✅ [**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" \t "_blank)**

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

**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:

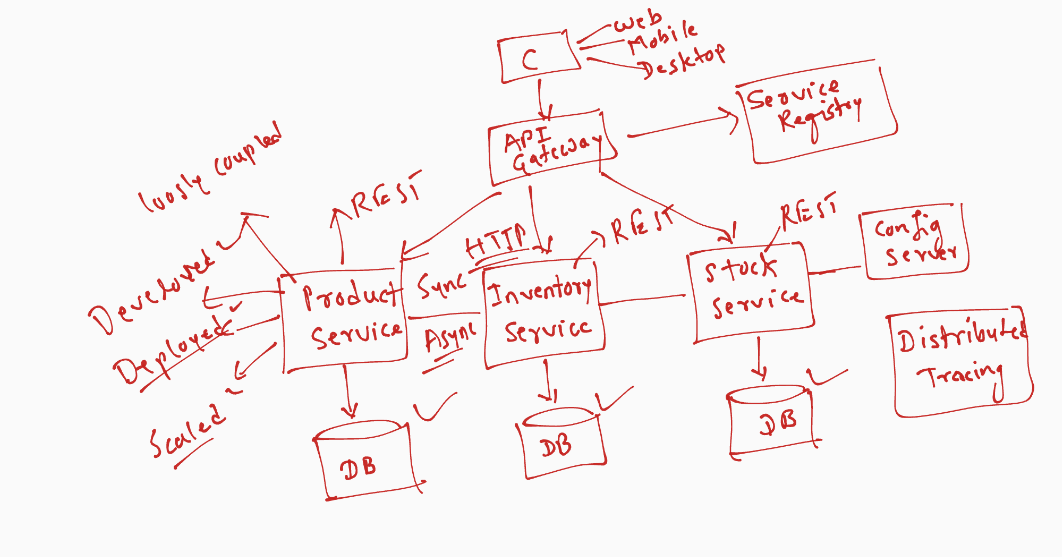
In this tutorial, we will discuss how to create a typical microservices architecture using Spring boot and Spring cloud.

**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.

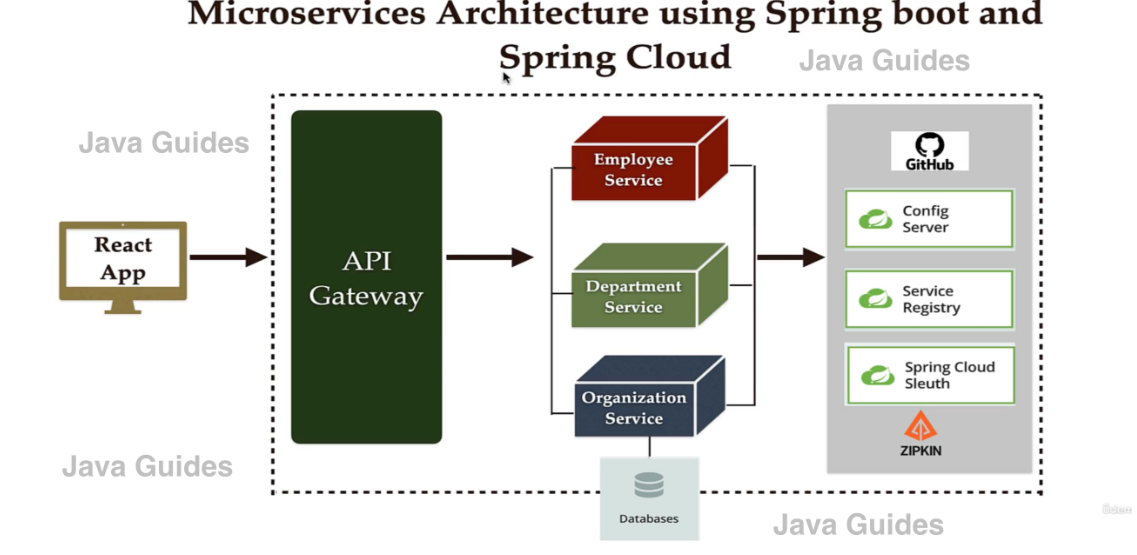


Each microservice has its own database. For example, **product service** has its own database, **inventory service** has its own database, and **stock service** has its own database.  
  
In the microservices project, all the microservices are loosely coupled. So loosely coupled, meaning all the services 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**

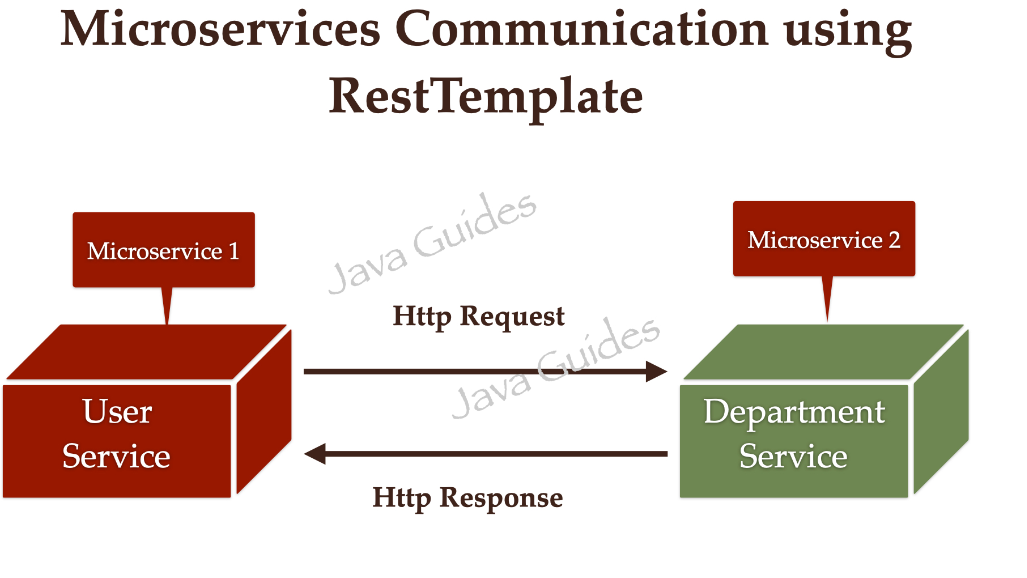
In the case of Asynchronous Communication, The client sends a request and does not wait for a response from the service. The client will continue executing its task - It doesn’t wait for the response from the service.

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.

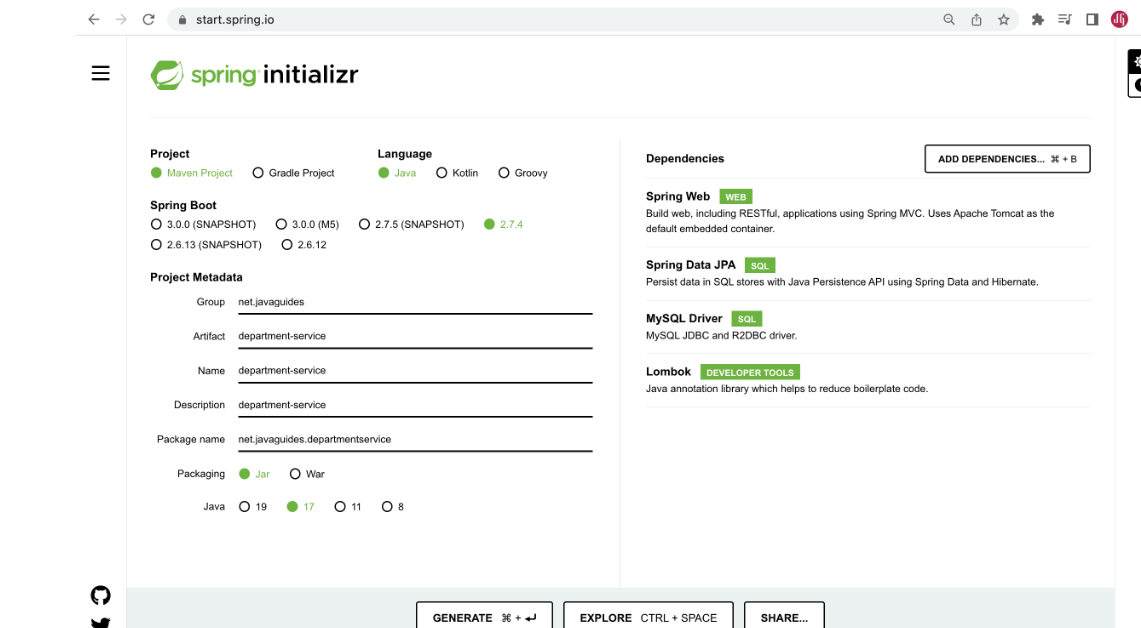
# Creating DepartmentService Microservice

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.

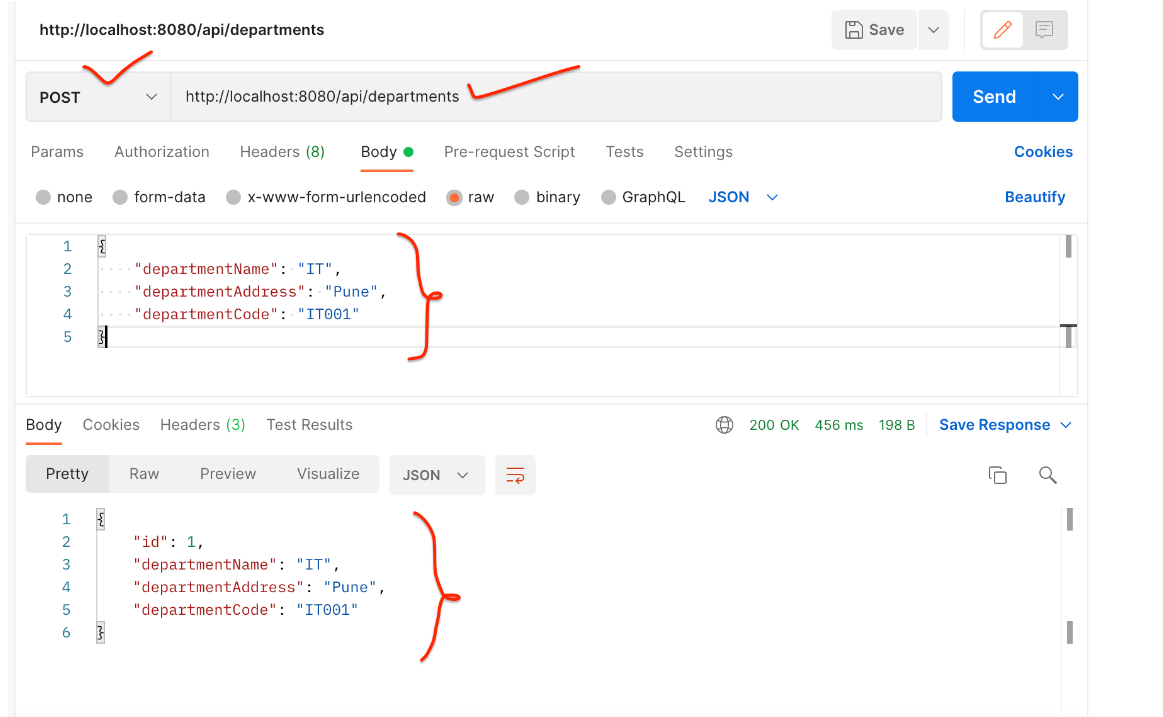
1. From the root directory of the application and type the following command to run it -

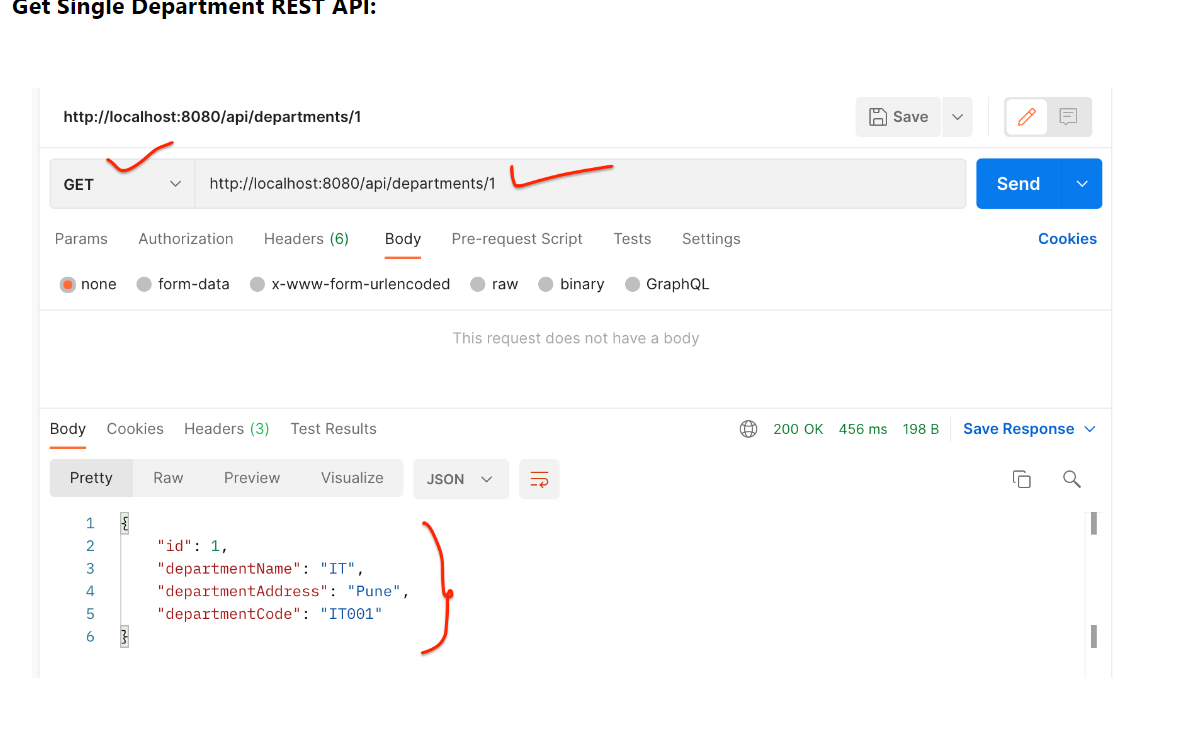
$ mvn spring-boot:run

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:





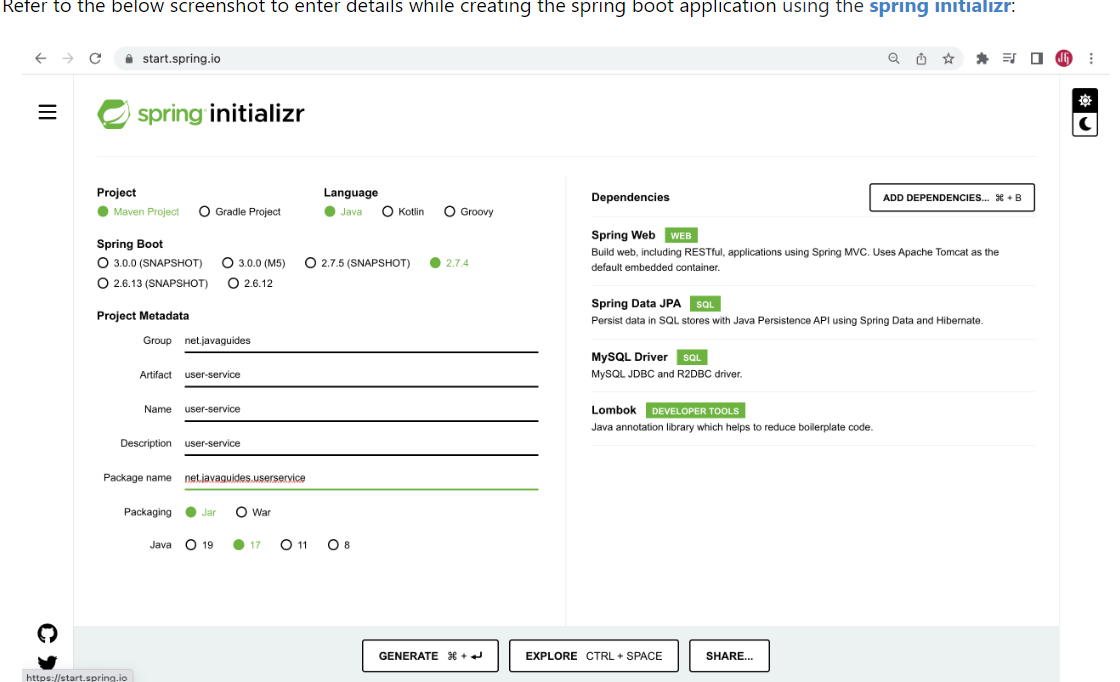
# 2. Creating UserService Microservice

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:

