**Lab02-kubernetes-whiskeyshop-mysql**

In this workshop we are going to use a Minikube environment with a Spring Application with an embedded H2 database and after that we are going to connect the Spring Application with an external dockerized MySQL database.

## Prerequisites

- Have a Kubernetes cluster up and running (for example Minikube)

This tutorial makes use of:

- IntelliJ IDE

- Minikube

## Let’s get started

The application we are going to use in this tutorial is the whiskeyshop application. This application can be retrieved from GitHub <https://github.com/AMIS-Services/sig-springboot-1>

See directory **lab7-docker**.

In the previous **lab01-docker** of sig-kubernetes/SIG01 you have already created a Dockerfile (as described in “SpringDockerWorkShop.docx”).

For this lab, use the already existing **whiskeyshop** directory or copy the complete **whiskeyshop** directory to an **applications** subdirectory of the **lab02-kubernetes** environment. For example:

C:\My\AMIS\Opleiding\2019SIG1\omgeving\applications\whiskeyshop

Open …\applications\whiskeyshop in IntelliJ (via Import Project and selecting the pom.xml file) or your favorite editor.

## Modifying the whiskeyshop application

Edit file …\applications\whiskeyshop\src\main\resources\**application-mysql.properties**, so that in the end it looks like:

spring.datasource.url=jdbc:mysql://**localhost**:3306/test?allowPublicKeyRetrieval=true&useSSL=false  
spring.datasource.username=root  
spring.datasource.password=**password**  
spring.jpa.database-platform=org.hibernate.dialect.MySQL5InnoDBDialect  
  
spring.jpa.hibernate.ddl-auto=create

For logging purposes in Minikube, we edit file …\applications\whiskeyshop\src\main\java\com\amis\whiskeyshop\**WhiskeyshopApplication.java**, so that in the end it looks like:

package com.amis.whiskeyshop;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.CommandLineRunner;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.core.env.Environment;  
  
import javax.sql.DataSource;  
  
@SpringBootApplication  
public class WhiskeyshopApplication **implements CommandLineRunner** {  
  
 **@Autowired  
 private Environment environment;  
  
 @Autowired  
 DataSource dataSource;  
  
 private static String *serverPort*;  
  
 @Override  
 public void run(String... args) throws Exception {  
 *serverPort* = System.*getProperty*("server.port");  
 System.*out*.println("\n----main----");  
  
 System.*out*.println("----System Properties from VM Arguments----");  
 System.*out*.println("server.port: " + *serverPort*);  
 System.*out*.println("----Program Arguments----");  
 for (String arg: args) {  
 System.*out*.println(arg);  
 }  
  
 if (environment != null) {  
 getActiveProfiles();  
 System.*out*.println("----Environment Properties----");  
 System.*out*.println("spring.datasource.url: " + environment.getProperty("spring.datasource.url"));  
 System.*out*.println("spring.datasource.username: " + environment.getProperty("spring.datasource.username"));  
 System.*out*.println("spring.datasource.password: " + environment.getProperty("spring.datasource.password"));  
 System.*out*.println("spring.jpa.database-platform: " + environment.getProperty("spring.jpa.database-platform"));  
 System.*out*.println("spring.jpa.hibernate.ddl-auto: " + environment.getProperty("spring.jpa.hibernate.ddl-auto"));  
 }  
  
 if (dataSource != null) {  
 System.*out*.println("dataSource: " + dataSource);  
 }  
 System.*out*.println("------------");  
 }**  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(WhiskeyshopApplication.class, args);  
 }  
  
 **private void getActiveProfiles() {  
 for (final String profileName : environment.getActiveProfiles()) {  
 System.*out*.println("Currently active profile - " + profileName);  
 }  
 }**  
}

Remark:

To keep things simple, in this lab we use **System.*out*.println** for logging.

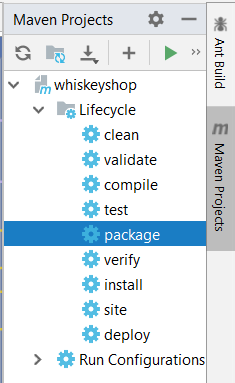
Build the application, with either of the following commands:

mvn clean package

./mvnw clean package

./mvnw.cmd clean package

Or from IntelliJ, via Maven Projects | package | Run ‘whiskeyshop [package]’.



This creates a jar file:

…\applications\whiskeyshop\target\**whiskeyshop-0.0.1-SNAPSHOT.jar**

**Building a Docker Image**

Now that we have a new jar file, let’s build a Docker Image for the application.

As described in “SpringDockerWorkShop.docx” we have a **Dockerfile** with the following contents:

# Start with a base image containing Java runtime

FROM openjdk:8-jdk-alpine

# Add Maintainer Info

LABEL maintainer="amis"

# Application runs on port 8080

EXPOSE 8080

# The application's jar file

ARG JAR\_FILE=target/whiskeyshop-0.0.1-SNAPSHOT.jar

# Add the application's jar to the container

ADD ${JAR\_FILE} whiskeyshop.jar

# Run the jar file

ENTRYPOINT ["java","-jar","/whiskeyshop.jar"]

Navigate to the directory which contains the **Dockerfile.**

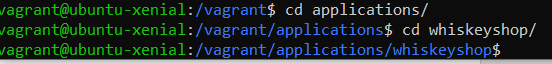
If you are using the Minikube VirtualBox Appliance, go to the directory with the Vagrantfile and scripts subdirectory and open a Windows Command Prompt (cmd) to access linux (within the VirtualBox Appliance) via ssh.

Navigate to the VirtualBox Appliance Shared Folder:

cd /vagrant



Navigate to the whiskeyshop directory:



Execute the following command:

docker build -t whiskeyshop:v1.0 .

Sending build context to Docker daemon 38.01MB

Step 1/6 : FROM openjdk:8-jdk-alpine

8-jdk-alpine: Pulling from library/openjdk

cd784148e348: Pull complete

35920a071f91: Pull complete

88b05767cad1: Pull complete

Digest: sha256:4745e7b0cc551f68592f75a524f4479e878cb726ffc851dfad3e2c9280828b01

Status: Downloaded newer image for openjdk:8-jdk-alpine

---> 2cfb1dc1f0c8

Step 2/6 : LABEL maintainer="amis"

---> Running in 9957507c972a

Removing intermediate container 9957507c972a

---> 330acb62a76a

Step 3/6 : EXPOSE 8080

---> Running in d01ee1898809

Removing intermediate container d01ee1898809

---> d02d5269eb9a

Step 4/6 : ARG JAR\_FILE=target/whiskeyshop-0.0.1-SNAPSHOT.jar

---> Running in 7fea7eed7cae

Removing intermediate container 7fea7eed7cae

---> c6d408cdb2c3

Step 5/6 : ADD ${JAR\_FILE} whiskeyshop.jar

---> 87376c55d6e8

Step 6/6 : ENTRYPOINT ["java","-jar","/whiskeyshop.jar"]

---> Running in c57f1583d0f6

Removing intermediate container c57f1583d0f6

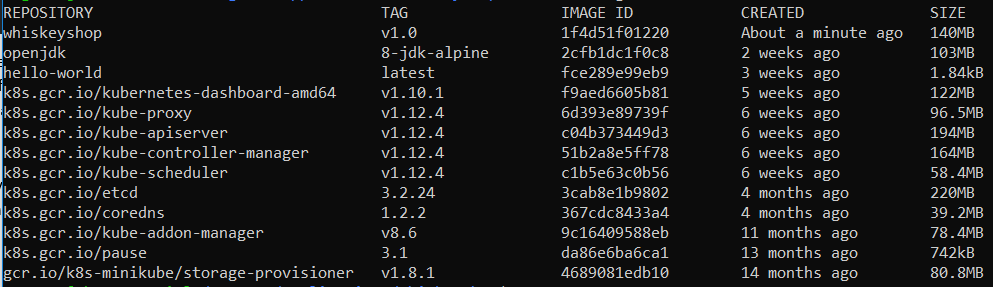
---> 1f4d51f01220

Successfully built 1f4d51f01220

Successfully tagged whiskeyshop:v1.0

To list all the docker images on your system, execute the following:

docker image ls



## Use kubectl to create a container, based on a Docker Image

Instead of using **docker run**, we are now going to use **kubectl**.

Create a **yaml** subdirectory of the **lab02-kubernetes** environment. For example:

C:\My\AMIS\Opleiding\2019SIG1\omgeving\yaml

If you are using the Minikube VirtualBox Appliance, go to the directory with the Vagrantfile and scripts subdirectory and open a Windows Command Prompt (cmd) to access linux (within the VirtualBox Appliance) via ssh.

Navigate to the yaml directory:



Within Minikube we are going to create resources with a specific namespace. So first we have to create these namespaces.

### Namespaces

Add to the **yaml** directory a file **namespace-development.yaml** with the following contents:

apiVersion: v1

kind: Namespace

metadata:

name: "nl-amis-development"

labels:

name: "nl-amis-development"

Add to the **yaml** directory a file **namespace-production.yaml** with the following contents:

apiVersion: v1

kind: Namespace

metadata:

name: "nl-amis-production"

labels:

name: "nl-amis-production"

Create the namespaces:

kubectl create -f namespace-development.yaml

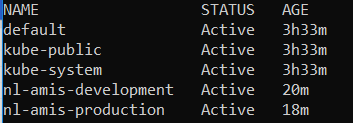
namespace/nl-amis-development created

kubectl create -f namespace-production.yaml

namespace/nl-amis-production created

You can list the current namespaces in a cluster using:

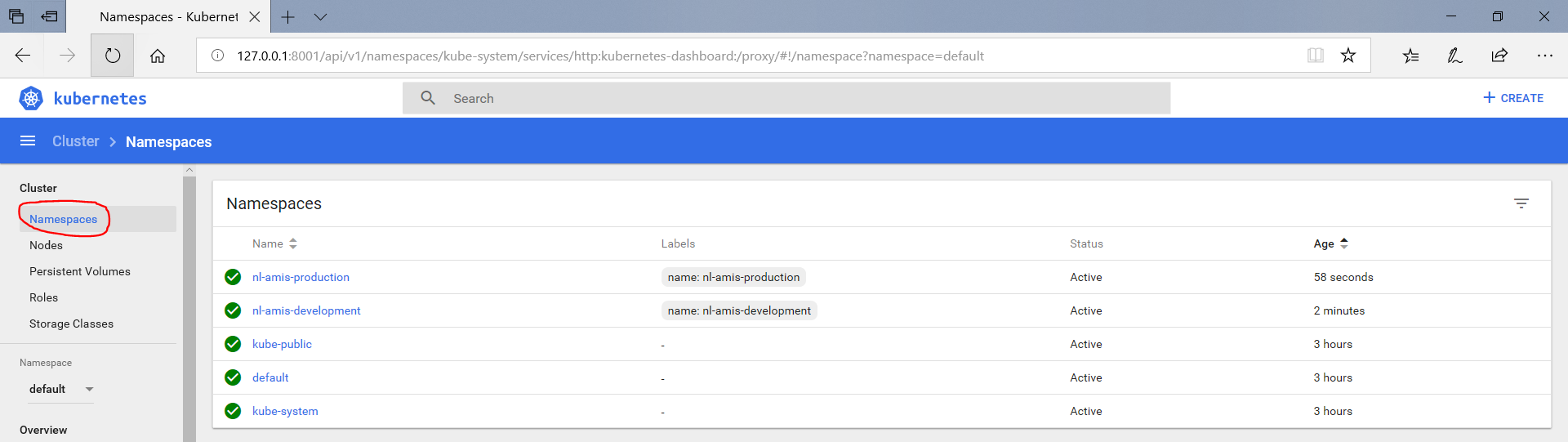
kubectl get namespaces



Or via the Kubernetes Web UI (Dashboard):

[http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/http:kubernetes-dashboard:/proxy/#!/node?namespace=default](http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/http:kubernetes-dashboard:/proxy/" \l "!/node?namespace=default)

Navigate to Cluster | Namespaces:



Remark:

In this lab the resources that we are going to create will have the Namespace: **nl-amis-development**

In a yaml file this can be done via:

**namespace: nl-amis-development**

### Deployments

Add to the **yaml** directory a file **deployment-whiskeyshop.yaml** with the following contents:

apiVersion: apps/v1

kind: Deployment

metadata:

name: whiskeyshop

namespace: nl-amis-development

labels:

app: whiskeyshop

version: "1.0"

environment: development

spec:

replicas: 2

selector:

matchLabels:

app: whiskeyshop

version: "1.0"

environment: development

template:

metadata:

labels:

app: whiskeyshop

version: "1.0"

environment: development

spec:

containers:

- name: whiskeyshop-container

image: whiskeyshop:v1.0

ports:

- containerPort: 8080

Remark:

This deployment uses the whiskeyshop (Spring Application) with H2 as an embedded in-memory database.

The replicas are set to 2, so the ReplicaSet ensures that 2 pod replicas are running at any given time.

The ReplicaSet manages all the pods with labels that match the selector. In our case these labels are:

|  |  |
| --- | --- |
| **Label key** | **Label value** |
| app | whiskeyshop |
| version | 1.0 |
| environment | development |

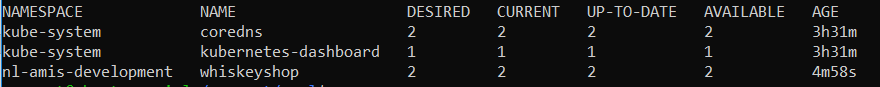
Create the deployment:

kubectl create -f deployment-whiskeyshop.yaml

deployment.apps/whiskeyshop created

You can list the deployments in a cluster using:

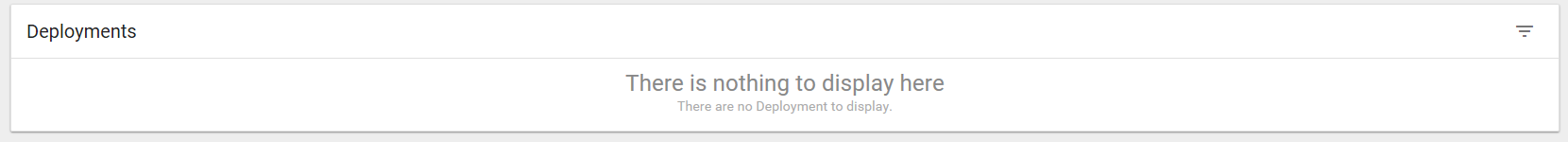
kubectl get deployments --all-namespaces



Or via the Kubernetes Web UI (Dashboard):

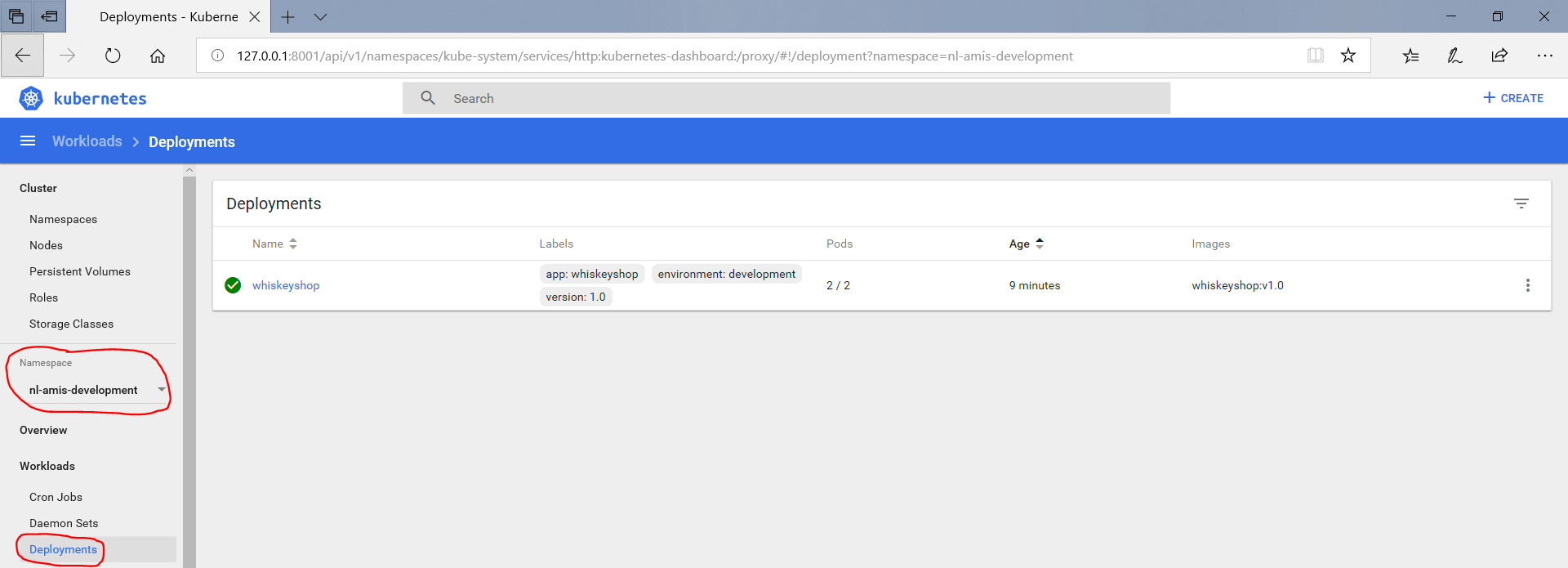
[http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/http:kubernetes-dashboard:/proxy/#!/node?namespace=default](http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/http:kubernetes-dashboard:/proxy/" \l "!/node?namespace=default)

Navigate to Workloads | Deployments:

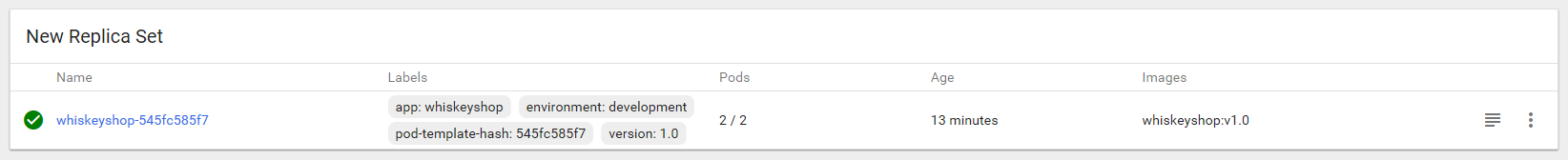


Remember, in Minikube the Namespace that is used is: default

So we first have to switch to our namespace (via Namespace | nl-amis-development) and then navigate to Workloads | Deployments:

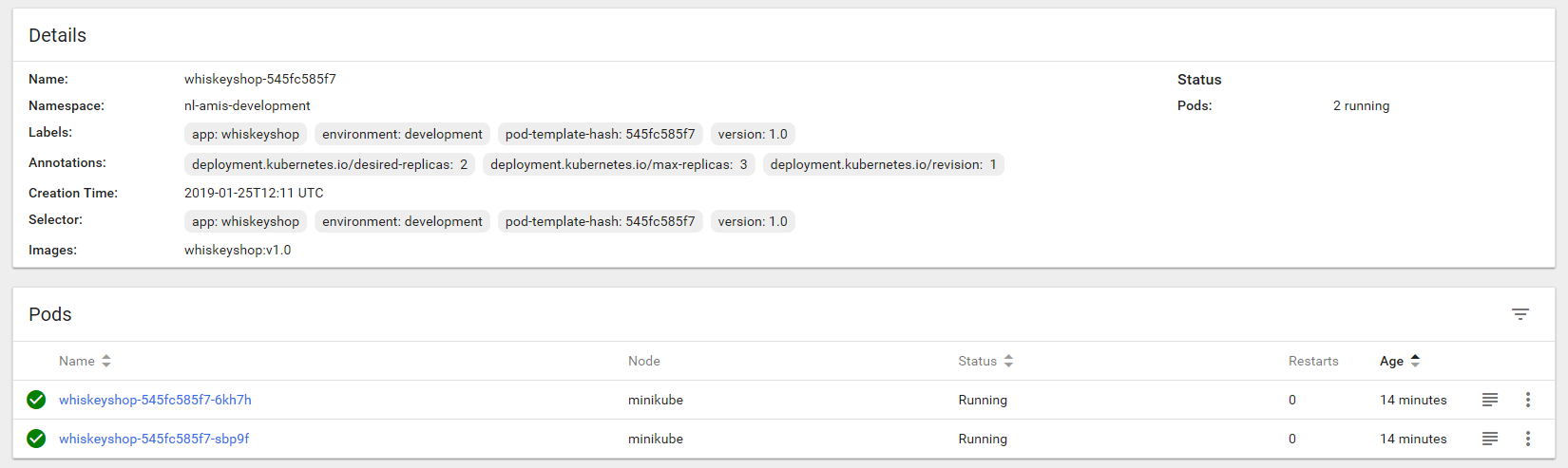


The Deployment we have created contains a Replica Set:



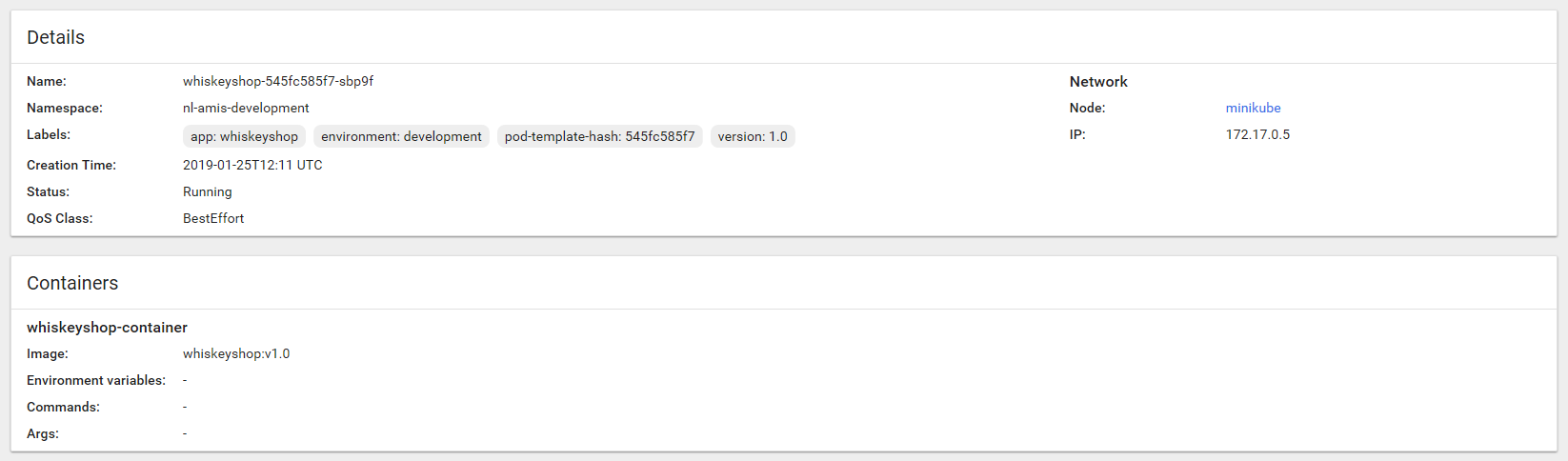
### Replica Sets

The Replica Set contains two Pods:

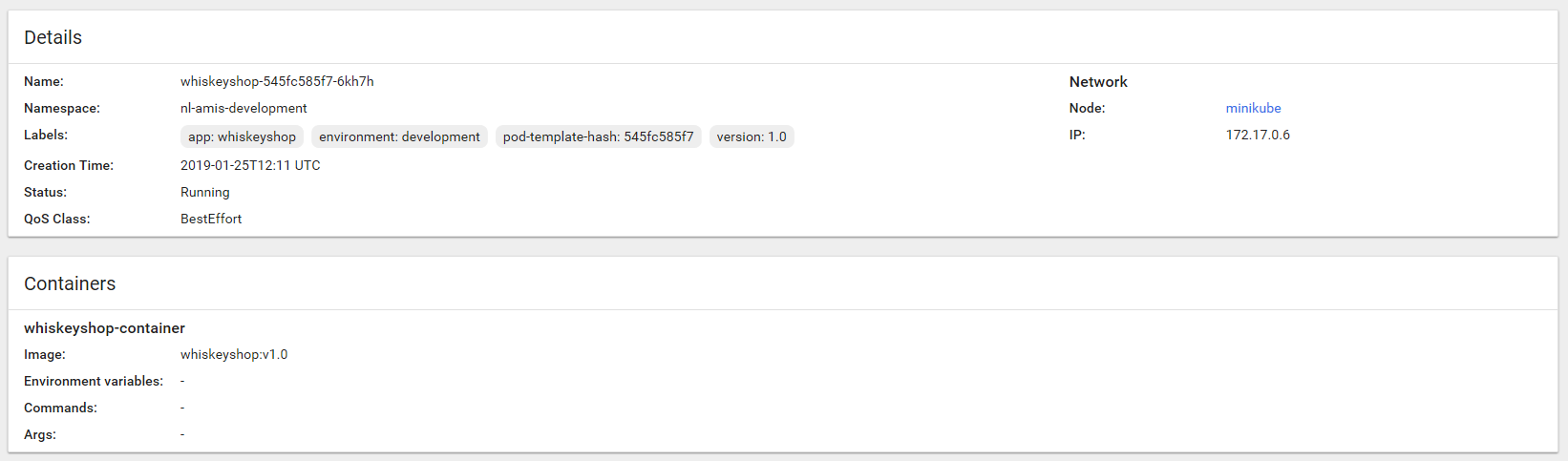


### Pods

The first Pod includes 1 container (based on image whiskeyshop:v1.0) and has IP address 172.17.0.5:

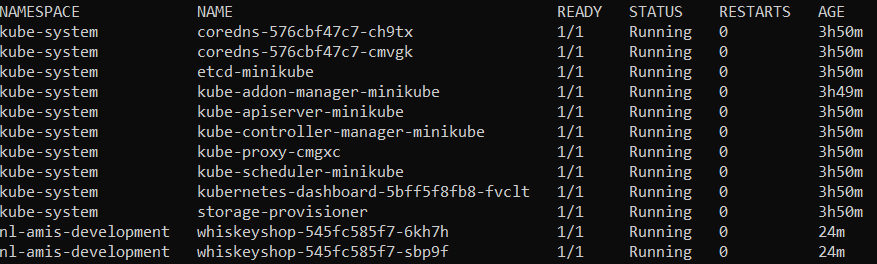


The second Pod includes 1 container (based on image whiskeyshop:v1.0) and has IP address 172.17.0.6:



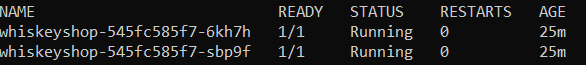
You can list the pods in a cluster using:

kubectl get pods --all-namespaces



You can list the pods in a certain namespace (of a cluster) using:

kubectl --namespace=nl-amis-development get pods



### Services

Add to the **yaml** directory a file **service-whiskeyshop.yaml** with the following contents:

kind: Service

apiVersion: v1

metadata:

name: whiskeyshop-service

namespace: nl-amis-development

labels:

app: whiskeyshop

version: "1.0"

environment: development

spec:

selector:

app: whiskeyshop

version: "1.0"

environment: development

type: NodePort

ports:

- protocol: TCP

nodePort: 31580

port: 8180

targetPort: 8080

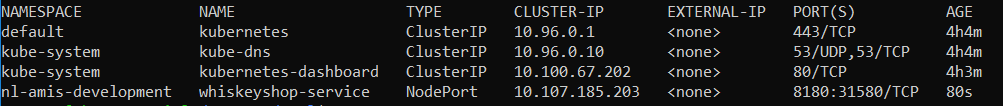
Create the service:

kubectl create -f service-whiskeyshop.yaml

service/whiskeyshop-service created

You can list the services in a cluster using:

kubectl get services --all-namespaces



You can list the services in a certain namespace (of a cluster) using:

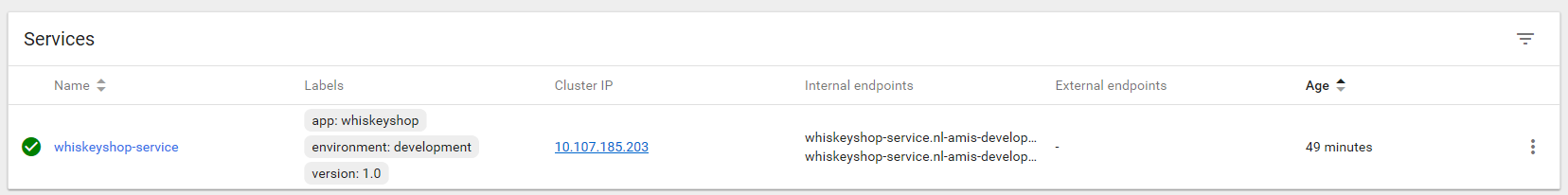
kubectl --namespace=nl-amis-development get services



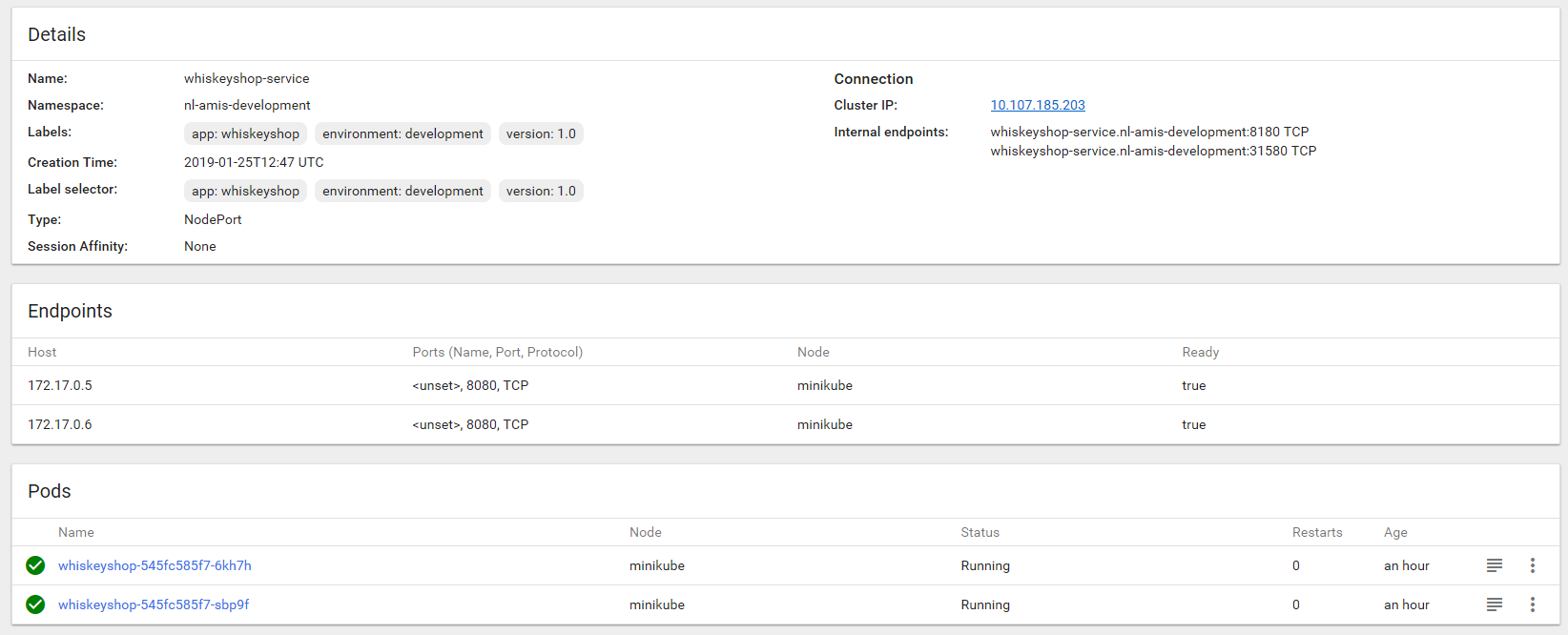
Or via the Kubernetes Web UI (Dashboard):

[http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/http:kubernetes-dashboard:/proxy/#!/node?namespace=default](http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/http:kubernetes-dashboard:/proxy/" \l "!/node?namespace=default)

Navigate to Discovery and Load Balancing | Services:



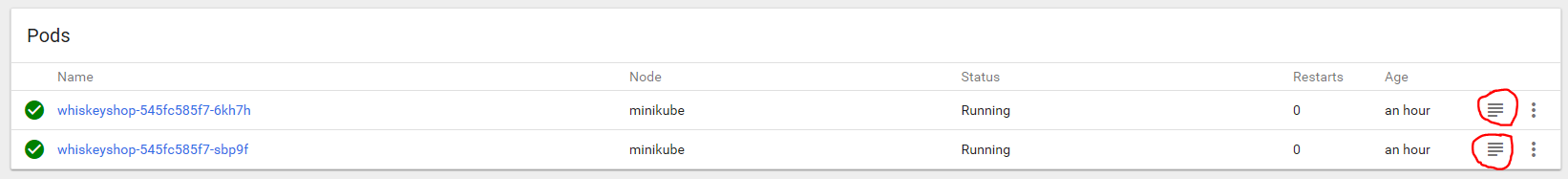
Click on the service.



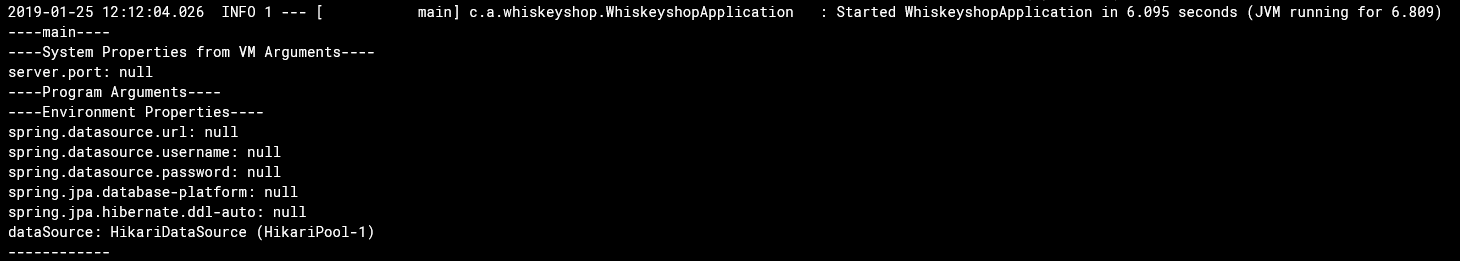
Via the Details you can see the following:

* The Cluster IP address of this service is: 10.107.185.203
* The Internal endpoints of this service are:
* whiskeyshop-service.nl-amis-development:8180 TCP
* whiskeyshop-service.nl-amis-development:31580 TCP
* The Endpoints of the Pods/Containers are:
* 172.17.0.5:8080 TCP
* 172.17.0.6:8080 TCP

If we look at the Logs of the Pods:

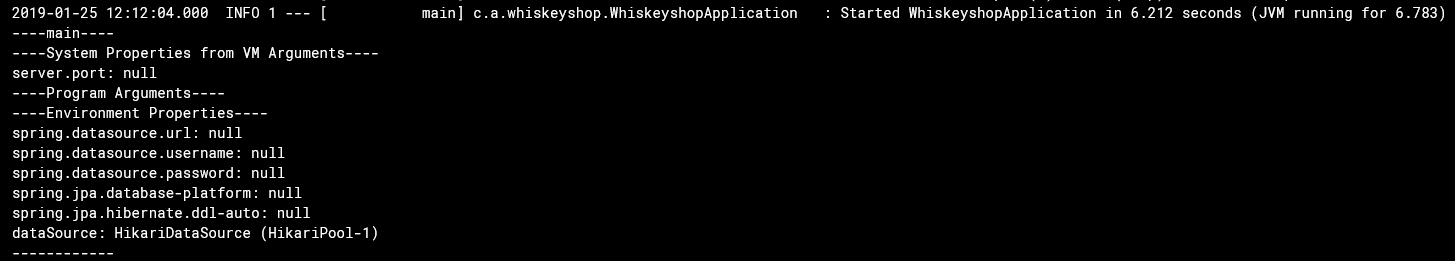


Logs from whiskeyshop-container, in whiskeyshop-545fc585f7-6kh7h:



Here we see for the first time, the results of editing **WhiskeyshopApplication.java** at the beginning of this lab.

Logs from whiskeyshop-container, in whiskeyshop-545fc585f7-sbp9f:



## Calling the whiskeyshop application

Alright, so now your application should be running two Docker containers. Let’s test if the application also responds by firing some requests.

First, we add some whiskey-products with POST-requests to the whiskeyshop application:

curl --header "Content-Type: application/json" --request POST --data '{"id": 1, "name": "**Jack Daniels Tennessee**"}' http://10.107.185.203:8180/whiskeys

curl --header "Content-Type: application/json" --request POST --data '{"id": 2, "name": "**Crown Royal Canadia**"}' http://10.107.185.203:8180/whiskeys

curl --header "Content-Type: application/json" --request POST --data '{"id": 3, "name": "**Jim Beam bourbon**"}' http://10.107.185.203:8180/whiskeys

As you can see, we are using the Cluster IP address of the service.

Next, you should be able to see your updated list at the /whiskeys path with a GET-request. First we try the service IP address:

curl http://10.107.185.203:8180/whiskeys

[{"id":"1","name":"**Jack Daniels Tennessee**"}]

We only see one result.

So let’s try, the first Pod IP address:

curl http://10.107.185.203:8180/whiskeys

[{"id":"1","name":"**Jack Daniels Tennessee**"}]

Again we only see one result (id:1).

And then let’s try, the second Pod IP address:

curl http://172.17.0.6:8080/whiskeys

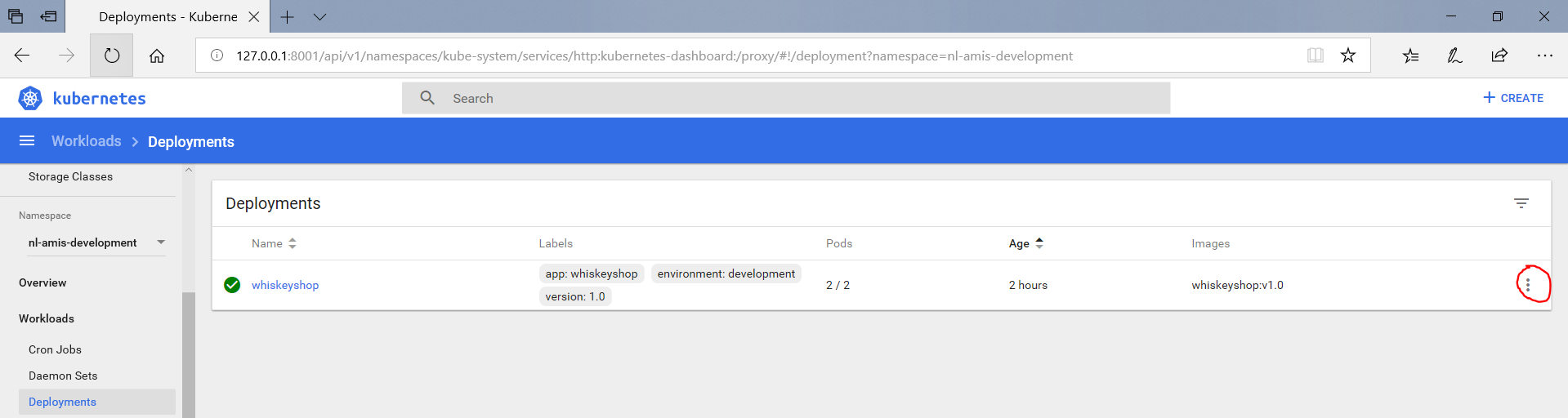
[{"id":"2","name":"**Crown Royal Canadia**"},{"id":"3","name":"**Jim Beam bourbon**"}]

Here we see the other two results (id:2 and id:3).

Remember, this deployment uses the whiskeyshop (Spring Application) with H2 as an embedded in-memory database. So the first and second Pod each have their own embedded in-memory database. The service obviously sent the first POST-request (id:1) to the first Pod and the second (id:2) and third (id:3) POST-requests to the second Pod.

## Running a MySQL database in a separate Docker container

Up til now we used the whiskeyshop (Spring Application) with H2 as an embedded in-memory database. So now let’s focus on using the whiskeyshop with an external MySQL database, running in a separate Docker container.

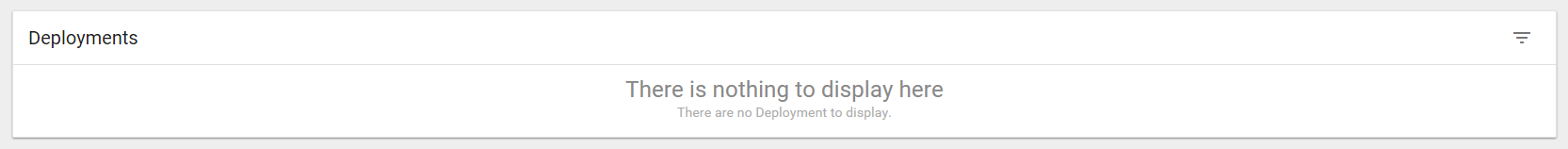


Click on DELETE.



And then confirm the delete action by a click on the DELETE button.

Then after some time the Deployment is deleted.



### PersistentVolume

Add to the **yaml** directory a file **persistent-volume-mysql.yaml** with the following contents:

kind: PersistentVolume

apiVersion: v1

metadata:

name: mysql-pv-volume

namespace: nl-amis-development

labels:

type: local

spec:

storageClassName: manual

capacity:

storage: 1Gi

accessModes:

- ReadWriteOnce

hostPath:

path: "/mnt/data"

Create the PersistentVolume:

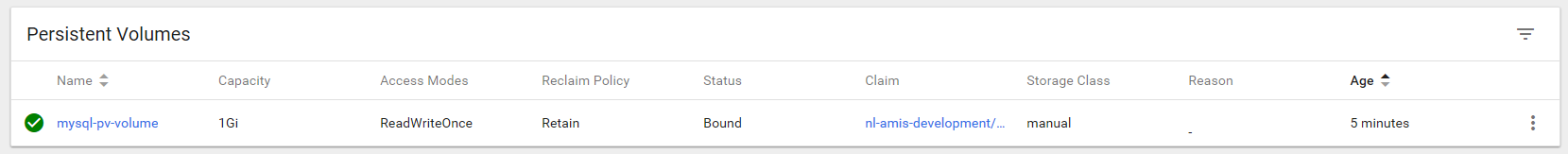
kubectl create -f persistent-volume-mysql.yaml

persistentvolume/mysql-pv-volume created

Via the Kubernetes Web UI (Dashboard):

[http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/http:kubernetes-dashboard:/proxy/#!/node?namespace=default](http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/http:kubernetes-dashboard:/proxy/" \l "!/node?namespace=default)

Navigate to Cluster | Persistent Volumes:



### PersistentVolumeClaim

Add to the **yaml** directory a file **persistent-volume-claim-mysql.yaml** with the following contents:

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: mysql-pv-claim

namespace: nl-amis-development

spec:

storageClassName: manual

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 1Gi

Create the PersistentVolumeClaim:

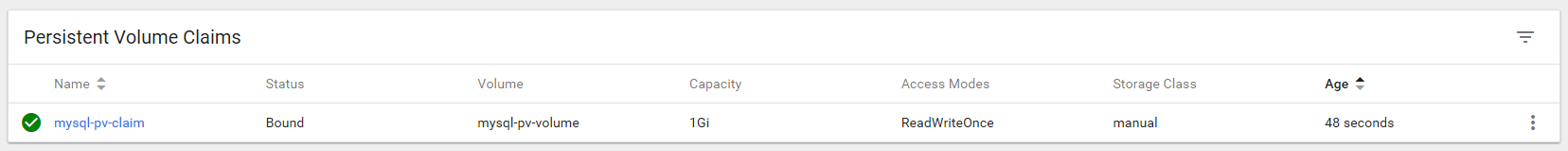
kubectl create -f persistent-volume-claim-mysql.yaml

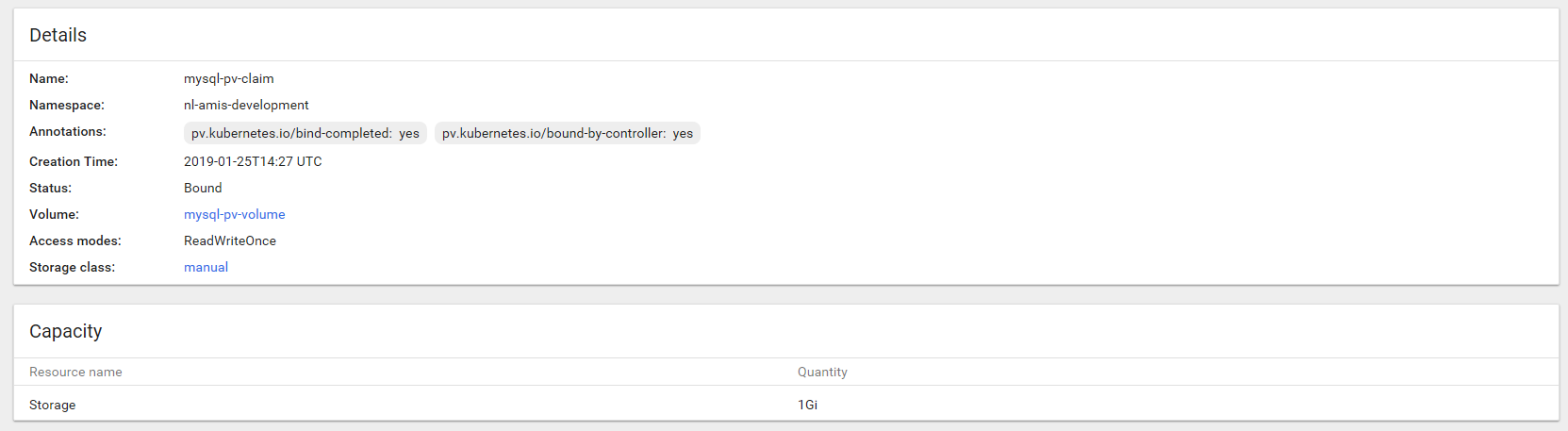
persistentvolumeclaim/mysql-pv-claim created

Via the Kubernetes Web UI (Dashboard):

[http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/http:kubernetes-dashboard:/proxy/#!/node?namespace=default](http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/http:kubernetes-dashboard:/proxy/" \l "!/node?namespace=default)

Navigate to Config and Storage | Persistent Volume Claims:





### Deployments

Add to the **yaml** directory a file **deployment-mysql.yaml** with the following contents:

apiVersion: apps/v1

kind: Deployment

metadata:

name: mysql

namespace: nl-amis-development

labels:

app: mysql

version: "1.0"

environment: development

spec:

replicas: 1

selector:

matchLabels:

app: mysql

version: "1.0"

environment: development

strategy:

type: Recreate

template:

metadata:

labels:

app: mysql

version: "1.0"

environment: development

spec:

containers:

- image: mysql:5.6

name: mysql

env:

# Use secret in real usage

- name: MYSQL\_ROOT\_PASSWORD

value: password

ports:

- containerPort: 3306

name: mysql-container

volumeMounts:

- name: mysql-persistent-storage

mountPath: /var/lib/mysql

volumes:

- name: mysql-persistent-storage

persistentVolumeClaim:

claimName: mysql-pv-claim

Create the deployment:

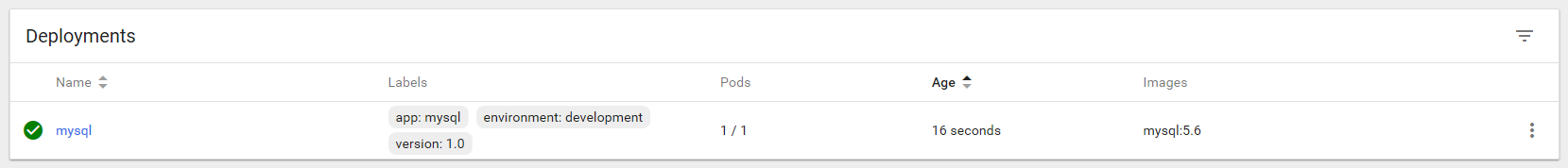
kubectl create -f deployment-mysql.yaml

deployment.apps/mysql created

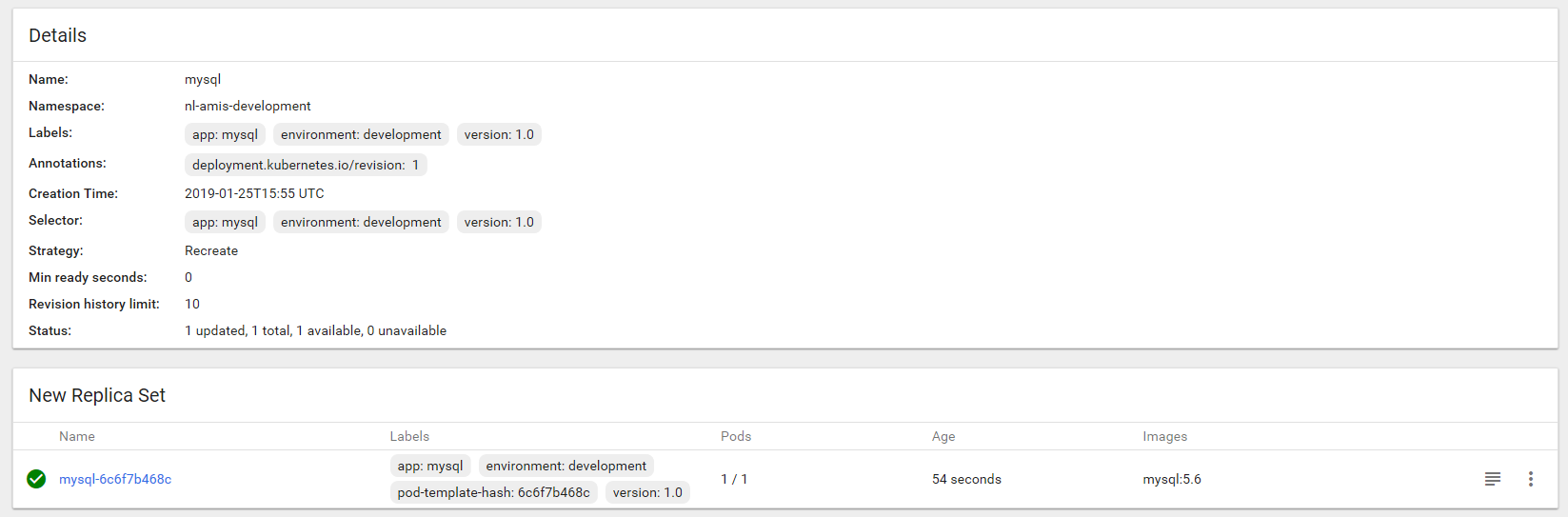
Via the Kubernetes Web UI (Dashboard):

[http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/http:kubernetes-dashboard:/proxy/#!/node?namespace=default](http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/http:kubernetes-dashboard:/proxy/" \l "!/node?namespace=default)

Navigate to Workloads | Deployments:

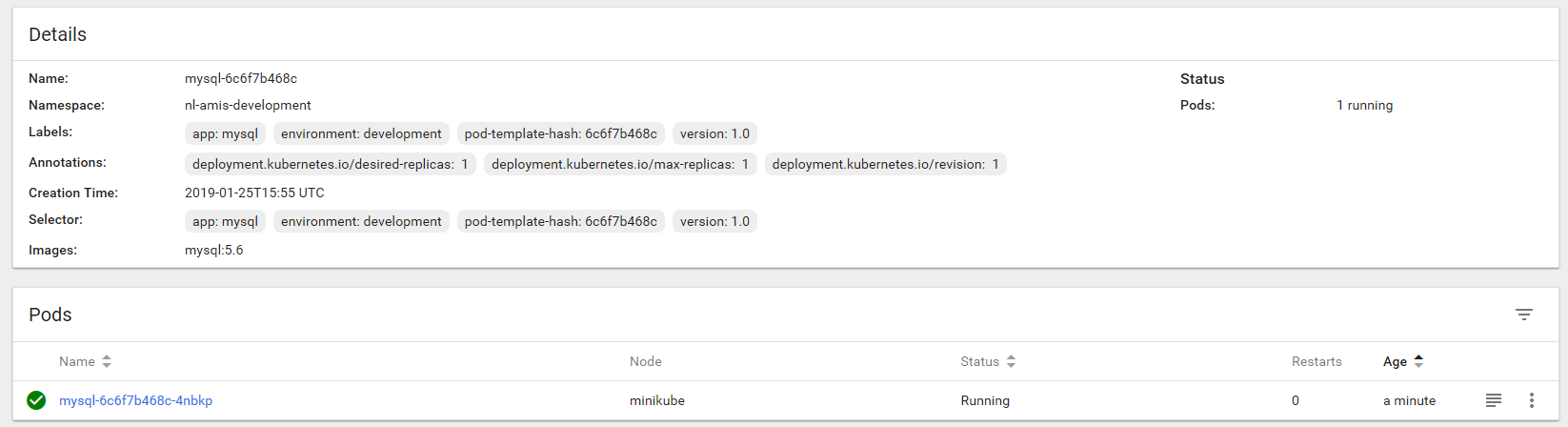


The Deployment we have created contains a Replica Set:



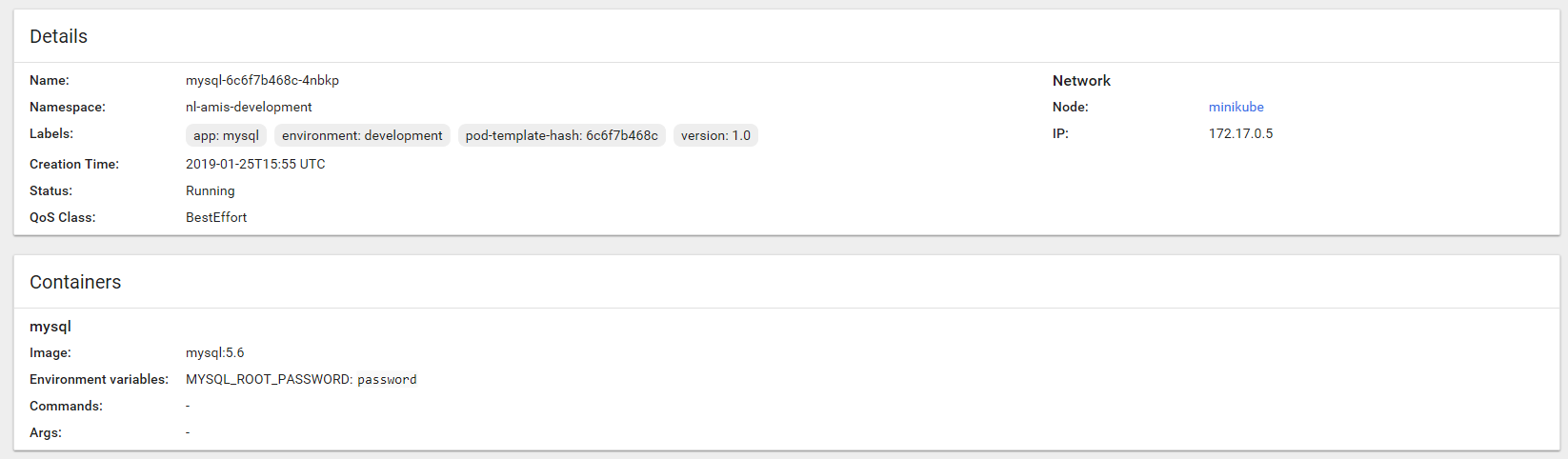
### Replica Sets

The Replica Set contains one Pod:



### Pods

The Pod includes 1 container (based on image mysql:5.6) and has IP address 172.17.0.5:



### Services

Add to the **yaml** directory a file **service-mysql.yaml** with the following contents:

kind: Service

apiVersion: v1

metadata:

name: mysql-service

namespace: nl-amis-development

labels:

app: mysql

version: "1.0"

environment: development

spec:

selector:

app: mysql

version: "1.0"

environment: development

ports:

- port: 3306

selector:

app: mysql

clusterIP: None

Create the service:

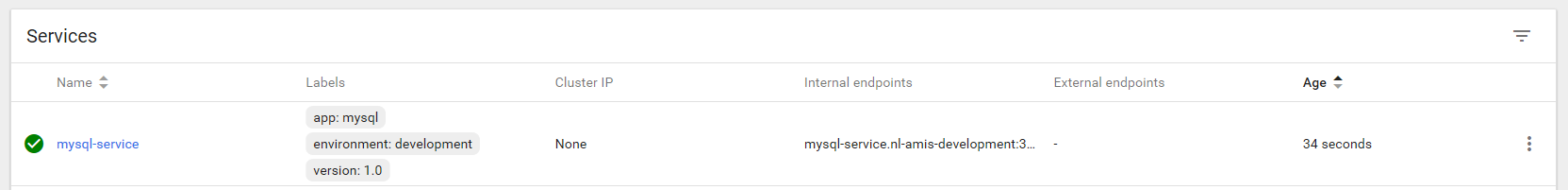
kubectl create -f service-mysql.yaml

service/mysql-service created

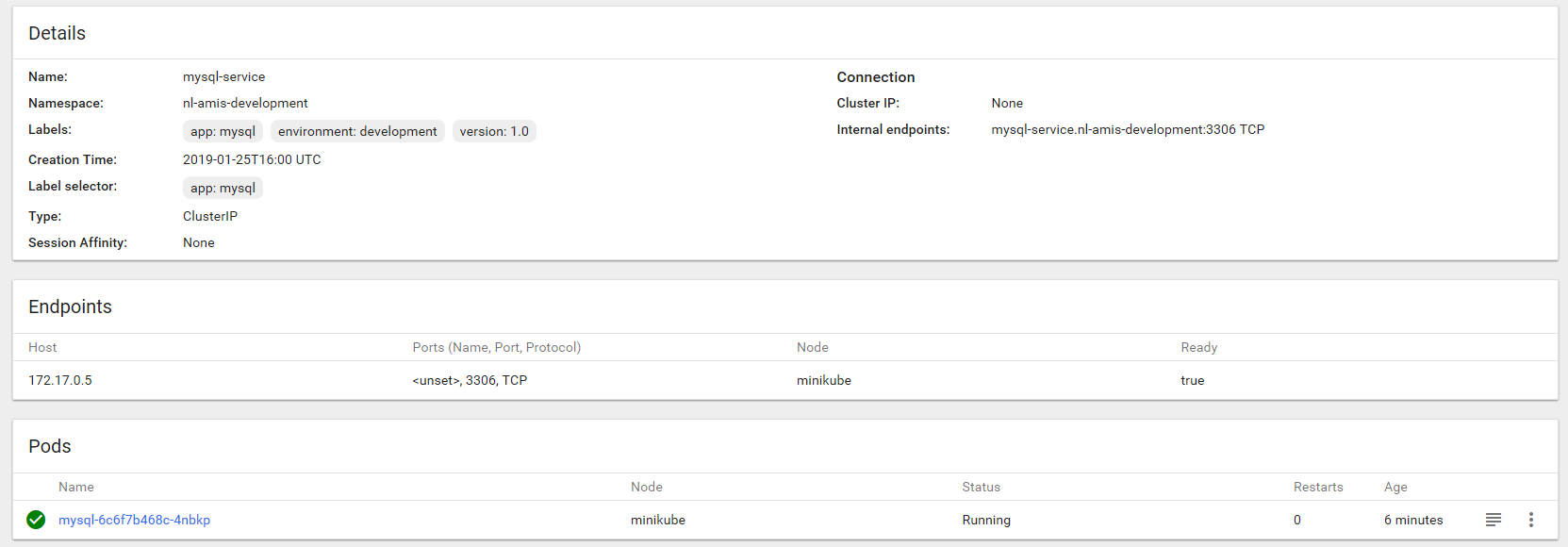
Via the Kubernetes Web UI (Dashboard):

[http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/http:kubernetes-dashboard:/proxy/#!/node?namespace=default](http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/http:kubernetes-dashboard:/proxy/" \l "!/node?namespace=default)

Navigate to Discovery and Load Balancing| Services:



Click on the service.



Via the Details you can see the following:

* The Cluster IP address of this service is: None
* The Internal endpoints of this service is:
* mysql-service.nl-amis-development:3306 TCP
* The Endpoint of the Pod/Container is:
* 172.17.0.5:3306 TCP

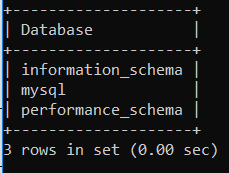
In this lab a MySQL database named **test** is used. So, we first create that database. For this we use a mysql-client.

kubectl --namespace=nl-amis-development run -it --rm --image=mysql:5.6 --restart=Never mysql-client -- mysql -h mysql-service.nl-amis-development -ppassword

If you don't see a command prompt, try pressing enter.

Probably you have to click on the Enter button.

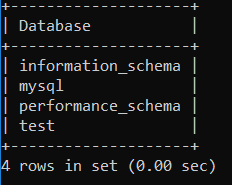
mysql> show databases;



mysql> create database test;

Query OK, 1 row affected (0.00 sec)

mysql> show databases;



mysql> exit

Bye

pod "mysql-client" deleted

## Again, we are going to use the whiskeyshop (Spring Application)

In the **yaml** directory edit the file **deployment-whiskeyshop.yaml,** so that in the end it looks like:

apiVersion: apps/v1

kind: Deployment

metadata:

name: whiskeyshop

namespace: nl-amis-development

labels:

app: whiskeyshop

version: "1.0"

environment: development

spec:

replicas: 2

selector:

matchLabels:

app: whiskeyshop

version: "1.0"

environment: development

template:

metadata:

labels:

app: whiskeyshop

version: "1.0"

environment: development

spec:

containers:

- name: whiskeyshop-container

image: whiskeyshop:v1.0

**env:**

**- name: spring.profiles.active**

**value: "mysql"**

**- name: spring.datasource.url**

**value: "jdbc:mysql://172.17.0.5:3306/test?allowPublicKeyRetrieval=true&useSSL=false"**

ports:

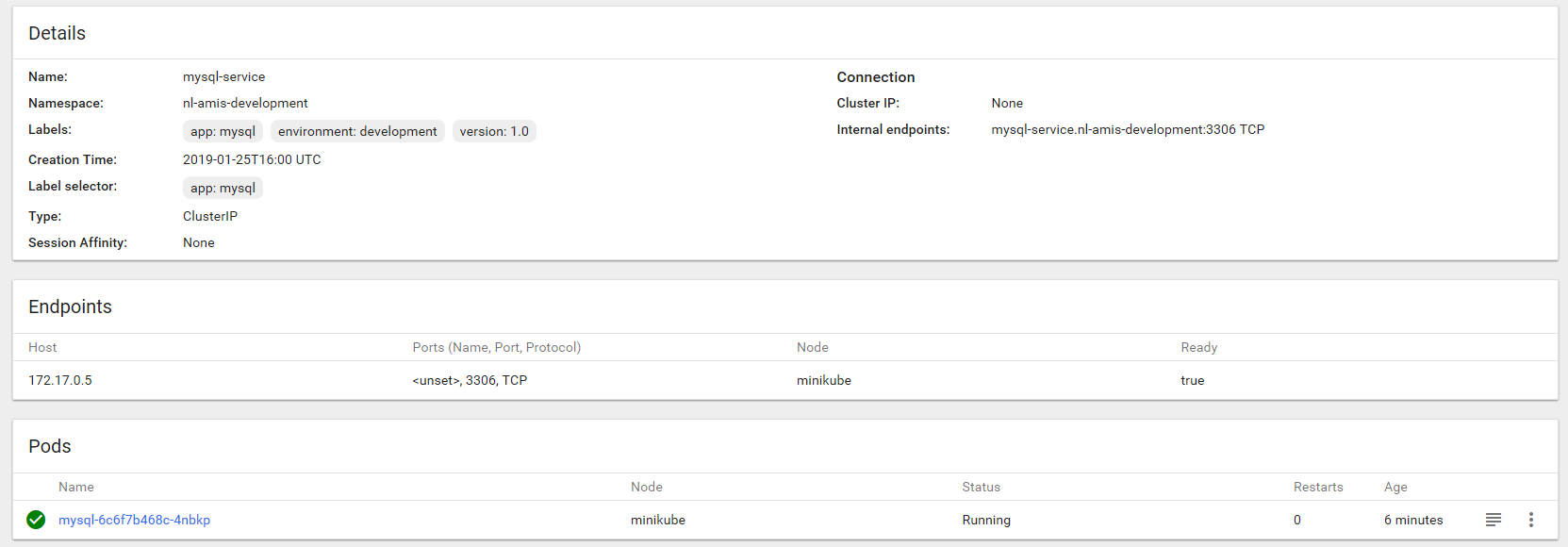
- containerPort: 8080

Remark:

This deployment uses the whiskeyshop (Spring Application) with an external MySQL database, running in a separate Docker container.

In the yaml file above be sure to use the correct IP address. In your environment this could be different then the one mentioned above (172.17.0.5).

See the details of your mysql-service for the correct IP address:



Create the deployment:

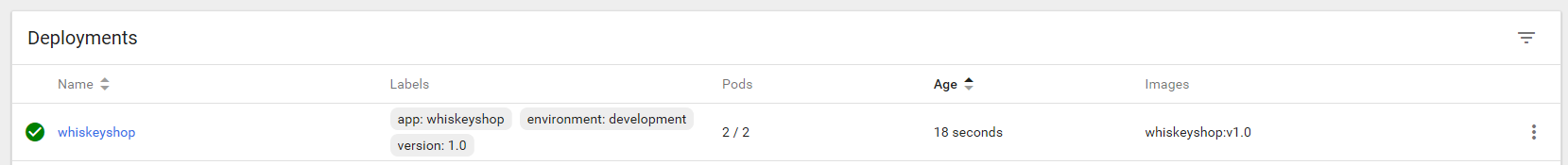
kubectl create -f deployment-whiskeyshop.yaml

deployment.apps/whiskeyshop created

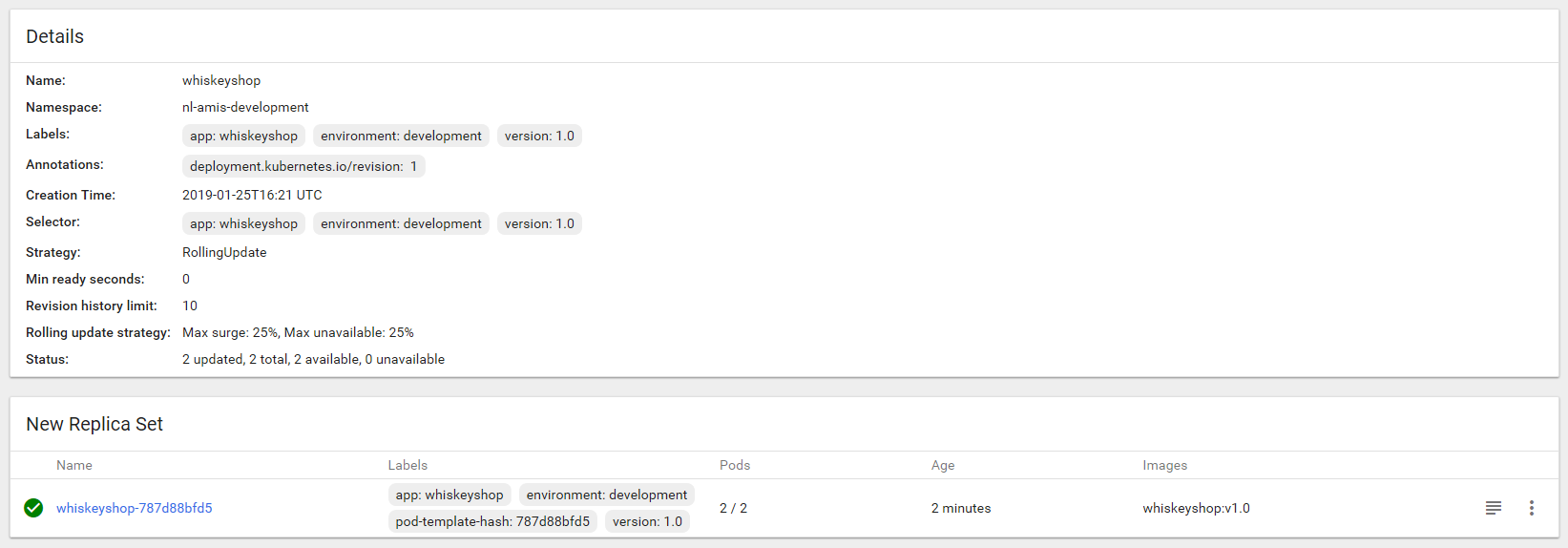
Via the Kubernetes Web UI (Dashboard):

[http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/http:kubernetes-dashboard:/proxy/#!/node?namespace=default](http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/http:kubernetes-dashboard:/proxy/" \l "!/node?namespace=default)

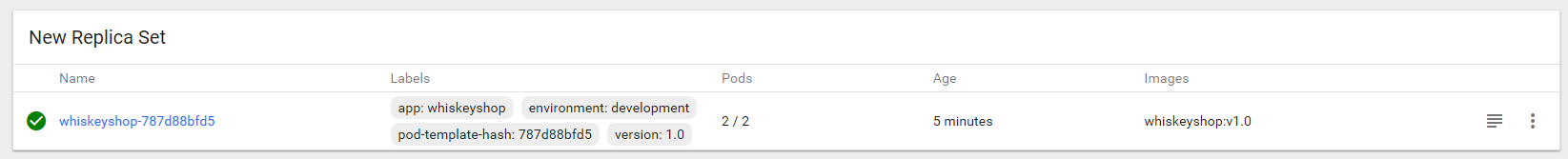
Navigate to Workloads | Deployments:



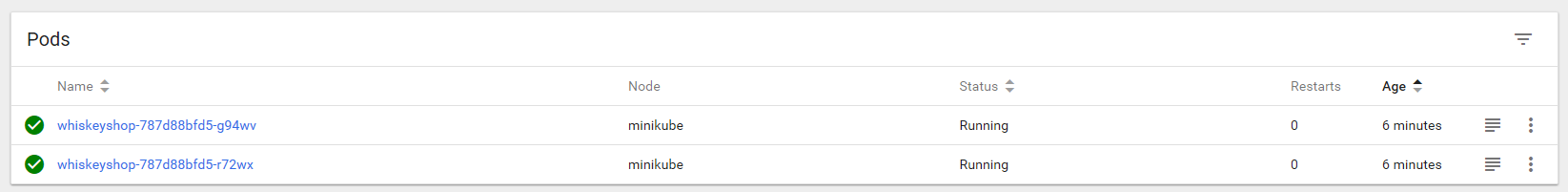
The Deployment we have created contains a Replica Set:



### Replica Sets



The Replica Set contains two Pods:



### Pods

The first Pod includes 1 container (based on image whiskeyshop:v1.0) and has IP address 172.17.0.6:



Here you can also see the Environment variables, this Pod/Container uses:

spring.profiles.active: mysql

spring.datasource.url: jdbc:mysql://172.17.0.5:3306/test?allowPublicKeyRetrieval=true&useSSL=false

The second Pod includes 1 container (based on image whiskeyshop:v1.0) and has IP address 172.17.0.7:

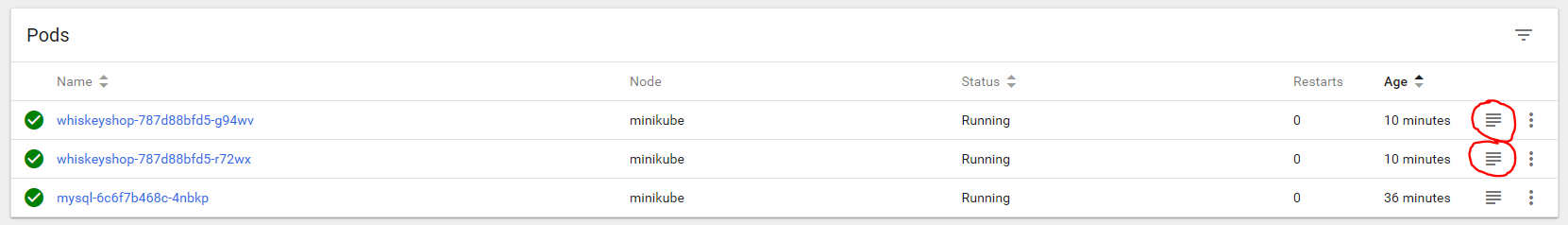


Here you can also see the Environment variables, this Pod/Container uses:

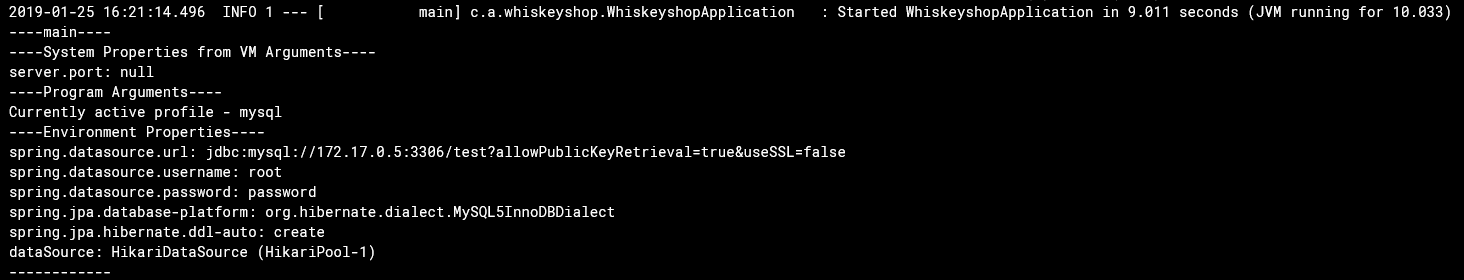
spring.profiles.active: mysql

spring.datasource.url: jdbc:mysql://172.17.0.5:3306/test?allowPublicKeyRetrieval=true&useSSL=false

If we look at the Logs of the Pods:

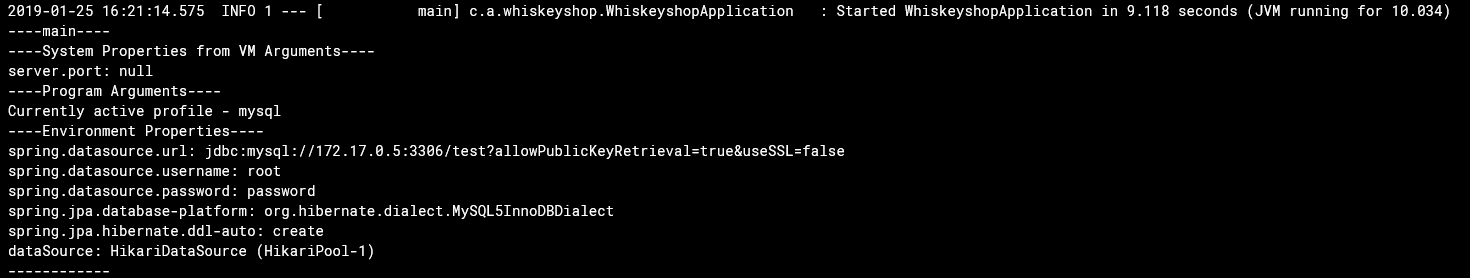


Logs from whiskeyshop-container, in whiskeyshop-787d88bfd5-g94wv:



Here you can see the Environment variables that were used, when creating this Pod/Container.

Logs from whiskeyshop-container, in whiskeyshop-787d88bfd5-r72wx:



Here you can see the Environment variables that were used, when creating this Pod/Container.

## Calling the whiskeyshop application

Alright, so now your application should be running two Docker container. Let’s test if the application also responds by firing some requests.

First, we add some whiskey-products with POST-requests to the whiskeyshop application:

curl --header "Content-Type: application/json" --request POST --data '{"id": 1, "name": "**Jack Daniels Tennessee**"}' http://10.107.185.203:8180/whiskeys

curl --header "Content-Type: application/json" --request POST --data '{"id": 2, "name": "**Crown Royal Canadia**"}' http://10.107.185.203:8180/whiskeys

curl --header "Content-Type: application/json" --request POST --data '{"id": 3, "name": "**Jim Beam bourbon**"}' http://10.107.185.203:8180/whiskeys

As you can see, we are using the Cluster IP address of the service.

Next, you should be able to see your updated list at the /whiskeys path with a GET request. First we try the service IP address:

curl http://10.107.185.203:8180/whiskeys

[{"id":"1","name":"**Jack Daniels Tennessee**"},{"id":"2","name":"**Crown Royal Canadia**"},{"id":"3","name":"**Jim Beam bourbon**"}]

Also let’s try, the first Pod IP address:

curl http://172.17.0.6:8080/whiskeys

[{"id":"1","name":"**Jack Daniels Tennessee**"},{"id":"2","name":"**Crown Royal Canadia**"},{"id":"3","name":"**Jim Beam bourbon**"}]

And then let’s try, the second Pod IP address:

curl http://172.17.0.7:8080/whiskeys

[{"id":"1","name":"**Jack Daniels Tennessee**"},{"id":"2","name":"**Crown Royal Canadia**"},{"id":"3","name":"**Jim Beam bourbon**"}]

Remember, this deployment uses the whiskeyshop (Spring Application) with an external MySQL database, running in a separate Docker container. So, the first and second Pod both use the same external MySQL database. Therefor it doesn’t matter to which Pod the service sent the POST-requests. Each of the above GET-requests responds with the same answer (all 3 whiskey-products).

As a final step in this lab we check the contents of the whiskey table. For this we use a mysql-client.

kubectl --namespace=nl-amis-development run -it --rm --image=mysql:5.6 --restart=Never mysql-client -- mysql -h mysql-service.nl-amis-development -ppassword

If you don't see a command prompt, try pressing enter.

Probably you have to click on the Enter button.

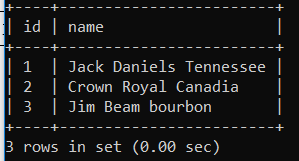
mysql> use test;

Reading table information for completion of table and column names

You can turn off this feature to get a quicker startup with -A

Database changed

mysql> select \* from whiskey;



Again we see all 3 whiskey-products.

mysql> exit

Bye

pod "mysql-client" deleted