Distributed application and Service Discovery

In this lab we will develop a distributed application comprising of pages and category microservices and deploy them on K8s cluster. We will be addressing service to service communication scenario.

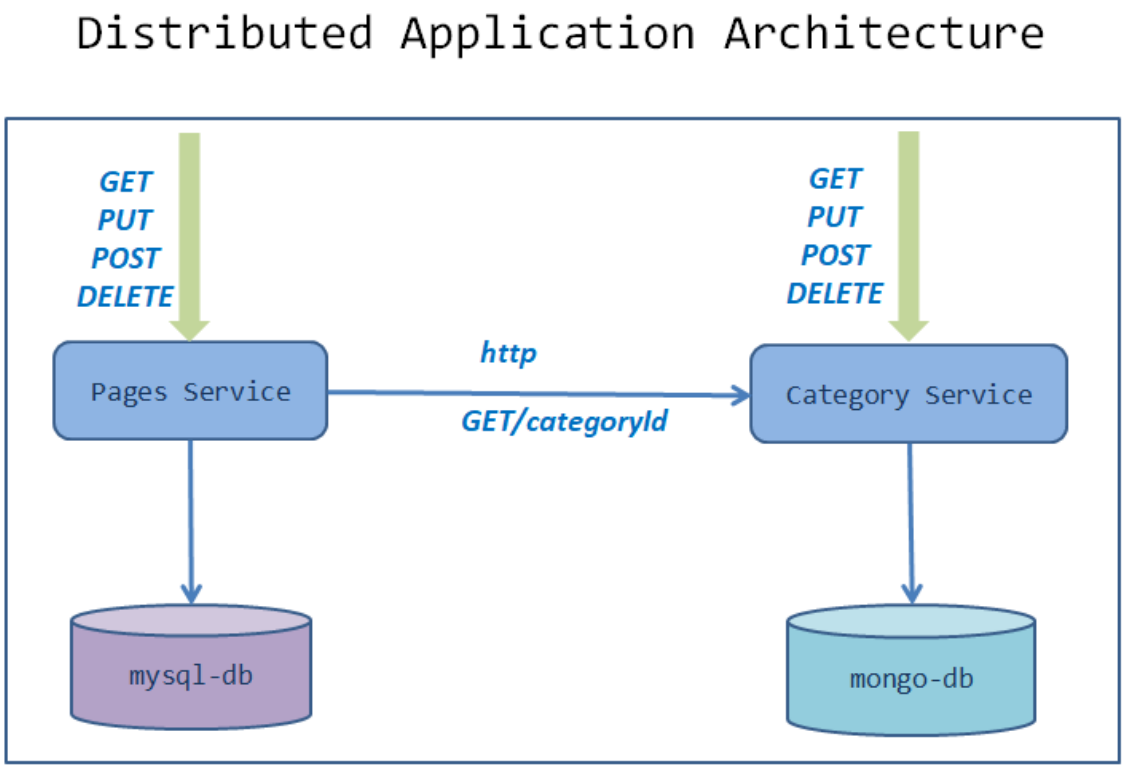
In order to accomplish our goal, we will make use of `Spring Cloud Kubernetes` for service discovery and implement client side loadbalancing using `Ribbon`. We will be refactoring pages and category microservice to enable service to service communication.

Learning Outcomes

After completing the lab, you will be able to:

1. Design and implement a distributed system
2. Understand and implement service discovery using Spring Cloud Kubernetes and native K8s service discovery
3. Understand and implement client side load balancing using Netflix Ribbon

Service dependencies



1. The pages microservice depends on the category microservice for fetching the `category` based on the `categoryId` in order to validate the business rules.
2. If the `page` entity associated with a non-existing or an invalid `category/categoryId` gets created, will lead to inconsistency within the system.

Design

1. During a POST reqest to pages service, the `categoryId` has to be validated against an existing category.
2. Category application has to be refactored, so that it is disoverable by pages application.
3. Pages should be relying on a `client load-balancing` feature in order to automatically discover at which endpoint(s) it can reach the Category service.
4. Client side load balancing enables Kubernetes client to populate a `Ribbon` ServerList containing information about interested endpoints.

Implementation - Refactor Category Application

1. Open category application in intellij
2. Add the necessary dependencies to provide kubernetes support to category application.
3. Apply the dependency management plugin and bring in spring cloud dependencies into the dependency management. Add the spring cloud kubernetes starter dependency. Ensure the version compatibility is maintained between spring boot and spring cloud kubernetes.

   Refer to the below `build.gradle` which contains all the necessary dependencies

```
buildscript {
        dependencies {
                classpath "io.spring.gradle:dependency-man
agement-plugin:1.0.9.RELEASE"
        }
}
plugins {
        id 'org.springframework.boot' version '2.3.1.RELEA
SE'
        id 'io.spring.dependency-management' version '1.0.
9.RELEASE'
        id 'java'
}
```

```
apply plugin: "io.spring.dependency-management"

dependencyManagement {
        imports {
                mavenBom 'org.springframework.cloud:spring
-cloud-dependencies:2020.0.0-M2'
        }
}
repositories {
        mavenCentral()
        maven {
                url 'https://repo.spring.io/milestone'
        }
}

dependencies {
        implementation 'org.springframework.boot:spring-bo
ot-starter-web'
        implementation 'org.springframework.boot:spring-bo
ot-starter-data-mongodb'
        implementation 'org.springframework.cloud:spring-c
loud-starter-kubernetes'
        testImplementation('org.springframework.boot:sprin
g-boot-starter-test') {
                exclude group: 'org.junit.vintage', modul
e: 'junit-vintage-engine'
        }
}

test {
        useJUnitPlatform()
}
```

4. Create `application.properties` files within `src/main/resources` folder, and add the application name property.

```
spring.application.name=category
```

5. Annotate CategoryApplication class with `@EnableDiscoveryClient` from the package `org.springframework.cloud.client.discovery.EnableDiscoveryClient`

6. Build the application and test it locally before deployment. Refer Category Curl Guide

Implementation - Refactor Pages Application

1. Open pages application in intellij
2. Add the necessary dependencies to provide kubernetes support to pages application.
3. Apply the dependency management plugin and bring in spring cloud dependencies into the dependency management. Add the spring cloud kubernetes starter dependency. Ensure the version compatibility is maintained between spring boot and spring cloud kubernetes.
4. Refer to the below `build.gradle` which contains all the necessary dependencies

```
buildscript {
        dependencies {
                classpath "io.spring.gradle:dependency-man
agement-plugin:1.0.9.RELEASE"
        }
}
plugins {
        id 'org.springframework.boot' version '2.3.1.RELEA
SE'
        id 'io.spring.dependency-management' version '1.0.
9.RELEASE'
        id 'java'
}

apply plugin: "io.spring.dependency-management"

dependencyManagement {
        imports {
                mavenBom 'org.springframework.cloud:spring
-cloud-dependencies:2020.0.0-M2'
        }
}
group = 'com.example'

repositories {
        mavenCentral()
                maven {
                                url 'https://repo.spring.i
o/milestone'
                        }
}

bootRun.environment([
```

```
                "PAGE_CONTENT": "YellowPages",
])

test.environment([
                "PAGE_CONTENT": "YellowPages",
])

dependencies {

        implementation 'org.springframework.boot:spring-bo
ot-starter-jdbc'
        implementation 'org.springframework.boot:spring-bo
ot-starter-web'
        implementation 'org.springframework.boot:spring-bo
ot-starter-actuator'
        implementation 'org.springframework.cloud:spring-c
loud-starter-kubernetes:1.1.3.RELEASE'
        implementation 'org.springframework.cloud:spring-c
loud-starter-netflix-ribbon'
        implementation 'org.springframework.cloud:spring-c
loud-starter-kubernetes-ribbon:1.1.1.RELEASE'
        implementation 'org.springframework.cloud:spring-c
loud-starter-openfeign:2.2.3.RELEASE'

        implementation 'org.springframework.cloud:spring-c
loud-commons'

        implementation 'mysql:mysql-connector-java:8.0.12'
        testImplementation('org.springframework.boot:sprin
g-boot-starter-test') {
                exclude group: 'org.junit.vintage', modul
e: 'junit-vintage-engine'
        }
}

test {
        useJUnitPlatform()
}
```

5. Annotate PageApplication class with `@EnableDiscoveryClient` and `@EnableFeignClients`.

6. `Feign` helps us to write declarative REST service interfaces, rather than programmtically constructing the URL to use the `RestTemplate`, which is more suitable for our usecase.

7. Create a class `Category.java` within `src/main/java`

```
package org.dell.kube.pages;

public class Category {
    private Long id;
    private String categoryName;
    private String description;

    public Long getId() {
        return id;
    }

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

    public String getCategoryName() {
        return categoryName;
    }

    public void setCategoryName(String categoryName) {
        this.categoryName = categoryName;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    @Override
    public String toString() {
        return "Category{" +
                "id=" + id +
                ", categoryName='" + categoryName + '\'' +
                ", description='" + description + '\'' +
                '}';
    }
}
```

8. Create an interface CategoryClient.java within `src/main/java`. This interface is a declarative REST service interface at Pages client.

```
package org.dell.kube.pages;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariabl
e;

@FeignClient(name = "category")
public interface CategoryClient {
    @GetMapping("/category/{categoryId}")
    Category findCategory(@PathVariable("categoryId") Long
categoryId);
}
```

9. Inject `categoryClient` dependency in the `PageController` class

```
@Autowired
CategoryClient categoryClient;
```

10. Update the create/POST method of `PageController` to.
    a. Invoke Category service to validate the `categoryId` .
    b. Upon successful validation, make a `POST` request to Pages service.
    c. Handle the `FeignClient` exception which is raised if the category is not found.

```
......

    @PostMapping
    public ResponseEntity<Page> create(@RequestBody Page p
age) {

        logger.info("CREATE-INFO:Creating a new page");
        logger.debug("CREATE-DEBUG:Creating a new  page");
        Category category = null;
        try {
            category = categoryClient.findCategory(page.ge
tCategoryId());
        }
        catch(FeignException ex){
            if(ex.getMessage().contains("404")) {
                return new ResponseEntity<>(HttpStatus.NOT
_FOUND);
            }
            else{
```

```
                return new ResponseEntity<>(HttpStatus.INT
ERNAL_SERVER_ERROR);
            }
        }

        if(category ==null || category.getId()==null) {
            return new ResponseEntity<>(HttpStatus.NOT_FOU
ND);
        }
        else
        {
            Page newPage = pageRepository.create(page);
            logger.info("CREATE-INFO:Created a new page wi
th id = " + newPage.id);
            logger.debug("CREATE-DEBUG:Created a new  page
with id = " + newPage.id);
            return new ResponseEntity<Page>(newPage, HttpS
tatus.CREATED);
        }
    }
......
```

11. Testing the application locally would fail since the ApiTest cases would need category to be running locally. Hence, for convenience we sill skip the testing and build the jar file.

```
./gradlew clean build -x test
```

Getting ready for Deployment

1. Dockerize pages & category using `distributed` tag and push them to docker hub. Make sure you are in the right directory when you run the docker commands

```
docker build -t [docker-username]/category:distributed .

docker push [docker-username]/category:distributed

docker build -t [docker-username]/pages:distributed .

docker push [docker-username]/pages:distributed
```

2. In order for the Spring Cloud integration with Kubernetes to work, the service account should be given permission to access K8s resources. Otherwise, you may see a `Forbidden` error since K8s internally uses `RBAC` for security. We will provide Spring Cloud Kubernetes access to these resources: `services, pods, config`

`maps, endpoints` . Lets provide `RBAC` access to these resources by creating `ClusterRole` and `RoleBinding` to the `default` service account within your namespace.

3. Create `rbac.yaml` with the below contents in `pages/deployments` folder which is needed by the pages application. Once we create this within the cluster there will be no need to repeat it for category application as both share the same namespace and service accounts.

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: [student-name]-cluster-role
rules:
  - apiGroups: [""] # "" indicates the core API group
    resources: ["services", "pods", "configmaps", "endpoin
ts"]
    verbs: ["get", "watch", "list"]

---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: default:[student-name]-cluster-role-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: [student-name]-cluster-role
subjects:
  - kind: ServiceAccount
    name: default
    namespace: [student-name]
```

Deployment Guide

Clean up resources

1. Delete all existing deployments `kubectl delete deploy,svc --all`
2. Delete any persistent volume and persistent volume claims
3. Clean up the resources by deleting `[student-name]` namespace.

```
kubectl delete namespace [student-name]
```

Deploy Category microservice

1. Create `[student-name]` namespace.

```
kubectl apply -f deployment/pages-namespace.yaml
```

2. Set up `[student-name]` namespace to point to the current context. If the namespace is not created, the deployments will not work.

```
kubectl config set-context --current --namespace=[student-name]
```

3. Create the Database tier

```
kubectl apply -f deployment/mongo-storage-class.yaml
kubectl apply -f deployment/mongo-pv.yaml
kubectl apply -f deployment/mongo-pvc.yaml
kubectl apply -f deployment/mongo-service.yaml
kubectl apply -f deployment/mongo-deployment.yaml
```

4. Verify the deployment of database tier

```
kubectl get deployment mongo
kubectl get service mongo
kubectl get pvc
```

5. Proceed further if there are no errors, otherwise troubleshoot and fix them.

6. Create the service tier

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

7. Verify the deployment of service tier

```
kubectl get deployment category
kubectl get service category
```

8. Access the category application

```
kubectl port-forward svc/category 8080:8080
```

9. Refer Category Curl Guide for testing and proceed with the next steps

Deploy Pages microservice

1. Create the Database tier

```
kubectl apply -f deployment/rbac.yaml
kubectl apply -f deployment/mysql-storage-class.yaml
kubectl apply -f deployment/mysql-pv.yaml
kubectl apply -f deployment/mysql-pvc.yaml
kubectl apply -f deployment/mysql-service.yaml
kubectl apply -f deployment/mysql-secret.yaml
kubectl apply -f deployment/mysql-deployment.yaml
kubectl apply -f deployment/flyway-configmap.yaml
kubectl apply -f deployment/flyway-job.yaml
```

2. Verify the deployment of database tier

```
kubectl get deployment mysql
kubectl get service mysql
kubectl get pvc
kubectl get jobs
```

3. Create the Service tier

```
kubectl apply -f deployment/pages-config.yaml
kubectl apply -f deployment/pages-service.yaml
kubectl apply -f deployment/pages-deployment.yaml
```

4. Verify the deployment of database tier

```
kubectl get deploy
kubectl get svc
```

5. Proceed further if there are no errors, otherwise troubleshoot and fix them.
6. Connect to the pages service by port-forwarding for testing. `kubectl port-forward svc/pages 8080:8080`
7. Test the pages application by performing CRUD operations using curl/postman. Refer Pages Curl Guide for testing.
8. You can now deploy the distributed application to the production cluster following the same steps of deployment and test both the microservices.

Task Accomplished

We developed and deployed a distributed microservice based application to K8s cluster. We implemented service discovery and client side load balancing.

Advanced usecases

1. Bring in resiliency by implementing circuit breaker with a fallback mechanism
2. Secure service to service communication using OAuth2.0 with mutual TLS.