Lab : Scaling and Connecting Your Services - Service discovery

In this chapter, we will look in greater detail at the process of deploying, scaling, and connecting your applications. You have already learned the basic information about deploying services to the OpenShift cloud. Now it's time to extend this knowledge and learn how to use it in practice.

Pre-reqs:

- https://www.katacoda.com/athertahir/courses/cloud-development-with-wildfly/cloud-development-with-wildfly-chapter-07

Service discovery

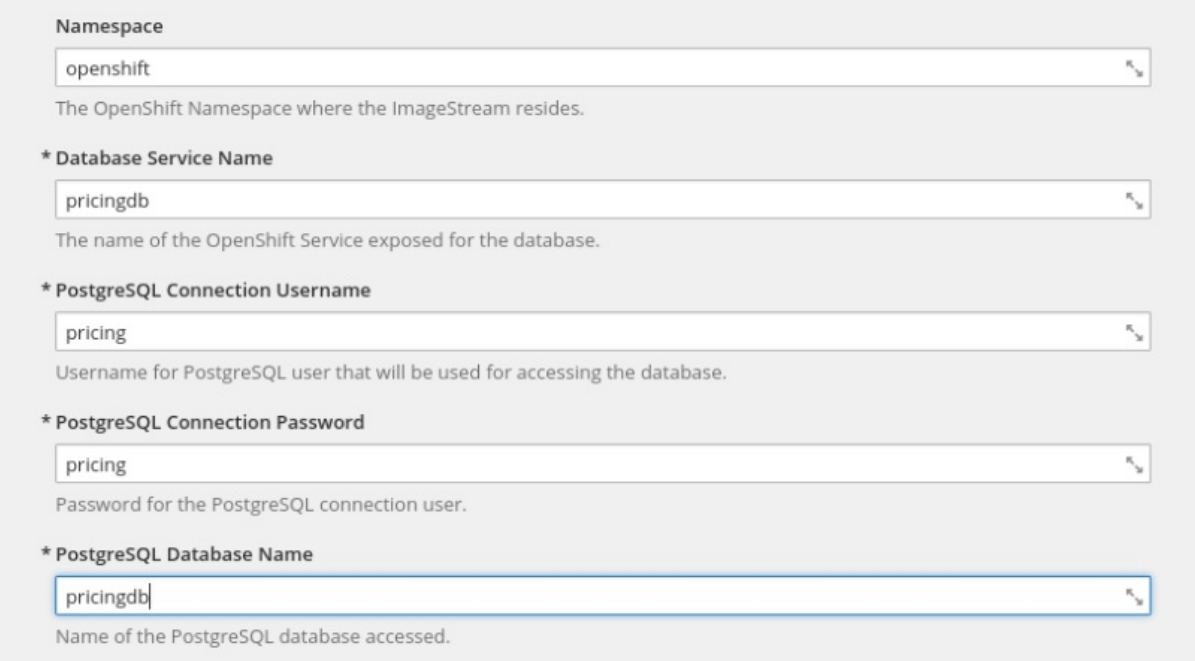
We have already shown you how to configure balancing for our application. We know now that you have access to the virtual cluster IP address behind which the request is being balanced by OpenShift. However, how do we actually know how to connect to our services? We are going to learn that in the next topic. Before we do that, we must introduce our new services that will be talking to each other.

New services

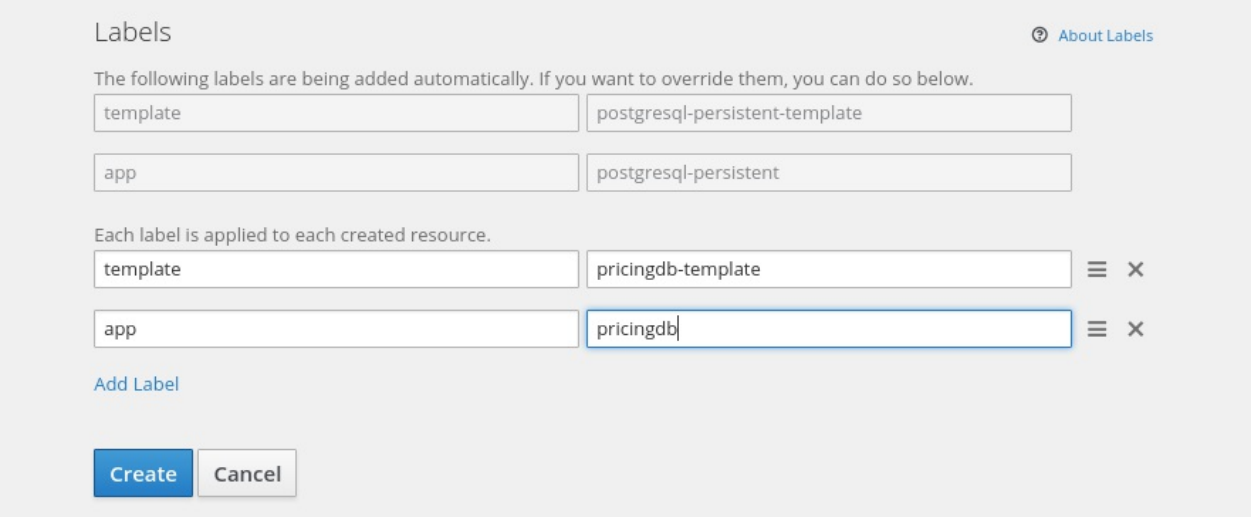
In the first chapter, we briefly introduced the pet store application and described the services that constitute it. By now, we have used solely the catalog service in our examples. Now it's time to implement both the pricing service and customer gateway service. These services will serve as an example in this and the future chapters. Let's start with the pricing service.

The pricing service

The pricing service is very similar to catalog service. It can be used to obtain prices for a pet using their names. Let's go straight to the implementation. Initially, we have to create the database. As before, we will use the PostgreSQL template:



As with the catalog service's database, we would also like to override the labels:



To populate the database, we have to create the following script:

```
vi pets.sql
```

Now, enter the sample data:

```
DROP TABLE IF EXISTS PRICE;

CREATE TABLE PRICE (id serial PRIMARY KEY, item_id varchar, price smallint);

INSERT INTO PRICE(item_id, price) VALUES ('dbf67f4d-f1c9-4fd4-96a8-65ee1a22b9ff', 50);
INSERT INTO PRICE(item_id, price) VALUES ('fc7ee3ea-8f82-4144-bcc8-9a71f4d871bd', 30);
INSERT INTO PRICE(item_id, price) VALUES ('725dfad2-0b4d-455c-9385-b46c9f356e9b', 15);
INSERT INTO PRICE(item_id, price) VALUES ('a2aa1ca7-add8-4aae-b361-b7f92d82c3f5', 3000);
```

To populate the database, we will execute the following script:

```
psql -U pricing pricingdb < pets.sql
```

Our pricing database is ready. We can now start writing the code.

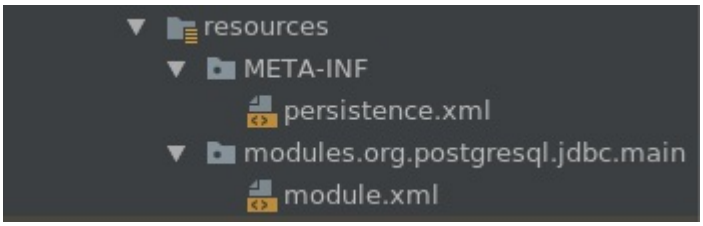
Note

Examples reference: `chapter8/pricing-service`.

We have to configure the database in the similar way that we did for `catalog-service`.

```
swarm:
  datasources:
    data-sources:
      PricingDS:
       driver-name: postgresql
       connection-url: jdbc:postgresql://pricingdb.petstore.svc/pricingdb
       user-name: pricing
       password: pricing
jdbc-drivers:
      postgresql:
        driver-class-name: org.postgresql.Driver
xa-datasource-name: org.postgresql.xa.PGXADataSource
driver-module-name: org.postgresql.jdbc
```

In order for the database to work, we have to provide the JDBC driver module:



As you can see, we also need `persistence.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="2.1"
xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="PricingPU" transaction-type="JTA">
        <jta-data-source>java:jboss/datasources/PricingDS</jta-data-source>
    </persistence-unit>
</persistence>
```

We have to provide an `Entity`:

```
package org.packt.swarm.petstore.pricing;

import com.fasterxml.jackson.annotation.JsonIgnore;
```

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;

//1
@Entity
//2
@Table(name = "Price")
//3
@NamedQueries({
        @NamedQuery(name="Price.findByName",
                query="SELECT p FROM Price p WHERE p.name = :name"),
})
public class Price {

    //4
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "price_sequence")
    @SequenceGenerator(name = "price_sequence", sequenceName = "price_id_seq")
    //5
    @JsonIgnore
    private int id;

    //6
    @Column(length = 30)
    private String name;
    @Column
    private int price;

    public int getId() {
return id;
}

public void setId(int id) {
this.id = id;
}

public String getName() {
return name;
}

public void setName(String name) {
this.name = name;
}


public int getPrice() {
return price;
}

public void setPrice(int price) {
this.price = price;
}
}
```

In the preceding snippet, we have created a JPA entity (1), which references the `"Price"` table that we have just created (2). We have provided `NamedQueries`, which will enable us to search the price of a pet by a name (3). An `id`, as in `catalogdb`, is generated using the Postgres sequence (4) and is not parsed in the JSON response (5). Finally, we have annotated the fields mapped to the `price` and `name` columns (6).

As in `catalog-service`, we will need a service:

```
package org.packt.swarm.petstore.pricing;

import org.packt.swarm.petstore.pricing.model.Price;

import javax.enterprise.context.ApplicationScoped;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.ws.rs.WebApplicationException;
import java.util.List;

@ApplicationScoped
public class PricingService {

@PersistenceContext(unitName = "PricingPU")
private EntityManager em;

    public Price findByItemId(String itemId) {
        return em.createNamedQuery("Price.findByItemId", Price.class).setParameter("itemId", itemId).getSingleResult();
}
}
```

We also need REST resource:

```
package org.packt.swarm.petstore.pricing;

import org.packt.swarm.petstore.pricing.model.Price;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.SecurityContext;

@Path("/")
public class PricingResource {

    @Inject
    private PricingService pricingService;

    @GET
    @Path("price/{item_id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response priceByName(@PathParam("item_id") String itemId) {
        Price result = pricingService.findByItemId(itemId);
        return Response.ok(result).build();
}
```

We would also need an application:

```
package org.packt.swarm.petstore.pricing;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/")
public class PricingApplication extends Application {
}
```

Our second service is ready. It's time to deploy it on OpenShift. Push your application to your GitHub repository and invoke:

```
oc new-app wildflyswarm-10-centos7~https://github.com/PacktPublishing/Hands-On-Cloud-Development-with-WildFly.git --contex
```

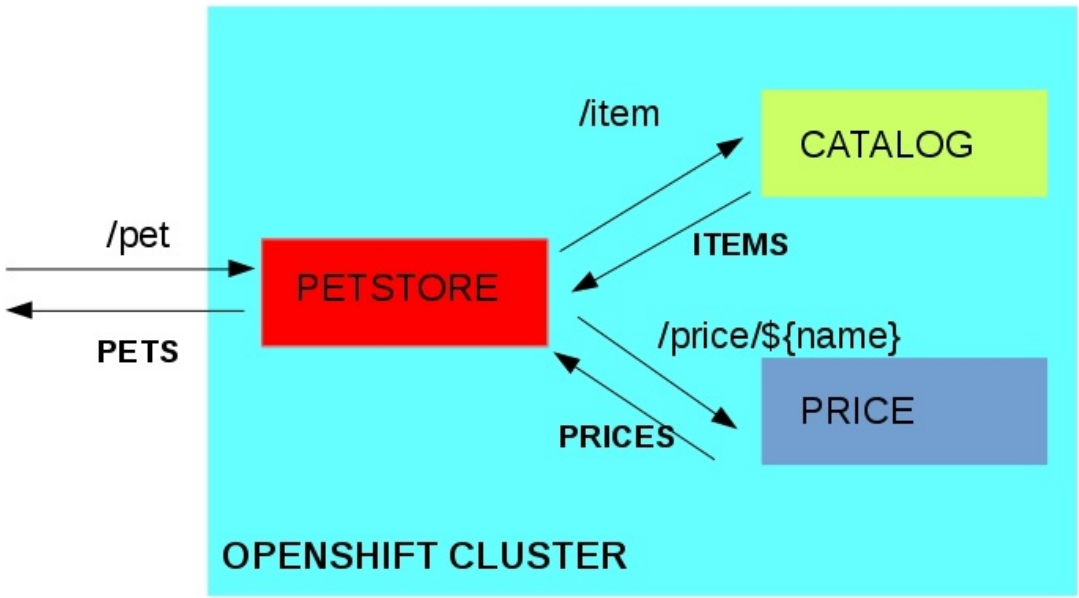
After your application is deployed, you can create a route to it and verify that it indeed w orks:


```
pricing-service-petstore.<update-me>-80-<update-me>.environments.katacoda.com/price/dbf67f4d-f1c9-4fd4-96a8-65ee1a22b9ff
```

It indeed does. Let's move to the second service.

The customer gateway service

In this section, the stuff becomes more interesting again. The customer-gatew ay service is a gatew ay to our application, w hich w ould provide the external interface for the w eb client. The first request that w e w ill implement is obtaining the list of pets. Let's take a look at the follow ing diagram:



When the `/catalog/item` request is being executed, the

service asks **CATALOG** for available items. Based on that information, the pet store service asks the **PRICE** service about the price of each pet, merges the results, and then returns them to the client. However, how will the gateway service know the addresses of the services? We will find that out soon.

Note

Examples reference: `chapter8/customer-gateway-env`.

The customer service is configured in a similar way to previous services. If you have doubts regarding some parts of configuration please refer to the description of those.

Let's look at the implementation details of `catalog/item` request starting with the REST resource:

```
package org.packt.swarm.petstore;

import org.packt.swarm.petstore.api.CatalogItemView;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import java.util.List;

@Path("/")
public class GatewayResource {

@Inject
private GatewayService gatewayService;

    //1
    @GET
    @Path("/catalog/item")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getItems() {
        //2
        List<CatalogItemView> result = gatewayService.getItems();
        return Response.ok(result).build();
    }
```

The `getItems` method gathers the items from the `CatalogService` (1), obtains a price for all of them, and merges the obtained results into the list of pets available in the store. Please note that we have introduced `CatalogItemView` —a transport object which is a part of the API for the web client.

We have also implemented the service:

```
package org.packt.swarm.petstore;

import org.packt.swarm.petstore.api.CatalogItemView;
import org.packt.swarm.petstore.catalog.api.CatalogItem;
import org.packt.swarm.petstore.pricing.api.Price;
import org.packt.swarm.petstore.proxy.CatalogProxy;
import org.packt.swarm.petstore.proxy.PricingProxy;

import javax.enterprise.context.ApplicationScoped;
```

```
import javax.inject.Inject;
import java.util.ArrayList;
import java.util.List;

@ApplicationScoped
public class GatewayService {

    //2
    @Inject
    private CatalogProxy catalogProxy;

    @Inject
    private PricingProxy pricingProxy;



    //1
    public List<CatalogItemView> getItems() {
        List<CatalogItemView> views = new ArrayList<>();
        for(CatalogItem item: catalogProxy.getAllItems()) {
            Price price = pricingProxy.getPrice(item.getItemId());

            CatalogItemView view = new CatalogItemView();
            view.setItemId(item.getItemId());
            view.setName(item.getName());
            view.setPrice(price.getPrice());
            view.setQuantity(item.getQuantity());
            view.setDescription(item.getDescription());
```

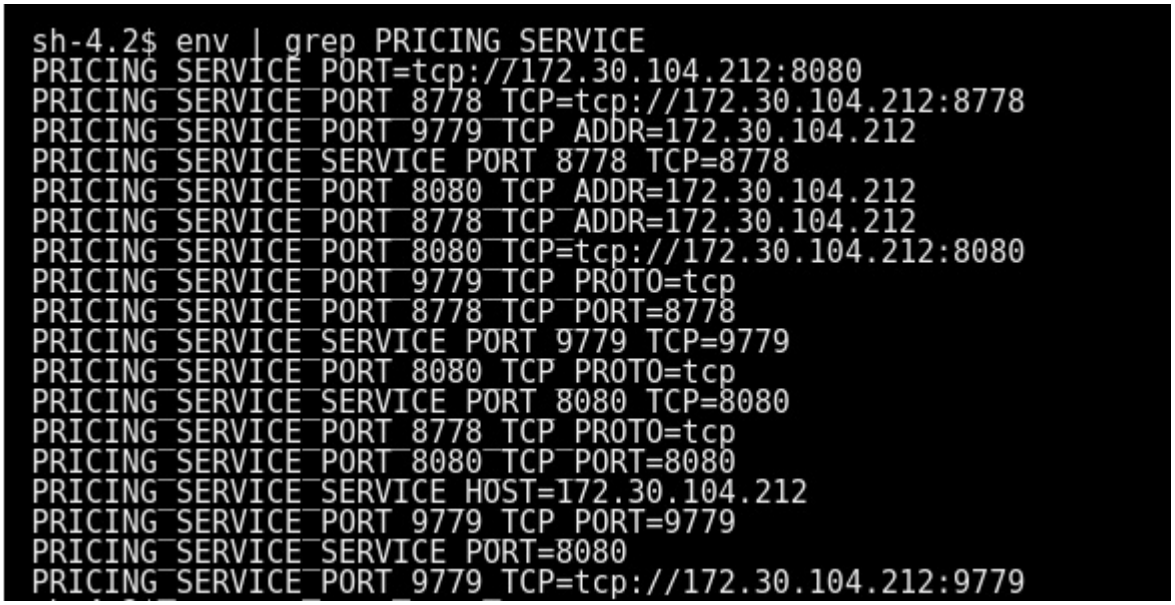
```
            views.add(view);
        }
        return views;
    }

}
```

The getItems method implementation (1) is pretty straightforward. We are combining data from catalog and pricing services and returning the list of resulting object. The most interesting part here is the proxies which enable us to communicate with those services (2). Let's learn how to implement them.

Environment variables

When the new service is created, its coordinates are written into environment variables in every pod in the cluster.

Let's log in to one of the pods inside the cluster and take a look at it. All the OpenShift environment variable names are written in uppercase, and we need the data about the pricing service:



In the preceding screenshot, note that there are a number of variables describing the coordinates of the service. The property that interests us is the host address:

```
PRICING_SERVICE_SERVICE_HOST=172.30.104.212
```

Note that this is the virtual cluster IP again. As a result, as long as the service is not removed, the proxy address will stay the same. Underlying infrastructure changes caused by deployments, node addition, or failures will not result in the change of the previous address.

Let's write proxies that will use this variable in order to connect to the services. We will start with the pricing-service proxy:

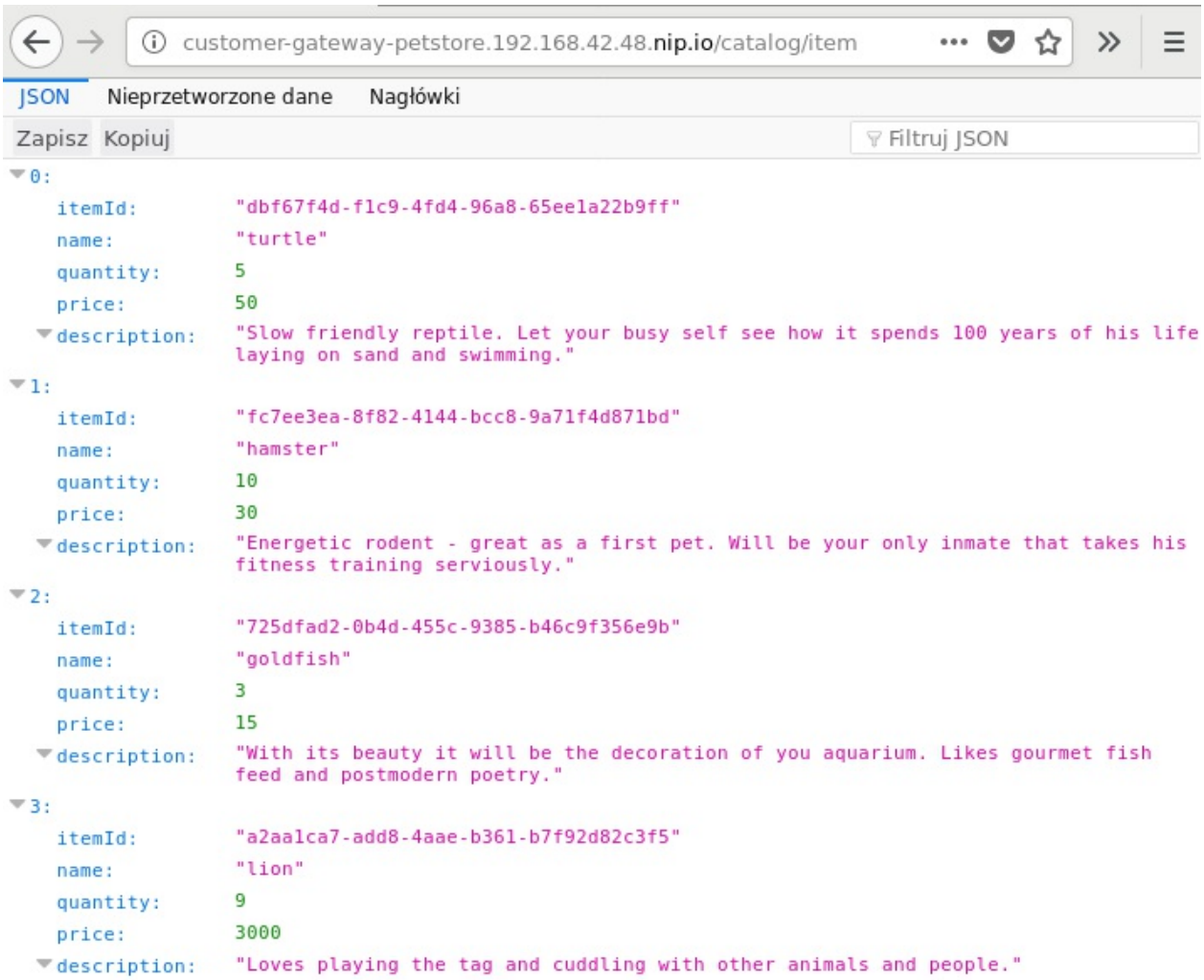
```
package org.packt.swarm.petstore.proxy;

importorg.packt.swarm.petstore.pricing.api.Price;

import javax.enterprise.context.ApplicationScoped;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;

@ApplicationScoped
public class PricingProxy {

private String targetPath;

PricingProxy(){
        //1
        targetPath = "http://" + System.getenv("PRICING_SERVICE_SERVICE_HOST")+":"+8080;
}

public PricegetPrice(String name){
        //2
        Client client = ClientBuilder.newClient();
        WebTarget target = client.target(targetPath +"/price/" + name);
        return target.request(MediaType.APPLICATION_JSON).get(Price.class);
}
}
```

That's just it. We obtained the `clusterIP` of the pricing-service when the proxy was being created (1) and the user straightforward REST Client API to provide an adapter for the `getPrice` method invocation (2).

The implementation of `catalogProxy` is analogous.

```
oc new-app wildflyswarm-10-centos7~https://github.com/PacktPublishing/Hands-On-Cloud-Development-with-WildFly.git --contex
```

Now we are ready to check whether our application is working. Let's create a route for the `petstore` service and check the web browser:



`customer-gateway-petstore.<update-me>-80-<update-me>.environments.katacoda.com/catalog/item`

It works indeed. This solution has a major disadvantage though—an ordering problem. If the pod is created before the service, then service coordinates won't be present in the pod environment. Is there a better way to discover the services, then? Yes, through **Domain Name Service** (**DNS**).

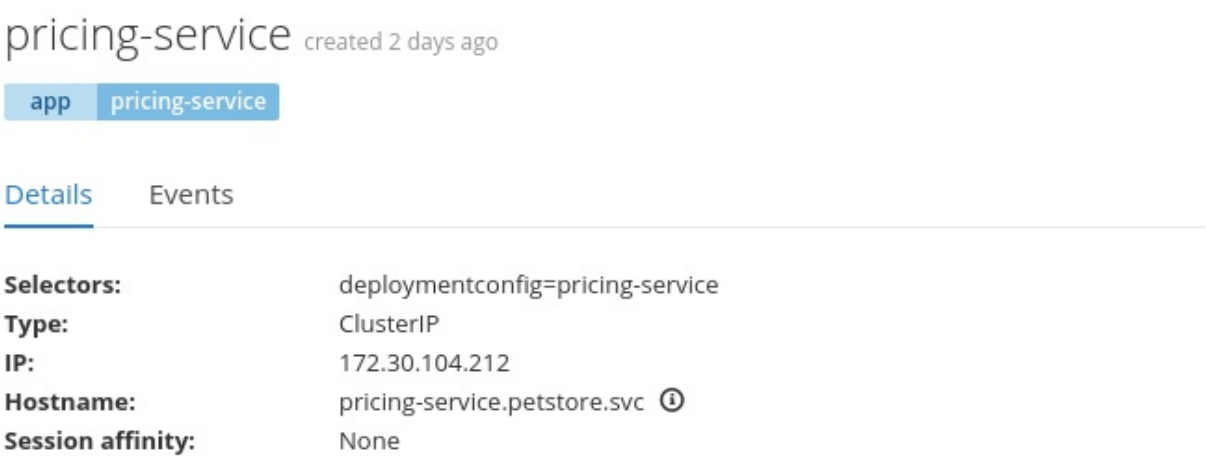
DNS discovery

Each OpenShift cluster contains a DNS service. This service allows you to discover services easily using the service name. Each service registers to the DNS service during the registration, and later periodically sends live messages to it. The DNS server creates a record using the following pattern:

```
${service name}.${application name}.svc
```

Let's take the pricing service as an example. We have created the `petstore` application. As a result, the name of the service created using the preceding pattern would be pricing-`service.petstore.svc`.

We can confirm that information inside web console. Let's navigate to `Applications ` | `Services ` | `pricing-service`:



Take note of the `hostname` field—this is the address that we created previously. Another important thing to note is that those service names are visible only from inside the cluster.

We are now ready to refactor our application to use elegant DNS discovery.

Note

Examples reference: `chapter8/customer-gateway-dns`.

We have to rewrite both our proxies. Let's start with `PricingProxy`:

```
package org.packt.swarm.petstore.proxy;

import org.packt.swarm.petstore.pricing.api.Price;

import javax.enterprise.context.ApplicationScoped;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
```

```
import javax.ws.rs.core.MediaType;

@ApplicationScoped
public class PricingProxy {

    //1
private final String targetPath = System.getProperty("proxy.pricing.url");

    public Price getPrice(String itemId){
        Client client = ClientBuilder.newClient();
WebTarget target = client.target(targetPath + "/price/" + itemId);
        return target.request(MediaType.APPLICATION_JSON).get(Price.class);
}
}
```

We defined a `targetPath` that we can use repeatedly to connect to services (1). We are going to provide it as a parameter using YAML configuration:

```
proxy:
  catalog:
    url: "http://catalog-service.petstore.svc:8080"
pricing:
    url: "http://pricing-service.petstore.svc:8080"
```

Again, `CatalogProxy` implementation is analogous.

Now we are ready to redeploy the customer-gateway service again. You can once again check whether it works correctly.

`oc delete all -l app=customer-gateway`

`oc new-app wildflyswarm-10-centos7~https://github.com/PacktPublishing/Hands-On-Cloud-Development-with-WildFly.git --context-dir=chapter8/customer-gateway-dns --name=customer-gateway`

`oc expose svc/customer-gateway`

`curl -X GET customer-gateway-petstore.<update-me>-80-<update-me>.environments.katacoda.com/catalog/item`

As you may recall, we were using the name of the service when we were creating the environment file for our databases. Each service in the cluster can be reached using this method.