Contents

SCENARIO 4: Transforming an existing monolith (Part 2) . . . 1
Intro . . . 1
What is Spring Framework? . . . 1
Aggregate microservices calls . . . 3
Setup for Exercise . . . 3
Examine the sample project . . . 3
Congratulations . . . 5
Create Domain Objects . . . 6
Creating a test. . . . 6
Implement the database repository . . . 7
Congratulations . . . 8
Create Catalog Service . . . 8
Congratulations . . . 11
Before moving on . . . 13
Congratulations! . . . 13
Get inventory data . . . 13
Extending the test . . . 13
Congratulations . . . 15
Create a fallback for inventory . . . 15
Congratulations . . . 17
Test Locally . . . 17
Congratulations . . . 19
Create the OpenShift project . . . 19
Deploy to OpenShift . . . 19
Congratulations! . . . 21
Strangling the monolith . . . 21
Congratulations! . . . 23
Summary . . . 24

SCENARIO 4: Transforming an existing monolith (Part 2)

- Purpose: Showing developers and architects how Red Hat jumpstarts modernization
- Difficulty: `intermediate`
- Time: `60-70 minutes`

Intro

In the previous scenarios, you learned how to take an existing monolithic app and refactor a single *inventory* service using WildFly Swarm. Since WildFly Swarm is using Java EE much of the technology from the monolith can be reused directly, like JPA and JAX-RS. The previous scenario resulted in you creating an inventory service, but so far we haven't started *strangling* the monolith. That is because the inventory service is never called directly by the UI. It's a backend service that is only used only by other backend services. In this scenario, you will create the catalog service and the catalog service will call the inventory service. When you are ready, you will change the route to tie the UI calls to new service.

To implement this, we are going to use the Spring Framework. The reason for using Spring for this service is to introduce you to Spring Development, and how Red Hat OpenShift Application Runtimes helps to make Spring development on Kubernetes easy. In real life, the reason for choosing Spring vs. WF Swarm mostly depends on personal preferences, like existing knowledge, etc. At the core Spring and Java EE are very similar.

The goal is to produce something like:

What is Spring Framework?

Spring is one of the most popular Java Frameworks and offers an alternative to the Java EE programming model. Spring is also very popular for building applications based on microservices architectures. Spring Boot is a popular tool in the Spring ecosystem that helps with organizing and using 3rd-party libraries together with Spring and also provides a mechanism for boot strapping embeddable runtimes, like Apache Tomcat. Bootable applications (sometimes also called

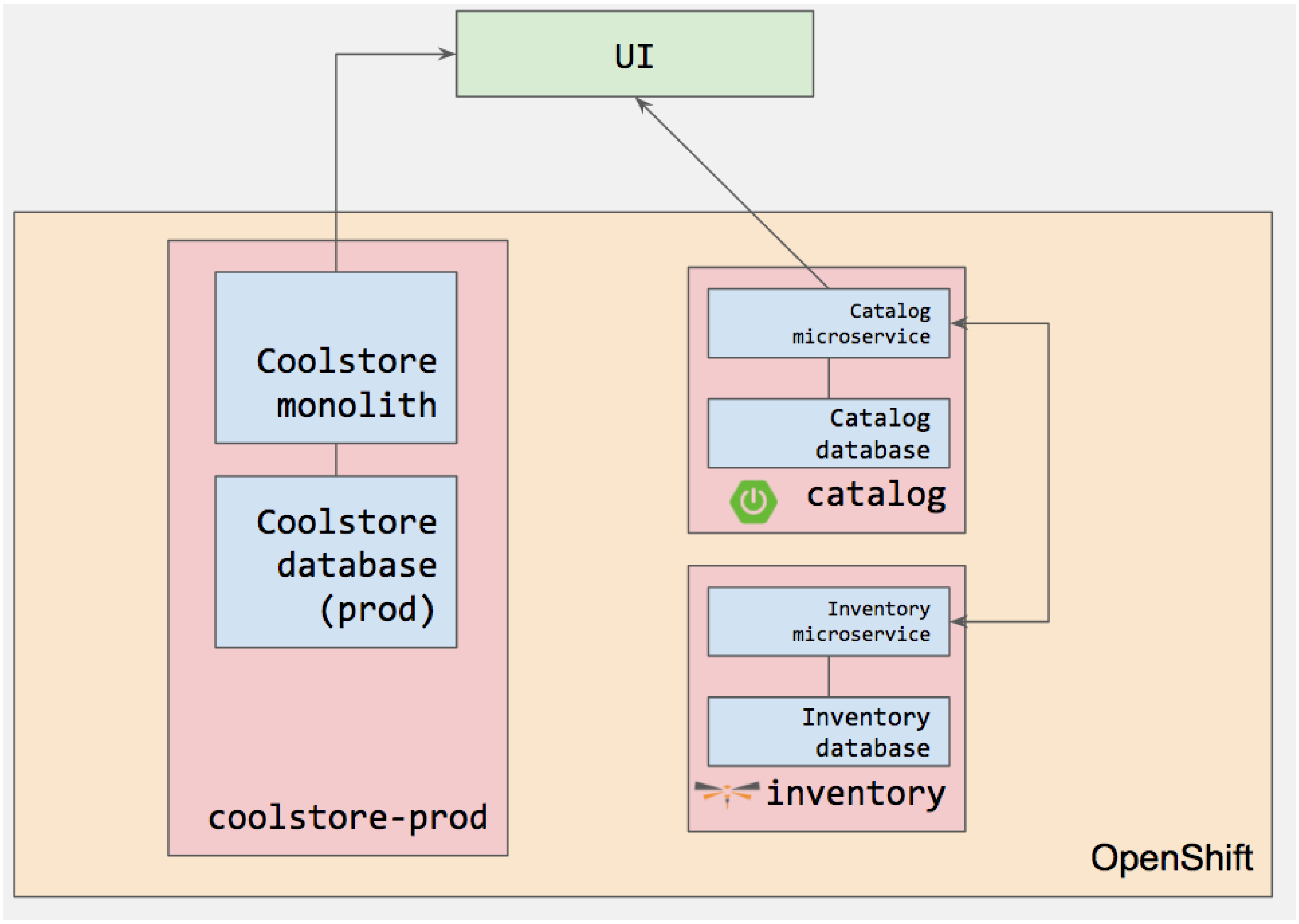


Figure 1: Greeting

fat jars) fits the container model very well since in a container platform like OpenShift responsibilities like starting, stopping and monitoring applications are then handled by the container platform instead of an Application Server.

Aggregate microservices calls

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

Setup for Exercise

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

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

Examine the sample project

For your convenience, this scenario has been created with a base project using the Java programming language and the Apache Maven build tool.

Initially, the project is almost empty and doesn't do anything. Start by reviewing the content by executing a `tree` in your terminal.

The output should look something like this

```
.
+-- pom.xml
+-- README.md
\-- src
    +-- main
    |   +-- fabric8
    |   |   +-- catalog-deployment.yml
    |   |   +-- catalog-route.yml
    |   |   \-- credential-secret.yml
    |   +-- java
    |   |   \-- com
    |   |       \-- redhat
    |   |           \-- coolstore
    |   |               +-- client
    |   |               +-- model
    |   |               |   +-- Inventory.java
    |   |               |   \-- Product.java
    |   |               +-- RestApplication.java
    |   |               \-- service
    |   \-- resources
    |       +-- application-default.properties
    |       +-- schema.sql
    |       \-- static
    |           \-- index.html
    \-- test
        \-- java
            \-- com
                \-- redhat
                    \-- coolstore
                        \-- service
```

As you can see, there are some files that we have prepared for you in the project. Under `src/main/resources/static/ind` we have for example prepared a simple html-based UI file for you. Except for the `fabric8/` folder and `index.html`, this matches very well what you would get if you generated an empty project from the Spring Initializr web page. For the moment you can ignore the content of the `fabric8/` folder (we will discuss this later).

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

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

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

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

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

Adding web (Apache Tomcat) to the application

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

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

To add Apache Tomcat to our project all we have to do is to add the following lines in `pom.xml`. Open the file to automatically add these lines at the `<!-- TODO: Add web (tomcat) dependency here -->` marker:

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

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

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

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

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

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

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

Test the application locally

As we develop the application, we might want to test and verify our change at different stages. We can do that locally, by using the `spring-boot` maven plugin.

Run the application by executing the below command:

```
mvn spring-boot:run
```

> **NOTE:** The Katacoda terminal window is like your local terminal. Everything that you run here you should be able to execute on your local computer as long as you have a `Java SDK 1.8` and Maven. In later steps, we will also use the oc command line tool.

Wait for it to complete startup and report `Started RestApplication in ***** seconds (JVM running for ******)`

3. Verify the application

To begin with, click on the **Local Web Browser** tab in the console frame of this browser window, which will open another tab or window of your browser pointing to port 8081 on your client.

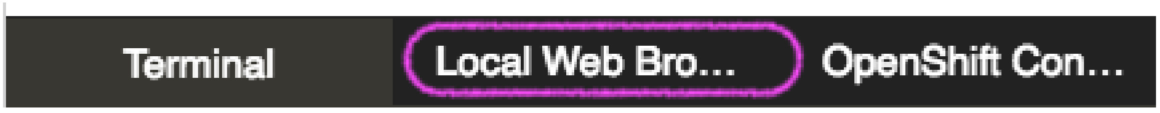
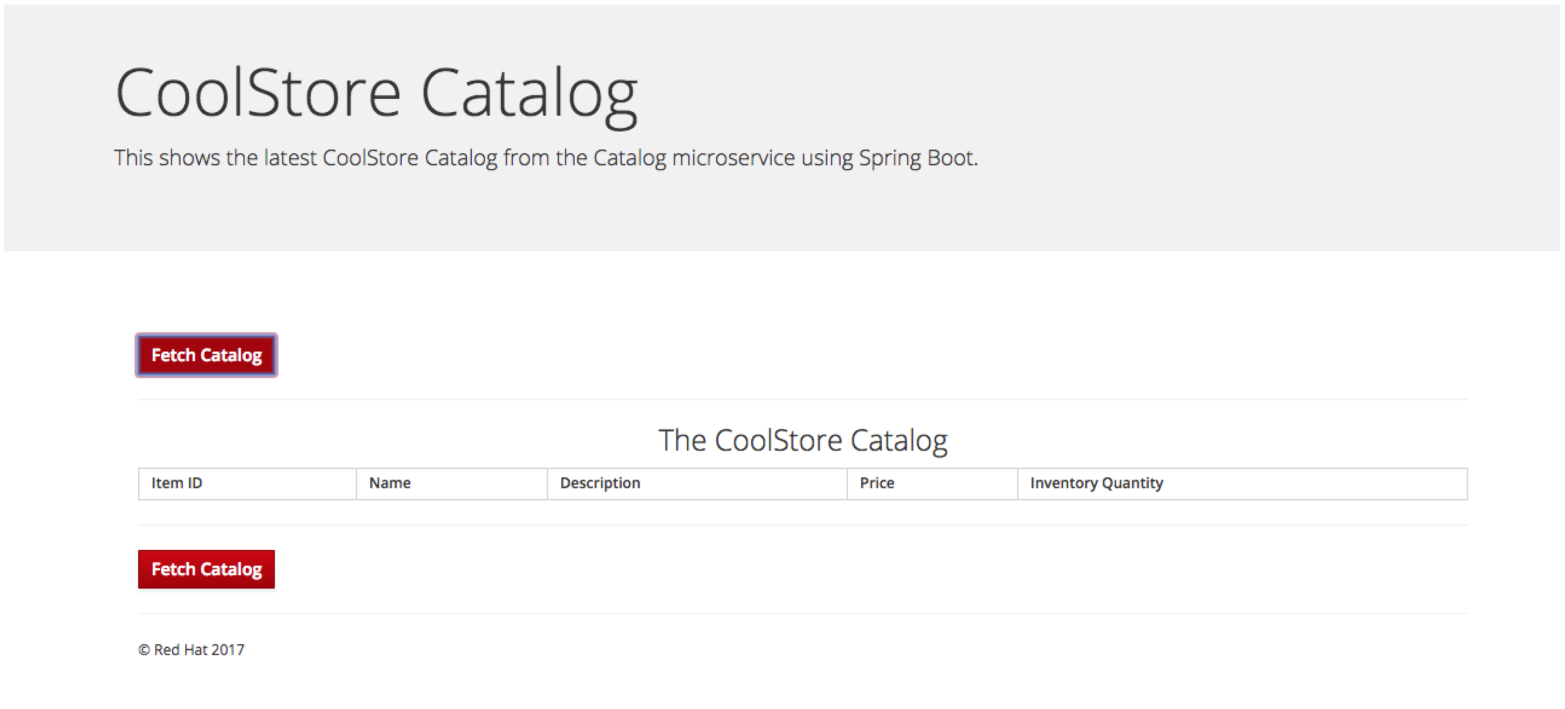


Figure 2: Local Web Browser Tab

or use this at

`http://localhost:8081` link.

You should now see an HTML page that looks like this:



CoolStore Catalog

This shows the latest CoolStore Catalog from the Catalog microservice using Spring Boot.

The CoolStore Catalog

Item ID	Name	Description	Price	Inventory Quantity

© Red Hat 2017

Figure 3: Local Web Browser Tab

> **NOTE:** The service calls to get products from the catalog doesn't work yet. Be patient! We will work on it in the next steps.

4. Stop the application

Before moving on, click here: `clear` to stop the running application.

Congratulations

You have now successfully executed the first step in this scenario.

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

In next step of this scenario, we will add the logic to be able to read a list of fruits from the database.

Create Domain Objects

Creating a test.

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

Click to open `src/test/java/com/redhat/coolstore/service/ProductRepositoryTest.java` to create the empty file and then **Copy to Editor** to copy the below code into the file:

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

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

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

import com.redhat.coolstore.model.Product;


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

//TODO: Insert Catalog Component here

//TODO: Insert test_readOne here

//TODO: Insert test_readAll here

}
```

Next, inject a handle to the future repository class which will provide access to the underlying data repository. It is injected with Spring's `@Autowired` annotation which locates, instantiates, and injects runtime instances of classes automatically, and manages their lifecycle (much like Java EE and it's CDI feature). Click to create this code:

```
@Autowired
ProductRepository repository;
```

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

Click to insert this code:

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

The `ProductRepository` should also provide a methods called `readAll()` that returns a list of all products in the catalog. We test this by making sure that the list contains a "Red Fedora", "Forge Laptop Sticker" and "Oculus Rift". Again, click to insert the code:

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

Implement the database repository

We are now ready to implement the database repository.

Create the `src/main/java/com/redhat/coolstore/service/ProductRepository.java` by clicking the open link.

Here is the base for the calls, click on the copy button to paste it into the editor:

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

import java.util.List;

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

@Repository
public class ProductRepository {

//TODO: Autowire the jdbcTemplate here

//TODO: Add row mapper here

//TODO: Create a method for returning all products

//TODO: Create a method for returning one product

}
```

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

Spring Data provides a convenient way for us to access data without having to write a lot of boiler plate code. One way to do that is to use a `JdbcTemplate`. First we need to autowire that as a member to `ProductRepository`. Click to add it:

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

The `JdbcTemplate` require that we provide a `RowMapper`so that it can map between rows in the query to Java Objects. We are going to define the `RowMapper` like this (click to add it):

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

Now we are ready to create the methods that are used in the test. Let’s start with the `readAll()`. It should return a `List<Product>` and then we can write the query as `SELECT * FROM catalog` and use the rowMapper to map that into `Product` objects. Our method should look like this (click to add it):

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

The `ProductRepositoryTest` also used another method called `findById(String id)` that should return a Product. The implementation of that method using the `JdbcTemplate` and `RowMapper` looks like this (click to add it):

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

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

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

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

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

Now we are ready to run the test to verify that everything works. Because we created the `ProductRepositoryTest.java` all we have todo is to run: `mvn verify`

The test should be successful and you should see **BUILD SUCCESS**, which means that we can read that our repository class works as as expected.

Congratulations

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

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

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

Create Catalog Service

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

Create a new class `CatalogService` by clicking: `src/main/java/com/redhat/coolstore/service/CatalogService.ja`

And then Open the file to implement the new service:

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

import com.redhat.coolstore.model.Inventory;
import com.redhat.coolstore.model.Product;
//import com.redhat.coolstore.client.InventoryClient;
import feign.hystrix.FallbackFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.netflix.feign.FeignClient;
import org.springframework.http.MediaType;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

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

@Service
public class CatalogService {
    @Autowired
    private ProductRepository repository;

    //TODO: Autowire Inventory Client

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

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

    //TODO: Add Callback Factory Component


}
```

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

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

Create the test case by opening: `src/test/java/com/redhat/coolstore/service/CatalogEndpointTest.java`

Add the following code to the test case and make sure to review it so that you understand how it works.

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

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

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

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

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
```

```
public class CatalogEndpointTest {

    @Autowired
    private TestRestTemplate restTemplate;

//TODO: Add ClassRule for HoverFly Inventory simulation

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


    @Test
    public void check_that_endpoint_returns_a_correct_list() {

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

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

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

}
```

Now we are ready to implement the `CatalogEndpoint`.

Start by creating the file by opening: `src/main/java/com/redhat/coolstore/service/CatalogEndpoint.java`

The add the following content:

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

import java.util.List;

import com.redhat.coolstore.model.Product;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

@Controller
@RequestMapping("/services")
public class CatalogEndpoint {
    @Autowired
    private CatalogService catalogService;

    @ResponseBody
    @GetMapping("/products")
    public ResponseEntity<List<Product>> readAll() {
        return new ResponseEntity<List<Product>>(catalogService.readAll(),HttpStatus.OK);
    }

    @ResponseBody
    @GetMapping("/product/{id}")
    public ResponseEntity<Product> read(@PathVariable("id") String id) {
        return new ResponseEntity<Product>(catalogService.read(id),HttpStatus.OK);
    }

}
```

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

Now you can run the `CatalogEndpointTest` and verify that it works.

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

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

Start the application by running the following command `mvn spring-boot:run`

Wait for the application to start. Then we can verify the endpoint by running the following command in a new terminal (Note the link below will execute in a second terminal)

```
curl http://localhost:8081/services/products ; echo
```

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

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

Also click on the **Local Web Browser** tab in the console frame of this browser window, which will open another tab or window of your browser pointing to port 8081 on your client.

> If you are using CDK, simply browse to http://localhost:8081

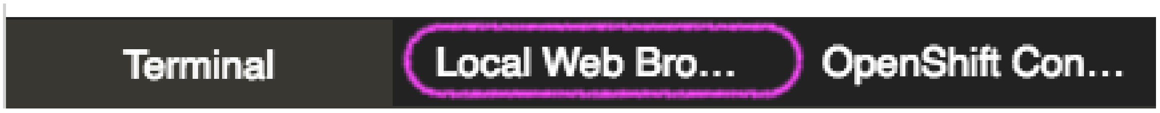


Figure 4: Local Web Browser Tab

or use this at

`http://localhost:8081` link.

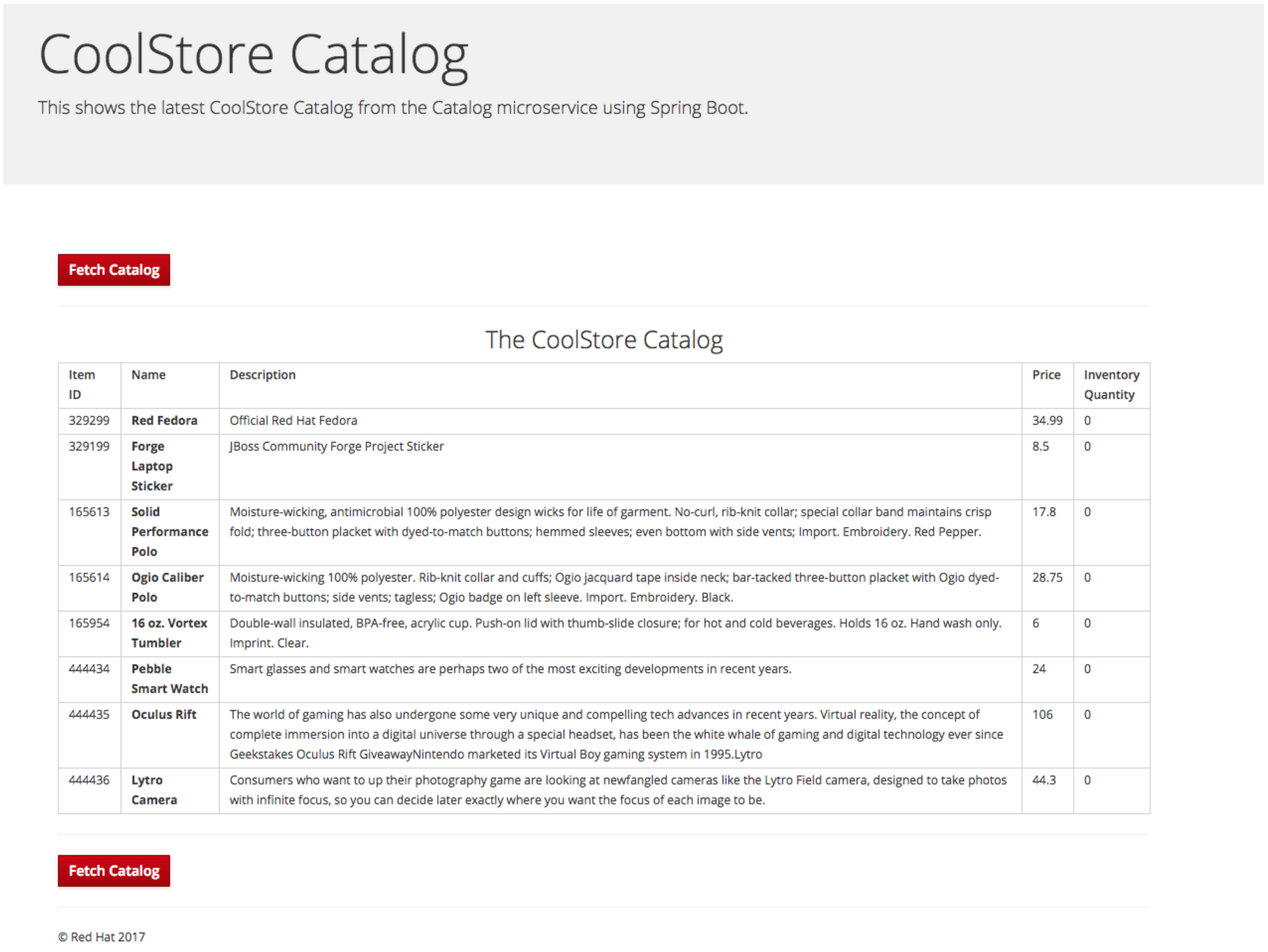
You should now see an HTML page that looks like this:

Congratulations

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

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

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



CoolStore Catalog

This shows the latest CoolStore Catalog from the Catalog microservice using Spring Boot.

The CoolStore Catalog

Item ID	Name	Description	Price	Inventory Quantity
329299	**Red Fedora**	Official Red Hat Fedora	34.99	0
329199	**Forge Laptop Sticker**	JBoss Community Forge Project Sticker	8.5	0
165613	**Solid Performance Polo**	Moisture-wicking, antimicrobial 100% polyester design wicks for life of garment. No-curl, rib-knit collar; special collar band maintains crisp fold; three-button placket with dyed-to-match buttons; hemmed sleeves; even bottom with side vents; Import. Embroidery. Red Pepper.	17.8	0
165614	**Ogio Caliber Polo**	Moisture-wicking 100% polyester. Rib-knit collar and cuffs; Ogio jacquard tape inside neck; bar-tacked three-button placket with Ogio dyed-to-match buttons; side vents; tagless; Ogio badge on left sleeve. Import. Embroidery. Black.	28.75	0
165954	**16 oz. Vortex Tumbler**	Double-wall insulated, BPA-free, acrylic cup. Push-on lid with thumb-slide closure; for hot and cold beverages. Holds 16 oz. Hand wash only. Imprint. Clear.	6	0
444434	**Pebble Smart Watch**	Smart glasses and smart watches are perhaps two of the most exciting developments in recent years.	24	0
444435	**Oculus Rift**	The world of gaming has also undergone some very unique and compelling tech advances in recent years. Virtual reality, the concept of complete immersion into a digital universe through a special headset, has been the white whale of gaming and digital technology ever since Geekstakes Oculus Rift GiveawayNintendo marketed its Virtual Boy gaming system in 1995.Lytro	106	0
444436	**Lytro Camera**	Consumers who want to up their photography game are looking at newfangled cameras like the Lytro Field camera, designed to take photos with infinite focus, so you can decide later exactly where you want the focus of each image to be.	44.3	0

© Red Hat 2017

Figure 5: Local Web Browser Tab

Before moving on

Be sure to stop the service by clicking on the first Terminal window and typing `CTRL-C` (or click `clear` to do it for you).

Congratulations!

Next, we'll add a call to the existing Inventory service to enrich the above data with Inventory information. On to the next challenge!

Get inventory data

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

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

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

Service interaction

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

1. **Client Side integration** - We could extend our UI to first call `/services/products` and then for each product item call `/services/inventory/{prodId}` to get the inventory status and then combine the result in the web browser. This would be the least intrusive method, but it also means that if we have 100 of products the client will make 101 request to the server. If we have a slow internet connection this may cause issues.
2. **Microservices Gateway** - Creating a gateway in-front of the `Catalog Service` that first calls the Catalog Service and then based on the response calls the inventory is another option. This way we can avoid lots of calls from the client to the server. Apache Camel provides nice capabilities to do this and if you are interested to learn more about this, please checkout the Coolstore Microservices example here
3. **Service-to-Service** - Depending on use-case and preferences another solution would be to do service-to-service calls instead. In our case means that the Catalog Service would call the Inventory service using REST to retrieve the inventory status and include that in the response.

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

Extending the test

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

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

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

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

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

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

Now if we run the test it **should fail**!

```
mvn verify
```

It failed:

```
Tests run: 4, Failures: 2, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
```

Again the test fails because we are trying to call the Inventory service which is not running. We will soon implement the code to call the inventory service, but first we need a away to test this service without having to really on the inventory services to be up an running. For that we are going to use an API Simulator called HoverFly and particular it's capability to simulate remote APIs. HoverFly is very convenient to use with Unit test and all we have to do is to add a `ClassRule` that will simulate all calls to inventory. Open the file to insert the code at the `//TODO: Add ClassRule for HoverFly Inventory simulation` marker:

@ClassRule public static HoverflyRule hoverflyRule = HoverflyRule.inSimulationMode(dsl(service("inventory:8080") // .andDelay(2500, TimeUnit.MILLISECONDS).forMethod("GET") .get(startsWith("/services/inventory")) // .willReturn(serverError()) .willReturn(success(json(new Inventory("9999",9999))))

)); "'

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

Implementing the Inventory Client

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

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

Create the Inventory client by clicking `src/main/java/com/redhat/coolstore/client/InventoryClient.java`

Add the followng small code to the file:

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

import com.redhat.coolstore.model.Inventory;
import feign.hystrix.FallbackFactory;
import org.springframework.http.MediaType;
import org.springframework.stereotype.Component;
import org.springframework.cloud.netflix.feign.FeignClient;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

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

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

//TODO: Add Fallback factory here
}
```

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

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

And add these properties by clicking **Copy to Editor** and adding to the `#TODO: Configure netflix libraries` marker:

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

By setting `inventory.ribbon.listOfServers` we are hard coding the actual URL of the service to `inventory:8080`. If we had multiple servers we could also add those using a comma. However using Kubernetes there is no need to have multiple endpoints listed here since Kubernetes has a concept of *Services* that will internally route between multiple instances of the same service. Later on we will update this value to reflect our URL when deploying to OpenShift.

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

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

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

```
@Autowired
InventoryClient inventoryClient;
```

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

```
product.setQuantity(inventoryClient.getInventoryStatus(product.getItemId()).getQuantity());
```

Also, don't forget to add the import statement by un-commenting the import statement `//import com.redhat.coolstore` near the top

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

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

```
productList.parallelStream()
            .forEach(p -> {
                p.setQuantity(inventoryClient.getInventoryStatus(p.getItemId()).getQuantity());
            });
```

> **NOTE:** The lambda expression to update the product list uses a `parallelStream`, which means that it will process the inventory calls asynchronously, which will be much faster than using synchronous calls. Optionally when we run the test you can test with both `parallelStream()` and `stream()` just to see the difference in how long the test takes to run.

We are now ready to test the service

`mvn verify`

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

Congratulations

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

Create a fallback for inventory

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

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

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

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

@Component static class InventoryClientFallbackFactory implements FallbackFactory { @Override public InventoryClient create(Throwable cause) { return new InventoryClient() { @Override public Inventory getInventoryStatus(@PathVariable(“itemId”) String itemId) { return new Inventory(itemId,-1); } }; } }

```
After creating the fallback factory all we have todo is to tell Feign to use that fallback in case o
it for you at the `@FeignClient(name="inventory")` line:

<pre class="file" data-filename="src/main/java/com/redhat/coolstore/client/InventoryClient.java"
data-target="insert" data-marker="@FeignClient(name=&quot;inventory&quot;)">
@FeignClient(name="inventory",fallbackFactory = InventoryClient.InventoryClientFallbackFactory.class
```

Test the Fallback

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

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

```
@ClassRule
public static HoverflyRule hoverflyRule = HoverflyRule.inSimulationMode(dsl(
        service("inventory:8080")
//                    .andDelay(2500, TimeUnit.MILLISECONDS).forMethod("GET")
                .get(startsWith("/services/inventory"))
//                    .willReturn(serverError())
                .willReturn(success(json(new Inventory("9999",9999))))

));
```

TO

```
@ClassRule
public static HoverflyRule hoverflyRule = HoverflyRule.inSimulationMode(dsl(
        service("inventory:8080")
//                    .andDelay(2500, TimeUnit.MILLISECONDS).forMethod("GET")
                .get(startsWith("/services/inventory"))
                .willReturn(serverError())
//                    .willReturn(success(json(new Inventory("9999",9999))))

));
```

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

Now if you run `mvn verify -Dtest=CatalogEndpointTest` the test will fail with the following error message:

```
Failed tests:   test_retriving_one_proudct(com.redhat.coolstore.service.CatalogEndpointTest):
expected:<[9999]> but was:<[-1]>
```

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

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

```
@ClassRule
public static HoverflyRule hoverflyRule = HoverflyRule.inSimulationMode(dsl(
        service("inventory:8080")
//                    .andDelay(2500, TimeUnit.MILLISECONDS).forMethod("GET")
                .get(startsWith("/services/inventory"))
//                    .willReturn(serverError())
                .willReturn(success(json(new Inventory("9999",9999))))

));
```

Make sure the test works again by running `mvn verify -Dtest=CatalogEndpointTest`

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

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

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

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

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

Now if you run `mvn verify -Dtest=CatalogEndpointTest` the test will fail with the following error message:

```
Failed tests:   test_retriving_one_proudct(com.redhat.coolstore.service.CatalogEndpointTest):
expected:<[9999]> but was:<[-1]>
```

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

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

Congratulations

You have now successfully executed the fifth step in this scenario.

In this step you've learned how to add Fallback logic to your class and how to add timeout to service calls.

In the next step we now test our service locally before we deploy it to OpenShift.

Test Locally

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

Execute the following command to run the new service locally:

```
mvn spring-boot:run
```

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

Once the application is done initializing you should see:

```
INFO  [           main] com.redhat.coolstore.RestApplication     : Started RestApplication ...
```

Running locally using `spring-boot: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 8081 on your client.

Users do not have a **Local Web Browser** link. Use the link below.

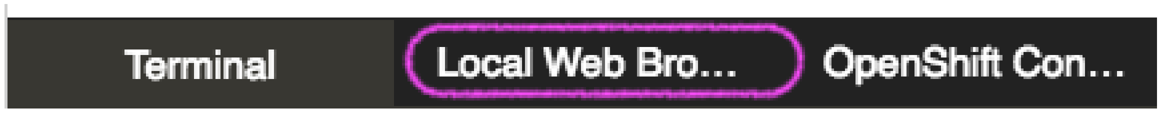


Figure 6: Local Web Browser Tab

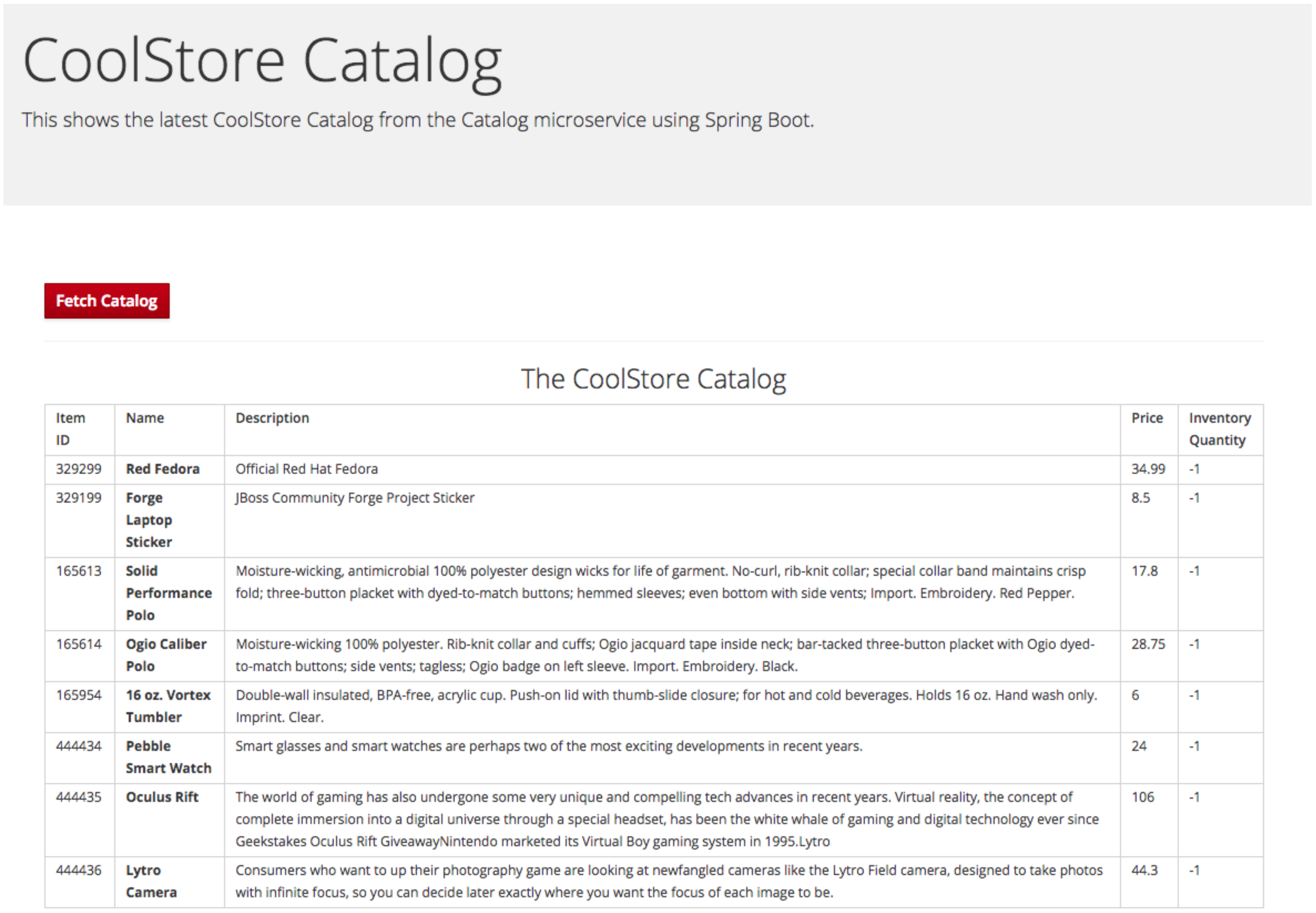
> or use this at

`http://localhost:8081` 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 Catalog** button to force it to refresh at any time.



CoolStore Catalog

This shows the latest CoolStore Catalog from the Catalog microservice using Spring Boot.

The CoolStore Catalog

Item ID	Name	Description	Price	Inventory Quantity
329299	**Red Fedora**	Official Red Hat Fedora	34.99	-1
329199	**Forge Laptop Sticker**	JBoss Community Forge Project Sticker	8.5	-1
165613	**Solid Performance Polo**	Moisture-wicking, antimicrobial 100% polyester design wicks for life of garment. No-curl, rib-knit collar; special collar band maintains crisp fold; three-button placket with dyed-to-match buttons; hemmed sleeves; even bottom with side vents; Import. Embroidery. Red Pepper.	17.8	-1
165614	**Ogio Caliber Polo**	Moisture-wicking 100% polyester. Rib-knit collar and cuffs; Ogio jacquard tape inside neck; bar-tacked three-button placket with Ogio dyed-to-match buttons; side vents; tagless; Ogio badge on left sleeve. Import. Embroidery. Black.	28.75	-1
165954	**16 oz. Vortex Tumbler**	Double-wall insulated, BPA-free, acrylic cup. Push-on lid with thumb-slide closure; for hot and cold beverages. Holds 16 oz. Hand wash only. Imprint. Clear.	6	-1
444434	**Pebble Smart Watch**	Smart glasses and smart watches are perhaps two of the most exciting developments in recent years.	24	-1
444435	**Oculus Rift**	The world of gaming has also undergone some very unique and compelling tech advances in recent years. Virtual reality, the concept of complete immersion into a digital universe through a special headset, has been the white whale of gaming and digital technology ever since Geekstakes Oculus Rift GiveawayNintendo marketed its Virtual Boy gaming system in 1995.Lytro	106	-1
444436	**Lytro Camera**	Consumers who want to up their photography game are looking at newfangled cameras like the Lytro Field camera, designed to take photos with infinite focus, so you can decide later exactly where you want the focus of each image to be.	44.3	-1

Figure 7: App

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:8081/services/product/329299 ; echo
```

You would see a JSON response like this:

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

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

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

4. Stop the application

Before moving on, click in the first terminal window where the app is running and then press CTRL-C to stop the running application! Or click `clear` to do it for you.

Congratulations

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

In next 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.

Create the OpenShift project

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

1. Create project

Create a new project for the *catalog* service:

```
oc new-project catalog --display-name="CoolStore Catalog Microservice Application"
```

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

Deploy to OpenShift

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

Deploy the Database

Our production catalog 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=catalog \              -e POSTGRESQL_PASSWORD=mysecretpassword \
-e POSTGRESQL_DATABASE=catalog \              openshift/postgresql:latest \              --name=cata
```

> **NOTE:** If you change the username and password you also need to update `src/main/fabric8/credential-secret.y` 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/catalog-database
```

Update configuration Create the file by clicking: `src/main/resources/application-openshift.properties`

Copy the following content to the file:

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

```
inventory.ribbon.listOfServers=inventory.inventory.svc.cluster.local:8080
```

> **NOTE:** The `application-openshift.properties` does not have all values of `application-default.properties`, that is because on the values that need to change has to be specified here. Spring will fall back to `application-default.properties` for the other values.

Build and Deploy

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

```
mvn package fabric8:deploy -Popenshift -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.

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/catalog
```

> **NOTE:** If you recall in the WildFly Swarm lab Fabric8 detected the `health` *fraction* and generated health check definitions for us, the same is true for Spring Boot if you have the `spring-boot-starter-actuator` dependency in our project.

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://catalog-catalog.$OPENSHIFT_MASTER` to access the sample UI.

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

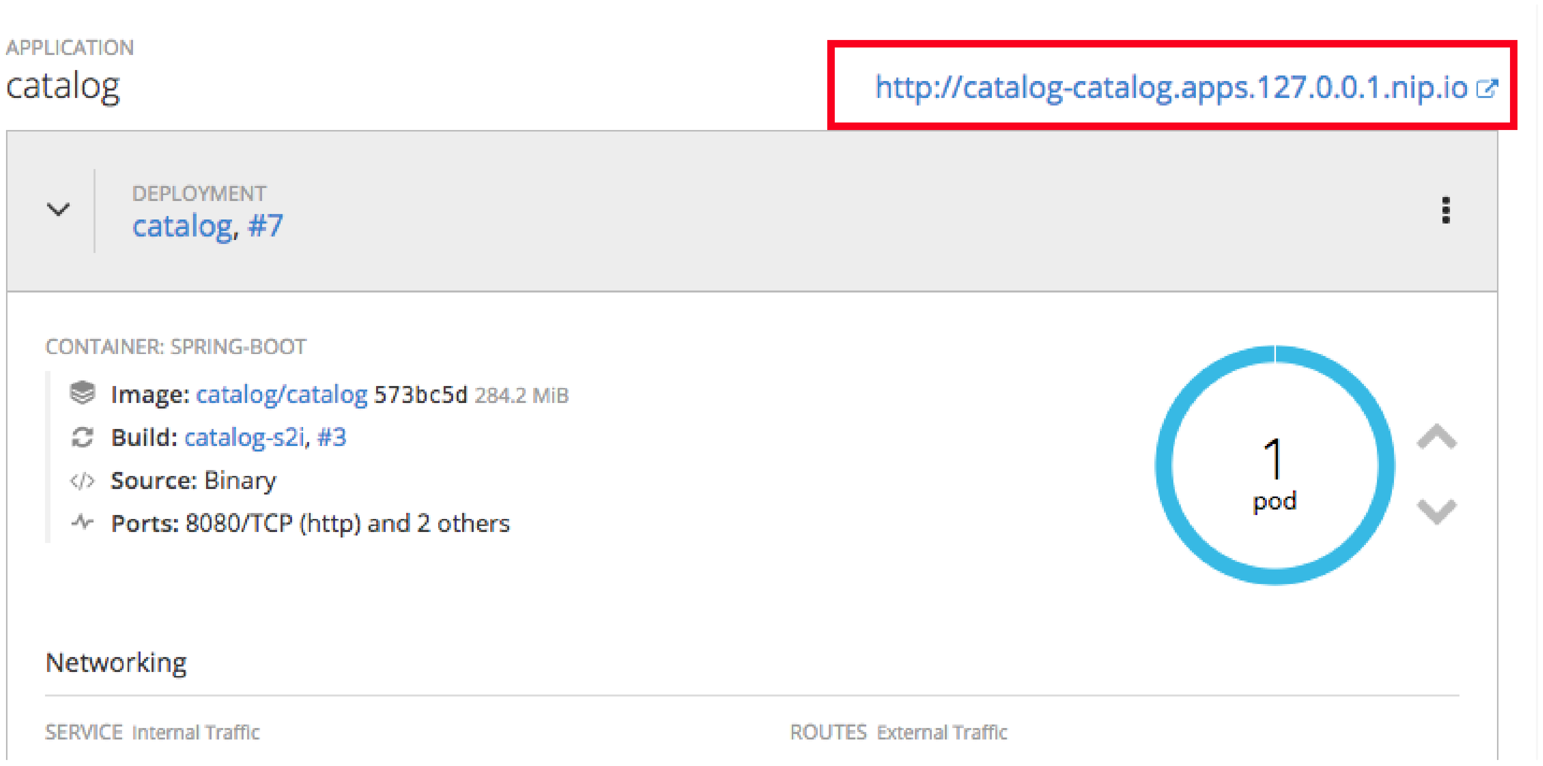

APPLICATION
catalog
http://catalog-catalog.apps.127.0.0.1.nip.io
DEPLOYMENT
catalog, #7
CONTAINER: SPRING-BOOT
Image: catalog/catalog 573bc5d 284.2 MiB
Build: catalog-s2i, #3
Source: Binary
Ports: 8080/TCP (http) and 2 others
1 pod
Networking
SERVICE Internal Traffic
ROUTES External Traffic

Figure 8: Overview link

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

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

Congratulations!

You have deployed the Catalog service as a microservice which in turn calls into the Inventory service to retrieve inventory data. However, our monolih UI is still using its own built-in services. Wouldn't it be nice if we could re-wire the monolith to use the new services, **without changing any code**? That's next!

Strangling the monolith

So far we haven't started strangling the monolith. To do this we are going to make use of routing capabilities in OpenShift. Each external request coming into OpenShift (unless using ingress, which we are not) will pass through a route. In our monolith the web page uses client side REST calls to load different parts of pages.

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

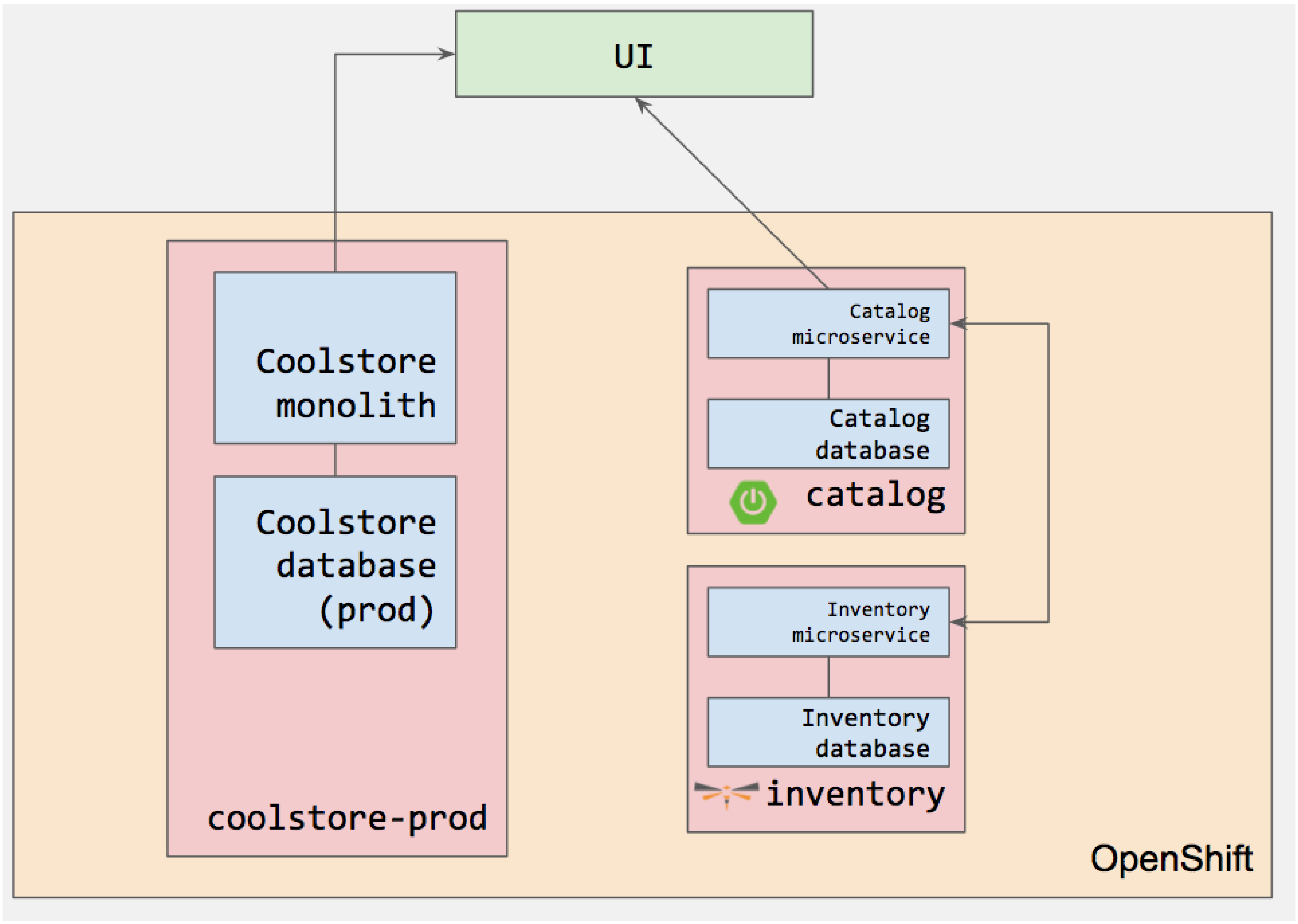


Figure 9: Greeting

Flow the steps below to create a path based route.

1. Obtain hostname of monolith UI from our Dev environment

```
oc get route/www -n coolstore-dev
```

The output of this command shows us the hostname:

```
NAME      HOST/PORT                                 PATH      SERVICES    PORT      TERMINATION   WI
www       www-coolstore-dev.apps.127.0.0.1.nip.io             coolstore   <all>                   No
```

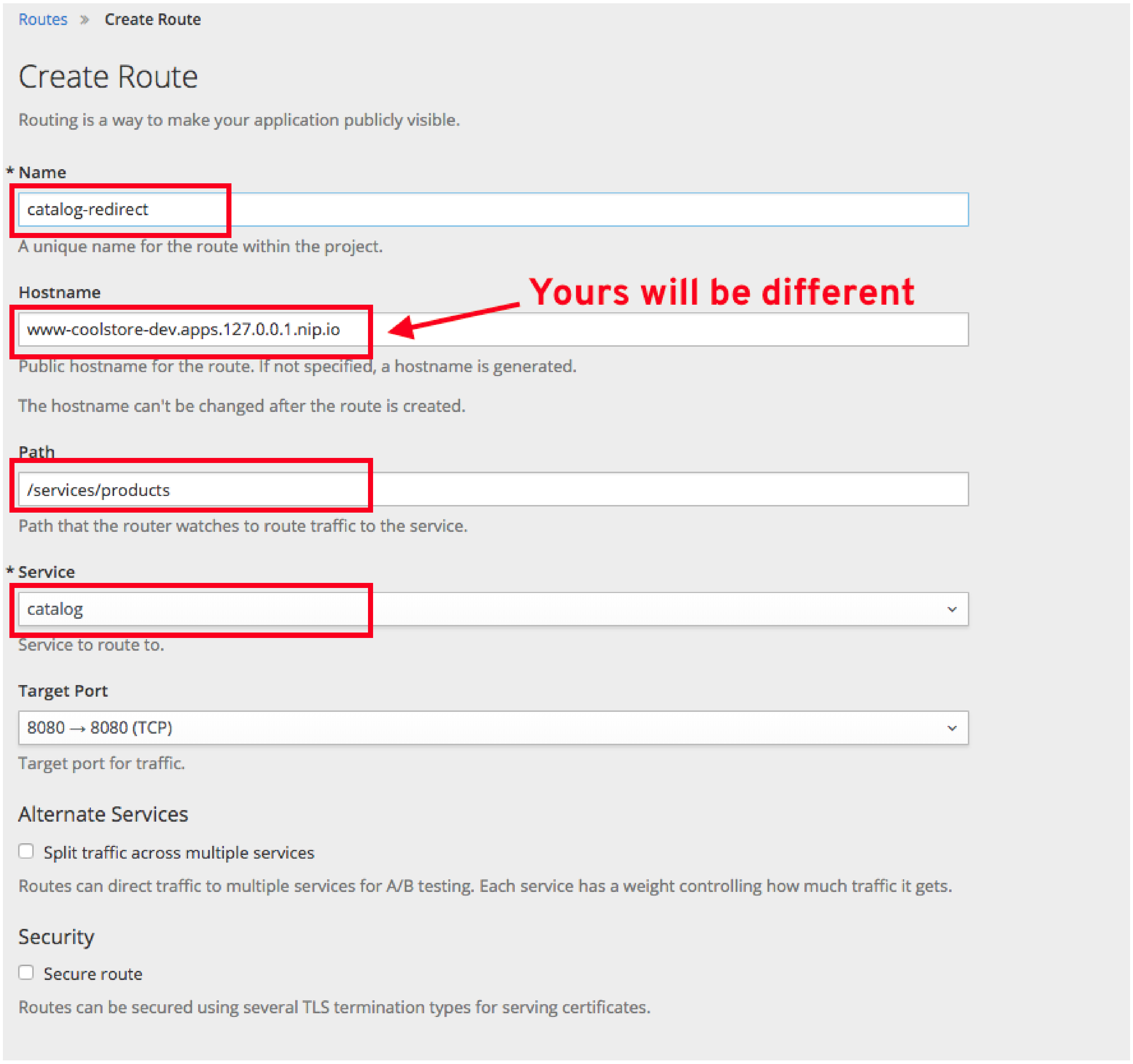
My hostname is `www-coolstore-dev.apps.127.0.0.1.nip.io` but **yours will be different**.

**2. Open the openshift console for Catalog - Applications - Routes at

`https://$OPENSHIFT_MASTER/console/project/catalog/browse/routes`**

3. Click on Create Route, and set

- **Name**: `catalog-redirect`
- **Hostname**: *the hostname from above*
- **Path**: `/services/products`
- **Service**: `catalog`



Create Route

Routing is a way to make your application publicly visible.

* Name

catalog-redirect

A unique name for the route within the project.

Hostname

www-coolstore-dev.apps.127.0.0.1.nip.io

Public hostname for the route. If not specified, a hostname is generated.

The hostname can't be changed after the route is created.

Path

/services/products

Path that the router watches to route traffic to the service.

* Service

catalog

Service to route to.

Target Port

8080 → 8080 (TCP)

Target port for traffic.

Alternate Services

☐ Split traffic across multiple services

Routes can direct traffic to multiple services for A/B testing. Each service has a weight controlling how much traffic it gets.

Security

☐ Secure route

Routes can be secured using several TLS termination types for serving certificates.

Figure 10: Greeting

Leave other values set to their defaults, and click **Save**

4. Test the route

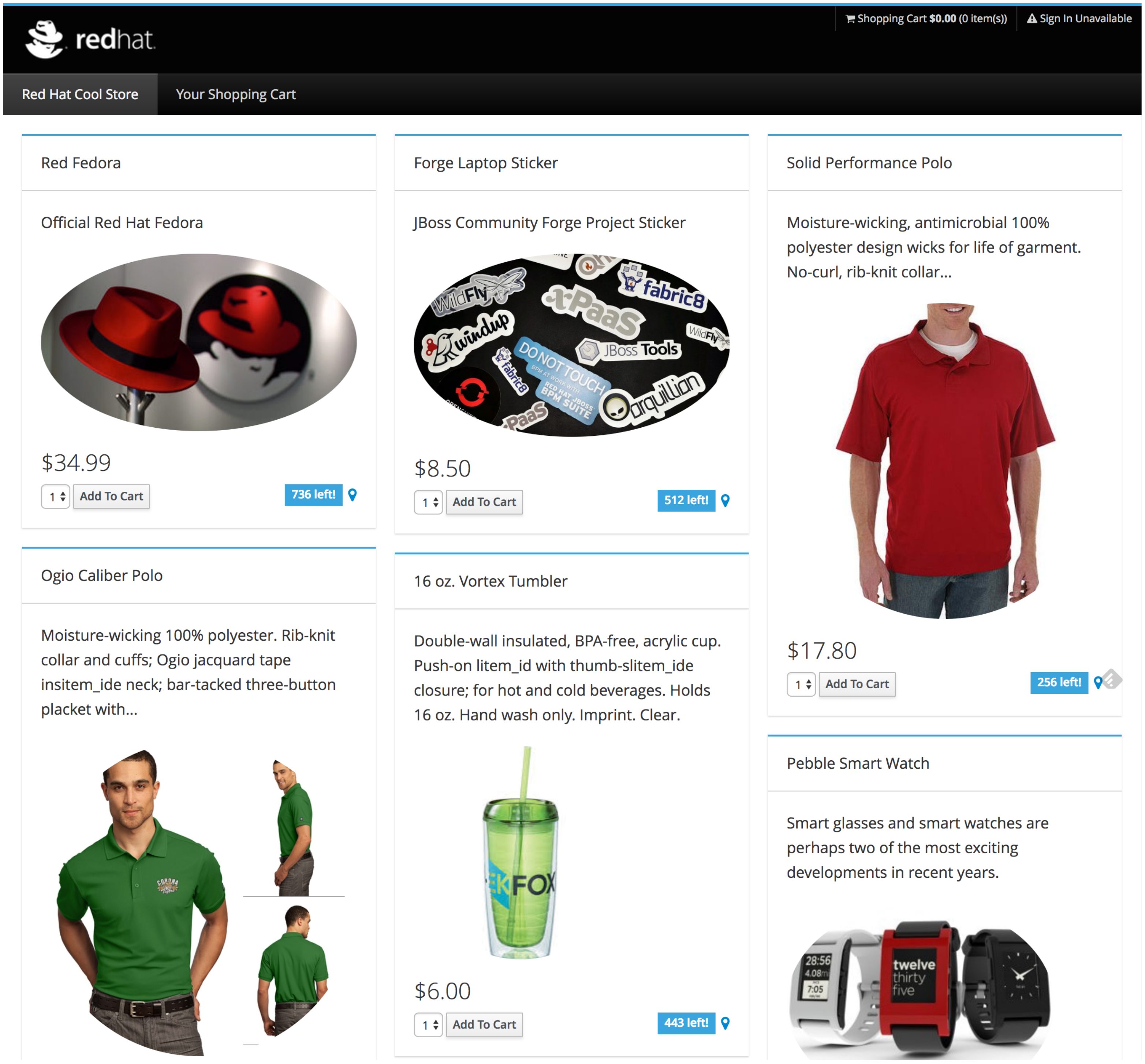
Test the route by running `curl http://www-coolstore-dev.[[HOST_SUBDOMAIN]]-80-[[KATACODA_HOST]].environme`

You should get a complete set of products, along with their inventory.

5. Test the UI

Open the monolith UI at

`http://www-coolstore-dev.$OPENSHIFT_MASTER` and observe that the new catalog is being used along with the monolith:



Red Fedora

Official Red Hat Fedora

$34.99

1 Add To Cart 736 left!

Forge Laptop Sticker

JBoss Community Forge Project Sticker

$8.50

1 Add To Cart 512 left!

Solid Performance Polo

Moisture-wicking, antimicrobial 100% polyester design wicks for life of garment. No-curl, rib-knit collar...

$17.80

1 Add To Cart 256 left!

Ogio Caliber Polo

Moisture-wicking 100% polyester. Rib-knit collar and cuffs; Ogio jacquard tape insitem_ide neck; bar-tacked three-button placket with...

16 oz. Vortex Tumbler

Double-wall insulated, BPA-free, acrylic cup. Push-on litem_id with thumb-slitem_ide closure; for hot and cold beverages. Holds 16 oz. Hand wash only. Imprint. Clear.

$6.00

1 Add To Cart 443 left!

Pebble Smart Watch

Smart glasses and smart watches are perhaps two of the most exciting developments in recent years.

Figure 11: Greeting

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

Congratulations!

You have now successfully begun to *strangle* the monolith. Part of the monolith's functionality (Inventory and Catalog) are now implemented as microservices, without touching the monolith. But there's a few more things left to do, which we’ll do in the next steps.

Summary

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

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