LEARN HANDS-ON HOW TO MODERNIZE LEGACY APPLICATIONS

Thomas Qvarnström
Pr. Technical Marketing Manager, Middleware
April 2017

James Falkner
Sr. Technical Marketing Manager, Middleware
April 2017

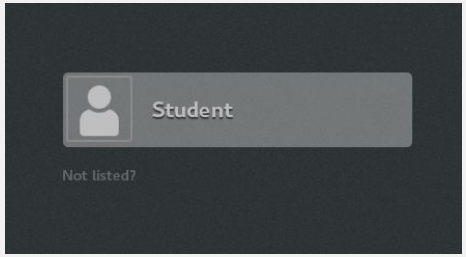
LAB ENVIRONMENT

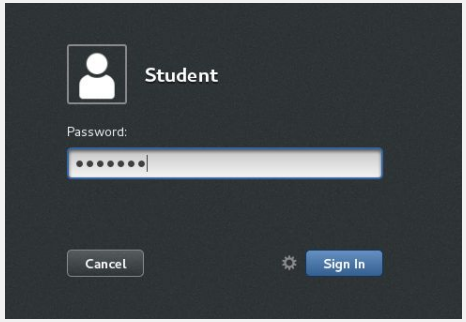
LOGIN TO YOUR ENVIRONMENT

All participants have a Red Hat Desktop

- Double-click **View Workstation** on the host machine's desktop
- To login click on the student user icon

- At the login screen type "student" as password and click **Sign In**





USEFUL COMMANDS

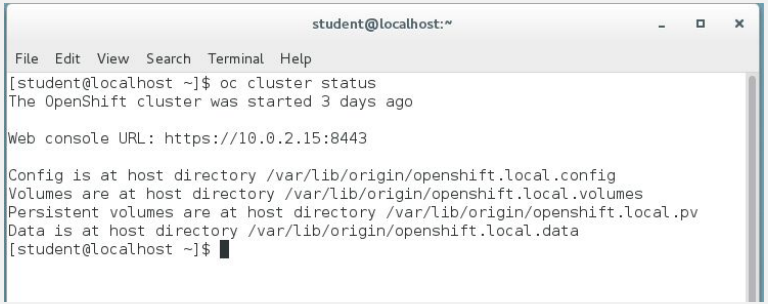
- Use the WinKey to show favorite applications
- To copy text use **CTRL+C** or in the terminal **CTRL+SHIFT+C**
- To paste text use **CTRL+V** or in the terminal **CTRL+SHIFT+V**

VERIFY THE ENVIRONMENT

- Open a terminal window and run the following command:

```
$ oc cluster status
```

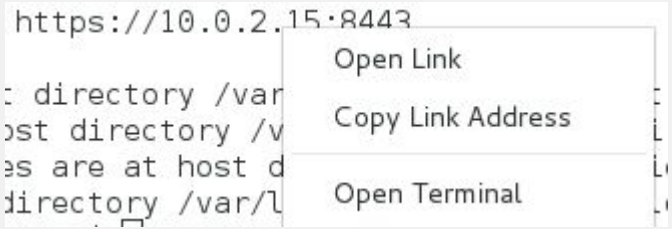
- The output should look something like the picture below, but the ip address may be different:



OPEN THE OPENSHIFT WEB CONSOLE

- Right click on the web console url in the terminal window and select **Open Link**

- This will open the firefox browser to a local openshift cluster running on your machine.



NOTE: *If you get a security exception don't worry, just click on* ***Advanced > Add Exception... > Confirm Security Exception***

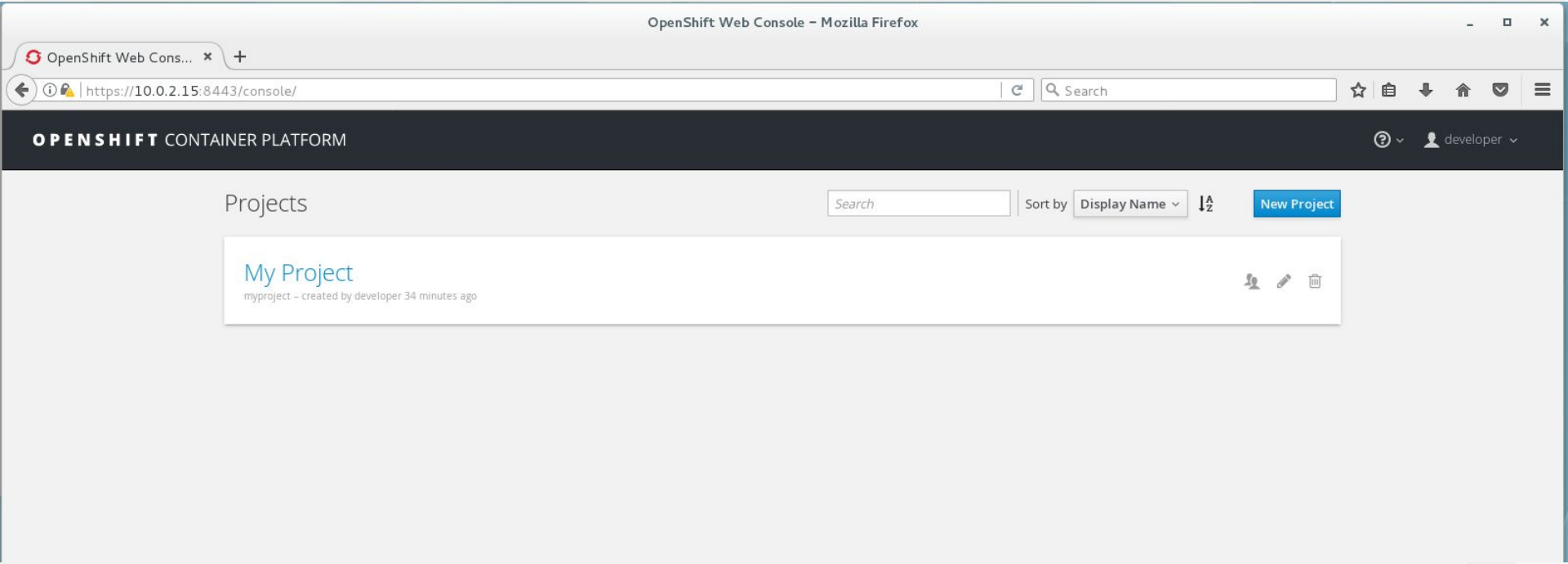
LOGIN TO THE OPENSHIFT WEB CONSOLE

- Login using the following credentials:

  `Username:` **`developer`**

  `Password:` **`developer`**

THE OPENSHIFT WEB CONSOLE



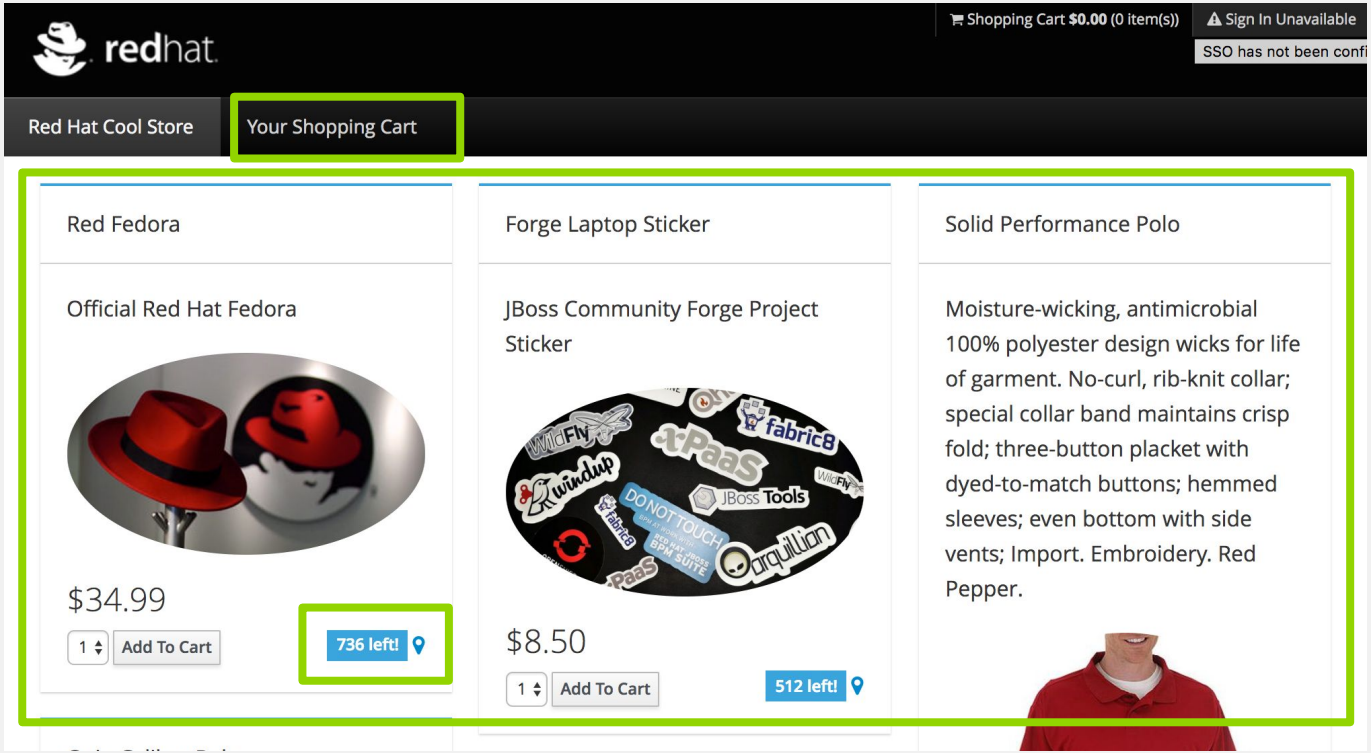
LOGIN TO THE CLI

- In the terminal window, run the following command:

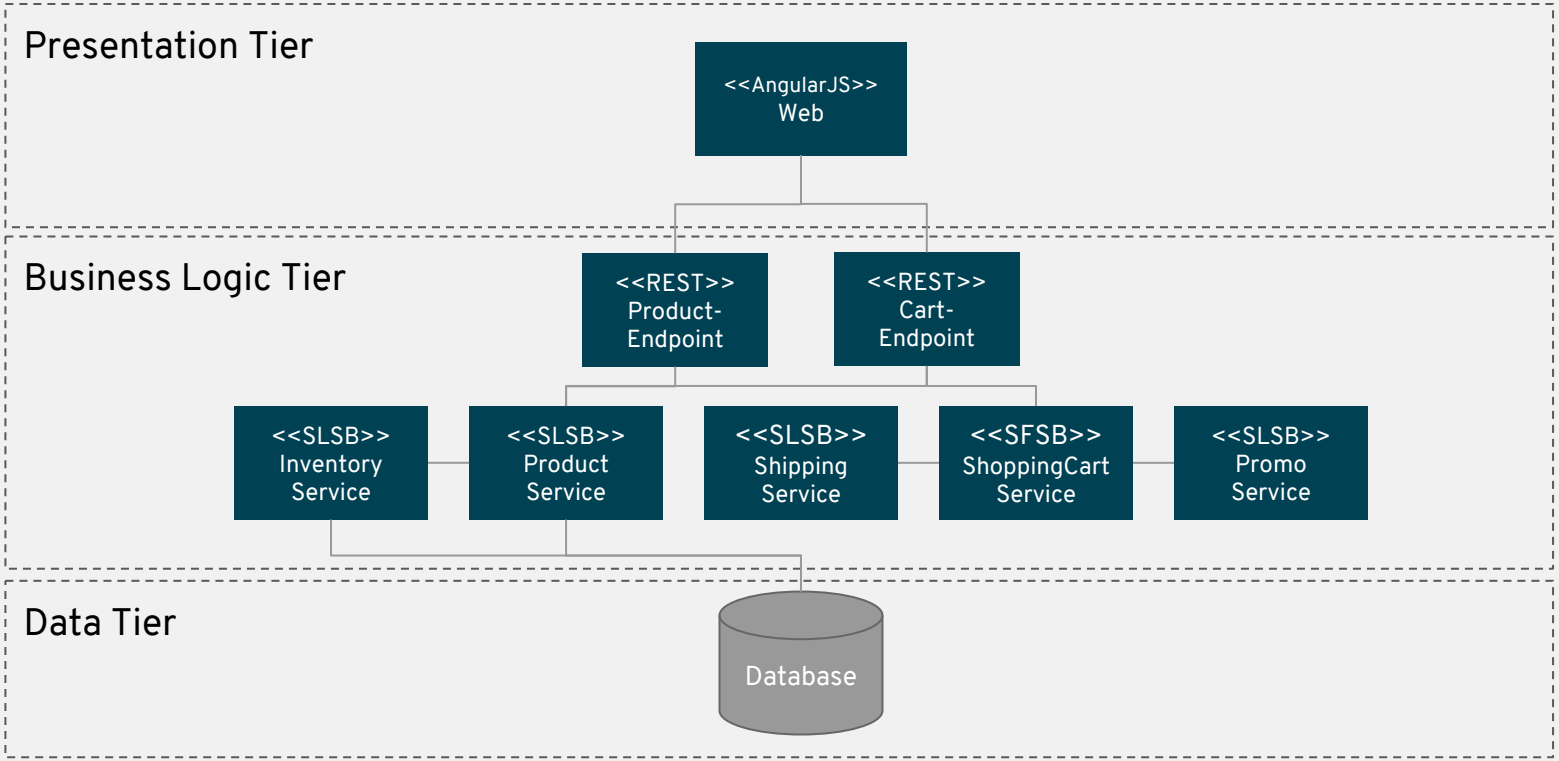
```
$ oc login -u developer -p developer
```

THE DEMO APPLICATION

COOLSTORE APPLICATION



COOLSTORE APPLICATION



FAST MOVING MONOLITH

FAST-MOVING MONOLITH

- Large organizations have a tremendous amount of resources **invested in existing** monolith applications
- Looking for a **sane way** to capture the benefits of containers and orchestration **without having to complete rewrite**
- **OpenShift** provides the platform for their existing investment with the benefit of a **path forward** for microservice based apps in the future

FAST-MOVING MONOLITH ADVANTAGES

- Easier to develop since all dependencies are included
- Single code base for teams to work on
- No API backwards compatibility issues since all logic is packaged with the application
- Single deployable unit

FAST-MOVING MONOLITH DISADVANTAGES

- Hard for large teams to all work in the same code base.
  - Who broke the build?
- Longer release cycles and even small changes goes through entire test suite and validation
- Same language / Same Framework in most cases
- Entropic resistance is low

LAB 1
BUILDING AND DEPLOYING A FAST-MOVING MONOLITH

GOAL FOR LAB 1

In this lab you will learn:

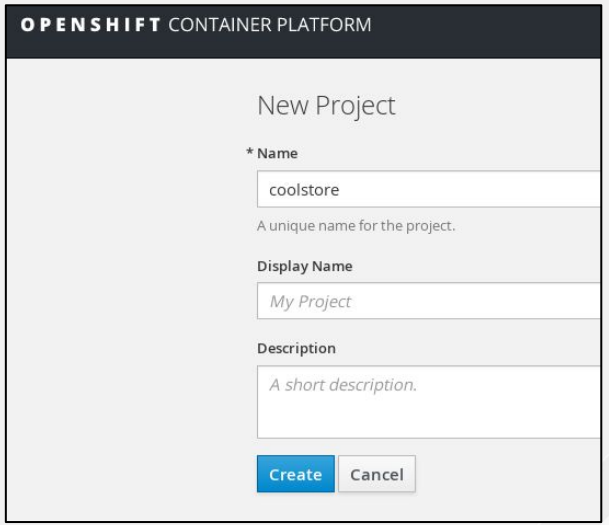
- How to deploy a Java EE application to a OpenShift container platform.
- Different alternatives to building and deploying an application.
- How to use deployment pipeline to deploy to different environments.

LAB 1 - FAST-MOVING MONOLITH

- In this lab, the coolstore monolith will be built and deployed to OpenShift from your local workstation demonstrating a typical Java application developer workflow
- A sample pipeline is included which will be used to deploy across dev and prod environment

STEP 1 - CREATE A PROJECT

- On the OpenShift web console home page, click on the **New Project** button
- Type “coolstore” in the **Name** and leave the rest to empty
- Click **Create**



OPENSHIFT TEMPLATES

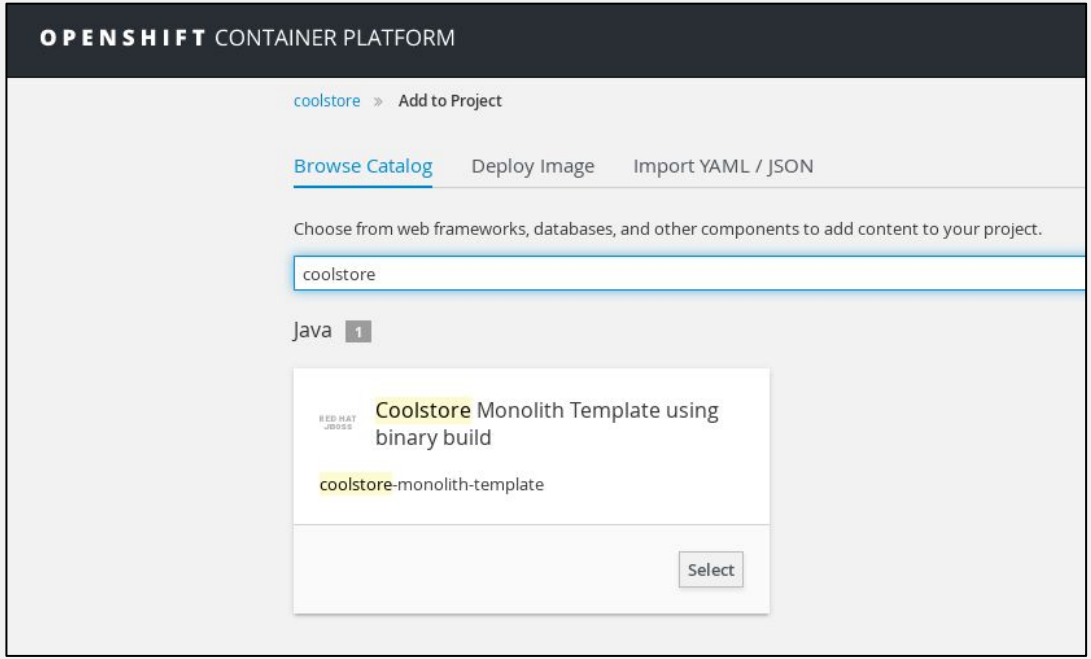
- Templates is an abstraction of services, build configuration, routes etc that can make up part or a full environment.
- OpenShift provides a number of templates for common infrastructure configurations, but you can also create your own templates.
- Using template developers can share environment configuration with other developers.

THE COOLSTORE MONOLITH TEMPLATE

- We have prepared a template for you to use to setup both the dev and production environment
- The template consists of
  - JBoss EAP 7 + PostgreSQL for Dev
  - JBoss EAP 7 + PostgreSQL for Prod
  - Jenkins + Pipeline definition.

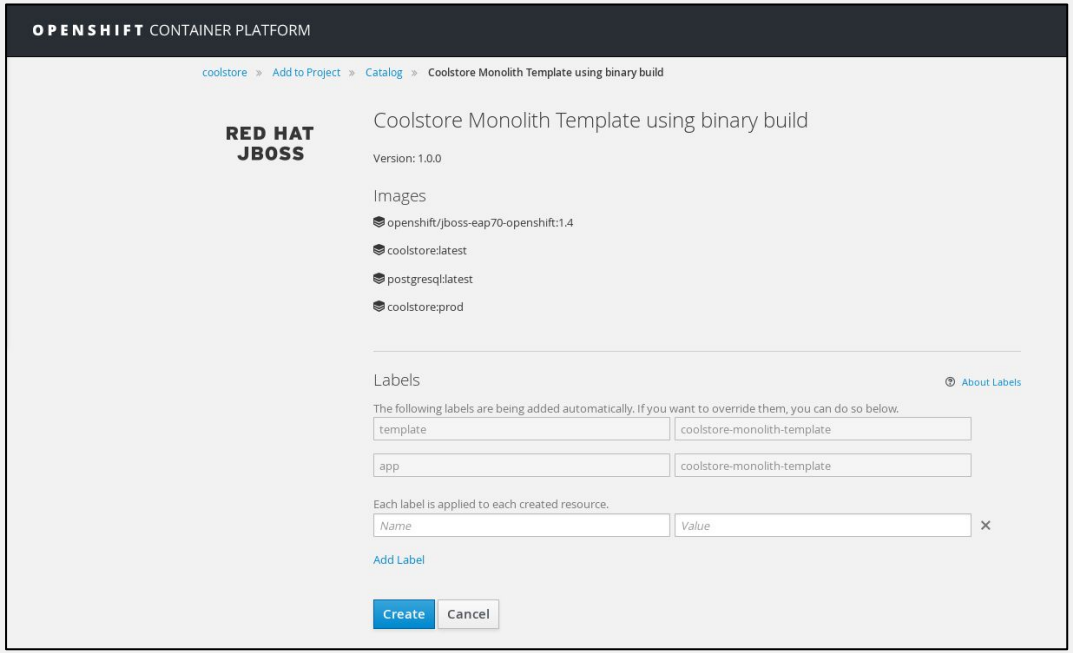
STEP 2a - INSTALL THE INFRASTRUCTURE

- From the Add to project page type **coolstore** in the filter and click **Select**



STEP 2b - INSTALL THE INFRASTRUCTURE

- From the template page **Create**

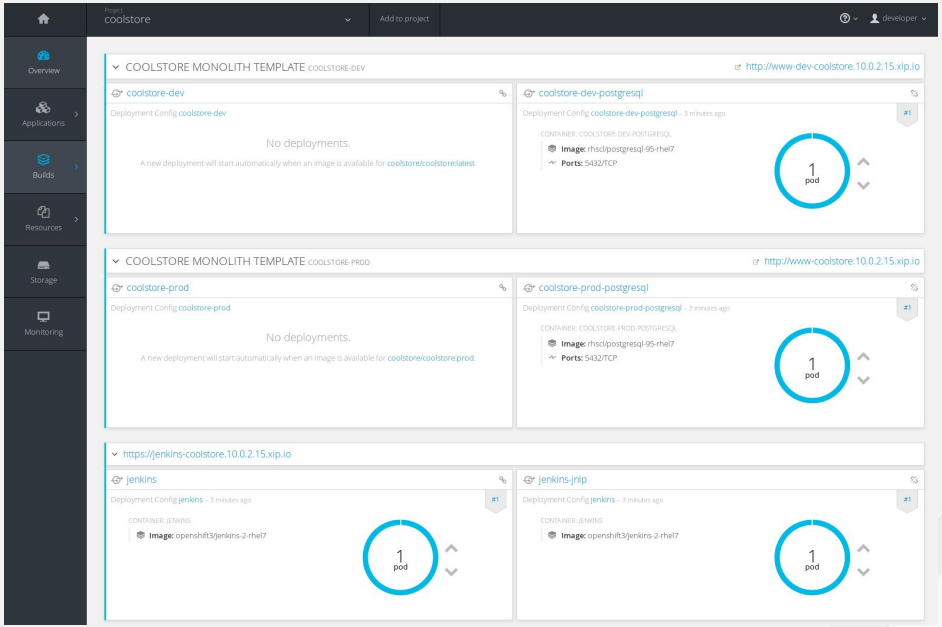


STEP 2c - INSTALL THE INFRASTRUCTURE

- From the Next steps page click **Continue To Overview**

STEP 2d - INSTALL THE INFRASTRUCTURE

- From the overview you can see the different environment being created
- Wait for them to finish
- Notice there is no deployment yet as there is no build yet.



TEMPLATE RESULT

- A number of OpenShift objects were created:
  - A Postgres database deployment, service, and route
  - A coolstore binary BuildConfig
  - Binary builds accept source code through stdin in later commands (vs. kicking off a build within an OpenShift pod)
  - Services, Routes and DeploymentConfig for coolstore-dev
  - Service Accounts and Secrets (for cluster permissions)
- Jenkins service and Jenkins slave service

BUILD CONFIGURATION

- There are many different possibility to build you application
- Since our application is a Java EE monolith application we are going to use JBoss EAP 7
- OpenShift and provides Source-To-Image (S2I) build image which has two optional inputs
  - Source - typically a maven project from a git repo
  - Binary - one or more artifacts to deploy into JBoss EAP 7.
- The output of a the S2I build process is a runnable docker image

STEP 3 - BUILD THE PROJECT

- In the terminal goto the monolith project directory:

```
$ cd ~/projects/monolith
```

- Build the project using maven and the openshift profile:

```
$ mvn clean package -Popenshift
```

- Select the coolstore project in the CLI

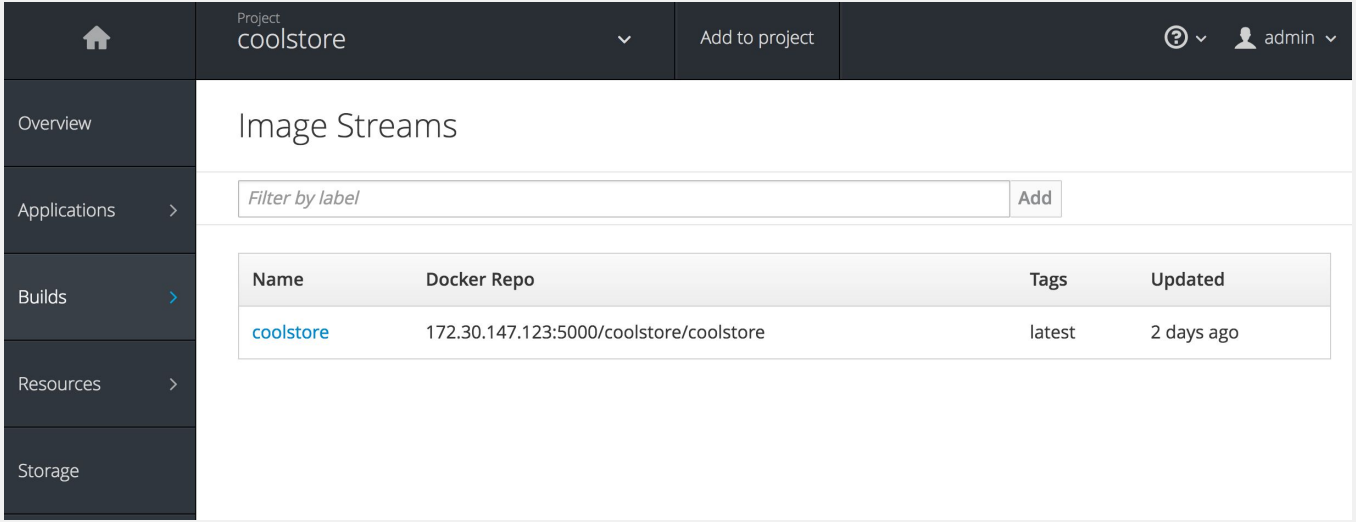
```
$ oc project coolstore
```

- Start the build:

```
$ oc start-build coolstore --from-file deployments/ROOT.war --follow
```

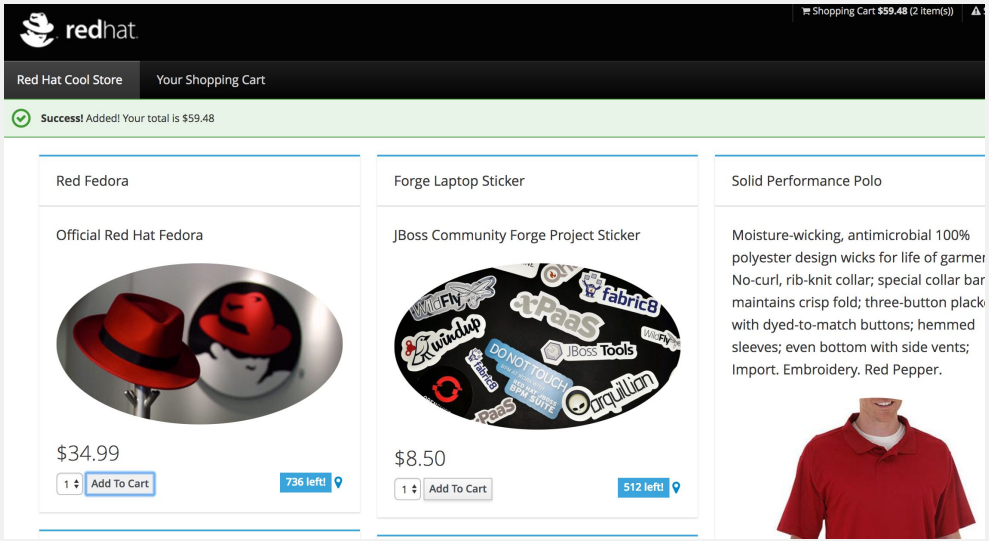
BINARY BUILDS

- Binary builds still produce Linux container images and are stored into the container registry.
- In the web console, navigate to Builds → Images to see the new image created as a result of combining the monolith .war file with the JBoss EAP builder image:



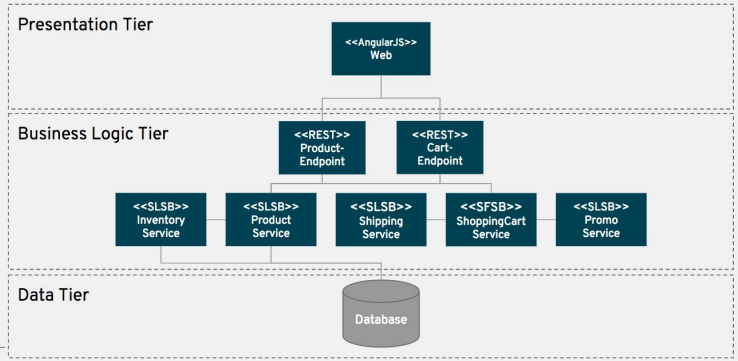
STEP 4 - VERIFYING THE APP IN DEV

- Once the deployment is complete, navigate to the application by clicking on the dev route in the OpenShift web console Overview and exercise the app by adding/removing products to your shopping cart:



THE APPLICATION

- The app contains an AngularJS web frontend which makes REST calls to its backend (e.g. /services/products)
- The app's backend contains services implement as Stateless and Stateful EJBs backed by JPA entities
- The product catalog and inventory are stored together in the database.



STEP 6 - VERIFY THE DATABASE

- Get the name of the postgresql dev pod:

```
$ oc get pods -l deploymentConfig=coolstore-dev-postgresql -o name
```

- Use remote shell (like ssh) to access pod

```
$ oc rsh <unique-podname>
```

- Start the postgres CLI client

```
sh-4.2$ psql -h localhost -d $POSTGRESQL_DATABASE -U $POSTGRESQL_USER
```

- Run SQL queries to see the content of the database

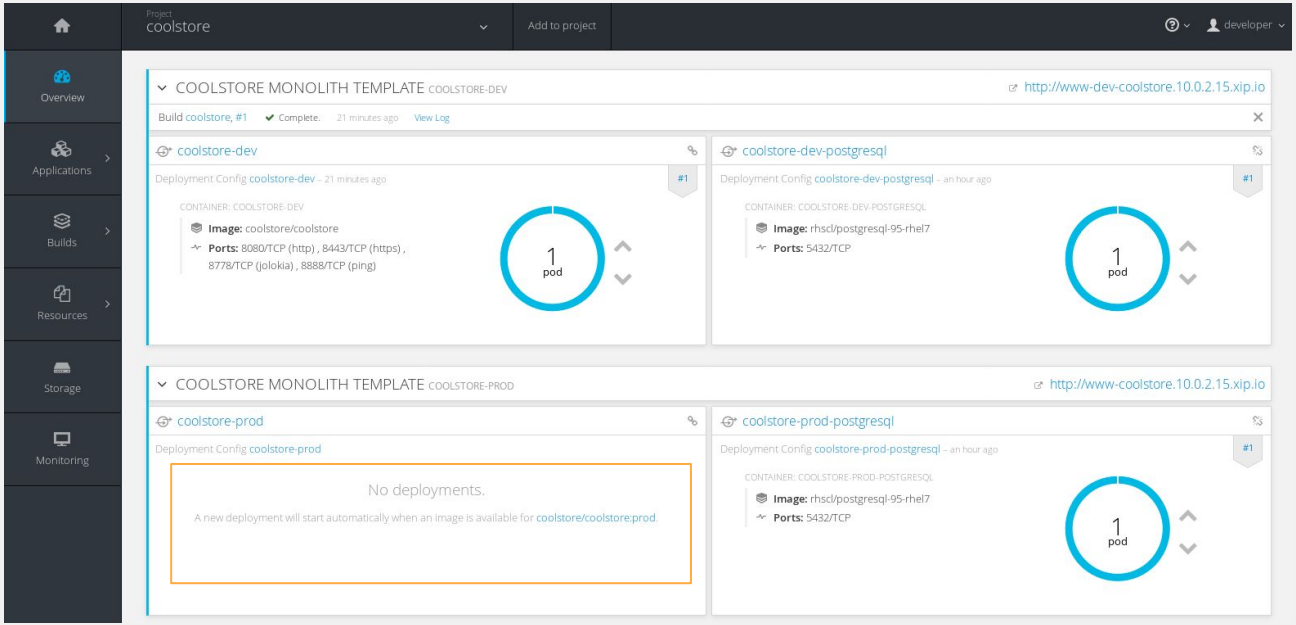
```
monolith=> select * from INVENTORY;
monolith=> select * from PRODUCT_CATALOG;
```

DIFFERENT ENVIRONMENTS

- The **coolstore-dev** build and deployment is responsible for building new versions of the app after code changes and deploying them to the development environment.
- Test and Production environments would typically use different projects and perhaps even different OpenShift clusters.
- For your simplicity we have created dev and prod in the same project and for resource limitation we don't have a common test or QA environment.

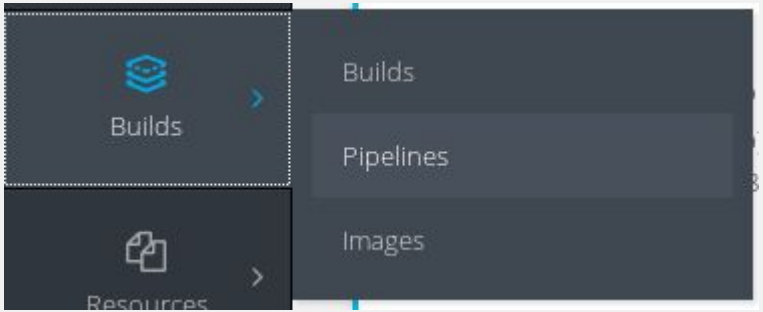
STEP 7a - PROMOTE THE BUILD TO PROD

- In the OpenShift Web console overview for the coolstore project we can see that we have a deployment for the dev environment, but not for the production environment.



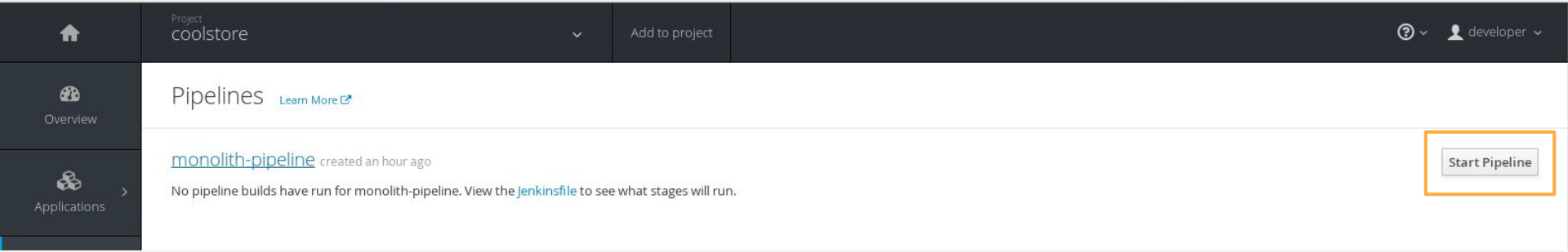
STEP 7b - PROMOTE THE BUILD TO PROD

- Click on **Builds** -> **Pipelines**



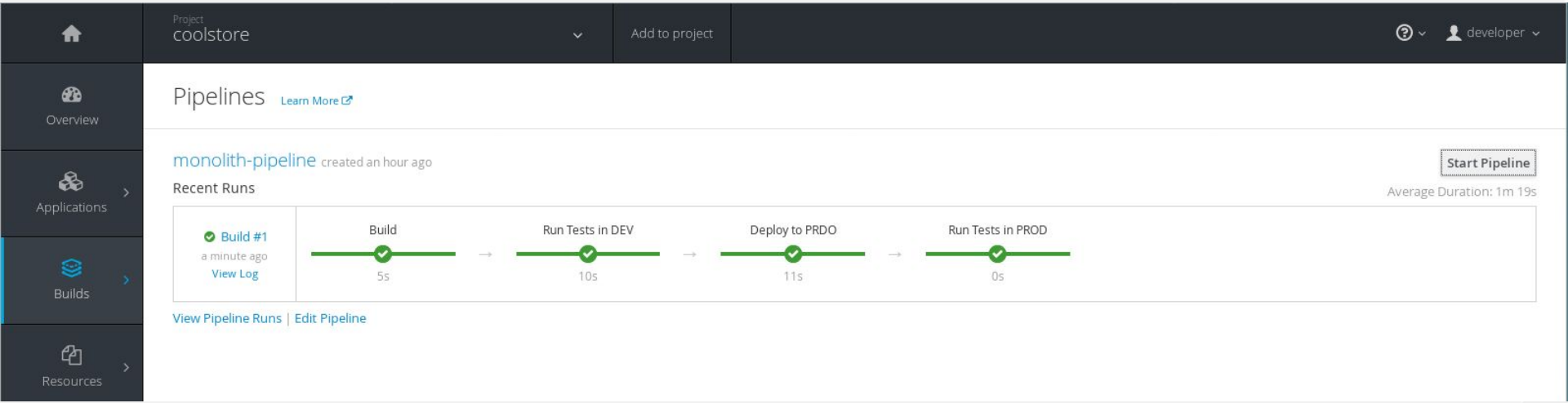
STEP 7c - PROMOTE THE BUILD TO PROD

- From the Pipelines page click **Start Pipeline** next to the monolith-pipeline



STEP 7d - PROMOTE THE BUILD TO PROD

- After a while the build pipeline starts executing, wait for it to finish.



THE PIPELINE

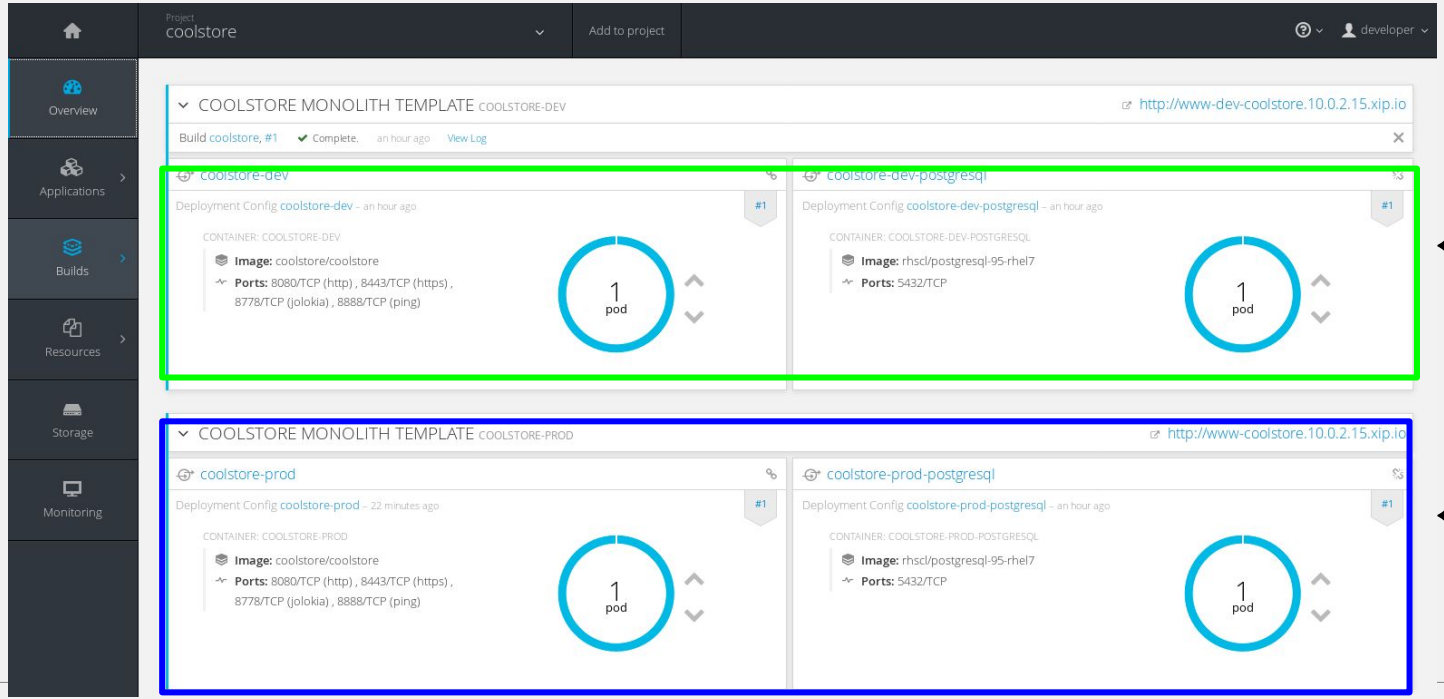
- If you click on the **monolith-pipeline** and go to **configuration**, you can edit the pipeline
- The current pipeline is mocking some steps, but this is where you in a real project would checkout code from a repository, build the code and deploy like we have done manually here.

NOTE: Because of network limitation in the lab environment it wouldn't be feasible to actually have a real pipeline. Here[1] is a example on a more complete pipeline in OpenShift that is using blue/green deployment.

[1]=https://gist.github.com/tqvarnst/9d63a29c2ac84c39103a54728f4c24d2

STEP 7e - PROMOTE THE BUILD TO PROD

- You should now have two identical copies of the app deployed, along with two databases:



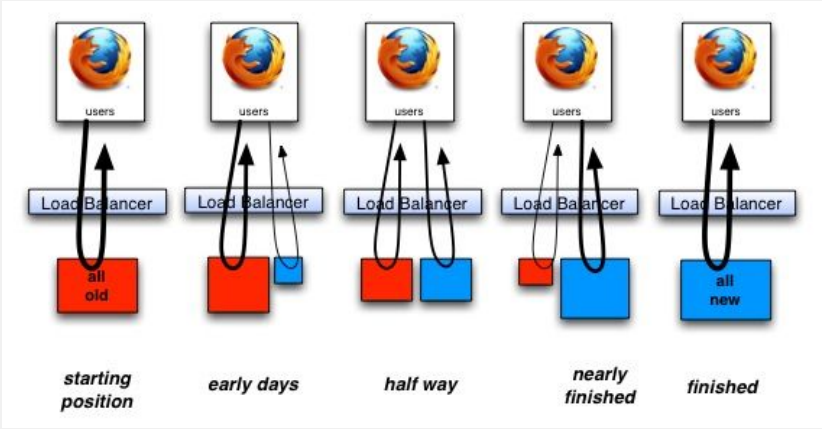
← Dev

← Prod

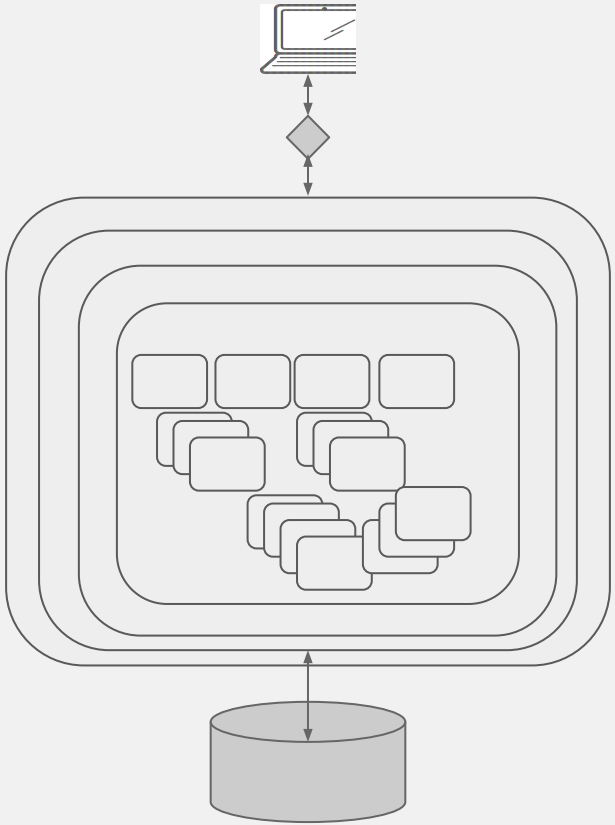
LAB 2
STRANGLING THE MONOLITH

LAB 2 - STRANGLING THE MONOLITH

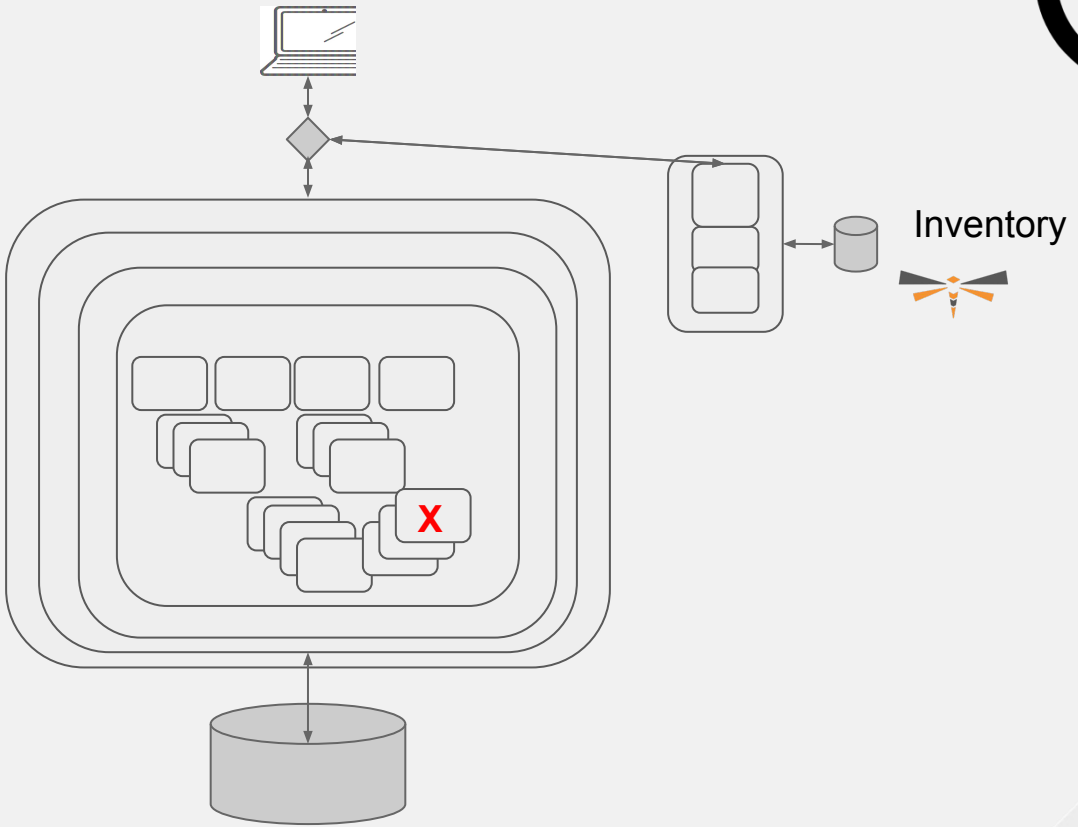
- Strangling - **incrementally** replacing functionality in app with something better (cheaper, faster, easier to maintain).
- As functionality is replaced, “dead” parts of monolith can be removed/retired.
- You can also wait for all functionality to be replaced before retiring anything!
- You can optionally include new functionality during strangulation to make it more attractive to business stakeholders.



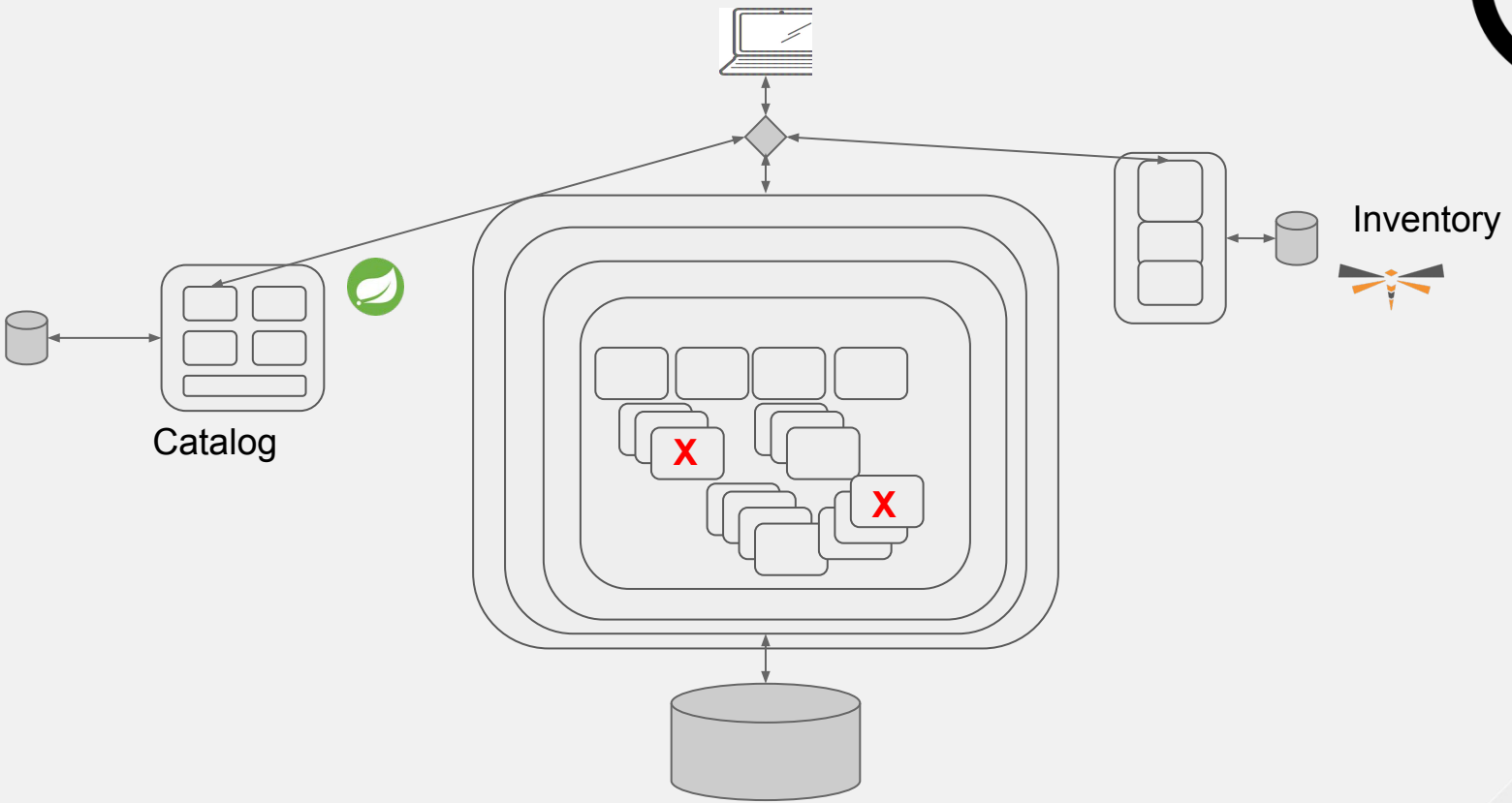
1 - STRANGLE MONOLITH



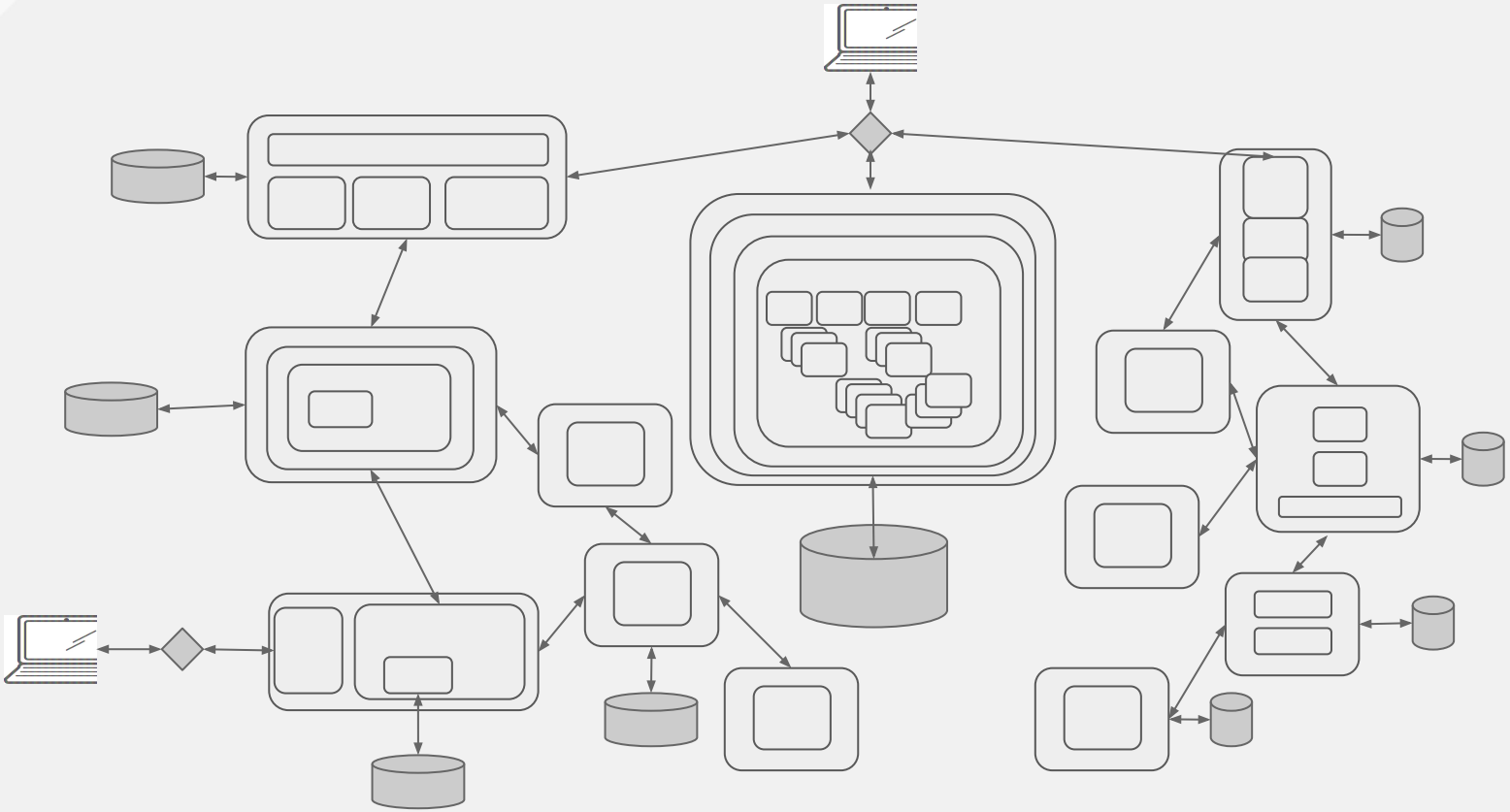
2 - ADD MICROSERVICE



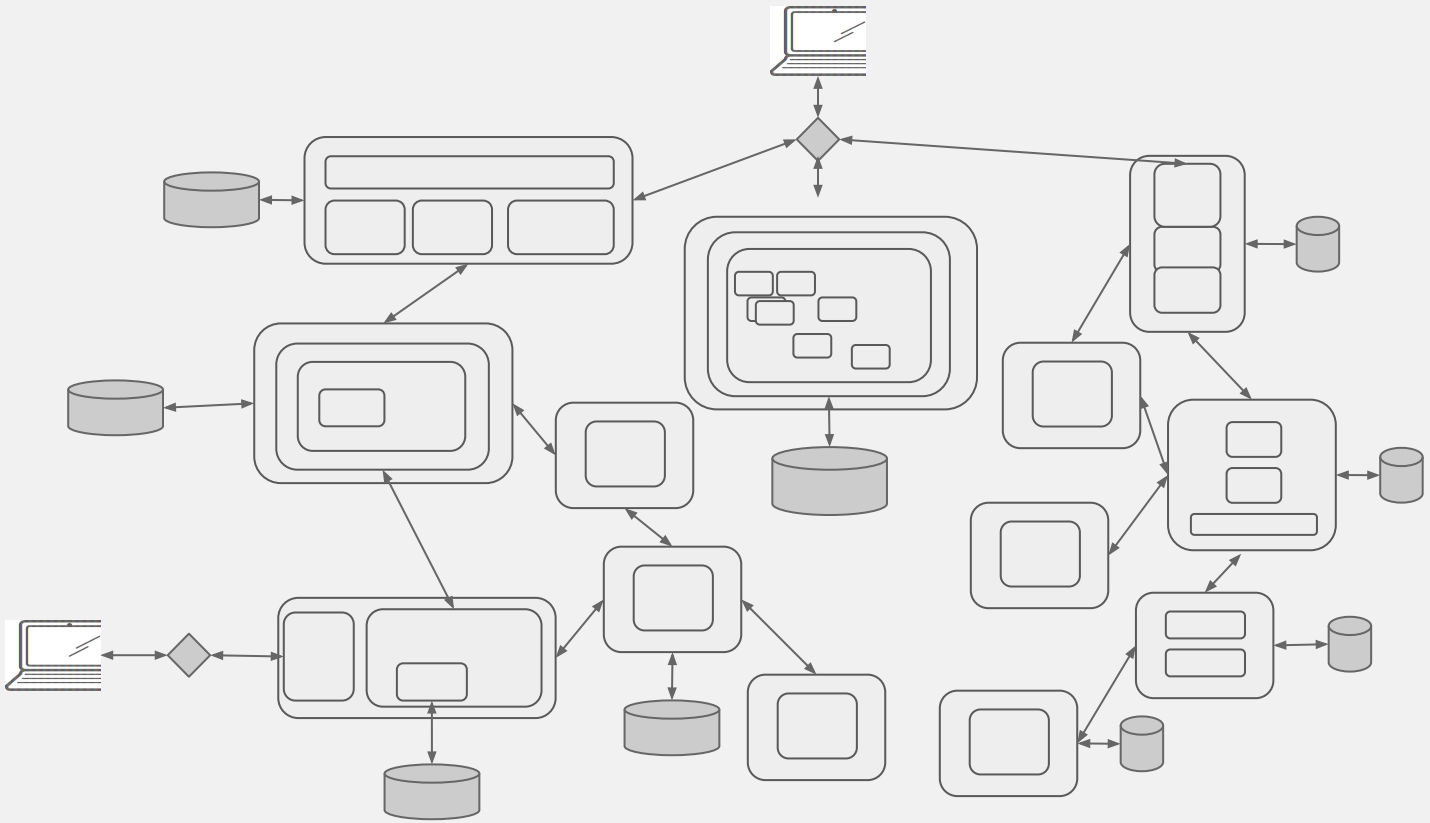
3 - AND ANOTHER



4 - SQUEEZE MONOLITH



5 - REMOVE DEAD



STEP 1 - STRANGELING THE MONOLITH

- In this lab, you will begin to ‘strangle’ the coolstore monolith by implementing two of its services as external microservices, split along business boundaries
- Once implemented, traffic destined to the original monolith’s services will be redirected (via OpenShift software-defined routing) to the new services
- First, build and deploy the catalog microservice (Based on Spring Boot)
- We will be using the Fabric8 Maven Plugin to automatically deploy to openshift:

```
$ cd ~/projects/catalog
$ mvn clean fabric8:deploy -Popenshift -DskipTests
```

STEP 2 - STRANGELING THE MONOLITH

- Once the new catalog service is deployed, test it with curl:

```
$ oc get route/catalog
NAME      HOST/PORT                              PATH      SERVICES   PORT      TERMINATION   WILDCARD
catalog   catalog-coolstore.10.0.2.15.xip.io             catalog    8080                    None

$ curl http://<copy-host-from-above>/services/products
[{"itemId":"329299","name":"Red Fedora","desc":"Official Red Hat
Fedora","price":34.99},{"itemId":"329199","name":"Forge Laptop Sticker","desc":"JBoss Community Forge Project
Sticker","price":8.5},
...
```

- You can also test it by clicking on the route to the catalog service and adding /services/products to the URL:

STEP 3 - STRANGELING THE MONOLITH

- Now we will do the same for the Inventory service, based on WildFly Swarm:

```
$ cd ~/projects/inventory
$ mvn clean fabric8:deploy -Popenshift  -DskipTests
```

- The catalog microservice is responsible for retrieving and returning a list of products and their descriptions (price, images, etc).
- The inventory service is responsible for retrieving and returning inventory for a given product.
- The catalog service does not specifically know about or care about inventory. In the next lab we will combine the two!

STEP 4 - STRANGELING THE MONOLITH

- Once the new service is deployed, test it with curl
- You should get a JSON object representing the inventory availability of product 329299

```
$ oc get route/inventory
NAME        HOST/PORT                              PATH      SERVICES    PORT      TERMINATION   WILDCARD
inventory   inventory-coolstore.10.0.2.15.xip.io             inventory   8080                    None

$ curl http://<copy-host-from-above>/services/inventory/329299
{"itemId":"329299","location":"Raleigh","quantity":736,"link":"http://maps.google.com/?q=Raleigh"}
```

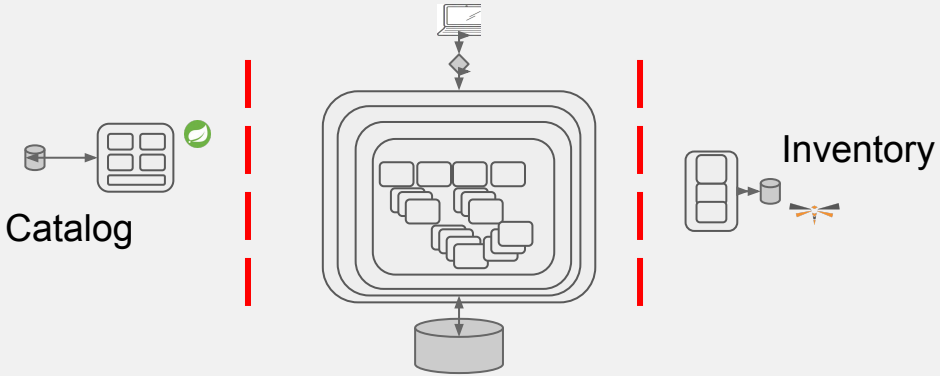
- You can also test it by clicking on the route to the inventory service and adding /services/inventory/329299 to the URL:

DIFFERENT ENVIRONMENTS

- The catalog and inventory microservice has a separate databases from the current monolith.
- This allows teams split along business concerns to independently develop, test, deploy and scale the service and its database

SUMMARY - STRANGLING THE MONOLITH

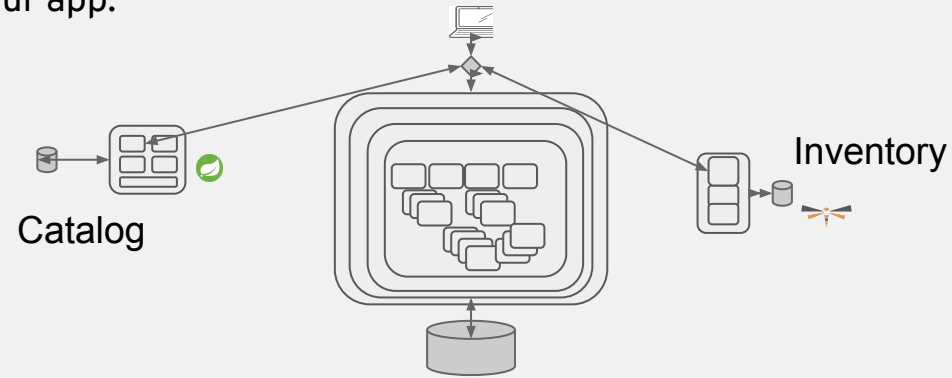
- You have begun to replace your monolith's services with new, lightweight Java-based microservices - congratulations!
- Although deployed, no web traffic will ever hit these new services as nothing (including the existing UI) knows of their existence.
- Microservice architectures have much more deployment flexibility but at the cost of increased complexity as the system becomes more distributed.
- In the next lab we'll discuss how to integrate the new microservices into the existing app while the development teams maintain complete development and deployment autonomy.



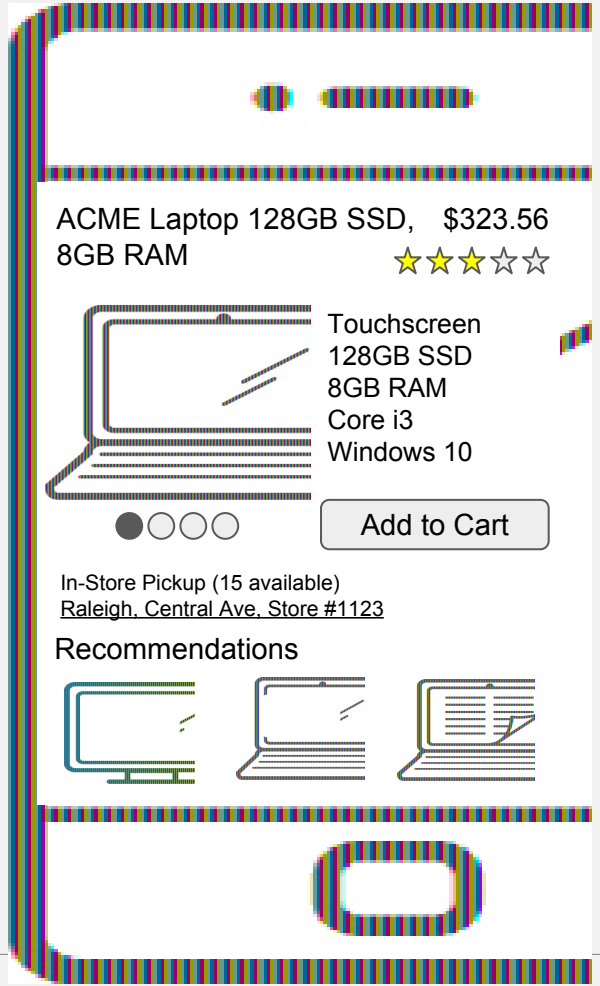
LAB 3
MICROSERVICE INTEGRATION PATTERNS

MICROSERVICE INTEGRATION PATTERNS

- In previous labs, we created two new microservices with the intention of replacing functionality (“strangling”) the monolithic application.
- Currently no traffic is routed to them.
- If you were to re-route traffic from the monolith’s /services/products API to the new catalog service’s /services/catalog endpoint, you would be missing the inventory data.
- In this lab we will consider different options and architectures for integrating the microservices’ functionality into our app.

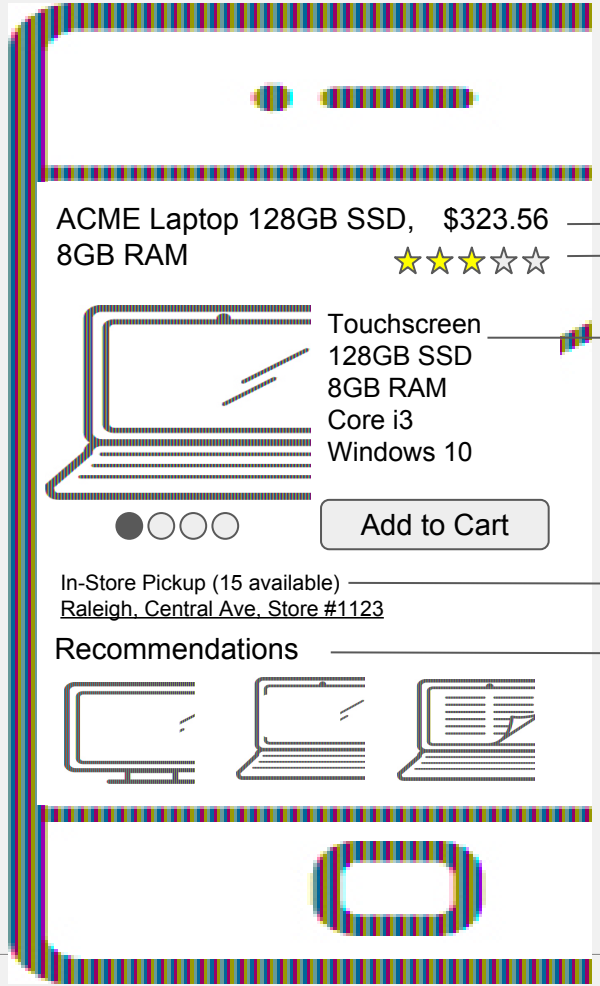


EXAMPLE



- This is very similar to our CoolStore retail application!
- UI constructed from multiple microservices implemented across business boundaries
- Typically runs in constrained environments like mobile
- Real-world apps consist of 100s of services

EXAMPLE



Pricing Engine

Reviews

Details/Specifications

Location-based availability

People who purchased also...

STEP 1 - MICROSERVICE INTEGRATION PATTERNS

- OpenShift routes can be created based on path so we can create a route to **/services/products** that points to our product catalog service instead of the monolith.
- Get the hostname for the **www-dev** route:

```
$ oc get route/www-dev
NAME      HOST/PORT                          PATH      SERVICES        PORT      TERMINATION   WILDCARD
www-dev   www-dev-coolstore.10.0.2.15.xip.io            coolstore-dev   <all>                   None
```

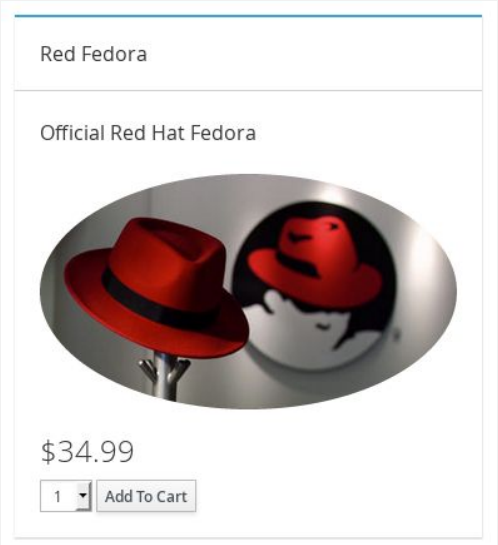
- Using the web OpenShift web console create a route via **Application** > **Routes** - **Create Route** and use the following settings:

```
Name: www-dev-catalog
Hostname: <copy-from-above>
Path: /services/products
Service: catalog
```

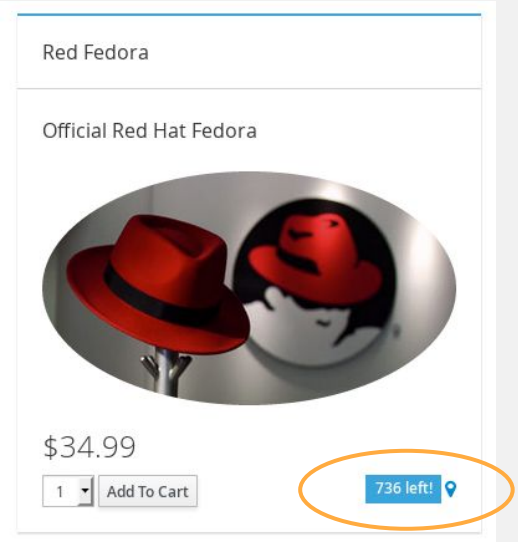
STEP 2 - MICROSERVICE INTEGRATION PATTERNS

- Compare the dev coolstore web site with the production

DEV



PROD



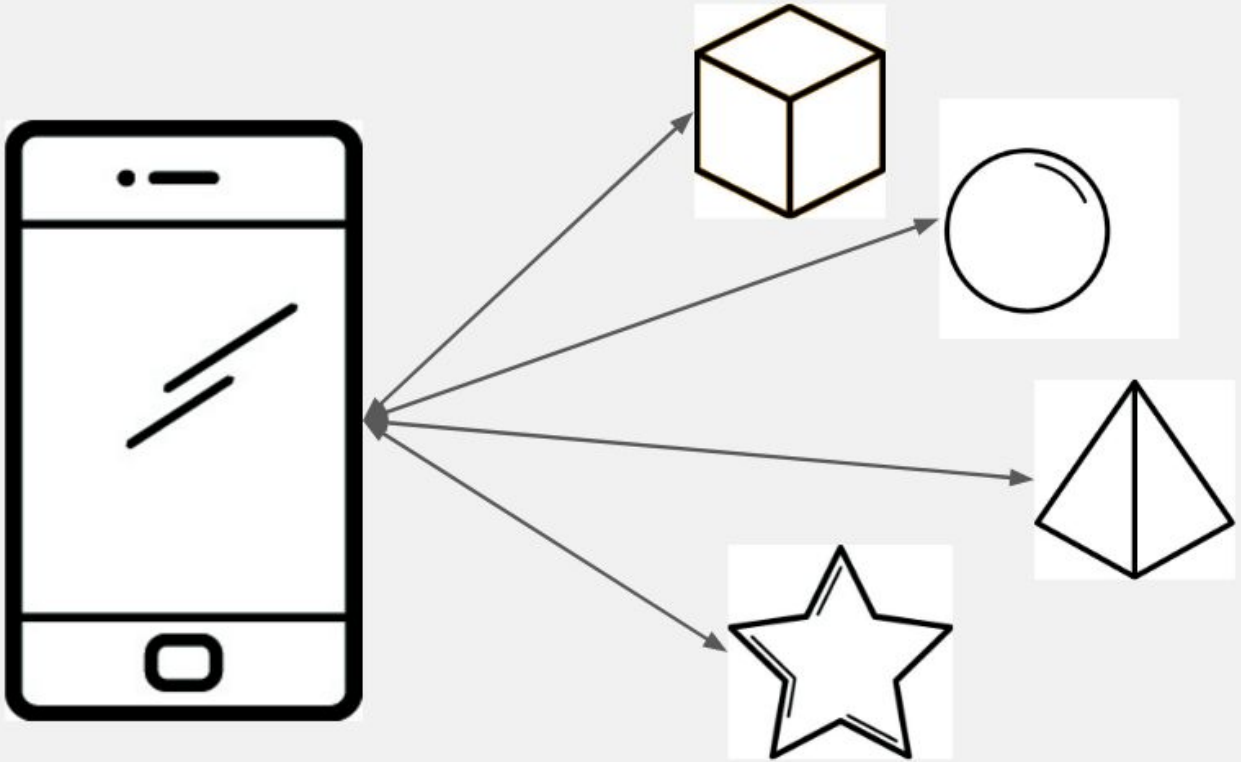
STEP 3 - MICROSERVICE INTEGRATION PATTERNS

- As we can see the dev instance is now missing the inventory data.
- That is because the monolith application did a single call to **/services/products** and got a list back combining both catalog data and inventory data.
- Since we need another solution we will go ahead and delete the route we created like this:

```
$ oc delete route/www-dev-catalog
```

- Now we will discuss what our alternatives are for integrating our services

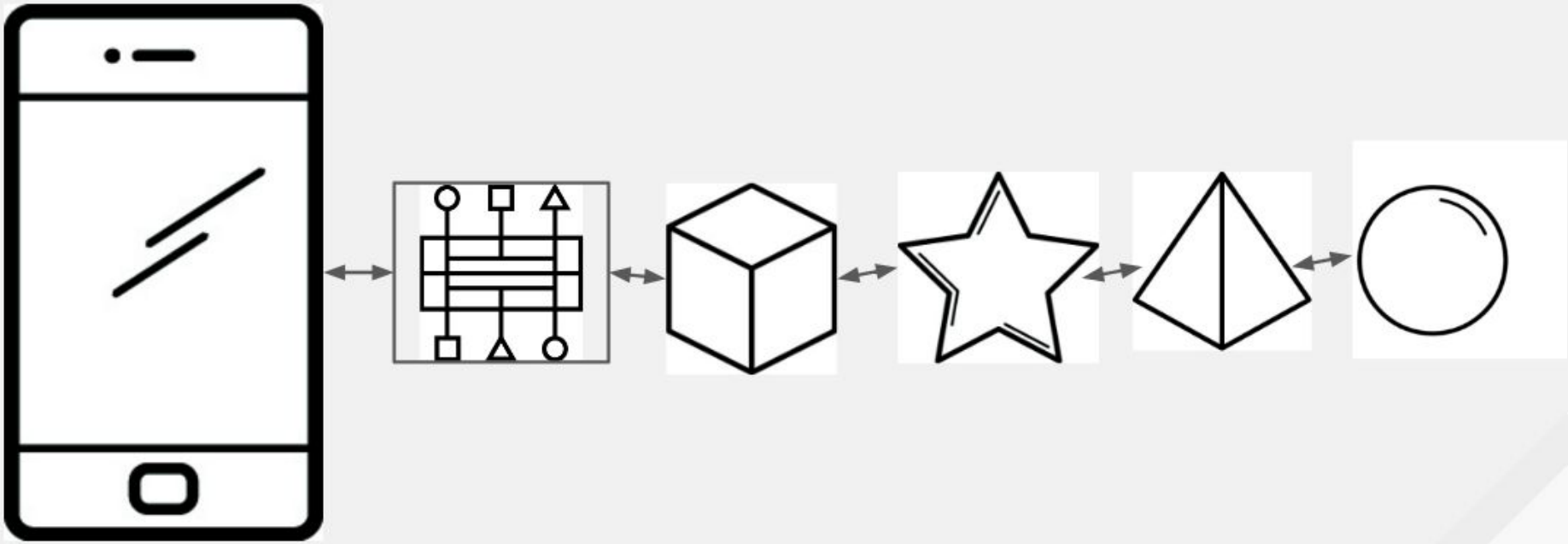
ALT 1: CLIENT AGGREGATION



CLIENT AGGREGATION

- Microservices implement functionality previously found in monoliths
- Some microservices depend upon other microservices
- Client applications (e.g. web browsers) depend on all of them in one way or another, and are usually “outside the firewall”
- This option means that the client side code (typically run in a browser) is responsible for talking to each microservice and aggregating/combining the results
- Client aggregation benefits
  - No development bottleneck on the server / ESB-style funnel
- Client aggregation drawbacks
  - Network bandwidth/latency of multiple calls to multiple microservices
  - Unfriendly protocols - web proxies, ports, etc
  - Difficulty in later refactoring microservices - the client must change too
  - Client application code complexity

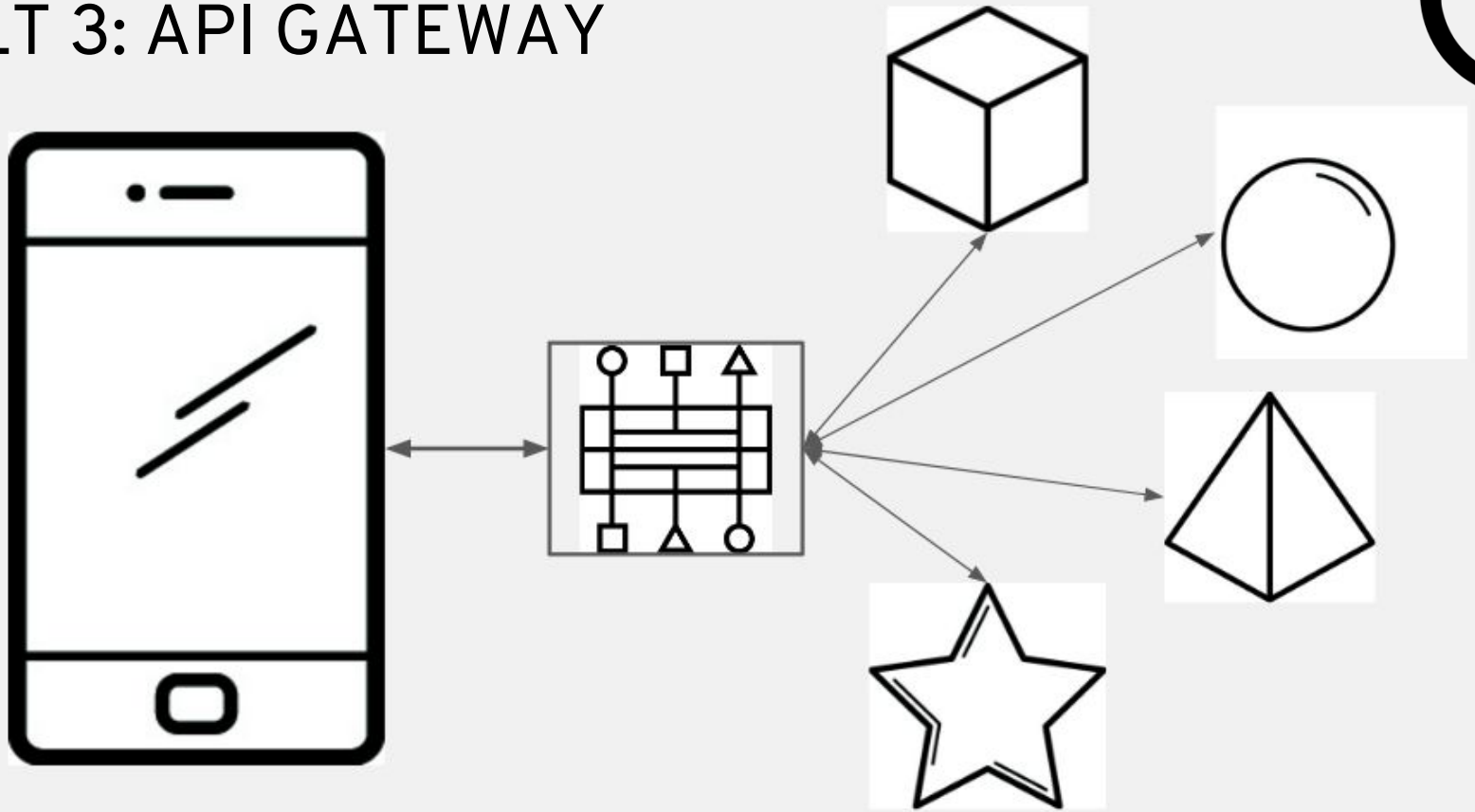
ALT 2: SERVICE CHAINING



SERVICE CHAINING

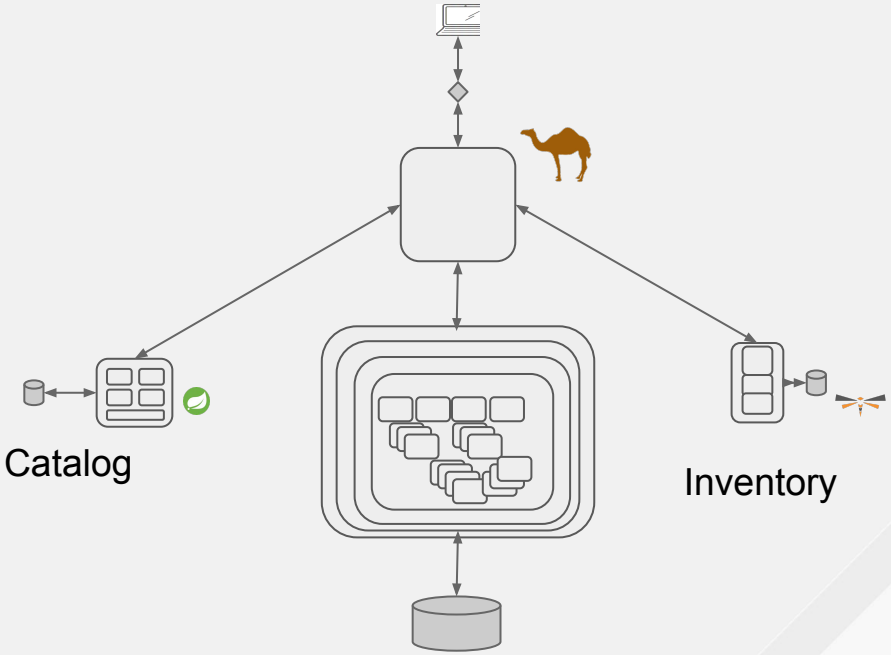
- Chaining means that one microservice calls another, which calls another, etc.
- A complete chain is typically not desirable or necessary, but short chains are OK
- Chaining benefits
  - Client code simpler - there is only a single entry into the chain
  - Less network bandwidth (also due to single entry point)
- Chaining drawbacks
  - Potential for cascading failures (resilience patterns can help minimize this)
  - Complex “stack traces” when things go wrong (tracing libraries a must)
  - Exposes internal structure of app logic (the first microservice in the chain would be difficult to change)

ALT 3: API GATEWAY



API GATEWAY

- The API Gateway pattern:
  - Keeps business logic on server side
  - Aggregates results from back-end services
- API Gateway Pattern benefits
  - Encapsulates internal structure of application's services
  - Less chatty network traffic
  - Simplified client code (no aggregation)
- Drawbacks
  - Possible bottleneck depending on difficulty of adding new services



STEP 4 - MICROSERVICE INTEGRATION PATTERNS

- In this lab, the previously developed microservices will be placed behind a gateway service
- The client application will then call the gateway service to retrieve its data
- This will “strangle” the monolith by replacing its catalog/inventory services with new microservices.
- First, deploy the API gateway:

```
$ cd ~/projects/gateway
$ mvn clean fabric8:deploy -Popenshift -DskipTests
```

STEP 5a - CAMEL DSL

- The Coolstore Gateway microservice is a Spring Boot application that implements its logic using an Apache Camel *route*.
- Open the Gateway project in Visual Studio Code

```
$ code .
```

- Open `src/main/java/com/redhat/coolstore/api_gateway/ProductGateway.java`
- Let's look at some implementation details:

```
        restConfiguration()
                .contextPath("/services").apiContextPath("/services-docs")
                .apiProperty("host", "")
                .apiProperty("api.title", "CoolStore Gateway API")
                .apiProperty("api.version", "1.0")
                .component("servlet")
                .bindingMode(RestBindingMode.json);
```

STEP 5b - CAMEL DSL

- The beginning of this route uses the Camel Java DSL (Domain-Specific Language) to configure the REST system and define the base paths of the API itself (`/services`) and paths to Swagger documentation (`/services-docs`).

```
restConfiguration()
        .contextPath("/services").apiContextPath("/services-docs")
        .apiProperty("host", "")
        .apiProperty("api.title", "CoolStore Gateway API")
        .apiProperty("api.version", "1.0")
        .component("servlet")
        .bindingMode(RestBindingMode.json);
```

STEP 5c - CAMEL DSL

```
rest("/products").description("Access the CoolStore products and their availability")
    .produces(MediaType.APPLICATION_JSON_VALUE)
```

- This begins the REST DSL portion, defining the primary access point for the catalog of products (`/products`) and the format of the data it produces (JSON)

```
.get("/").description("Retrieves the product catalog, including inventory availability").outType(Product.class)
    .route().id("productRoute")
              .setBody(simple("null"))
              .removeHeaders("CamelHttp*")
              .recipientList(simple("http4://{{env:CATALOG_ENDPOINT:catalog:8080}}/api/catalog")).end()
              .unmarshal(productFormatter)
               .split(body()).parallelProcessing()
               .enrich("direct:inventory", new InventoryEnricher())
          .end()
.endRest();
```

- This configures the endpoint for retrieving a list of products by first contacting the Catalog microservice, `.split()`ing the resulting list, and *enriching* (via `enrich()`) each of the products with its inventory by passing each product to the direct:inventory route.

STEP 5d - CAMEL DSL

```
from("direct:inventory")
    .id("inventoryRoute")
    .setHeader("itemId", simple("${body.itemId}"))
            .setBody(simple("null"))
            .removeHeaders("CamelHttp*")
    .recipientList(simple("http4://{{env:INVENTORY_ENDPOINT:inventory:8080}}/api/inventory/${header.itemId}")).end()
    .setHeader("CamelJacksonUnmarshalType", simple(Inventory.class.getName()))
    .unmarshal().json(JsonLibrary.Jackson, Inventory.class);
```

- This is the direct:inventory route, which takes in a Product object (in the body()) and calls out to the Inventory microservice to retrieve its inventory. The resulting inventory is placed back into the Camel exchange for enrichment by the enricher:

STEP 5e - CAMEL DSL

```
@Override
public Exchange aggregate(Exchange original, Exchange resource) {

       // Add the discovered availability to the product and set it back
       Product p = original.getIn().getBody(Product.class);
       Inventory i = resource.getIn().getBody(Inventory.class);
       p.setQuantity(i.getQuantity());
       p.setLocation(i.getLocation());
       p.setLink(i.getLink());
       original.getOut().setBody(p);

       return original;

}
```

- This is the enricher logic which takes the Product and matching Inventory objects, *enriches* the Product object with information from the Inventory object, and returns it.
- The resulting list sent back to the client is the list of products, each of which is enriched with inventory information
- The client then renders the aggregate list in the UI.

OPENSHIFT ROUTING

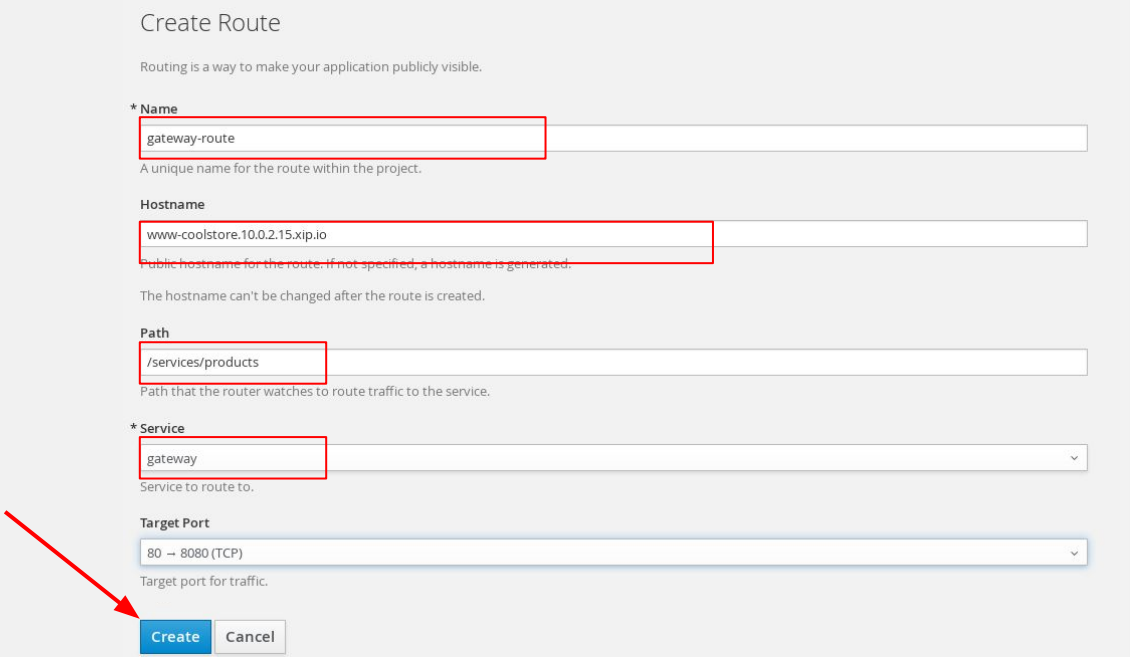
- Now that the gateway microservice is deployed, let's hook it into the application using OpenShift routing.
- A route is a way to expose a service by giving it an externally-reachable hostname like www.example.com, or in our example `www-coolstore.<domain>`
- Routes can be created using `oc expose` command line, or through the GUI.
- *Path based* routes specify a path component that can be compared against a URL such that multiple routes can be served using the same underlying service/pod, each with a different path.
- In this case, we already have a route that sends all traffic destined for our monolith to the monolith deployment.
- We want to setup a route such that when the monolith's GUI calls `/services/products`, it is re-routed to our new CoolStore gateway microservice, thus completing the partial strangulation of the Inventory and Product Catalog features of our app.

STEP 6a - CREATING A ROUTE

- Navigate to Applications → Routes to list the current routes.
- Notice the `www` route is the primary route for our monolith:

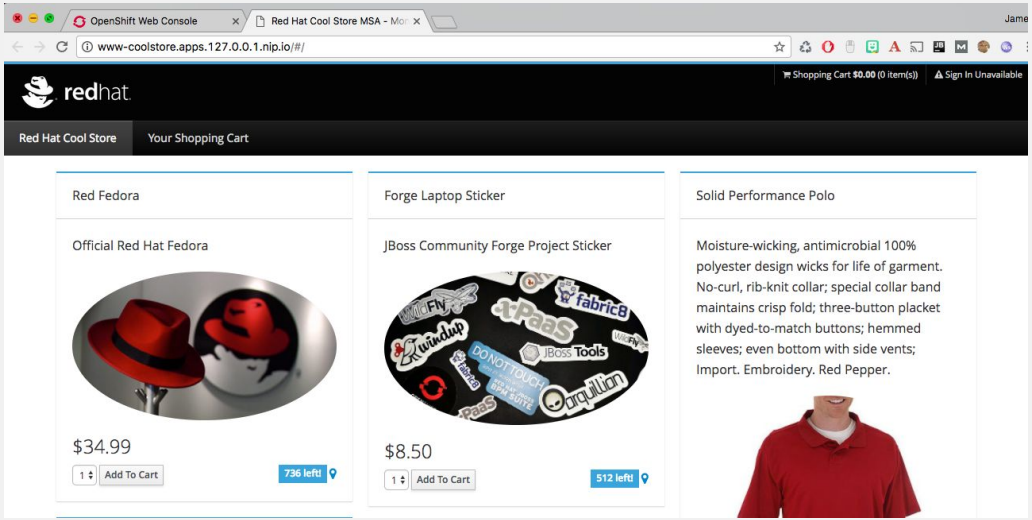
- Click *Create Route* to begin creating a new route with the following values:
  - Name: gateway-redirect
  - Hostname: The full hostname of the existing route **as seen above** (without the `http://`). For example,
    - **`www-coolstore.10.0.2.15.xip.io`**
  - Path: **`/services/products`**
  - Service: **`gateway`**
  - Leave other values as-is (see next page for complete example)
- Click Create to create the route

STEP 6a - CREATING A ROUTE



STEP 6b - CREATING A ROUTE

- Test the new route by visiting the application UI (click on the coolstore-prod route in the Overview). It will be no different than the original monolith. How do you know your new microservices are being used in place of the original services?



STEP 7a - FILTER OUT PRODUCTS

- Let's pretend that there is a lengthy and cumbersome process for getting products and inventories into and out of the backend system.
- We have a high priority task to remove the Red Hat Fedoras from the product list due to a manufacturing defect.
- Let's filter the product out of the result using our gateway. Open the gateway source code file using visual code editor and un-comment the lines that implement a filter based on product ID (around line 80).
- The highlighted code shows you the predicate used for the filter

~/coolstore/gateway/src/main/java/com/redhat/coolstore/api_gateway/ProductGateway.java

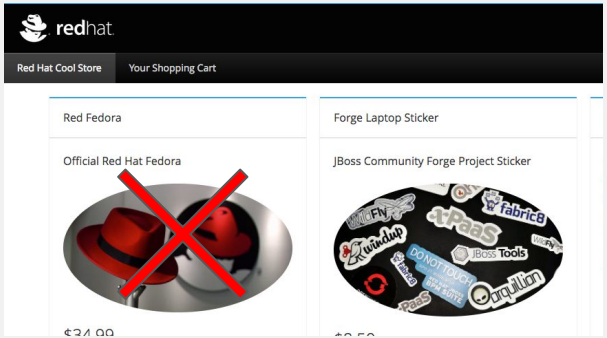
```
//
// Uncomment the below lines to filter out products
//
//            .process(exchange -> {
//                   List<Product> originalProductList = (List<Product>)exchange.getIn().getBody(List.class);
//                   List<Product> newProductList = originalProductList.stream().filter(product ->
//                                                !("329299".equals(product.itemId)))
//                                         .collect(Collectors.toList());
//                   exchange.getIn().setBody(newProductList);
//               })
```

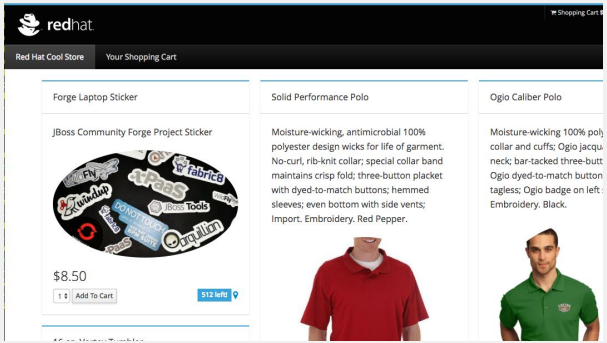
STEP 7b - FILTER OUT PRODUCTS

- Re-deploy the modified code using the same procedure as before:

```
$ cd ~/projects/gateway
$ mvn clean fabric8:deploy -Popenshift -DskipTests
$ oc logs -f dc/gateway
....
--> Success       # --> wait for it!
```

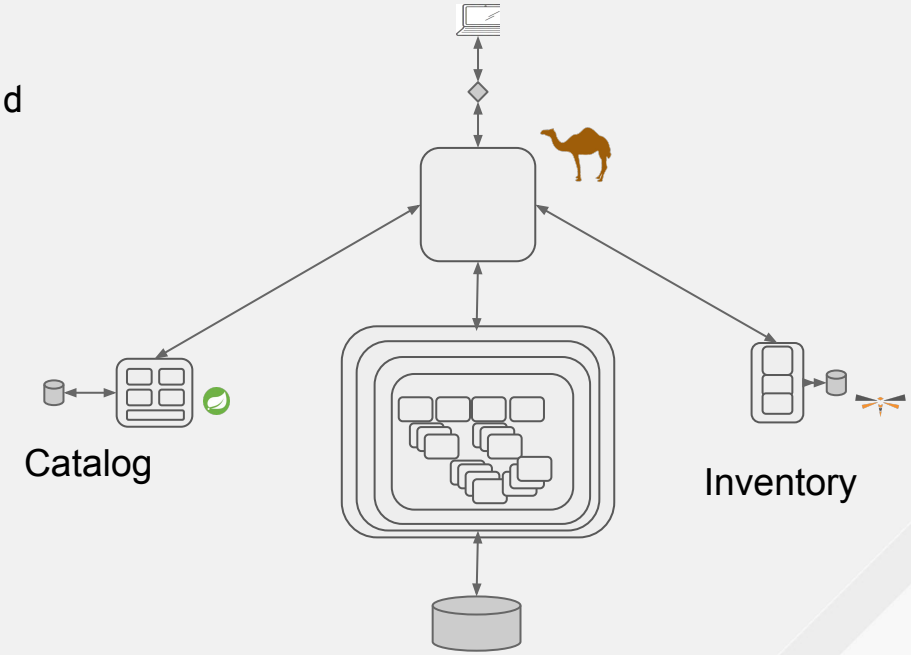
- Once the new version of the code is deployed and up and running, reload the browser and the Red Hat Fedora product should be gone:





LAB 3 - SUMMARY

- Our monolith's UI is now talking to our new Camel-based API Gateway to retrieve products and inventories from our WildFly Swarm and Spring Boot microservices!
- Further strangling can eventually eliminate the monolith entirely.

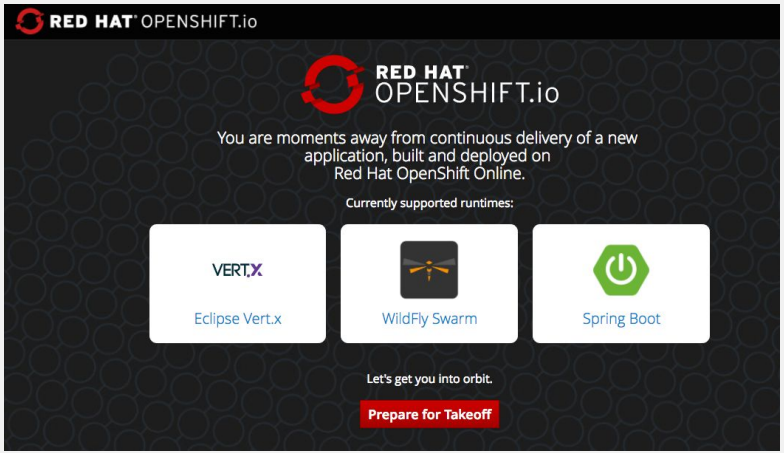


YOU MADE IT!

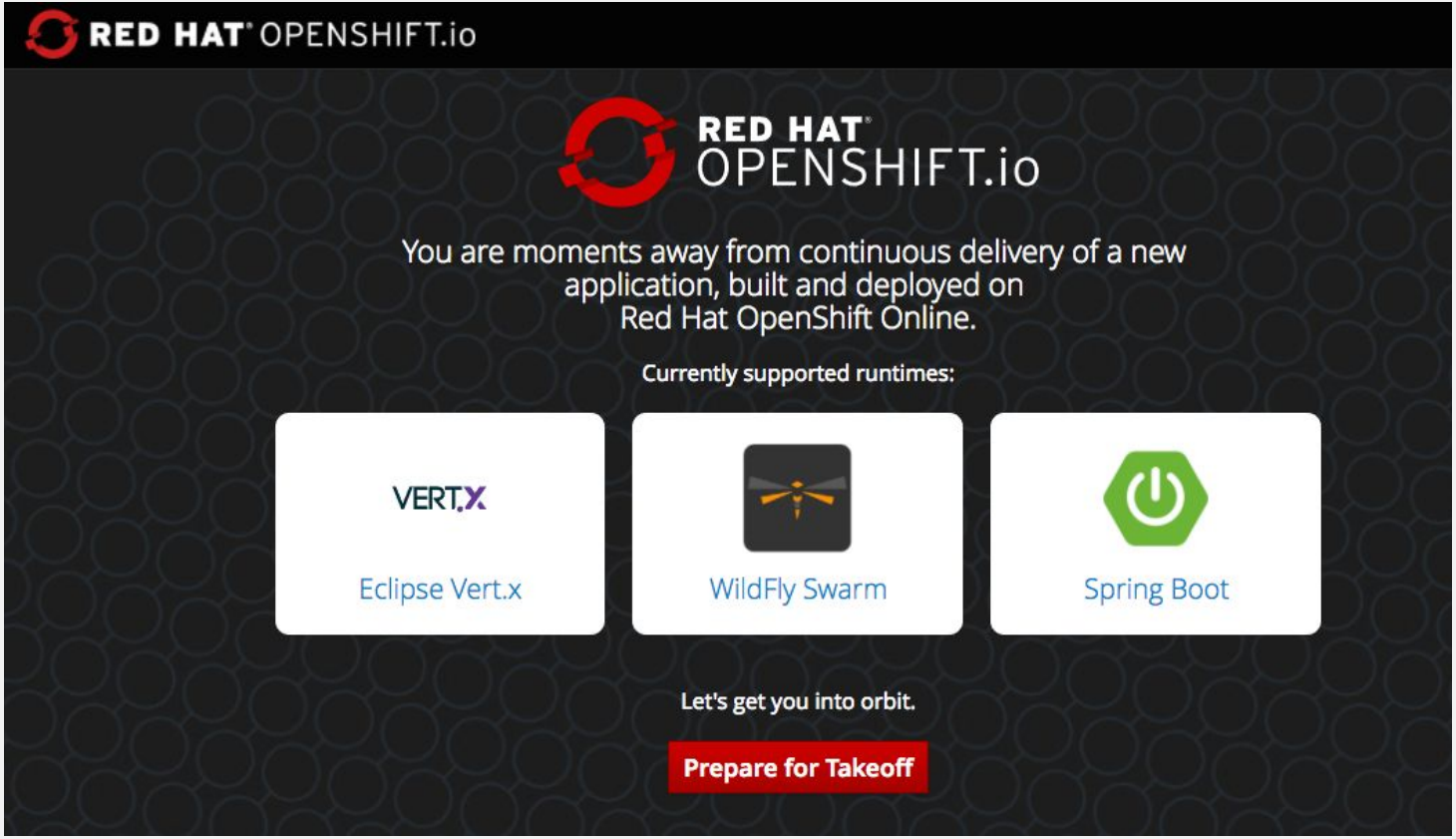
EXTRA LAB

EXTRA LAB

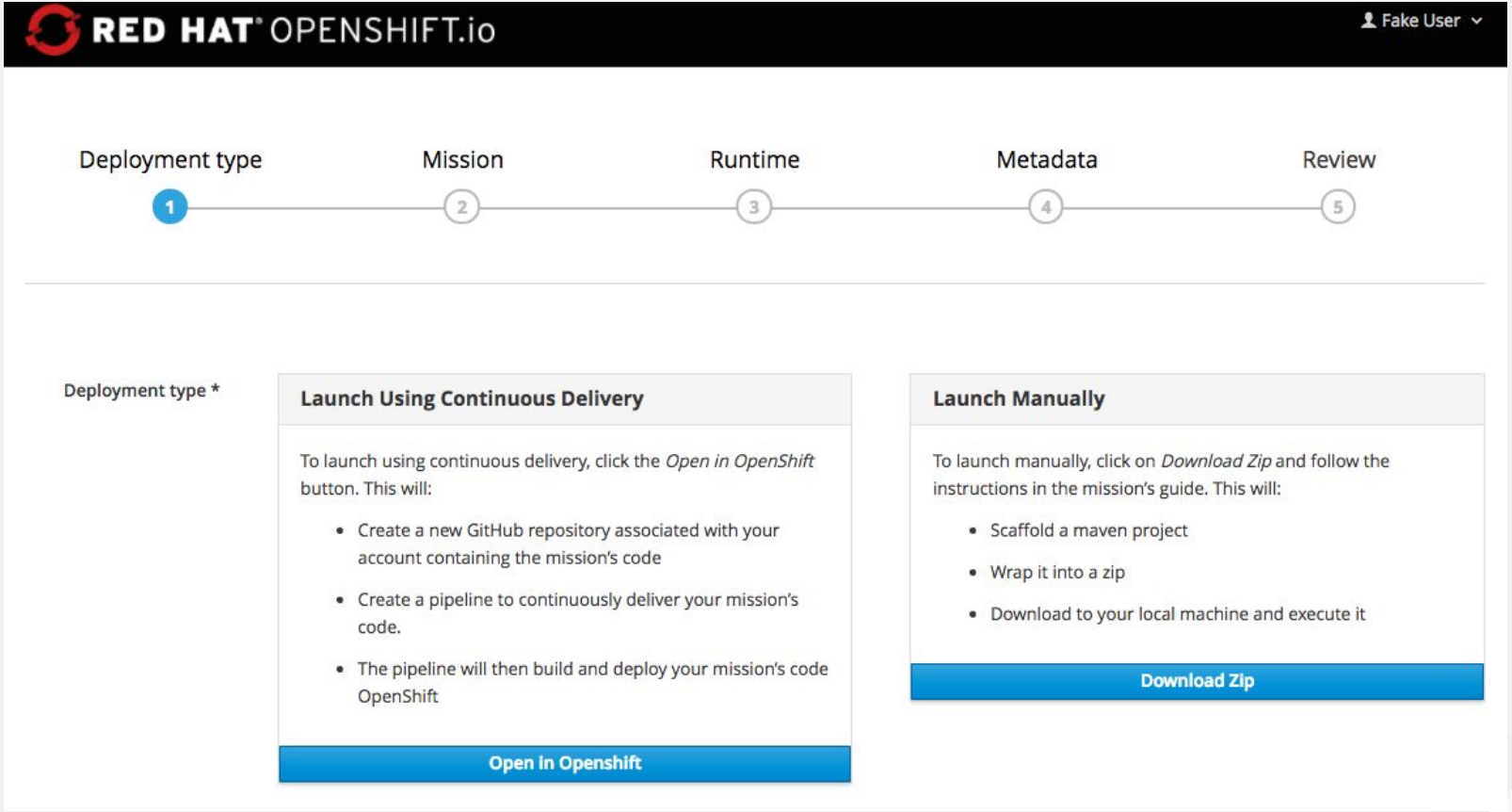
- If there is still time left you can do this extra lab
- Yesterday (3rd of May) Red Hat released OpenShift.io beyond developer preview. In this lab you are going to deploy the catalog service to OpenShift.io.
- To get access to Openhift.io you need to do the following:
  - Go to developers.redhat.com and register as a developer (it's free)
  - Go to manage.openshift.com and login with your redhat account
  - Go to launch.openshift.io and create a Spring Boot project online using REST



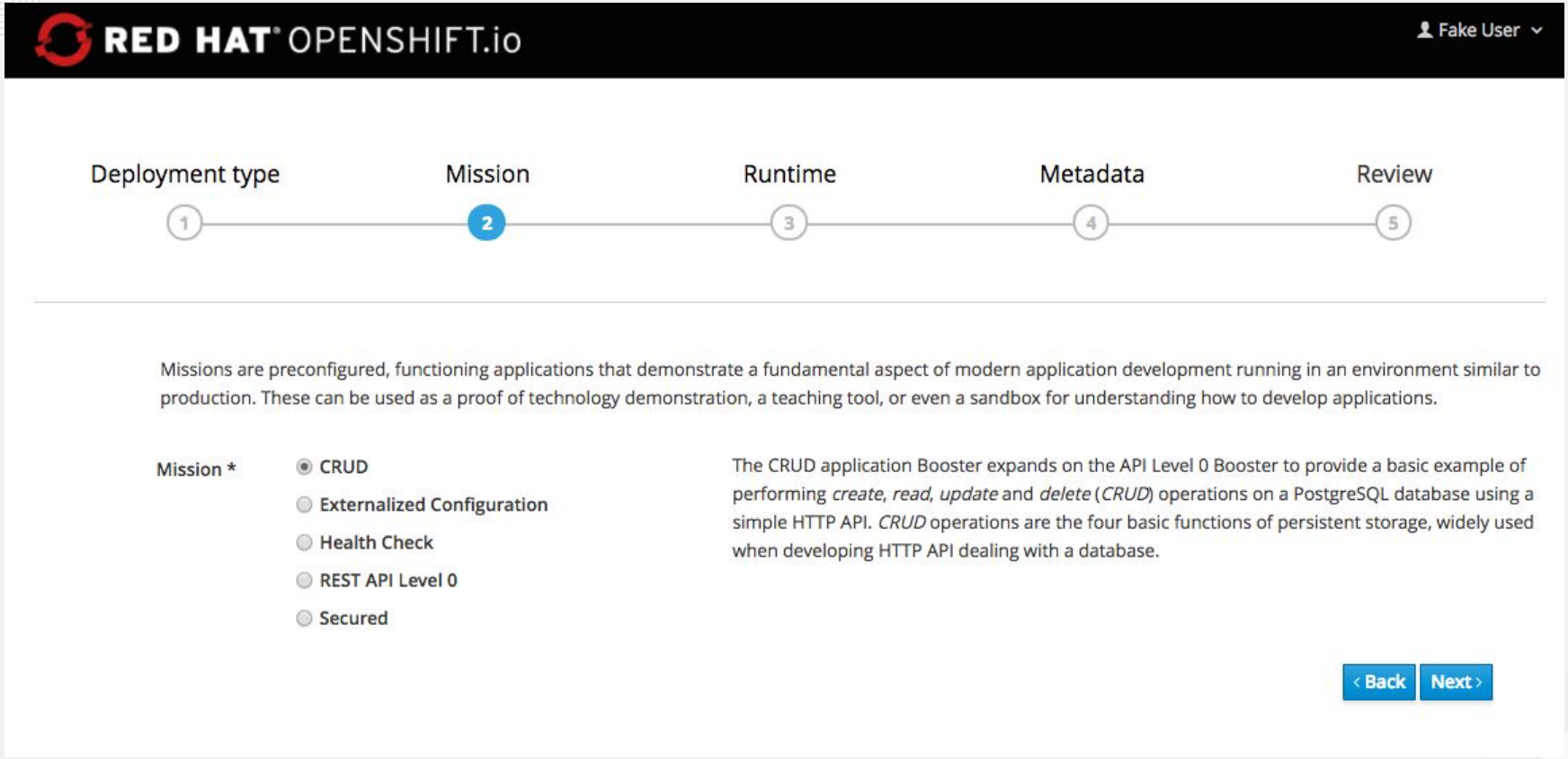
launch.openshift.io



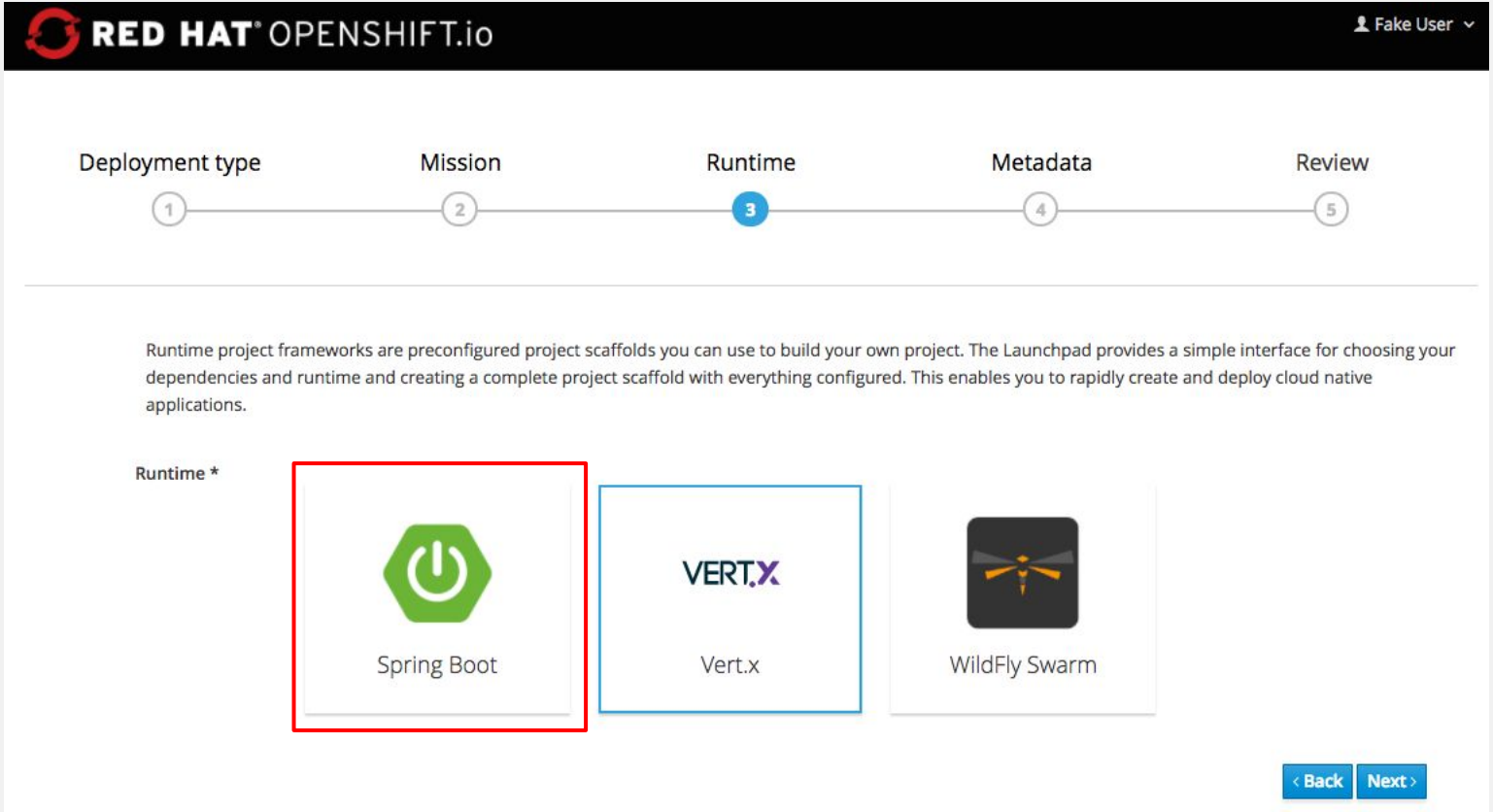
Select Deployment Type



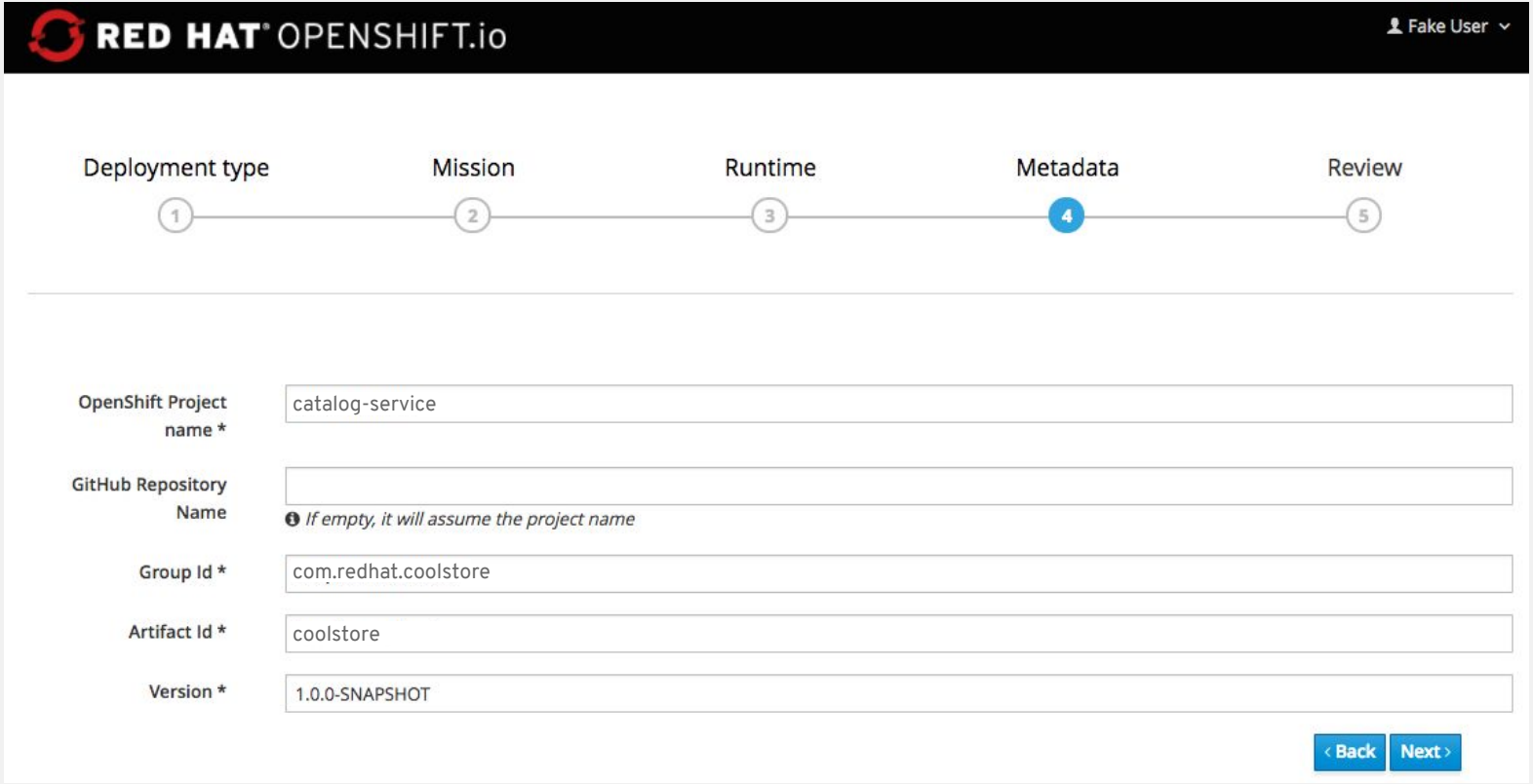
Select Mission (Quickstart)



Select Runtime



Provide Project Information



Launch on OpenShift