✅ [**What are Microservices and How to Build Microservices in Java?**](https://www.javaguides.net/2022/12/what-is-microservices-in-java.html)

✅ [**Spring Boot Microservices Architecture**](https://www.javaguides.net/2023/01/spring-boot-microservices-architecture.html)

✅ [**What is Spring Cloud?**](https://www.javaguides.net/2022/12/what-is-spring-cloud.html)

✅ [**Spring Cloud Interview Questions**](https://www.javaguides.net/2022/12/spring-cloud-interview-questions.html)

✅ [**Spring Boot Microservices Communication Example using RestTemplate**](https://www.javaguides.net/2022/10/spring-boot-microservices-communication-using-resttemplate.html)

✅ [**Spring Boot Microservices Communication Example using WebClient**](https://www.javaguides.net/2022/10/spring-boot-microservices-communication-using-webclient.html)

✅ [**Spring Boot Microservices Communication Example using Spring Cloud Open Feign**](https://www.javaguides.net/2022/10/spring-boot-microservices-communication-using-spring-cloud-open-feign.html)

✅ [**Spring Boot Microservices - Spring Cloud Config Server**](https://www.javaguides.net/2022/10/spring-boot-microservices-spring-cloud-config-server.html)

✅ [**Spring Boot Microservices - Spring Cloud Netflix Eureka-based Service Registry**](https://www.javaguides.net/2022/10/spring-boot-microservices-spring-cloud-netflix-eureka-.html)

✅ [**Spring Boot Microservices - Spring Cloud API Gateway**](https://www.javaguides.net/2022/10/spring-boot-microservices-spring-cloud-api-gateway.html)

✅  [**Event-Driven Microservices using Spring Boot and Kafka**](https://www.javaguides.net/2022/07/event-driven-microservices-using-spring-boot-and-apache-kafka.html)

✅ [**Microservices Project using Spring Boot**](https://www.javaguides.net/2023/04/microservices-using-spring-boot.html)

**Spring Boot Microservices Shopping Cart Project**

✅ Part 1 - [**Spring Boot Microservices Shopping Cart Example**](https://www.javaguides.net/2022/11/spring-boot-microservices-shopping-cart.html)

✅ Part 2 - [**Spring Cloud OpenFeign**](https://www.javaguides.net/2022/11/spring-cloud-openfeign.html)

✅ Part 3 - [**Spring Cloud Netflix Eureka**](https://www.javaguides.net/2022/11/spring-cloud-netflix-eureka.html)

**What are Microservices?**

Microservices are a software architectural style in which a large application is built as a collection of small, independent services that communicate with each other over a network.

Each service is a self-contained unit of functionality that can be developed, tested, and deployed independently of the other services. This allows for more flexibility and scalability than a monolithic architecture, where all the functionality is contained in a single, large codebase.

Microservices can be written in different programming languages and use different technologies, as long as they can communicate with each other through a common API.

They are designed to be loosely coupled, meaning that changes to one service should not affect the other services. This makes it easier to update, maintain, and scale the application. Microservices architecture is best suited for large and complex applications that need to handle a high volume of traffic and need to be scaled horizontally.

# Key Components of a Microservices Architecture

Key components of a microservices architecture include:

1. **Core Services**: Each service is a self-contained unit of functionality that can be developed, tested, and deployed independently of the other services.
2. **Service registry**: A service registry is a database of all the services in the system, along with their locations and capabilities. It allows services to discover and communicate with each other.
3. **API Gateway:** An API gateway is a single entry point for all incoming requests to the microservices. It acts as a reverse proxy, routing requests to the appropriate service and handling tasks such as authentication and rate limiting.
4. **Message bus:** A message bus is a messaging system that allows services to communicate asynchronously with each other. This can be done through protocols like HTTP, RabbitMQ, or Kafka.
5. **Monitoring and logging:** Monitoring and logging are necessary to track the health of the services and troubleshoot problems.
6. **Service discovery and load balancing:** This component is responsible for discovering service instances and directing traffic to the appropriate service instances based on load and availability.
7. **Continuous integration and continuous deployment (CI/CD):**To make the development and deployment process of microservices as smooth as possible, it is recommended to use a tool such as Jenkins, TravisCI, or CircleCI to automate the process of building, testing, and deploying microservices.

# What are Microservices or Microservice Architecture?

Well, a microservice architecture enables large teams to build scalable applications that are composed of many loosely coupled services.

Here is what a typical microservice architecture looks like. For example, consider this microservice architecture for a simple shopping cart application. It has different services like product service, inventory service, and stock service, and these are the independent and loosely coupled services in the microservices projects.

Each microservice has its own database. For example, product service has its own database, inventory service has its own database, and stock service has its own database.

In the microservices project, all the microservices are loosely coupled. So loosely coupled, meaning all the services in a microservices project are independent of each other and each microservice should be developed independently and each microservice should be deployed independently and each microservice should be scaled independently.

So basically Microservice following characteristics:

* Each microservice can have its own database.
* Each microservice should be developed independently
* Each microservice should be deployed independently
* Each microservice should be scaled independently

In microservices projects, the services can communicate with each other. For example, product service can communicate with inventory service and inventory service can communicate with stock service.  Microservice can communicate with multiple services as well.

Well, there are two types of communication styles. One is synchronous and another is asynchronous.

In the case of synchronous, we can use the HTTP protocol to make an HTTP request from one microservice to the microservice.

And in the case of asynchronous communication, we have to use a message broker for asynchronous communication between multiple microservices. For example, we can use a RabbitMQ or Apache Kafka as a message broker in order to make an asynchronous communication between multiple microservices and each microservice in a microservices project can expose REST API's.

## Key Components in a Microservices Architecture

Now let's take a look into the key components in a typical microservices architecture.

Well, the key component is the API gateway. Well, whenever the client sends a request to the API gateway and then an API gateway will route that request to the relevant microservices All right.

The client can be a web application, a mobile application, or a desktop application and whenever a client wants to consume the REST API's of backend services, the client has to first send a request to the API gateway, and then the API gateway will route that request to the relevant microservice.

Here one more key component is a service registry. Well, all the microservices in our microservice project will register to the service registry, and the API gateway will discover the particular microservice hostname and port using the service registry so that the API gateway can allow that request to a particular microservice.

One more key component is the config server. So this config server component will basically externalize the configuration of microservices.

One more key component is distributed tracing. Well, in order to maintain the logs or complete log hierarchy for a particular HTTP call from start to end, we can use distributed tracing.

One more key component is Security. We can implement centralized security in API-Gateway.

So these are the few key components in a microservices architecture.

### Spring Boot Microservices Architecture:

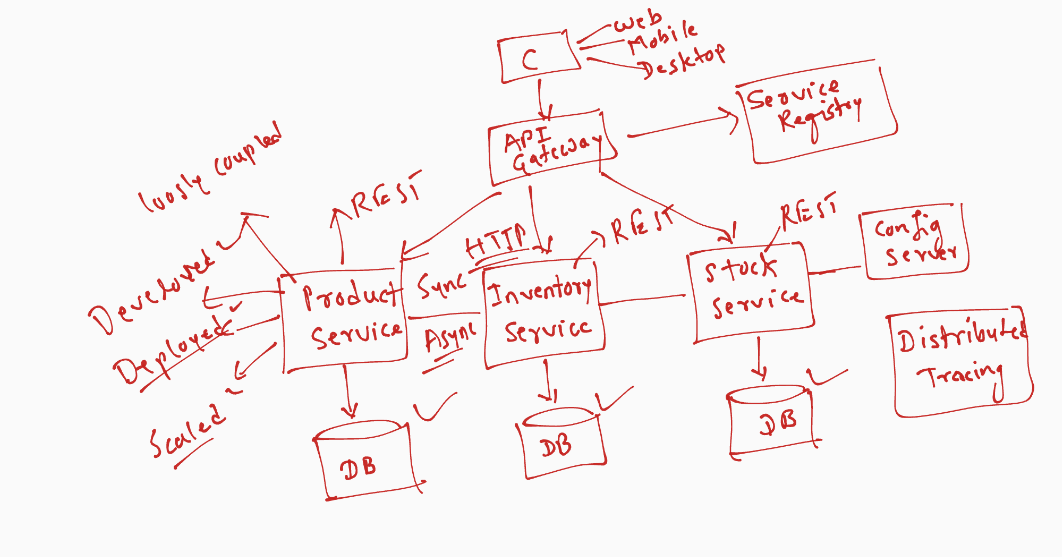
In this tutorial, we will discuss how to create a typical microservices architecture using Spring boot and Spring cloud.

**Spring Boot** is a very popular Java framework for building Restful web services and microservices. The main goal of Spring Boot is to quickly create Spring-based applications without requiring developers to write the same boilerplate configuration again and again.  
  
**Spring Cloud** provides various tools or modules for developers to build common design patterns to solve different infrastructural concerns in Microservices projects and focus on their main business problems.

In the Java community, Spring Boot and Spring Cloud become a de-facto standard for building microservices architecture.

# What are Microservices or Microservices Architecture?

Well, a microservice architecture enables large teams to build scalable applications that are composed of many loosely coupled services.  
  
Here is what a typical microservice architecture looks like. For example, consider this microservice architecture for a simple shopping cart application. It has different services like **product service**, **inventory service**, and **stock service**, and these are the independent and loosely coupled services in the microservices projects.

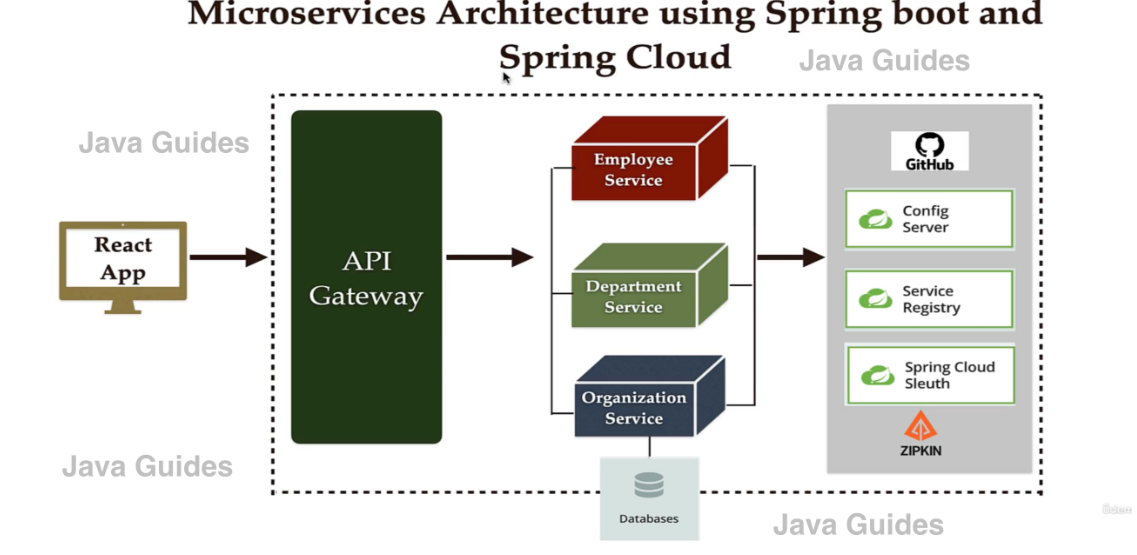


Each microservice has its own database. For example, **product service** has its own database, **inventory service** has its own database, and **stock service** has its own database.  
  
In the microservices project, all the microservices are loosely coupled. So loosely coupled, meaning all the services in a microservices project are independent of each other and each microservice should be developed independently and each microservice should be deployed independently and each microservice should be scaled independently.  
  
So basically Microservice following characteristics:

* Each microservice can have its own database.
* Each microservice should be developed independently
* Each microservice should be deployed independently
* Each microservice should be scaled independently

**Microservices Architecture using Spring Boot and Spring Cloud**

Well, we are going to use the **Employee Management**Project as an example to create a microservices architecture.



Well, you can take any example of any project. For example, you can take an e-commerce application or you can take any healthcare domain-related application. So you can take any project as an example, but follow the same steps to create a microservice architecture using Spring Boot and Spring Cloud.

## Core Microservices

Consider we have developed three core backend Spring boot microservices such as **employee service**, **department service,** and **organization service,** and all these three microservices have their own databases. You can use a relational database or NoSQL database as a database for these microservices. So whenever you create a microservice in your project, make sure that each microservice should have its own database. All right.

## Microservices Communication

Once we build these 3 microservices. Next, we'll see how these microservices communicate with each other. Well, there are different ways to make a REST API call from one microservice to another Microservice. For example, we can use a RestTemplate or WebClient or Spring cloud-provided open feign library. All right.  
  
Well, there are two types of communication styles. One is synchronous and another is asynchronous.  
  
In the case of synchronous, we can use the HTTP protocol to make an HTTP request from one microservice to the microservice.  
  
And in the case of asynchronous communication, we have to use a message broker for asynchronous communication between multiple microservices. For example, we can use RabbitMQ or Apache Kafka as a message broker in order to make an asynchronous communication between multiple microservices and each microservice in a microservices project can expose REST APIs.

## Registry and Discovery Pattern

Well, once we know how microservices communicate with each other, next you need to know how to implement a service Registry and discovery pattern in our microservices project.

Well, Spring Cloud provides a [**Spring Cloud Netflix Eureka Based Service Registry**](https://www.javaguides.net/2022/10/spring-boot-microservices-spring-cloud-netflix-eureka-.html) module that we can use to implement service registry and discovery patterns in our microservices project. Well, service Registry and discovery is a really essential pattern that we can use to avoid hard coding hostnames and ports.

## Config-Server to Externalize the Configurations

Next, we will implement a config server to externalize the configurations of all these three microservices into a central place which is the git repository.

Well, Spring cloud provides a Spring cloud config module that we can use to implement a config server to externalize the configuration files of all these three microservices into a central place. We are going to use the git repository as storage for the config server.

## API Gateway Pattern

Well, once we know how to use the config server to externalize the configuration files. Next, we have to implement an API gateway.

Well, API Gateway plays a very important role in our microservices architecture. So whenever a client wants to make a call to different microservices, the client has to remember the host names and ports of all these microservices. So there should be a solution where a client can send a request to the central component so that is where the API gateway comes into the picture. So whenever a client sends a request to the backend microservices, then the client has to send a request to the API gateway first, and then the API gateway based on the routing rules will route that request to the appropriate microservice. So this is how the API gateway plays an important role in a microservices architecture.

Well, Spring Cloud provides [**Spring Cloud Gateway module**](https://www.javaguides.net/2022/10/spring-boot-microservices-spring-cloud-api-gateway.html) to implement API gateway patterns in a microservices architecture.

## Distributed Tracing

Next, once you know how to implement an API gateway in a microservices project, next you can implement distributed tracing in a microservices architecture. Well, Spring Cloud provides a Spring Cloud sleuth module, which we can use to implement distributed tracing in our microservices project.  
  
Well, along with Spring Cloud Sleuth, we'll also use Zipkin to visualize the tracing log information in a user interface. Well, Zipkin provides a user interface to track and trace information through web applications.

## React Front Service

Next, you can use React/Angular to create a client-side service that will make a call to backend microservices.

## Circuit Breaker Pattern

Next, you can implement a circuit breaker pattern in an employee service because the employee service is internally calling department service, and let's say due to some reason, department service is down then employee service won't get a response from the department server, isn't it? And then again, employee service will send an internal server error to the API gateway and then API Gateway will send that response back to the client. All right. So in order to avoid this kind of issue, we can use a circuit breaker pattern.  
  
So this circuit breaker pattern helps the employee service to avoid continuous calls to the department service Whenever department service is done and this circuit breaker pattern will help employee service to return some default response back to the API Gateway and the API Gateway will send that default response to the client.  
  
All right. So this is a simple microservices architecture using Spring Boot and Spring Cloud.

### What is Spring Cloud?

Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems or Microservices projects.  
  
For example, common patterns such as configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, and cluster state.  
  
**Spring Cloud provides various tools or modules for developers to build common design patterns to solve different infrastructural concerns in Microservices projects and focus on their main business problems.**

**Notable Spring Cloud Features**

Spring Cloud offers below features:

* Distributed/versioned configuration
* Service registration and discovery
* Routing
* Service-to-service calls
* Load balancing
* Circuit Breakers
* Distributed messaging
* API Gateway
* Distributed tracing

[**Spring Boot Microservices Communication Example using RestTemplate**](https://www.javaguides.net/2022/10/spring-boot-microservices-communication-using-resttemplate.html):

In this tutorial, we will learn how to create multiple Spring boot microservices and how to use RestTemplate class to make Synchronous communication between multiple microservices.

There are two styles of Microservices Communications:

1. Synchronous Communication
2. Asynchronous Communication

**Synchronous Communication**

In the case of Synchronous Communication, the client sends a request and waits for a response from the service. The important point here is that the protocol (HTTP/HTTPS) is synchronous and the client code can only continue its task when it receives the HTTP server response.

For example, **Microservice1 acts as a client that sends a request and waits for a response from Microservice2.**

We can use RestTemplate or WebClient or Spring Cloud Open Feign library to make a Synchronous Communication multiple microservices.

**Asynchronous Communication**

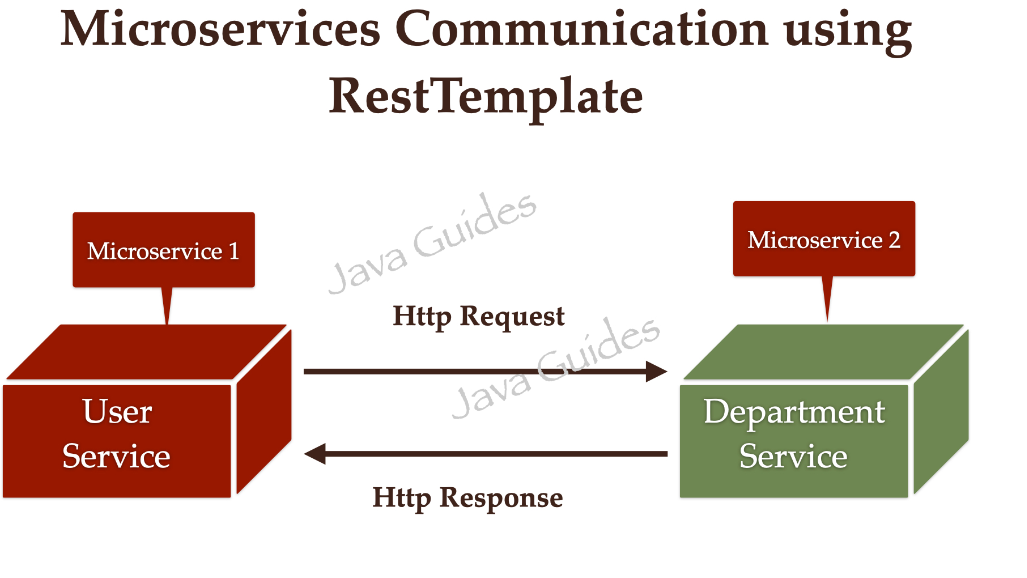
In the case of Asynchronous Communication, The client sends a request and does not wait for a response from the service. The client will continue executing its task - It doesn’t wait for the response from the service.

For example, **Microservice1 acts as a client that sends a request and doesn't wait for a response from Microservice2.**

We can use Message brokers such as RabbitMQ and Apache Kafka to make Asynchronous Communication between multiple microservices.

# What we will Build?

Well, we will create two microservices such as *department-service* and *user-service* and we will make a REST API call from *user-service* to *department-service* to fetch a particular user department.



We will create a separate MySQL database for each microservice.

We will create and set up two Spring boot projects as two microservices in IntelliJ IDEA.

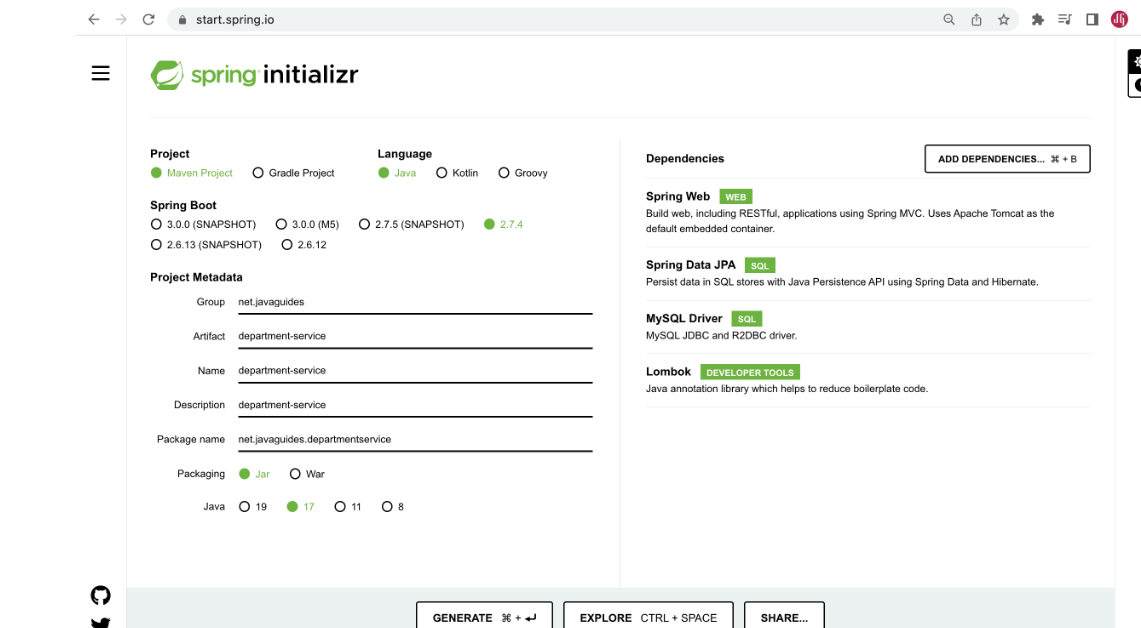
# Creating DepartmentService Microservice

Let's first create and setup the *department-service* Spring boot project in IntelliJ IDEA

## 1. Create and setup spring boot project (department-service) in IntelliJ IDEA

Let's create a Spring boot project using the [**spring initializr**](https://start.spring.io/).

Refer to the below screenshot to enter details while creating the spring boot application using the [**spring initializr**](https://start.spring.io/):



Click on Generate button to download the Spring boot project as a zip file. Unzip the zip file and import the Spring boot project in IntelliJ IDEA.

Here is the *pom.xml* file for your reference:

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.7.4</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>net.javaguides</groupId>

<artifactId>department-service</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>department-service</name>

<description>department-service</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>mysql</groupId>

<artifactId>mysql-connector-java</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>

## DepartmentService - Configure MySQL Database

Since we’re using MySQL as our database, we need to configure the URL, username, and password so that our Spring boot can establish a connection with the database on startup.

Open the src/main/resources/application.properties file and add the following properties to it:

spring.datasource.url=jdbc:mysql://localhost:3306/department\_db

spring.datasource.username=root

spring.datasource.password=Mysql@123

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

spring.jpa.hibernate.ddl-auto=update

Don’t forget to change the spring.datasource.username and spring.datasource.password as per your MySQL installation. Also, create a database named **department\_db** in MySQL before proceeding to the next section.

You don’t need to create any tables. The tables will automatically be created by Hibernate from the Departmententity that we will define in the next step. This is made possible by the property spring.jpa.hibernate.ddl-auto = update.

## DepartmentService - Create Department JPA Entity

package net.javaguides.departmentservice.entity;

import javax.persistence.\*;

import lombok.AllArgsConstructor;

import lombok.Getter;

import lombok.NoArgsConstructor;

import lombok.Setter;

@Entity

@Table(name = "departments")

@NoArgsConstructor

@AllArgsConstructor

@Setter

@Getter

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String departmentName;

private String departmentAddress;

private String departmentCode;

}

## DepartmentService - Create Spring Data JPA Repository

package net.javaguides.departmentservice.repository;

import net.javaguides.departmentservice.entity.Department;

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

public interface DepartmentRepository extends JpaRepository<Department, Long> {

}

## DepartmentService - Create Service Layer

#### DepartmentService Interface

package net.javaguides.departmentservice.service;

import net.javaguides.departmentservice.entity.Department;

public interface DepartmentService {

Department saveDepartment(Department department);

Department getDepartmentById(Long departmentId);

}

#### DepartmentServiceImpl class

package net.javaguides.departmentservice.service.impl;

import lombok.AllArgsConstructor;

import lombok.extern.slf4j.Slf4j;

import net.javaguides.departmentservice.entity.Department;

import net.javaguides.departmentservice.repository.DepartmentRepository;

import net.javaguides.departmentservice.service.DepartmentService;

import org.springframework.stereotype.Service;

@Service

@AllArgsConstructor

@Slf4j

public class DepartmentServiceImpl implements DepartmentService {

private DepartmentRepository departmentRepository;

@Override

public Department saveDepartment(Department department) {

return departmentRepository.save(department);

}

@Override

public Department getDepartmentById(Long departmentId) {

return departmentRepository.findById(departmentId).get();

}

}

### DepartmentService - Create Controller Layer: DepartmentController

package net.javaguides.departmentservice.controller;

import lombok.AllArgsConstructor;

import net.javaguides.departmentservice.entity.Department;

import net.javaguides.departmentservice.service.DepartmentService;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("api/departments")

@AllArgsConstructor

public class DepartmentController {

private DepartmentService departmentService;

@PostMapping

public ResponseEntity<Department> saveDepartment(@RequestBody Department department){

Department savedDepartment = departmentService.saveDepartment(department);

return new ResponseEntity<>(savedDepartment, HttpStatus.CREATED);

}

@GetMapping("{id}")

public ResponseEntity<Department> getDepartmentById(@PathVariable("id") Long departmentId){

Department department = departmentService.getDepartmentById(departmentId);

return ResponseEntity.ok(department);

}

}

## DepartmentService - Start Spring Boot Application

Two ways we can start the standalone Spring boot application.

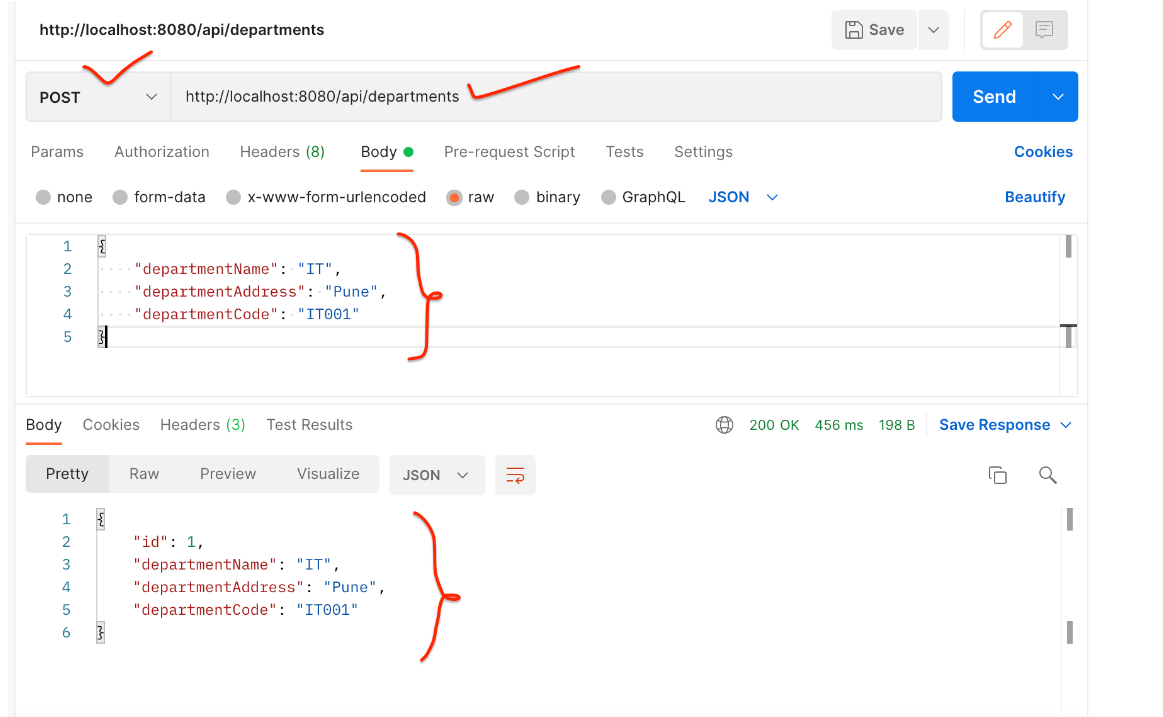
1. From the root directory of the application and type the following command to run it -

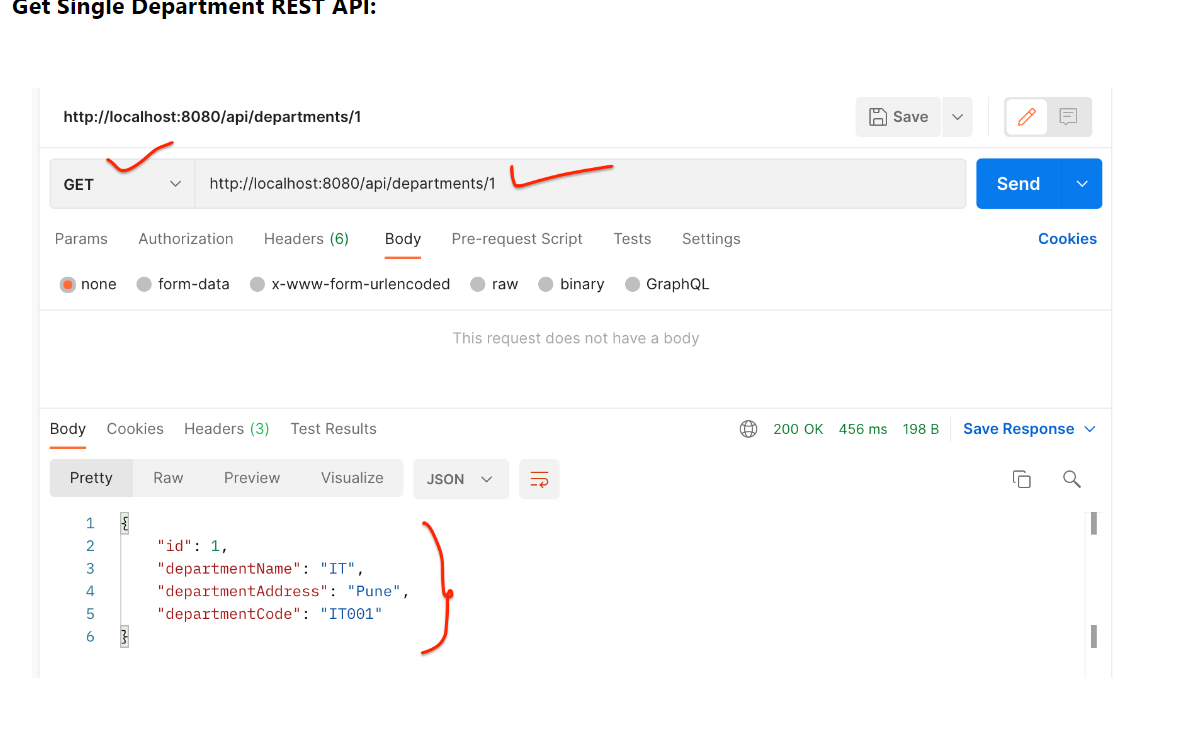
$ mvn spring-boot:run

2. From your IDE, run the *DepartmentServiceApplication.main()* method as a standalone Java class that will start the embedded Tomcat server on port 8080 and point the browser to [**http://localhost:8080/**](http://localhost:8080/).

## DepartmentService - Test REST APIs using Postman Client

### Save Department REST API:





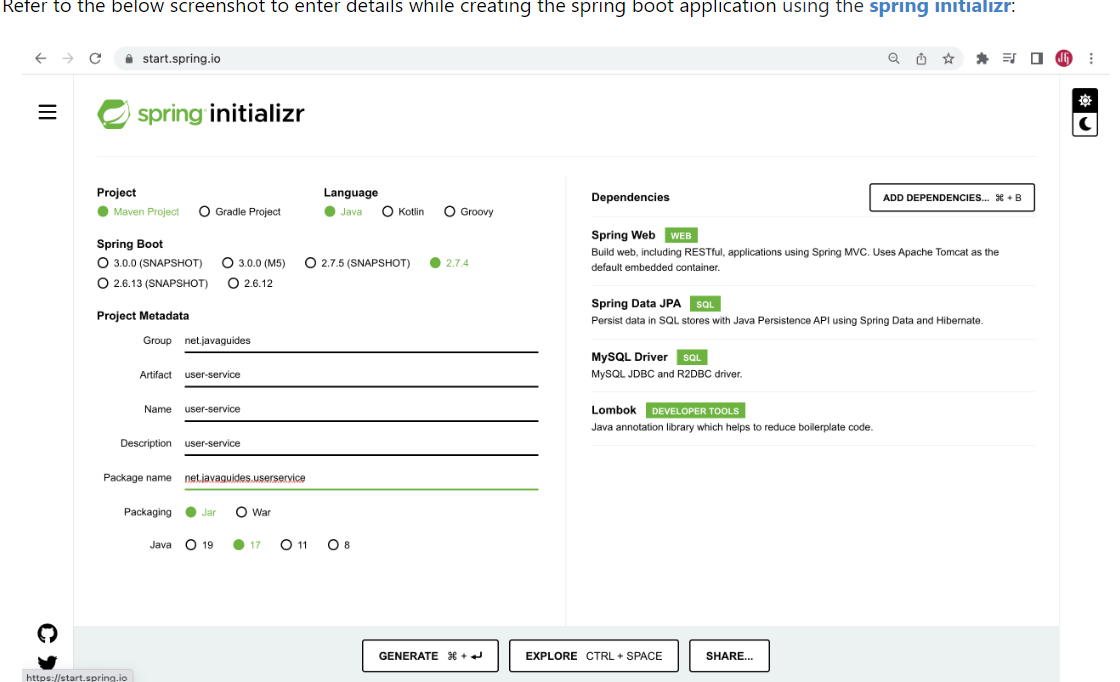
# 2. Creating UserService Microservice

Let's first create and setup the *user-service* Spring boot project in IntelliJ IDEA

## 1. Create and setup spring boot project (user-service) in IntelliJ IDEA

Let's create a Spring boot project using the [**spring initializr**](https://start.spring.io/).

Refer to the below screenshot to enter details while creating the spring boot application using the [**spring initializr**](https://start.spring.io/):



Click on Generate button to download the Spring boot project as a zip file. Unzip the zip file and import the Spring boot project in IntelliJ IDEA.

Here is the *pom.xml* file for your reference:

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.7.4</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>net.javaguides</groupId>

<artifactId>user-service</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>user-service</name>

<description>user-service</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>mysql</groupId>

<artifactId>mysql-connector-java</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>

## UserService - Configure MySQL Database

Open the src/main/resources/application.properties file and add the following properties to it:

spring.datasource.url=jdbc:mysql://localhost:3306/employee\_db

spring.datasource.username=root

spring.datasource.password=Mysql@123

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

spring.jpa.hibernate.ddl-auto=update

Don’t forget to change the spring.datasource.username and spring.datasource.password as per your MySQL installation. Also, create a database named **employee\_db** in MySQL before proceeding to the next section.

You don’t need to create any tables. The tables will automatically be created by Hibernate from the Userentity that we will define in the next step. This is made possible by the property spring.jpa.hibernate.ddl-auto = update.

## UserService - Change the Server Port

Note that the department service Spring boot project is running on the default tomcat server port 8080.

For user service, we need to change the embedded tomcat server port to 8081 using the below property:

server.port = 8081

## UserService - Create User JPA Entity

package net.javaguides.userservice.entity;

import javax.persistence.\*;

import lombok.AllArgsConstructor;

import lombok.Getter;

import lombok.NoArgsConstructor;

import lombok.Setter;

@Entity

@Table(name = "users")

@Setter

@Getter

@NoArgsConstructor

@AllArgsConstructor

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String firstName;

private String lastName;

@Column(nullable = false, unique = true)

private String email;

private String departmentId;

}

## UserService - Create Spring Data JPA Repository

package net.javaguides.userservice.repository;

import net.javaguides.userservice.entity.User;

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

public interface UserRepository extends JpaRepository<User, Long> {

}

## UserService - Create DTO Classes

#### DepartmentDto

package net.javaguides.userservice.dto;

import lombok.AllArgsConstructor;

import lombok.Getter;

import lombok.NoArgsConstructor;

import lombok.Setter;

@Setter

@Getter

@AllArgsConstructor

@NoArgsConstructor

public class DepartmentDto {

private Long id;

private String departmentName;

private String departmentAddress;

private String departmentCode;

}

#### UserDto

package net.javaguides.userservice.dto;

import lombok.AllArgsConstructor;

import lombok.Getter;

import lombok.NoArgsConstructor;

import lombok.Setter;

@Setter

@Getter

@NoArgsConstructor

@AllArgsConstructor

public class UserDto {

private Long id;

private String firstName;

private String lastName;

private String email;

}

#### ResponseDto

package net.javaguides.userservice.dto;

import lombok.AllArgsConstructor;

import lombok.Getter;

import lombok.NoArgsConstructor;

import lombok.Setter;

@Setter

@Getter

@NoArgsConstructor

@AllArgsConstructor

public class ResponseDto {

private DepartmentDto department;

private UserDto user;

}

## UserService - Configure RestTemplate as Spring Bean

Let's configure RestTemplate class as Spring bean so that we can inject and use it.

package net.javaguides.userservice;

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 UserServiceApplication {

public static void main(String[] args) {

SpringApplication.run(UserServiceApplication.class, args);

}

@Bean

public RestTemplate restTemplate(){

return new RestTemplate();

}

}

## UserService - Create Service Layer

#### UserService Interface

package net.javaguides.userservice.service;

import net.javaguides.userservice.dto.ResponseDto;

import net.javaguides.userservice.entity.User;

public interface UserService {

User saveUser(User user);

ResponseDto getUser(Long userId);

}

#### UserServiceImpl class

package net.javaguides.userservice.service.impl;

import lombok.AllArgsConstructor;

import net.javaguides.userservice.dto.DepartmentDto;

import net.javaguides.userservice.dto.ResponseDto;

import net.javaguides.userservice.dto.UserDto;

import net.javaguides.userservice.entity.User;

import net.javaguides.userservice.repository.UserRepository;

import net.javaguides.userservice.service.UserService;

import org.springframework.http.ResponseEntity;

import org.springframework.stereotype.Service;

import org.springframework.web.client.RestTemplate;

@Service

@AllArgsConstructor

public class UserServiceImpl implements UserService {

private UserRepository userRepository;

private RestTemplate restTemplate;

@Override

public User saveUser(User user) {

return userRepository.save(user);

}

@Override

public ResponseDto getUser(Long userId) {

ResponseDto responseDto = new ResponseDto();

User user = userRepository.findById(userId).get();

UserDto userDto = mapToUser(user);

ResponseEntity<DepartmentDto> responseEntity = restTemplate

.getForEntity("http://localhost:8080/api/departments/" + user.getDepartmentId(),

DepartmentDto.class);

DepartmentDto departmentDto = responseEntity.getBody();

System.out.println(responseEntity.getStatusCode());

responseDto.setUser(userDto);

responseDto.setDepartment(departmentDto);

return responseDto;

}

private UserDto mapToUser(User user){

UserDto userDto = new UserDto();

userDto.setId(user.getId());

userDto.setFirstName(user.getFirstName());

userDto.setLastName(user.getLastName());

userDto.setEmail(user.getEmail());

return userDto;

}

}

Note that we are using *RestTemplate* to make a REST API call to department-service:

ResponseEntity<DepartmentDto> responseEntity = restTemplate

.getForEntity("http://localhost:8080/api/departments/" + user.getDepartmentId(),

DepartmentDto.class);

## UserService - Create Controller Layer: UserController

package net.javaguides.userservice.controller;

import lombok.AllArgsConstructor;

import net.javaguides.userservice.dto.ResponseDto;

import net.javaguides.userservice.entity.User;

import net.javaguides.userservice.service.UserService;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("api/users")

@AllArgsConstructor

public class UserController {

private UserService userService;

@PostMapping

public ResponseEntity<User> saveUser(@RequestBody User user){

User savedUser = userService.saveUser(user);

return new ResponseEntity<>(savedUser, HttpStatus.CREATED);

}

@GetMapping("{id}")

public ResponseEntity<ResponseDto> getUser(@PathVariable("id") Long userId){

ResponseDto responseDto = userService.getUser(userId);

return ResponseEntity.ok(responseDto);

}

}

## UserService - Start Spring Boot Application

Two ways we can start the standalone Spring boot application.

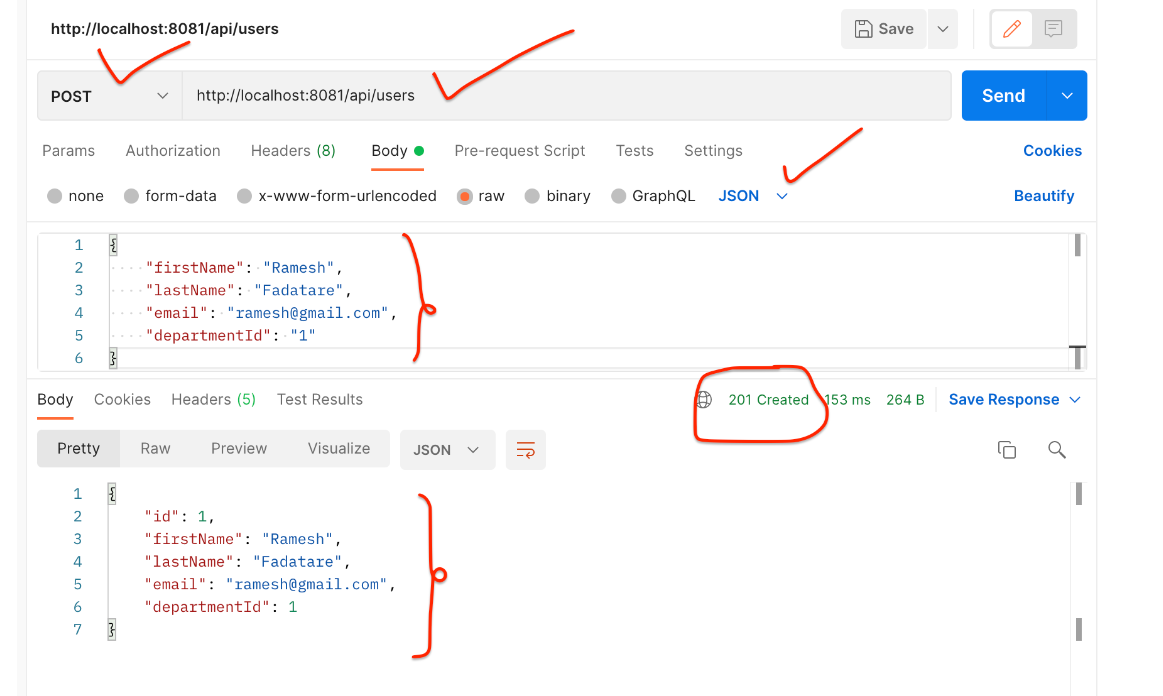
1. From the root directory of the application and type the following command to run it -

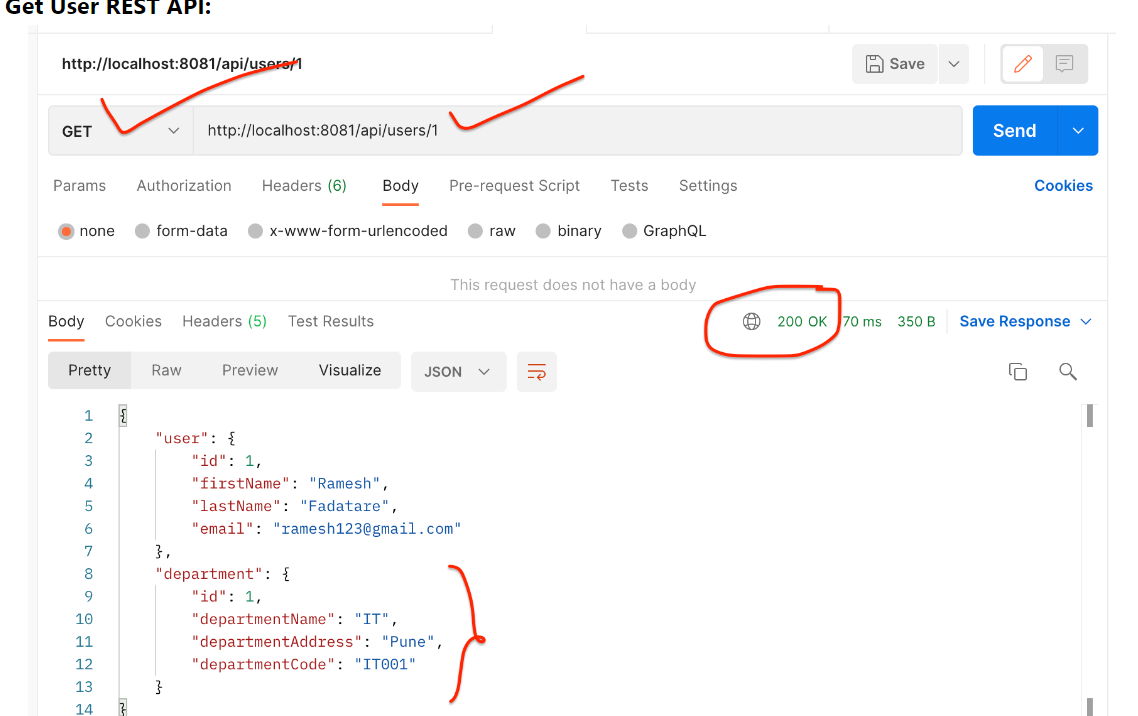
$ mvn spring-boot:run

2. From your IDE, run the *UserServiceApplication.main()* method as a standalone Java class that will start the embedded Tomcat server on port 8080 and point the browser to [**http://localhost:8081/**](http://localhost:8081/).

## UserService - Test REST APIs using Postman Client

### Save User REST API:





### Spring Boot Microservices Communication Example using WebClient

In the previous tutorial, we have seen the [**Spring Boot Microservices Communication Example using RestTemplate**](https://www.javaguides.net/2022/10/spring-boot-microservices-communication-using-resttemplate.html).

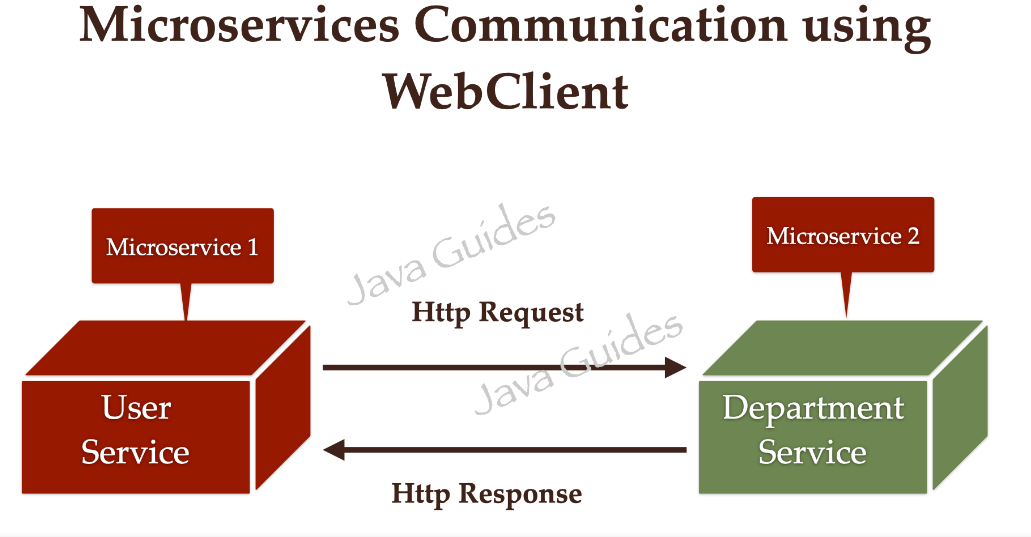
As of 5.0, the *RestTemplate* class is in maintenance mode and soon will be deprecated. So the Spring team recommended using *org.springframework.web.reactive.client.WebClient*that has a modern API and supports sync, async, and streaming scenarios.  
  
In this tutorial, we will learn how to use *WebClient*to make REST API calls (Synchronous communication) between multiple microservices.

*WebClient* is a non-blocking, reactive client to perform HTTP requests, exposing a fluent, reactive API over underlying HTTP client libraries such as Reactor Netty.

To use *WebClient* in our Spring boot project, we have to add *Spring WebFlux* dependency to the classpath.

# What we will Build?

Well, we will create two microservices such as department-service and user-service and we will make a REST API call using *WebClient* from user-service to department-service to fetch a particular user department.



# Prerequisites

Refer to the below tutorial to create*department-service* and *user-service* microservices.

We have created two microservices in the previous tutorial: [**Spring Boot Microservices Communication Example using RestTemplate**](https://www.javaguides.net/2022/10/spring-boot-microservices-communication-using-resttemplate.html).

# Step 1: Add Spring WebFlux Dependency

Open the *pom.xml* file of the *user-service* project and add the below dependency:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-webflux</artifactId>

</dependency>

# Step 2: Configure WebClient as Spring Bean

package net.javaguides.userservice;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;

import org.springframework.web.reactive.function.client.WebClient;

@SpringBootApplication

public class UserServiceApplication {

public static void main(String[] args) {

SpringApplication.run(UserServiceApplication.class, args);

}

@Bean

public WebClient webClient(){

return WebClient.builder().build();

}

}

We have configured *WebClient* as Spring bean:

@Bean

public WebClient webClient(){

return WebClient.builder().build();

}

# Step 3: Inject and Use WebClient to Call the REST API

Let's inject *WebClient* and use it to make a REST API call:

DepartmentDto departmentDto = webClient.get()

.uri("http://localhost:8080/api/departments/" + user.getDepartmentId())

.retrieve()

.bodyToMono(DepartmentDto.class)

.block();

Here is the complete code of the *UserServiceImpl* class for your reference:

package net.javaguides.userservice.service.impl;

import lombok.AllArgsConstructor;

import net.javaguides.userservice.dto.DepartmentDto;

import net.javaguides.userservice.dto.ResponseDto;

import net.javaguides.userservice.dto.UserDto;

import net.javaguides.userservice.entity.User;

import net.javaguides.userservice.repository.UserRepository;

import net.javaguides.userservice.service.UserService;

import org.springframework.stereotype.Service;

import org.springframework.web.reactive.function.client.WebClient;

@Service

@AllArgsConstructor

public class UserServiceImpl implements UserService {

private UserRepository userRepository;

private RestTemplate restTemplate;

private WebClient webClient;

private APIClient apiClient;

@Override

public User saveUser(User user) {

return userRepository.save(user);

}

@Override

public ResponseDto getUser(Long userId) {

ResponseDto responseDto = new ResponseDto();

User user = userRepository.findById(userId).get();

UserDto userDto = mapToUser(user);

DepartmentDto departmentDto = webClient.get()

.uri("http://localhost:8080/api/departments/" + user.getDepartmentId())

.retrieve()

.bodyToMono(DepartmentDto.class)

.block();

responseDto.setUser(userDto);

responseDto.setDepartment(departmentDto);

return responseDto;

}

private UserDto mapToUser(User user){

UserDto userDto = new UserDto();

userDto.setId(user.getId());

userDto.setFirstName(user.getFirstName());

userDto.setLastName(user.getLastName());

userDto.setEmail(user.getEmail());

return userDto;

}

}

That's it. Now run both the Microservices and let's test.

# Demo: Start Both Microservices

First, start the *department-service* project and then start a *user-service* project.

Once both the projects are up and running on different ports. Next, let's call the **Get User REST API** to test the *user-service* REST API call to the *department-service*.

### Get User REST API:

Note that the response contains a Department for a User.  This demonstrates that we have successfully made a REST API call from *user-service* to *department-service* using *WebClient*.

### Spring Boot + Spring Cloud Open Feign Microservices Communication Example

In the previous couple of tutorials we have seen:

[**Spring Boot Microservices Communication Example using RestTemplate**](https://www.javaguides.net/2022/10/spring-boot-microservices-communication-using-resttemplate.html).

[**Spring Boot Microservices Communication Example using WebClient**](https://www.javaguides.net/2022/10/spring-boot-microservices-communication-using-webclient.html)

In this tutorial, we will learn how to use the *Spring Cloud Open Feign* library to make REST API calls (Synchronous communication) between multiple microservices.

# Spring Cloud Open Feign Overview

Feign makes writing web service clients easier with pluggable annotation support, which includes Feign annotations and JAX-RS annotations. Also, Spring Cloud adds support for Spring MVC annotations and for using the same **HttpMessageConverters** as used in Spring Web.

One great thing about using Feign is that we don't have to write any code for calling the service, other than an interface definition.

**For example:**

package net.javaguides.userservice.service;

import net.javaguides.userservice.dto.DepartmentDto;

import org.springframework.cloud.openfeign.FeignClient;

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

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

@FeignClient(value = "DEPARTMENT-SERVICE", url = "http://localhost:8080")

public interface APIClient {

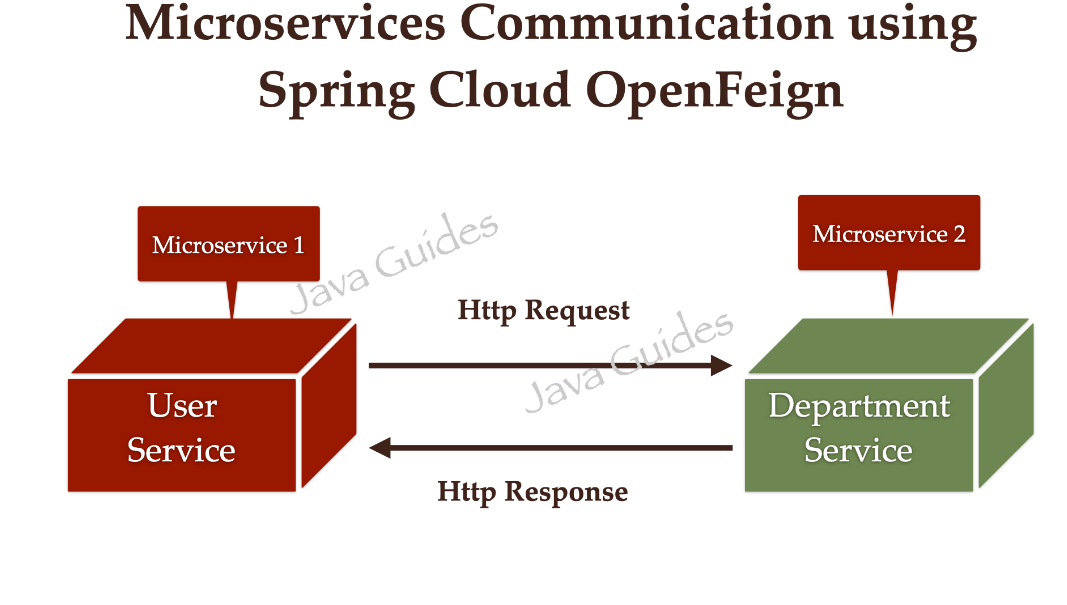
@GetMapping(value = "/api/departments/{id}")

DepartmentDto getDepartmentById(@PathVariable("id") Long departmentId);

}

# What we will Build?

# Well, we will create two microservices such as department-service and user-service and we will make a REST API call using *Spring Cloud Open Feign* from user-service to department-service to fetch a particular user department.



# Prerequisites

Refer to the below tutorial to create department-service and user-service microservices:

[**Spring Boot Microservices Communication Example using RestTemplate**](https://www.javaguides.net/2022/10/spring-boot-microservices-communication-using-resttemplate.html).

# Step 1: Add Spring cloud open feign Maven dependency to User-Service

Open the pom.xml file of the user-service project and add the below dependency:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

Make sure to add spring cloud dependencies and their version.

Here is the complete *pom.xml* file after adding **Spring cloud open feign** dependency:

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.7.4</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>net.javaguides</groupId>

<artifactId>user-service</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>user-service</name>

<description>user-service</description>

<properties>

<java.version>17</java.version>

<spring-cloud.version>2021.0.4</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>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</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>

<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>

# Step 2: Enable Feign Client using @EnableFeignClients

package net.javaguides.userservice;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.openfeign.EnableFeignClients;

@SpringBootApplication

@EnableFeignClients

public class UserServiceApplication {

public static void main(String[] args) {

SpringApplication.run(UserServiceApplication.class, args);

}

}

Note that *@EnableFeignClients* annotation enables component scanning for interfaces that declare they are Feign clients.

# Step 3: Create feign API client

After that, we need to have a feign API client with the necessary methods, requests, and responses.

Let's create an interface named *APIClient* and add the following code:

package net.javaguides.userservice.service;

import net.javaguides.userservice.dto.DepartmentDto;

import org.springframework.cloud.openfeign.FeignClient;

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

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

@FeignClient(value = "DEPARTMENT-SERVICE", url = "http://localhost:8080")

public interface APIClient {

@GetMapping(value = "/api/departments/{id}")

DepartmentDto getDepartmentById(@PathVariable("id") Long departmentId);

}

We declare a Feign client using the *@FeignClient* annotation:

@FeignClient(value = "DEPARTMENT-SERVICE")

The value argument passed in the *@FeignClient* annotation is a mandatory, arbitrary client name, while with the URL argument, we specify the API base URL.

@FeignClient(value = "DEPARTMENT-SERVICE", url = "http://localhost:8080")

Furthermore, since this interface is a Feign client, we can use the Spring Web annotations to declare the APIs that we want to reach out to.

# Step 4: Change the getUser method to call APIClient

First, inject *APIClient* and then use it:

DepartmentDto departmentDto = apiClient.getDepartmentById(user.getDepartmentId());

Here is the complete code of *UserServiceImpl* using Feign client for your reference:

package net.javaguides.userservice.service.impl;

import lombok.AllArgsConstructor;

import net.javaguides.userservice.dto.DepartmentDto;

import net.javaguides.userservice.dto.ResponseDto;

import net.javaguides.userservice.dto.UserDto;

import net.javaguides.userservice.entity.User;

import net.javaguides.userservice.repository.UserRepository;

import net.javaguides.userservice.service.APIClient;

import net.javaguides.userservice.service.UserService;

import org.springframework.stereotype.Service;

@Service

@AllArgsConstructor

public class UserServiceImpl implements UserService {

private UserRepository userRepository;

private APIClient apiClient;

@Override

public User saveUser(User user) {

return userRepository.save(user);

}

@Override

public ResponseDto getUser(Long userId) {

ResponseDto responseDto = new ResponseDto();

User user = userRepository.findById(userId).get();

UserDto userDto = mapToUser(user);

DepartmentDto departmentDto = apiClient.getDepartmentById(user.getDepartmentId());

responseDto.setUser(userDto);

responseDto.setDepartment(departmentDto);

return responseDto;

}

private UserDto mapToUser(User user){

UserDto userDto = new UserDto();

userDto.setId(user.getId());

userDto.setFirstName(user.getFirstName());

userDto.setLastName(user.getLastName());

userDto.setEmail(user.getEmail());

return userDto;

}

}

That's it. Now run both the Microservices and let's test.

# Demo: Start Both Microservices

First, start the department-service project and then start a user-service project.

Once both the projects are up and running on different ports. Next, let's call the **Get User REST API** to test the user-service REST API call to the department-service.

### Get User REST API:

Note that the response contains a Department for a User. This demonstrates that we have successfully made a REST API call from user-service to department-service using WebClient.

### Spring Boot Microservices - Spring Cloud Config Server

# Problem and Solution

## Problem

In the microservices project, there could be a large number of microservices and multiple instances of those microservices are running. Updating configuration properties and restarting all those instances manually or even with automated scripts may not be feasible.

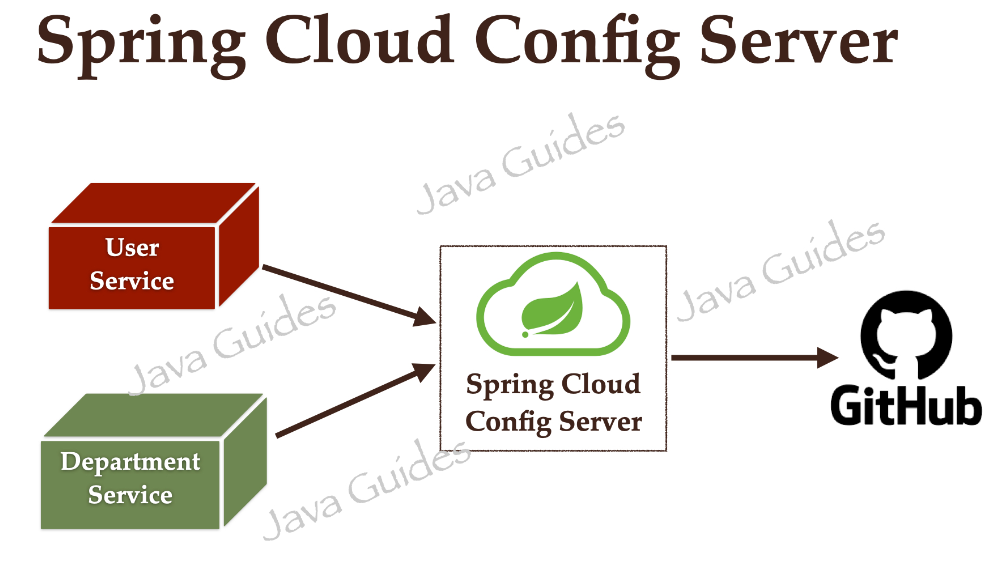
## Solution

Spring Cloud Config addresses this problem.  
  
We can create a Spring Cloud Config Server which provides the configuration values for all of our microservices. We can use git, svn, database, or Consul as a backend to store the configuration parameters.

Next, we can configure the location of the Spring Cloud Config server in our microservice so that it will load all the properties when we start the application. In addition to that, whenever we update the properties we can invoke the *actuator/refresh* the REST endpoint in our microservice so that it will reload the configuration changes without requiring us to restart the application.

# What we will build?

Let's create Spring Cloud Config Server using Git as a backend to store the configurations. Spring Cloud Config Server is nothing but a SpringBoot project.



# Prerequisites

Refer to the below tutorial to create department-service and user-service microservices:

[**Spring Boot Microservices Communication Example using RestTemplate**](https://www.javaguides.net/2022/10/spring-boot-microservices-communication-using-resttemplate.html).

# 1. Create and Setup Spring Boot Project in IntelliJ IDEA

Let's create a Spring boot project using the [**spring initializr**](https://start.spring.io/).

Refer to the below screenshot to enter details while creating the spring boot application using the [**spring initializr**](https://start.spring.io/):

Click on Generate button to download the Spring boot project as a zip file. Unzip the zip file and import the Spring boot project in IntelliJ IDEA.

Here is the pom.xml file for your reference:

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.7.4</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>net.javaguides</groupId>

<artifactId>config-server</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>config-server</name>

<description>Demo project for Spring Boot</description>

<properties>

<java.version>17</java.version>

<spring-cloud.version>2021.0.4</spring-cloud.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</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>

</plugin>

</plugins>

</build>

</project>

# 2. Enable Config Server using @EnableConfigServer Annotation

To make our Spring Boot application as a Spring Cloud Config Server, we just need to add the *@EnableConfigServer* annotation to the main entry point class:

package net.javaguides.configserver;

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

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.config.server.EnableConfigServer;

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

@EnableConfigServer

@SpringBootApplication

public class ConfigServerApplication {

public static void main(String[] args) {

SpringApplication.run(ConfigServerApplication.class, args);

}

}

# 3. Configure the Location of the Git repository

On Github, create a Git repository named "microservices-config-repo".

Now, let's configure the location of the git repository where we are going to store all our configuration files in the *application.properties* file.

spring.application.name=config-server

server.port=8888

spring.cloud.config.server.git.uri=https://github.com/RameshMF/microservices-config-repo.git

spring.cloud.config.server.git.skipSslValidation=true

spring.cloud.config.server.git.clone-on-start=true

management.endpoints.web.exposure.include=\*

That’s it. This is all you need to do to create Spring Cloud Config Server and you just need to add application-specific config files in the git repository.

You can refer to my GitHub repository: [**https://github.com/RameshMF/microservices-config-repo**](https://github.com/RameshMF/microservices-config-repo)

# 4. Refactor department-service to use Config Server

Our *department-service* will become a client for Config Server. So, let us add Config Client starter dependency to the *department-service*:

In *department-service*, add the below dependencies to pom.xml:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

# 5. Push department-service.properties File to the GitHub

Now we need to add all the properties of our *department-service* in *department-service.properties* and commit/push it to our git repo **microservices-config-repo**.

Create a new file *department-service.properties*on the GitHub repository, add the below content and commit it:

spring.datasource.url=jdbc:mysql://localhost:3306/department\_db

spring.datasource.username=root

spring.datasource.password=Mysql@123

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

spring.jpa.hibernate.ddl-auto=update

# 6. Configure Config Server in department-service

Next, change *resources/application.properties* with the config server

spring.application.name=department-service

spring.config.import=optional:configserver:http://localhost:8888

management.endpoints.web.exposure.include=\*

Next, start the Config Server application and then the department-service application. This should work fine. You can check the console logs that department-service is fetching the properties from config server [**http://localhost:8888/**](http://localhost:8888/) at startup.

# 7. Refactor the user-service to use Config Server

Our *user-service* will become a client for Config Server. So, let us add Config Client starter to *user-service* which will add the following dependency.

In user-service, add below dependencies to pom.xml:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

# 8. Push user-service.properties File to the GitHub

Now we need to add all the properties of our *user-service* in *user-service.properties* and commit/push it to our git repo **microservices-config-repo**.

Create a new file *user-service.properties*on the GitHub repository, add the below content, and commit it:

spring.datasource.url=jdbc:mysql://localhost:3306/employee\_db

spring.datasource.username=root

spring.datasource.password=Mysql@123

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

spring.jpa.hibernate.ddl-auto=update

server.port = 8081

# 9. Configure Config Server in user-service

Next, change *resources/application.properties* with the config server

spring.application.name=USER-SERVICE

spring.config.import=optional:configserver:http://localhost:8888

management.endpoints.web.exposure.include=\*

Now first start the Config Server application and then the *user-service* application. This should work fine. You can check the console logs that the *user-service* is fetching the properties from config server [**http://localhost:8888/**](http://localhost:8888/) at startup.

# 10. Refresh Use case

We also want to enable the */refresh* endpoint, to demonstrate dynamic configuration changes.

The client can access any value in the Config Server by using traditional mechanisms (such as @ConfigurationProperties or @Value("${…​}") or through the Environment abstraction). Now you need to create a Spring MVC REST controller that returns the resolved message property’s value.

In department-service, create below REST API:

package net.javaguides.departmentservice.controller;

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

import org.springframework.cloud.context.config.annotation.RefreshScope;

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

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

@RefreshScope

@RestController

class MessageRestController {

@Value("${message:Hello default}")

private String message;

@GetMapping("/message")

String getMessage() {

return this.message;

}

}

By default, the configuration values are read on the client’s startup and not again. You can force a bean to refresh its configuration (that is, to pull updated values from the Config Server) by annotating the MessageRestController with the Spring Cloud Config *@RefreshScope* and then triggering a refresh event.

## Test Refresh Use Case

You can test the end-to-end result by starting the Config Service first and then, once it is running, starting the client. Visit the client app in the browser at [**http://localhost:8080/message**](http://localhost:8080/message). There, you should see Hello world in the response.

Change the message key in the *department-service.properties* file in the Git repository to something different (Hello, Ramesh!).

You need to invoke the */refresh* Spring Boot Actuator endpoint in order to force the client to refresh itself and draw in the new value. Spring Boot’s Actuator exposes operational endpoints (such as health checks and environment information) about an application.

You can invoke the refresh Actuator endpoint by sending an empty HTTP POST to the client’s refresh endpoint: [**http://localhost:8080/actuator/refresh**](http://localhost:8080/actuator/refresh). Then you can confirm it worked by visiting the [**http://localhost:8080/message**](http://localhost:8080/message) endpoint.

# 11. Testing department-service and user-service

First, start the config-server, and next, department-service, and user-service.

If your *department-service* able to connect to the MySQL database then you have successfully configured the Config server.

If your *user-service* able to connect to the MySQL database then you have successfully configured the Config server.

Next, you can test the REST endpoints of *department-service* and *user-service* microservices.

Refer to [**Spring Boot Microservices Communication Example using RestTemplate**](https://www.javaguides.net/2022/10/spring-boot-microservices-communication-using-resttemplate.html) for testing REST endpoints.

# Conclusion

In this tutorial, we learned how to create a Spring cloud config server with Git as a backend to centralize configurations of the Spring boot microservices.

### Spring Boot Microservices - Spring Cloud Netflix Eureka based Service Registry

In this tutorial, we will learn how to create a Service Registry using **Spring Cloud Netflix Eureka** in the Spring boot microservices project.

# Service Registry and Discovery Overview

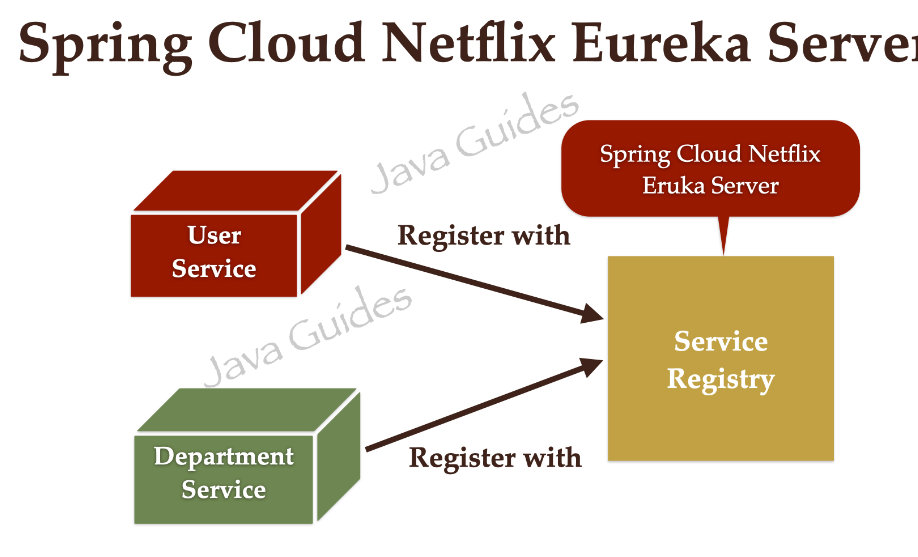
In the microservices projects, Service Registry and Discovery play an important role because we most likely run multiple instances of services and we need a mechanism to call other services without hardcoding their hostnames or port numbers. In addition to that, in Cloud environments service instances may come up and go down anytime. So we need some automatic service registration and discovery mechanism.

**Spring Cloud addresses this problem** by providing **Spring Cloud Netflix Eureka** project to create Service Registry and Discovery.

In this tutorial, we will learn how to use **SpringCloud Netflix Eureka** for Service Registry and Discovery.

# What we will build?

We can use Netflix Eureka Server to create a Service Registry and make our microservices (*department-service* and *user-service*) as Eureka Clients so that as soon as we start a microservice it will get registered with Eureka Server automatically with a logical Service ID. Then, the other microservices, which are also Eureka Clients, can use Service ID to invoke REST endpoints.



Spring Cloud makes it very easy to create a Service Registry and discover other services using Load Balanced *RestTemplate:*

@Bean

@LoadBalanced

public RestTemplate restTemplate(){

return new RestTemplate();

}

# Prerequisites

Refer to the below tutorial to create department-service and user-service microservices:

[**Spring Boot Microservices Communication Example using RestTemplate**](https://www.javaguides.net/2022/10/spring-boot-microservices-communication-using-resttemplate.html).

# 1. Create and Setup Spring boot project in IntelliJ IDEA

Let us create a Service Registry using Netflix Eureka which is nothing but a SpringBoot application with a Eureka Server starter.

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

Let's create a Spring boot project using the [**spring initializr**](https://start.spring.io/).

Refer to the below screenshot to enter details while creating the spring boot application using the [**spring initializr**](https://start.spring.io/):

Click on Generate button to download the Spring boot project as a zip file. Unzip the zip file and import the Spring boot project in IntelliJ IDEA.

Here is the pom.xml file for your reference:

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.7.4</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>net.javaguide</groupId>

<artifactId>service-registry</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>service-registry</name>

<description>service-registry</description>

<properties>

<java.version>17</java.version>

<spring-cloud.version>2021.0.4</spring-cloud.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</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>

</plugin>

</plugins>

</build>

</project>

# 2. Add @EnableEurekaServer annotation

We need to add *@EnableEurekaServer* annotation to make our SpringBoot application a Eureka Server-based Service Registry.

package net.javaguides.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);

}

}

# 3. Disable Eureka Server as Eureka Client

By default, each Eureka Server is also a Eureka client and needs at least one service URL to locate a peer. As we are going to have a single Eureka Server node (Standalone Mode), we are going to disable this client-side behavior by configuring the following properties in the *application.properties* file.

spring.application.name=service-registry

server.port=8761

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

# 4. Launch Eureka Server (Demo)

Netflix Eureka Service provides UI where we can see all the details about registered services.

Now run *ServiceRegistryApplication* and access [**http://localhost:8761**](http://localhost:8761/) which will display the UI similar to the below screenshot.

# 5. Registering Department-Service Microservice as Eureka Client

Refer to this tutorial to create department-service and user-service microservices: [**Spring Boot Microservices Communication Example using RestTemplate**](https://www.javaguides.net/2022/10/spring-boot-microservices-communication-using-resttemplate.html).

Let us make this *department-service* as a Eureka Client and register with the Eureka Server.

Add the Eureka Discovery starter to *department-service*:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

Also, add the Spring cloud 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>

Add version as property:

<properties>

<java.version>17</java.version>

<spring-cloud.version>2021.0.4</spring-cloud.version>

</properties

With *spring-cloud-starter-netflix-eureka-client* on the classpath, we just need to configure *eureka.client.service-url.defaultZone* property in *application.properties* to automatically register with the Eureka Server.

spring.application.name=DEPARTMENT-SERVICE

eureka.instance.client.serviceUrl.defaultZone=http://localhost:8761/eureka/

When a service is registered with Eureka Server it keeps sending heartbeats for certain intervals. If the Eureka server didn’t receive a heartbeat from any service instance it will assume the service instance is down and take it out from the pool.

# 6. Run department-service Eureka Client (Demo)

With this configuration in place, start *department-service* and visit [**http://localhost:8761**](http://localhost:8761/).

You should see that *department-service* is registered with SERVICE ID as DEPARTMENT-SERVICE. You can also notice the status as UP(1) which means the services are up and running and one instance of *department-service* is running.

# 7. Registering User-Service Microservice as Eureka Client

Refer to this tutorial to create department-service and user-service microservices: [**Spring Boot Microservices Communication Example using RestTemplate**](https://www.javaguides.net/2022/10/spring-boot-microservices-communication-using-resttemplate.html).

Let us make this *user-service* as a Eureka Client and register with the Eureka Server.

Add the Eureka Discovery starter to *user-service*:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

Also, add the Spring cloud 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>

Add version as property:

<properties>

<java.version>17</java.version>

<spring-cloud.version>2021.0.4</spring-cloud.version>

</properties

With *spring-cloud-starter-netflix-eureka-client* on the classpath, we just need to configure *eureka.client.service-url.defaultZone* property in *application.properties* to automatically register with the Eureka Server.

spring.application.name=USER-SERVICE

eureka.instance.client.serviceUrl.defaultZone=http://localhost:8761/eureka/

# 8. Run user-service Eureka Client (Demo)

With this configuration in place, start user-service and visit [**http://localhost:8761**](http://localhost:8761/). You should see *user-service* is registered with SERVICE ID as USER-SERVICE.

You can also notice the status as UP(1) which means the services are up and running and one instance of *user-service* is running.

# Multiple Instances of Department-Service

Let us start another instance of *department-service* on a different port using the following command.

java -jar -Dserver.port=8082 department-service-0.0.1-SNAPSHOT.jar

Suppose we want to invoke the *department-service* REST endpoint from the *user-service*. We can use *RestTemplate* to invoke the REST endpoint but there are 2 instances running.

We can register RestTemplate as a Spring bean with *@LoadBalanced* annotation:

@Bean

@LoadBalanced

public RestTemplate restTemplate(){

return new RestTemplate();

}

The *RestTemplate* with *@LoadBalanced* annotation will internally use **Ribbon LoadBalancer** to resolve the ServiceID and invoke the REST endpoint using one of the available servers.

Change the URL in *UserServiceImpl* class:

ResponseEntity<DepartmentDto> responseEntity = restTemplate

.getForEntity("http://DEPARTMENT-SERVICE/api/departments/" + user.getDepartmentId(),

DepartmentDto.class);

With this kind of automatic Service Registration and Discovery mechanism, we no need to worry about how many instances are running and what are their hostnames and ports, etc

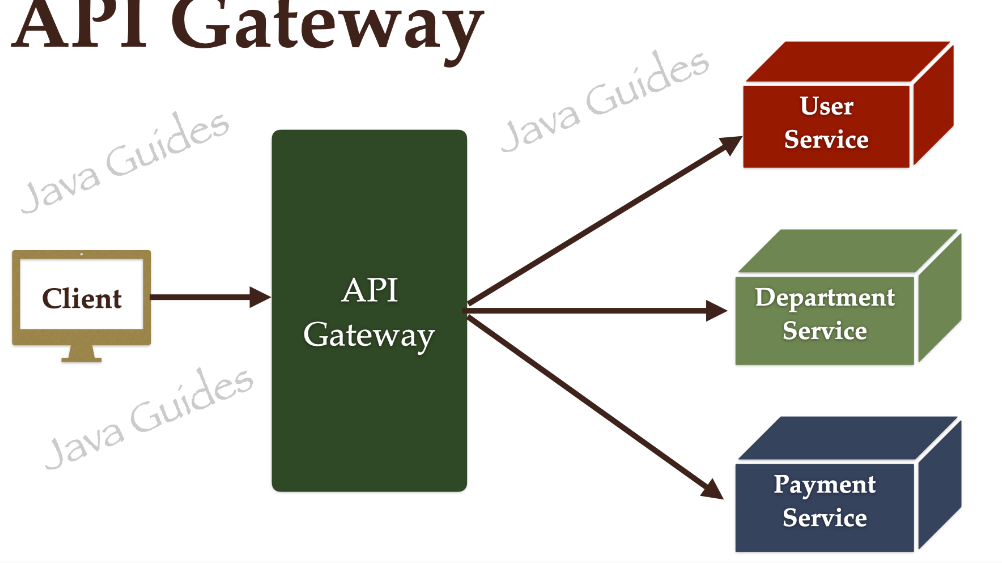
# Conclusion

In this tutorial, we learned how to create a Service Registry using **Spring Cloud Netflix Eureka** in the Spring boot microservices project.

### Spring Boot Microservices - Spring Cloud API Gateway

# Spring Cloud Gateway Overview

Spring Cloud Gateway provides a library for building an API Gateway on top of Spring WebFlux. Spring Cloud Gateway aims to provide a simple, yet effective way to route to APIs and provide cross-cutting concerns to them such as security, monitoring/metrics, and resiliency.



**The Spring Cloud Gateway has three important parts to it:**

**Route** − These are the building blocks of the gateway which contain the URL to which the request is to be forwarded to and the predicates and filters that are applied to the incoming requests.

**Predicate** − These are the set of criteria that should match for the incoming requests to be forwarded to internal microservices. For example, a path predicate will forward the request only if the incoming URL contains that path.

**Filters** − These act as the place where you can modify the incoming requests before sending the requests to the internal microservices or before responding back to the client.

To know more read [**Spring Cloud Gateway documentation**](https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/).

# Prerequisites

Refer to the below tutorial to create department-service and user-service microservices and configure Netflix Eureka Service Registry:

[**Spring Boot Microservices - Spring Cloud Netflix Eureka-based Service Registry**](https://www.javaguides.net/2022/10/spring-boot-microservices-spring-cloud-netflix-eureka-.html)

# 1. Create and Setup Spring Boot Project in IntelliJ IDEA

Let's create a Spring boot project using the [**spring initializr**](https://start.spring.io/).

Refer to the below screenshot to enter details while creating the spring boot application using the [**spring initializr**](https://start.spring.io/):

Click on Generate button to download the Spring boot project as a zip file. Unzip the zip file and import the Spring boot project in IntelliJ IDEA.

Here is the pom.xml file for your reference:

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.7.4</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>nt.javaguides</groupId>

<artifactId>api-gateway</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>api-gateway</name>

<description>api-gateway</description>

<properties>

<java.version>17</java.version>

<spring-cloud.version>2021.0.4</spring-cloud.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-gateway</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</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>

</plugin>

</plugins>

</build>

</project>

Now we have all the dependencies that we need to have in our API gateway application. So now let’s configure the Routes and other API gateway-specific configurations to use in our project.

# 2. Enable Eureka Client using @EnableEurekaClient

The *@EnableEurekaClient* annotation makes your Spring Boot application act as a Eureka client.

package nt.javaguides.apigateway;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

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

@SpringBootApplication

@EnableEurekaClient

public class ApiGatewayApplication {

public static void main(String[] args) {

SpringApplication.run(ApiGatewayApplication.class, args);

}

}

Don't miss this step, you have to configure the API gateway as a Eureka Client for this project. Otherwise, you couldn’t use discovery functions to identify the correct API from the service registry.

# 3. Configure Eureka Server URL

To register the Spring Boot application into Eureka Server we need to add the following configuration in our *application.properties* file and specify the Eureka Server URL in our configuration.

spring.application.name=API-GATEWAY

server.port=9191

eureka.instance.client.serviceUrl.defaultZone=http://localhost:8761/eureka/

management.endpoints.web.exposure.include=\*

# 4. Configuring API Gateway Routes With Spring Cloud Gateway

Now, you might be wondering how API Gateway knows the hostname or IP and port of microservices right.

When a client sends a request to the API gateway, It will discover the correct service IP and PORT using the service registry to communicate and route the request.

Let's configure Routes using properties:

spring.application.name=API-GATEWAY

server.port=9191

eureka.instance.client.serviceUrl.defaultZone=http://localhost:8761/eureka/

management.endpoints.web.exposure.include=\*

spring.cloud.gateway.routes[0].id=USER-SERVICE

spring.cloud.gateway.routes[0].uri=lb://USER-SERVICE

spring.cloud.gateway.routes[0].predicates[0]=Path=/api/users/\*\*

spring.cloud.gateway.routes[1].id=DEPARTMENT-SERVICE

spring.cloud.gateway.routes[1].uri=lb://DEPARTMENT-SERVICE

spring.cloud.gateway.routes[1].predicates[0]=Path=/api/departments/\*\*

spring.cloud.gateway.routes[2].id=DEPARTMENT-SERVICE

spring.cloud.gateway.routes[2].uri=lb://DEPARTMENT-SERVICE

spring.cloud.gateway.routes[2].predicates[0]=Path=/message/\*\*

What are the properties that we set for API gateway routes?

* id – This is just an identification of the routes.
* URI – Here we can use either URL [**http://localhost:8080**](http://localhost:8080/) or lb://DEPARTMENT-SERVICE. But if we need to use the inbuilt load balancer on the Netflix Eureka server, we should use lb://DEPARTMENT-SERVICE, then the API registry will take over the request and show a load-balanced request destination to the API gateway.
* predicates – In here we can set multiple paths to identify a correct routing destination. Eg:- If the API gateway gets and request like [**http://localhost:9191/api/users/1**](http://localhost:9191/api/users/1) then it will be routed into [**http://localhost:8081/api/users/1**](http://localhost:8081/api/users/1).

# 5. Run All the Microservices

Department Service running on port: [**http://localhost:8080**](http://localhost:8080/)

User Service running on port: [**http://localhost:8081**](http://localhost:8081/)

API Gateway service running on port: [**http://localhost:9191**](http://localhost:9191/)

Service Registry service running on port: [**http://localhost:8761**](http://localhost:8761/)

# 6. Verify Registered Instances in Service Registry

Go to the browser and hit this link in a new tab: [**http://localhost:8761**](http://localhost:8761/)

# 7. Testing API Gateway using Postman Client

## Get Department REST API:

Note that we are using API-Gateway service port (9191) to call department-service API (port 8080)

Department Service running on port: [**http://localhost:8080**](http://localhost:8080/)

API Gateway service running on port: [**http://localhost:9191**](http://localhost:9191/)

API Gateway route the request from [**http://localhost:9191/api/departments/1**](http://localhost:9191/api/departments/1) to [**http://localhost:8080/api/departments/1**](http://localhost:8080/api/departments/1)

## Get User REST API:

Note that we are using API-Gateway service port (9191) to call the user-service API port (8081).

User Service running on port: [**http://localhost:8081**](http://localhost:8081/)

API Gateway service running on port: [**http://localhost:9191**](http://localhost:9191/)

API Gateway route the request from [**http://localhost:9191/api/users/1**](http://localhost:9191/api/users/1) to [**http://localhost:8081/api/users/1**](http://localhost:8081/api/users/1)

# 8. Conclusion

In this tutorial, we learned how to set up an API gateway into our microservices project using the Spring Cloud Gateway library.

### Microservices Project using Spring Boot

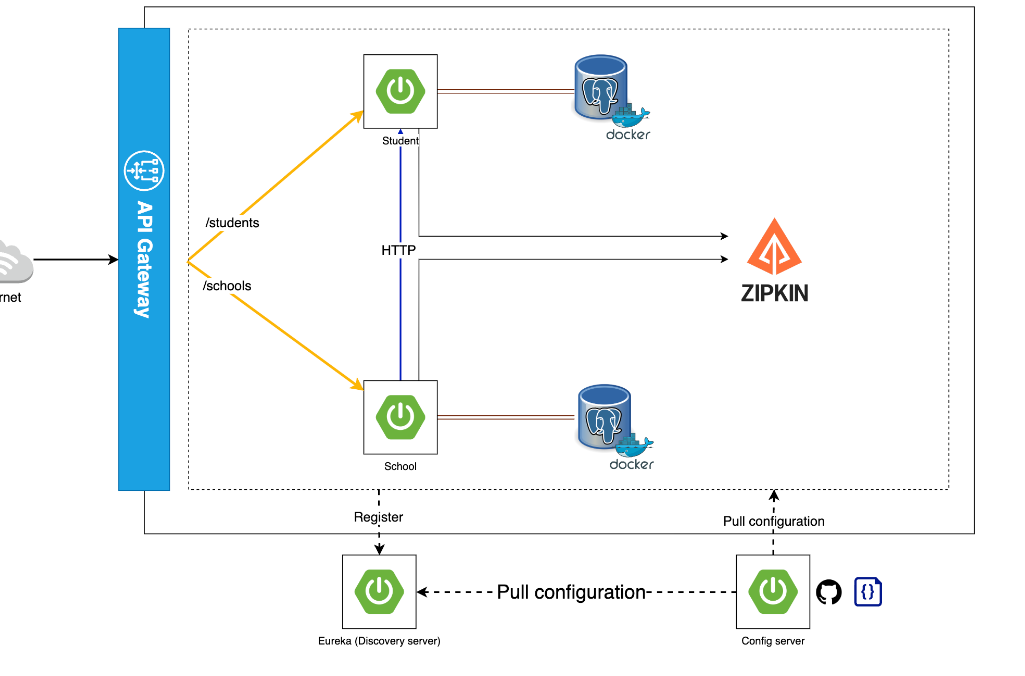
**Prerequisites**

Ensure you have the following software installed on your system before proceeding:

* Java Development Kit (JDK) 17 or later
* Maven
* Docker (optional, for containerization)

**Microservices Architecture**

Here is the Microservices Architecture for our Microservices project that we are going to build in this tutorial:



Let's understand the Microservices project components:

### API Gateway

The API Gateway serves as the single entry point for all client requests, managing and routing them to the appropriate microservices.

### Config Server

The Config Server centralizes configuration management for all microservices, simplifying application maintenance and consistency across environments.

### Discovery Server

The Discovery Server provides service registration and discovery, enabling seamless service-to-service communication within the microservices ecosystem.

### Student Microservice

The Student Microservice is responsible for managing student-related data and operations, such as adding, updating, and retrieving student records.

### School Microservice

The School Microservice manages school-related data and operations, including adding, updating, and retrieving school records.

### Using OpenFeign - Inter-Service Communication

This project demonstrates inter-service communication using OpenFeign, a declarative REST client that simplifies service-to-service communication within the microservices ecosystem.

### Distributed Tracing - Using Zipkin

The project showcases the use of Zipkin for distributed tracing, enhancing application observability and enabling the visualization and troubleshooting of latency issues.

# 1. Create Student Microservice

Let's build student microservice step by step.

## Create Spring Boot Project

Let's create a**student** microservice as a Spring boot project using the [**spring initializr**](https://start.spring.io/).

Refer to the below screenshot to enter details while creating the spring boot application using the [**spring initializr**](https://start.spring.io/):

Here is the complete *pom.xml* file for your reference:

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.0.6</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>net.javaguides</groupId>

<artifactId>student</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>student</name>

<description>Demo project for Spring Boot</description>

<properties>

<java.version>17</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<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>org.postgresql</groupId>

<artifactId>postgresql</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>

## Configure PostgreSQL Database

Create an *application.yml* file under */resources* folder and add the following content:

server:

port: 8090

spring:

application:

name: students

datasource:

driver-class-name: org.postgresql.Driver

url: jdbc:postgresql://localhost:5432/students

username: username

password: password

jpa:

hibernate:

ddl-auto: create

database: postgresql

database-platform: org.hibernate.dialect.PostgreSQLDialect

Don’t forget to change the *spring.datasource.username* and *spring.datasource.password* as per your PostgreSQL installation. Also, create a database named students in PostgreSQL before proceeding to the next section.

You don’t need to create any tables. The tables will automatically be created by Hibernate from the *Student* entity that we will define in the next step. This is made possible by the property *spring.jpa.hibernate.ddl-auto = create.*

## JPA Entity - Student

Let's create a *Student* JPA entity and add the following code to it:

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.Id;

import lombok.\*;

@Entity

@Getter

@Setter

@AllArgsConstructor

@NoArgsConstructor

@Builder

public class Student {

@Id

@GeneratedValue

private Integer id;

private String firstname;

private String lastname;

private String email;

private Integer schoolId;

}

## Repository Layer - StudentRepository

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

import java.util.List;

public interface StudentRepository extends JpaRepository<Student, Integer> {

List<Student> findAllBySchoolId(Integer schoolId);

}

## Service Layer - StudentService

import lombok.RequiredArgsConstructor;

import org.springframework.stereotype.Service;

import java.util.List;

@Service

@RequiredArgsConstructor

public class StudentService {

private final StudentRepository repository;

public void saveStudent(Student student) {

repository.save(student);

}

public List<Student> findAllStudents() {

return repository.findAll();

}

public List<Student> findAllStudentsBySchool(Integer schoolId) {

return repository.findAllBySchoolId(schoolId);

}

}

## Controller Layer - StudentController

import lombok.RequiredArgsConstructor;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

@RestController

@RequestMapping("/api/v1/students")

@RequiredArgsConstructor

public class StudentController {

private final StudentService service;

@PostMapping

@ResponseStatus(HttpStatus.CREATED)

public void save(

@RequestBody Student student

) {

service.saveStudent(student);

}

@GetMapping

public ResponseEntity<List<Student>> findAllStudents() {

return ResponseEntity.ok(service.findAllStudents());

}

@GetMapping("/school/{school-id}")

public ResponseEntity<List<Student>> findAllStudents(

@PathVariable("school-id") Integer schoolId

) {

return ResponseEntity.ok(service.findAllStudentsBySchool(schoolId));

}

}

That's completed our student microservice development.

# 2. Create School Microservice

Let's build a school microservice step by step.

## Create Spring Boot Project

Let's create a**school**microservice as a Spring boot project using the [**spring initializr**](https://start.spring.io/).

Refer to the below screenshot to enter details while creating the spring boot application using the [**spring initializr**](https://start.spring.io/):

Here is the complete *pom.xml* file for your reference:

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.0.6</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>net.javaguides</groupId>

<artifactId>school</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>school</name>

<description>Demo project for Spring Boot</description>

<properties>

<java.version>17</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<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>org.postgresql</groupId>

<artifactId>postgresql</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>

## Configure PostgreSQL Database

Create an *application.yml* file under */resources* folder and add the following content:

server:

port: 8090

spring:

application:

name: schools

datasource:

driver-class-name: org.postgresql.Driver

url: jdbc:postgresql://localhost:5432/students

username: username

password: password

jpa:

hibernate:

ddl-auto: create

database: postgresql

database-platform: org.hibernate.dialect.PostgreSQLDialect

Don’t forget to change the *spring.datasource.username* and *spring.datasource.password* as per your PostgreSQL installation. Also, create a database named **schools** in PostgreSQL before proceeding to the next section.

You don’t need to create any tables. The tables will automatically be created by Hibernate from the *School* entity that we will define in the next step. This is made possible by the property *spring.jpa.hibernate.ddl-auto = create.*

## Domain Layer - School JPA Entity

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.Id;

import lombok.\*;

@Entity

@Getter

@Setter

@AllArgsConstructor

@NoArgsConstructor

@Builder

public class School {

@Id

@GeneratedValue

private Integer id;

private String name;

private String email;

}

## Repository Layer - SchoolRepository

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

public interface SchoolRepository extends JpaRepository<School, Integer> {

}

## Create a DTO Class

import lombok.\*;

import java.util.List;

@Getter

@Setter

@AllArgsConstructor

@NoArgsConstructor

@Builder

public class FullSchoolResponse {

private String name;

private String email;

List<Student> students;

}

## Service Layer - SchoolService

import lombok.RequiredArgsConstructor;

import org.springframework.stereotype.Service;

import java.util.List;

@Service

@RequiredArgsConstructor

public class SchoolService {

private final SchoolRepository repository;

private final StudentClient client;

public void saveSchool(School school) {

repository.save(school);

}

public List<School> findAllSchools() {

return repository.findAll();

}

}

## Controller Layer - SchoolController

import lombok.RequiredArgsConstructor;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

@RestController

@RequestMapping("/api/v1/schools")

@RequiredArgsConstructor

public class SchoolController {

private final SchoolService service;

@PostMapping

@ResponseStatus(HttpStatus.CREATED)

public void save(

@RequestBody School school

) {

service.saveSchool(school);

}

@GetMapping

public ResponseEntity<List<School>> findAllSchools() {

return ResponseEntity.ok(service.findAllSchools());

}

}

That's completed our school microservice development.

# 3. Using OpenFeign - Inter-Service Communication

In this section, we will see the inter-service communication using OpenFeign, a declarative REST client that simplifies service-to-service communication within the microservices ecosystem.

We will see how to use the OpenFeign library to make a REST API call from school service to student service.

## Add OpenFeign Dependency

In order to use the OpenFeign library, open the pom.xml file of the school service project and add the below dependency:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

Make sure to add spring cloud dependencies and their version.

Here is the complete pom.xml file after adding **Spring cloud open feign** dependency:

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.0.6</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>net.javaguides</groupId>

<artifactId>school</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>school</name>

<description>Demo project for Spring Boot</description>

<properties>

<java.version>17</java.version>

<spring-cloud.version>2022.0.2</spring-cloud.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<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>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

<dependency>

<groupId>org.postgresql</groupId>

<artifactId>postgresql</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>

<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>

## Enable OpenFeign Client in School Service

Next, let's enable OpenFeign Client using *@EnableFeignClients* annotation:

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.openfeign.EnableFeignClients;

@EnableFeignClients

@SpringBootApplication

public class SchoolApplication {

public static void main(String[] args) {

SpringApplication.run(SchoolApplication.class, args);

}

}

Note that *@EnableFeignClients* annotation enables component scanning for interfaces that declare they are Feign clients.

## Create feign API client

After that, we need to have a feign API client with the necessary methods, requests, and responses.  
  
Let's create an interface named *StudentClient* and add the following code:

import net,javaguides.school.Student;

import org.springframework.cloud.openfeign.FeignClient;

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

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

import java.util.List;

@FeignClient(name = "student-service", url = "${application.config.students-url}")

public interface StudentClient {

@GetMapping("/school/{school-id}")

List<Student> findAllStudentsBySchool(@PathVariable("school-id") Integer schoolId);

}

## Change SchoolService Class

Let's create a method *findSchoolsWithStudents()* to retrieve students by school id:

public FullSchoolResponse findSchoolsWithStudents(Integer schoolId) {

var school = repository.findById(schoolId)

.orElse(

School.builder()

.name("NOT\_FOUND")

.email("NOT\_FOUND")

.build()

);

var students = client.findAllStudentsBySchool(schoolId);

return FullSchoolResponse.builder()

.name(school.getName())

.email(school.getEmail())

.students(students)

.build();

}

Note that we are using the OpenFeign client to make a REST API call:

var students = client.findAllStudentsBySchool(schoolId);

Here is the complete *StudentService* class code:

import lombok.RequiredArgsConstructor;

import org.springframework.stereotype.Service;

import java.util.List;

@Service

@RequiredArgsConstructor

public class SchoolService {

private final SchoolRepository repository;

private final StudentClient client;

public void saveSchool(School school) {

repository.save(school);

}

public List<School> findAllSchools() {

return repository.findAll();

}

public FullSchoolResponse findSchoolsWithStudents(Integer schoolId) {

var school = repository.findById(schoolId)

.orElse(

School.builder()

.name("NOT\_FOUND")

.email("NOT\_FOUND")

.build()

);

var students = client.findAllStudentsBySchool(schoolId);

return FullSchoolResponse.builder()

.name(school.getName())

.email(school.getEmail())

.students(students)

.build();

}

}

## Create REST API - Retrieve Students by School Id

@GetMapping("/with-students/{school-id}")

public ResponseEntity<FullSchoolResponse> findAllStudentsBySchoolId(

@PathVariable("school-id") Integer schoolId

) {

return ResponseEntity.ok(service.findSchoolsWithStudents(schoolId));

}

Here is the complete *SchoolController* class code:

import lombok.RequiredArgsConstructor;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

@RestController

@RequestMapping("/api/v1/schools")

@RequiredArgsConstructor

public class SchoolController {

private final SchoolService service;

@PostMapping

@ResponseStatus(HttpStatus.CREATED)

public void save(

@RequestBody School school

) {

service.saveSchool(school);

}

@GetMapping

public ResponseEntity<List<School>> findAllSchools() {

return ResponseEntity.ok(service.findAllSchools());

}

@GetMapping("/with-students/{school-id}")

public ResponseEntity<FullSchoolResponse> findAllStudentsBySchoolId(

@PathVariable("school-id") Integer schoolId

) {

return ResponseEntity.ok(service.findSchoolsWithStudents(schoolId));

}

}

# 4. Discovery

# In this section, we will learn how to use SpringCloud Netflix Eureka for Service Registry and Discovery.

## Create Spring Boot Project

Let's create a Spring boot project using the [**spring initializr**](https://start.spring.io/).

Refer to the below screenshot to enter details while creating the spring boot application using the [**spring initializr**](https://start.spring.io/):

Here is the complete *pom.xml* file for your reference:

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.0.6</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>net.javaguides</groupId>

<artifactId>discovery</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>discovery</name>

<description>Demo project for Spring Boot</description>

<properties>

<java.version>17</java.version>

<spring-cloud.version>2022.0.2</spring-cloud.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</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>

</plugin>

</plugins>

</build>

<repositories>

<repository>

<id>netflix-candidates</id>

<name>Netflix Candidates</name>

<url>https://artifactory-oss.prod.netflix.net/artifactory/maven-oss-candidates</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

</project>

## Add @EnableEurekaServer annotation

We need to add *@EnableEurekaServer* annotation to make our SpringBoot application a Eureka Server-based Service Registry.

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

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

@EnableEurekaServer

@SpringBootApplication

public class DiscoveryApplication {

public static void main(String[] args) {

SpringApplication.run(DiscoveryApplication.class, args);

}

}

## Configure the Eureka Server in an application.yml File

Create an *application.yml* file under /resources folder and add the following code:

eureka:

instance:

hostname: localhost

client:

register-with-eureka: false

fetch-registry: false

service-url:

defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

server:

port: 8761

## Registering Student Microservice as Eureka Client

Add the Eureka Discovery starter to student service:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

With spring-cloud-starter-netflix-eureka-client on the classpath, we just need to configure eureka.client.service-url.defaultZone property in application.yml to automatically register with the Eureka Server.

eureka:

instance:

hostname: localhost

client:

service-url:

defaultZone: http://localhost:8761/eureka

When a service is registered with Eureka Server it keeps sending heartbeats for certain intervals. If the Eureka server didn’t receive a heartbeat from any service instance it will assume the service instance is down and take it out of the pool.

## Registering School Microservice as Eureka Client

Add the Eureka Discovery starter to school service:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

With spring-cloud-starter-netflix-eureka-client on the classpath, we just need to configure eureka.client.service-url.defaultZone property in application.yml to automatically register with the Eureka Server.

eureka:

instance:

hostname: localhost

client:

service-url:

defaultZone: http://localhost:8761/eureka

When a service is registered with Eureka Server it keeps sending heartbeats for certain intervals. If the Eureka server didn’t receive a heartbeat from any service instance it will assume the service instance is down and take it out of the pool.

## Run Discovery Service

With this configuration in place, start the discovery service and visit [**http://localhost:8761**](http://localhost:8761/):

# 5. API Gateway

In this section, we will learn how to set up an API gateway into our microservices project using the Spring Cloud Gateway library.

Spring Cloud Gateway provides a library for building an API Gateway on top of Spring WebFlux. Spring Cloud Gateway aims to provide a simple, yet effective way to route to APIs and provide cross-cutting concerns to them such as security, monitoring/metrics, and resiliency.

## Create Spring Boot Project

Let's create a Spring boot project using the [**spring initializr**](https://start.spring.io/).

Refer to the below screenshot to enter details while creating the spring boot application using the [**spring initializr**](https://start.spring.io/):

Here is the complete pom.xml file for your reference:

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.0.6</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>net.javaguides</groupId>

<artifactId>gateway</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>gateway</name>

<description>Demo project for Spring Boot</description>

<properties>

<java.version>17</java.version>

<spring-cloud.version>2022.0.2</spring-cloud.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-gateway</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</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>

</plugin>

</plugins>

</build>

<repositories>

<repository>

<id>netflix-candidates</id>

<name>Netflix Candidates</name>

<url>https://artifactory-oss.prod.netflix.net/artifactory/maven-oss-candidates</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

</project>

## Configuring API Gateway Routes With Spring Cloud Gateway

eureka:

client:

register-with-eureka: false

server:

port: 8222

spring:

application:

name: gateway

cloud:

gateway:

discovery:

locator:

enabled: true

routes:

- id: students

uri: http://localhost:8090

predicates:

- Path=/api/v1/students/\*\*

- id: schools

uri: http://localhost:8070

predicates:

- Path=/api/v1/schools/\*\*

management:

tracing:

sampling:

probability: 1.0

# 6. Config Server

In this section, we will learn how to create a Spring cloud config server to centralize configurations of the Spring boot microservices.

Well, we will keep the configuration files of all the microservices in a local central place.

## Create Spring Boot Project

Let's create a Spring boot project using the [**spring initializr**](https://start.spring.io/).

Refer to the below screenshot to enter details while creating the spring boot application using the [**spring initializr**](https://start.spring.io/):

Here is the pom.xml file for your reference:

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.0.6</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>net.javaguides</groupId>

<artifactId>config-server</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>config-server</name>

<description>Demo project for Spring Boot</description>

<properties>

<java.version>17</java.version>

<spring-cloud.version>2022.0.2</spring-cloud.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</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>

</plugin>

</plugins>

</build>

</project>

## Enable Config Server using @EnableConfigServer Annotation

To make our Spring Boot application a Spring Cloud Config Server, we just need to add the *@EnableConfigServer* annotation to the main entry point class:

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.config.server.EnableConfigServer;

@EnableConfigServer

@SpringBootApplication

public class ConfigServerApplication {

public static void main(String[] args) {

SpringApplication.run(ConfigServerApplication.class, args);

}

}

## Configure the Location for Centralized Configuration

Now, let's configure the location of the local repository where we are going to store all our configuration files in the *application.properties* file:

server:

port: 8888

spring:

profiles:

active: native

application:

name: config-server

cloud:

config:

server:

native:

search-locations: classpath:/configurations

Under */resources/configurations* folder and keep all the configuration files:

## students.yml

eureka:

instance:

hostname: localhost

client:

service-url:

defaultZone: http://localhost:8761/eureka

server:

port: 8090

spring:

application:

name: students

datasource:

driver-class-name: org.postgresql.Driver

url: jdbc:postgresql://localhost:5432/students

username: username

password: password

jpa:

hibernate:

ddl-auto: create

database: postgresql

database-platform: org.hibernate.dialect.PostgreSQLDialect

management:

tracing:

sampling:

probability: 1.0

## School.yml

eureka:

instance:

hostname: localhost

client:

service-url:

defaultZone: http://localhost:8761/eureka

server:

port: 8070

spring:

application:

name: schools

datasource:

driver-class-name: org.postgresql.Driver

url: jdbc:postgresql://localhost:5432/schools

username: username

password: password

jpa:

hibernate:

ddl-auto: create

database: postgresql

database-platform: org.hibernate.dialect.PostgreSQLDialect

application:

config:

students-url: http://localhost:8222/api/v1/students

management:

tracing:

sampling:

probability: 1.0

## discovery.yml

eureka:

instance:

hostname: localhost

client:

register-with-eureka: false

fetch-registry: false

service-url:

defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

server:

port: 8761

## gateway.yml

eureka:

client:

register-with-eureka: false

server:

port: 8222

spring:

application:

name: gateway

cloud:

gateway:

discovery:

locator:

enabled: true

routes:

- id: students

uri: http://localhost:8090

predicates:

- Path=/api/v1/students/\*\*

- id: schools

uri: http://localhost:8070

predicates:

- Path=/api/v1/schools/\*\*

management:

tracing:

sampling:

probability: 1.0

## Refactor the Student Service to use Config Server

Add the below dependency to the student service pom.xml file to use the config server:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

## Configure config server URL in student service

spring:

application:

name: students

config:

import: optional:configserver:http://localhost:8888

## Refactor the School Service to use Config Server

Add the below dependency to the **school** service pom.xml file to use the config server:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

## Configure config server URL in School service

spring:

application:

name: students

config:

import: optional:configserver:http://localhost:8888

## Refactor the Discovery Service to use Config Server

Add the below dependency to the **discovery** service pom.xml file to use the config server:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

## Configure config server URL in Discovery service

spring:

application:

name: students

config:

import: optional:configserver:http://localhost:8888

## Refactor the Gateway Service to use Config Server

Add the below dependency to the **gateway** service pom.xml file to use the config server:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

## Configure config server URL in Gateway service

spring:

application:

name: students

config:

import: optional:configserver:http://localhost:8888

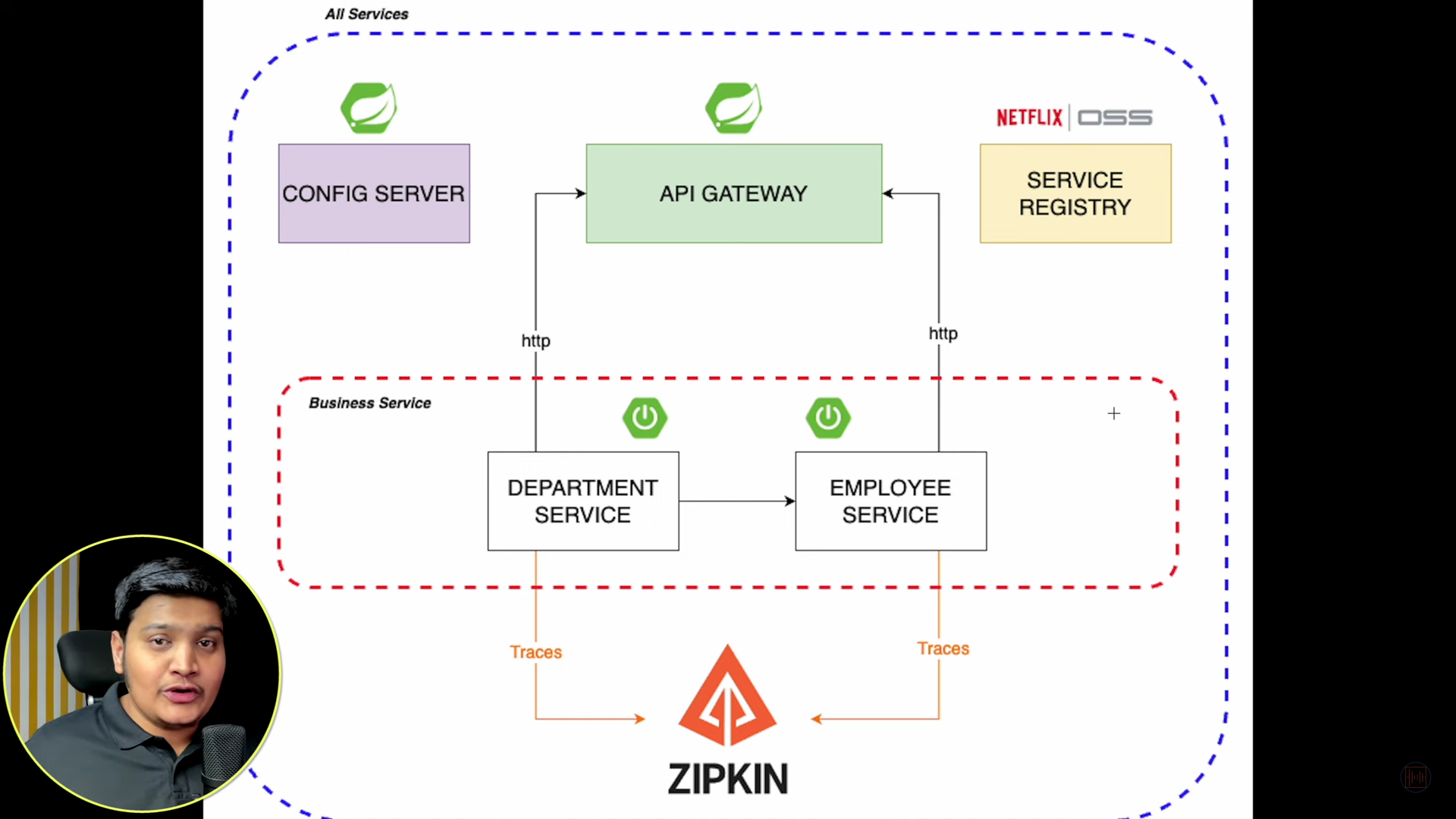
# Demo

Let's insert a few students for a school and call get school by students:

# Conclusion

In this tutorial, we learned how to build a step-by-step microservice project using Spring Boot 3, Spring Cloud, and the PostgreSQL database.

<https://www.javaguides.net/p/spring-boot-microservices-tutorial.html>



Let's understand the Microservices project components:

### API Gateway

The API Gateway serves as the single entry point for all client requests, managing and routing them to the appropriate microservices.

### Config Server

The Config Server centralizes configuration management for all microservices, simplifying application maintenance and consistency across environments.

### Discovery Server

The Discovery Server provides service registration and discovery, enabling seamless service-to-service communication within the microservices ecosystem.

### Using OpenFeign - Inter-Service Communication

This project demonstrates inter-service communication using OpenFeign, a declarative REST client that simplifies service-to-service communication within the microservices ecosystem.

### Distributed Tracing - Using Zipkin

The project showcases the use of Zipkin for distributed tracing, enhancing application observability and enabling the visualization and troubleshooting of latency issues.