

Open in app ↗

Sign up

Sign in

Medium



Search



Get started with microservices using Spring Boot



Senura Vihan Jayadeva · Follow

Published in MS Club of SLIIT

17 min read · Feb 10, 2023



Listen



Share



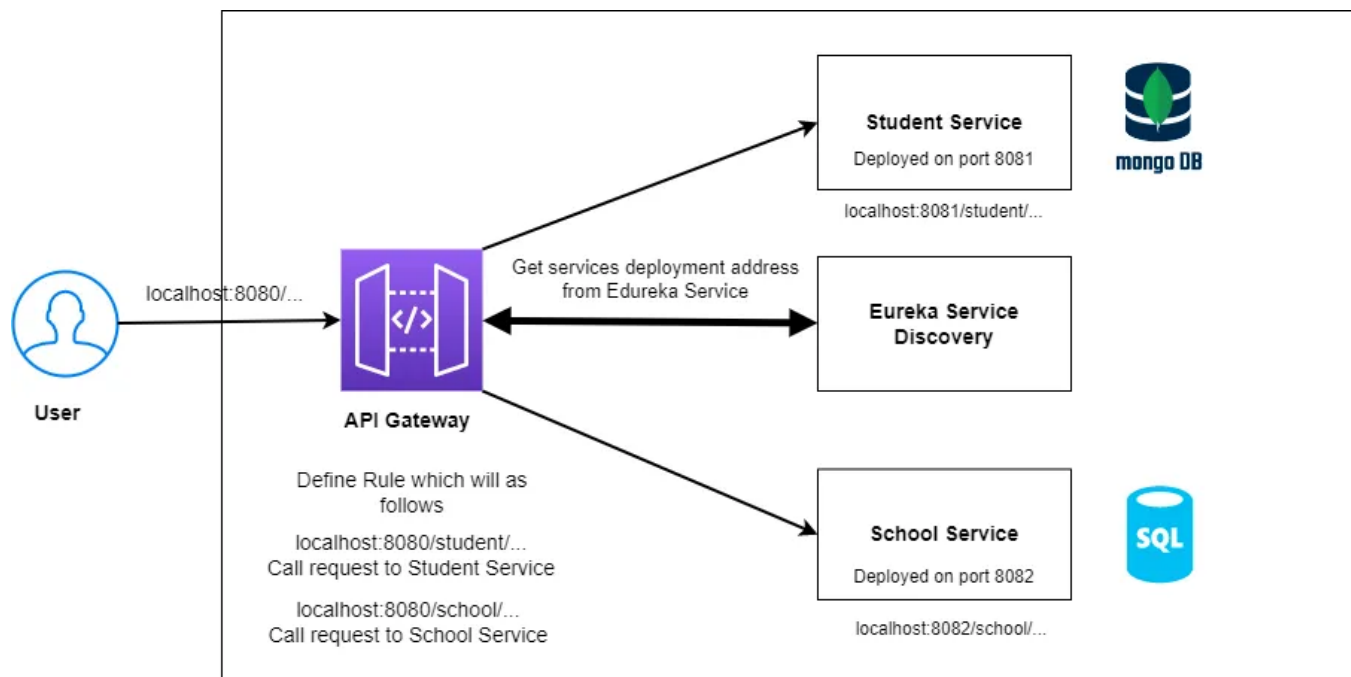
Today we are going to build a very simple microservice system with spring boot. Let's first see what is are microservices.

What is a microservice?

Microservices are a software architecture approach where an application is divided

into small, independent and self-contained services that communicate with each other through APIs. This allows for faster development, deployment and scalability.

This is our system architecture diagram to have a basic idea about the system before going to the implementation.



As you can see we have Student microservice and School microservice for this example. For the demonstration purpose student service has a NoSQL database called MongoDB and School service uses a SQL database.

Then you can see the Eureka Service Discovery. Service Discovery is a pattern where client applications can discover the location of services dynamically at runtime. The main idea is to have a central registry of all the services and their current status. This registry can be queried by clients to determine the location of a particular service.

The next thing I want to talk about is **API Gateway**. An API Gateway is a reverse proxy that sits in front of your microservices and acts as a single entry point for incoming API requests. It is responsible for request routing, composition, and protocol translation, among other things.

The connection between Service Discovery and API Gateway is that the API Gateway

uses the information from the Service Discovery registry to route requests to the appropriate microservice. The API Gateway queries the registry to determine the location of the target microservice, and then forwards the incoming request to that service. In this way, the API Gateway acts as a bridge between client applications and the microservices, providing a unified entry point for incoming requests and abstracting the underlying complexity of the microservices.

Right, First of all, let's see what I'm using for this project

- IDE or text editor (Eclipse , IntelliJ)
- JDK 17
- Maven or Gradle (We will be relying on Maven for this article)

In this tutorial, I will be using **IntelliJ** as the IDE and **MySQL** and **MongoDB** as the Databases.

I guess now you have a very good idea about the System and its technologies. So now we can directly go to the implementation part. I will start with the School Service.

School Microservice

To initialise the project go to <https://start.spring.io/> and you can specify your own **Group Id**, **Artifact Id**, **Project Name**, and **Java Version** as you wish. We'll add a few dependencies here as well, as we'll want to use them in our project: For now, I need **Lombok**, **Spring Data JPA**, **Spring Web** and **MySQL Driver** as dependencies.

- **Spring Web** — To include Spring MVC and embedded Tomcat into your project
- **Spring Data JPA** — Java Persistence API and Hibernate
- **MySQL Driver** — JDBC Driver
- **Lombok** — Java library tool that generates code for minimizing boilerplate code. The library replaces boilerplate code with easy-to-use annotations

After that, you can click on generate button and you can download a zip file. That's your project.

Project
☐ Gradle - Groovy
☐ Gradle - Kotlin
☒ Maven

Language
☒ Java
☐ Kotlin
☐ Groovy

Spring Boot
☐ 3.0.3 (SNAPSHOT)
☒ 3.0.2
☐ 2.7.9 (SNAPSHOT)
☐ 2.7.8

Project Metadata

Group

Artifact

Name

Description

Package name

Packaging ☒ Jar ☐ War

Java ☐ 19 ☒ 17 ☐ 11 ☐ 8

Dependencies ADD DEPENDENCIES... CTRL + B

Lombok DEVELOPER TOOLS
Java annotation library which helps to reduce boilerplate code.

Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

MySQL Driver SQL
MySQL JDBC driver.

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

GENERATE CTRL + G

EXPLORE CTRL + SPACE

SHARE...

After we created the project, if you go to **pom.xml** you can see the **dependencies** that we have been installed.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org
<modelVersion>4.0.0</modelVersion>
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.0.2</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.hexagon</groupId>
<artifactId>school-service</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>school-service</name>
<description>School Microservice</description>
<properties>
  <java.version>17</java.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
  <groupId>com.mysql</groupId>
  <artifactId>mysql-connector-j</artifactId>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <excludes>
          <exclude>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
          </exclude>
        </excludes>
      </configuration>
    </plugin>
  </plugins>
</build>

</project>
```

Connect Spring Boot Application With the database

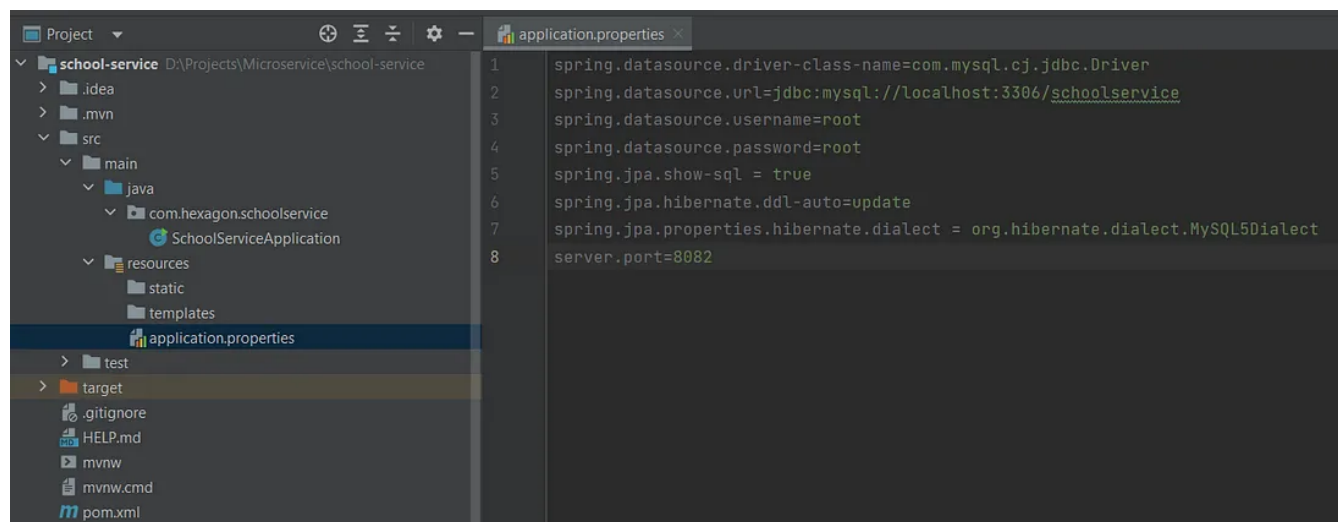
Next, before we start working on the application, we'll want to set up the database.

This can *easily* be done through Spring Data JPA, which allows us to set this connection up with just a couple of parameters.

For that under the **resources** folder, you can see a file called *application.properties*.

Inside the application.properties file add those details. As I mentioned earlier here we use **MySQL** as the database. Here we have to specify our **Server port**, **database name**, **username**, and **password**.

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:port/databasename
spring.datasource.username=username
spring.datasource.password=password
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
server.port=port number
```



Right, Now we are done with connecting the School service with the database.

Creating a School Entity (Domain Model)

Now let's create our school entity. First of all, I will create a package for models. Inside the entity package, I'm going to create a class called School.java. Then add the following code.

```
package com.hexagon.schoolservice.entity;

import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name = "school")
public class School {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String schoolName;
    private String location;
    private String principalName;
}
```

Here you don't have to generate constructors, getters, and setters because those are done by **@Data**, **@AllArgsConstructor**, and **@NoArgsConstructor** annotations which were provided by **Lombok** dependency.

The **@Entity** annotation specifies that the class is an entity and is mapped to a database table. The **@Table** annotation specifies the name of the database table to be used for mapping. The **@Id** annotation specifies the primary key of an entity and **@GeneratedValue** provides for the specification of generation strategies for the values of primary keys.

Creating Repository Classes (Persistence Layer)

Our next step is creating a repository class. **@Repository** is a Spring annotation that indicates that the decorated class is a repository. A repository is a mechanism for encapsulating storage, retrieval, and search behavior that emulates a collection of

objects.

First of all, I'm going to create a package for repository classes. Inside the package create an *Interface* called *SchoolRepository*. Then add the following code.

```
package com.hexagon.schoolservice.repository;

import com.hexagon.schoolservice.entity.School;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface SchoolRepository extends JpaRepository<School,Integer> {
}
```

Here you can see I extends the interface by JpaRepository and passed the *School* Model and *data type of its Id*.

JpaRepository is JPA specific extension of Repository . It contains the full API of CrudRepository and PagingAndSortingRepository . So it contains API for basic CRUD operations and also API for pagination and sorting.

Creating a Service (Business Layer)

We mark beans with @Service to indicate that it's holding the business logic.

Here we are going to implement the functions like **Insert**, **retrieve**, **update** and **delete** the Student records. We wrote the **business logic** in the service class file.

Now create a package called *service* and under the service package create a class called *SchoolService*. Finally, add the following code inside the SchoolService class.

```
package com.hexagon.schoolservice.service;

import com.hexagon.schoolservice.entity.School;
import com.hexagon.schoolservice.repository.SchoolRepository;
import org.springframework.beans.factory.annotation.Autowired;
```



```
import org.springframework.stereotype.Service;

import java.util.List;
@Service
public class SchoolService {
    @Autowired
    private SchoolRepository schoolRepository;

    public School addSchool(School school){
        return schoolRepository.save(school);
    }
    public List<School> fetchSchools(){
        return schoolRepository.findAll();
    }
    public School fetchSchoolById(int id){
        return schoolRepository.findById(id).orElse(null);
    }
}
```

Creating a Controller

The code below shows the Rest Controller class file, here we @Autowired the StudentService class and called the methods for crud operations.

```
package com.hexagon.schoolservice.controller;

import com.hexagon.schoolservice.entity.School;
import com.hexagon.schoolservice.service.SchoolService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@CrossOrigin("*")
@RequestMapping(value = "/school")
@RestController
public class SchoolController {
    @Autowired
    private SchoolService schoolService;

    @PostMapping
    public School addSchool(@RequestBody School school){
        return schoolService.addSchool(school);
    }
}
```

```
@GetMapping
public List<School> fetchSchools(){
    return schoolService.fetchSchools();
}

@GetMapping("/{id}")
public School fetchSchoolById(@PathVariable int id){
    return schoolService.fetchSchoolById(id);
}
}
```

Check why I used `@CrossOrigin("*")`: <https://spring.io/guides/gs/rest-service-cors>

Okay, Now Right-click on the `DemoApplication.java` file and click on Run. Before running the application make sure that you have created a database with the name that you have given in the `application.properties` file. In my case **schoolservice**.

Right, now we are almost finished **School Microservice**. Now we can test the **REST APIs** using software called **Postman**.

The screenshot displays two Postman requests. The first is a POST request to `http://localhost:8082/school` with a JSON body containing school details. The second is a GET request to the same URL, showing the response in the 'Body' tab.

POST Request:

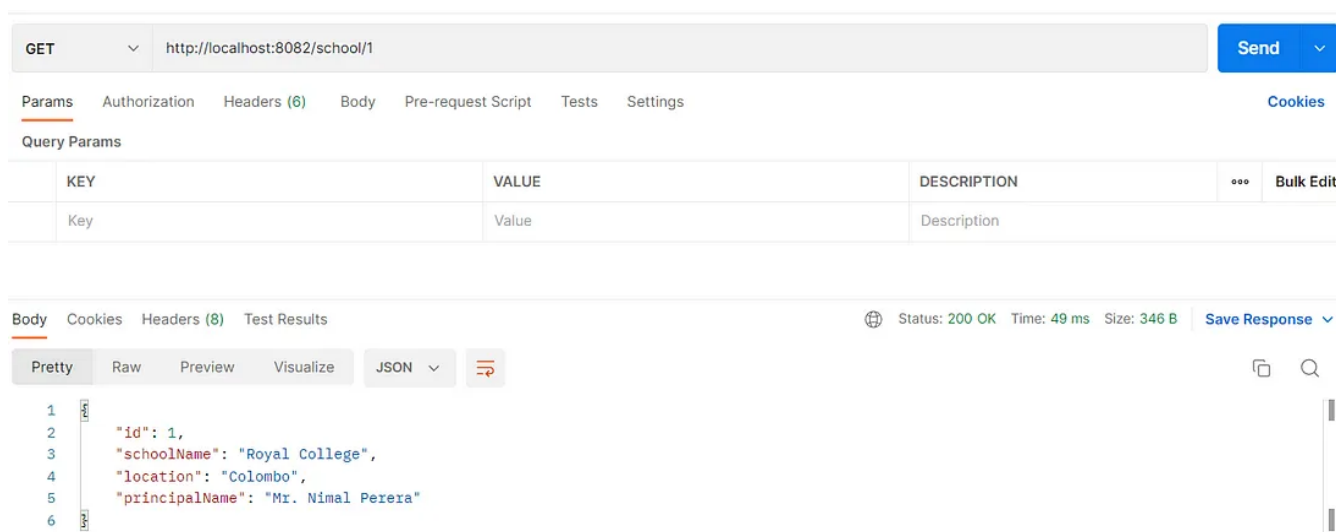
- Method: POST
- URL: `http://localhost:8082/school`
- Body (JSON):

```
{  "schoolName": "Royal College",  "location": "Colombo",  "principalName": "Mr. Nimal Perera"}
```

GET Request:

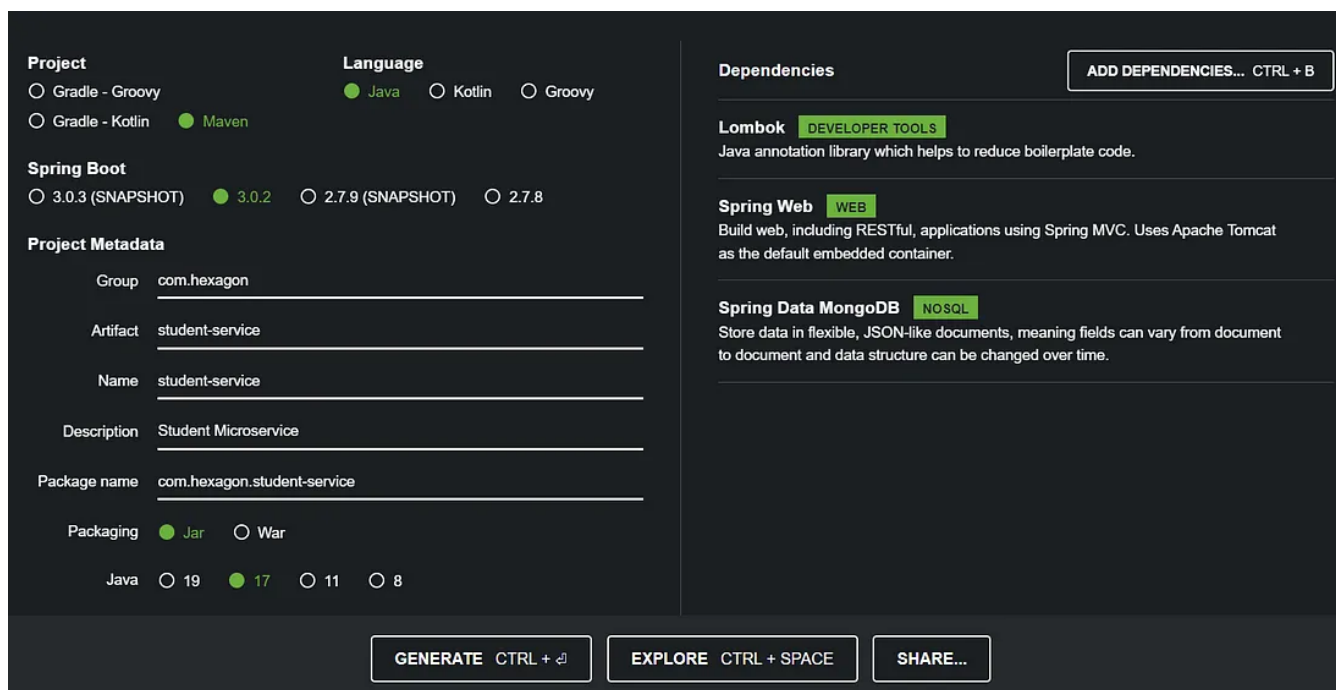
- Method: GET
- URL: `http://localhost:8082/school`
- Status: 200 OK
- Time: 719 ms
- Size: 348 B
- Response (JSON):

```
{  "id": 1,  "schoolName": "Royal College",  "location": "Colombo",  "principalName": "Mr. Nimal Perera"}
```



Student Microservice

Next Im going to create the Student Sevice. Again go to spring intializer and specify your own Groud Id, Artifact Id, Project Name, Java Version and Dependencies.



Spring Data MongoDB : MongoDB stores data in collections. Spring Data MongoDB maps the Customer class into a collection called customer . If you want to change the name of the collection, you can use Spring Data MongoDB's @Document annotation on the class

After we created the project, if you go to **pom.xml** you can see the **dependencies** that we have been installed.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org
<modelVersion>4.0.0</modelVersion>
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.0.2</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.hexagon</groupId>
<artifactId>student-service</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>student-service</name>
<description>Student Microservice</description>
<properties>
  <java.version>17</java.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
```

```
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
<configuration>
<excludes>
<exclude>
<groupId>org.projectlombok</groupId>
<artifactId>lombok</artifactId>
</exclude>
</excludes>
</configuration>
</plugin>
</plugins>
</build>

</project>
```

Now let's set up the database. For that inside the application.properties file add the connection string

```
spring.data.mongodb.uri=put the connection string here
server.port=8081
```

Creating a Student Model (Domain Model)

Now let's create our student model. As previously I will create a package for models. Inside the model package, I'm going to create a class called Student.java. Then add the following code.

```
package com.hexagon.studentservice.model;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
```

```
@Setter
@Getter
@AllArgsConstructor
@NoArgsConstructor
@Document(collection = "students")
public class Student {
    @Id
    private String id;
    private String name;
    private int age;
    private String gender;
    private Integer schoolId;
}
```

Creating Repository Classes (Persistence Layer)

Next, let's create a package for repository classes. Inside the package create an *Interface* called *StudentRepository*. Then add the following code.

```
package com.hexagon.studentservice.repository;

import com.hexagon.studentservice.model.Student;
import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface StudentRepository extends MongoRepository<Student, String> {
}
```

MongoRepository extends the PagingAndSortingRepository and QueryByExampleExecutor interfaces that further extend the CrudRepository interface. MongoRepository provides all the necessary methods which help to create a CRUD application and it also supports the custom derived query methods

Creating a Service (Business Layer)

To write the **business logic** in the service class file create a package called *service* and under the service package create a class called *StudentService*. Then, add the following code inside the StudentService class.

Ohh... Before that, I will create a package called `dto` and create two classes called `School` and `StudentResponse`. You will realise why I created those later. Then add the following code for each class respectively.

```
package com.hexagon.studentservice.dto;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class School {
    private int id;
    private String schoolName;
    private String location;
    private String principalName;
}
```

```
package com.hexagon.studentservice.dto;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class StudentResponse {
    private String id;
    private String name;
    private int age;
    private String gender;
    private School school;
}
```

Here I'm going to call a school service endpoint in student service. For that we need RestTemplate for that.

Rest Template is used to create applications that consume RESTful Web Services. You can use the **exchange()**, **getForObject** etc methods to consume the web services for all HTTP methods. The code given below shows how to create Bean for Rest Template to auto wiring the Rest Template object. Let's go to our main class and add the following.

```
package com.hexagon.studentservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
public class StudentServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(StudentServiceApplication.class, args);
    }

    @Bean
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }
}
```

After that now you can add the following code to your StudentService Class.

```
package com.hexagon.studentservice.service;

import com.hexagon.studentservice.dto.School;
import com.hexagon.studentservice.dto.StudentResponse;
import com.hexagon.studentservice.model.Student;
import com.hexagon.studentservice.repository.StudentRepository;
import org.springframework.beans.factory.annotation.Autowired;
```



```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

import java.util.List;
import java.util.Optional;

@Service
public class StudentService {
    @Autowired
    private StudentRepository studentRepository;

    @Autowired
    private RestTemplate restTemplate;

    public ResponseEntity<?> createStudent(Student student){
        try{
            return new ResponseEntity<Student>(studentRepository.save(student),
        }catch(Exception e){
            return new ResponseEntity<>(e.getMessage(), HttpStatus.INTERNAL_SER
        }
    }

    public ResponseEntity<?> fetchStudentById(String id){
        Optional<Student> student = studentRepository.findById(id);
        if(student.isPresent()){
            School school = restTemplate.getForObject("http://localhost:8082/sc
            StudentResponse studentResponse = new StudentResponse(
                student.get().getId(),
                student.get().getName(),
                student.get().getAge(),
                student.get().getGender(),
                school
            );
            return new ResponseEntity<>(studentResponse, HttpStatus.OK);
        }else{
            return new ResponseEntity<>("No Student Found",HttpStatus.NOT_FOUND
        }
    }

    public ResponseEntity<?> fetchStudents(){
        List<Student> students = studentRepository.findAll();
        if(students.size() > 0){
            return new ResponseEntity<List<Student>>(students, HttpStatus.OK);
        }else {
            return new ResponseEntity<>("No Students",HttpStatus.NOT_FOUND);
        }
    }
}
```

```
}  
  
}
```

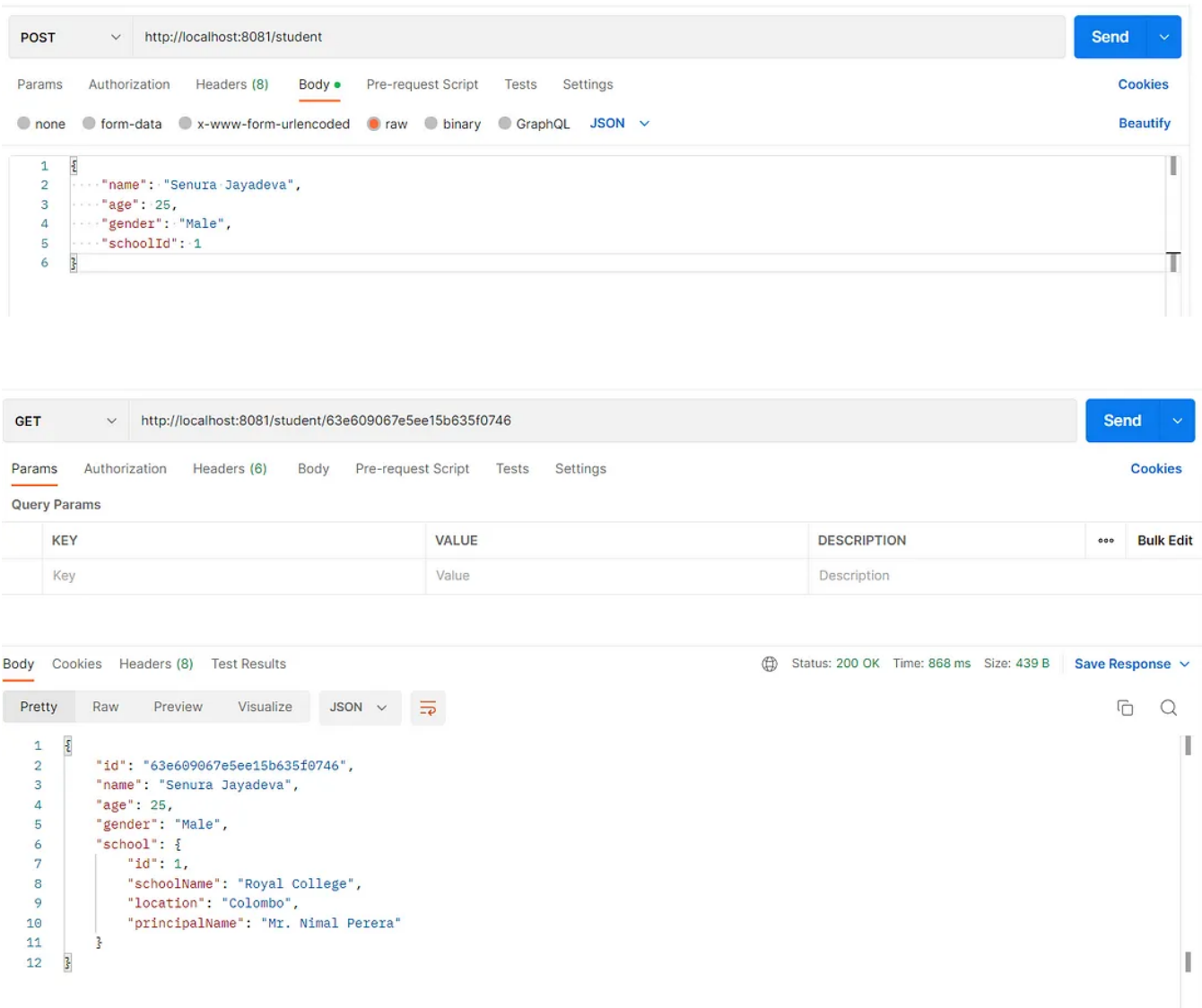
Here you can see in the `fetchStudentById` method we are calling an endpoint of school service in order to get the details of a school using the Id.

Now create a controller package and inside that create a class called `StudentController` and add the below code.

```
package com.hexagon.studentservice.controller;  
  
import com.hexagon.studentservice.model.Student;  
import com.hexagon.studentservice.service.StudentService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.http.ResponseEntity;  
import org.springframework.web.bind.annotation.*;  
  
@CrossOrigin("*")  
@RestController  
@RequestMapping("/student")  
public class StudentController {  
    @Autowired  
    private StudentService studentService;  
    @GetMapping("/{id}")  
    public ResponseEntity<?> fetchStudentById(@PathVariable String id){  
        return studentService.fetchStudentById(id);  
    }  
    @GetMapping  
    public ResponseEntity<?> fetchStudents(){  
        return studentService.fetchStudents();  
    }  
    @PostMapping  
    public ResponseEntity<?> createStudent(@RequestBody Student student){  
        return studentService.createStudent(student);  
    }  
}
```

Now you can run the application and test all the endpoints with a help of tool like

Postman.



Right now we already created our two services. Now we need to create the service registry.

What is a Service Registry ?

A service registry is a central repository in a microservice architecture that acts as a source of truth for the location and status of each microservice. Service registry helps microservices to discover each other and communicate with each other by providing a central place where services can register themselves and look up information about other services.

The need for a service registry in a microservice architecture arises due to the

following reasons:

1. **Dynamic nature of microservices:** Microservices can come and go dynamically, for example, due to scaling, upgrades, or failures. Service registry keeps track of this changing landscape and provides up-to-date information about the availability of each service.
2. **Load balancing:** Service registry provides information about the current load of each microservice instance and can be used by a load balancer to distribute requests to the least loaded instance.
3. **Service discovery:** Service registry helps microservices to discover each other and enables them to communicate with each other. This is important in a microservice architecture where services are decoupled and loosely coupled.
4. **Monitoring and management:** Service registry can also be used to monitor the health of microservices and provide an overview of the system status.

So the service registry is an essential component in a microservice architecture as it provides a centralized way to manage and coordinate the interactions between microservices.

Create the Service Registry

Again go to <https://start.spring.io/> and you can specify your own **Group Id**, **Artifact Id**, **Project Name**, and **Java Version** as you wish. I'm adding **Eureka Server** as a dependency.

Eureka Server : This is a service registry server in the Netflix OSS (Open Source Software) suite, and it is used to provide service discovery for microservices in a distributed environment. Eureka Server provides a registry of available microservices, allowing client microservices to discover and communicate with each other.

The screenshot shows the Spring Initializr web interface. On the left, under 'Project', 'Gradle - Maven' is selected. Under 'Language', 'Java' is selected. Under 'Spring Boot', '3.0.2' is selected. The 'Project Metadata' section includes: Group (com.hexagon), Artifact (service-registry), Name (service-registry), Description (Service Registry), Package name (com.hexagon.service-registry), and Packaging (Jar). On the right, the 'Dependencies' section shows 'Eureka Server' with 'SPRING CLOUD DISCOVERY' selected. At the bottom, there are buttons for 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'.

First I will go to main class and annotate with `@EnableEurekaServer`

```
package com.hexagon.serviceregistry;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class ServiceRegistryApplication {

    public static void main(String[] args) {
        SpringApplication.run(ServiceRegistryApplication.class, args);
    }

}
```

Next in the application.properties file add the following.

```
server.port=8761
eureka.client.register-with-eureka=false
```

```
eureka.client.fetch-registry=false
```

1. `server.port` : This property sets the port on which the Eureka server will run. In this case, the server will run on port 8761.
2. `eureka.client.register-with-eureka` : This property indicates whether this instance should register with Eureka server as a service. In this case, the value is set to `false`, which means that this instance will not register with the Eureka server.
3. `eureka.client.fetch-registry` : This property indicates whether this instance should fetch the registry information from the Eureka server. In this case, the value is set to `false`, which means that this instance will not fetch the registry information from the Eureka server.

Now go again to `pom.xml` of both student service and school service and do the following changes.

```
<properties>
  <java.version>17</java.version>
  <spring-cloud.version>2022.0.1</spring-cloud.version>
</properties>
```

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
```

```
        <type>pom</type>
        <scope>import</scope>
    </dependency>
</dependencies>
</dependencyManagement>
```

After adding these dependencies pom.xml files will look like this in school service and student service.

In Student Service

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.o
<modelVersion>4.0.0</modelVersion>
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.2</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.hexagon</groupId>
<artifactId>student-service</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>student-service</name>
<description>Student Microservice</description>
<properties>
    <java.version>17</java.version>
    <spring-cloud.version>2022.0.1</spring-cloud.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-mongodb</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
```

```
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <excludes>
                    <exclude>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
                    </exclude>
                </excludes>
            </configuration>
        </plugin>
    </plugins>
</build>

</project>
```

In School Service


```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.o
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.2</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.hexagon</groupId>
  <artifactId>school-service</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>school-service</name>
  <description>School Microservice</description>
  <properties>
    <java.version>17</java.version>
    <spring-cloud.version>2022.0.1</spring-cloud.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
      <groupId>com.mysql</groupId>
      <artifactId>mysql-connector-j</artifactId>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
```

```
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <excludes>
                    <exclude>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
                    </exclude>
                </excludes>
            </configuration>
        </plugin>
    </plugins>
</build>

</project>
```

Now we need to do the configuration then these services can connect to Eureka Server.

For that first go to school service application properties and do the below changes.

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/schoolservice
```

```
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.show-sql = true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
server.port=8082

spring.application.name=SCHOOL-SERVICE
eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
eureka.instance.hostname=localhost
```

- The `spring.application.name` property sets the name of the application as "SCHOOL-SERVICE".
- `eureka.client.register-with-eureka` property is set to `true` which means the application will register itself with the Eureka server.
- `eureka.client.fetch-registry` property is set to `true`, meaning that the application will retrieve the registry information from the Eureka server.
- `eureka.client.service-url.defaultZone` property sets the URL of the Eureka server to `http://localhost:8761/eureka/`.
- `eureka.instance.hostname` property sets the hostname of the instance to `localhost`.

This configuration allows the application to register itself with the Eureka server and retrieve the registry information from the server, so that other services can discover and communicate with it.

I guess now you are having a clear idea. Now do the same thing for student service.

```
spring.data.mongodb.uri=put the connection string
server.port=8081

spring.application.name=STUDENT-SERVICE
```

```
eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
eureka.instance.hostname=localhost
```

Okay. All set. Now start the service registry and open your web browser and go to <http://localhost:8761>. You should be able to see a page like below.

The screenshot shows the Spring Eureka web interface. At the top, there's a header with the 'spring Eureka' logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'. Below the header, the 'System Status' section is displayed, containing two tables. The first table shows 'Environment' as 'test' and 'Data center' as 'default'. The second table shows 'Current time' as '2023-02-10T18:26:27 +0530', 'Uptime' as '00:00', 'Lease expiration enabled' as 'false', 'Renews threshold' as '1', and 'Renews (last min)' as '0'. Below this, the 'DS Replicas' section shows 'localhost'. The 'Instances currently registered with Eureka' section features a table with columns 'Application', 'AMIs', 'Availability Zones', and 'Status', with a message 'No instances available'. At the bottom, the 'General Info' section has a table with columns 'Name' and 'Value'.

System Status	
Environment	test
Data center	default

System Status	
Current time	2023-02-10T18:26:27 +0530
Uptime	00:00
Lease expiration enabled	false
Renews threshold	1
Renews (last min)	0

DS Replicas

localhost

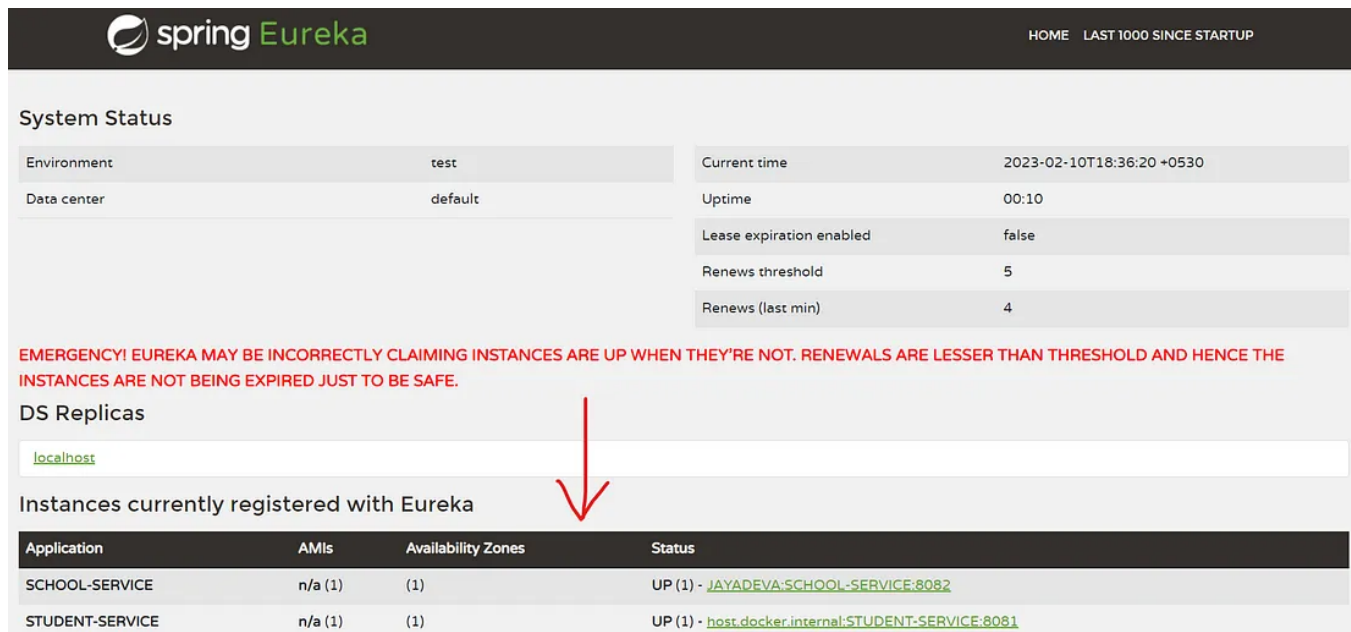
Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

General Info

Name	Value
------	-------

Currently you can't see any application registered with the server. So after changing the application properties you have to restart both student and school services and then again refresh the browser. Then you will be able to see STUDENT-SERVICE and SCHOOL-SERVICE under the instances currently registered with Eureka.



The screenshot shows the Spring Eureka dashboard. At the top, there's a header with the Spring Eureka logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'. Below the header, the 'System Status' section displays two tables. The left table shows 'Environment: test' and 'Data center: default'. The right table shows 'Current time: 2023-02-10T18:36:20 +0530', 'Uptime: 00:10', 'Lease expiration enabled: false', 'Renews threshold: 5', and 'Renews (last min): 4'. Below this, a red warning message states: 'EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.' Under the 'DS Replicas' section, 'localhost' is listed. The 'Instances currently registered with Eureka' section features a red arrow pointing to a table with the following data:

Application	AMIs	Availability Zones	Status
SCHOOL-SERVICE	n/a (1)	(1)	UP (1) - JAYADEVA:SCHOOL-SERVICE:8082
STUDENT-SERVICE	n/a (1)	(1)	UP (1) - host.docker.internal:STUDENT-SERVICE:8081

Right. If you can remember in Student service I have called the School service with <http://localhost:8082/school/{id}>. Consider that we have multiple instances of this service and those services working on different ports or different urls. Then it would be difficult to get the correct url information. So in our case instead of giving localhost:8082 we are giving the Application name as you can see below. Advantage is we dont have to worry about Host name or the Port Number. We have to give only Application name.

```
public ResponseEntity<?> fetchStudentById(String id){
    Optional<Student> student = studentRepository.findById(id);
    if(student.isPresent()){
        School school = restTemplate.getForObject("http://SCHOOL-SERVICE/school
        StudentResponse studentResponse = new StudentResponse(
            student.getId(),
            student.getName(),
            student.getAge(),
            student.getGender(),
            school
        );
        return new ResponseEntity<>(studentResponse, HttpStatus.OK);
    }else{
        return new ResponseEntity<>("No Student Found",HttpStatus.NOT_FOUND);
    }
}
```

Using the application name instead of the host name and port number provides a more flexible and scalable approach to service discovery and makes it easier to manage your microservice architecture.

Test the endpoints again with Postman and check the results.

We connected to service registry and now we want **load balance** the requests. Go to Student service main class and annotate the RestTemplate with **@LoadBalanced** annotation.

```
package com.hexagon.studentservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
public class StudentServiceApplication {
```

```
public static void main(String[] args) {  
    SpringApplication.run(StudentServiceApplication.class, args);  
}  
@Bean  
@LoadBalanced  
public RestTemplate restTemplate(){  
    return new RestTemplate();  
}  
}
```

As in this example, the `RestTemplate` instance is annotated with `@LoadBalanced`, which means that it will use a load balancing algorithm to distribute requests to multiple instances of a service. To use this `RestTemplate` instance for making RESTful web service calls, you can simply autowire it into your component class and use it as needed.

Note that in order for the `@LoadBalanced` annotation to be effective, you must also have a load balancer service running, such as Netflix Eureka, in your microservice architecture. The load balancer will be responsible for distributing the requests to the available instances of a service.

Now let's go to the last part of this long article.

CREATE API GATEWAY

What is an API Gateway?

An API gateway is an intermediary that sits between your microservices and the client applications that consume them. It acts as a single entry point for all API requests, providing a unified interface for your microservices and simplifying the client-side architecture.

The use of an API gateway provides several benefits:

1. **Routing:** The API gateway can route requests to the appropriate microservice based on the request URL or other attributes. This allows you to change the underlying microservices without affecting the client applications.
2. **Load balancing:** The API gateway can perform load balancing, distributing

incoming requests across multiple instances of a microservice. This helps to improve the reliability and availability of the microservices.

3. **Caching:** The API gateway can cache frequently requested data to reduce the load on the microservices. This helps to improve the performance of the microservices and reduce response times for the client applications.
4. **Security:** The API gateway can act as a security layer, providing authentication and authorization for incoming requests. This helps to secure the microservices and ensure that only authorized clients can access them.
5. **Monitoring:** The API gateway can provide monitoring and logging for all API requests. This makes it easier to diagnose issues and monitor the performance of the microservices.

Overall, the use of an API gateway provides a centralized and efficient way to manage access to your microservices, improving the reliability, security, and performance of your microservice architecture.

Okay now you should have a good idea about API Gateway. Currently our request directly going to each microservice. Therefore now we are going to create an API Gateway so all the requests coming from the user will come to the Gateway first. Then based on the url pattern it will redirect to relevant microservice.

For the last time let's create another project.

Project
☐ Gradle - Groovy
☐ Gradle - Kotlin
☒ Maven

Language
☒ Java
☐ Kotlin
☐ Groovy

Spring Boot
☐ 3.0.3 (SNAPSHOT)
☒ 3.0.2
☐ 2.7.9 (SNAPSHOT)
☐ 2.7.8

Project Metadata

Group

Artifact

Name

Description

Package name

Packaging ☒ Jar ☐ War

Dependencies ADD DEPENDENCIES... CTRL + B

Eureka Discovery Client SPRING CLOUD DISCOVERY
A REST based service for locating services for the purpose of load balancing and failover of middle-tier servers.

Gateway SPRING CLOUD ROUTING
Provides a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as security, monitoring/metrics, and resiliency.

Spring Boot Actuator OPS
Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.

GENERATE CTRL + G EXPLORE CTRL + SPACE SHARE...

Now under the resources I'm creating a application.yml file and add the following code.

```
server:
  port: 8080

spring:
  application:
    name: API-GATEWAY
  cloud:
    gateway:
      routes:
        - id: STUDENT-SERVICE
          uri: lb://STUDENT-SERVICE
          predicates:
            - Path=/student/**
        - id: SCHOOL-SERVICE
          uri: lb://SCHOOL-SERVICE
          predicates:
            - Path=/school/**

eureka:
  client:
    register-with-eureka: true
    fetch-registry: true
    service-url:
```

```
defaultZone: http://localhost:8761/eureka/  
instance:  
  hostname: localhost
```

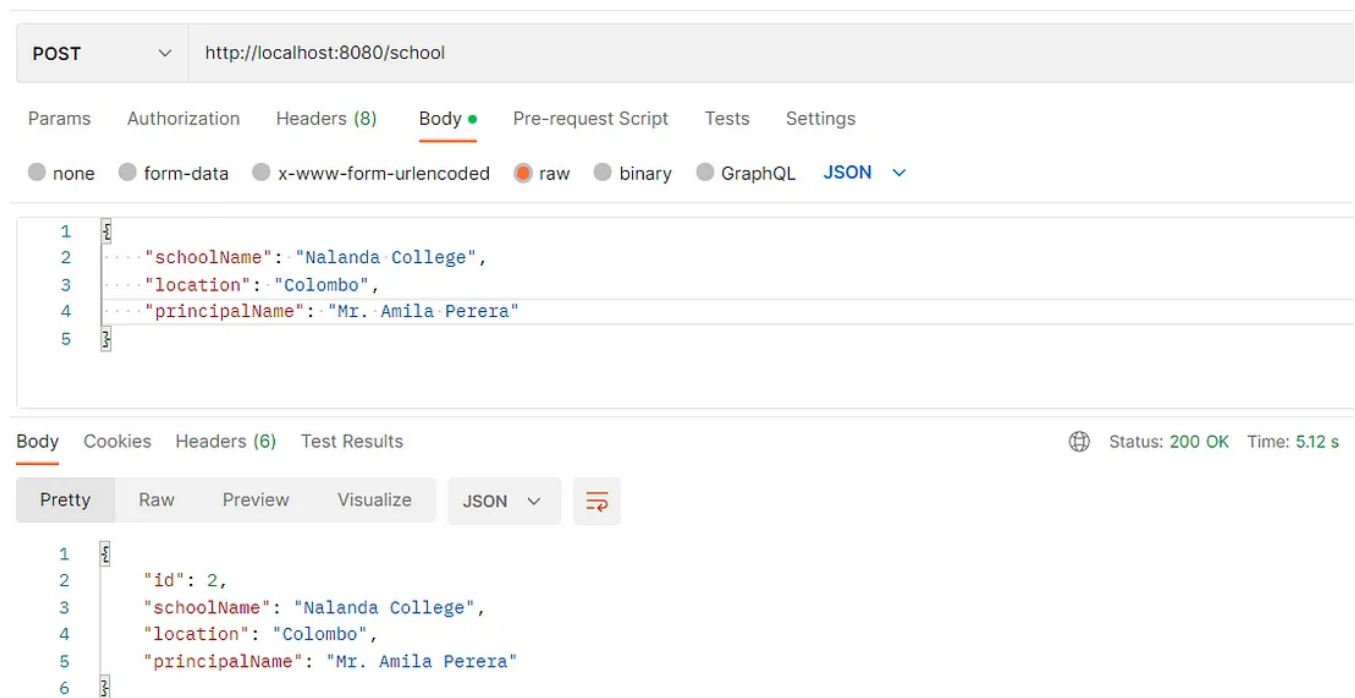
The `cloud.gateway` section defines two routes, "STUDENT-SERVICE" and "SCHOOL-SERVICE". These routes are used to forward incoming requests to different services based on the URL path. For example, requests starting with `/student/` will be forwarded to "STUDENT-SERVICE", while requests starting with `/school/` will be forwarded to "SCHOOL-SERVICE".

Start the cloud gateway application and service should be registered with the Eureka Service Registry. Now you should call the request with hostname `localhost` and port `8080` as given in the API-GATEWAY service.

Let's get an example like <http://localhost:8080/student>. So as per the above URL pattern given in the `application.yml` file, it should redirect to the Student Service.

Now test all the endpoints like above with a tool such as Postman. Here are some examples.

Create a School



Get student by Id

GET

http://localhost:8080/student/63e609067e5ee15b635f0746

Params

Authorization

Headers (6)

Body

Pre-request Script

Tests

Settings

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body

Cookies

Headers (6)

Test Results

Status: 200 OK Time: 8.69 s

Pretty

Raw

Preview

Visualize

JSON

```
1  {
2    "id": "63e609067e5ee15b635f0746",
3    "name": "Senura Jayadeva",
4    "age": 25,
5    "gender": "Male",
6    "school": {
7      "id": 1,
8      "schoolName": "Royal College",
9      "location": "Colombo",
10     "principalName": "Mr. Nimal Perera"
11   }
12 }
```

Github Link: <https://github.com/senuravihanjayadeva/Spring-Microservice>

We come to the end of a very long article. Hope you learned something. Thank you !

Microservices

Spring Boot

Eureka

Api Gateway

Service Discovery



Follow

Published in MS Club of SLIIT

113 Followers · Last published Nov 10, 2024

A student-driven community based on SLIIT aiming to bridge the skill gap between an Undergraduate and an Industry Professional



Follow

Written by Senura Vihan Jayadeva

195 Followers · 51 Following

Software Engineering undergraduate of Sri Lanka Institute of Information Technology | Physical Science Undergraduate of University of Sri Jayewardenepura

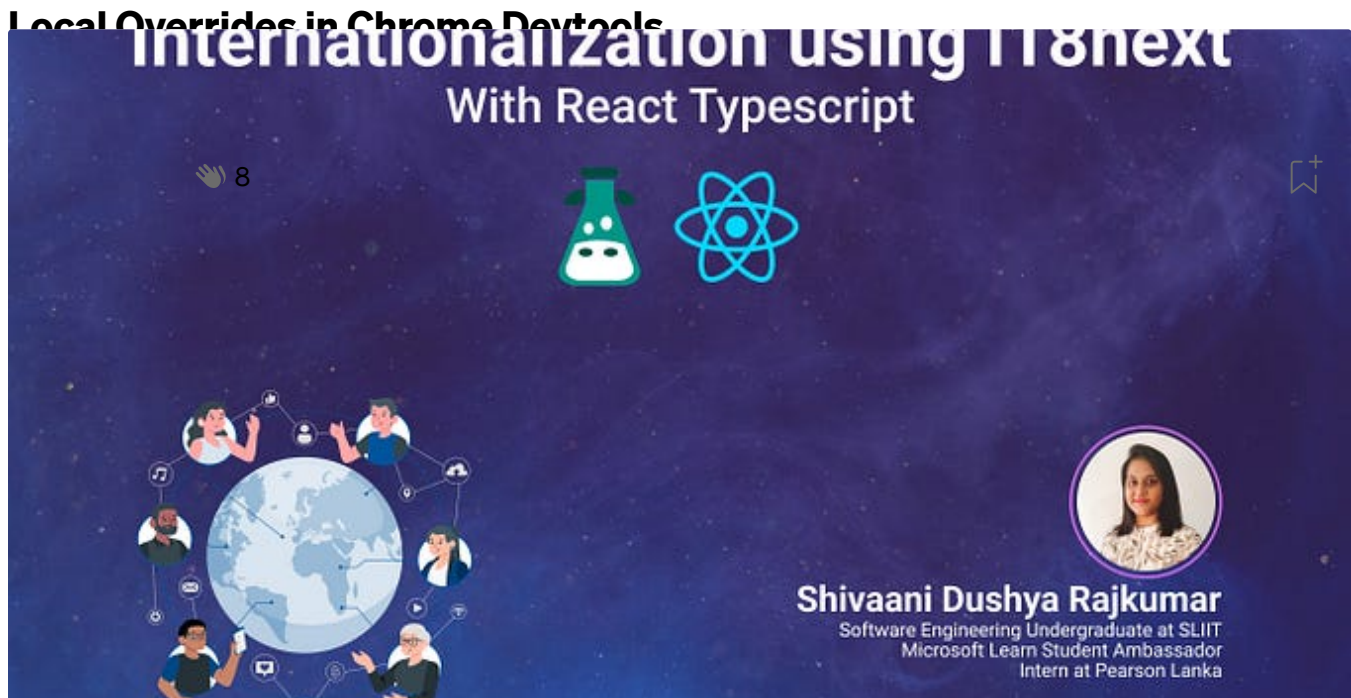
More from Senura Vihan Jayadeva and MS Club of SLIIT





Senura Vihan Jayadeva

Local Overrides in Chrome Devtools

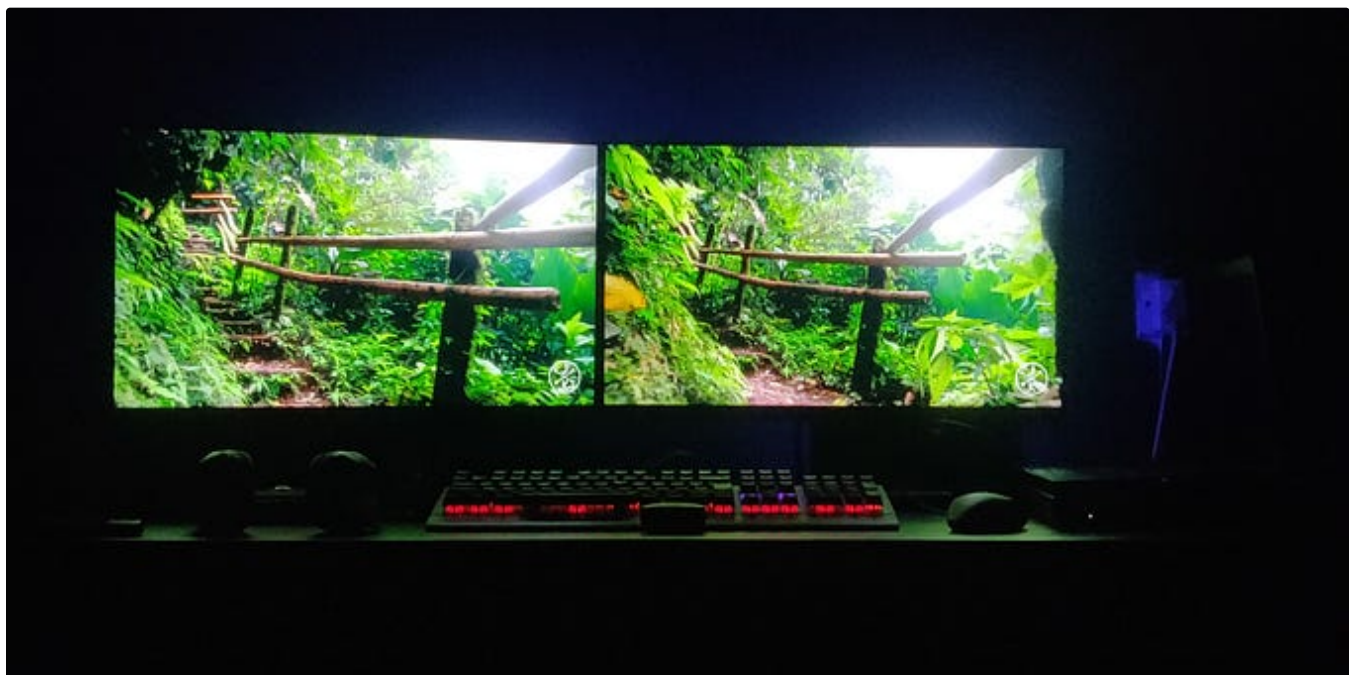


In MS Club of SLIIT by Shivaani Dushya

Internationalization using i18next with React Typescript

Hi all,

Aug 2, 2022 🖱️ 203





In MS Club of SLIIT by Gihan Siriwardhana

How to Build a Two Laptops Two Monitors Setup ?

I always loved to build a dual monitor, dual laptop setup. After doing some intensive research I found several ways to achieve that.

Sep 5, 2021 🖱 95



Microfrontend with Webpack Module Federation

**Senura Vihan Jayadeva**

Medium.com/@senuravihanjayadeva



Senura Vihan Jayadeva

Microfrontend with Webpack Module Federation

Hello everyone, in this write-up, I'll provide a brief overview of Microfrontends and subsequently dive into Module Federation, explaining...

Mar 8, 2023 🖱 20 💬 1

[See all from Senura Vihan Jayadeva](#)[See all from MS Club of SLIIT](#)

Recommended from Medium



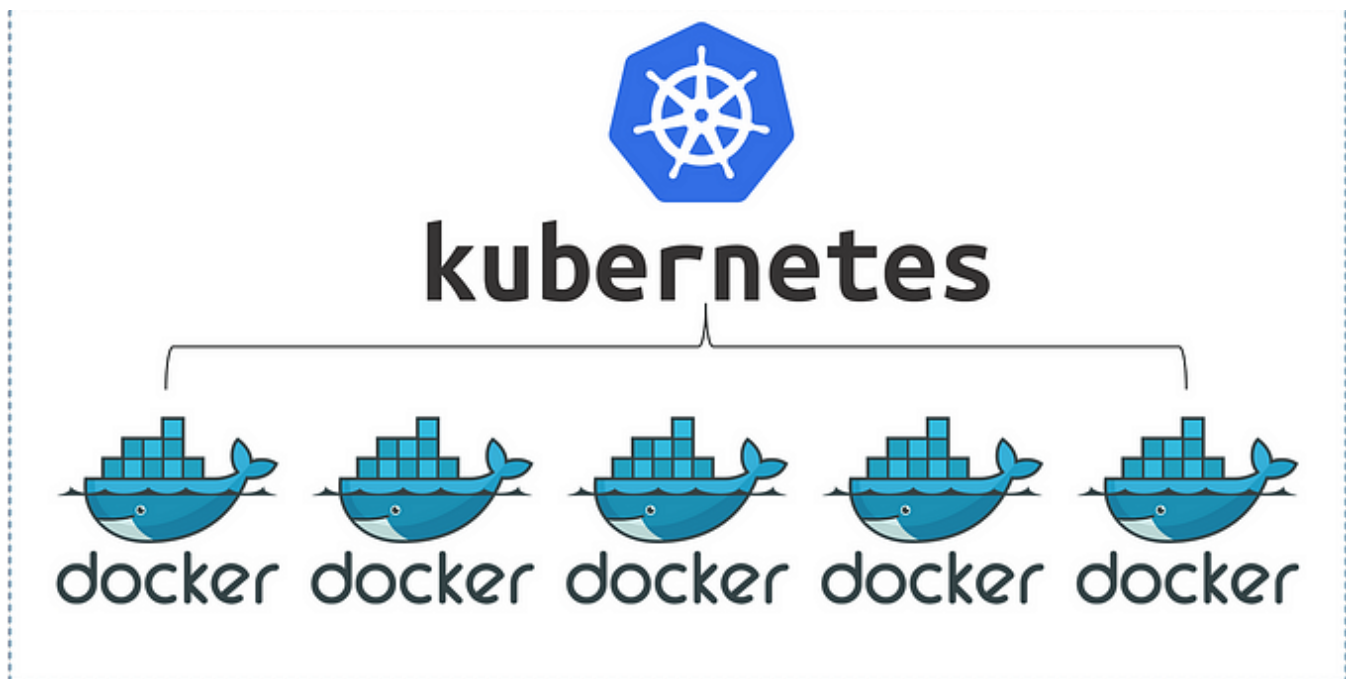
 Full Stack Developer

@RequestParam vs @QueryParam vs @PathParam vs @PathVariable in REST

Pretty confusing if you don't know this simple fact.

★ Sep 3 🖱 13





Wend-Kuuni Etienne BELEMGNEGRE

Mastering Microservices with Spring Boot : Docker and Kubernetes [Part 3]

* Note: This article is based on Spring Boot 3.2.x Please ensure compatibility with your project's requirements. * Here's the previous ...

Jun 27 🖱 56



Lists



Staff picks

772 stories · 1457 saves



Stories to Help You Level-Up at Work

19 stories · 873 saves



Self-Improvement 101

20 stories · 3061 saves



Productivity 101

20 stories · 2583 saves

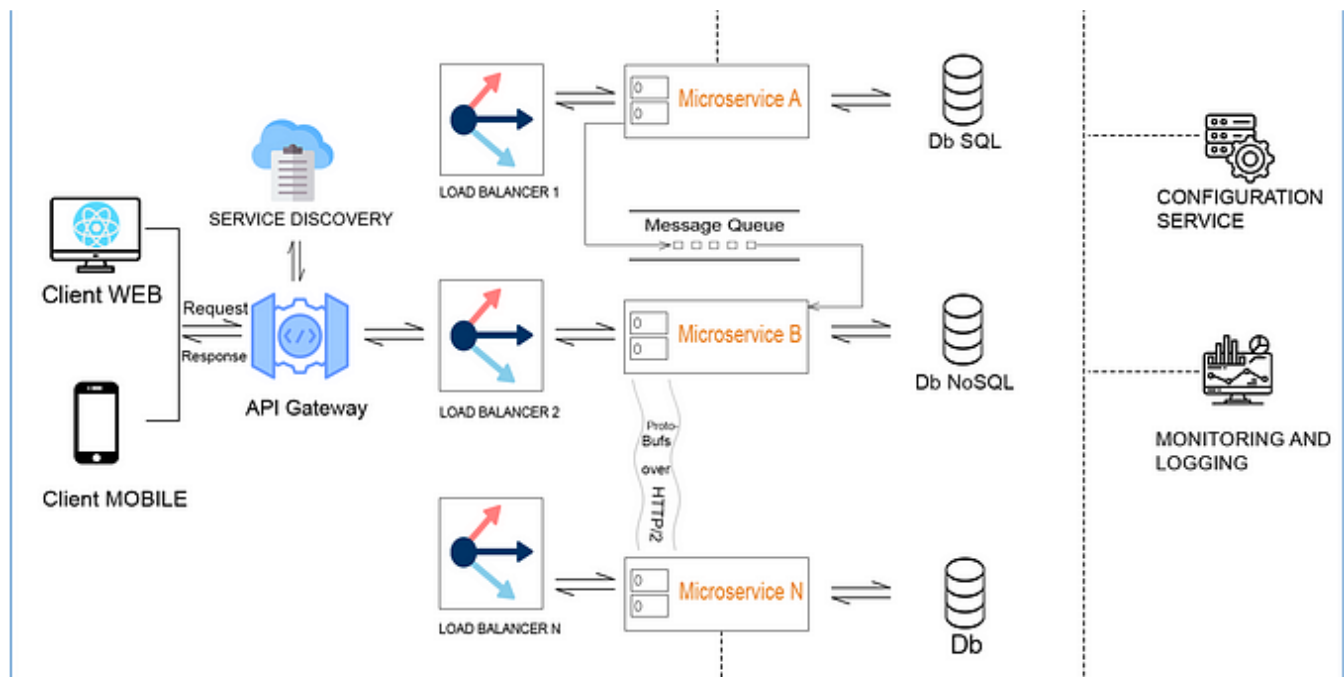
zomato

 Sahil Kumar

Zomato SDE Interview Experience

When I applied for a Software Development Engineer (SDE) role at Zomato through getjobs.today, I was thrilled to explore an opportunity...

★ 6d ago 🖱 14 💬 1





In Technology Hits by Atul Kumar

Microservices Simplified

Essential Insights for Modern Software Development



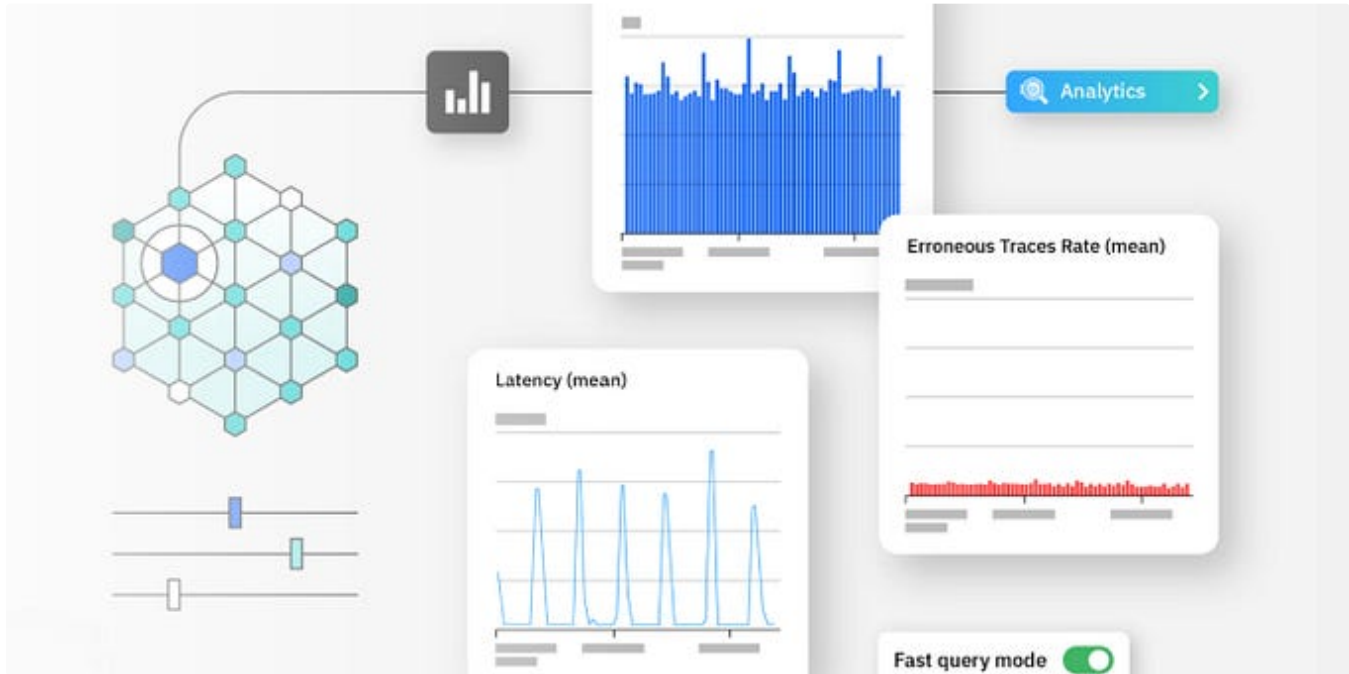
Nov 11



205



5



In DevOps.dev by Rahul Soni

Top 10 Concepts for Java Spring Boot Developers Part-2 (Distributed Tracing in Microservices)

Introduction



Nov 9




17



1





 Eric Anicet

Custom Banner in Spring Boot

In this story, we'll explore how to create a custom banner in a Spring Boot application.

★ Nov 14 🖱 13



See more recommendations