KUBERNETES with Spring Boot

Get information

```
kubectl get all
```

Create deployment

```
kubectl create deployment my-deployment --image <DcokerHub image> --dry-run client
-o=yaml > deployment.yaml
```

This will create a deployment.yaml file with the deployment configuration. There you can change the configuration as needed.

Create service

```
kubectl create service clusterip my-service --tcp=80:80 --dry-run=client -o=yaml >
service.yaml
```

Apply deployment and service

```
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
```

Now if you run `kubectl get all` you should see the deployment and service running.

Delete deployment and service

```
kubectl delete -f deployment.yaml
kubectl delete -f service.yaml
```

Expose service

To expose the service to the internet you can edit the service.yaml file and change the type from ClusterIP to LoadBalancer or NodePort. Then run `kubectl apply -f service.yaml` again. Now you can access the service using the external IP of the service.

Setting environment variables

One very common use cases to use environment variables is to override log levels.

For example, if you have a Spring Boot application you can set the log level to DEBUG by setting the environment variable `LOG_LEVEL` to `DEBUG`.

In the deployment.yaml file you can add the following configuration to set the environment variable:

```
containers:
- image: springframeworkguru/kbe-rest-brewery
  name: kbe-rest-brewery
  resources: {}
  env:
  - name: LOGGING_LEVEL_GURU_SPRINGFRAMEWORK_SFGRESTBREWERY
    value: info
```

The name of the environment variable comes from the `application.properties` file. So it's the path to the property in the application.properties file.

To apply the changes run `kubectl apply -f deployment.yaml`. We will see another pod created with the new configuration. If we do `kubectl logs <pod-name>` we will see there is no DEBUG logs anymore.

Readiness and Liveness probes

Readiness and Liveness probes are used to check if the application is ready to receive traffic and if the application is still alive. Spring Boot Actuator provides endpoints to check if the application is ready and alive. To add the probes to the deployment.yaml file you can add the following configuration:

```
    - name: MANAGEMENT_ENDPOINT_HEALTH_PROBES_ENABLED
      value: "true"
    - name: MANAGEMENT_HEALTH_READINESSTATE_ENABLED
      value: "true"
    readinessProbe:
      httpGet:
      port: 8080
      path: /actuator/health/readiness
```

The `readinessProbe` will check if the application is ready to receive traffic, sending a GET request to the `/actuator/health/readiness` endpoint. If the application is not ready, the pod will be restarted. Note: Kubernetes expects the values as strings, so you need to put the values in quotes.

Now if we go to Postman and sent the following URL:

```
http://localhost:<NodePort>/actuator/health/readiness
```

We should see the response `{"status":"UP"}`.

Now we can add the liveness probe to the deployment.yaml file:

```
    - name: MANAGEMENT_HEALTH_LIVENESSTATE_ENABLED
      value: "true"
    livenessProbe:
      httpGet:
      port: 8080
      path: /actuator/health/liveness
```

Restart the pod and check in Postman if the liveness probe is working:

```
http://localhost:<NodePort>/actuator/health/liveness
```

We should see the response `{"status":"UP"}`.

Now Kubernetes will check periodically if the application is ready to receive traffic and if the application is still alive. If the application is not ready or alive, the pod will be restarted.

Graceful Shutdown

When a pod is terminated, Kubernetes sends a SIGTERM signal to the pod. The pod has 30 seconds to shutdown gracefully. If the pod doesn't shutdown in 30 seconds, Kubernetes will send a SIGKILL signal to the pod, which will kill the pod immediately.

We have to set and environment variable in the deployment.yaml file to enable the graceful shutdown in Spring Boot:

```
    - name: SERVER_SHUTDOWN
      value: "graceful"
```

Now set the property for Kubernetes:

```
    lifecycle:
      preStop:
        exec:
          command: ["sh", "-c", "sleep 10"]
```

This will make the pod wait 10 seconds before shutting down. This is useful if you have a database connection that needs to be closed before the pod is terminated.

There is no an easy way to test the graceful shutdown. You can try to simulate a shutdown by running `kubectl delete pod <pod-name>`. The pod should wait 10 seconds before shutting down.

MICROSERVICES with Kubernetes

When you have multiple microservices running in Kubernetes, you can use Kubernetes Services to communicate between the microservices.

Guru Brewery example:

- 4 microservices: beer, inventory-failover, inventory-service, and order-service.
  - The order-service communicates with the inventory-service to check if there is enough inventory to place an order.
  - The inventory-service communicates with the inventory-failover to check if the inventory-service is down.
  - The inventory-failover is a dummy service that always returns true.
  - The beer service is a dummy service that returns a list of beers.
- Each service is a Spring Boot application (like the brewery example).
- Brewery gateway: a Spring Boot application that acts as a gateway to the microservices.
- Each microservice has its own deployment and service in Kubernetes so each one will be a container in a pod.

Docker Compose

Define and run multi-container Docker applications. With Docker Compose you can define the services, networks, and volumes in a YAML file. Then you can run the application with a single command.

The configuration file is called `compose-local.yml`. Here there is some services defined:

- mysql: a MySQL database, give persistence to the application (user and password)
- elasticsearch, kibana and filebeat: for loggin (ports to redirect the logs and volumes to store the logs)
- jms: a JMS (Java Message Service) server (ports to redirect the messages)
- And the services that are part of the application (defined before).

To run the application you can run `docker-compose -f compose-local.yml up`. This will start all the services defined in the configuration file. We can check the services running with `docker ps`.

Now if we go to Postman and send a GET request to `http://localhost:9090/api/v1/beer` we should see the response with the list of beers. Because we are using 9090 port, we are using the gateway service. If we send the request to `http://localhost:8080/api/v1/beer` we should see the same response, but we are using the beer service directly.

Finally, we can visit the kiabana dashboard to see the logs of the application. To do this we have to go to `http://localhost:5601` and create an index pattern to see the logs (the option may be in a footer). Not working for me.

For shutting down the application you can run `docker compose -f compose-local.yml down`.

Infrastructure Services

Infrastructure services are services that are needed for the application to run. For example, a database, a message broker, a logging system, etc. These services are not part of the application, but the application

needs them to run.

MySQL

If we take a look to the Docker Compose file we have MySQL defined with two environment variables: `MYSQL_ROOT_PASSWORD` and `MYSQL_DATABASE`. The first one is the password for the root user and the second one is the name of the database. Now, we have to replicate this in a Kubernetes context. We creaate a deployment.yaml file for MySQL:

```
kubectl create deployment mysql --image=mysql:5.7 --dry-run=client -o=yaml >
mysql-deployment.yml
```

In this file we have to add the environment variables of the Docker Compose file:

```
  env:
  - name: MYSQL_ROOT_PASSWORD
    value: dbpassword
  - name: MYSQL_DATABASE
    value: beerservice
```

Now we can apply the deployment:

```
kubectl apply -f mysql-deployment.yml
```

Finally, we have to create a service for MySQL. We can create a service.yaml file:

```
kubectl create service clusterip mysql --tcp=3306:3306 --dry-run=client -o=yaml >
mysql-service.yml
```

We use tcp port to expose the MySQL service. Now we can apply the service:

```
kubectl apply -f mysql-service.yml
```

If we check the services running with `kubectl get all` we should see the MySQL service running.

```
NAME                 TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
service/kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP    24h
service/mysql        ClusterIP   10.101.45.110   <none>        3306/TCP   4s

NAME                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/mysql   1/1     1            1           99s
```

Note: we can declare the deployment and service in the same file.

JMS (Java Message Service)

JMS is a messaging standard that allows Java EE applications to create, send, receive, and read messages. It is a messaging system that allows communication between different components of a distributed application.

The JMS server is defined in the Docker Compose file with only the port mapping, so it's quite simple to replicate in Kubernetes. We can create a deployment.yaml file for the JMS server:

```
kubectl create deployment jms --image=vromero/activemq-artemis --dry-run=client -
o=yaml > jms-deployment.yml
kubectl apply -f jms-deployment.yml
```

Now we can create a service for the JMS server. Because of the JMS only has two port mappings, we can use the `--tcp` flag to expose the ports and then replicate in the service.yaml file:

```
kubectl create service clusterip jms --tcp=8161:8161 --tcp=61616:61616 --dry-
run=client -o=yaml > jms-service.yml
kubectl apply -f jms-service.yml
```

Now we can check the services running with `kubectl get all` and we should see the JMS service running.

```
NAME                 TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
service/jms          ClusterIP   10.97.118.113   <none>        8161/TCP,61616/TCP
8s
service/kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP
24h
service/mysql        ClusterIP   10.101.45.110   <none>        3306/TCP
3m44s

NAME                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/jms     1/1     1            1           28s
deployment.apps/mysql   1/1     1            1           5m19s
```

Spring Boot Microservices

Recap of the microservices: beer, inventory-failover, inventory-service, and order-service. Each microservice has its own deployment and service in Kubernetes. Microservices main goal is to deploy some functionality in a separate service. This way we can scale the services independently, and we can deploy new features without affecting the other services.

Services

We start creating the deployment.yaml file for each microservice. And for those which have environment variables we have to add them to the deployment.yaml file manually.

```
kubectl create deployment inventory-service --image=springframeworkguru/kbe-brewery-inventory-service --dry-run=client -o=yaml > inventory-deployment.yml
kubectl create deployment inventory-failover --image=springframeworkguru/kbe-brewery-inventory-failover --dry-run=client -o=yaml > inventory-failover-deployment.yml
kubectl create deployment beer-service --image=springframeworkguru/kbe-brewery-beer-service --dry-run=client -o=yaml > beer-deployment.yml
kubectl create deployment order-service --image=springframeworkguru/kbe-brewery-order-service --dry-run=client -o=yaml > order-deployment.yml
```

Now we create the services for each microservice, keeping an eye on the ports that the services are using.

```
kubectl create service clusterip inventory-service --tcp=8082:8082 --dry-run=client -o=yaml > inventory-service.yml
kubectl create service clusterip inventory-failover --tcp=8083:8083 --dry-run=client -o=yaml > inventory-failover-service.yml
kubectl create service clusterip beer-service --tcp=8080:8080 --dry-run=client -o=yaml > beer-service.yml
kubectl create service clusterip order-service --tcp=8081:8081 --dry-run=client -o=yaml > order-service.yml
```

At this point, with every environment variable and port mapping set, we can apply the deployment and service files.

```
kubectl apply -f inventory-deployment.yml
kubectl apply -f inventory-failover-deployment.yml
kubectl apply -f beer-deployment.yml
kubectl apply -f order-deployment.yml
kubectl apply -f inventory-service.yml
kubectl apply -f inventory-failover-service.yml
kubectl apply -f beer-service.yml
kubectl apply -f order-service.yml
```

Now we can check the services running with `kubectl get all` and we should see all the microservices up and running.

Readiness and Liveness Probes

We can add the readiness and liveness probes to the deployment.yaml file of each microservice. As we did before in the service exercise, now we define the same variables but in all the microservices (except the inventory-failover service).

```
    - name: MANAGEMENT_ENDPOINT_HEALTH_PROBES_ENABLED
      value: "true"
    - name: MANAGEMENT_HEALTH_READINESSTATE_ENABLED
      value: "true"
    - name: MANAGEMENT_HEALTH_LIVENESSSTATE_ENABLED
      value: "true"
    readinessProbe:
        httpGet:
        port: 8080
    livenessProbe:
        httpGet:
        port: 8080
```

Now if we apply the changes to the deployment files and check the services running with `kubectl get all` we could see services pods duplicated, because they are restarting to apply the changes (in a few seconds they should be running again).

Graceful Shutdown

We can add the graceful shutdown to the deployment.yaml file of each microservice. We have to add the environment variable `SERVER_SHUTDOWN` with the value `graceful` and the lifecycle configuration to wait 10 seconds before shutting down.

```
    - name: SERVER_SHUTDOWN
      value: "graceful"
    lifecycle:
      preStop:
        exec:
          command: ["sh", "-c", "sleep 10"]
```

And apply the changes to the deployment files with `kubectl apply -f <deployment-file>`. Now if we delete a pod with `kubectl delete pod <pod-name>` we should see the pod waiting 10 seconds before shutting down.

Ingess Controllers

Ingress controllers are used to expose the services to the internet. In Kubernetes, the services are only accessible inside the cluster. To expose the services to the internet we can use an Ingress controller.

```
graph LR;
  client([client])-. Ingress-managed
 load balancer .->ingress[Ingress];
  ingress-->|routing rule|service[Service];
  subgraph cluster
  ingress;
  service-->pod1[Pod];
  service-->pod2[Pod];
```

```
    end
    classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000;
    classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff;
    classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5;
    class ingress,service,pod1,pod2 k8s;
    class client plain;
    class cluster cluster;
```