Weekly Assessment-5

Practical Question:

**Q1. How do you include Eureka Service discovery server and service?**

To include Eureka Server in a project, use the starter with a group ID of org.springframework.cloud and an artifact ID of spring-cloud-starter-netflix-eureka-server.

or include eureka dependencies when creating the spring project.

**Q2. How to Implement Retry Circuit Breaker in Microservices Application?**

To include Eureka Server in a project, use the starter with a group ID of org.springframework.cloud and an artifact ID of spring-cloud-starter-circuitbreaker-resilience4j.

or include the resilience4j dependencies when creating the spring project.

**Q3. What does one mean by Service Registration and Discovery? How is it implemented in**

**Spring Cloud?**

The process of a service registering its location in a central registry. It usually registers its host and port and sometimes authentication credentials, protocols, versions numbers, and/or environment details is service registering.

The process of a client application querying the central registry to learn of the location of services is service discovery.

To implement Eureka Server in a project, use the starter with a group ID of org.springframework.cloud and an artifact ID of spring-cloud-starter-netflix-eureka-server.

or include eureka dependencies when creating the spring project.

**Q4. Develop a restful webservice to perform CRUD operations. Entities should have Student,**

**Courses and Teachers.**

**Program:**

**Courses Class:**

package com.springboot\_task.models;

import java.io.Serializable;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

public class Courses implements Serializable {

/\*\*

\*

\*/

private static final long serialVersionUID = -1772978403863902891L;

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long courseId;

@Column(name = "CourseName", nullable = true)

private String courseName;

public Long getCourseId() {

return courseId;

}

public void setCourseId(Long courseId) {

this.courseId = courseId;

}

public String getCourseName() {

return courseName;

}

public void setCourseName(String courseName) {

this.courseName = courseName;

}

public static long getSerialversionuid() {

return serialVersionUID;

}

@Override

public String toString() {

return "Courses [courseId=" + courseId + ", courseName=" + courseName + "]";

}

}

**Student Class:**

package com.springboot\_task.models;

import java.io.Serializable;

import java.util.List;

import javax.persistence.CascadeType;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.JoinColumn;

import javax.persistence.OneToMany;

import javax.persistence.OneToOne;

import javax.persistence.Table;

@Entity

@Table(name = "student")

public class Student implements Serializable {

/\*\*

\*

\*/

private static final long serialVersionUID = 1064153100716867148L;

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long rollNumber;

@Column(name = "StudentName", nullable = true)

private String name;

@Column(name = "PhoneNumber", nullable = true)

private Integer phoneNumber;

@OneToOne(cascade = CascadeType.ALL)

private Courses course;

@OneToMany(cascade = CascadeType.ALL)

@JoinColumn(name ="student\_roll\_number")

private List<Teachers> teacher;

public Long getRollNumber() {

return rollNumber;

}

public void setRollNumber(Long rollNumber) {

this.rollNumber = rollNumber;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Integer getPhoneNumber() {

return phoneNumber;

}

public void setPhoneNumber(Integer phoneNumber) {

this.phoneNumber = phoneNumber;

}

public Courses getCourse() {

return course;

}

public void setCourse(Courses course) {

this.course = course;

}

public List<Teachers> getTeachers() {

return teacher;

}

public void setTeachers(List<Teachers> teachers) {

this.teacher = teachers;

}

public static long getSerialversionuid() {

return serialVersionUID;

}

@Override

public String toString() {

return "Student [rollNumber=" + rollNumber + ", name=" + name + ", phoneNumber=" + phoneNumber + ", course="

+ course + ", teachers=" + teacher + "]";

}

}

**Teachers Class:**

package com.springboot\_task.models;

import java.io.Serializable;

import javax.persistence.CascadeType;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.JoinColumn;

import javax.persistence.ManyToOne;

import javax.persistence.Table;

@Entity

@Table(name = "Teachers")

public class Teachers implements Serializable {

/\*\*

\*

\*/

private static final long serialVersionUID = 3751001576456959207L;

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long teacherId;

@Column(name = "TeacherName", nullable = true)

private String name;

@Column(name = "PhoneNumber", nullable = true)

private Integer phoneNumber;

@ManyToOne(cascade = CascadeType.MERGE)

@JoinColumn(name ="student\_roll\_number")

private Student student;

public Long gettId() {

return teacherId;

}

public void settId(Long tId) {

this.teacherId = tId;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Integer getPhoneNumber() {

return phoneNumber;

}

public void setPhoneNumber(Integer phoneNumber) {

this.phoneNumber = phoneNumber;

}

public Student getStudent() {

return student;

}

public void setStudent(Student student) {

this.student = student;

}

public static long getSerialversionuid() {

return serialVersionUID;

}

@Override

public String toString() {

return "Teachers [tId=" + teacherId + ", name=" + name + ", phoneNumber=" + phoneNumber + ", student=" + student

+ "]";

}

}

**Courses Repository:**

package com.springboot\_task.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

import com.springboot\_task.models.Courses;

@Repository

public interface CoursesRepository extends JpaRepository<Courses, Long>{

}

**Student Repository:**

package com.springboot\_task.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

import com.springboot\_task.models.Student;

@Repository

public interface StudentsRepository extends JpaRepository<Student, Long>{

}

**Teachers Repository:**

package com.springboot\_task.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

import com.springboot\_task.models.Teachers;

@Repository

public interface TeachersRepository extends JpaRepository<Teachers, Long>{

}

**Courses RestController:**

package com.springboot\_task.rest\_controller;

import java.util.List;

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

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

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

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

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

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

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

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

import com.springboot\_task.models.Courses;

import com.springboot\_task.repository.CoursesRepository;

@RestController

@RequestMapping("/course")

public class CoursesRestController {

@Autowired

private CoursesRepository coursesRepository;

@PostMapping("/create")

public Courses createCourse(@RequestBody Courses course) {

return coursesRepository.save(course);

}

@GetMapping("/list")

public List<Courses> listCourses(){

return coursesRepository.findAll();

}

@PutMapping("/update")

public Courses updateCourses(@RequestBody Courses course) {

return coursesRepository.save(course);

}

@DeleteMapping("/delete")

public String deleteCourse(@RequestBody Courses course) {

coursesRepository.delete(course);

return "Deleted Course Record";

}

}

**Student RestController:**

package com.springboot\_task.rest\_controller;

import java.util.List;

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

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

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

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

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

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

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

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

import com.springboot\_task.models.Student;

import com.springboot\_task.repository.StudentsRepository;

@RestController

@RequestMapping("/student")

public class StudentsRestController {

@Autowired

private StudentsRepository studentsRepository;

@PostMapping("/create")

public Student createStudent(@RequestBody Student student) {

return studentsRepository.save(student);

}

@GetMapping("/list")

public List<Student> listStudents(){

return studentsRepository.findAll();

}

@PutMapping("/update")

public Student updateStudent(@RequestBody Student student) {

return studentsRepository.save(student);

}

@DeleteMapping("/delete")

public String deleteStudent(@RequestBody Student student) {

studentsRepository.delete(student);

return "Deleted Student Record";

}

}

**Teachers RestController:**

package com.springboot\_task.rest\_controller;

import java.util.List;

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

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

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

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

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

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

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

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

import com.springboot\_task.models.Teachers;

import com.springboot\_task.repository.TeachersRepository;

@RestController

@RequestMapping("/teacher")

public class TeachersRestController {

@Autowired

private TeachersRepository teachersRepository;

@PostMapping("/create")

public Teachers createTeacher(@RequestBody Teachers teacher) {

return teachersRepository.save(teacher);

}

@GetMapping("/list")

public List<Teachers> listTeachers(){

return teachersRepository.findAll();

}

@PutMapping("/update")

public Teachers updateTeacher(@RequestBody Teachers teacher) {

return teachersRepository.save(teacher);

}

@DeleteMapping("/delete")

public String deleteTeacher(@RequestBody Teachers teacher) {

teachersRepository.delete(teacher);

return "Deleted Teacher Record";

}

}

**Main Class:**

package com.springboot\_task;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class SpringBootTaskApplication {

public static void main(String[] args) {

SpringApplication.run(SpringBootTaskApplication.class, args);

}

}

**Application.properties:**

spring.datasource.url=jdbc:mysql://localhost:3306/task?useSSL=false&serverTimezone=UTC&useLegacyDatetimeCode=false

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

spring.jpa.generate-ddl=true

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=update

server.port=8086

**Q5. Develop wrapper restful webservice to consume downstream restful service using Feign**

**Client. Use service in Step 1 as downstream service by the integration of the following:**

1. **YAML Configuration.**
2. **Swagger 3.0.**
3. **Spring JPA.**
4. **SpringData.**
5. **Retries(Retry 2 times in case of any failure while connecting to DB).**
6. **Circuit Breaker.**
7. **Rate limiters.**
8. **Unit Testing.**
9. **Lombook.**
10. **Acutator(Metrics & health check URL configuration).**
11. **Spring Security Basic authentication.**

Add Swagger, spring jpa, spring web, spring data, Lombok, resilience4j, Actuator, Spring Security, MySql Driver and Junit dependencies in your project.

**Program:**  
**Application.yml:**

eureka:

client:

fetch-registry: true

register-with-eureka: true

service-url:

default-zone: http://localhost:8761/eureka

management:

endpoint:

health:

enabled: true

loggers:

enabled: true

refresh:

enabled: true

restart:

enabled: true

shutdown:

enabled: true

endpoints:

enabled-by-default: false

web:

exposure:

include: '\*'

health:

db:

enabled: true

diskspace:

enabled: true

security:

enabled: false

server:

port: 8092

spring:

application:

name: STUDENT-SERVICE

datasource:

driver-class-name: com.mysql.cj.jdbc.Driver

password: root

url: jdbc:mysql://localhost:3306/task?useSSL=false&serverTimezone=UTC&useLegacyDatetimeCode=false

username: root

jpa:

database-platform: org.hibernate.dialect.MySQL8Dialect

generate-ddl: true

hibernate:

show-sql: true

**Book Class:**

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@NoArgsConstructor

@AllArgsConstructor

public class Book {

private Integer bookId;

private String bookName;

private Double bookCost;

public Integer getBookId() {

return bookId;

}

public void setBookId(Integer bookId) {

this.bookId = bookId;

}

public String getBookName() {

return bookName;

}

public void setBookName(String bookName) {

this.bookName = bookName;

}

public Double getBookCost() {

return bookCost;

}

public void setBookCost(Double bookCost) {

this.bookCost = bookCost;

}

public Book(Integer bookId, String bookName, Double bookCost) {

super();

this.bookId = bookId;

this.bookName = bookName;

this.bookCost = bookCost;

}

@Override

public String toString() {

return "Book [bookId=" + bookId + ", bookName=" + bookName + ", bookCost=" + bookCost + "]";

}

}

**BookRestConsumer:**

import java.util.List;

import org.springframework.cloud.openfeign.FeignClient;

import org.springframework.http.ResponseEntity;

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

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

@FeignClient(name="BOOK-SERVICE")

public interface BookRestConsumer {

@GetMapping("/book/data")

public String getBookData();

@GetMapping("/book/{id}")

public Book getBookById(@PathVariable Integer id);

@GetMapping("/book/all")

public List<Book> getAllBooks();

@GetMapping("/book/entity")

public ResponseEntity<String> getEntityData();

}

**StudentRestController:**

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

import org.springframework.http.ResponseEntity;

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

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

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

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

@RestController

@RequestMapping("/student")

public class StudentRestController {

@Autowired

private BookRestConsumer consumer;

@GetMapping("/data")

public String getStudentInfo() {

System.out.println(consumer.getClass().getName()); // prints as a proxy class

return "Accessing from STUDENT-SERVICE ==> " + consumer.getBookData();

}

@GetMapping("/allBooks")

public String getBooksInfo() {

return "Accessing from STUDENT-SERVICE ==> " + consumer.getAllBooks();

}

@GetMapping("/getOneBook/{id}")

public String getOneBookForStd(@PathVariable Integer id) {

return "Accessing from STUDENT-SERVICE ==> " + consumer.getBookById(id);

}

@GetMapping("/entityData")

public String printEntityData() {

ResponseEntity<String> resp = consumer.getEntityData();

return "Accessing from STUDENT-SERVICE ==> " + resp.getBody() + " , status is:" + resp.getStatusCode();

}

}

**Main Class:**

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

import org.springframework.cloud.openfeign.EnableFeignClients;

import springfox.documentation.swagger2.annotations.EnableSwagger2;

@SpringBootApplication

@EnableEurekaClient

@EnableFeignClients

@EnableSwagger2

@EnableCircuitBreaker

@EnableHystrix

public class SpringBootBookConsumerApplication {

public static void main(String[] args) {

SpringApplication.run(SpringBootBookConsumerApplication.class, args);

}

}

Create a Server Now and deploy the Application.

**Server Program:**

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication

@EnableEurekaServer

@EnableDiscoveryClient

public class SpringBootEurekaServerDemoApplication {

public static void main(String[] args) {

SpringApplication.run(SpringBootEurekaServerDemoApplication.class, args);

}

}

**Application.yml:**

eureka:

client:

fetch-registry: false

register-with-eureka: false

service-url:

default-zone: <http://localhost:8761/eureka>

instance:

hostname:localhost

logging:

level:

com:

netflix:

eureka:OFF

discovery:OFF

server:

port: 8761

Theoretical Questions:

**Q1. What does one mean by Load Balancing? How is it implemented in Spring Cloud?**

Ans. Load balancing describes the process of distributing computing workloads across many different resources. A single computer to a computer cluster and across network links, different processing units (CPUs) and disk drives, load balancing strives to organize and use each resource to its capacity without overloading any singular resource. By using a network of resources, with load balancing, instead of a single computer or individual unit the resources are optimized to maximize the function and speed of the unit and minimize the response time of all the components. Dedicated hardware and/or software are usually required for load balancing. Multilayer switches or a DNS server process are regularly used for load balancing.

It can be implemented using dependencies like Open Feign, Ribbon, Cloud Load Balancer, Eureka, etc.

**Q2. In which business scenario to use Netflix Hystrix?**

Ans. Netflix Hystrix is a design and implementation of circuit breaker pattern to isolate failure points and stop cascading them. It uses circuit breaker and fall back mechanisms to detect failures. This can be done with annotation @HystrixCommand where classes are annotated with @Component or @Sevice.

When we use @HystrixCommand, Hystrix monitors failures till threshold point then subsequent calls are failed and redirected to fallback method.

**Q3. What is Spring Cloud Gateway? What are its advantages over Netflix Zuul?**

Ans.Zuul is a blocking API. A blocking gateway api makes use of as many threads as the number of incoming requests. So this approach is more resource intensive. If no threads are available to process incoming request then the request has to wait in queue.

So, we will be implementing API Gateway using Spring Cloud Gateway. Spring Cloud Gateway is a non blocking API. When using non blocking API, a thread is always available to process the incoming request. These request are then processed asynchronously in the background and once completed the response is returned. So no incoming request never gets blocked when using Spring Cloud Gateway.

**Q4. What is Spring Cloud Bus? Need for it?**

Ans. Consider the scenario that we have multiple applications reading the properties using the Spring Cloud Config and the Spring Cloud Config in turn reads these properties from GIT. What will happen if suppose the eureka registration property in GIT changes to point to another Eureka server. In such a scenario we will have to restart the services to get the updated properties. There is another way of using Actuator Endpoint /refresh. But we will have to individually call this url for each of the modules. For example if Employee Producer1 is deployed on port 8080 then call http://localhost:8080/refresh. Similarly for Employee Producer2 http://localhost:8081/refresh and so on. This is again cumbersome. This is where Spring Cloud Bus comes. The Spring Cloud Bus provides feature to refresh configurations across multiple instances. So in above example if we refresh for Employee Producer1, then it will automatically refresh for all other required modules. This is particularly useful if we are having multiple microservice up and running. This is achieved by connecting all microservices to a single message broker. Whenever an instance is refreshed, this event is subscribed to all the microservices listening to this broker and they also get refreshed. The refresh to any single instance can be made by using the endpoint /bus/refresh

**Q5. What is Spring Cloud Data Flow? Need for it?**

Ans. Spring Cloud Data Flow is a toolkit to build real-time data integration and data processing pipelines by establishing message flows between Spring Boot applications that could be deployed on top of different runtimes. Long lived applications require Stream Applications while Short lived applications require Task Applications.  
In this example we make use of Stream Applications. Previously we had already developed Spring Cloud Stream applications to understand the concept of [Spring Cloud Stream Source](https://www.javainuse.com/spring/cloud-stream-rabbitmq-1) and [Spring Cloud Sink](https://www.javainuse.com/spring/cloud-stream-rabbitmq-2) and their benefit. Pipelines consist of Spring Boot apps, built using the Spring Cloud Stream or Spring Cloud Task microservice frameworks. SCDF can be accessed using the REST API exposed by it or the web UI console.  
We can make use of metrics, health checks, and the remote management of each microservice application Also we can scale stream and batch pipelines without interrupting data flows. With SCDF we build data pipelines for use cases like data ingestion, real-time analytics, and data import and export.

**Q6. What is Docker? How to deploy Spring Boot Microservices to Docker?**

Ans. Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. Create self-contained development environments inside Docker containers. So we share an environment already configured. This is done using a docker image.  
An image is a lightweight, stand-alone, executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and config files.

**To Deploy Spring Boot Microservices to Docker do the following:**

1. Create a Maven File.
2. Create the dockerfile.Docker file is a list of commands that we want the docker engine to execute. Go to the spring boot project folder and create a docker file.
3. Open the terminal and start the docker. (type “start docker” in terminal)
4. Now open the terminal and go to the Spring Boot project folder.
5. Next we will build an image with the name producer. (type “docker image build -t employee-producer” in terminal)
6. Next we will run the above image as a container. Also we will be publishing the docker port 8080 to centos port 8080.
7. Until now we made use of the unique container id when we created a new container. Instead now we can give an name to the container and use it for referring the container. (type “docker container run –name producer -p 8080:8080 -d employee-producer” in terminal)
8. Here we have started the container with name as producer. Now using the following command check the logs.( type “docker container logs producer” in terminal)
9. The application has started successfully. Go to localhost:8080/employee, we will see that our application is deployed successfully.

**Q7. How to deploy multiple microservices to docker?**

Ans. To Deploy Multiple Microservices to Docker.

1. Import two projects.
2. Open the terminal and start the docker. (type “start docker” in terminal)
3. Next we will create our own network will be of type bridge with the name consumer-producer. (type “docker network create consumer-producer” in terminal)
4. Lets start the employee producer container on the newly created network (type “docker container run --network consumer-producer –name producer -p 8080:8080 -d employee-producer” in terminal)
5. Lets start the employee consumer container on the newly created network (type “docker container run --network consumer-producer –name consumer -d employee-consumer” in terminal)

**Q8. How to implement feign client using springboot?**

Ans. To implement feign client using springboot follow these:

1. Create a Spring Boot Project.
2. Apply Annotation @EnableEurekaClient and @EnableFeignClients at the main class
3. Modify application.properties file
4. Create Model class
5. Create an interface
6. Create a RestController
7. Run the program and test it.

**Q9. How to implement security for microservices?**

Ans. Microservices Security can be implemented either using OAuth2 or JWT.

* Develop a Spring Boot Application to expose a Simple REST GET API with mapping /hello.
* Configure Spring Security for JWT. Expose REST POST API with mapping /authenticate using which User will get a valid JSON Web Token. And then allow the user access to the api /hello only if it has a valid token

**Q10. How to implement distributed logging for microservices?**

Ans. Microservices architecture involve multiple services which interact with each other. So a functionality may involve call to multiple microservices. Usually for systems developed using Microservices architecture, there are many microservices involved. These microservices collaborate with each other.

If suppose during such calls there are some issues like exception has occurred. Or may be there are latency issues due to a particular service taking more than expected time. How do we identify where the issue is occurring. In regular project we would have used logging to analyze the logs to know more about occurred exceptions and also performance timing. But since microservices involves multiple services we cannot use regular logging. Each Service will be having its own separate logs. So we will need to go through the logs of each service. Also how do we correlate the logs to a request call chain i.e which logs of microservices are related to Request1, which are related to Request2. To resolve these issues we make use of Spring Cloud Sleuth and Zipkin.