<https://k8s-1490992.cfapps.io/>

Workshop overview

Welcome to the cloud native journey on Kubernetes using Spring Boot!

The program is focused on enabling advanced software developers and solution architects to quickly ramp up their kubernetes skills for development and production.

You will learn how to Develop and Deploy a Microservices Application on Kubernetes . In addition, you may have the opportunity to be exposed to some of Agile practices including test driven development, continuous integration/delivery, pair-programming.

Learning Outcomes

After completing the course, you will be able to:

1. Demonstrate the ability to build greenfield, cloud native applications using Java, and Spring Boot
2. Containerizing a web application using Spring Boot
3. Dockerizing a containerized application - Build, ship and run using docker images.
4. Kubernetizing - Deploying, Running and Monitoring your web applications Kubernetes cluster.
5. Explain how to implement common distributed systems patterns by leveraging the Kubernetes resources.

# Building a Spring Boot Application

## Learning Outcomes

After completing the lab, you will be able to:

1. Describe how to create runnable Spring Boot application
2. Describe how to create a controller that responds to HTTP requests
3. Use gradle to run gradle tasks

The [pages](https://dell-edu-lab-store.s3.ap-south-1.amazonaws.com/repository/pages.zip) codebase contains a local Git repository with the starting points and the solutions for all the labs in this unit. Download the linked zip file and extract the codebase in the ~/workspace directory. The extracted directory will contain a single text file as well as the (hidden) Git files. You will be building up the code in this directory bit by bit, and we have provided reference implementations at each stage identified by tags in the Git repository. Take some time to navigate through the tags and branches using the following command:

git log --graph --decorate --oneline --all

Create a repository called pages in your GitHub account. Add this repository as a remote called origin of your local repository.Keep everything default, while creating the repository, don’t change anything other than default. You will push all of your work to this repository during the next few labs.

We will start by pushing the initial commit to GitHub, complete with the start and solutions tags.

git push origin master --tags

We can then navigate to GitHub and view the solution tags. This is handy when you get stuck during a lab and need a little help.

Create a build.gradle file with following content

plugins {

id 'org.springframework.boot' version '2.3.1.RELEASE'

id 'io.spring.dependency-management' version '1.0.9.RELEASE'

id 'java'

}

group = 'com.example'

sourceCompatibility = '11'

repositories {

mavenCentral()

}

dependencies {

implementation 'org.springframework.boot:spring-boot-starter-web'

testImplementation('org.springframework.boot:spring-boot-starter-test') {

exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'

}

}

test {

useJUnitPlatform()

}

Create the gradle ecosystem by using the following commands

gradle wrapper --gradle-version 6.4.1 --distribution-type all

Open the project in Intellij

Right-click on the project, select the option to crate new directory and create src/main/java and src/test/java

Create two packages org.dell.kube.pages and org.dell.kube.pagesapi under src/test/java

Create a test class called HomeControllerTest within package org.dell.kube.pages with below content

package org.dell.kube.pages;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Assertions.assertThat;

public class HomeControllerTest {

private final String message = "YellowPages";

@Test

public void itSaysYellowPagesHello() throws Exception {

HomeController controller = new HomeController();

assertThat(controller.getPage()).contains(message);

}

}

Create a Test class called HomeApiTest under the package org.dell.kube.pagesapi with below content

package org.dell.kube.pagesapi;

import org.dell.kube.pages.PageApplication;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.boot.test.web.client.TestRestTemplate;

import static org.assertj.core.api.Assertions.assertThat;

import static org.springframework.boot.test.context.SpringBootTest.WebEnvironment.RANDOM\_PORT;

@SpringBootTest(classes = PageApplication.class, webEnvironment = RANDOM\_PORT)

public class HomeApiTest {

@Autowired

private TestRestTemplate restTemplate;

@Override

protected Object clone() throws CloneNotSupportedException {

return super.clone();

}

@Test

public void readTest() {

String body = this.restTemplate.getForObject("/", String.class);

assertThat(body).contains("YellowPages");

}

}

Create a settings.gradle file in the root project directory with below content

rootProject.name = 'pages'

Create package org.dell.kube.pages within src/main/java

Create class PageApplication within the package org.dell.kube.pages

PageApplication.java - Example code snippet

package org.dell.kube.pages;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;

@SpringBootApplication

public class PageApplication {

public static void main(String[] args) {

SpringApplication.run(PageApplication.class, args);

}

}

Create class HomeController within the package org.dell.kube.pages

HomeController.java - Example code snippet

package org.dell.kube.pages;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

@RequestMapping("/")

public class HomeController {

@GetMapping

public String getPage(){

return "Hello from page : YellowPages";

}

}

Add application.properties in both test and src

Add the below content in both the properties files

spring.application.name=pages

Build & test the application

./gradlew clean build

Run the application

./gradlew bootRun

Access your application

Browse to http://localhost:8080

Commit your code to your github repository

git add .

git commit -m "commit message"

git push -u origin master

Lab2:

# Building your first container

## Learning Outcomes

After completing the lab, you will be able to:

1. Describe how to create Docker container
2. Describe how to write a docker file
3. Run your Spring boot application as a docker container

## Dockerizing

1. Create a new file named Dockerfile inside root project folder
2. Add instructions to download the base image. In order to run Java application using JDK 11, use - adoptopenjdk:11-jre-openj9
3. Add instructions to copy the dependencies & build artifacts(jar/war) from the local directory into the docker image
4. Provide a command or an entrypoint to start the application within the docker container

Example snippet for reference

FROM adoptopenjdk:11-jre-openj9

ARG JAR\_FILE=build/libs/pages.jar

COPY ${JAR\_FILE} app.jar

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

1. Building the Docker Image

The next step is to build the docker image. To ensure that the jar file is available locally build the project once using the gradle command.

./gradlew clean build

1. Example snippet to build the docker image

docker build -t pages .

|  |  |
| --- | --- |
|  | If you get an error containing Permission Denied while trying to connect to the docker daemon socket you will need to execute the command sudo chmod 666 /var/run/docker.sock |

1. After it finishes, run this to see the image it has built

docker image list

1. Run the image as a container

docker run -p 8080:8080 pages

In the run command, we have specified that the port 8080 on the container should be mapped to the port 8080 on the Host OS.

Once the application is started, you should be able to access it at [http://localhost:8080](http://localhost:8080/)

The container runs in the foreground. You can run the container in the background using -d option.

Pressing CTRL + C sometimes might not stop the process. You will need to manually terminate the container.

* 1. Use docker ps and fetch the container id.
  2. docker kill <container-id>

## Pushing the docker image to docker hub

1. Login with your Docker ID to push or pull images from Docker Hub.

If you don’t have a Docker ID, head over to [docker hub](https://hub.docker.com/) to create one, before proceeding futher.

docker login

1. Tag the image using the notation docker-username/repository:tag.

docker tag pages <docker-username>/pages:1.0

Make sure to replace username with your docker id in the above command.

1. Verify the newly created tagged image

docker images

1. Push the image to docker hub

docker push <docker-username>/pages:1.0

1. Pull the image from docker hub and test it on local machine. Kill the container after you test it.
2. docker run -p 8080:8080 <docker-username>/pages:1.0
3. docker ps

docker kill <container-id>

1. Commit your code to your github repository
2. git add .
3. git commit -m "commit message"

git push -u origin master

## Troubleshooting guide

1. docker container list --all. This will show both running and stopped containers. Note the CONTAINER-ID and/or the NAMES of the failed container. We’ll need it next.
2. docker container logs CONTAINER-ID. This shows the console output from the failed container.
3. To know more about docker commands Run docker -h

### **Resource clean up**

1. docker container list - View running containers. Note the CONTAINER ID and/or the NAMES of the running container.
2. docker container stop CONTAINER-ID .This stops the container.
3. docker container list - Verify that the container is now stopped.
4. docker container rm CONTAINER-ID - Delete the container.
5. docker image list. The image is still there, only the container we created is removed.
6. Verify that the docker image exists with the right tag in [docker hub](https://hub.docker.com/)

Lab3:

# Kubernetize - Deploy the application to kubernetes cluster

In this lab, we shall deploy the application to a kubernetes cluster.

## Learning Outcomes

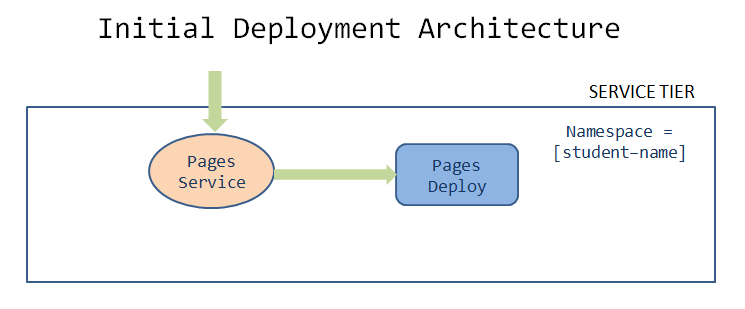
After completing the lab, you will be able to:

1. Describe how to create Kubernetes Objects
2. Describe how to write yaml files for pods, deployments and services
3. Run your Spring boot application within a kubernetes cluster
4. Basics of K8s service discovery concepts

## Starting to Kubernetize

Before starting the lab, verify the pages image created in the previous lab exists in [docker hub](https://hub.docker.com/)

Below the inital version of our deployment architecture which will evolve as our application evolves.



### **Complete the following tasks to implement the above architecture**

1. Create a unique namespace [student-name] in the cluster. For example, if your name is Bryan Evans you can replace [student-name] as bryan or evans which ever is unique across the cluster.
2. Create a deployment object with the following specifications.
3. name -> pages
4. namespace -> [student-name]
5. labels -> app: pages
6. replicas -> 1
7. image -> [docker-username]/pages:[tag](name of the image created in the previous lab)

containerPort -> 8080

1. Create a service object to expose the deployment with the following specifications.
2. name -> pages
3. namespace -> [student-name]
4. labels -> app: pages
5. selector -> pages
6. Type -> NodePort
7. targetPort -> 8080
8. port -> 8080

protocol -> TCP

### **Solution guide:**

Create manifest files for deployment

1. Create deployment/pages-namespace.yaml from the root project folder
2. apiVersion: v1
3. kind: Namespace
4. metadata:

name: [student-name]

1. Create deployment/pages-deployment.yaml from the root project folder
2. apiVersion: apps/v1
3. kind: Deployment
4. metadata:
5. labels:
6. app: pages
7. name: pages
8. namespace: [student-name]
9. spec:
10. replicas: 1
11. selector:
12. matchLabels:
13. app: pages
14. strategy: {}
15. template:
16. metadata:
17. labels:
18. app: pages
19. spec:
20. containers:
21. - image: [docker-username]/pages:[tag]
22. name: pages
23. ports:

- containerPort: 8080

1. Create deployment/pages-service.yaml from the root project folder
2. apiVersion: v1
3. kind: Service
4. metadata:
5. labels:
6. app: pages
7. name: pages
8. namespace: [student-name]
9. spec:
10. ports:
11. - name: pages-service-port
12. port: 8080
13. protocol: TCP
14. targetPort: 8080
15. selector:
16. app: pages

type: NodePort

### **Start minikube locally**

1. Start minikube locally minikube start --driver=virtualbox
2. Verify the kubectl context kubectl config get-contexts is set to minikube. If not, set it to minikube kubectl config use-context minikube
3. Deploy and test the application in minikube. Refer to [Deployment Guide](https://k8s-1490992.cfapps.io/labs/03-Kubernetes-Start.html#_deployment_guide)

### **Deploy the application to production cluster**

1. Follow [Production Cluster Guide](https://k8s-1490992.cfapps.io/labs/EKS.html#login-section) to login/connect to the production cluster.
2. Deploy and test the application in production cluster. Refer to [Deployment Guide](https://k8s-1490992.cfapps.io/labs/03-Kubernetes-Start.html#_deployment_guide)
3. Commit code changes to the github repository
4. git add .
5. git commit -m "Added K8 deployment objects"

git push -u origin master

## Deployment Guide

1. Create kubernetes objects

kubectl apply -f deployment/pages-namespace.yaml

kubectl apply -f deployment/pages-service.yaml

kubectl apply -f deployment/pages-deployment.yaml

1. Verify the created objects

kubectl get deployment pages --namespace [student-name]

kubectl get service pages --namespace [student-name]

1. Set up [student-name] namespace to point to the current context

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

1. Access the pages application by port-forwarding using kubectl, enabling the application can be served via localhost on port 8080

kubectl port-forward svc/pages 8080:8080

curl localhost:8080

Lab 4

# Externalize Configuration in Kubernetes

## Learning Outcomes

After completing the lab, you will be able to:

1. Use environment variables to configure an application running locally
2. Describe how to configure a Spring Boot application on Kubernetes with ConfigMap

Before starting the lab, cherry pick the config-start tag.

git cherry-pick config-start --strategy-option theirs

|  |  |
| --- | --- |
|  | In case you get an error when you cherry-pick, open intellij, right-click on the project, select git → resolve-conflicts → accept theirs |

## Externalize the welcome message

1. As a result of cherry-pick some test cases have been updated. Have a look at HomeControllerTest class.
2. Externalize the message being displayed into an environment variable called PAGE\_CONTENT
3. The goal is to ensure that your test cases are passing before dockerizing.
4. Update HomeController class to externalize the welcome message through constructor injection
5. private String pageContent;
6. public HomeController(@Value("${page.content}") String pageContent){
7. this.pageContent=pageContent;
8. }
9. @GetMapping
10. public String getPage(){
11. return "Hello from page : "+pageContent+" ";

}

1. Set the environment variable in build.gradle file for test and dev environments
2. bootRun.environment([
3. "PAGE\_CONTENT": "YellowPages",
4. ])
5. test.environment([
6. "PAGE\_CONTENT": "YellowPages",

])

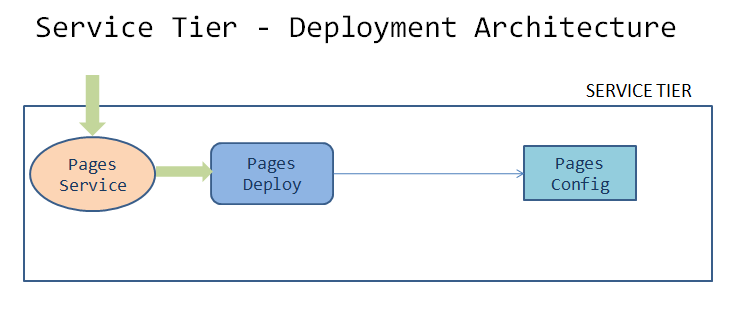
1. Build and test the application just like you did in previous labs.
2. Dockerize the application and tag it appropriately

docker build -t [docker\_username]/pages:[tag] .

1. Push the image to docker

docker push [docker\_username]/pages:[tag]

## Kubernetize the application



### **Below are the 2 tasks to be completed within the kubernetes cluster**

1. Create a config map object with the following specifications.
2. name -> page-config-map
3. namespace -> [student-name]
4. config data key-value pair:

PAGE\_CONTENT -> "Green-Pages from Yellow World!"

1. Edit the pages deployment in the namespace [student-name] & add an environment variable to the container
2. name -> PAGE\_CONTENT

value -> fetch it from configmap name page-config-map [key is PAGE\_CONTENT]

### **Implement the above tasks by creating manifest/yaml files**

1. Create deployment/pages-config.yaml
2. apiVersion: v1
3. data:
4. PAGE\_CONTENT: Green-Pages coming from Yellow-World!
5. kind: ConfigMap
6. metadata:
7. name: pages-config-map

namespace: [student-name]

1. Update deployment/pages-deployment.yaml
2. containers:
3. - image: [docker-username]/pages:[tag]
4. name: pages
5. ports:
6. - containerPort: 8080
7. env:
8. - name: PAGE\_CONTENT
9. valueFrom:
10. configMapKeyRef:
11. name: pages-config-map

key: PAGE\_CONTENT

1. Start minikube locally minikube start --driver=virtualbox
2. Verify the kubectl context kubectl config get-contexts is set to minikube. If not, set it to minikube kubectl config use-context minikube
3. Deploy and test the application in minikube. Refer to [Deployment Guide](https://k8s-1490992.cfapps.io/labs/04-Config-Start.html#_deployment_guide)

### **Deploy the application to production cluster**

1. Follow [Production Cluster Guide](https://k8s-1490992.cfapps.io/labs/EKS.html#login-section) to login/connect to the production cluster.
2. Deploy and test the application in production cluster. Refer to [Deployment Guide](https://k8s-1490992.cfapps.io/labs/04-Config-Start.html#_deployment_guide)
3. Commit code changes to the github repository

git add .

git commit -m "Externalized config"

git push -u origin master

## Deployment Guide

1. Set up [student-name] namespace to point to the current context

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

1. Create kubernetes objects

kubectl apply -f deployment/pages-config.yaml

kubectl apply -f deployment/pages-deployment.yaml

1. Verify the created objects

kubectl get deployment pages

kubectl get configmap pages-config-map

kubectl get pods -o wide

1. Access the pages application by port-forwarding using kubectl, enabling the application can be served via localhost on port 8080

kubectl port-forward svc/pages 8080:8080

curl localhost:8080

## Advanced usecase challenges

1. Enable refresh scope for automatic synchronization of config map properties in spring boot
2. Watch for changes in kubernetes api resources and objects, for dynamic loading
3. Security concerns

Lab 5

# Continuous Integration/Deployment

This lab demonstrates the concepts of continuous integration/continuous deployment (CI/CD) for our development. We will be leveraging Github Actions which comes integrated with Github, for automating the process of CI/CD.

## Learning Outcomes

After completing the lab, you will be able to:

1. Describe how to create pipeline using Github Actions
2. Continuously deploy your code to Kubernetes cluster

## Set-up Github Actions

1. Configure the jobs within the pipeline workflow.

Pipeline workflow:

Job sequence in the workflow:

1.build-artifact ---> 2.deploy-to-cluster

Tasks within each job:

1.build-artifact:

a.Build with Gradle

b.Upload Artifact

c.Build-Docker-Image

2.deploy-to-cluster:

a.Install Kubectl

b.Configure AWS credentials

c.Login to production cluster

d.Create K8s deployments and resources

1. Create the following secrets in github

DOCKER\_USERNAME

DOCKER\_PASSWORD

AWS\_ACCESS\_KEY\_ID

AWS\_SECRET\_ACCESS\_KEY

1. To add secrets, click on settings and select secrets from the left navigation menu within the github repository.
2. The aws access key id and secret will be provided to you.
3. Create .github/workflows/pipeline.yaml in the root project directory with the configuration. Refer to [pipeline.yaml](https://k8s-1490992.cfapps.io/labs/EKS-Pipeline.html" \l "eks)
4. Update the tag name in pages-deployment.yaml file to pipeline
5. Push your code to git repository
6. Navigate to github actions menu in the github dashboard to see the progress of the pipeline.
7. Upon successful completion of the workflow, you should be able to see the deployment and other objects in K8s cluster.

## Advanced Usecases

1. Automating security concerns
2. Automating code quality compliance
3. Advanced testing using test clusters

Lab 6

# Design and Develop Database Tier

The architecture of pages microservice consists of two major components, database tier and service tier. In thi lab, we will focus only on the database tier. The database tier will be developed and deployed first, which paves the way for developing the application/service tier.

We will be making using of database schema migrations for versioning our schema changes as our database evolves. There are various automation tools for implementing the database migrations. We will be using Flyway, as it is well adopted and widely accepted in the spring boot community.

## Learning Outcomes

After completing the lab, you will be able to:

1. Develop the Database Tier using MySql
2. Create MySql deployment in K8s
3. Expose MySql to be discovered by other services within the cluster
4. Implement Database Migrations using Flyway locally and on the production cluster
5. Leverage the concept of K8s Jobs for background processing

Before starting the lab, checkout the database-tier-start task.

git status

# Ensure the source code is checked in to github

# You can take a back up of your codebase, to keep your deployment files from being overwritten

# Checkout into a feature branch

git checkout database-tier-start -b db

# You are on branch db

|  |  |
| --- | --- |
|  | In case you get an error when you cherry-pick/check-out, open intellij, right-click on the project, select git → resolve-conflicts → View changes and merge them based on the differences |

## Creating the database locally

1. The updated codebase contains the refactored and re-organized pipeline. It also brings in database folder containing the DDL and DML scripts for creating the database and schema migrations. Take some time to walk-through them.
2. database/create\_database.sql contains sql script for creating the database pages and user pages\_user for developing and testing locally.
3. database/migrations/V1\_\_initial\_schema.sql contains the script for creating the schema for the tables.
4. Create the database locally.

mysql -uroot < database/create\_database.sql

They do not have any tables yet.

1. Run the flyway database migration command to create the schema

flyway -url="jdbc:mysql://localhost:3306/pages" -user=pages\_user -password=password -locations=filesystem:database migrate

1. Inspect the database created locally using mysql client.

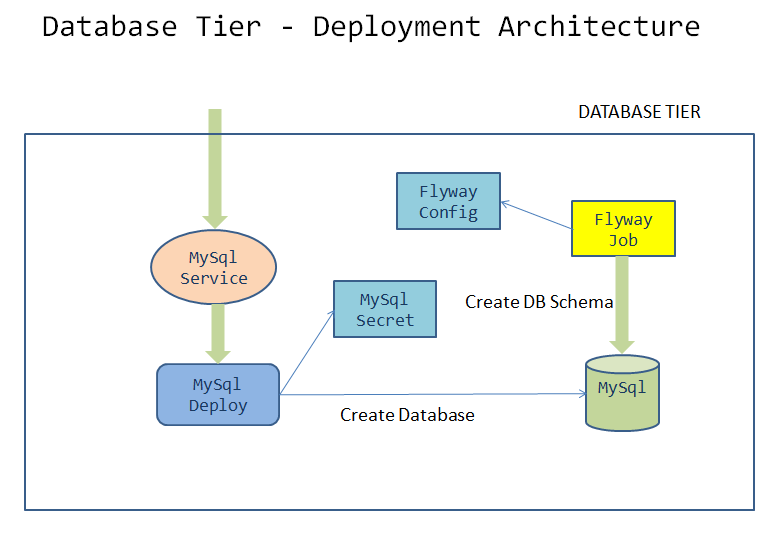
mysql -u pages\_user -p

Enter password: \*\*\*\*\*\*\*\*

use pages;

describe pages;

## Kubernetize the Database Tier



1. Create the file deployment/mysql-secret.yaml containing the manifest for secret in K8s with the following specs:

name-> mysql-pass

namespace -> [your namespace]

data ->

spring.datasource.password: [base64 encoded password]

password: [base64 encoded password]

For generating base64 encoded password use one of the 2 methods below: (Use the first method)

1.You can directly generate the secret in imperative way.

kubectl create secret generic mysql-pass --namespace=[student-name] --from-literal=spring.datasource.password=password --from-literal=password=password --dry-run=client -o yaml

Use the command: echo -n password | base64 and copy that into the yaml file if you want to write the yaml from scratch.

For more information refer to http://kubernetes.io/docs

1. Create the file deployment/mysql-deployment.yaml containing the manifest for mysql deployment in K8s with the following specs:

name -> mysql

namespace -> [your namespace]

labels-> app: pages

tier: database

selectors-> app: pages

tier: database

image -> mysql:8.0

container name -> mysql

expose port -> 3306

Create a volume called mysql-persistent-storage of type emptyDir.

Mount the volume at path -> mountPath: /docker-entrypoint-initdb.d

Add 5 environment variables to the container with key-value pair as follows:

MYSQL\_SERVICE\_HOST: "pages-mysql"

MYSQL\_SERVICE\_PORT: 3306

MYSQL\_DATABASE: pages

MYSQL\_USER: root

MYSQL\_ROOT\_PASSWORD: value from a secret called mysql-pass with key-name as password.

Constructing the deployment object in yaml file, as given in the above specification.

SOLUTION:

Generate the yaml file

kubectl create deploy mysql --image=mysql:8.0 --namespace=student-name --dry-run=client -o yaml

Copy the output onto the yaml file.

Edit/Update the labels & selectors as given in the specs.

Define volume under containers sections with name mysql-persistence-volume

and type -

emptyDir: {}

Mount the volume in the containers section. Use the name mysql-persistence-volume and mounthPath as given in the spec.

snippet for env section under containers:

env:

- name: MYSQL\_ROOT\_PASSWORD

valueFrom:

secretKeyRef:

name: mysql-pass

key: password

- name: MYSQL\_SERVICE\_HOST

value: "pages-mysql"

- name: MYSQL\_SERVICE\_PORT

value: "3306"

- name: MYSQL\_DATABASE

value: "pages"

- name: MYSQL\_USER

value: "root"

If you get stuck, read the documentation for creating a deployment, labels & selectors, volumes, volumemounts, configmaps, secrets and environment variables from http://kubernetes.io/docs

1. Create the deployment/mysql-service.yaml with the manifest for exposing the mysql deployment with following specs:

apiVersion: v1

kind: Service

metadata:

name: pages-mysql

namespace: student-name

labels:

app: pages

tier: database

spec:

ports:

- port: 3306

selector:

app: pages

tier: database

type: ClusterIP

1. Create the file deployment/flyway-configmap.yaml with thethe manifest for generating the config map in K8s with the following specs:

name-> flyway-configmap

namespace -> [your namespace]

data ->

spring.datasource.username -> root

V1\_\_inital\_schema.sql -> This should be the content of your database/migrations/V1\_\_inital\_schema.sql

Refer to https://kubernetes.io/docs/concepts/configuration/configmap/ on how k8s treats multiline values for a single key.

1. Create the file deployment/flyway-job.yaml with manifest for running a K8s job with the following specs:

apiVersion: batch/v1

kind: Job

metadata:

name: flyway-job

namespace: [student-name]

labels:

app: pages

spec:

template:

spec:

containers:

- name: flyway

image: flyway/flyway:6.4.4

args:

- info

- migrate

- info

env:

- name: FLYWAY\_URL

value: jdbc:mysql://pages-mysql/pages

- name: FLYWAY\_USER

value: root

- name: FLYWAY\_PASSWORD

valueFrom:

secretKeyRef:

name: mysql-pass

key: password

- name: FLYWAY\_PLACEHOLDER\_REPLACEMENT

value: "true"

- name: FLYWAY\_PLACEHOLDERS\_USERNAME

valueFrom:

configMapKeyRef:

name: flyway-configmap

key: spring.datasource.username

- name: FLYWAY\_PLACEHOLDERS\_PASSWORD

valueFrom:

secretKeyRef:

name: mysql-pass

key: spring.datasource.password

volumeMounts:

- mountPath: /flyway/sql

name: sql

volumes:

- name: sql

configMap:

name: flyway-configmap

restartPolicy: Never

## Testing locally

1. Switch the kubectl context to minikube and set the context to point to your namespace.
2. Create all the 5 resources inside your namespace. Wait for some time for the migration job to complete. Verify the resources were created without errors.
3. To verify the database was created with the table pages use kubectl exec to get a shell to the mysql container

kubectl get pods

#copy the name of mysql pod

kubectl exec -it <pod-name> -- sh

mysql -uroot -p

password

show databases;

use pages;

show tables;

describe pages;

exit ;

1. Verify scripts/deploy-to-k8s.sh is updated to create all the 5 resources
2. Update the docker image tag in pipeline.yaml and pages-deployment.yaml and verify they are same

## Deploy to the production cluster

1. In the pipeline.yml file update the branch to pick up the commit from the feature branch. Replace it with the branch name db
2. Pushing the souce code to github

git status

git add .

git commit -m "database-tier"

git push -u origin db --tags

1. Ci/CD pipeline will trigger the build and deployment. Wait for it to succeed.
2. Switch the kubectl context to production cluster pointing to your namespace
3. Verify the database is provisioned as per the deployment architecture. Use kubectl exec into mysql pod through the terminal and verify the table pages is created with 1 successful migration. Also, you can check the logs from the flyway job.

|  |  |
| --- | --- |
|  | The background job will not be deleted automatically. You will have to manually delete the flyway job by running kubectl delete job flyway-job |

### **The database tier is now ready to accept requests.**

Lab 7

# Design and Develop Service Tier

We will make use of the database tier created in the previous lab and implement our service tier to communicate with MySql database.

## Learning Outcomes

After completing the lab, you will be able to:

1. Develop the service-tier consisting of a single microservice using TDD approach.
2. Implement Repository pattern
3. Deploy the service tier
4. Understand native K8s service discovery by using labels and selectors to discover database service deployed in the previous lab.

Before starting the lab, checkout the service-tier-start task.

git status

# Ensure the source code is checked in to github

# You can take a back up of your codebase, to keep your deployment files from being overwritten

# Checkout into a feature branch

git checkout service-tier-start -b service

# You are on branch service

|  |  |
| --- | --- |
|  | In case you get an error when you cherry-pick/check-out, open intellij, right-click on the project, select git → resolve-conflicts → View changes and merge them based on the differences |

## Develop the pages microservice

1. Few test classes were added to the test package. IPagesRepository interface is provided to implement the repository pattern. Observe these changes in intellij. The source code will not compile at this stage.
2. The first step is to get all the test cases passing.
3. Create a class src/org/dell/kube/pages/Page with the below fields
4. public Long id;
5. public String businessName;
6. public Long categoryId;
7. public String address;

public String contactNumber;

Generate getters,setters & constructor/s as expected by the test classes

1. Create a repository class MySqlPageRepository which implements IPageRepository as expected by the test class MySqlPageRepositoryTest. Annotate the class with @Repository
2. Create PageController class and implement methods as expected by the test class PageApiTest. Inject the dependency of type MySqlPageRepository in the constructor of the controller class.

## Building locally

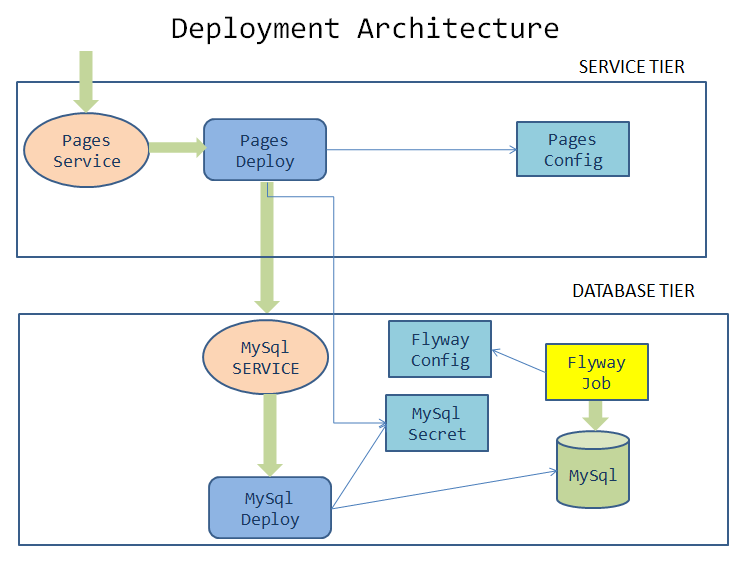
1. Ensure that you have created the database pages locally, which was done in the previous lab. Otherwise follow the previous lab instructions to create the database and run the flyway migrations for the pages table to be created.
2. Provide the spring datasource url in application.properties for both development and testing.
3. spring.datasource.url=jdbc:mysql://localhost:3306/pages?useSSL=false
4. spring.datasource.username=pages\_user

spring.datasource.password=password

We are using the same database for both development and testing for demonstration purposes. However, we recommend using separate database for development, testing and production in real projects.

1. Ensure that your application builds and all the test cases pass.

## Connecting service tier with database tier



## Kubernetize

1. The pages service needs to connect to the mysql database. The communication happens through mysql service that we created in the previous lab. However, the pages deployment should be aware of the spring datasource related properties.
2. Update the pages deployment by adding environment variables to the container
3. SPRING\_DATASOURCE\_URL -> jdbc:mysql://pages-mysql/pages
4. SPRING\_DATASOURCE\_USERNAME -> "root"

SPRING\_DATASOURCE\_PASSWORD -> value from secret name mysql-pass with key name password. This secret is already created in the previous lab.

With the above environment variables, spring boot can auto-configure the things necessary for database connectivity.

The updated env section looks like the below snippet.

env:

- name: PAGE\_CONTENT

valueFrom:

configMapKeyRef:

name: pages-config-map

key: PAGE\_CONTENT

- name: SPRING\_DATASOURCE\_URL

value: jdbc:mysql://pages-mysql/pages

- name: SPRING\_DATASOURCE\_USERNAME

value: "root"

- name: SPRING\_DATASOURCE\_PASSWORD

valueFrom:

secretKeyRef:

name: mysql-pass

key: password

1. Delete the pages deployment. This is not recommended in production, however we want to ensure that we have a fresh deployment for local testing. kubectl delete deploy pages
2. Build the docker image & push it to docker hub.

docker build -t [docker-username]/pages:service

docker push [docker-username]/pages:service

1. Update the pages deployment with the appropriate namespace, image & tag name.
2. Add label &/or selector tier=service (in addition to app=pages) in pages deployment yaml file at 3 places:

metadata.labels

spec.selector.matchLabels

spec.template.metadata.labels

1. Add label &/or selector tier=service (in addition to app=pages) in pages service yaml file at 2 places:

metadata.labels

spec.selector

1. Adding the labels as above not only allows the pages service to serve the incoming requests but also helps the pages-service to discover pages-mysql service using native K8s support, without adding any additional code. This enables the pages deployment to communicate with mysql deployment

## Testing locally on minikube

1. Switch the kubectl context to minikube - kubectl config use-context minikube

Set the kubectl context namespace to your namespace - kubectl config set-context --current --namespace [student-name]

1. Since the pages service is updated with new labels, updating the immutable propery of the service is not allowed. Delete the service

kubectl delete svc pages

1. Create the service

kubectl apply -f deployment/pages-service.yaml

1. Create pages deployment.

kubectl apply -f deployment/pages-deployment.yaml

1. Test the pages application by performing CRUD operations using curl/postman. Refer [Pages Curl Guide](https://k8s-1490992.cfapps.io/labs/07-Pages-Curl-Commands.html#pages-curl-section) for testing.
2. Update the docker image tag in pipeline.yaml and pages-deployment.yaml and verify they are same

## Deploy to the production cluster

1. Ensure that the tag names are the same in deployment and pipeline.
2. In the pipeline.yml file update the branch to pick up the commit from the feature branch. Replace it with the branch name service
3. Commit your changes and push them to github. The pipeline will deploy your new image to the production cluster.
4. Pushing the souce code to github
5. git status
6. git add .
7. git commit -m "service-tier"

git push -u origin service

1. Ci/CD pipeline will trigger the build and deployment. Wait for it to succeed.
2. Switch the kubectl context to production cluster pointing to your namespace
3. Port forward to connect to pages service running inside K8s from the local machine

kubectl port-forward svc/pages 8080:8080

1. Test the pages application by performing CRUD operations using curl/postman. Refer [Pages Curl Guide](https://k8s-1490992.cfapps.io/labs/07-Pages-Curl-Commands.html#pages-curl-section) for testing.

## Task Accomplished

We completed integrating the service-tier with the database-tier & successfully deployed a two-tier cloud native application to K8s cluster.

Lab 8

# Logging and Monitoring in Kubernetes

## Learning Outcomes

After completing the lab, you will be able to:

1. Use logback to implement application level logs
2. Externalize log messages
3. Configure application to use actuators
4. Configure observability using probes
5. Access container logs for debugging and troubleshooting

## Configure logback logging

We will make use of logback which is intendend to be a successor for slf4j. The choice for using logback also comes for the benefits that we want to use to add multiple appenders for our usecase. For more information on logback refer to [http://logback.qos.ch](http://logback.qos.ch/)

1. Add the following test method to HomeApiTest class

@Test

public void healthTest(){

String body = this.restTemplate.getForObject("/actuator/health", String.class);

assertThat(body).contains("UP");

}

1. Add the acutator dependency in build.gradle file and verify all the test cases pass.

implementation 'org.springframework.boot:spring-boot-starter-actuator'

1. We dont need to add any external dependecy for logback, as it comes as a part of web starter dependency.
2. Create a file called logback.xml in src/main/resources directory and update the file with the configuration as below. Make sure to replace student-name] with your namespace name created in K8s.

This example contains the basic logging configuration for FILE and STDOUT appender

<?xml version = "1.0" encoding = "UTF-8"?>

<configuration>

<appender name="FILE" class="ch.qos.logback.core.FileAppender">

<file>[student-name]/logs/app.log</file>

<encoder>

<pattern>%date %level [%thread] %logger{10} [%file:%line] %msg%n</pattern>

</encoder>

</appender>

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>%date %level [%thread] %logger{10} [%file:%line] %msg%n</pattern>

</encoder>

</appender>

<root level="info">

<appender-ref ref="FILE" />

<appender-ref ref="STDOUT" />

</root>

</configuration>

1. Add imports to refer to logback in PageController

import ch.qos.logback.classic.Logger;

import org.slf4j.LoggerFactory;

1. Create Logger instance in PageController

Logger logger =(Logger)LoggerFactory.getLogger(this.getClass());

1. Update methods in PageController to log messages upon invocation of CRUD operations.

Refer the code below for example:

@GetMapping("{id}")

public ResponseEntity<Page> read(@PathVariable long id) {

logger.info("READ-INFO:Fetching page with id = " + id);

logger.debug("READ-DEBUG:Fetching page with id = " + id);

Page page = pageRepository.read(id);

if(page!=null)

return new ResponseEntity<Page>(page, HttpStatus.OK);

else {

logger.error("READ-ERROR:Could not find page with id = " + id);

return new ResponseEntity(HttpStatus.NOT\_FOUND);

}

}

1. Verify the code builds and test cases pass.
2. Run the code. Perform crud operations using curl or postman. Verify the log messages are logged in console and in the external log file.
3. Delete the log file and the directory which was generated by logback as we dont want to checkin the auto-generated directory and file as part of the code.
4. Update the pages-deployment.yaml to add volume, volume mount and logger configuration as below

Replace [student-name] with namespace name

Create a volume with name -> node-dir

type -> hostpath

path -> /[student-name]

Create a volume mount with name -> node-dir

mountPath -> /[student-name]

Add environment variable with key:value pairs as below:

DEBUG: "true"

LOGGING\_FILE\_NAME: "[student-name]/logs/app.log"

LOGGING\_LEVEL\_ORG\_SPRINGFRAMEWORK\_WEB: debug

LOGGING\_LEVEL\_ROOT: debug

1. Update the pages-deployment.yaml file to enable the actuator endpoints
2. Add the environment variable:

MANAGEMENT\_ENDPOINTS\_WEB\_EXPOSURE\_INCLUDE: \*

1. Introduce Liveness probe to instruct the kubelet on when to kill the unhealthy container, subjected to its restart policy. Update the `pages-deployment.yaml file to add the below spec:
2. livenessProbe
3. type -> httpGet
4. path -> /actuator/health
5. port -> 8080
6. initialDelaySeconds -> 15

periodSeconds -> 30

1. Introduce Readiness probe which indicates whether the application is ready to respond to requests. Update the pages-deployment manifest to add the below spec:
2. readinessProbe
3. type -> tcpSocket
4. port -> 8080
5. initialDelaySeconds -> 15

periodSeconds -> 30

1. Dockerize the application by creating a new tag monitor. Test it once on minikube with the new image. By now, you should be familiar with dockerizing and kubernetizing. If not, refer back to the earlier lab instructions.

### **Reference Code Snippet for pages-deployment manifest**

env:

- name: PAGE\_CONTENT

valueFrom:

configMapKeyRef:

name: pages-config-map

key: PAGE\_CONTENT

- name: SPRING\_DATASOURCE\_URL

value: jdbc:mysql://pages-mysql/pages?useSSL=false

- name: SPRING\_DATASOURCE\_USERNAME

value: "root"

- name: SPRING\_DATASOURCE\_PASSWORD

valueFrom:

secretKeyRef:

name: mysql-pass

key: password

- name: DEBUG

value: "true"

- name: LOGGING\_FILE\_NAME

value: "student-name/logs/app.log"

- name: LOGGING\_LEVEL\_ORG\_SPRINGFRAMEWORK\_WEB

value: debug

- name: LOGGING\_LEVEL\_ROOT

value: debug

- name: MANAGEMENT\_ENDPOINTS\_WEB\_EXPOSURE\_INCLUDE

value: "\*"

volumeMounts:

- name: node-dir

mountPath: /[student-name]

readinessProbe:

tcpSocket:

port: 8080

initialDelaySeconds: 15

periodSeconds: 30

livenessProbe:

httpGet:

path: /actuator/health

port: 8080

initialDelaySeconds: 15

periodSeconds: 30

volumes:

- name: node-dir

hostPath:

path: /[student-name]

## Viewing Logs in K8s

1. Ensure kubectl context is set to minikube.
2. Use kubectl logs [pod-name] for fetching application logs. Refer to <http://kubernetes.io/docs> for more information.
3. Type the command minikube dashboard , which can also be used to view the logs.

## Deploy

1. Change the value of tag in pipeline.yaml to match the pages-deployment.yaml with the tag name monitor.
2. Push the code to github repository to start the pipeline, which will deploy the application to the production cluster.

## Debug and troubleshoot

1. Delete and recreate the mysql-deployment

Ensure your kubectl context is set to minikube.

kubectl config current-context

kubectl config get-contexts

kubectl config use-context minikube

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

kubectl delete deploy mysql

kubectl apply -f deployment/mysql-deployment.yaml

kubectl get deploy

1. Try to access the pages application
2. kubectl port-forward svc/pages 8080:8080

curl http://locahost:8080/pages

1. What error do you see?
2. Find out the root cause of the problem and fix it.

## Advanced usecase challenges

1. Implement centralized logging by aggregating the logs and persisting it from all the pods and nodes into a single datastore.
2. Add instrumentation such as counters and guages
3. Implement correlation of events by using correlation ids for all the requests
4. Implement service mesh pattern for designing a unified logging layer.
5. In order to view the logs using Kibana dashboard with EFK on minikube, enable EFK addons on minikube.

Ensure your kubectl context is set to minikube.

minikube addons list

minikube addons enable efk

Explore the Kibana dashboard

minikube addons open efk

Use \* when prompted to create index pattern. Select timestamp for filtering and click ok. From the left side menu, select Discover and use queries for searching your log messages. Note that, this addon might consume more memory and might not start as expected which is a known issue.

Lab 9

# Scalability & Availability

## Learning Outcomes

After completing the lab, you will be able to:

1. Configure Memory and CPU Quotas for a Namespace
2. Scale application vertically & horizontally
3. Implement storage using Persistent Volume and Persistent Volume Claims
4. Implement AutoScaling

## Configuring resource quota for a namespace

1. To set the desired hard limits for each named resource, create a resource quota object using below specification.
2. name -> resource-quota
3. namespace -> [student-name]
4. limits:
5. cpu -> 2
6. memory -> 20Gi

pods -> 10

**Solution**

Create deployment/resource-quota.yaml file with the following content. Replace [student-name] with your namespace name.

apiVersion: v1

kind: ResourceQuota

metadata:

name: resource-quota

namespace: [student-name]

spec:

hard:

cpu: 2

memory: 20Gi

pods: 10

1. Ensure kubectl context is set to minikube with the right namespace
2. Run the below command

kubectl apply -f deployment/resource-quota.yaml

1. Verify that the quota is created in the appropriate namespace and the limits are set as specified in the specification
2. kubectl get quota

kubectl describe quota resource-quota

1. Delete deployments and pods
2. kubectl delete deploy pages
3. kubectl delete deploy mysql

kubectl delete job flyway-job

1. Create mysql deployment

kubectl apply -f deployment/mysql-deployment.yaml

1. Check the status of mysql deployment. Have the pods been scheduled? How do you debug and troubleshoot this error?
2. Check the logs and events of mysql

kubectl describe deploy mysql

kubectl get events --sort-by=.metadata.creationTimestamp

1. Error might look something like this:

failed quota: resource-quota: must specify cpu,memory

1. In order to fix this, we need to specify the compute requests in all resources needing them. This applies to all pods and deployments which has compute resource requirements. In our case we have 2 deployments and 1 job .
2. We need to add the resource requests to the container section of pages-deployment, mysql-deployment and flyway-job to include the resource requests. Note that, if you request for more than available resources, your pods will not get scheduled.
3. resources:
4. requests:
5. memory: 500Mi
6. cpu: 0.25
7. limits:
8. memory: 900Mi

cpu: 1

1. Create the deployments and run the flyway job.
2. kubectl apply -f deployment/mysql-deployment.yaml
3. kubectl apply -f deployment/flyway-job.yaml

kubectl apply -f deployment/pages-deployment.yaml

1. Pages deployment will take some time to startup. Confirm the pod has started successfully before moving on to the next step.
2. Test the pages application by navigating to <http://localhost:8080/pages> after port-forwarding using kubectl on port 8080.

## Vertical Scaling

1. Scale the pages application vertically, by adding 100 Mi to memory.
2. Update the pages-deployment.yaml file with the new request values and re-deploy pages.
3. resources:
4. requests:
5. memory: 600Mi #old value 500Mi
6. cpu: 0.25
7. limits:
8. memory: 900Mi

cpu: 1

Alternatively, you can edit the deployment imperatively like:

kubectl edit deploy pages

vi-editor opens up automatically with the live deployment file which can be edited. Use this option if you are well versed in using vi editor and save it to see the configuration updated.

## Horizontal Scaling

1. Scale out the pages application horizontally, by increasing the number of replicas to 2.
2. There are 2 ways of acheiving this:
   1. Update the Deployment.spec.replicas field of pages-deployment.yaml file and then using the command kubectl apply -f deployment/pages-deployment.yaml
   2. Imperative way: kubectl scale deploy pages --replicas=2
3. Let’s use the imperative way, which is much faster.
4. Verify the number of running pods for pages after scaling. This might take a while.

kubectl get deploy pages

kubectl port-forward svc/pages 8080:8080

curl <http://localhost:8080/pages>

1. Scale in the pages application to 1 replica.

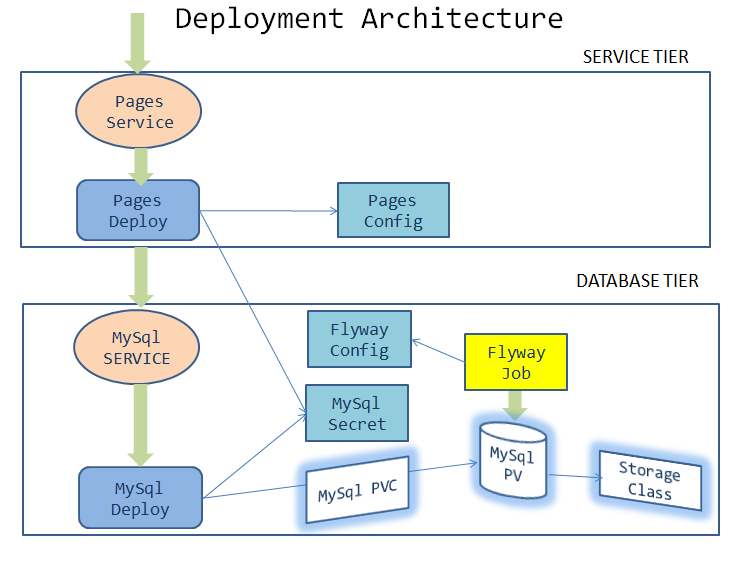
kubectl scale deploy pages --replicas=1

## Peristent volume and persistent volume claims

**Understanding the scenario**

1. POST few entries into pages application.
2. Delete the mysql pod. Since we have the replicaset, the pod will be recreated to maintain the desired state of the deployment.
3. Wait till the pod gets recreated. Issue a GET request to the pages controller.
4. What happened? You need to run the migration job again.
5. Wait till the job is completed. Issue a GET request to the pages controller.
6. What happened to the data that was POST’ed?

**Need to refactor the design for persistence of data**



We have used volumes in our labs. They allowed us to mount a storage unit such as file system folder on a node or a cloud storage bucket and also share information between nodes.These regular volumes are evicted when the pod is evicted.

Kubernetes provides a persistent storage solution for containers called Persistent Volumes (PV). Persistent Volume can remain alive for as long as necessary for ongoing operations.

1. Persistence Volume normally makes use of StorageClass objects in K8s, creating an abstraction to connect to various storage solutions for multiple storage requirements.

For more information, refer <https://kubernetes.io/docs/concepts/storage/persistent-volumes>

1. Create storage class object definition in deployment/mysql-storage-class.yaml as given below.
2. kind: StorageClass
3. apiVersion: storage.k8s.io/v1
4. metadata:
5. name: database
6. labels:
7. addonmanager.kubernetes.io/mode: EnsureExists
8. provisioner: k8s.io/minikube-hostpath
9. reclaimPolicy: Retain

volumeBindingMode: Immediate

1. Create persistent volume definition in deployment/mysql-pv.yaml as given below. Replace [student-name] with your namespace name at 2 places.
2. apiVersion: v1
3. kind: PersistentVolume
4. metadata:
5. name: mysql-pv
6. namespace: [student-name]
7. labels:
8. type: local
9. spec:
10. storageClassName: database
11. capacity:
12. storage: 10Gi
13. accessModes:
14. - ReadWriteMany
15. hostPath:

path: "/var/lib/mysql/[student-name]"

1. We can now request for the persistent volume, by creating a Persistent Volume Claim object with below specification in deployments/mysql-pvc.yaml. Replace [student-name] with your namespace name.
2. apiVersion: v1
3. kind: PersistentVolumeClaim
4. metadata:
5. name: mysql-pvc
6. namespace: [student-name]
7. spec:
8. accessModes:
9. - ReadWriteMany
10. resources:
11. requests:

storage: 1Gi

1. Edit and update the volume and volume mount section of mysql-deployment.yaml file to use the persistent volume claim defined earlier.
2. ---
3. volumeMounts:
4. - name: mysql-persistent-storage
5. mountPath: /var/lib/mysql
6. volumes:
7. - name: mysql-persistent-storage
8. persistentVolumeClaim:
9. claimName: mysql-pvc

---

1. Update the environment variable for spring datasource url in pages-deployment.yaml file with the below value
2. - name: SPRING\_DATASOURCE\_URL

value: jdbc:mysql://pages-mysql/pages?allowPublicKeyRetrieval=true&useSSL=false

1. Delete deployments and job
2. kubectl delete deploy pages
3. kubectl delete deploy mysql

kubectl delete job flyway-job

1. Create storage related objects in minikube
2. kubectl apply -f deployment/mysql-storage-class.yaml
3. kubectl apply -f deployment/mysql-pv.yaml
4. kubectl apply -f deployment/mysql-pvc.yaml
5. kubectl get storageclasses
6. kubectl get pv

kubectl get pvc

1. Create all the deployments and job in minikube
2. kubectl apply -f deployment/mysql-deployment.yaml
3. kubectl apply -f deployment/flyway-job.yaml

kubectl apply -f deployment/pages-deployment.yaml

1. Testing on minikube
2. kubectl get deploy
3. kubectl get jobs
4. kubectl get pods
5. #Delete the job as you dont need it after completion.
6. kubectl delete job flyway-job
7. kubectl port-forward svc/pages 8080:8080
8. curl -i -XPOST -H"Content-Type: application/json" localhost:8080/pages -d"{\"businessName\": \"Uber\", \"address\": \"SanFrancisco, CA, USA\", \"categoryId\": 123, \"contactNumber\": \"0045987869\"}"
9. curl localhost:8080/pages
10. #Stop the port forwarding by presing CTRL+C
11. #Let's delete the mysql deployment and recreate it
12. kubectl delete deploy mysql
13. curl localhost:8080/pages
14. #What's the error?
15. #Pages app will be DOWN, as the health checks will fail, and the kubelet will kill and restart the container. See the logs and events.
16. #Let's create the mysql deployment. This time, we will not run the flyway migration job.
17. kubectl apply -f deployment/mysql-deployment.yaml
18. kubectl get deploy
19. kubectl get pods
20. #Verify that all the pods are running.
21. kubectl port-forward svc/pages 8080:8080
22. curl localhost:8080/pages

#Was the data persistent? Were you able to get all the entries added earlier?

## Automatically scale your deployment

1. Before proceeding further, let’s delete the resource quota that we created earlier kubectl delete quota resource-quota
2. Create Horizontal Pod Autoscaler with the following spec:
3. pages deployment:
4. minimum number of replicas -> 1
5. maximum number of replicas -> 2

dependent metric is cpu-percent -> 50

For help refer to https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough

**Solution**

kubectl autoscale deployment pages --cpu-percent=50 --min=1 --max=10

kubectl get hpa

1. Ensure you have a single replica of pages deployment running. Create a pod with busybox image to generate lot of traffic to the pages application in an infinite loop.
2. kubectl run -it --rm load-generator --image=busybox /bin/sh
3. Hit enter for command prompt

while true; do wget -q -O- http://pages:8080/pages; done

1. Check the replicas and the hpa now
2. kubectl get deploy

kubectl get hpa

1. Stop the load-generator and verify the automatic scaling down of pages due to decreased cpu utilization.
2. kubectl get deploy
3. kubectl get hpa
4. # Delete HPA as we dont need it anymore

kubectl delete hpa

## Additional tasks to explore:

1. Performing a rolling update
2. Updating the deployment with a new version
3. Roll back to a previous version of deployment

You can do the tasks given in this link to understand the concepts of the rolling updates and roll back <https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-interactive>

Lab 10

# Category microservice

In this lab we will look into an existing brown field category microservice and deploy it to the K8s cluster.

## Learning Outcomes

After completing the lab, you will be able to:

1. Understand working with brown-field applications.
2. Lift and shift a brown-field application into K8s cluster
3. Deploy category microservice without modifying its functionality

Download the codebase [category](https://dell-edu-lab-store.s3.ap-south-1.amazonaws.com/repository/category.zip) into workspace directory.

Create a repository called category in your GitHub account. Add this repository as a remote called origin of your local repository.

We will start by pushing the initial commit to GitHub, complete with the start and solutions tags.

git push origin master --tags

Before starting the lab, checkout the distributed-start tag into a new feature branch.

git checkout distributed-start -b category-wip

## Category microservice design & implementation

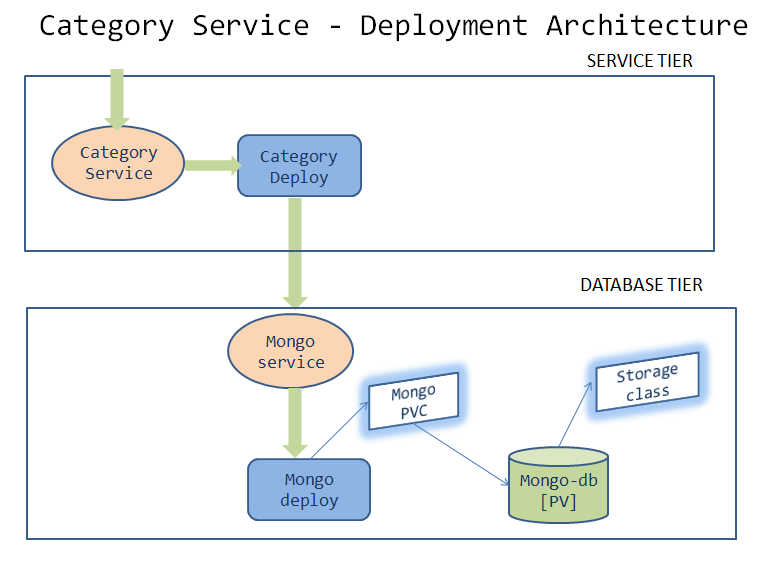
1. Open the source code in intellij.
2. Take time to do a code walkthrough and understand the design and functionality of category micoservice.
3. The service uses mongodb for persisting the categories.
4. Remember, this is a brown field application which is already developed and your main goal is to lift and shift to K8s cluster.
5. Build the source code and test it using curl/postman.

./gradlew bootRun

1. Refer [Curl Guide](https://k8s-1490992.cfapps.io/labs/10-Category-Curl-Commands.html#category-curl-section) for testing and proceed with the next steps
2. Build the jar file and dockerize the category service
3. ./gradlew clean build
4. docker build -t [docker-username]/category:distributed .

docker push [docker-username]/category:distributed

## Deploying category microservice to K8s



1. Observe the deployments directory, which contains the manifest files for K8s Deployments
2. Walkthrough the yaml files & understand the solution to the deployment architecture.
3. Before we start deploying, replace [student-name] with your namespace in all the yaml files. Also, update the image name in the category deployment with [docker-username]/category:distributed replacing with your docker user name
4. We will first deploy our application on minikube and then deploy it to the production cluster

### **Deploy and test locally using minikube**

1. Start minikube locally minikube start --driver=virtualbox
2. Verify the kubectl context kubectl config get-contexts is set to minikube. If not, set it to minikube kubectl config use-context minikube
3. Follow the [Deployment Guide](https://k8s-1490992.cfapps.io/labs/10-Category-Start.html#_deployment_guide) to deploy in the minikube and test the application locally.

### **Deploy and test in the production cluster**

1. Verify the kubectl context kubectl config get-contexts is set to production cluster. If not, set it to the production cluster kubectl config use-context [cluster-name]
2. Follow the [Deployment Guide](https://k8s-1490992.cfapps.io/labs/10-Category-Start.html#_deployment_guide) to deploy and test the application in production

## Deployment Guide

1. 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]

1. Create the Database tier
2. kubectl apply -f deployment/mongo-storage-class.yaml
3. kubectl apply -f deployment/mongo-pv.yaml
4. kubectl apply -f deployment/mongo-pvc.yaml
5. kubectl apply -f deployment/mongo-service.yaml

kubectl apply -f deployment/mongo-deployment.yaml

1. Verify the deployment of database tier
2. kubectl get deployment mongo
3. kubectl get service mongo

kubectl get pvc

1. Proceed further if there are no errors, otherwise troubleshoot and fix them.
2. Create the service tier
3. kubectl apply -f deployment/category-service.yaml

kubectl apply -f deployment/category-deployment.yaml

1. Verify the deployment of service tier
2. kubectl get deployment category

kubectl get service category

1. Access the category application

kubectl port-forward svc/category 8080:8080

1. Refer [Curl Guide](https://k8s-1490992.cfapps.io/labs/10-Category-Curl-Commands.html#category-curl-section) for testing and proceed with the next steps
2. Commit code changes to the github repository
3. git add .
4. git commit -m "Category Start"

git push -u origin category-start:master

## Task Accomplished

We successfully deployed a 2 tier category microservice application to K8s cluster.

Lab 11

# 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.

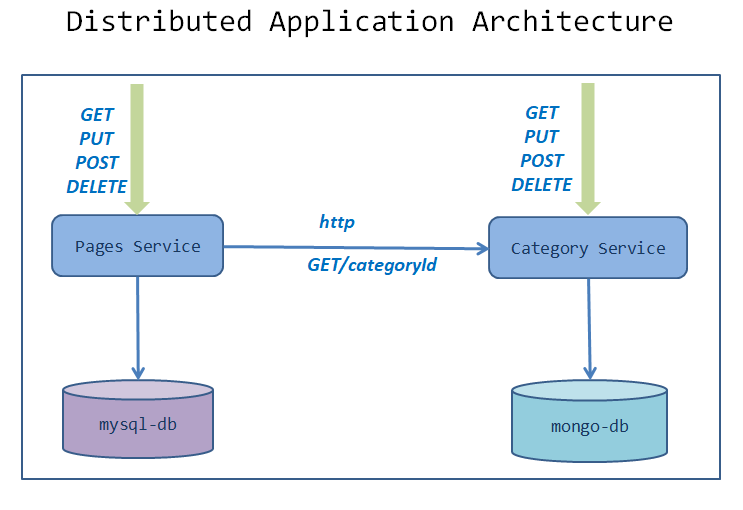
In order to accomplish our goal, we will make use of Spring Cloud Kubernetes for service discovery and implement client side loadbalancing using Ribbon. We will be refactoring pages and category microservice to enable service to service communication.

## 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. Update build.gradle file.
4. buildscript {
5. dependencies {
6. classpath "io.spring.gradle:dependency-management-plugin:1.0.9.RELEASE"
7. }
8. }
9. plugins {
10. id 'org.springframework.boot' version '2.3.1.RELEASE'
11. id 'io.spring.dependency-management' version '1.0.9.RELEASE'
12. id 'java'
13. }
14. apply plugin: "io.spring.dependency-management"
15. dependencyManagement {
16. imports {
17. mavenBom 'org.springframework.cloud:spring-cloud-dependencies:2020.0.0-M2'
18. }
19. }
20. repositories {
21. mavenCentral()
22. maven {
23. url 'https://repo.spring.io/milestone'
24. }
25. }
26. dependencies {
27. implementation 'org.springframework.boot:spring-boot-starter-web'
28. implementation 'org.springframework.boot:spring-boot-starter-data-mongodb'
29. implementation 'org.springframework.cloud:spring-cloud-starter-kubernetes'
30. testImplementation('org.springframework.boot:spring-boot-starter-test') {
31. exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
32. }
33. }
34. test {
35. useJUnitPlatform()

}

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

spring.application.name=category

1. Annotate CategoryApplication class with @EnableDiscoveryClient from the package org.springframework.cloud.client.discovery.EnableDiscoveryClient
2. Build the application and test it locally before deployment. Refer [Category Curl Guide](https://k8s-1490992.cfapps.io/labs/10-Category-Curl-Commands.html#category-curl-section)

## 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. Update build.gradle file.
4. buildscript {
5. dependencies {
6. classpath "io.spring.gradle:dependency-management-plugin:1.0.9.RELEASE"
7. }
8. }
9. plugins {
10. id 'org.springframework.boot' version '2.3.1.RELEASE'
11. id 'io.spring.dependency-management' version '1.0.9.RELEASE'
12. id 'java'
13. }
14. apply plugin: "io.spring.dependency-management"
15. dependencyManagement {
16. imports {
17. mavenBom 'org.springframework.cloud:spring-cloud-dependencies:2020.0.0-M2'
18. }
19. }
20. group = 'com.example'
21. repositories {
22. mavenCentral()
23. maven {
24. url 'https://repo.spring.io/milestone'
25. }
26. }
27. bootRun.environment([
28. "PAGE\_CONTENT": "YellowPages",
29. ])
30. test.environment([
31. "PAGE\_CONTENT": "YellowPages",
32. ])
33. dependencies {
34. implementation 'org.springframework.boot:spring-boot-starter-jdbc'
35. implementation 'org.springframework.boot:spring-boot-starter-web'
36. implementation 'org.springframework.boot:spring-boot-starter-actuator'
37. implementation 'org.springframework.cloud:spring-cloud-starter-kubernetes:1.1.3.RELEASE'
38. implementation 'org.springframework.cloud:spring-cloud-starter-netflix-ribbon'
39. implementation 'org.springframework.cloud:spring-cloud-starter-kubernetes-ribbon:1.1.1.RELEASE'
40. implementation 'org.springframework.cloud:spring-cloud-starter-openfeign:2.2.3.RELEASE'
41. implementation 'org.springframework.cloud:spring-cloud-commons'
42. implementation 'mysql:mysql-connector-java:8.0.12'
43. testImplementation('org.springframework.boot:spring-boot-starter-test') {
44. exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
45. }
46. }
47. test {
48. useJUnitPlatform()

}

1. Annotate PageApplication class with @EnableDiscoveryClient and @EnableFeignClients.
2. 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.
3. Create a class Category.java within src/main/java
4. package org.dell.kube.pages;
5. public class Category {
6. private Long id;
7. private String categoryName;
8. private String description;
9. public Long getId() {
10. return id;
11. }
12. public void setId(Long id) {
13. this.id = id;
14. }
15. public String getCategoryName() {
16. return categoryName;
17. }
18. public void setCategoryName(String categoryName) {
19. this.categoryName = categoryName;
20. }
21. public String getDescription() {
22. return description;
23. }
24. public void setDescription(String description) {
25. this.description = description;
26. }
27. @Override
28. public String toString() {
29. return "Category{" +
30. "id=" + id +
31. ", categoryName='" + categoryName + '\'' +
32. ", description='" + description + '\'' +
33. '}';
34. }

}

1. Create an interface CategoryClient.java within src/main/java. This interface is a declarative REST service interface at Pages client.
2. package org.dell.kube.pages;
3. import org.springframework.cloud.openfeign.FeignClient;
4. import org.springframework.web.bind.annotation.GetMapping;
5. import org.springframework.web.bind.annotation.PathVariable;
6. @FeignClient(name = "category")
7. public interface CategoryClient {
8. @GetMapping("/category/{categoryId}")
9. Category findCategory(@PathVariable("categoryId") Long categoryId);

}

1. Inject categoryClient dependency in the PageController class
2. @Autowired

CategoryClient categoryClient;

1. Update the create/POST method of PageController to.
   1. Invoke Category service to validate the categoryId.
   2. Upon successful validation, make a POST request to Pages service.
   3. Handle the FeignClient exception which is raised if the category is not found.
2. ......
3. @PostMapping
4. public ResponseEntity<Page> create(@RequestBody Page page) {
5. logger.info("CREATE-INFO:Creating a new page");
6. logger.debug("CREATE-DEBUG:Creating a new page");
7. Category category = null;
8. try {
9. category = categoryClient.findCategory(page.getCategoryId());
10. }
11. catch(FeignException ex){
12. if(ex.getMessage().contains("404")) {
13. return new ResponseEntity<>(HttpStatus.NOT\_FOUND);
14. }
15. else{
16. return new ResponseEntity<>(HttpStatus.INTERNAL\_SERVER\_ERROR);
17. }
18. }
19. if(category ==null || category.getId()==null) {
20. return new ResponseEntity<>(HttpStatus.NOT\_FOUND);
21. }
22. else
23. {
24. Page newPage = pageRepository.create(page);
25. logger.info("CREATE-INFO:Created a new page with id = " + newPage.id);
26. logger.debug("CREATE-DEBUG:Created a new page with id = " + newPage.id);
27. return new ResponseEntity<Page>(newPage, HttpStatus.CREATED);
28. }
29. }

......

1. 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
2. docker build -t [docker-username]/category:distributed .
3. docker push [docker-username]/category:distributed
4. docker build -t [docker-username]/pages:distributed .

docker push [docker-username]/pages:distributed

1. 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.
2. 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.
3. kind: ClusterRole
4. apiVersion: rbac.authorization.k8s.io/v1
5. metadata:
6. name: [student-name]-cluster-role
7. rules:
8. - apiGroups: [""] # "" indicates the core API group
9. resources: ["services", "pods", "configmaps", "endpoints"]
10. verbs: ["get", "watch", "list"]
11. ---
12. kind: RoleBinding
13. apiVersion: rbac.authorization.k8s.io/v1
14. metadata:
15. name: default:[student-name]-cluster-role-binding
16. roleRef:
17. apiGroup: rbac.authorization.k8s.io
18. kind: ClusterRole
19. name: [student-name]-cluster-role
20. subjects:
21. - kind: ServiceAccount
22. 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

1. 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]

1. Create the Database tier
2. kubectl apply -f deployment/mongo-storage-class.yaml
3. kubectl apply -f deployment/mongo-pv.yaml
4. kubectl apply -f deployment/mongo-pvc.yaml
5. kubectl apply -f deployment/mongo-service.yaml

kubectl apply -f deployment/mongo-deployment.yaml

1. Verify the deployment of database tier
2. kubectl get deployment mongo
3. kubectl get service mongo

kubectl get pvc

1. Proceed further if there are no errors, otherwise troubleshoot and fix them.
2. Create the service tier
3. kubectl apply -f deployment/category-service.yaml

kubectl apply -f deployment/category-deployment.yaml

1. Verify the deployment of service tier
2. kubectl get deployment category

kubectl get service category

1. Access the category application

kubectl port-forward svc/category 8080:8080

1. Refer [Category Curl Guide](https://k8s-1490992.cfapps.io/labs/10-Category-Curl-Commands.html#category-curl-section) for testing and proceed with the next steps

### **Deploy Pages microservice**

1. Create the Database tier
2. kubectl apply -f deployment/rbac.yaml
3. kubectl apply -f deployment/mysql-storage-class.yaml
4. kubectl apply -f deployment/mysql-pv.yaml
5. kubectl apply -f deployment/mysql-pvc.yaml
6. kubectl apply -f deployment/mysql-service.yaml
7. kubectl apply -f deployment/mysql-secret.yaml
8. kubectl apply -f deployment/mysql-deployment.yaml
9. kubectl apply -f deployment/flyway-configmap.yaml

kubectl apply -f deployment/flyway-job.yaml

1. Verify the deployment of database tier
2. kubectl get deployment mysql
3. kubectl get service mysql
4. kubectl get pvc

kubectl get jobs

1. Create the Service tier
2. kubectl apply -f deployment/pages-config.yaml
3. kubectl apply -f deployment/pages-service.yaml

kubectl apply -f deployment/pages-deployment.yaml

1. Verify the deployment of database tier
2. kubectl get deploy

kubectl get svc

1. Proceed further if there are no errors, otherwise troubleshoot and fix them.
2. Connect to the pages service by port-forwarding for testing. kubectl port-forward svc/pages 8080:8080
3. Test the pages application by performing CRUD operations using curl/postman. Refer [Pages Curl Guide](https://k8s-1490992.cfapps.io/labs/07-Pages-Curl-Commands.html#pages-curl-section) for testing.
4. 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 use cases

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