Lab 29 - Kubernetes an Canaries

* Open the message-service project we've worked on in the previous recipes. Add the following Dockerfile file to the root directory of the project:

**FROM openjdk:8-jdk-alpine**  
**VOLUME /tmp**  
**EXPOSE 8082  
ARG JAR\_FILE=build/libs/message-service-1.0-SNAPSHOT.jar**  
**ADD ${JAR\_FILE} app.jar**  
**ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]**

* In order for Kubernetes to know whether the service is running, we need to add a liveness probe endpoint. Open the MessageController.java file and add a method to respond to GET requests at the /ping path:

package com.packtpub.microservices.ch09.message.controllers;  
  
import com.packtpub.microservices.ch09.message.MessageRepository;  
import com.packtpub.microservices.ch09.message.clients.SocialGraphClient;  
import com.packtpub.microservices.ch09.message.exceptions.MessageNotFoundException;  
import com.packtpub.microservices.ch09.message.exceptions.MessageSendForbiddenException;  
import com.packtpub.microservices.ch09.message.models.Message;  
import com.packtpub.microservices.ch09.message.models.UserFriendships;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.http.ResponseEntity;  
import org.springframework.scheduling.annotation.Async;  
import org.springframework.web.bind.annotation.\*;  
import org.springframework.web.client.RestTemplate;  
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;  
  
import java.net.URI;  
import java.util.List;  
import java.util.concurrent.CompletableFuture;  
  
@RestController  
public class MessageController {  
  
 @Autowired  
 private MessageRepository messagesStore;  
  
 @Autowired  
 private SocialGraphClient socialGraphClient;  
  
 @RequestMapping(path = "/{id}", method = RequestMethod.GET, produces = "application/json")  
 public Message get(@PathVariable("id") String id) throws MessageNotFoundException {  
 return messagesStore.get(id);  
 }  
  
 @RequestMapping(path = "/ping", method = RequestMethod.GET)  
 public String readinessProbe() {  
 return "ok";  
 }  
  
 @RequestMapping(path = "/", method = RequestMethod.POST, produces = "application/json")  
 public ResponseEntity<Message> send(@RequestBody Message message) throws MessageSendForbiddenException {  
 List<String> friendships = socialGraphClient.getFriendships(message.getSender());  
  
 if (!friendships.contains(message.getRecipient())) {  
 throw new MessageSendForbiddenException("Must be friends to send message");  
 }  
  
 Message saved = messagesStore.save(message);  
 URI location = ServletUriComponentsBuilder  
 .fromCurrentRequest().path("/{id}")  
 .buildAndExpand(saved.getId()).toUri();  
 return ResponseEntity.created(location).build();  
 }  
  
 @RequestMapping(path = "/user/{userId}", method = RequestMethod.GET, produces = "application/json")  
 public ResponseEntity<List<Message>> getByUser(@PathVariable("userId") String userId) throws MessageNotFoundException {  
 List<Message> inbox = messagesStore.getByUser(userId);  
 if (inbox.isEmpty()) {  
 throw new MessageNotFoundException("No messages found for user: " + userId);  
 }  
 return ResponseEntity.ok(inbox);  
 }  
  
 @Async  
 public CompletableFuture<Boolean> isFollowing(String fromUser, String toUser) {  
 String url = String.format(  
 "http://localhost:4567/followings?user=%s&filter=%s",  
 fromUser, toUser);  
  
 RestTemplate template = new RestTemplate();  
 UserFriendships followings = template.getForObject(url, UserFriendships.class);  
  
 return CompletableFuture.completedFuture(  
 followings.getFriendships().isEmpty()  
 );  
 }  
}

* Let's start our container registry on port 5000:

**$ docker run -d -p 5000:5000 --restart=always --name registry registry:2**

* As we're using a local repository that is not configured with a valid SSL cert, start minikube with the ability to pull from insecure repositories:

**$ minikube start --insecure-registry 127.0.0.1**

* Build the message-service docker image, and then push the image to the local container registry with the following commands:

**$ docker build . -t message-service:0.1.1**  
**...**  
**$ docker tag message-service:0.1.1 localhost:5000/message-service**  
**...**  
**$ docker push localhost:5000/message-service**

* A **Kubernetes Deployment** object describes the desired state for a pod and ReplicaSet. In our deployment, we'll specify that we want three replicas of our message-service pod running at all times, and we'll specify the liveness probe that we created a few steps earlier. To create a deployment for our message-service, create a file called deployment.yaml with the following contents:

apiVersion: extensions/v1beta1  
kind: Deployment  
metadata:  
 name: message-service  
spec:  
 replicas: 3  
 template:  
 metadata:  
 labels:  
 app: "message-service"  
 track: "stable"  
 spec:  
 containers:  
 - name: "message-service"  
 image: "localhost:5000/message-service"  
 imagePullPolicy: IfNotPresent  
 ports:  
 - containerPort: 8082  
 livenessProbe:  
 httpGet:  
 path: /ping  
 port: 8082  
 scheme: HTTP  
 initialDelaySeconds: 10  
 periodSeconds: 30  
 timeoutSeconds: 1

* Next, using kubectl, we'll create our deployment object:

**$ kubectl create -f deployment.yaml**

* You can now verify that our deployment is live and that Kubernetes is creating the pod replicas by running kubectl get pods:

**$ kubectl get pods**

* Now that our application is running in Kubernetes, the next step is to create an update and roll it out to a subset of pods. First, we need to create a new docker image; in this case, we'll call it version 0.1.2 and push it to the local repository:

**$ docker build . -t message-service:0.1.2**  
**...**  
**$ docker tag message-service:0.1.2 localhost:5000/message-service**  
**$ docker push localhost:5000/message-service**

* We can now configure a deployment to run the newest version of our image before rolling it out to the rest of the pods.