console/project/inventory/`


OPENSHIFT CONTAINER PLATFORM
developer
CoolStore Inventory Microservice Application
Add to Project
Overview
Applications
Builds
Resources
Browse Catalog

Get started with your project.

Use your source or an example repository to build an application image, or add components like databases and message queues.

Figure 7: Web Console Overview

There’s nothing there now, but that’s about to change.

Deploy to OpenShift

Let’s deploy our new inventory microservice to OpenShift!

1. Deploy the Database

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

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

> **NOTE:** If you change the username and password you also need to update `src/main/fabric8/credential-secret.` which contains the credentials used when deploying to OpenShift.

This will deploy the database to our new project. Wait for it to complete:

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

2. Build and Deploy

Red Hat OpenShift Application Runtimes includes a powerful maven plugin that can take an existing WildFly Swarm application and generate the necessary Kubernetes configuration. You can also add additional config, like `src/main/fabric8/inventory-deployment.yml` which defines the deployment characteristics of the app (in this case we declare a few environment variables which map our credentials stored in the secrets file to the application), but OpenShift supports a wide range of Deployment configuration options for apps).

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

```
mvn clean fabric8:deploy -Popenshift
```

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.

> **NOTE**: If you see messages like `Current reconnect backoff is 2000 milliseconds (T1)` you can safely ignore them, it is a known issue and is harmless.

After the maven build finishes it will take less than a minute for the application to become available. To verify that everything is started, run the following command and wait for it complete successfully:

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

> **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.

3. Access the application running on OpenShift

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

`http://inventory-inventory.$ROUTE_SUFFIX` to access the sample UI.

> You can also access the application through the link on the OpenShift Web Console Overview page.

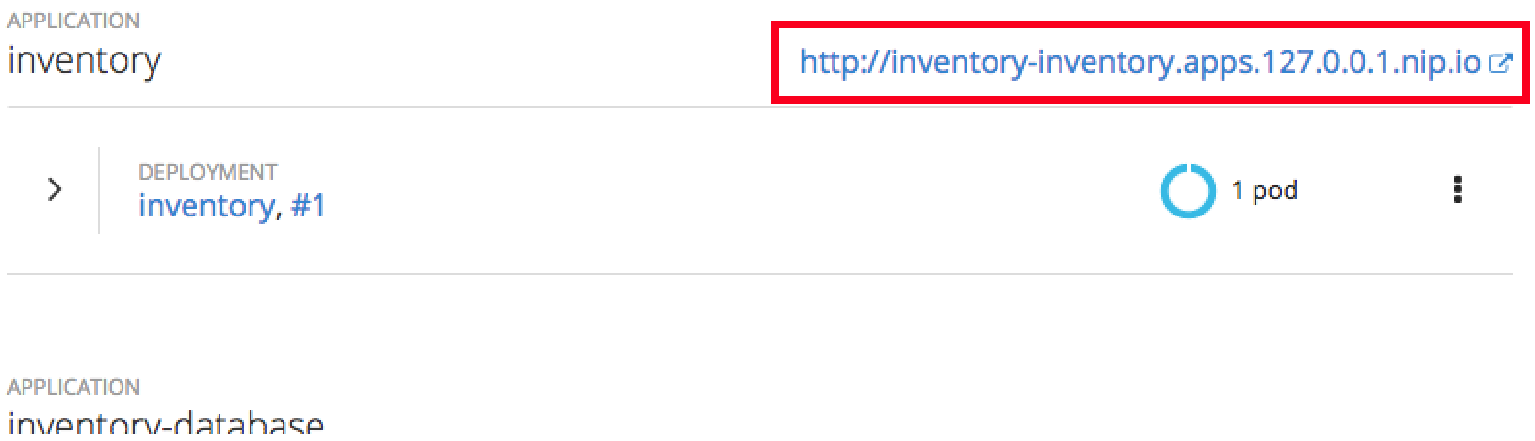


Figure 8: Overview link

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

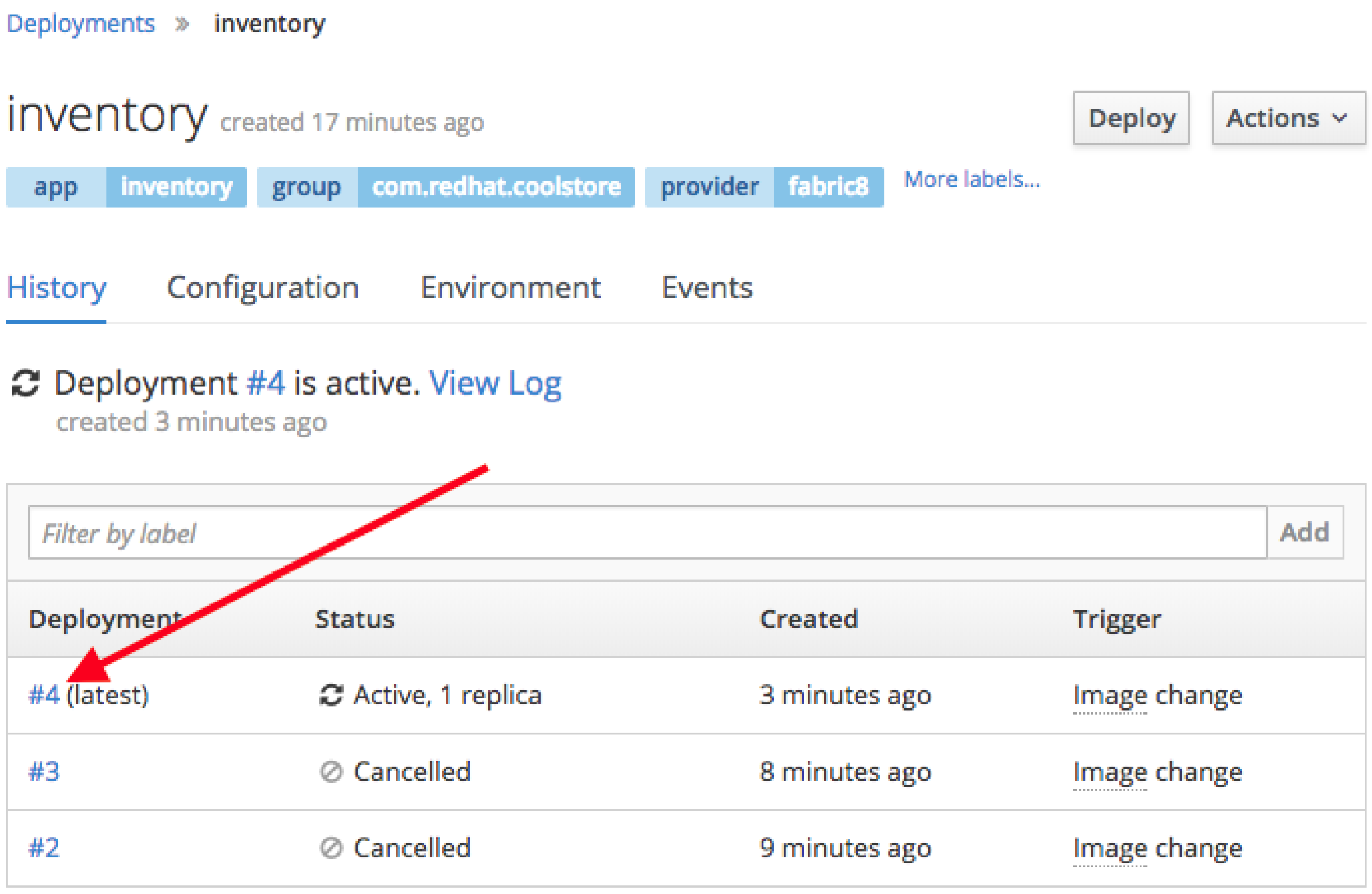
The UI will refresh the inventory table every 2 seconds, as before.

Back on the OpenShift console, Navigate to *Applications* -> *Deployments* -> `inventory` (or just click this link at

`https://$OPENSHIFT_MASTER/console/project/inventory/browse/dc/inventory?tab=history)` and then click on the top-most `(latest)` deployment in the listing (most likely #1 or #2):

Notice OpenShift is warning you that the inventory application has no health checks:

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



Deployments » inventory

inventory created 17 minutes ago

Deploy | Actions

app inventory | group com.redhat.coolstore | provider fabric8 | More labels...

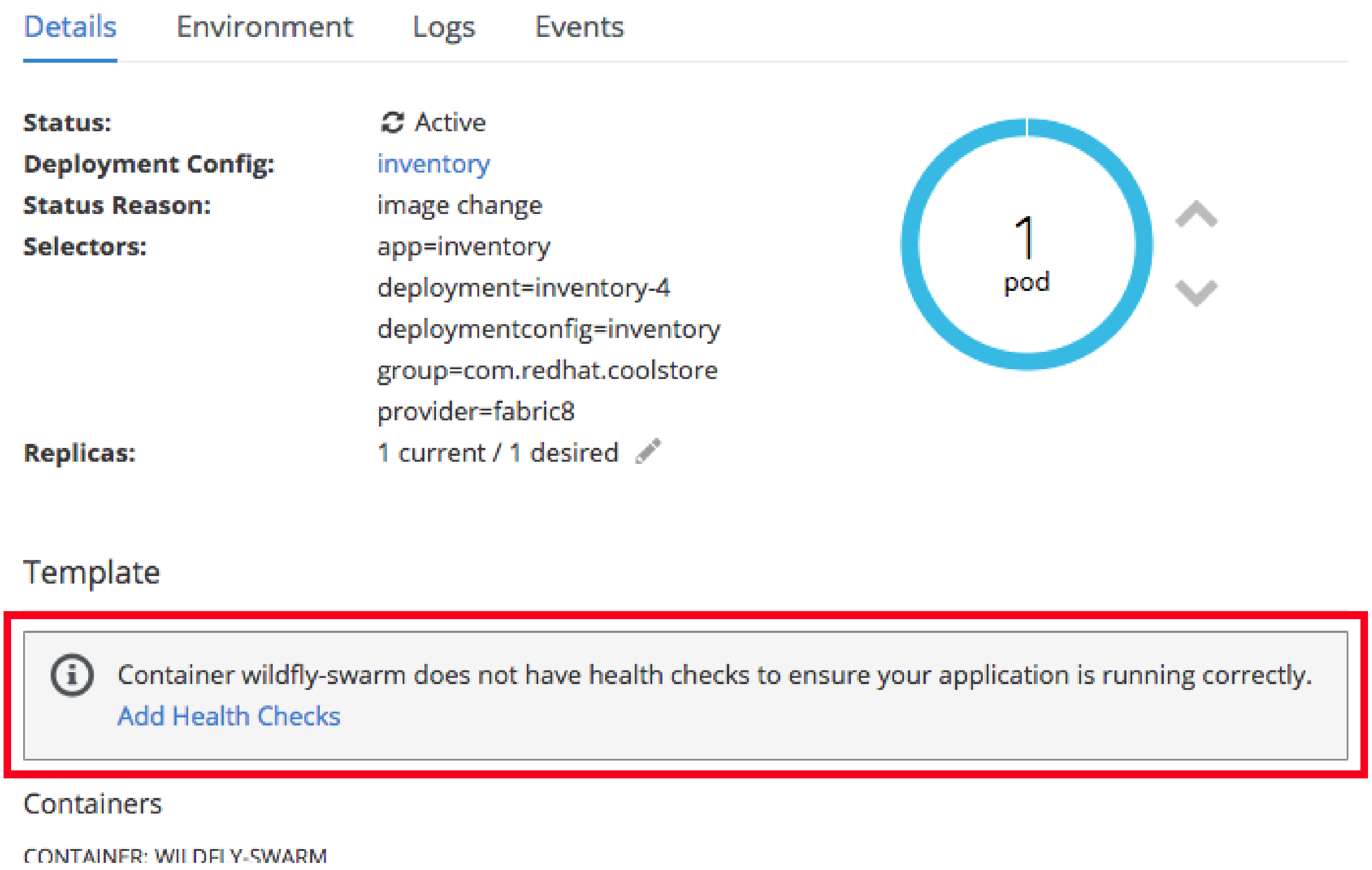
History | Configuration | Environment | Events

Deployment #4 is active. View Log
created 3 minutes ago

Filter by label | Add

Deployment	Status	Created	Trigger
#4 (latest)	Active, 1 replica	3 minutes ago	Image change
#3	Cancelled	8 minutes ago	Image change
#2	Cancelled	9 minutes ago	Image change

Figure 9: Overview link



Status: Active
Deployment Config: inventory
Status Reason: image change
Selectors: app=inventory
deployment=inventory-4
deploymentconfig=inventory
group=com.redhat.coolstore
provider=fabric8
Replicas: 1 current / 1 desired

Template

Container wildfly-swarm does not have health checks to ensure your application is running correctly.
Add Health Checks

Containers

CONTAINER: WILDFLY-SWARM

Figure 10: Health Check Warning

Add Health Check Fraction

What is a Fraction?

WildFly Swarm is defined by an unbounded set of capabilities. Each piece of functionality is called a fraction. Some fractions provide only access to APIs, such as JAX-RS or CDI; other fractions provide higher-level capabilities, such as integration with RHSSO (Keycloak).

The typical method for consuming WildFly Swarm fractions is through Maven coordinates, which you add to the pom.xml file in your application. The functionality the fraction provides is then packaged with your application into an *Uberjar*. An uberjar is a single Java .jar file that includes everything you need to execute your application. This includes both the runtime components you have selected, along with the application logic.

What is a Health Check?

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

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

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

** 2. Add `monitor` fraction**

First, open the `pom.xml` file.

WildFly Swarm includes the `monitor` fraction which automatically adds health check infrastructure to your application when it is included as a fraction in the project. Open the file to insert the new dependencies into the `pom.xml` file at the `<!-- Add monitor fraction-->` marker:

```
<dependency>
    <groupId>org.wildfly.swarm</groupId>
    <artifactId>monitor</artifactId>
</dependency>
```

By adding the `monitor` fraction, Fabric8 will automatically add a *readinessProbe* and *livenessProbe* to the OpenShift *DeploymentConfig*, published at `/health`, once deployed to OpenShift. But you still need to implement the logic behind the health check, which you'll do next.

Define Health Check Endpoint

We are now ready to define the logic of our health check endpoint.

1. Create empty Java class

The logic will be put into a new Java class.

Click this link to create and open the file which will contain the new class: `src/main/java/com/redhat/coolstore/rest/H`

Methods in this new class will be annotated with both the JAX-RS annotations as well as WildFly Swarm's `@Health` annotation, indicating it should be used as a health check endpoint.

2. Add logic

Next, let's fill in the class by creating a new RESTful endpoint which will be used by OpenShift to probe our services.

Click on **Copy To Editor** below to implement the logic.

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

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;

import com.redhat.coolstore.service.InventoryService;
import org.wildfly.swarm.health.Health;
import org.wildfly.swarm.health.HealthStatus;

@Path("/infra")
public class HealthChecks {

    @Inject
    private InventoryService inventoryService;

    @GET
    @Health
    @Path("/health")
    public HealthStatus check() {

        if (inventoryService.isAlive()) {
            return HealthStatus.named("service-state").up();
        } else {
            return HealthStatus.named("service-state").down();
        }
    }
}
```

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

With our new health check in place, we'll need to build and deploy the updated application in the next step.

Re-Deploy to OpenShift

1. Rebuild and re-deploy

With our health check in place, lets rebuild and redeploy using the same command as before:

```
mvn fabric8:undeploy clean fabric8:deploy -Popenshift
```

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

During build and deploy, you'll notice WildFly Swarm adding in health checks for you:

```
[INFO] F8: wildfly-swarm-health-check: Adding readiness probe on port 8080, path='/health', scheme='
[INFO] F8: wildfly-swarm-health-check: Adding liveness probe on port 8080, path='/health', scheme='H
```

To verify that everything is started, run the following command and wait for it report `replication controller "inventory-xxxx" successfully rolled out`

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

Once the project is deployed, you should be able to access the health check logic at the `/health` endpoint using a simple *curl* command. This is the same API that OpenShift will repeatedly poll to determine application health.

Click here to try it (you may need to try a few times until the project is fully deployed):

```
curl http://inventory-inventory.[[HOST_SUBDOMAIN]]-80-[[KATACODA_HOST]].environments.katacoda.com/he
```

You should see a JSON response like:

```
{"checks": [
{"id":"service-state","result":"UP"}],
"outcome": "UP"
}
```

You can see the definition of the health check from the perspective of OpenShift:

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

You should see:

```
    Liveness:   http-get http://:8080/health delay=180s timeout=1s period=10s #success=1 #failure=3
    Readiness:  http-get http://:8080/health delay=10s timeout=1s period=10s #success=1 #failure=3
```

2. Adjust probe timeout

The various timeout values for the probes can be configured in many ways. Let's tune the *liveness probe* initial delay so that we don't have to wait 3 minutes for it to be activated. Use the **oc** command to tune the probe to wait 20 seconds before starting to poll the probe:

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

And verify it's been changed (look at the delay= value for the Liveness probe):

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

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

You can also edit health checks from the OpenShift Web Console, for example click on this link at

`https://$OPENSHIFT_MASTER/console/project/inventory/edit/health-checks?kind=DeploymentConfig&name=ir`
to access the health check edit page for the Inventory deployment.

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

Exercise Health Check

From the OpenShift Web Console overview page, click on the route link to open the sample application UI:

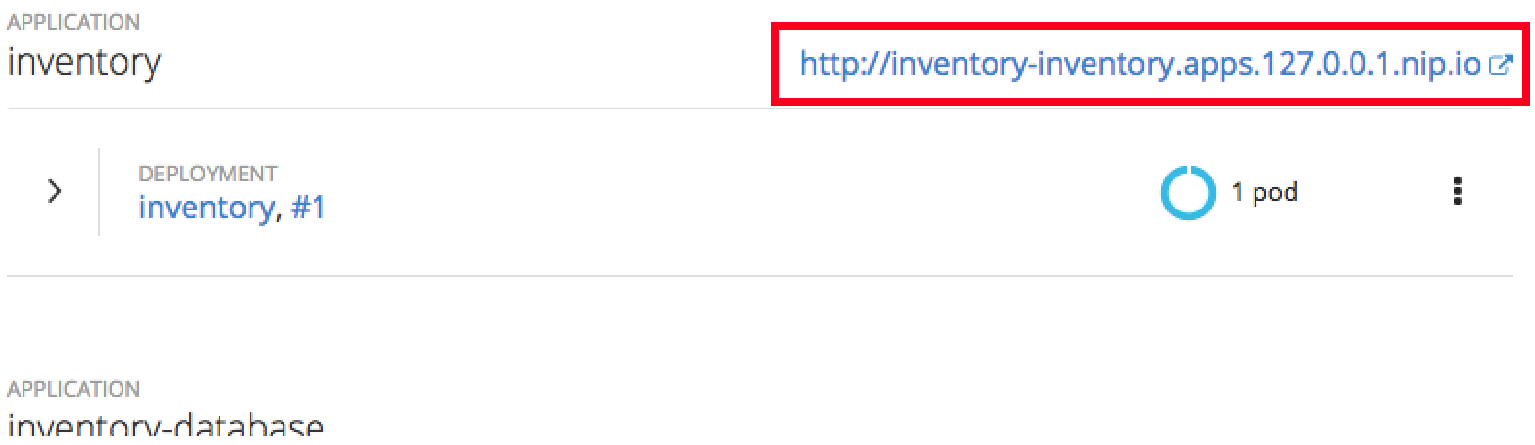


Figure 11: Route Link

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

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

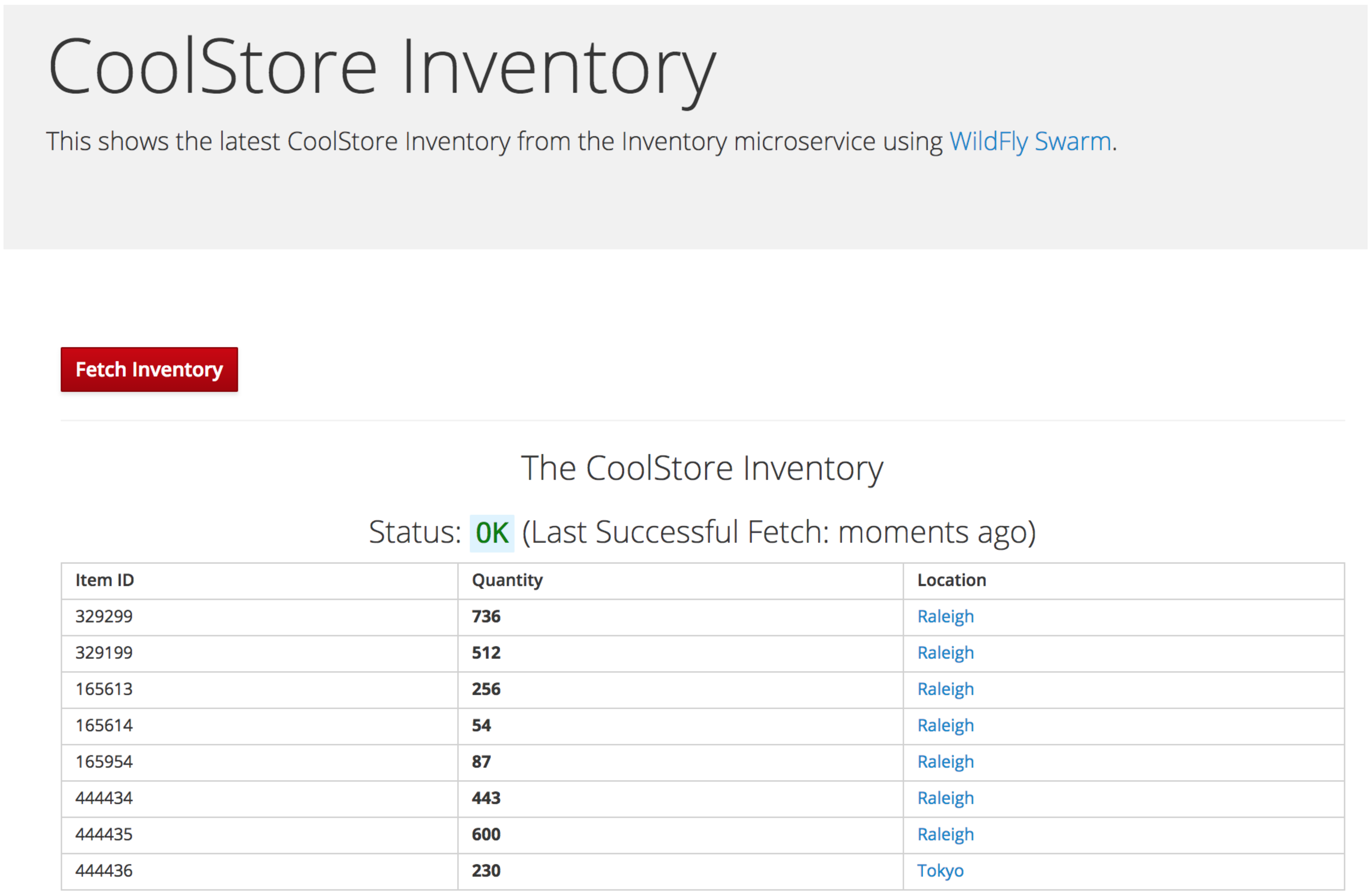
Now you will corrupt the service and cause its health check to start failing. To simulate the app crasing, let's kill the underlying service so it stops responding. Execute:

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

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

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

At this point, return to the OpenShift web console and click on the *Overview* tab for the project. Notice that the dark blue circle has now gone light blue, indicating the application is failing its *liveness probe*:


CoolStore Inventory

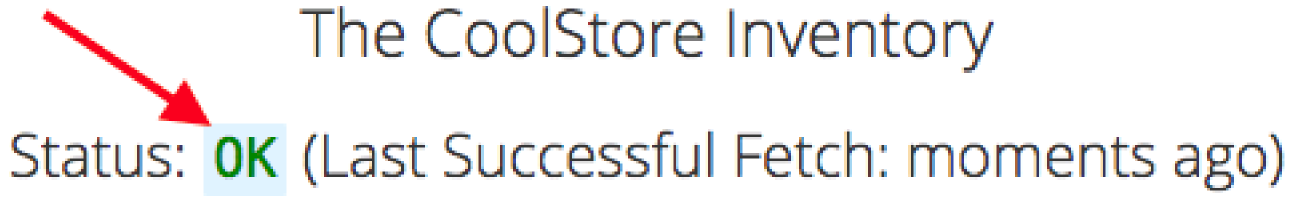
This shows the latest CoolStore Inventory from the Inventory microservice using WildFly Swarm.

The CoolStore Inventory

Status: OK (Last Successful Fetch: moments ago)

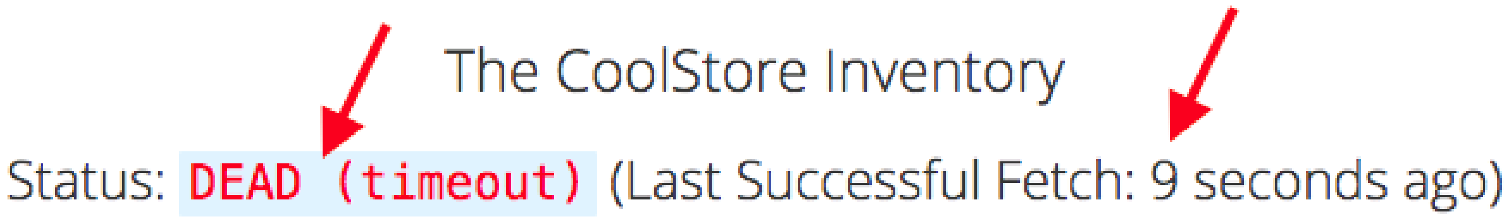
Item ID	Quantity	Location
329299	**736**	Raleigh
329199	**512**	Raleigh
165613	**256**	Raleigh
165614	**54**	Raleigh
165954	**87**	Raleigh
444434	**443**	Raleigh
444435	**600**	Raleigh
444436	**230**	Tokyo

Figure 12: App UI


The CoolStore Inventory

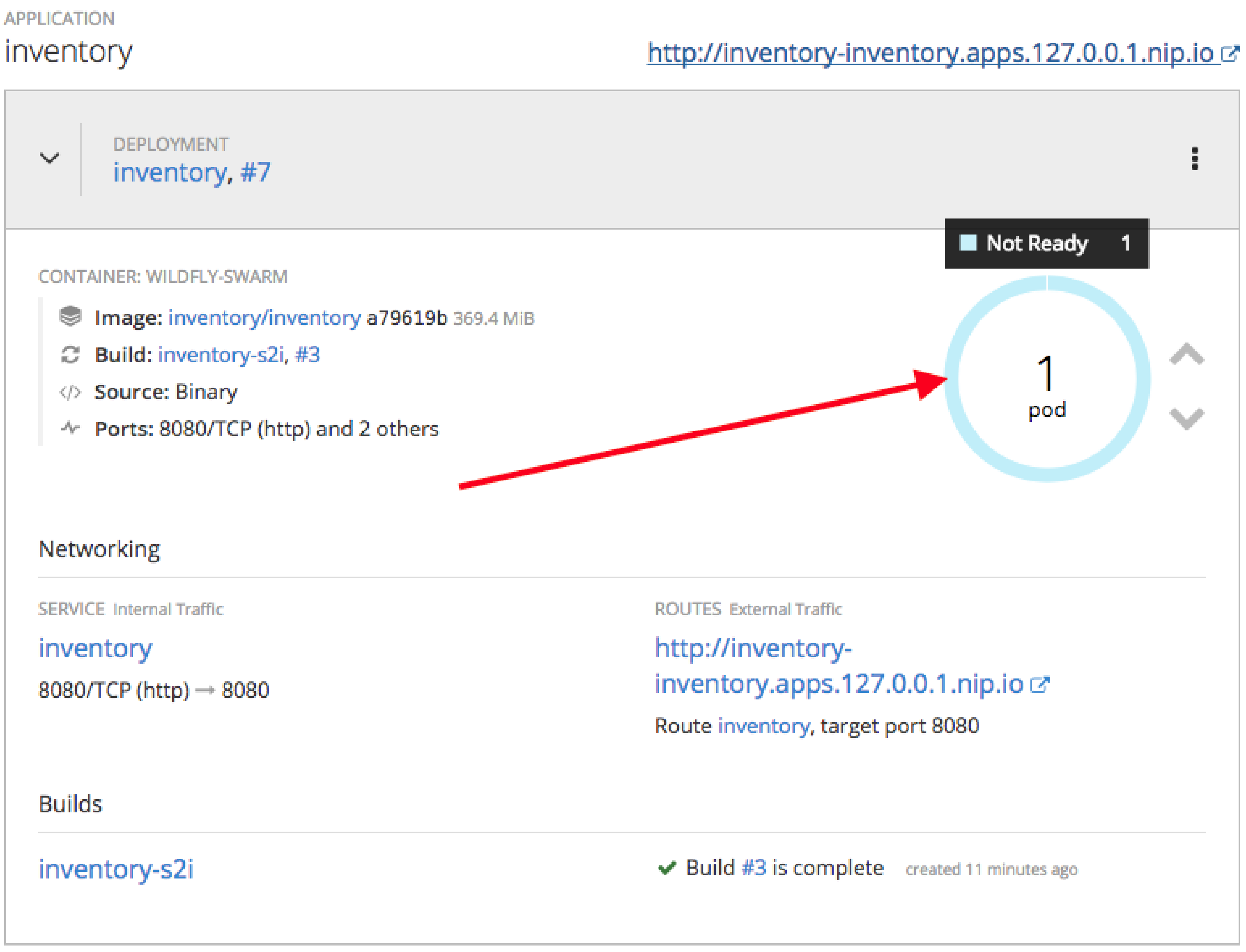
Status: OK (Last Successful Fetch: moments ago)

Figure 13: Greeting


The CoolStore Inventory

Status: DEAD (timeout) (Last Successful Fetch: 9 seconds ago)

Figure 14: Greeting



APPLICATION

inventory

http://inventory-inventory.apps.127.0.0.1.nip.io

DEPLOYMENT

inventory, #7

CONTAINER: WILDFLY-SWARM

Image: inventory/inventory a79619b 369.4 MiB

Build: inventory-s2i, #3

Source: Binary

Ports: 8080/TCP (http) and 2 others

Networking

SERVICE Internal Traffic

inventory

8080/TCP (http) → 8080

ROUTES External Traffic

http://inventory-inventory.apps.127.0.0.1.nip.io

Route inventory, target port 8080

Builds

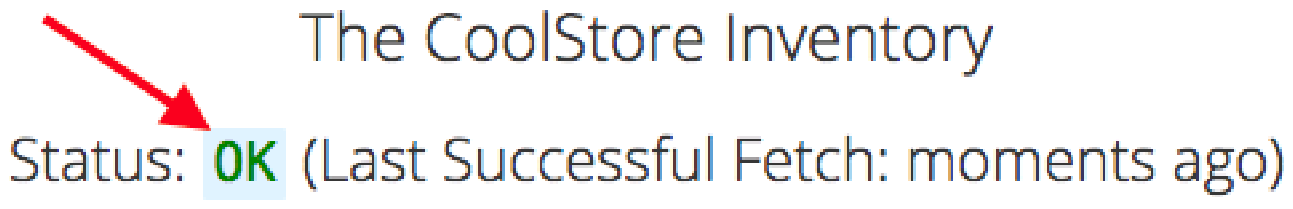
inventory-s2i

✓ Build #3 is complete created 11 minutes ago

Figure 15: Not Ready

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

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



The CoolStore Inventory

Status: OK (Last Successful Fetch: moments ago)

Figure 16: Greeting

Summary

In this scenario you learned a bit more about what WildFly Swarm is, and how it can be used to create modern Java microservice-oriented applications.

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

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

WildFly Swarm is one of many components of Red Hat OpenShift Application Runtimes. In the next scenario you'll use Spring Boot, another popular framework, to implement additional microservices. Let's go!