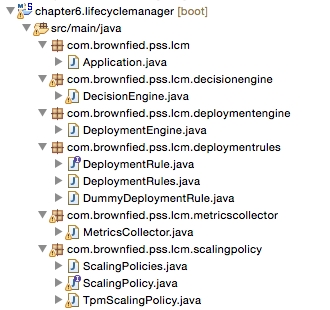
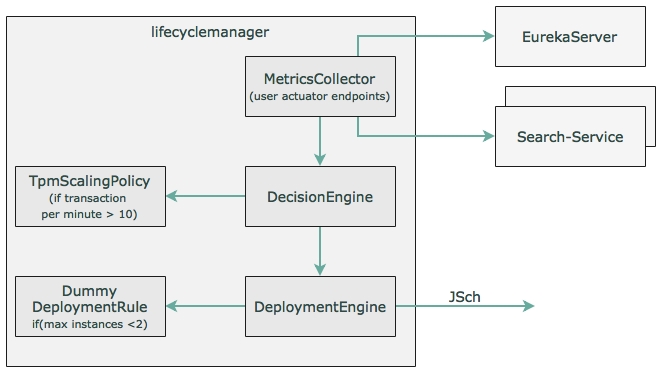
Lab 20 - Autoscale

* Create a new Spring Boot application and name it chapter6.lifecyclemanager. The project structure is shown in the following diagram:
* 
* The flowchart for this example is as shown in the following diagram:
* 
* The components of this diagram are explained in details here.
* Create a MetricsCollector class with the following method. At the startup of the Spring Boot application, this method will be invoked using CommandLineRunner, as follows:
* public void start(){  
   while(true){   
   eurekaClient.getServices().forEach(service -> { System.out.println("discovered service "+ service);  
   Map metrics = restTemplate.getForObject("http://"+service+"/metrics",Map.class);  
   decisionEngine.execute(service, metrics);  
   });   
   }   
  }
* The preceding method looks for the services registered in the Eureka server and gets all the instances. In the real world, rather than polling, the instances should publish metrics to a common place, where metrics aggregation will happen.
* The following DecisionEngine code accepts the metric and applies certain scaling policies to determine whether the service requires scaling up or not:
* public boolean execute(String serviceId, Map metrics){  
   if(scalingPolicies.getPolicy(serviceId).execute(serviceId, metrics)){   
   return deploymentEngine.scaleUp(deploymentRules.getDeploymentRules(serviceId), serviceId);   
   }  
   return false;  
   }
* Based on the service ID, the policies that are related to the services will be picked up and applied. In this case, a minimal TPM scaling policy is implemented in TpmScalingPolicy, as follows:
* public class TpmScalingPolicy implements ScalingPolicy {  
   public boolean execute(String serviceId, Map metrics){  
   if(metrics.containsKey("gauge.servo.tpm")){  
   Double tpm = (Double) metrics.get("gauge.servo.tpm");  
   System.out.println("gauge.servo.tpm " + tpm);  
   return (tpm > 10);  
   }  
   return false;  
   }  
  }
* If the policy returns true, DecisionEngine then invokes DeploymentEngine to spin up another instance. DeploymentEngine makes use of DeploymentRules to decide how to execute scaling. The rules can enforce the number of min and max instances, in which region or machine the new instance has to be started, the resources required for the new instance, and so on. DummyDeploymentRule simply makes sure the max instance is not more than 2.
* DeploymentEngine, in this case, uses the **JSch** (**Java Secure Channel**) library from JCraft to SSH to the destination server and start the service. This requires the following additional Maven dependency:
* <dependency>  
   <groupId>com.jcraft</groupId>  
   <artifactId>jsch</artifactId>  
   <version>0.1.53</version>  
  </dependency>
* The current SSH implementation is kept simple enough as we will change this in future chapters. In this example, DeploymentEngine sends the following command over the SSH library on the target machine:
* **String command ="java -jar -Dserver.port=8091 ./work/codebox/chapter6/chapter6.search/target/search-1.0.jar";**
* Integration with Nexus happens from the target machine using Linux scripts with Nexus CLI or using curl. In this example, we will not explore Nexus.
* The next step is to change the Search microservice to expose a new gauge for TPM. We have to change all the microservices developed earlier to submit this additional metric.
* We will only examine Search in this chapter, but in order to complete it, all the services have to be updated. In order to get the gauge.servo.tpm metrics, we have to add TPMCounter to all the microservices.
* The following code counts the transactions over a sliding window of 1 minute:
* class TPMCounter {  
   LongAdder count;  
   Calendar expiry = null;   
   TPMCounter(){  
   reset();  
   }   
   void reset (){  
   count = new LongAdder();  
   expiry = Calendar.getInstance();  
   expiry.add(Calendar.MINUTE, 1);  
   }  
   boolean isExpired(){  
   return Calendar.getInstance().after(expiry);  
   }  
   void increment(){  
   if(isExpired()){  
   reset();  
   }  
   count.increment();  
   }  
  }
* The following code needs to be added to SearchController to set the tpm value:
* class SearchRestController {  
   TPMCounter tpm = new TPMCounter();  
   @Autowired  
   GaugeService gaugeService;  
   //other code
* The following code is from the get REST endpoint (the search method) of SearchRestController, which submits the tpm value as a gauge to the actuator endpoint:
* tpm.increment();  
  gaugeService.submit("tpm", tpm.count.intValue());

### Running the life cycle manager

Perform the following steps to run the life cycle manager developed in the previous section:

* Edit DeploymentEngine.java and update the password to reflect the machine's password, as follows. This is required for the SSH connection:
* session.setPassword("rajeshrv");
* Build all the projects by running Maven from the root folder (Chapter 6) via the following command:
* **mvn -Dmaven.test.skip=true clean install**
* Then, run RabbitMQ, as follows:
* **./rabbitmq-server**
* Ensure that the Config server is pointing to the right configuration repository. We need to add a property file for the life cycle manager.
* Run the following commands from the respective project folders:
* **java -jar target/config-server-0.0.1-SNAPSHOT.jar**  
  **java -jar target/eureka-server-0.0.1-SNAPSHOT.jar**  
  **java -jar target/lifecycle-manager-0.0.1-SNAPSHOT.jar**  
  **java -jar target/search-1.0.jar**  
  **java -jar target/search-apigateway-1.0.jar**  
  **java -jar target/website-1.0.jar**
* Once all the services are started, open a browser window and load http://localhost:8001.
* Execute the flight search 11 times, one after the other, within a minute. This will trigger the decision engine to instantiate another instance of the Search microservice.
* Open the Eureka console (http://localhost:8761) and watch for a second **SEARCH-SERVICE**. Once the server is started, the instances will appear as shown here:
* 