redhat.

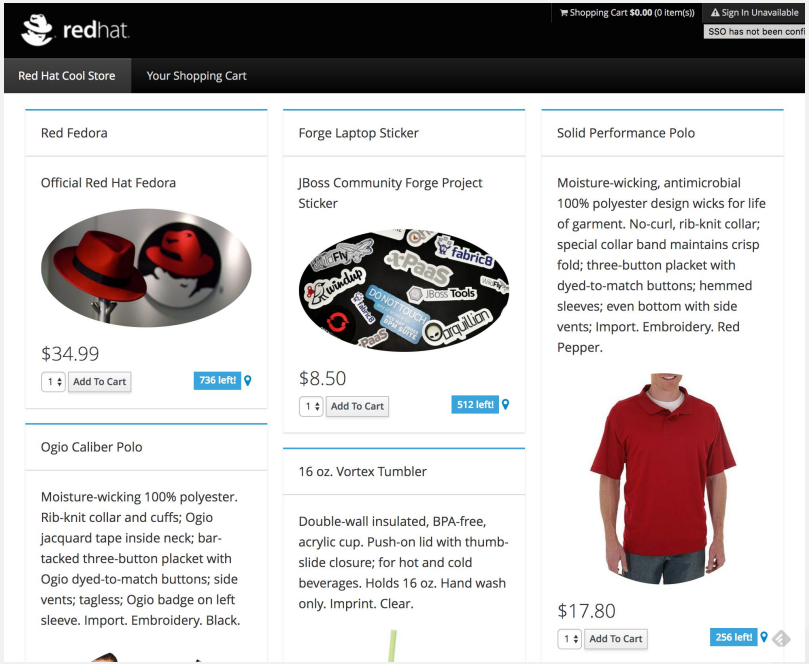
MULTI PRODUCT MICROSERVICES LAB

James Falkner
Sr. Technical Marketing Manager, Middleware
April 2017

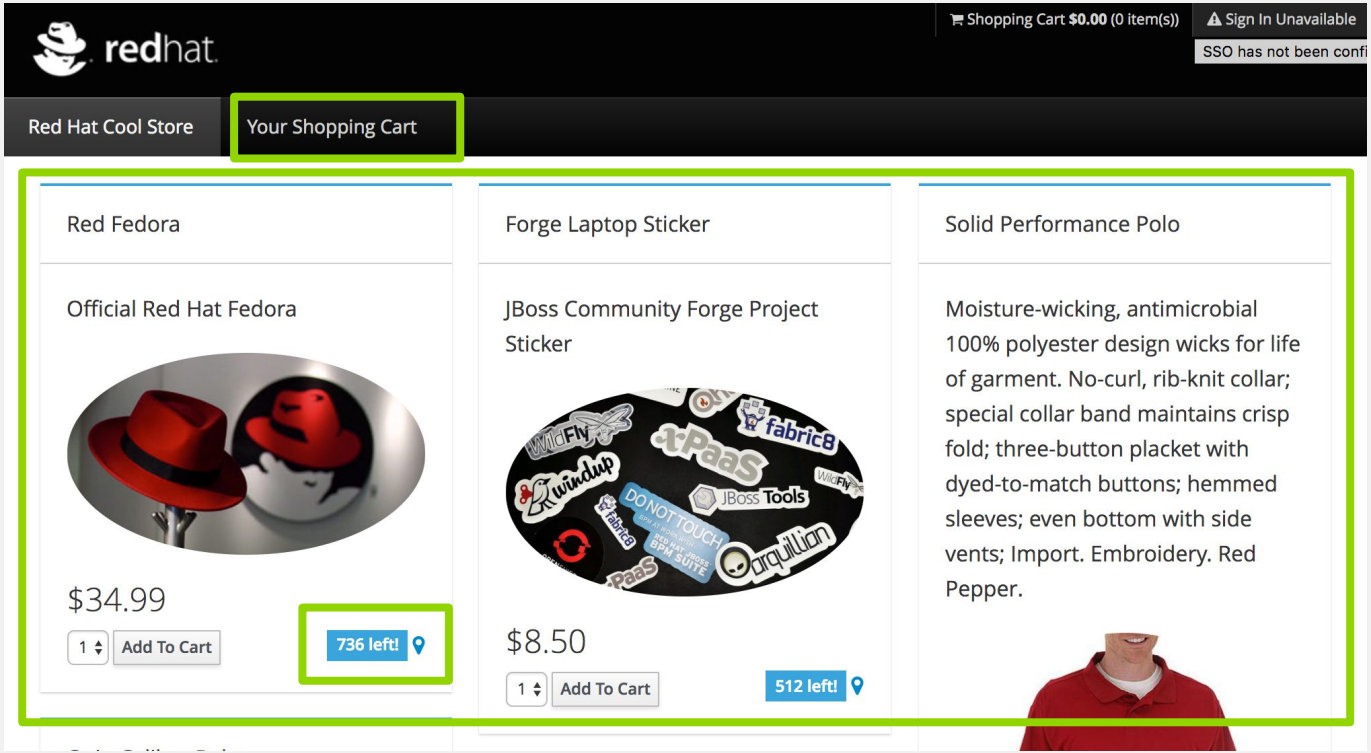

COOLSTORE APPLICATION DEMO

COOLSTORE APPLICATION

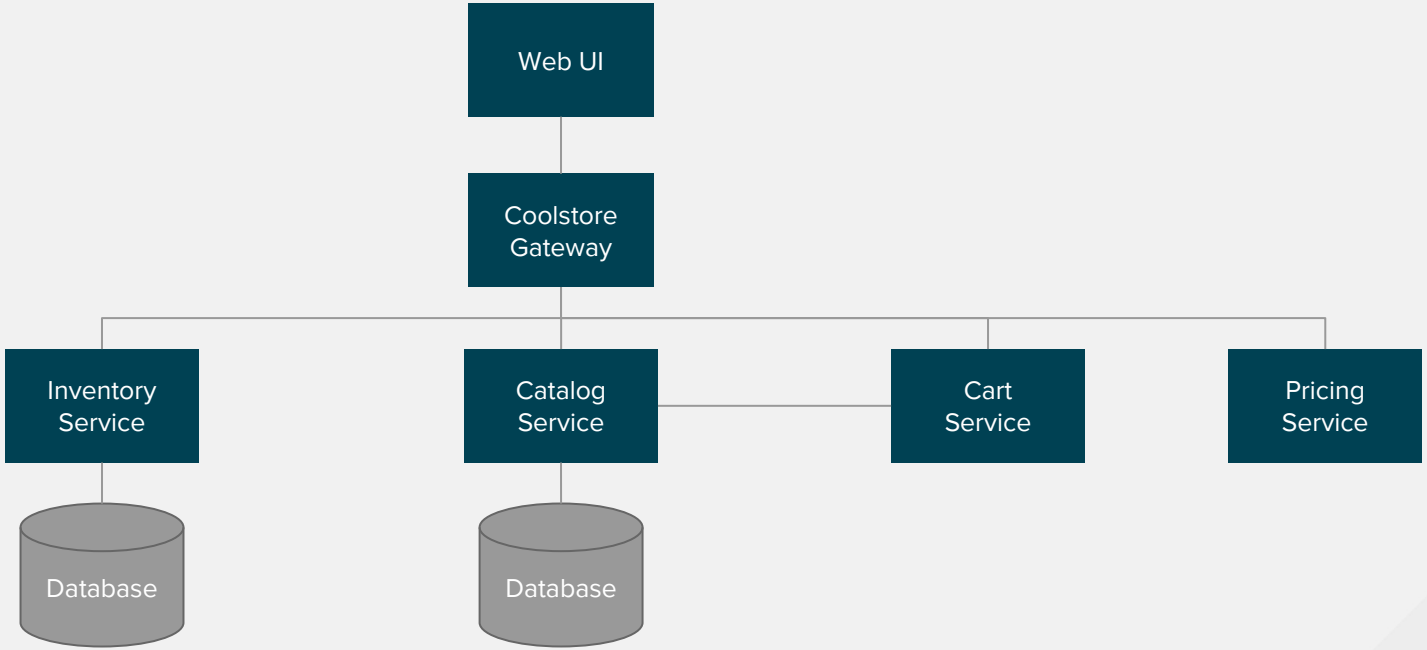
- Online shop for selling products
- Web-based polyglot application using
  - AngularJS
  - Node
  - Java with multiple frameworks (Spring, Java EE, etc)
- Microservices architecture
- Deployed using containers



COOLSTORE SERVICES

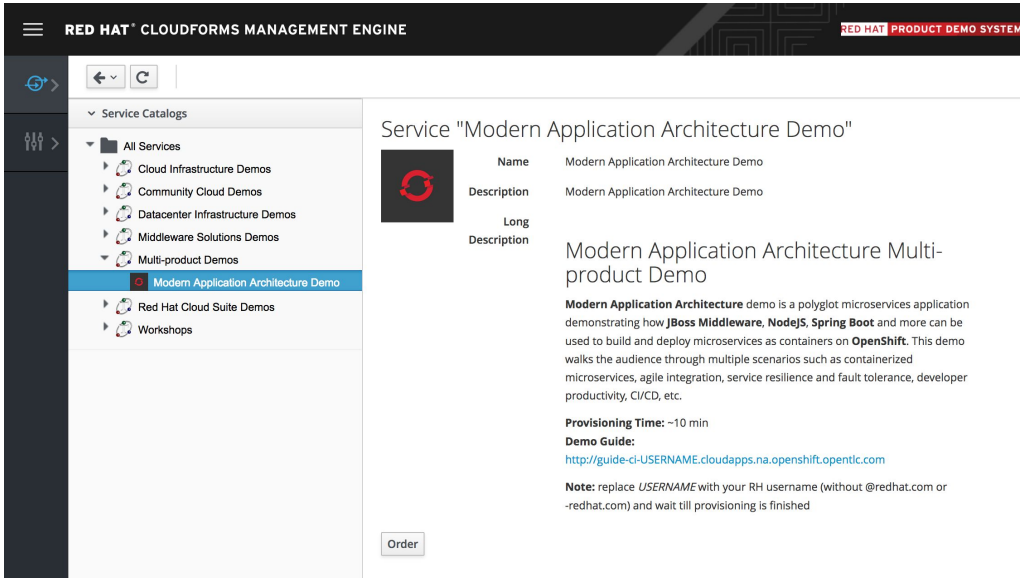


COOLSTORE SERVICES



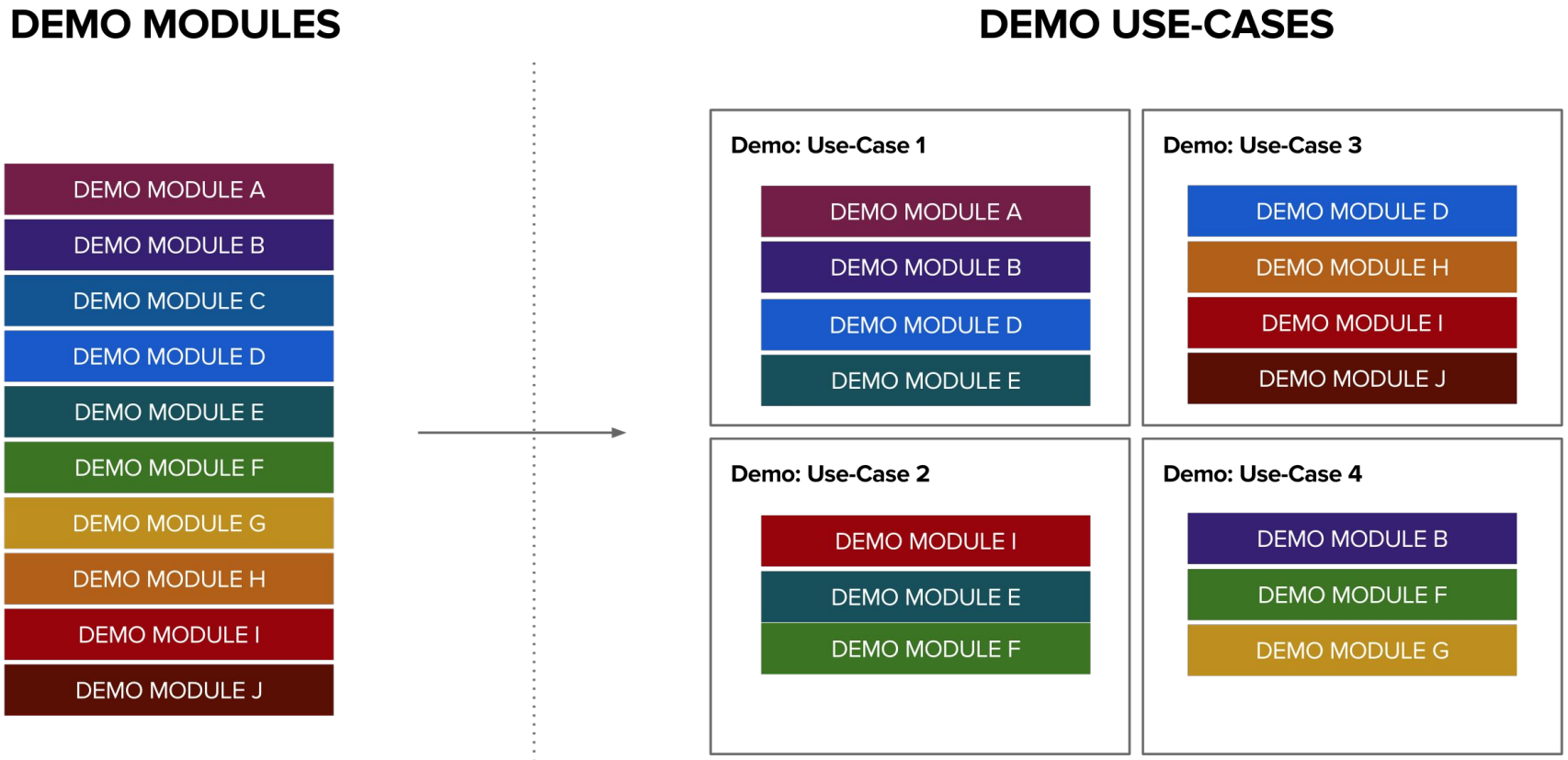
DECK GUIDE: ORDERING THE DEMO

1. Log in into RHPDS (demo system): https://rhpds.redhat.com
2. Order the demo



3. Demo will be ready ~15 min later

MODULES VS. USE CASES



GUIDES AND VIDEOS

These demos are targeted at Red Hat Partners and SAs, SSAs and SSPs and provide a set of deliverables to facilitate the demo experience of reach target group. The following is provided for each demo module in order to train the presenter on how to deliver the demo:

- **Guide**: describes how to demo, what to do and what to explain during the demo
- **Slide**: supporting slides to be used during the demo for explaining concepts
- **Video**: recording of the live demo using the guides and slides
- **Simulation**: an interactive recorded demo which can be used to demo the scenario or train for a live-demo
- **Scripts**: in order to setup the demo infrastructure on any OpenShift environment with enough resources and quota
- **Environment**: automatic provisioning of the demo infrastructure on Red Hat Product Demo System at https://rhpds.redhat.com

GUIDES AND VIDEOS

All Modules

1. CoolStore Demo Application Overview
2. Containerized Microservices
3. Agile Integration for Microservices
4. Service Resilience and Fault Tolerance
5. Creating On-Demand Environment
6. Increase Developer Productivity with Containers (with JBoss Developer Studio)
7. Building Quality into Development Process (with JBoss Developer Studio)
8. Building Quality into Development Process (without JBoss Developer Studio)
9. Increasing Delivery Speed with CI/CD
10. Continuous Security Compliance for Containers

Workshopper

Source code @ Github

All Modules

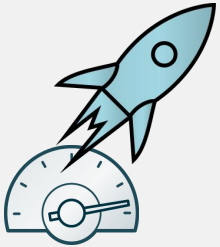
Multi-product Demo: Quick Code-to-Production without Downtime

Multi-product Demo: Agile Integration

Multi-product Demo: Modern App Architecture

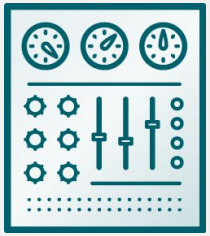

MICROSERVICES ARCHITECTURE
OVERVIEW
redhat.

VALUES OF MICROSERVICES



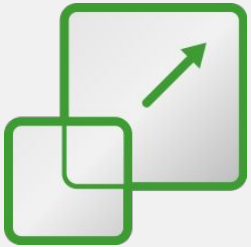
FAST TIME TO MARKET

Small autonomous services can be developed and delivered faster



EFFICIENCY

Automating delivery and monitoring of small services is easier



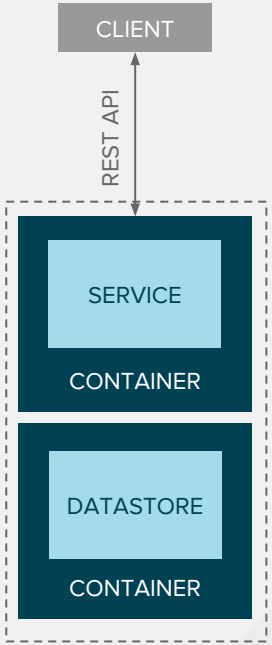
SCALABILITY

Fine grained scalability is easier and uses less resources

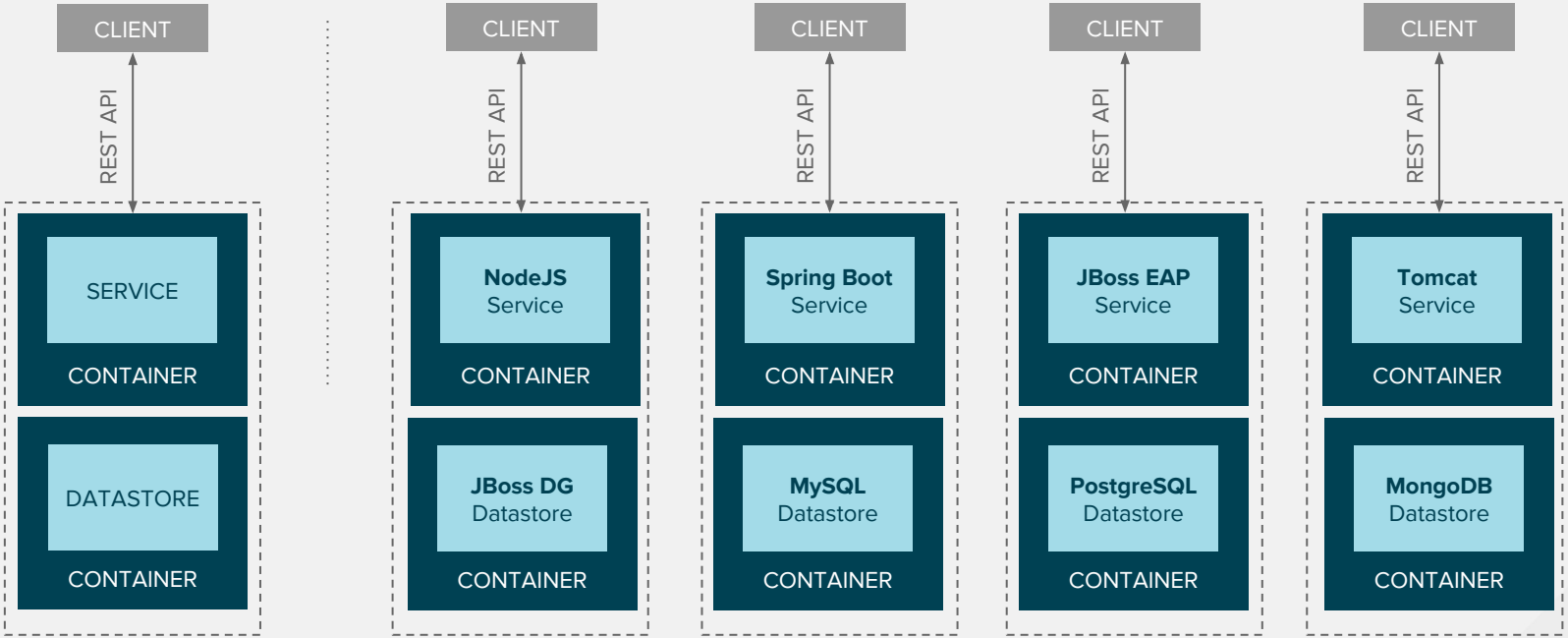
MICROSERVICES ARCHITECTURE

Principles of microservices architecture:

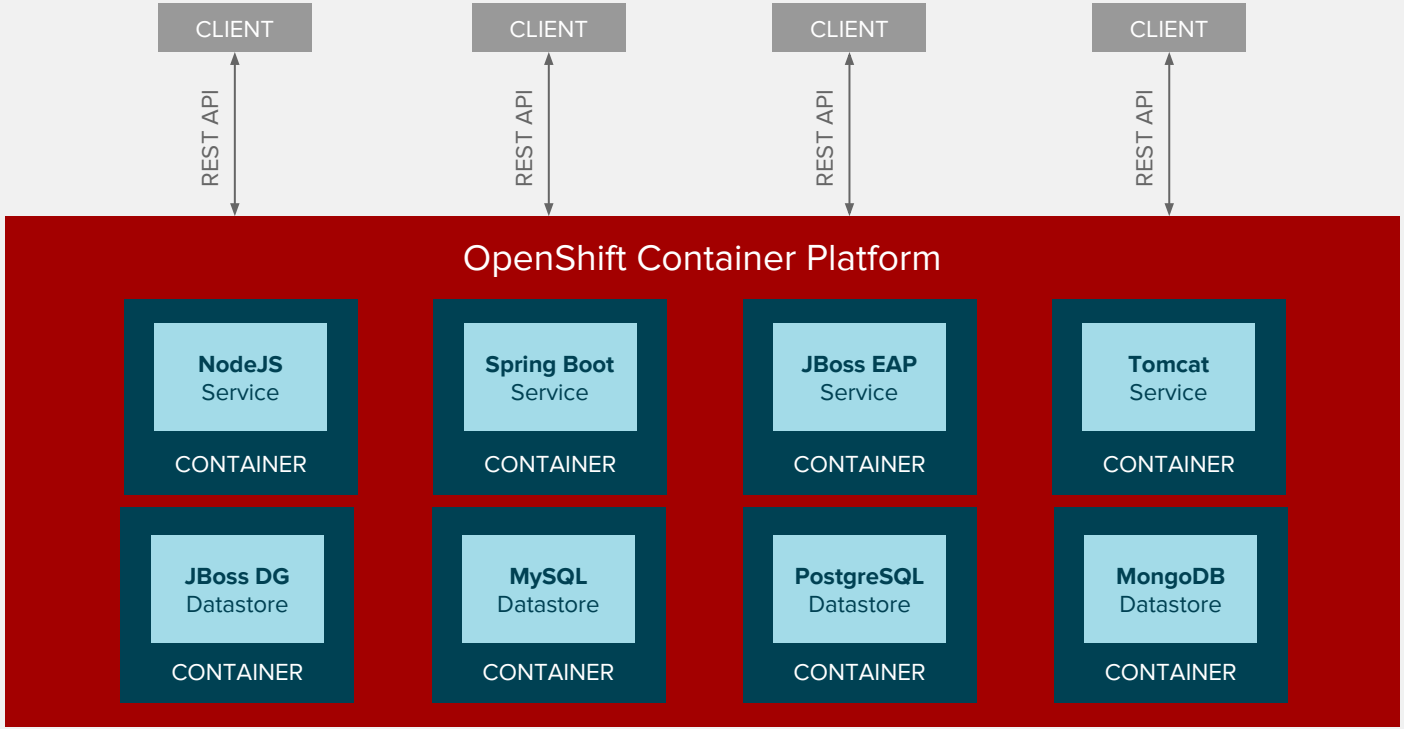
- Model around Business Domain
- Culture of Automation (DevOps)
- Deploy Independently
- Scale Independently
- Release Independently
- Combined to form a system or application
- Antifragile - increased robustness and resilience under pressure
- Polyglot (language and framework independence)
- API / contract focused
- Decentralized data management



MICROSERVICES ARCHITECTURE



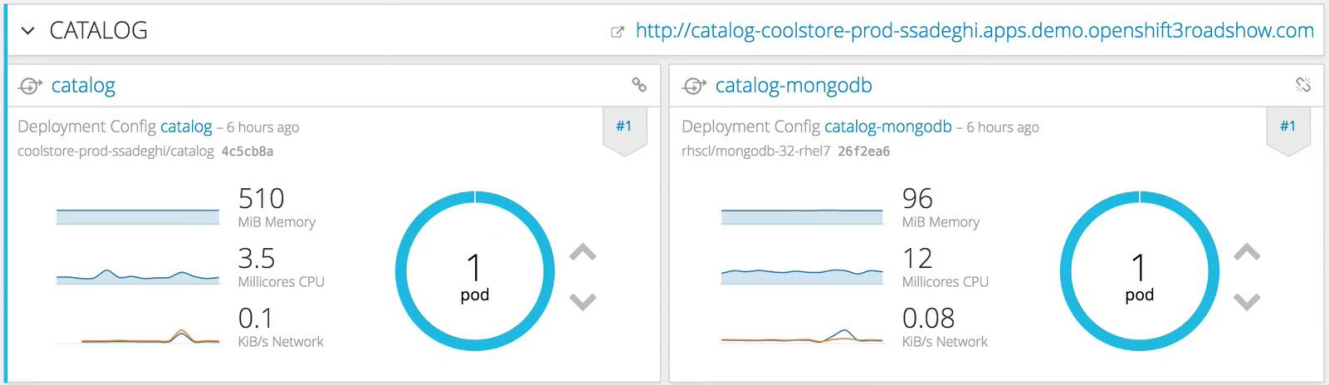
MICROSERVICES ARCHITECTURE



CONTAINER DEPLOYMENT & MANAGEMENT

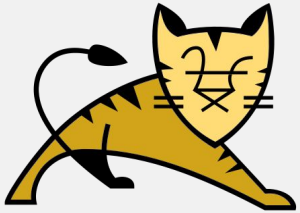
OpenShift Container Platform provides features like:

- Deploy Independently
- Scale Independently
- Release Independently
- CI/CD (DevOps)
- Antifragile
- Polyglot



POLYGLOT

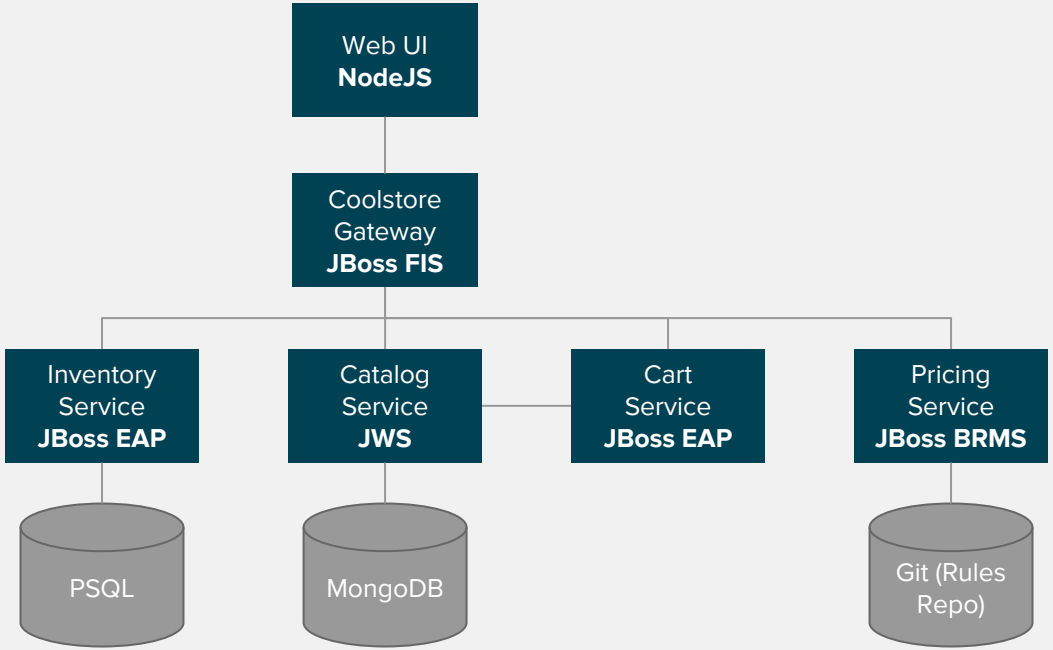
RED HAT® JBOSS®
ENTERPRISE
APPLICATION PLATFORM



RED HAT® JBOSS®
BRMS

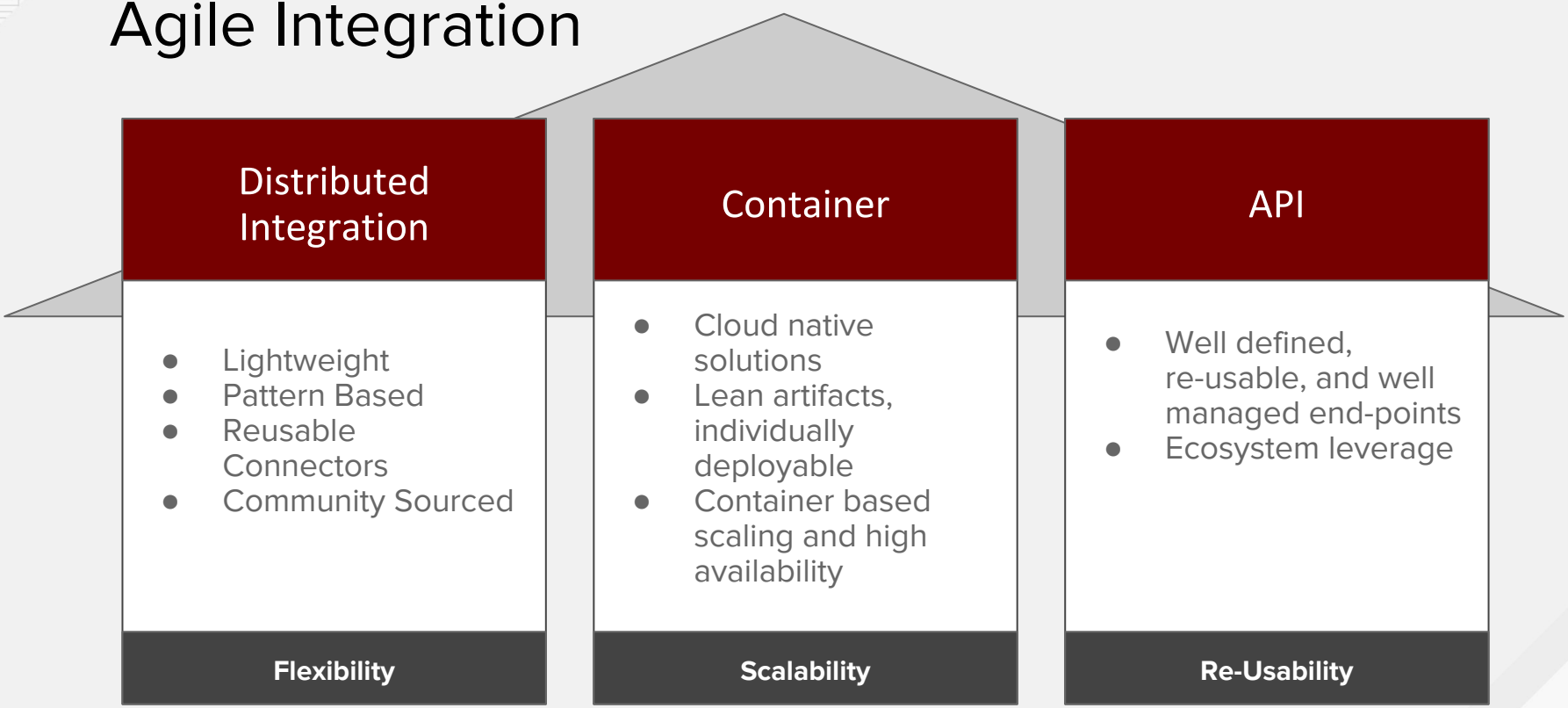
RED HAT® JBOSS®
FUSE

DEMO APPLICATION

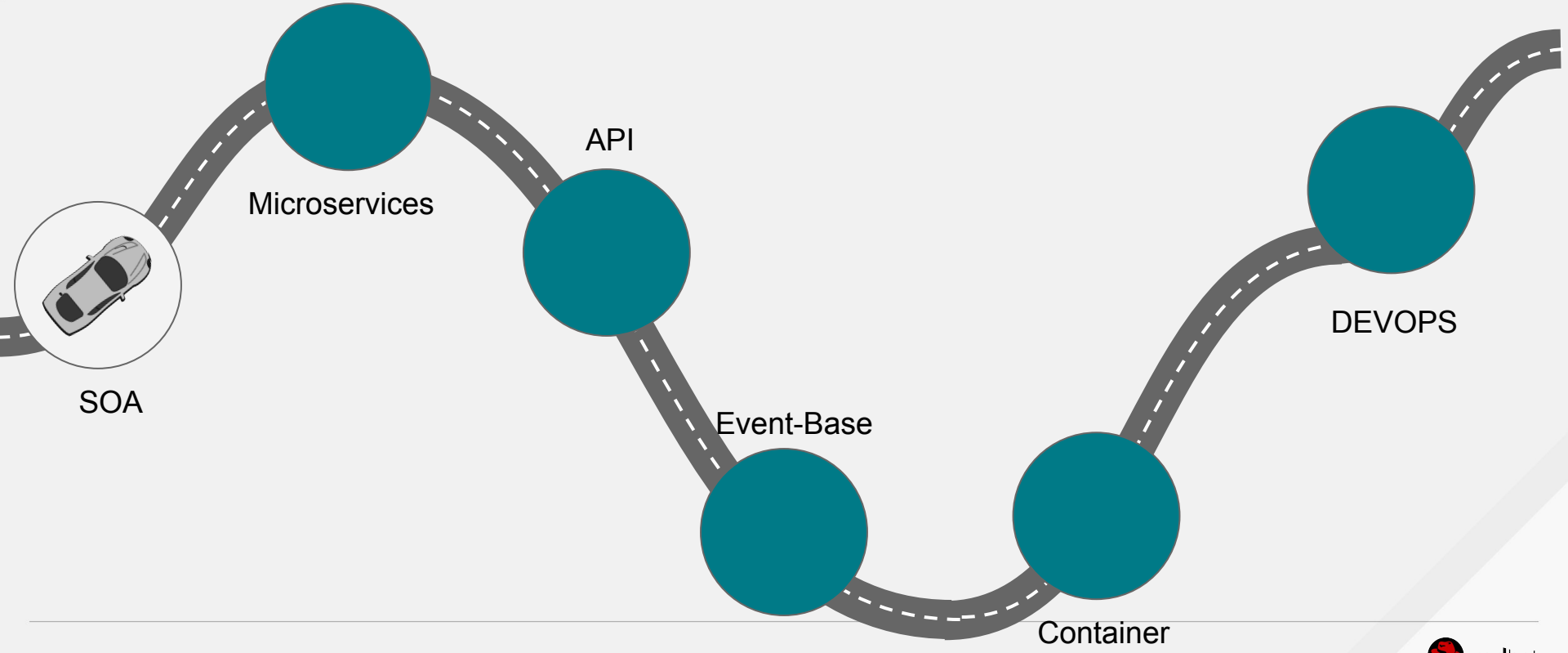



AGILE INTEGRATION
FOR MICROSERVICES
redhat.

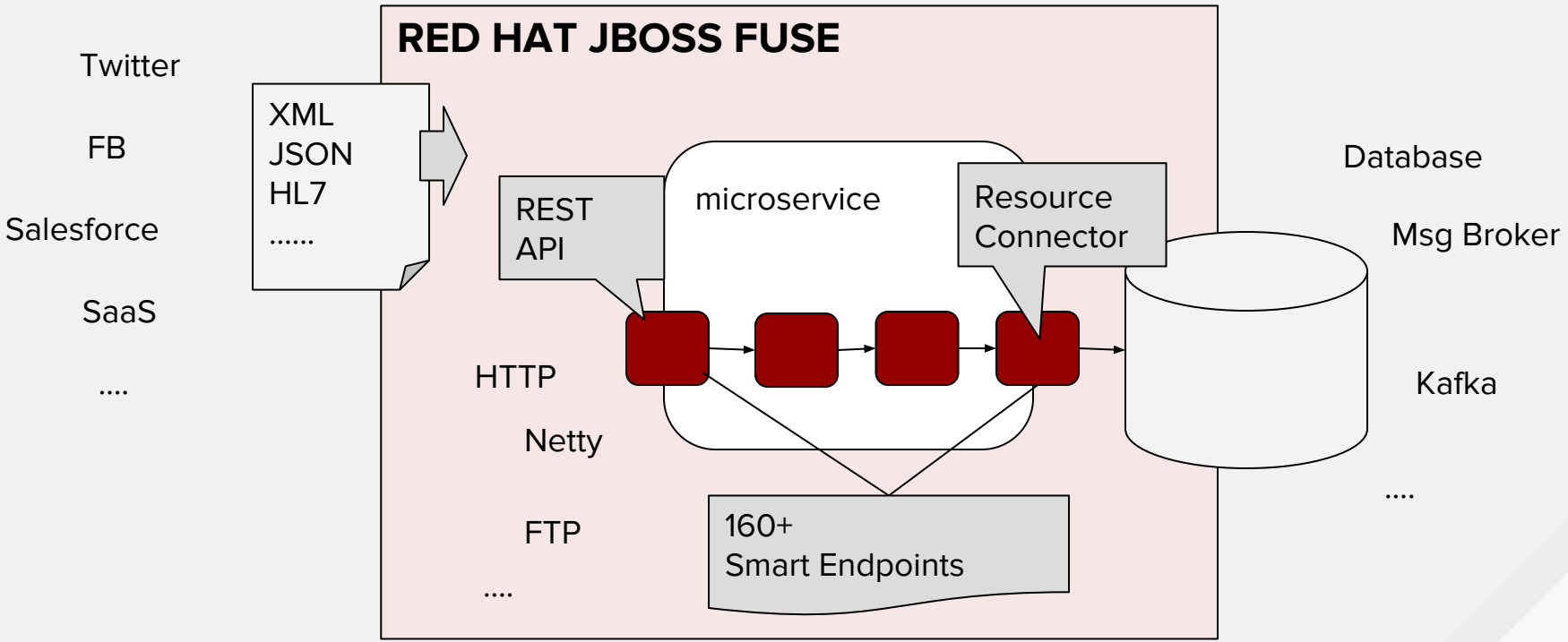
Agile Integration



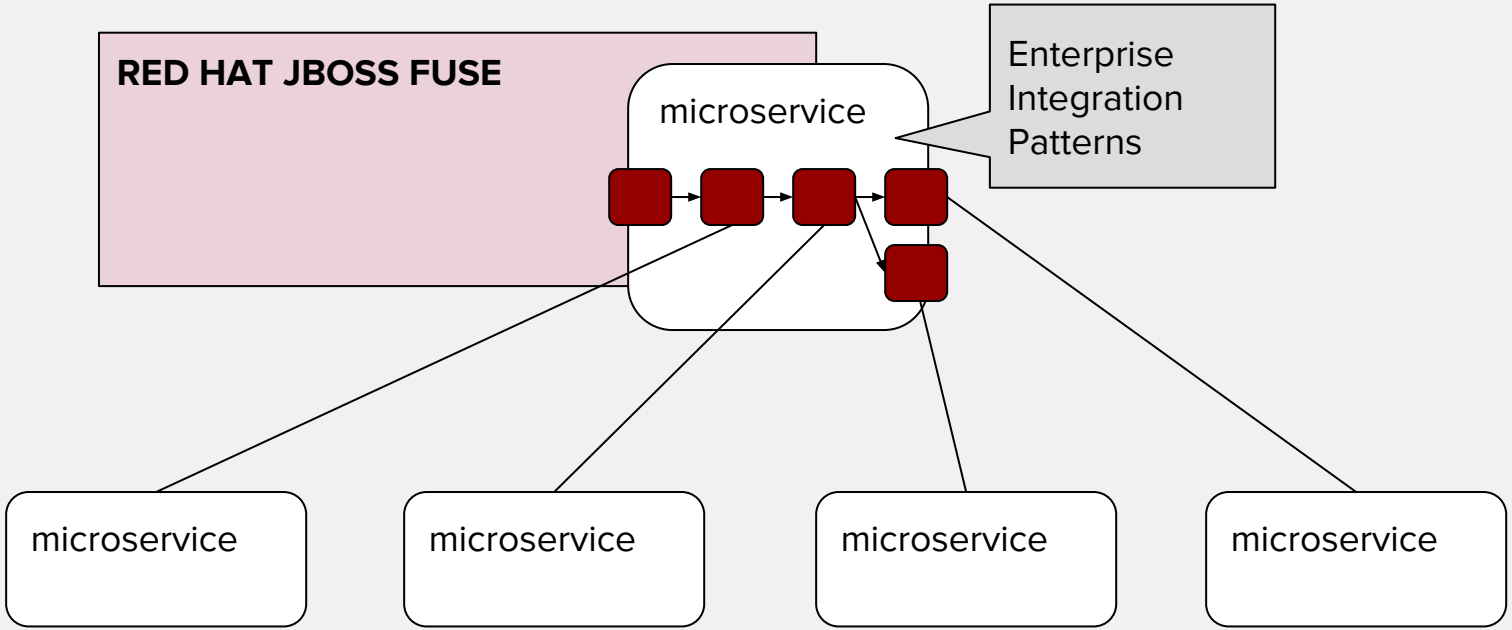
AGILE INTEGRATION



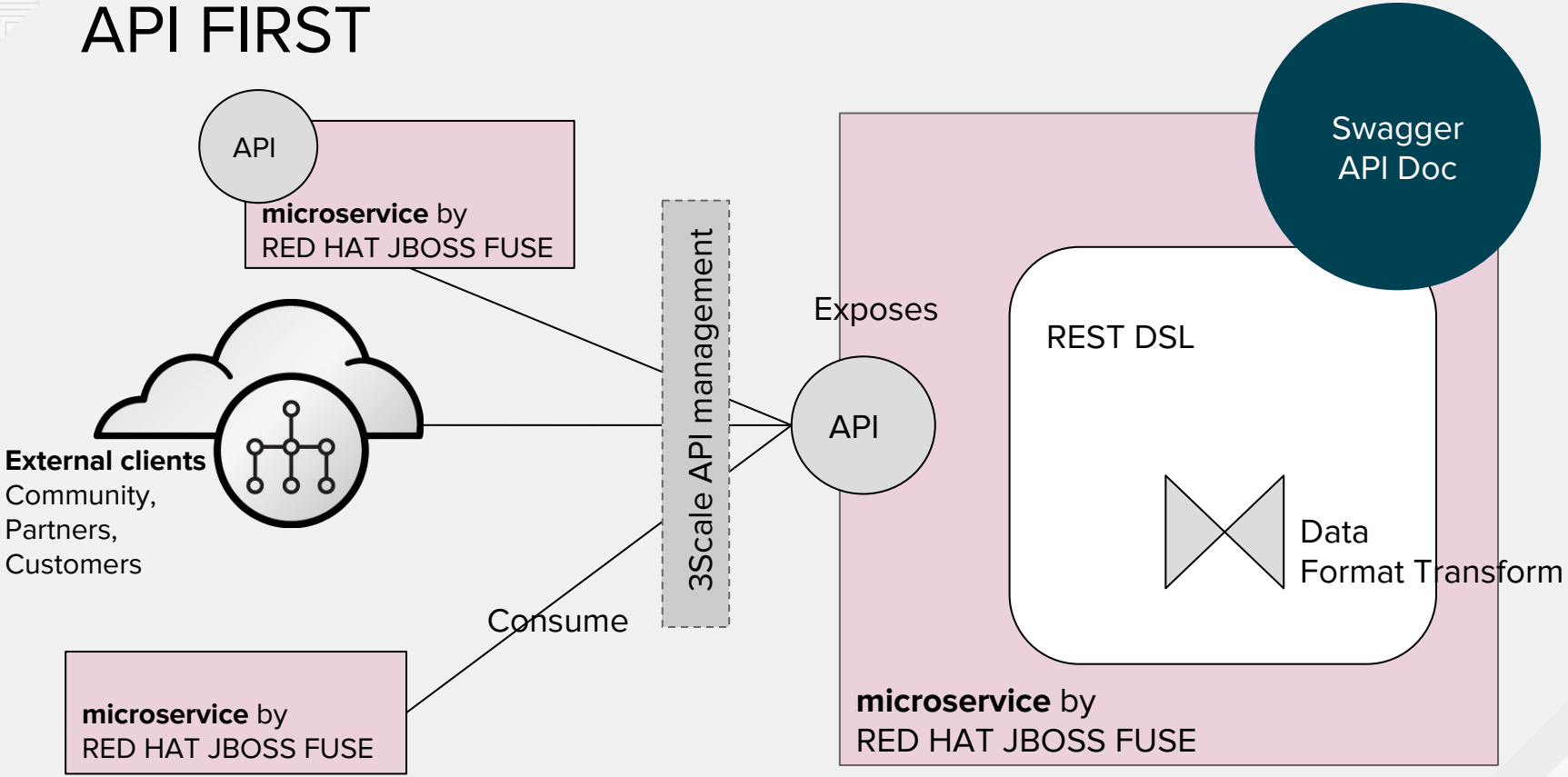
BUILDING MICROSERVICES



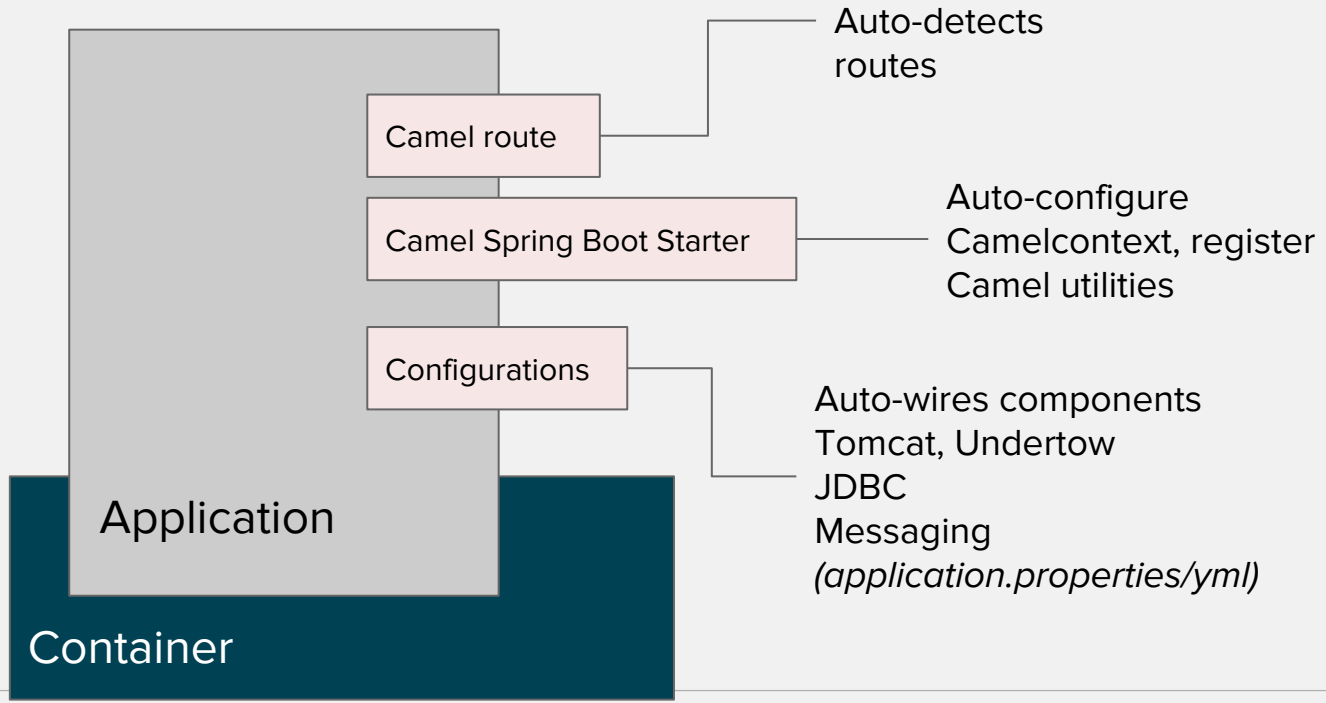
COMPOSING MICROSERVICES



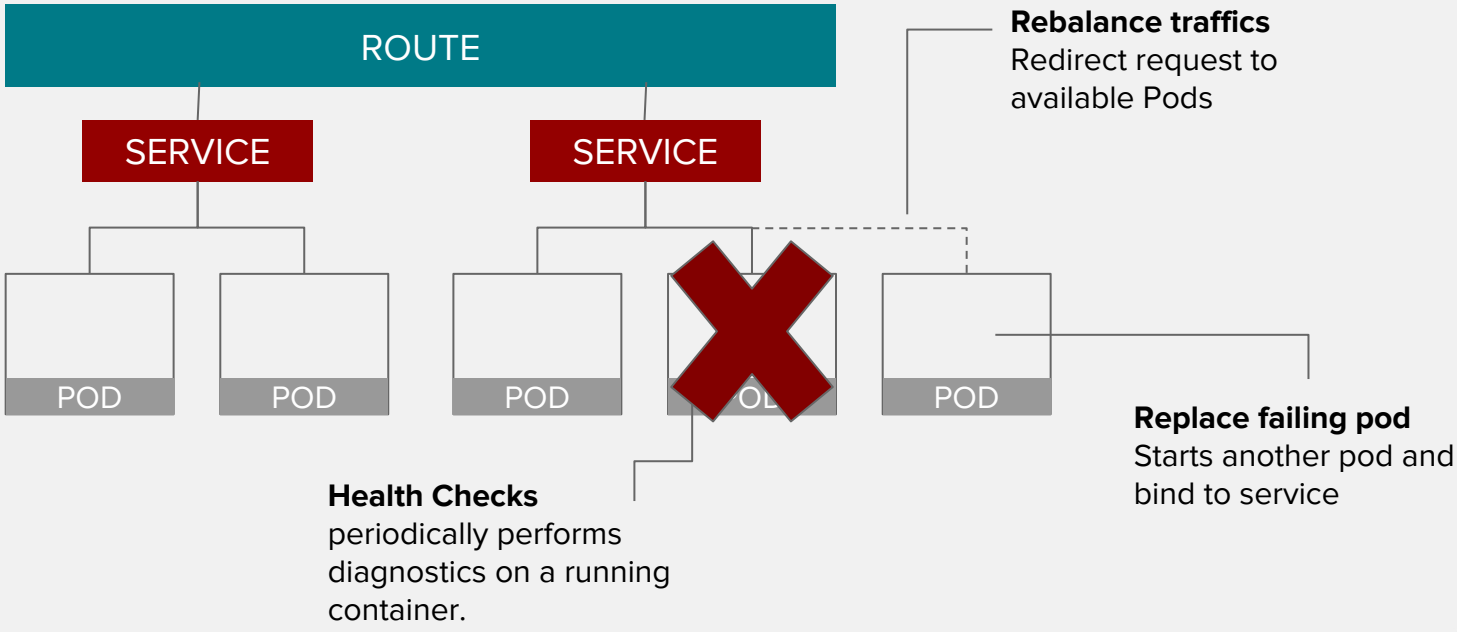
API FIRST


API
microservice by
RED HAT JBOSS FUSE
Swagger
API Doc
Exposes
REST DSL
3Scale API management
API
External clients
Community,
Partners,
Customers
Data
Format Transform
Consume
microservice by
RED HAT JBOSS FUSE
microservice by
RED HAT JBOSS FUSE

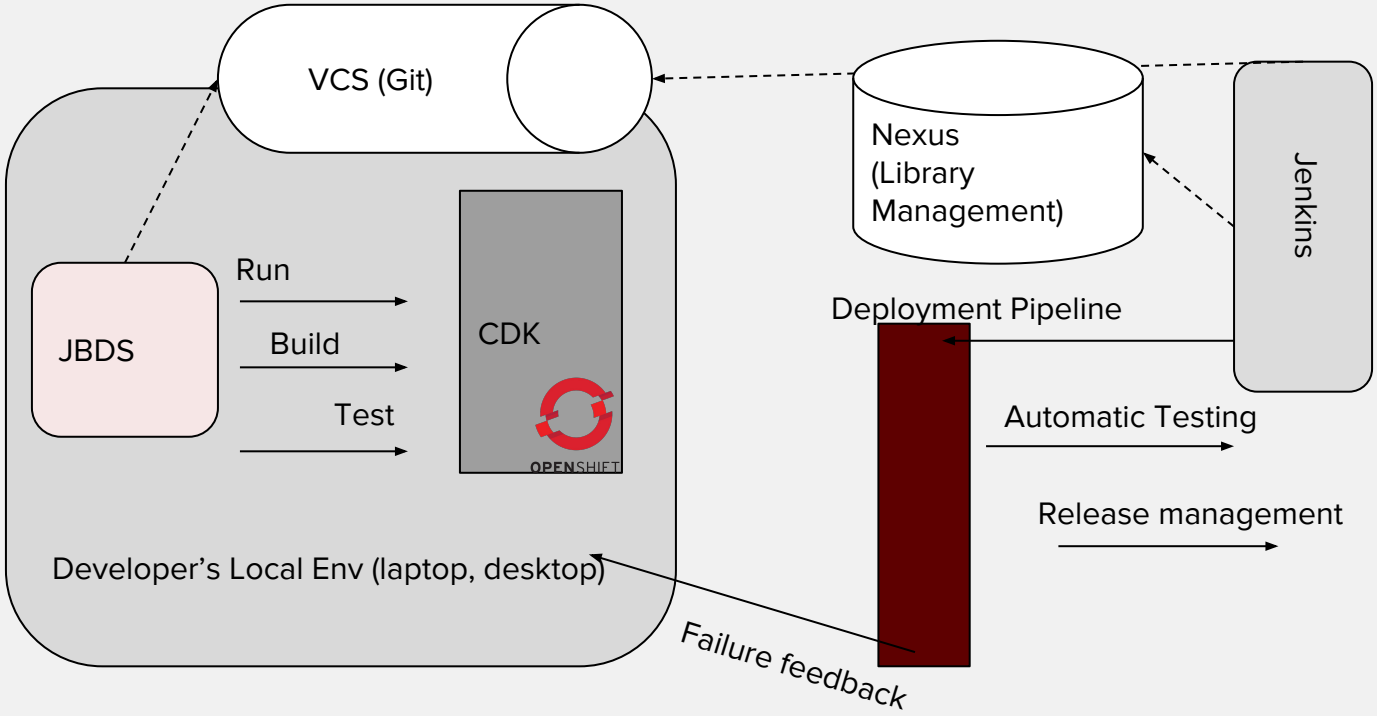
Fuse and Spring Boot



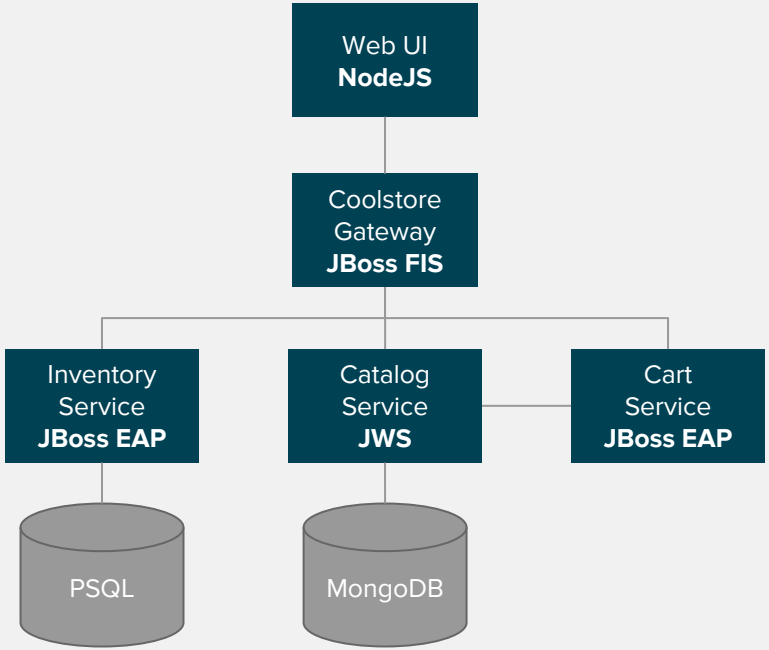
FAILURE RECOVERY



CONTAINER

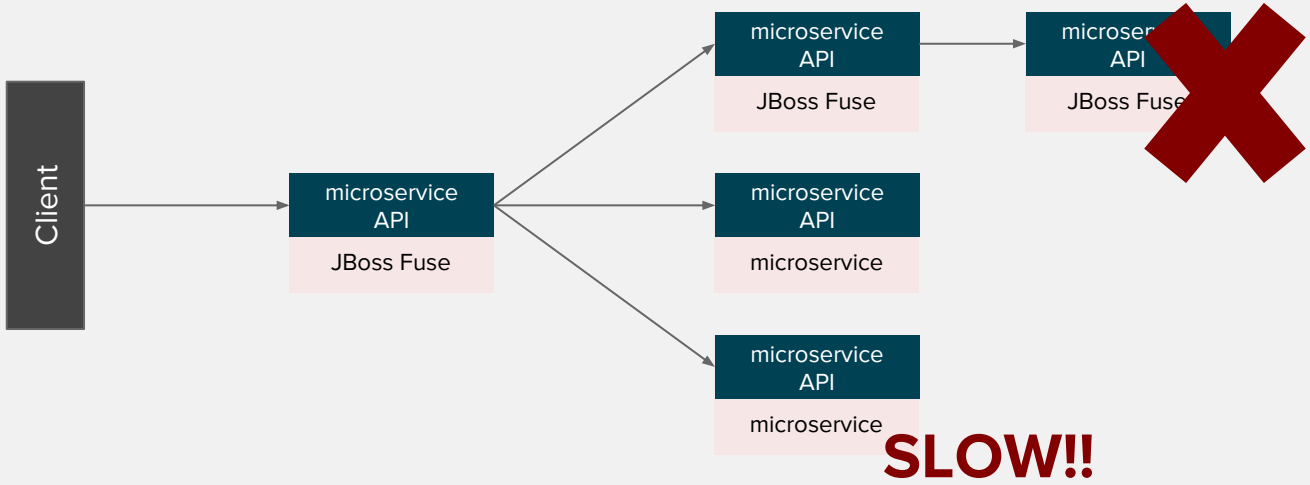

VCS (Git)
Nexus
(Library
Management)
Jenkins
Run
Build
Test
JBDS
CDK
OPENSHIFT
Deployment Pipeline
Automatic Testing
Release management
Developer's Local Env (laptop, desktop)
Failure feedback

DEMO APPLICATION

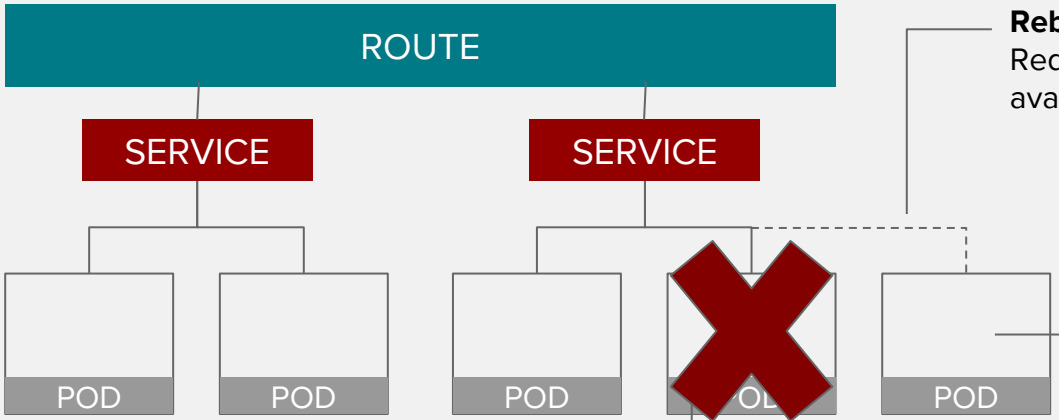



SERVICE RESILIENCE
AND FAULT TOLERANCE

SERVICE RESILIENCE



FAULT TOLERANCE

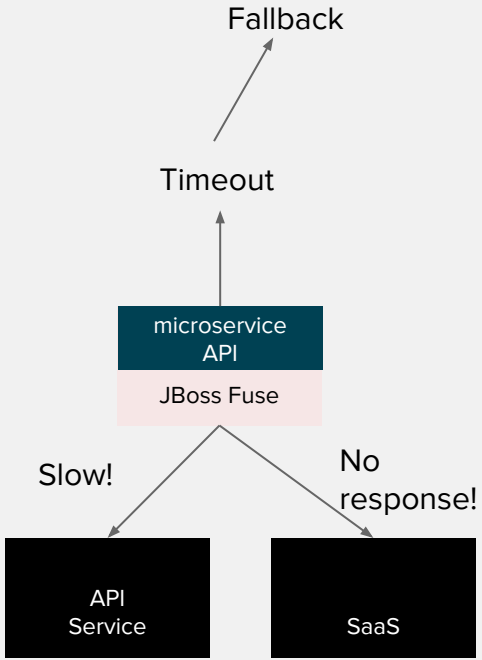


Rebalance traffics
Redirect request to available Pods

Replace failing pod
Starts another pod and bind to service

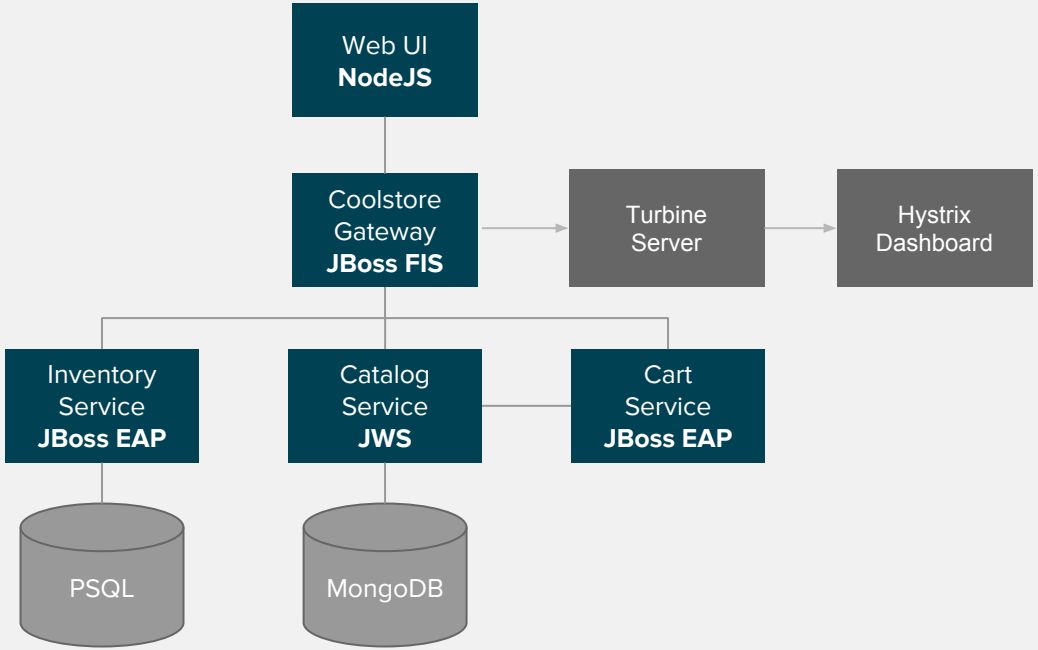
Health Checks
periodically performs diagnostics on a running container.

Hystrix EIP




```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <hystrix>
      <to uri="http://fooservice.com/slow"/>
      <onFallback>
        <transform>
          <constant>Fallback message</constant>
        </transform>
      </onFallback>
    </hystrix>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

COOLSTORE APPLICATION




CREATING ON-DEMAND
ENVIRONMENTS

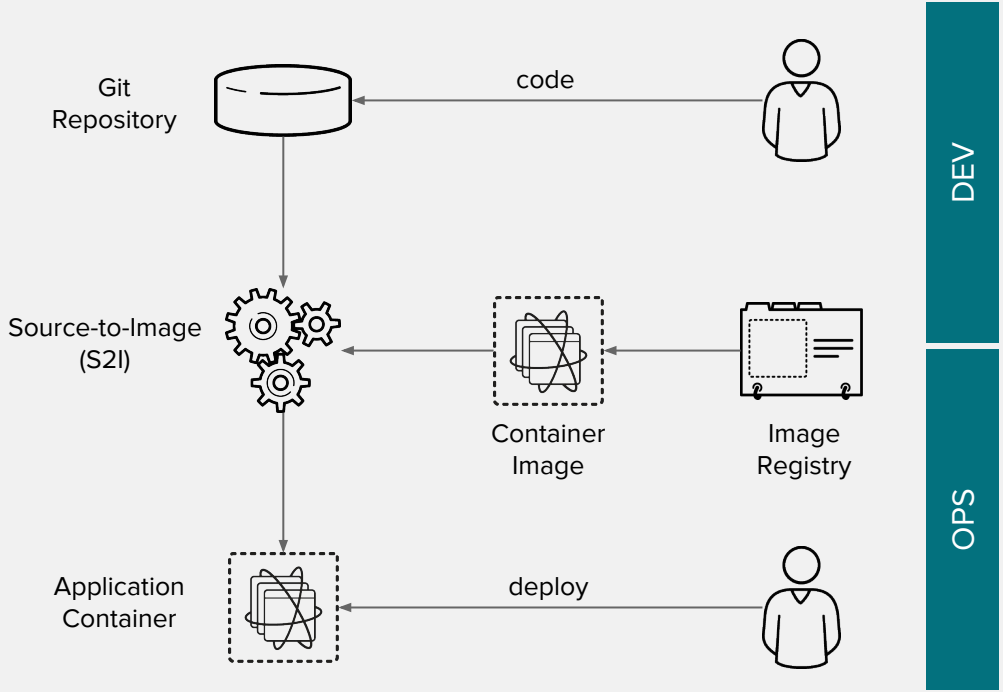
FORKING A GIT REPOSITORY

BUILDING CONTAINER IMAGES WITH SOURCE-TO-IMAGE

CODE

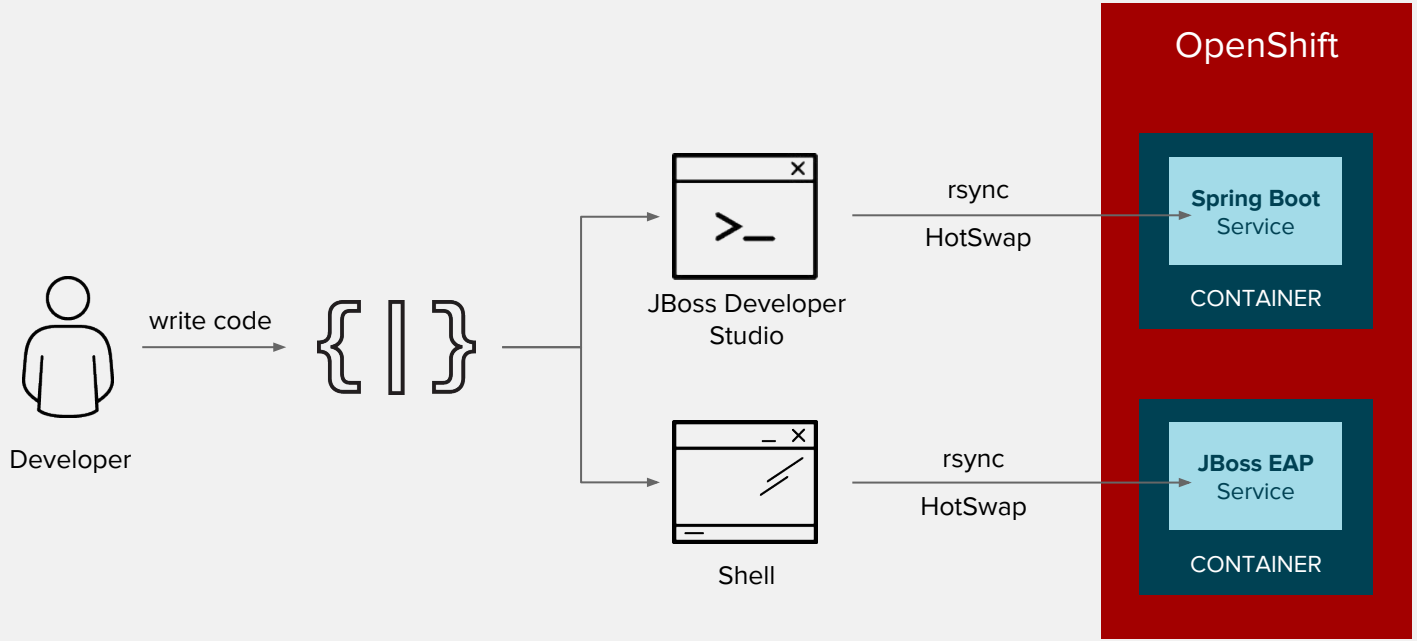
BUILD

DEPLOY



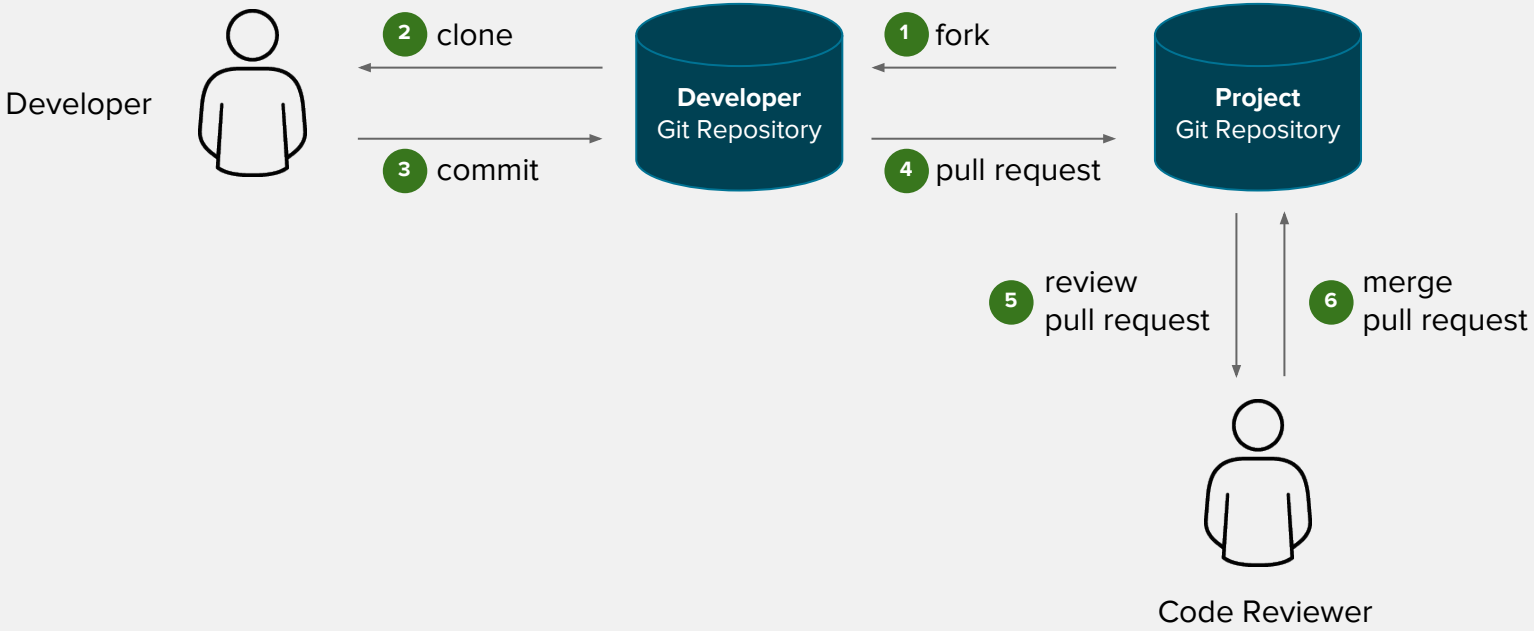

DEVELOPER PRODUCTIVITY
WITH CONTAINERS
redhat.

DEPLOY CODE WITHOUT BUILD DELAYS



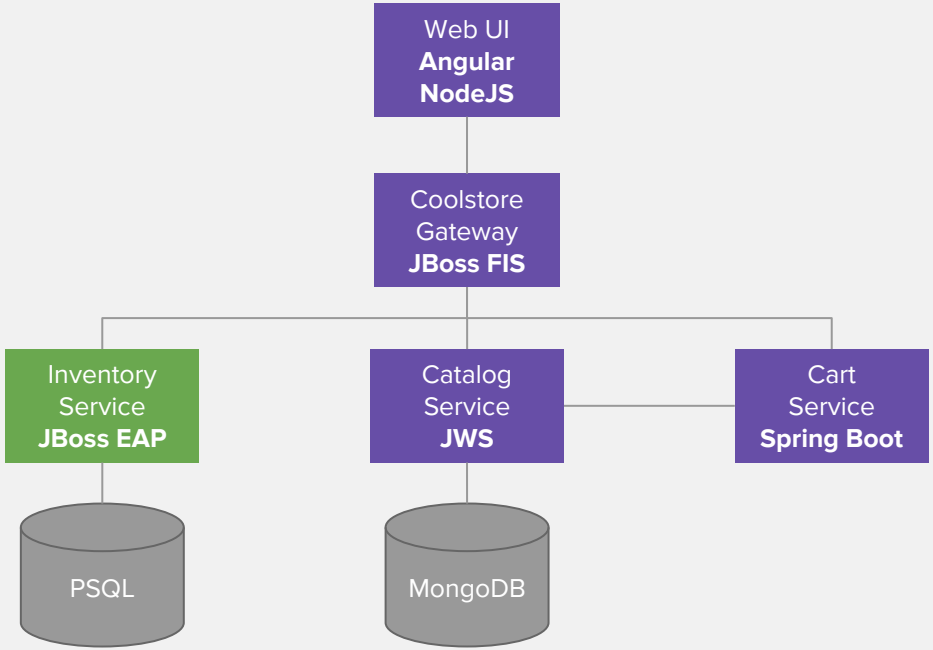

BUILDING QUALITY INTO
THE DEVELOPMENT PROCESS

BUILDING QUALITY INTO THE PROCESS

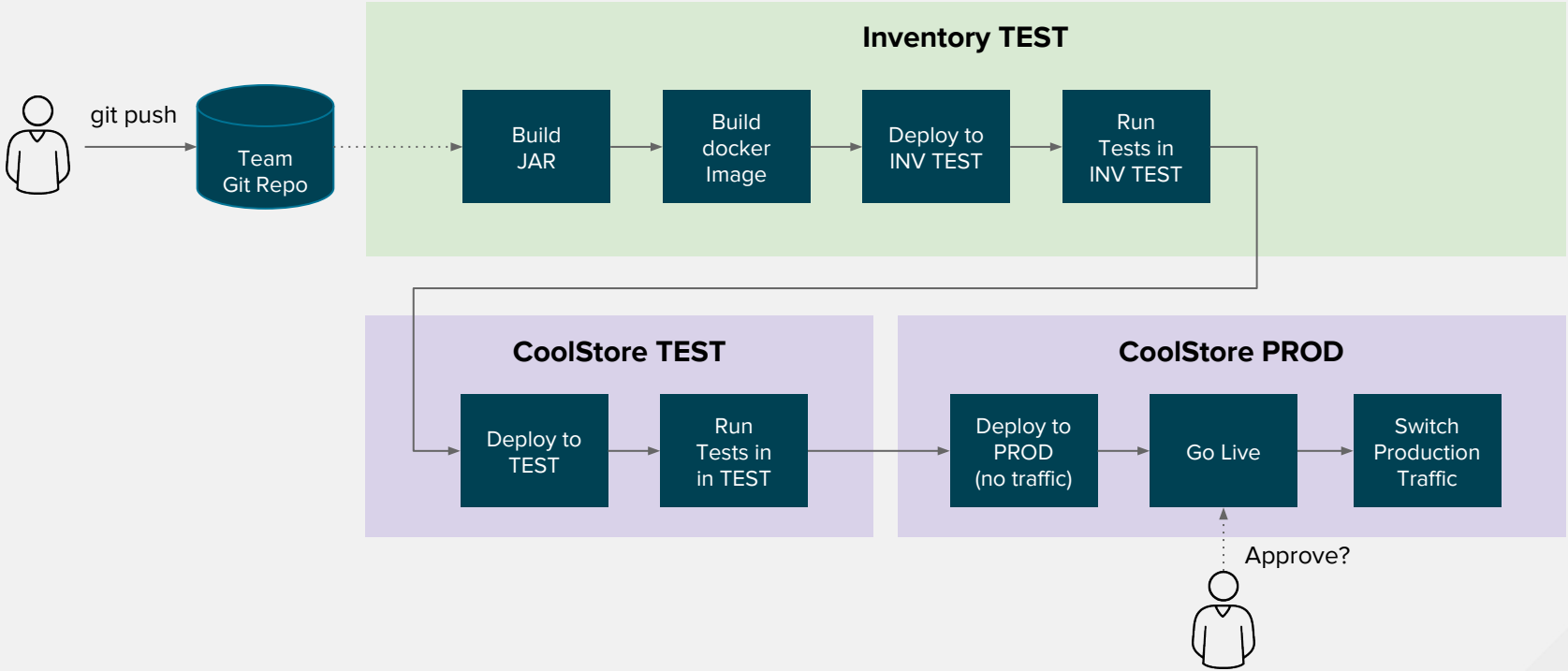



INCREASING DELIVERY SPEED WITH
CONTINUOUS INTEGRATION
AND
CONTINUOUS DELIVERY
redhat.

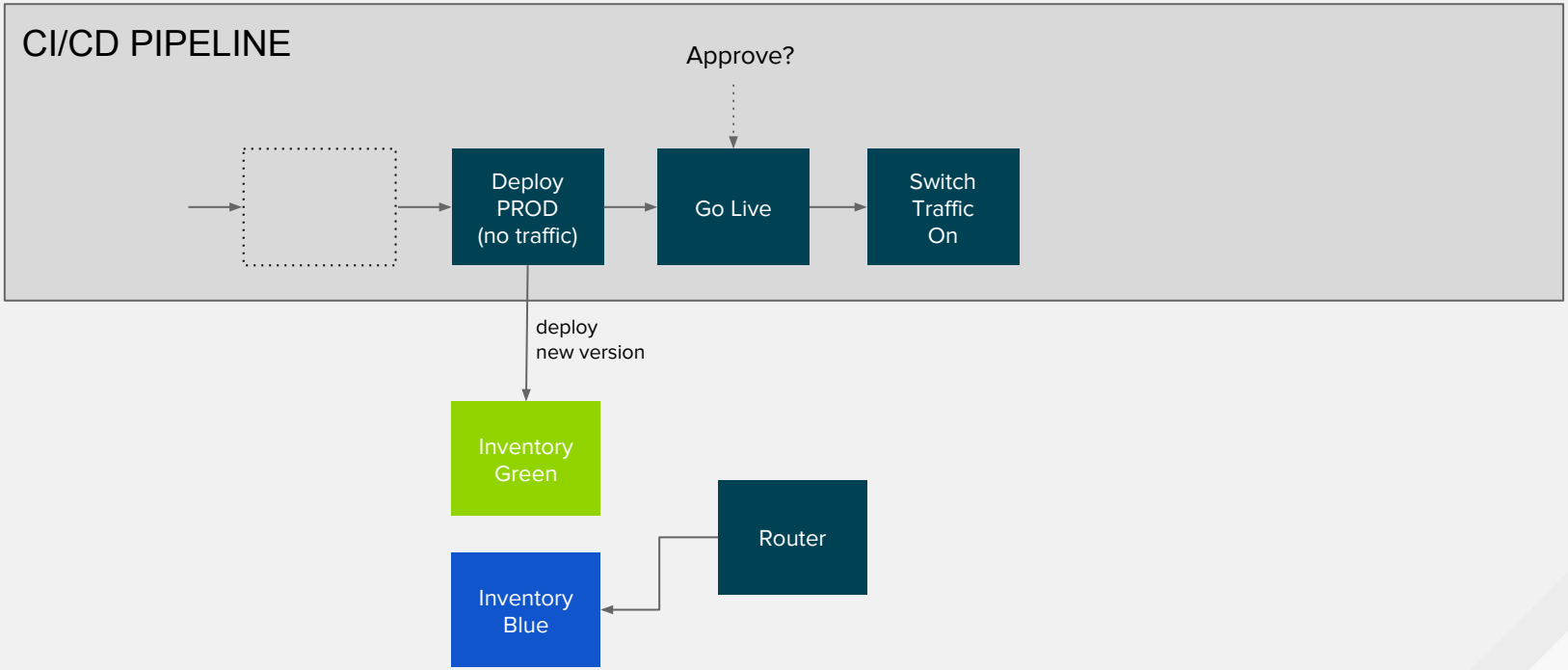
COOLSTORE MICROSERVICES APPLICATION



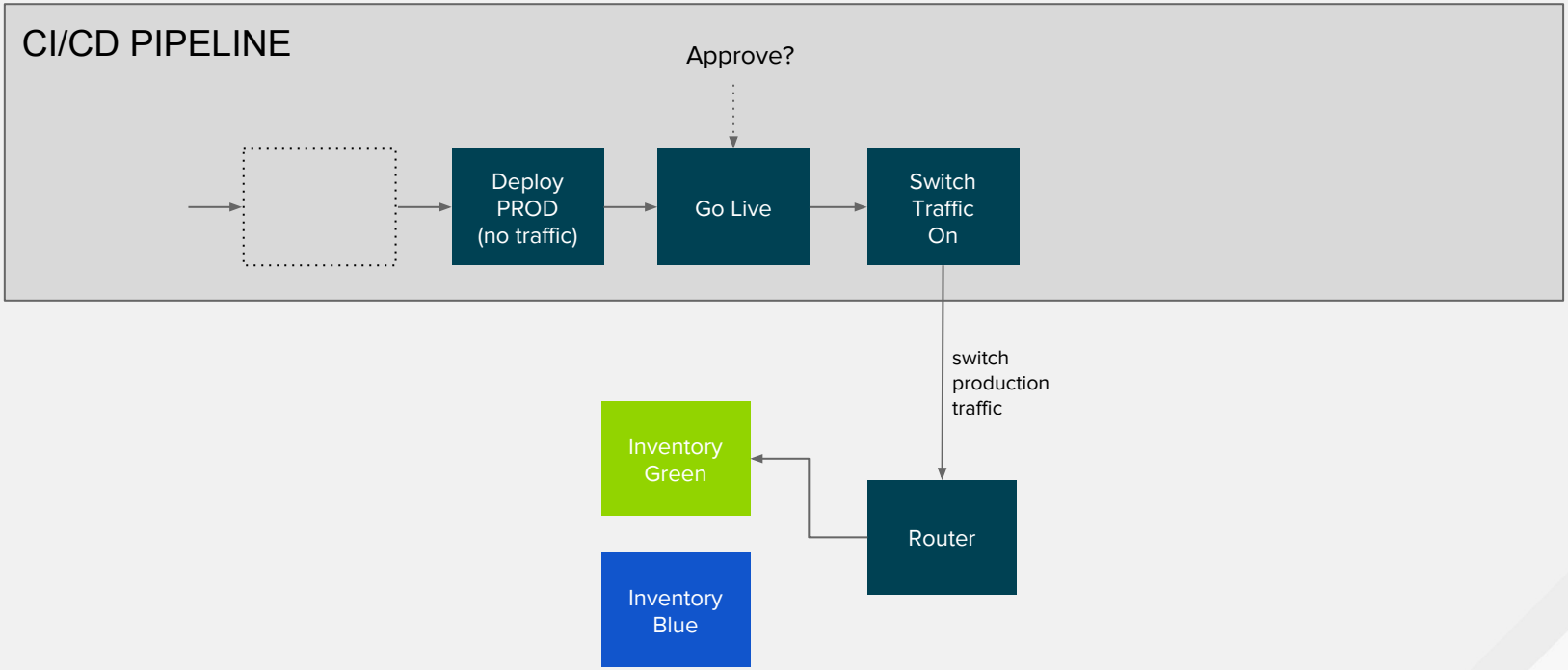
CONTINUOUS DELIVERY PIPELINE



ZERO DOWNTIME WITH BLUE/GREEN DEPLOYMENT



ZERO DOWNTIME WITH BLUE/GREEN DEPLOYMENT




LAB 1
BUILDING AND DEPLOYING A
FAST-MOVING MONOLITH
redhat.

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

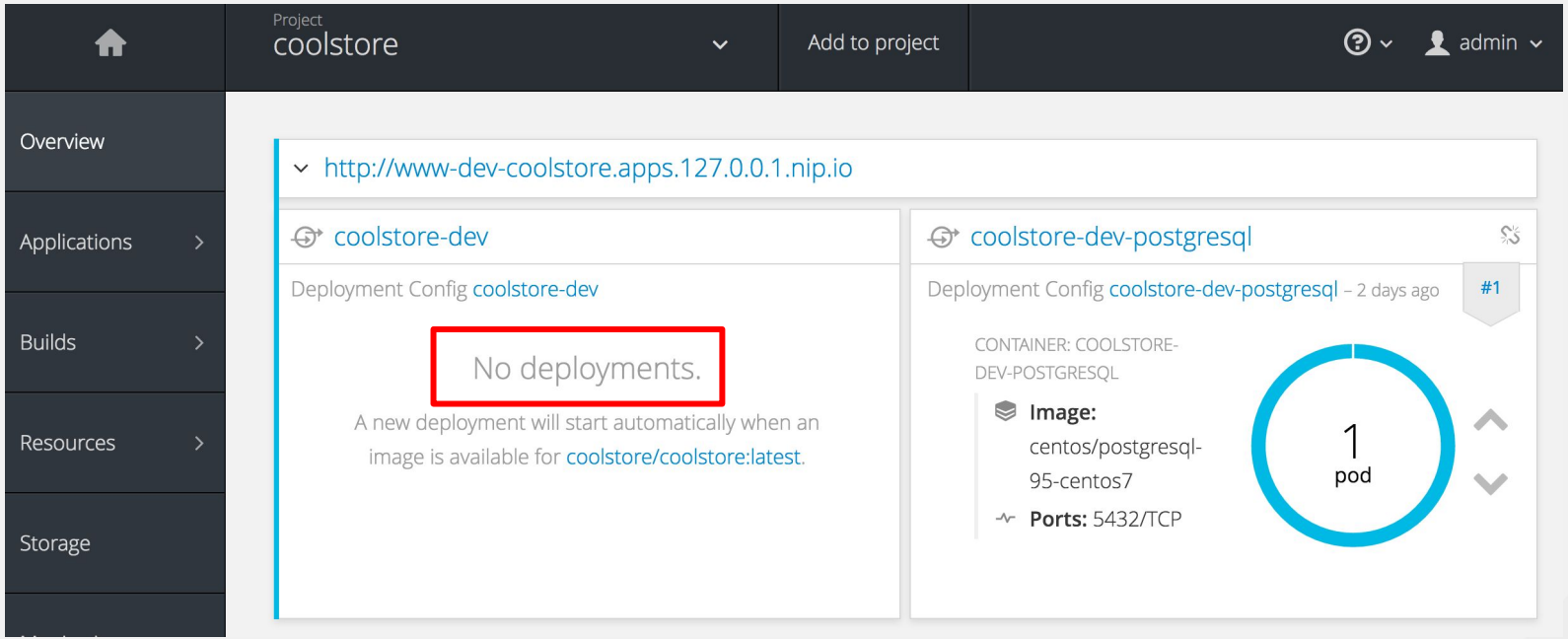
LAB 1 - FAST-MOVING MONOLITH

- 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
- First, deploy the coolstore monolith dev project (don't forget to include `-b app-partner` when you run *git clone* - this is the branch in use for this lab!):

```
$ mkdir ~/coolstore
$ cd ~/coolstore
$ git clone -b app-partner https://github.com/modernize-legacy-apps/monolith
$ cd monolith
$ oc new-project coolstore-<USERNAME>
$ oc process -f src/main/openshift/template.json | oc create -f -
```

LAB 1 - FAST-MOVING MONOLITH

- Notice there is **no deployment yet** as there is no build yet:



LAB 1 - FAST-MOVING MONOLITH

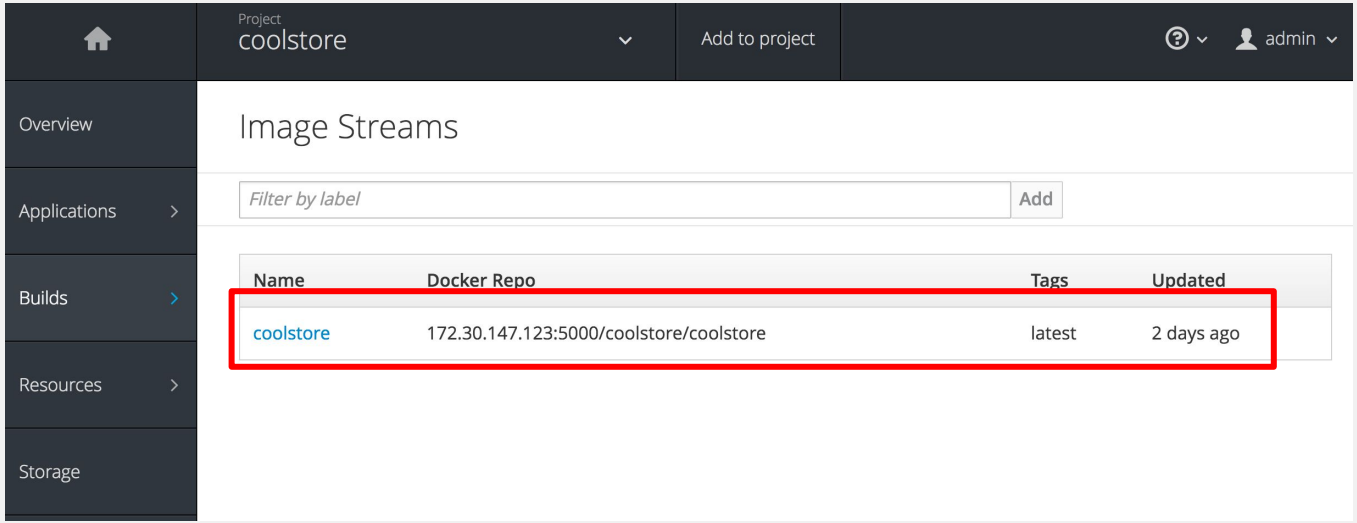
- A number of OpenShift objects were created:
  - A Postgres database deployment, service, and route
  - A coolstore *binary* BuildConfig
    - Binary builds accept source code through stdin in later commands (vs. kicking off a build within an OpenShift pod)
  - Services, Routes and DeploymentConfig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

- Build the coolstore monolith (.war file) locally and deploy into the OpenShift via the binary build:

```
$ cd ~/coolstore/monolith
$ mvn clean package -Popenshift
$ oc start-build coolstore --from-file deployments/ROOT.war --follow
```

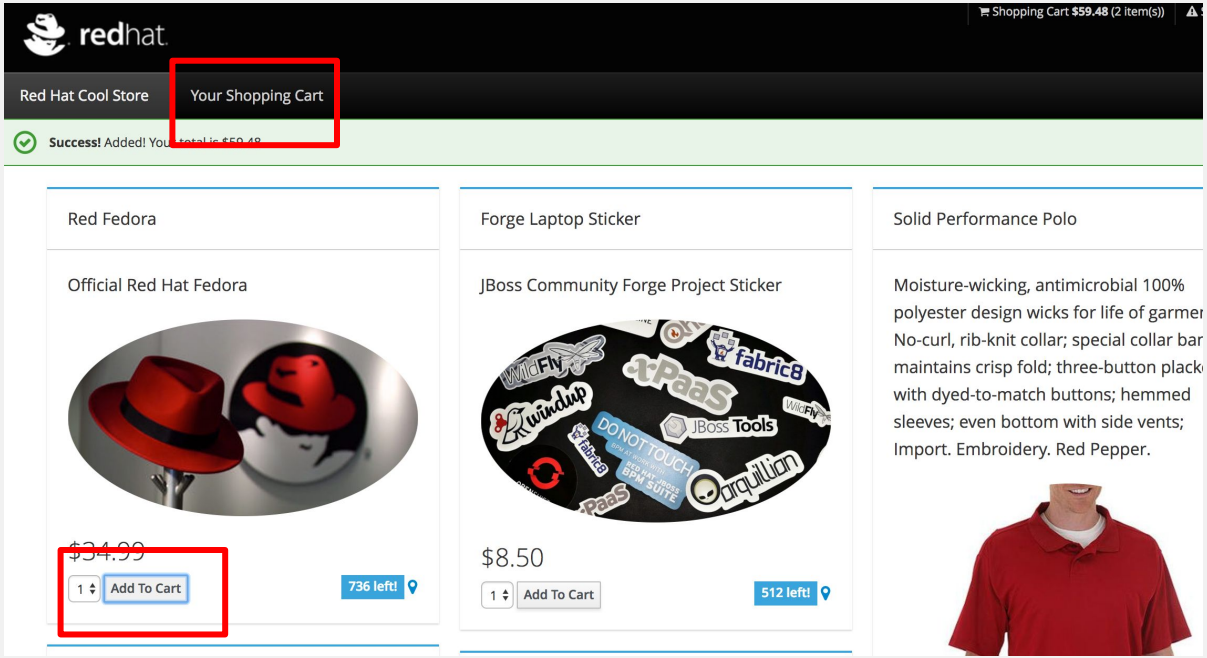
LAB 1 - FAST-MOVING MONOLITH

- 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:



LAB 1 - FAST-MOVING MONOLITH

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



LAB 1 - FAST-MOVING MONOLITH

- 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.
- Log into the database pod (with `postgresql` in the pod name) and inspect the values stored (you will need to copy/paste the password from the `POSTGRESQL_PASSWORD` environment variable when running the `psql` command):

```
% oc env dc/coolstore-dev-postgresql --list       # copy value of POSTGRESQL_PASSWORD to clipboard
$ oc get pods
$ oc rsh coolstore-dev-postgresql-1-3rfuw
sh-4.2$ psql -h $HOSTNAME -d $POSTGRESQL_DATABASE -U $POSTGRESQL_USER
Password for user userV31:XXXXXXXX
psql (9.5.4)
Type "help" for help.

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

LAB 1 - FAST-MOVING MONOLITH

- 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.
- Production environments will ideally use different projects, and even different nodes in an OpenShift cluster. For simplicity we have dev and prod in the same project.
- Create the production deployment objects that will be used to promote builds from dev to prod using the supplied CI/CD pipeline:

```
$ oc process -f ~/coolstore/monolith/src/main/openshift/template-prod.json | oc create -f -
```

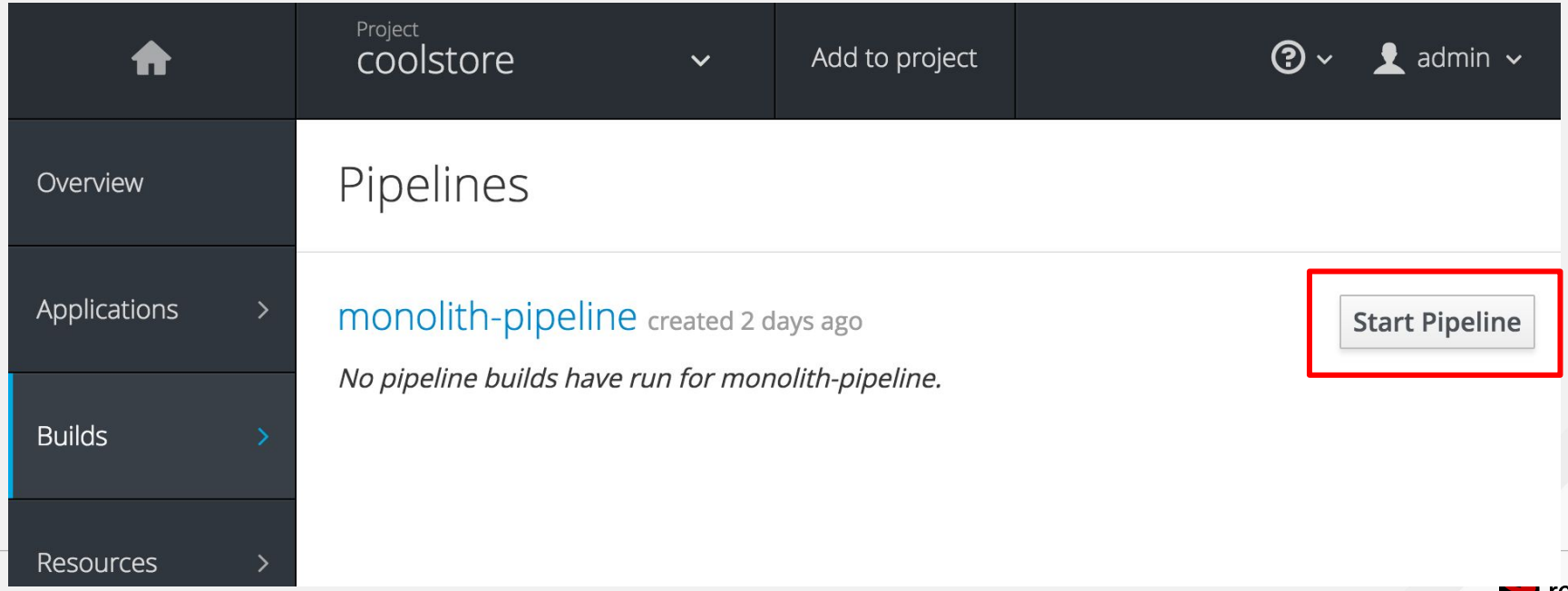
LAB 1 - FAST-MOVING MONOLITH

- Several types of OpenShift objects are created:
  - A production Postgres database deployment, service, and route
  - Services, Routes and DeploymentConfig for coolstore-prod (production env)
  - Service Accounts and Secrets (for cluster permissions)
  - A CI/CD server (Jenkins)
  - A CI/CD pipeline
- Notice that **no deployment occurs** for the production version of the app.
- Deployment of the production app occurs as a result of images from the dev environment being appropriately tagged. Inspect the pipeline and observe the steps it executes during the pipeline run. Notice at the end of the script, the latest coolstore image is tagged with :prod, triggering a production deployment:

```
$ oc get bc/monolith-pipeline -o yaml
...
          openshiftTag(sourceStream: 'coolstore', sourceTag: 'latest', namespace: '', destinationStream: 'coolstore',
destinationTag: 'prod', destinationNamespace: '')
...
```

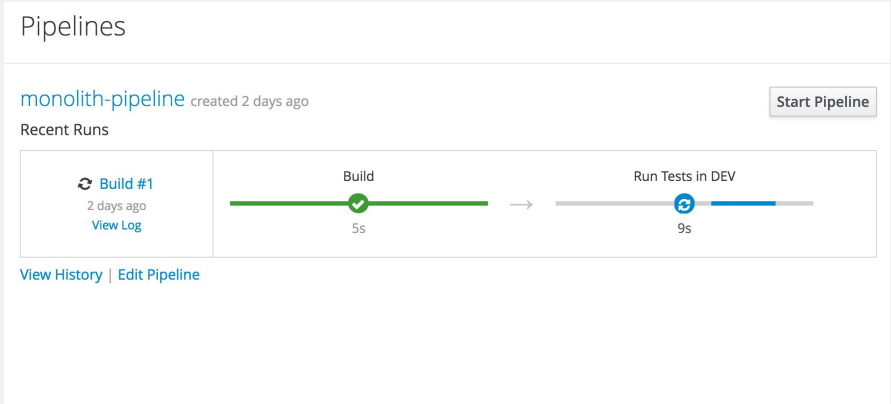
LAB 1 - FAST-MOVING MONOLITH

- Wait for the `jenkins` service to be completely available, then execute the pipeline by navigating to *Builds*→ *Pipelines* and click on *Start Pipeline* next to the Monolith Pipeline:



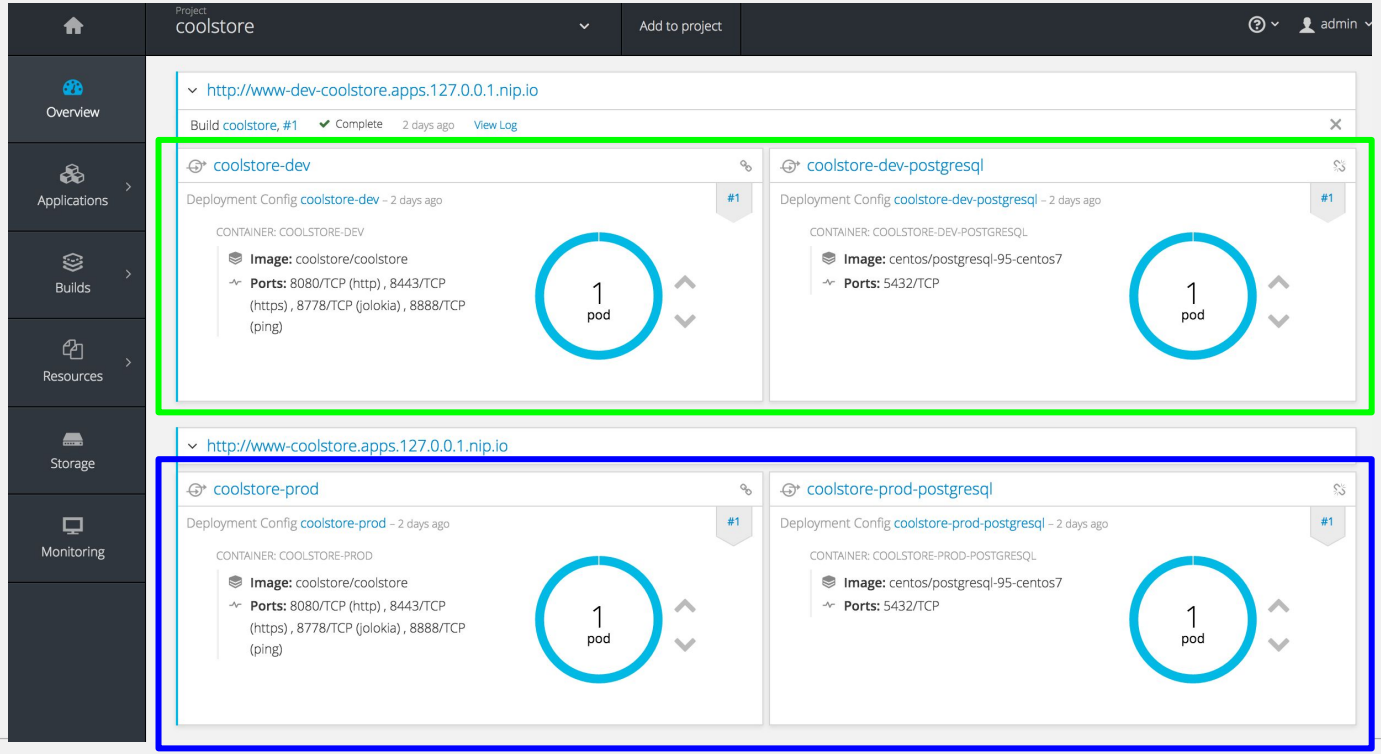
LAB 1 - FAST-MOVING MONOLITH

- Observe the pipeline as it executes its stages
- Once the pipeline is complete, the production app is deployed.
- Typically there would be a human approval step before going live
- Pipelines are also good at executing advanced deployment strategies (blue/green, canary)
- Return to the Overview page, wait for the deployment to complete, then click on the route for the `coolstore-prod` deployment to test the production app (identical to `coolstore-dev`).



LAB 1 - FAST-MOVING MONOLITH

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

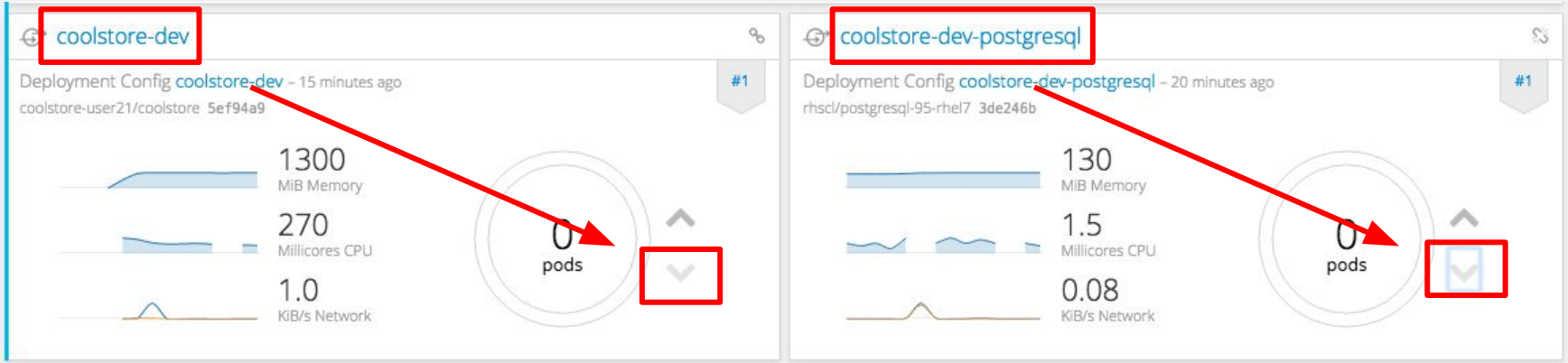


← Dev

← Prod

LAB 1 - FAST-MOVING MONOLITH

- (Optional) If your OpenShift environment seems show, you can scale down the coolstore-dev environment by clicking the down arrow next to the coolstore-dev and coolstore-dev-postgresql pods:



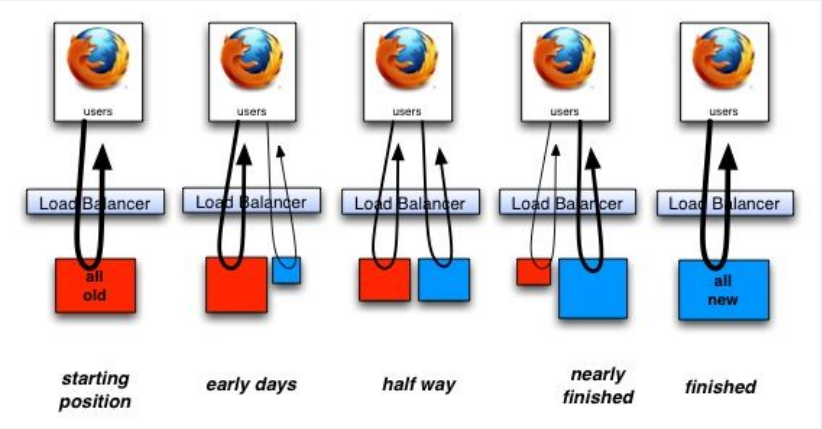
- (Optional) If your the build pipeline fails to start, you can manually tag the images:

```
$ oc tag coolstore:latest coolstore:prod
```


LAB 2
STRANGLING THE MONOLITH

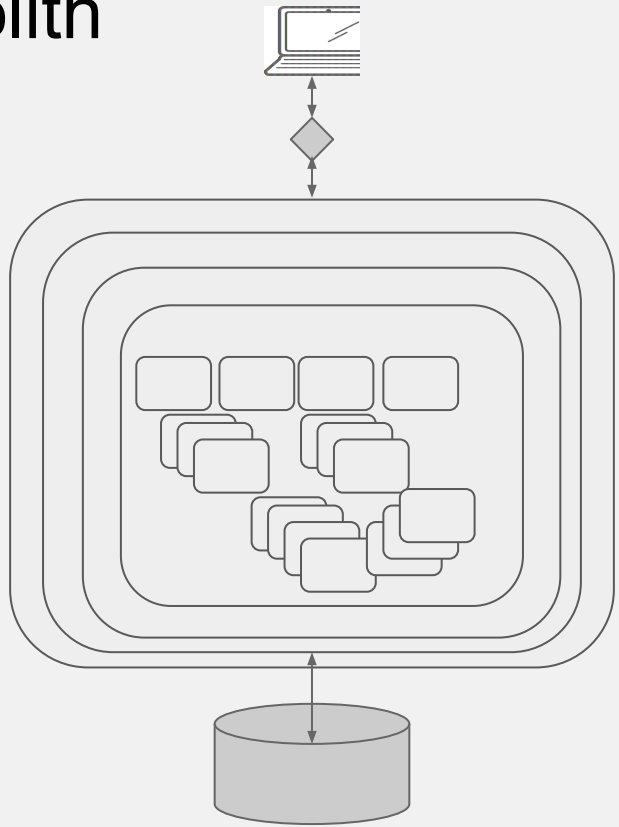
LAB 2 - STRANGLING THE MONOLITH

- 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.

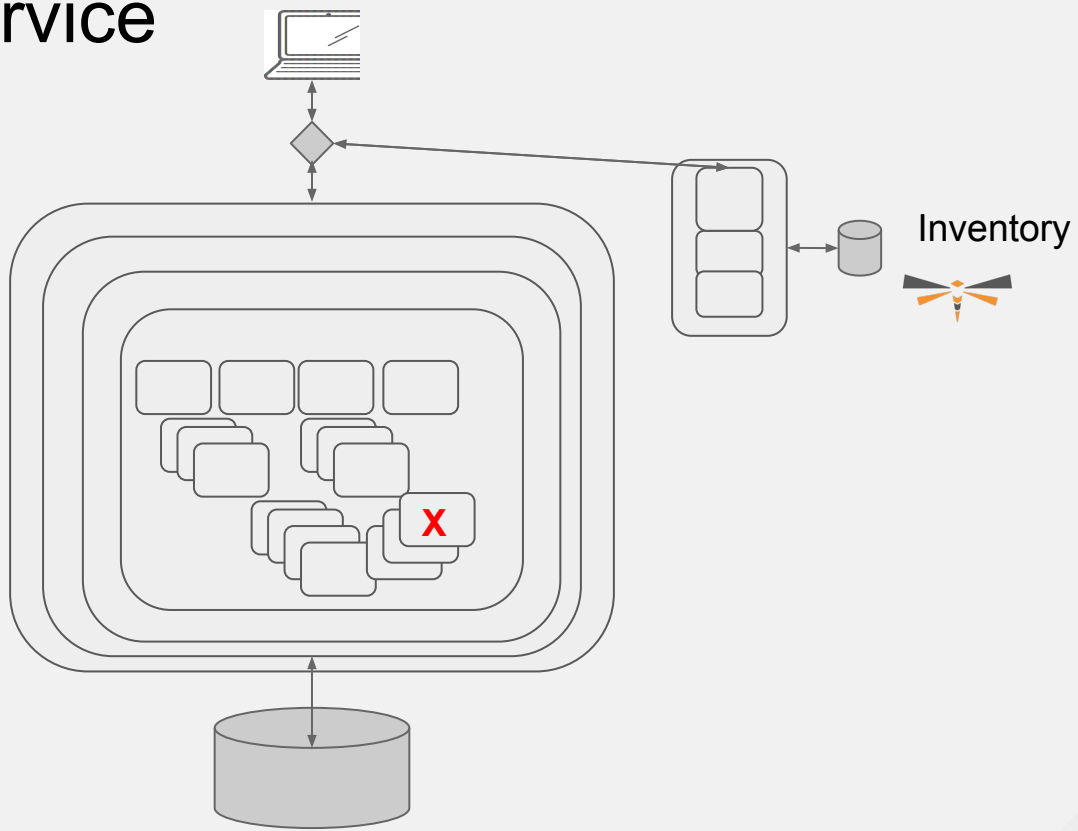


http://paulhammant.com/2013/07/14/legacy-application-strangulation-case-studies

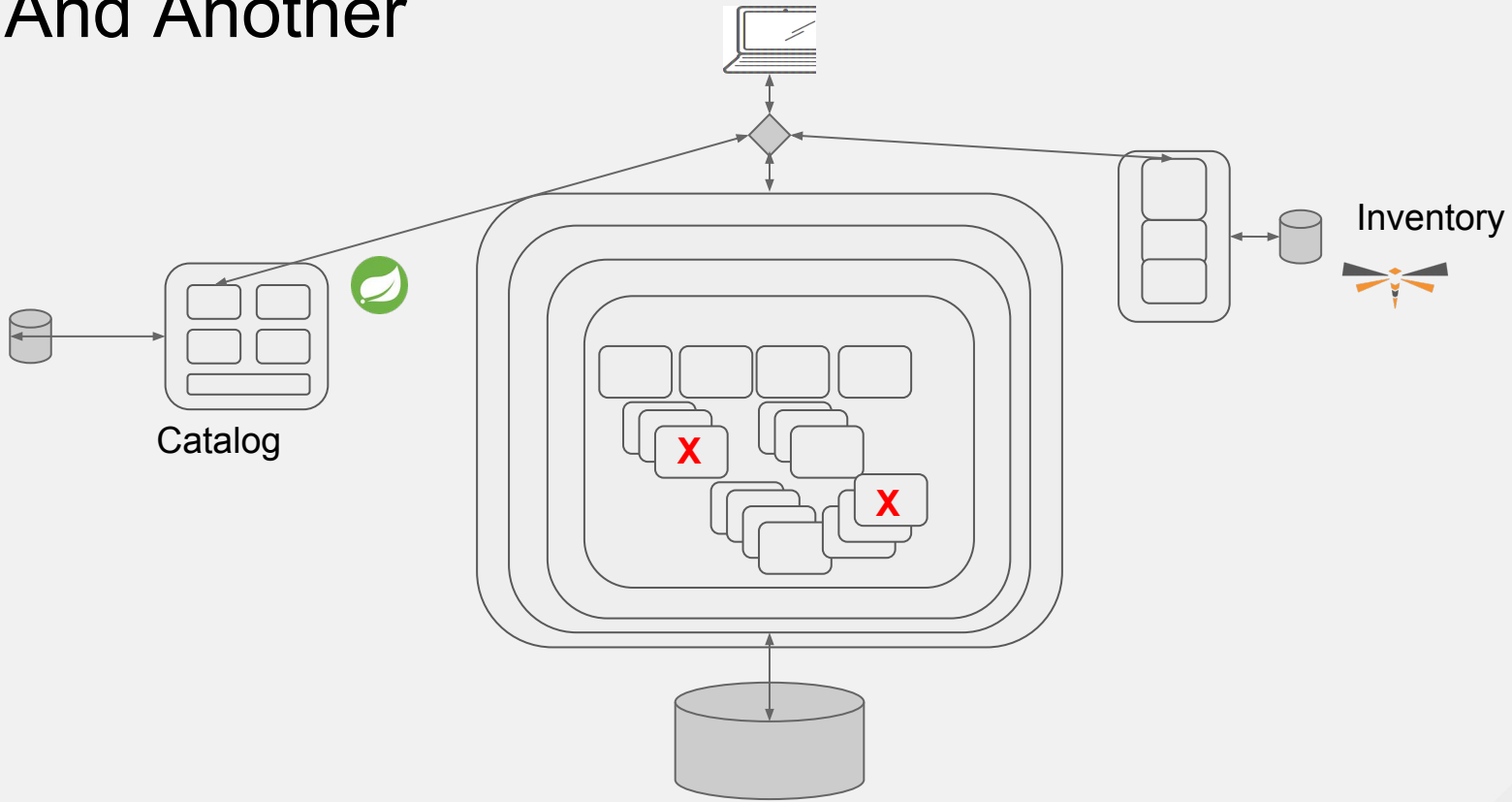
1) Strangle Monolith



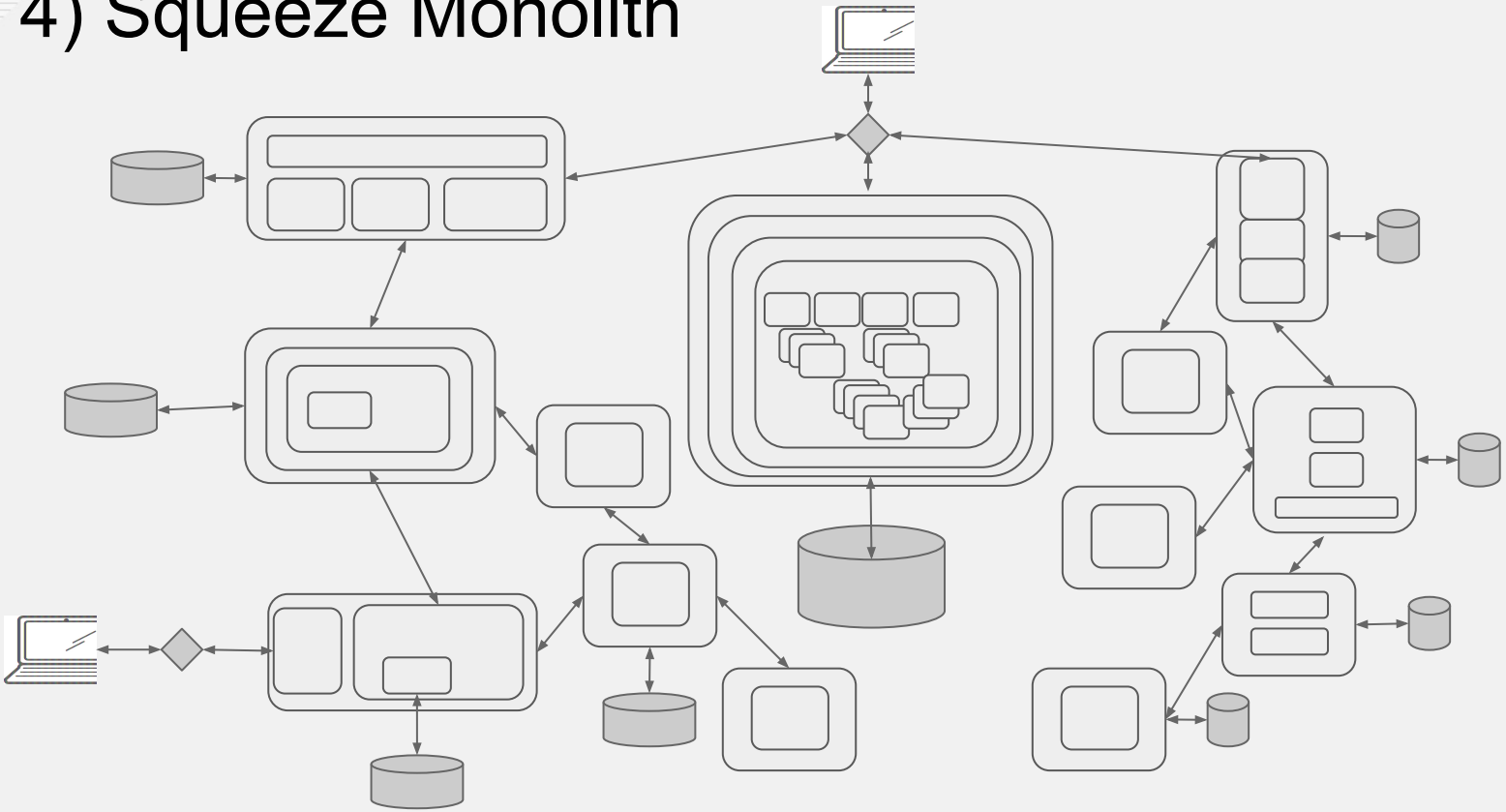
2) Add a Microservice



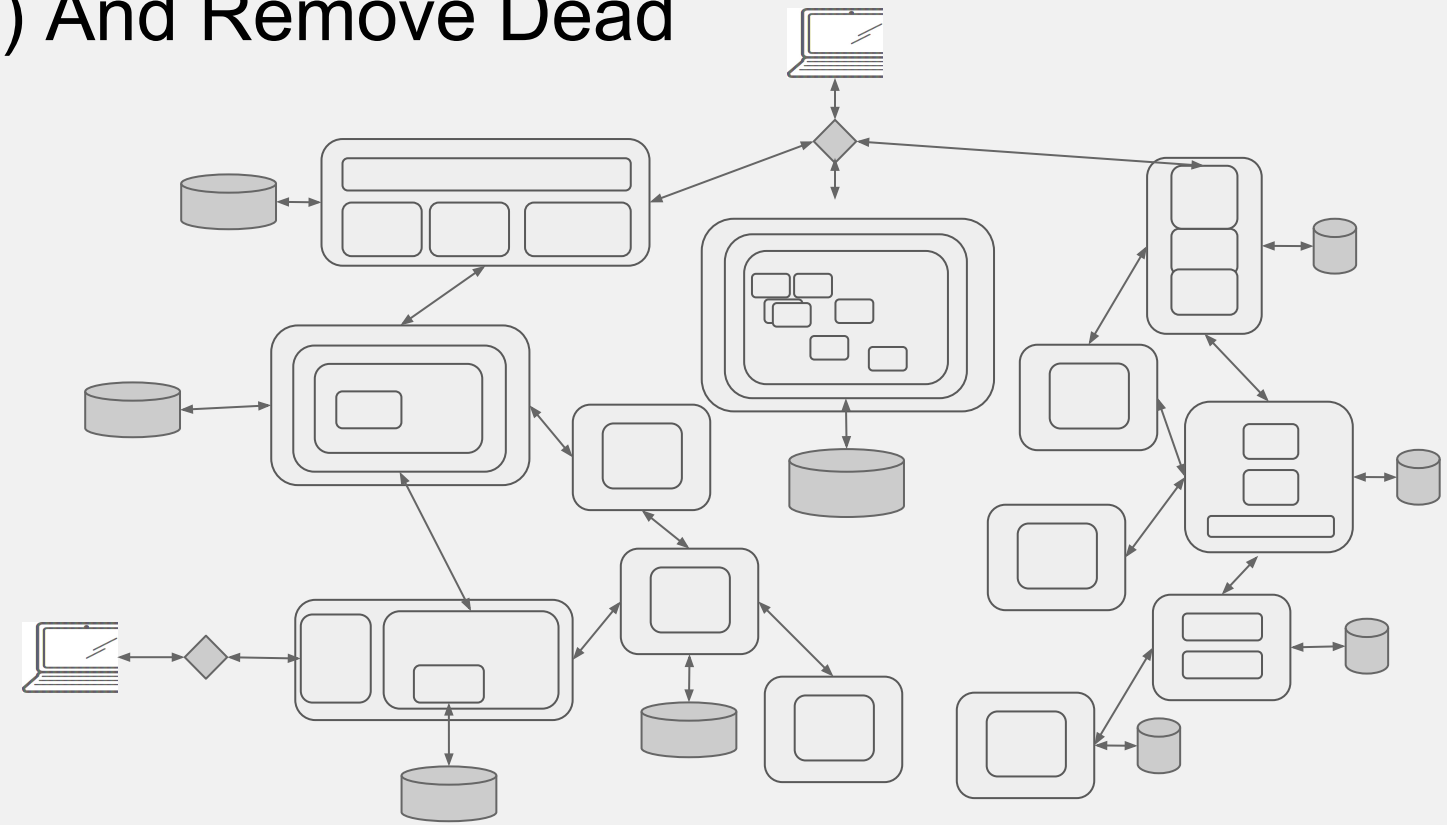
3) And Another



4) Squeeze Monolith



5) And Remove Dead



LAB 2 - STRANGLING 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

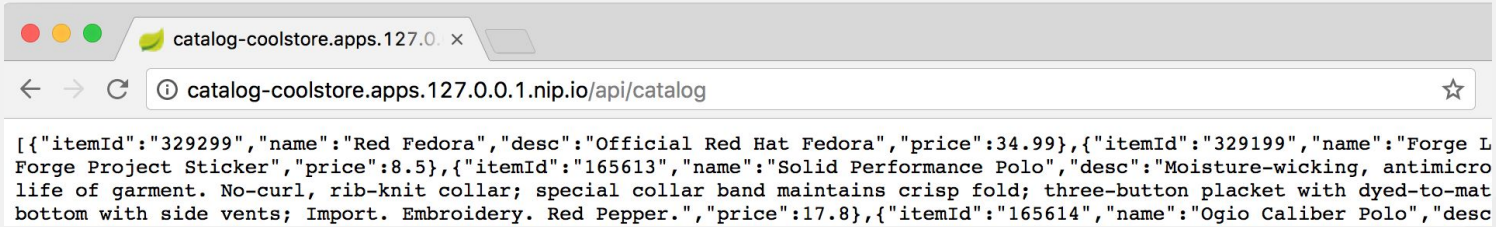
```
$ cd ~/coolstore
$ git clone -b app-partner https://github.com/modernize-legacy-apps/catalog
$ cd catalog
$ oc project coolstore-<USERNAME>
$ mvn clean fabric8:deploy -Popenshift -DskipTests
```

LAB 2 - STRANGLING THE MONOLITH

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

```
$ oc get route/catalog
NAME      HOST/PORT                                PATH      SERVICES   PORT      TERMINATION
catalog   catalog-coolstore.apps.127.0.0.1.nip.io             catalog    8080

$ curl http://catalog-coolstore.apps.127.0.0.1.nip.io/api/catalog

[{"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 `/api/catalog` to the URL:



LAB 2 - STRANGLING THE MONOLITH

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

```
$ cd ~/coolstore
$ git clone -b app-partner https://github.com/modernize-legacy-apps/inventory
$ cd inventory
$ oc project coolstore-<USERNAME>
$ 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!

LAB 2 - STRANGLING THE MONOLITH

- The inventory microservice has a separate database from the current monolith.
- This allows teams split along business concerns to independently develop, test, deploy and scale the service and its database
- Once the new inventory service is deployed, dump its database:

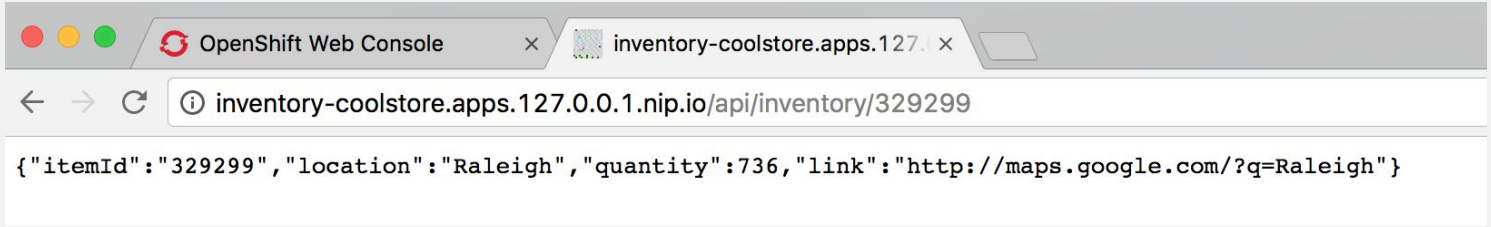
```
% oc env dc/inventory-database --list # get the POSTGRESQL_PASSWORD
$ oc get pods
$ oc rsh inventory-database-1-kkxs2
sh-4.2$ psql -h $HOSTNAME -d $POSTGRESQL_DATABASE -U $POSTGRESQL_USER
Password for user userV31:XXXXXXXX
psql (9.5.4)
Type "help" for help.

monolith=> select * from inventory;
 itemid |              link                | location | quantity
--------+----------------------------------+----------+----------
 329299 | http://maps.google.com/?q=Raleigh | Raleigh  |      736
...
```

LAB 2 - STRANGLING 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

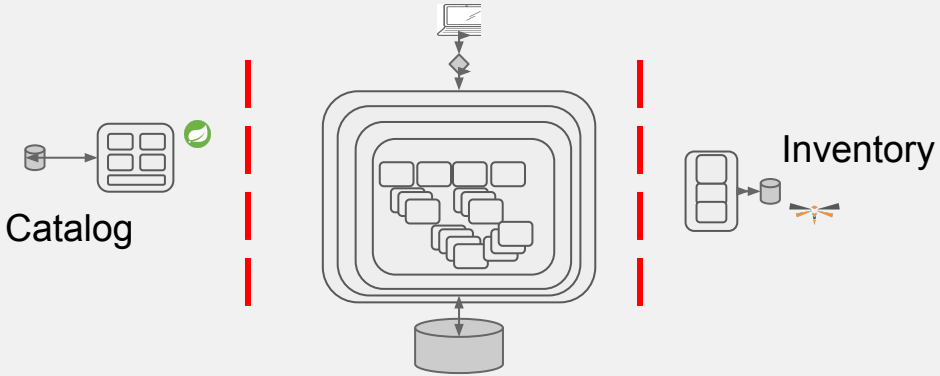
```
$ oc get route/inventory
NAME        HOST/PORT                                   PATH      SERVICES    PORT      TERMINATION
inventory   inventory-coolstore.apps.127.0.0.1.nip.io             inventory   8080

% curl http://inventory-coolstore.apps.127.0.0.1.nip.io/api/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 `/api/inventory/329299` to the URL:



LAB 2 - STRANGLING THE MONOLITH

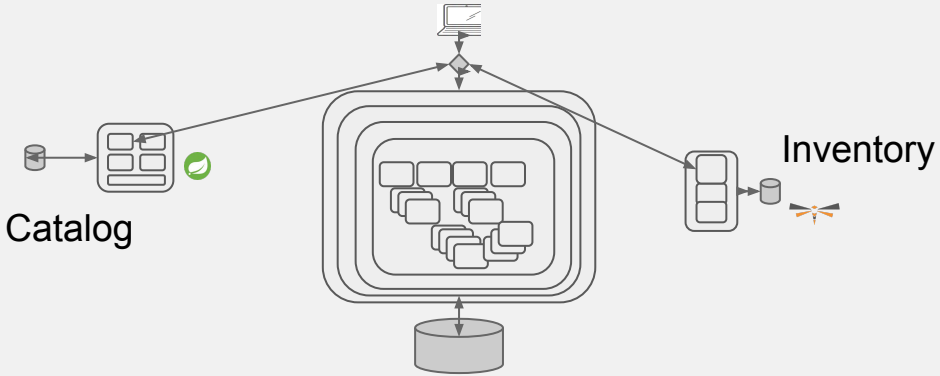
- You have begun to replace your monolith's services with new, lightweight Java-based microservices - congratulations!
- Although deployed, no web traffic will ever hit these new services as nothing (including the existing UI) knows of their existence.
- Microservice architectures have much more deployment 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.



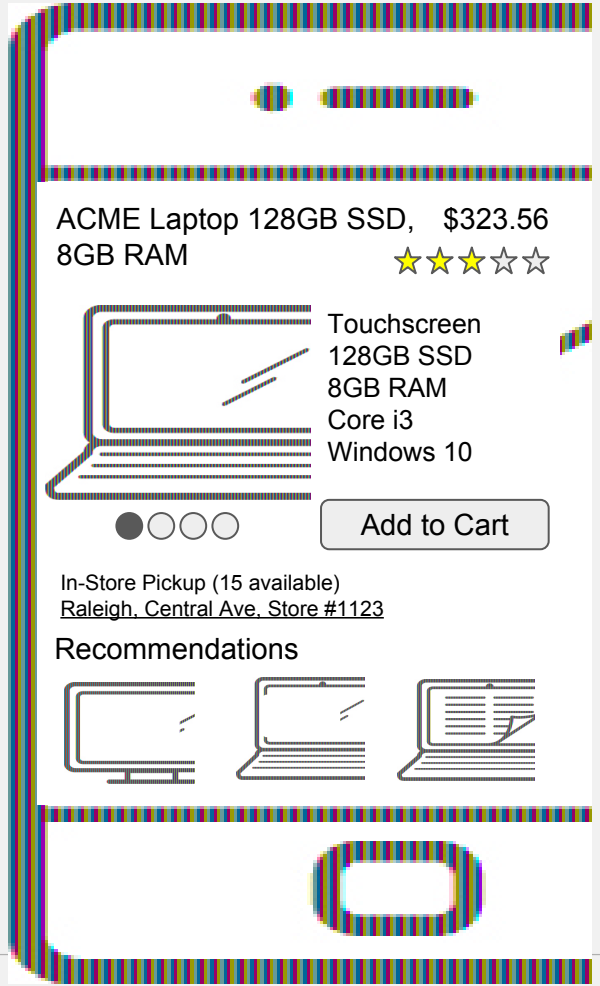

LAB 3
MICROSERVICE INTEGRATION
PATTERNS
redhat.

LAB 3 - 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.

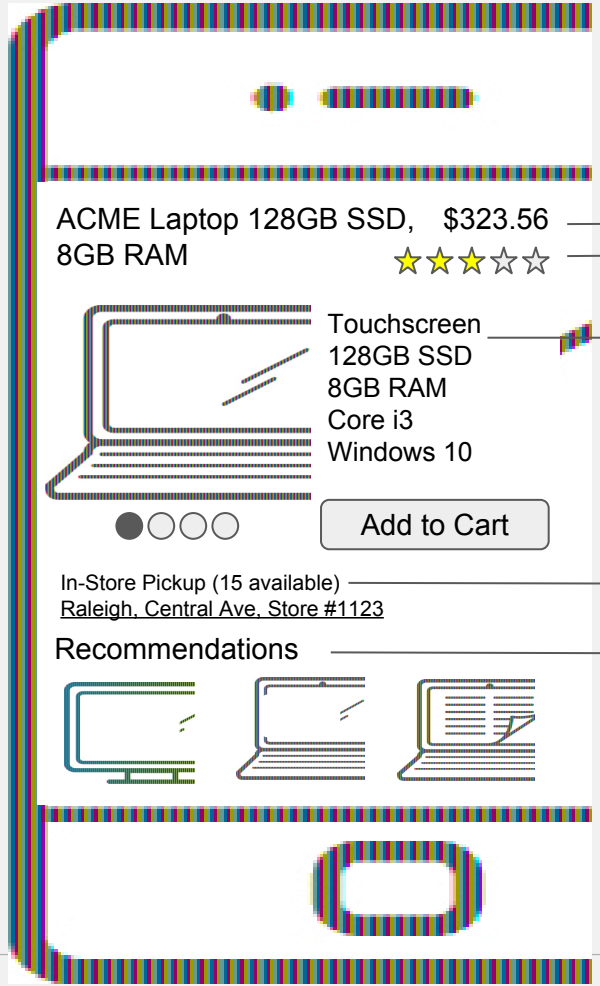


Example

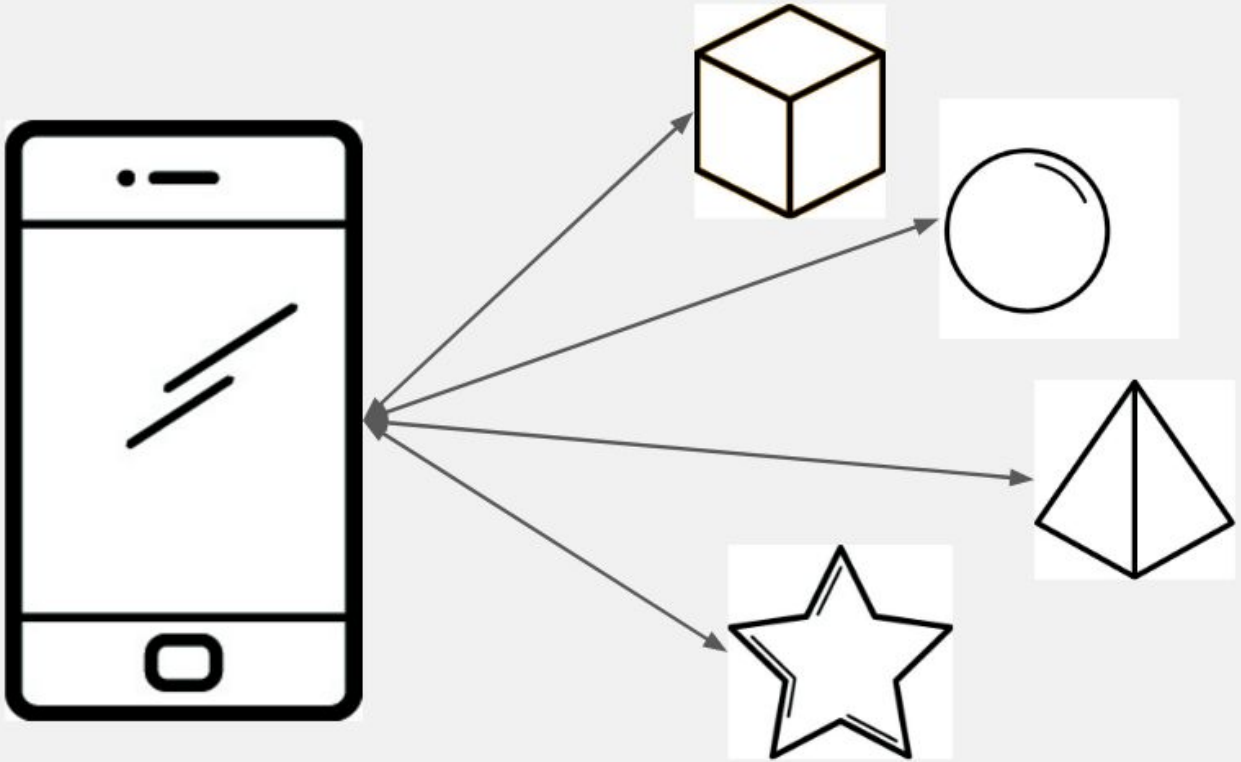


- This is very similar to our CoolStore retail application!
- UI constructed from multiple microservices implemented across business boundaries
- Typically runs in constrained environments like mobile
- Real-world apps consist of 100s of services

Example



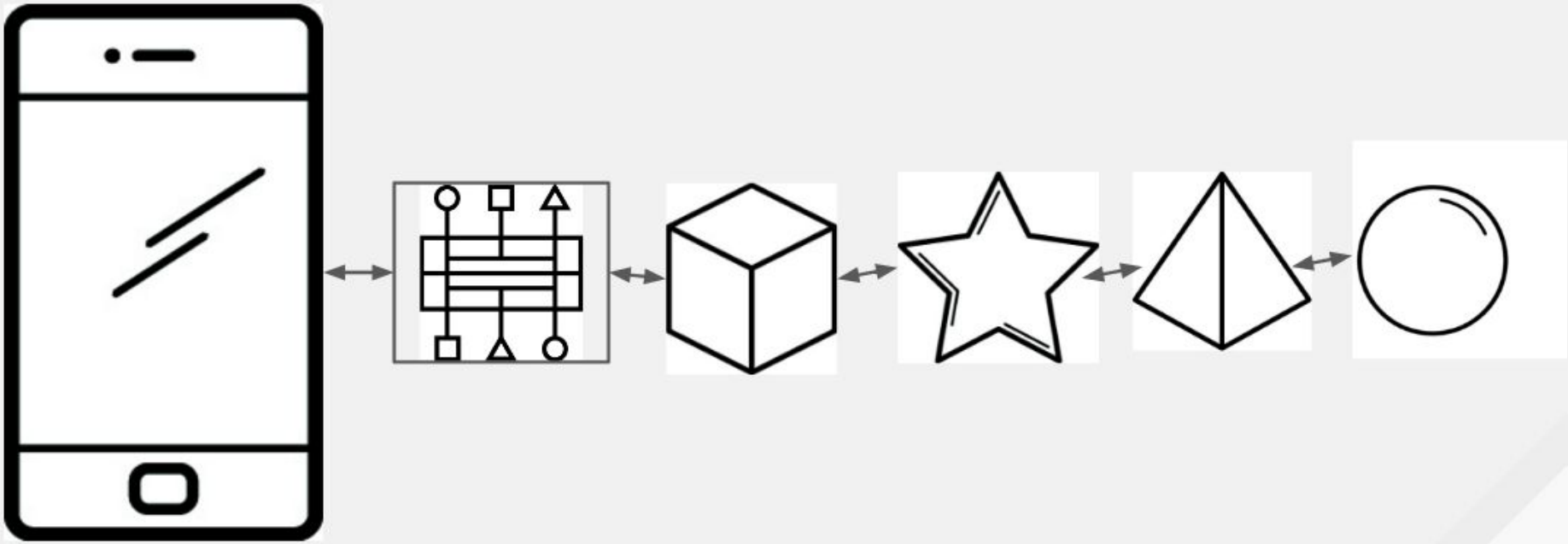
Option 1: Client Aggregation



LAB 3 - MICROSERVICE INTEGRATION PATTERNS

- 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

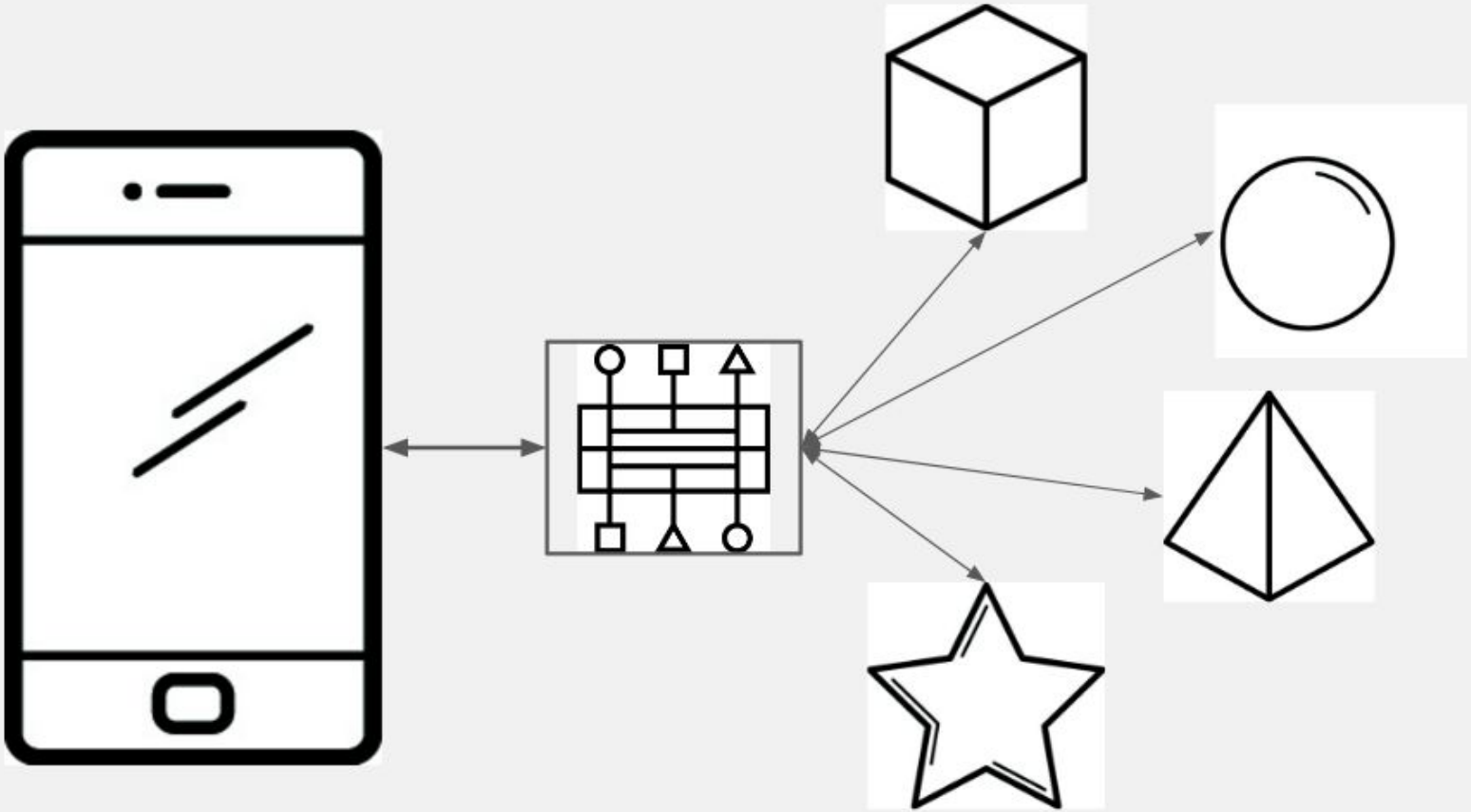
Option 2: Chaining



LAB 3 - MICROSERVICE INTEGRATION PATTERNS

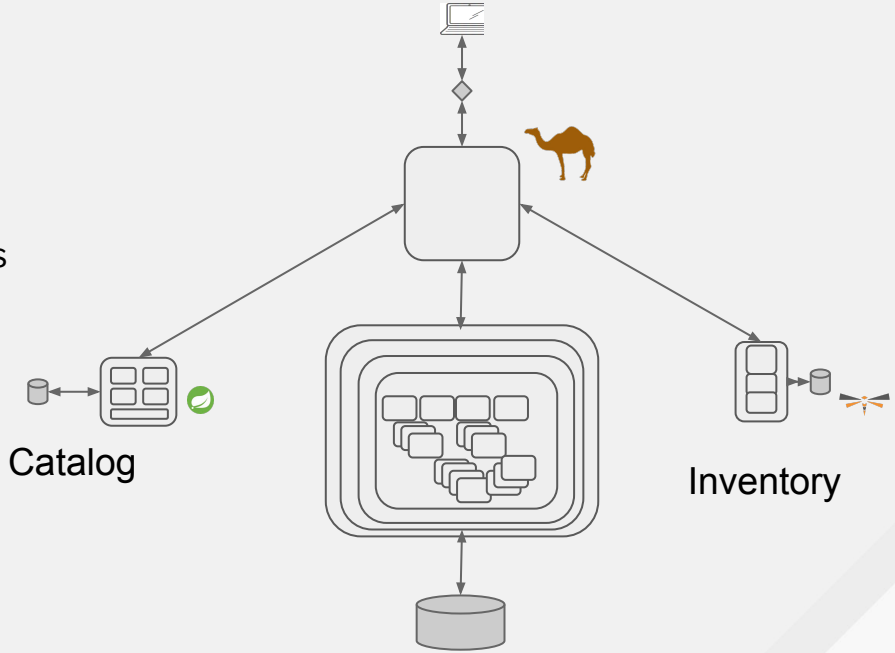
- Chaining means that one microservice calls another, which calls another, etc.
- A complete chain is typically not desirable or necessary, but short chains are OK
- Chaining benefits
  - Client code simpler - there is only a single entry into the chain
  - Less network bandwidth (also due to single 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)

Option 3: API “Gateway”



LAB 3 - MICROSERVICE INTEGRATION PATTERNS

- 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



LAB 3 - 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 ~/coolstore
$ git clone -b app-partner https://github.com/modernize-legacy-apps/gateway
$ cd gateway
$ oc project coolstore-<USERNAME>
$ mvn clean fabric8:deploy -Popenshift -DskipTests
```

LAB 3 - MICROSERVICE INTEGRATION PATTERNS

- The Coolstore Gateway microservice is a Spring Boot application that implements its logic using an Apache Camel *route*.
- Take a look at the code for the Gateway:
- In your favorite text editor, open `src/main/java/com/redhat/coolstore/api_gateway/ProductGateway.java`
- Or in your browser: https://github.com/modernize-legacy-apps/gateway/blob/master/src/main/java/com/redhat/coolstore/api_gateway/ProductGateway.java

LAB 3 - MICROSERVICE INTEGRATION PATTERNS

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

- 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`).

LAB 3 - MICROSERVICE INTEGRATION PATTERNS

```
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.

LAB 3 - MICROSERVICE INTEGRATION PATTERNS

```
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:

LAB 3 - MICROSERVICE INTEGRATION PATTERNS

```
@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;

}
```

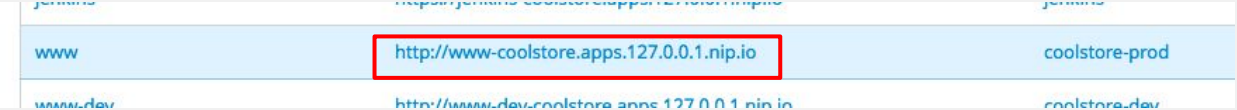
- This is the enricher logic which takes the Product and matching Inventory objects, *enriches* the Product object with information from the Inventory object, and returns it.
- The resulting list sent back to the client is the list of products, each of which is enriched with inventory information
- The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

LAB 3 - MICROSERVICE INTEGRATION PATTERNS

- 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.

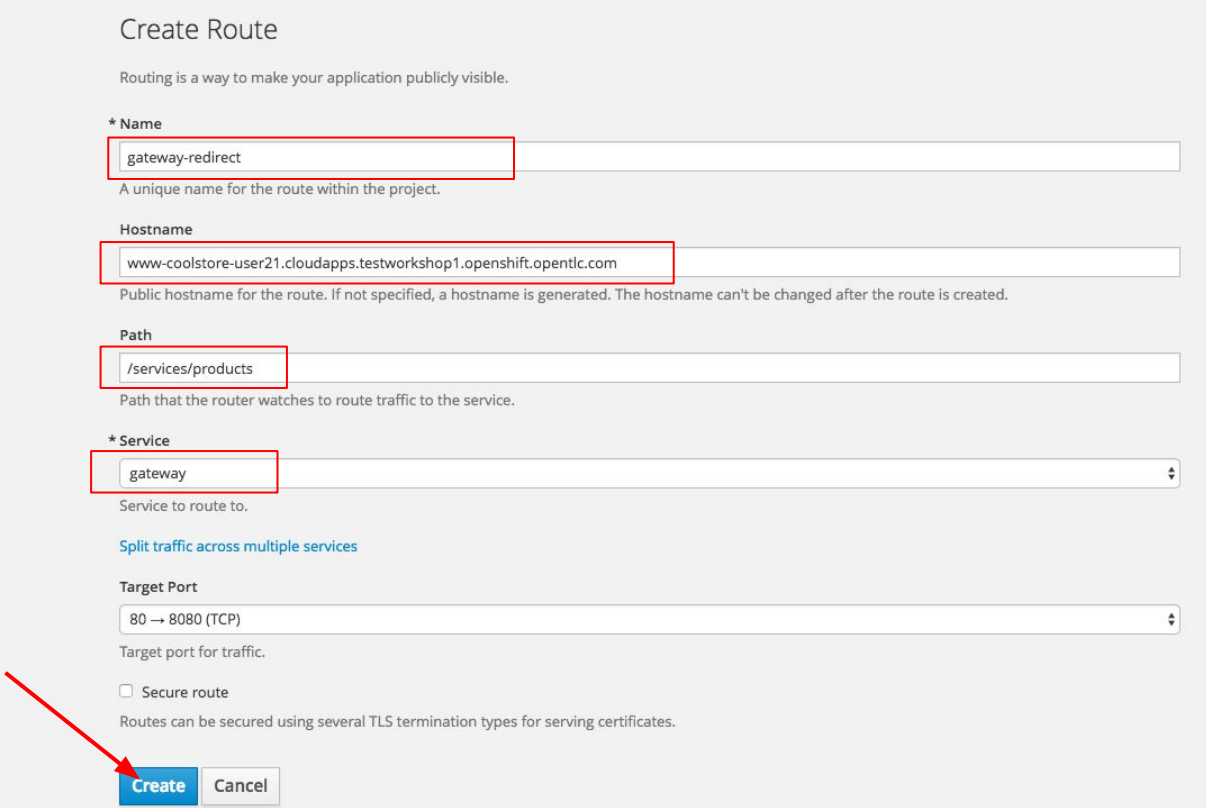
LAB 3 - MICROSERVICE INTEGRATION PATTERNS

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



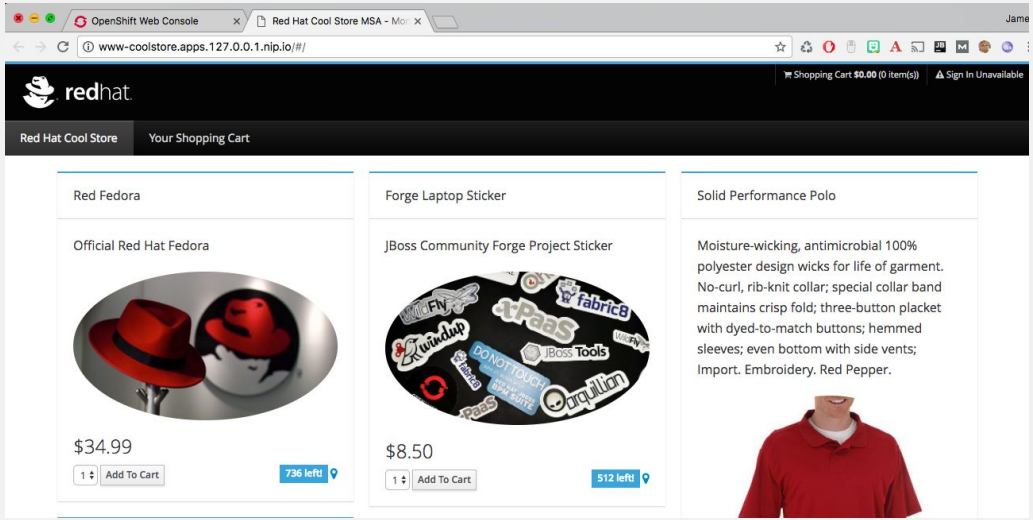
- Click *Create Route* to begin creating a new route with the following values:
  - Name: gateway-redirect
  - Hostname: The full hostname of the existing route **as seen above** (without the `http://`). For example,
    - **`www-coolstore-user21.cloudapps.testworkshop1.openshift.opentlc.com`**
  - Path: **`/services/products`**
  - Service: **`gateway`**
  - Leave other values as-is (see next page for complete example)
- Click Create to create the route

LAB 3 - MICROSERVICE INTEGRATION PATTERNS



LAB 3 - MICROSERVICE INTEGRATION PATTERNS

- 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?



LAB 3 - MICROSERVICE INTEGRATION PATTERNS

- 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 your preferred text editor (vi, gedit, etc) 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

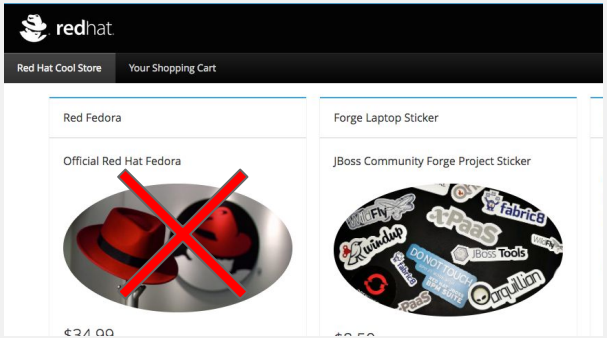
```
//
// Uncomment the below lines to filter out products
//
//            .process(exchange -> {
//                   List<Product> originalProductList = (List<Product>)exchange.getIn().getBody(List.class);
//                   List<Product> newProductList = originalProductList.stream().filter(product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

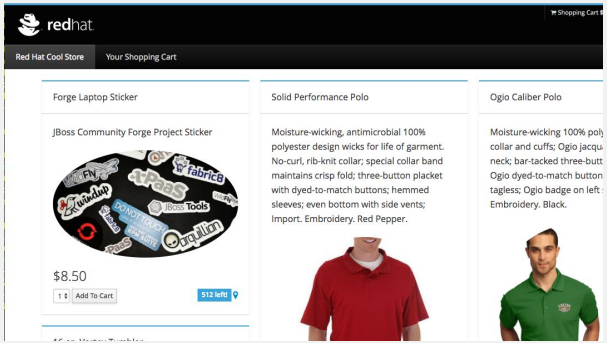
LAB 3 - MICROSERVICE INTEGRATION PATTERNS

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

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

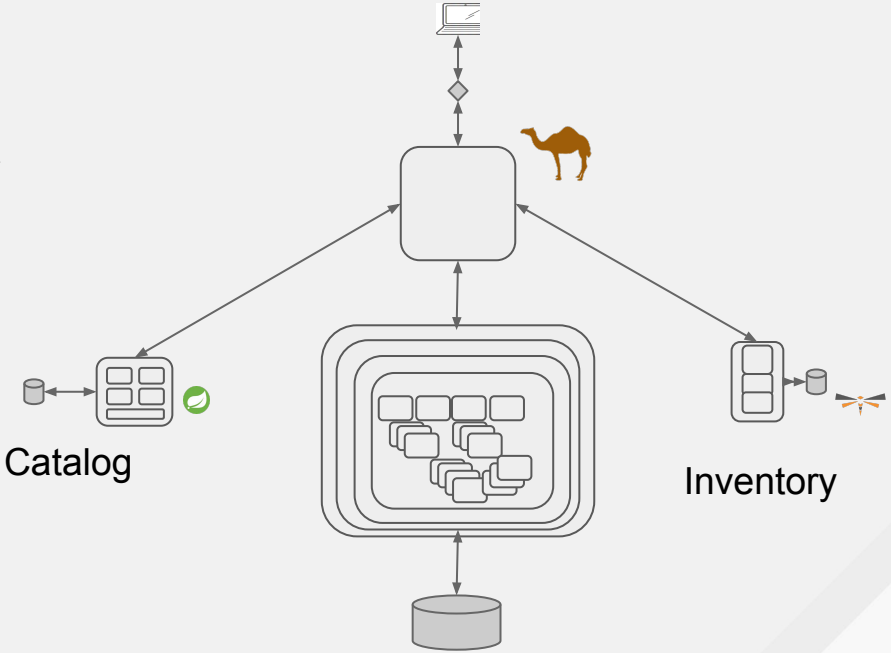
- Once the new version of the code is deployed and up and running, reload the browser and the Red Hat Fedora product should be gone:





LAB 3 - MICROSERVICE INTEGRATION PATTERNS

- 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.



redhat.

YOU MADE IT!

James Falkner
Sr. Technical Marketing Manager, Middleware
April 2017