Monolithic vs. Microservices Architecture

Monolithic Architecture

When developing a server-side application you can start it with a modular hexagonal or layered architecture which consists of different types of components:

- Presentation — responsible for handling HTTP requests and responding with either HTML or JSON/XML (for web services APIs).
- Business logic — the application's business logic.
- Database access — data access objects responsible for access the database.
- Application integration — integration with other services (e.g. via messaging or REST API).

Despite having a logically modular architecture, the application is packaged and deployed as a monolith.

Benefits of Monolithic Architecture

- Simple to develop.
- Simple to test. For example you can implement end-to-end testing by simply launching the application and testing the UI with Selenium.
- Simple to deploy. You just have to copy the packaged application to a server.
- Simple to scale horizontally by running multiple copies behind a load balancer.

In the early stages of the project it works well and basically most of the big and successful applications which exist today were started as a monolith.

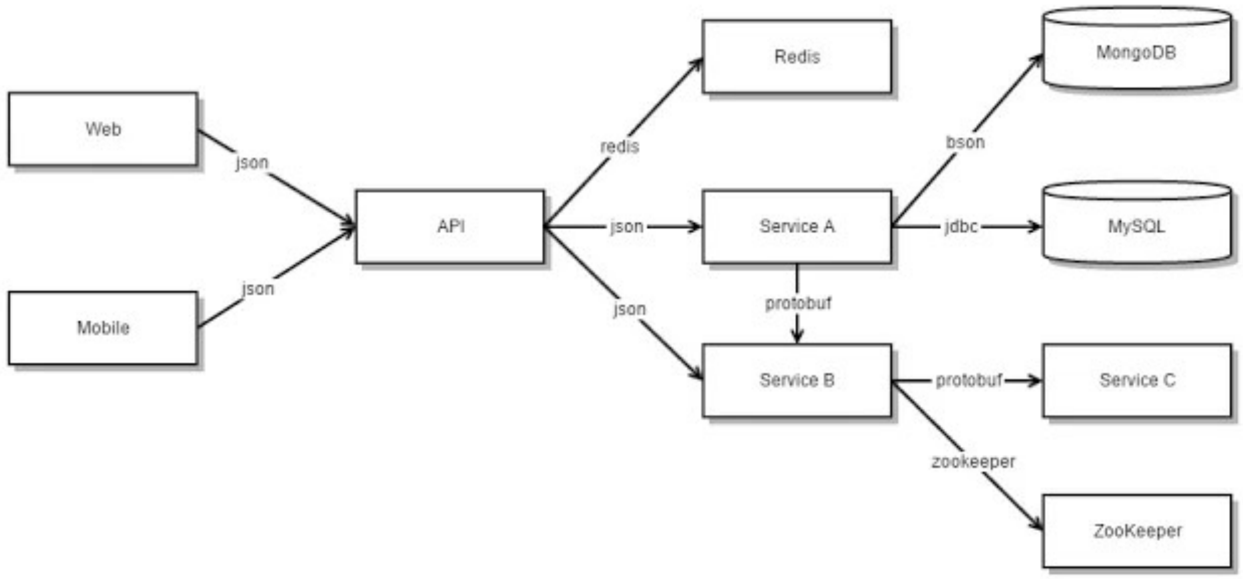
Drawbacks of Monolithic Architecture

- This simple approach has a limitation in size and complexity.
- Application is too large and complex to fully understand and made changes fast and correctly.
- The size of the application can slow down the start-up time.
- You must redeploy the entire application on each update.
- Impact of a change is usually not very well understood which leads to do extensive manual testing.
- Continuous deployment is difficult.
- Monolithic applications can also be difficult to scale when different modules have conflicting resource requirements.
- Another problem with monolithic applications is reliability. Bug in any module (e.g. memory leak) can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug will impact the availability of the entire application.
- Monolithic applications has a barrier to adopting new technologies. Since changes in frameworks or languages will affect an entire application it is extremely expensive in both time and cost.

Microservices Architecture

The idea is to split your application into a set of smaller, interconnected services instead of building a single monolithic application. Each microservice is a small application that has its own hexagonal architecture consisting of business logic

along with various adapters. Some microservices would expose a REST, RPC or message-based API and most services consume APIs provided by other services. Other microservices might implement a web UI.
The Microservice architecture pattern significantly impacts the relationship between the application and the database. Instead of sharing a single database schema with other services, each service has its own database schema. On the one hand, this approach is at odds with the idea of an enterprise-wide data model. Also, it often results in duplication of some data. However, having a database schema per service is essential if you want to benefit from microservices, because it ensures loose coupling. Each of the services has its own database. Moreover, a service can use a type of database that is best suited to its needs, the so-called polyglot persistence architecture.
Some APIs are also exposed to the mobile, desktop, web apps. The apps don't, however, have direct access to the back-end services. Instead, communication is mediated by an intermediary known as an API Gateway. The API Gateway is responsible for tasks such as load balancing, caching, access control, API metering, and monitoring.



The Microservice architecture pattern corresponds to the Y-axis scaling of the Scale Cube model of scalability.
Benefits of Microservices Architecture

- It tackles the problem of complexity by decomposing application into a set of manageable services which are much faster to develop, and much easier to understand and maintain.
- It enables each service to be developed independently by a team that is focused on that service.
- It reduces barrier of adopting new technologies since the developers are free to choose whatever technologies make sense for their service and not bounded to the choices made at the start of the project.

- Microservice architecture enables each microservice to be deployed independently. As a result, it makes continuous deployment possible for complex applications.
- Microservice architecture enables each service to be scaled independently.

Drawbacks of Microservices Architecture

- Microservices architecture adding a complexity to the project just by the fact that a microservices application is a distributed system. You need to choose and implement an inter-process communication mechanism based on either messaging or RPC and write code to handle partial failure and take into account other fallacies of distributed computing.
- Microservices has the partitioned database architecture. Business transactions that update multiple business entities in a microservices-based application need to update multiple databases owned by different services. Using distributed transactions is usually not an option and you end up having to use an eventual consistency based approach, which is more challenging for developers.
- Testing a microservices application is also much more complex then in case of monolithic web application. For a similar test for a service you would need to launch that service and any services that it depends upon (or at least configure stubs for those services).
- It is more difficult to implement changes that span multiple services. In a monolithic application you could simply change the corresponding modules, integrate the changes, and deploy them in one go. In a Microservice architecture you need to carefully plan and coordinate the rollout of changes to each of the services.
- Deploying a microservices-based application is also more complex. A monolithic application is simply deployed on a set of identical servers behind a load balancer. In contrast, a microservice application typically consists of a large number of services. Each service will have multiple runtime instances. And each instance need to be configured, deployed, scaled, and monitored. In addition, you will also need to implement a service discovery mechanism. Manual approaches to operations cannot scale to this level of complexity and successful deployment a microservices application requires a high level of automation.

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.

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.

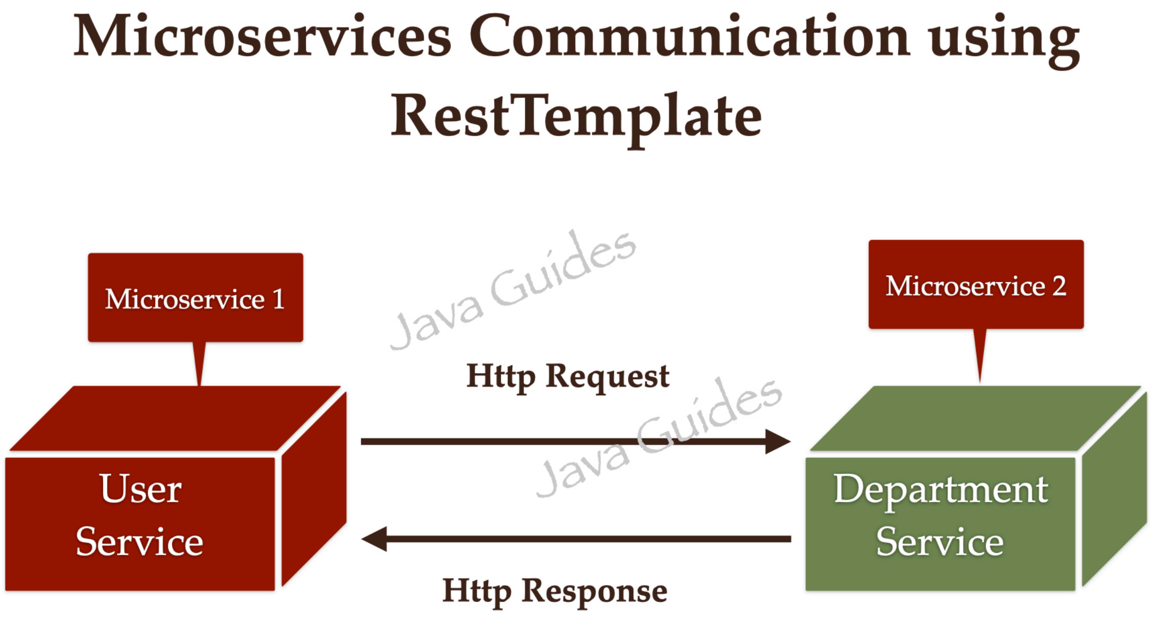
Asynchronous Communication

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.

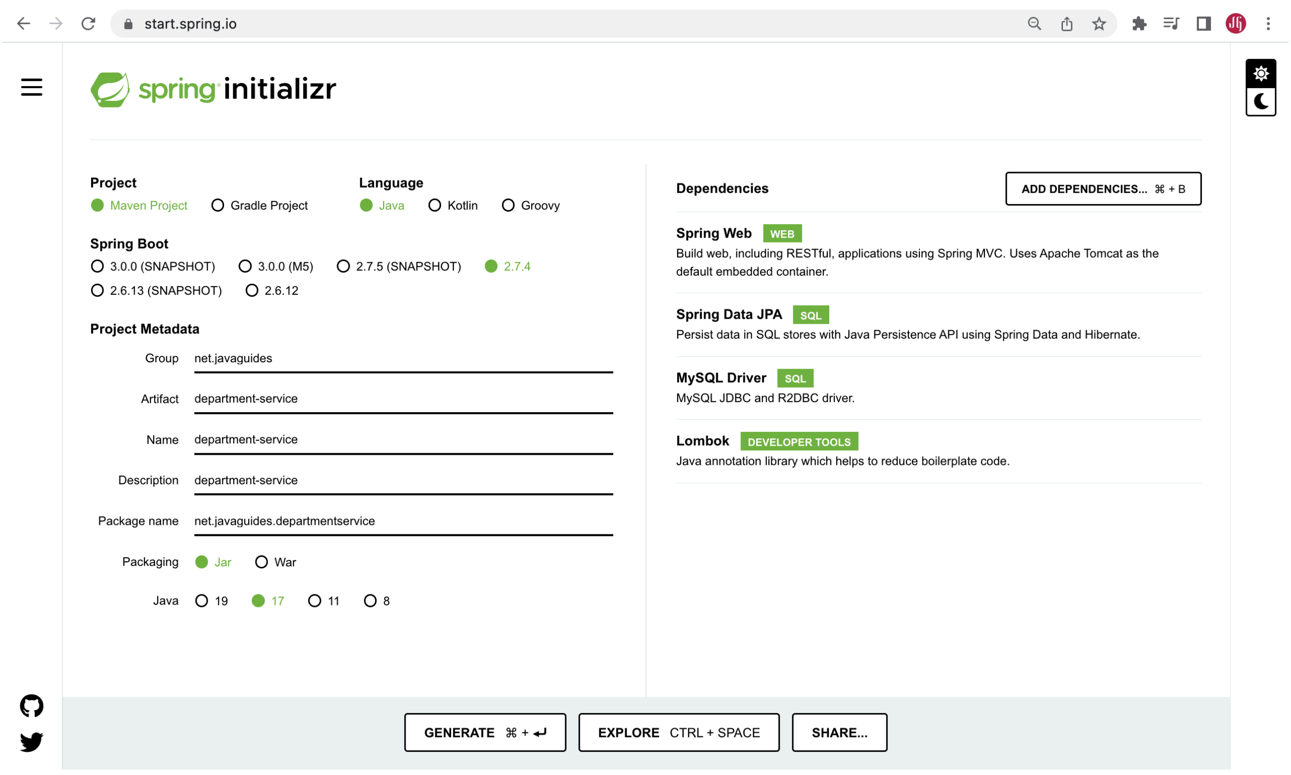
Creating DepartmentService Microservice

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**.

Refer to the below screenshot to enter details while creating the spring boot application using the **spring initializr**:



Project: [✓ Maven Project] Gradle Project

Language: [✓ Java] Kotlin Groovy

Spring Boot: 3.0.0 (SNAPSHOT) 3.0.0 (M5) 2.7.5 (SNAPSHOT) [✓ 2.7.4] 2.6.13 (SNAPSHOT) 2.6.12

Project Metadata

Group	net.javaguides
Artifact	department-service
Name	department-service
Description	department-service
Package name	net.javaguides.departmentservice
Packaging	[✓ Jar] War
Java	19 [✓ 17] 11 8

Dependencies

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

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

MySQL Driver SQL
MySQL JDBC and R2DBC driver.

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

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
<?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>com.example</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 `Department`entity 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 com.example.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 com.example.departmentservice.repository;

import com.example.departmentservice.entity.Department;
import org.springframework.data.jpa.repository.JpaRepository;

public interface DepartmentRepository extends JpaRepository<Department, Long>
{
}
```

DepartmentService - Create Service Layer

DepartmentService Interface

```
package com.example.departmentservice.service;

import com.example.departmentservice.entity.Department;

public interface DepartmentService {
    Department saveDepartment(Department department);

    Department getDepartmentById(Long departmentId);
}
```

DepartmentServiceImpl class

```
package com.example.departmentservice.service.impl;

import lombok.AllArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import com.example.departmentservice.entity.Department;
import com.example.departmentservice.repository.DepartmentRepository;
import com.example.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 com.example.departmentservice.controller;

import lombok.AllArgsConstructor;
import com.example.departmentservice.entity.Department;
import com.example.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

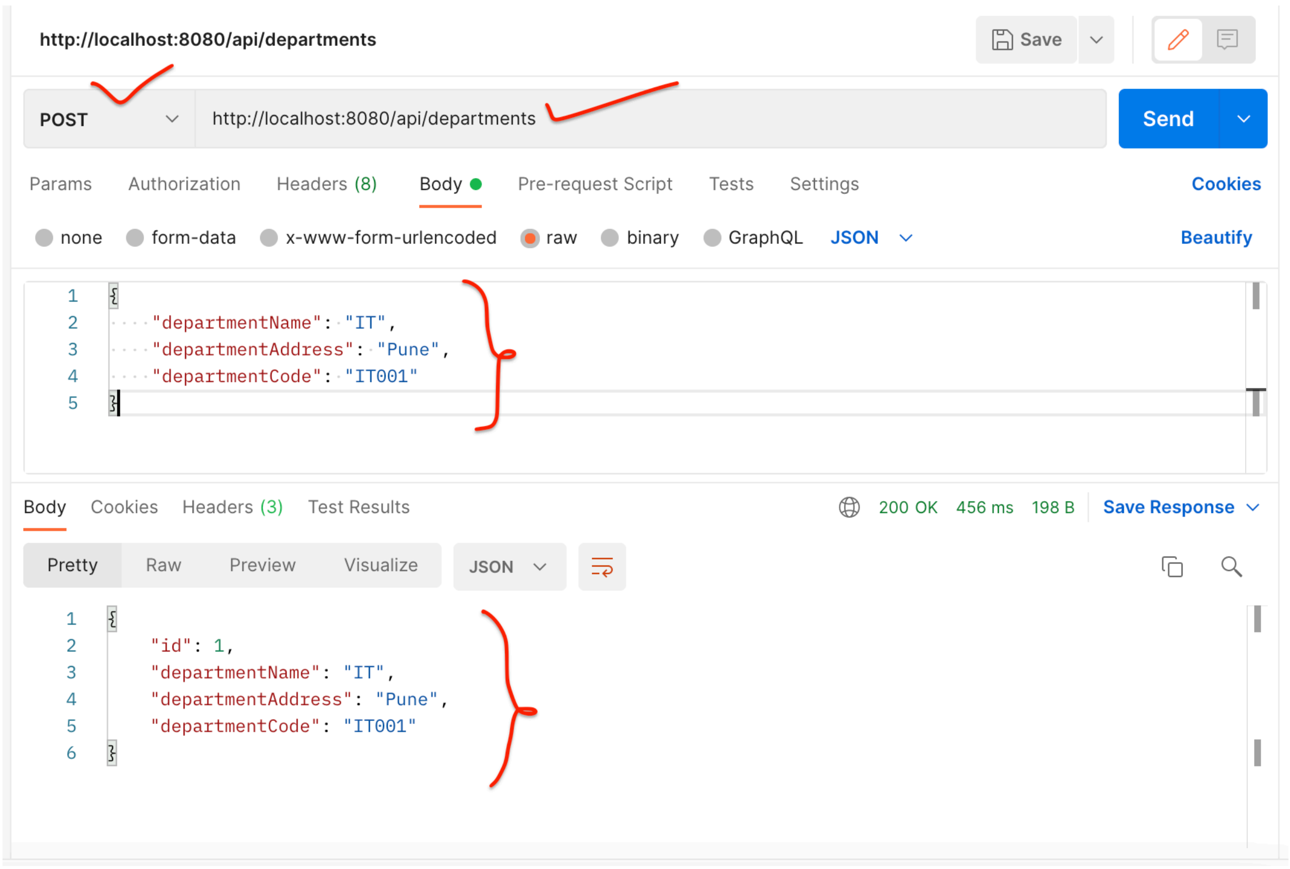
Two ways we can start the standalone Spring boot application.
1. From the root directory of the application and type the following command to run it -

```
$ 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/**.

DepartmentService - Test REST APIs using Postman Client

Save Department REST API:



POST http://localhost:8080/api/departments

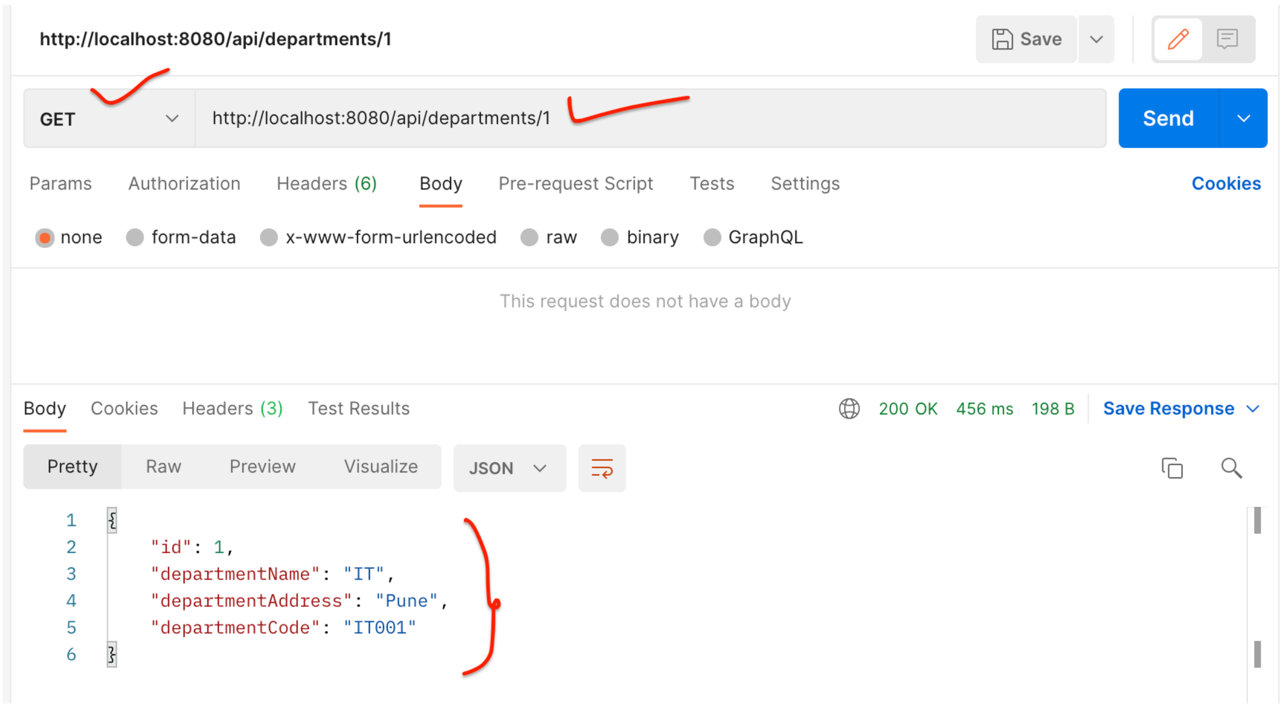
```
{
    "departmentName": "IT",
    "departmentAddress": "Pune",
    "departmentCode": "IT001"
}
```

200 OK 456 ms 198 B

Body: JSON

```
{
    "id": 1,
    "departmentName": "IT",
    "departmentAddress": "Pune",
    "departmentCode": "IT001"
}
```

Get Single Department REST API:


http://localhost:8080/api/departments/1

GET http://localhost:8080/api/departments/1

This request does not have a body

200 OK 456 ms 198 B

```
{
    "id": 1,
    "departmentName": "IT",
    "departmentAddress": "Pune",
    "departmentCode": "IT001"
}
```

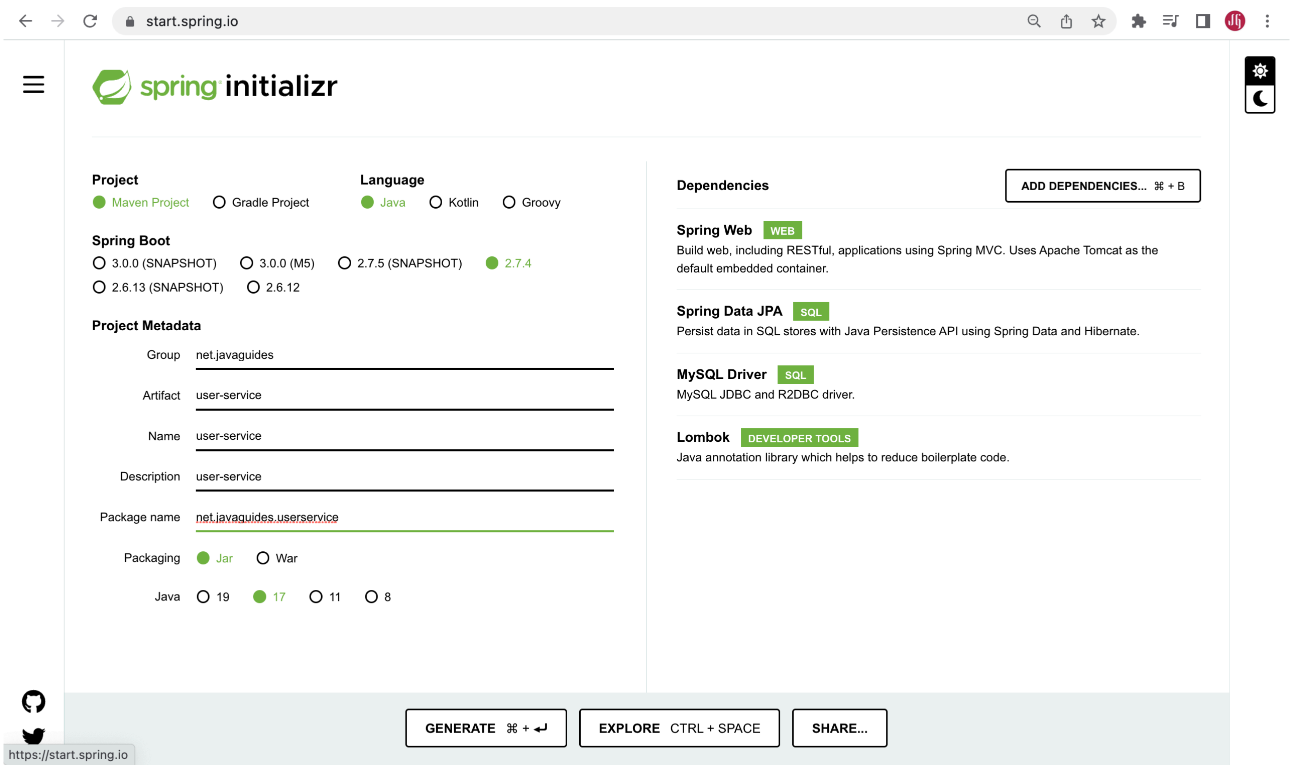
2. Creating UserService Microservice

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**.

Refer to the below screenshot to enter details while creating the spring boot application using the **spring initializr**:


start.spring.io

spring initializr

Project
[✓ Maven Project] Gradle Project

Language
[✓ Java] Kotlin Groovy

Spring Boot
3.0.0 (SNAPSHOT) 3.0.0 (M5) 2.7.5 (SNAPSHOT) [✓ 2.7.4] 2.6.13 (SNAPSHOT) 2.6.12

Project Metadata

Group	net.javaguides
Artifact	user-service
Name	user-service
Description	user-service
Package name	net.javaguides.userservice
Packaging	[✓ Jar] War
Java	19 [✓ 17] 11 8

Dependencies

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

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

MySQL Driver SQL
MySQL JDBC and R2DBC driver.

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

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>com.example</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>
```

```xml
			<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 `User`entity 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 com.example.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 com.example.userservice.repository;

import com.example.userservice.entity.User;
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {

}
```

UserService - Create DTO Classes

DepartmentDto

```
package com.example.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 com.example.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 com.example.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 com.example.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 com.example.userservice.service;

import com.example.userservice.dto.ResponseDto;
import com.example.userservice.entity.User;

public interface UserService {
    User saveUser(User user);

    ResponseDto getUser(Long userId);
}
```

UserServiceImpl class

```
package com.example.userservice.service.impl;

import lombok.AllArgsConstructor;
import com.example.userservice.dto.DepartmentDto;
import com.example.userservice.dto.ResponseDto;
import com.example.userservice.dto.UserDto;
import com.example.userservice.entity.User;
import com.example.userservice.repository.UserRepository;
import com.example.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 com.example.userservice.controller;

import lombok.AllArgsConstructor;
import com.example.userservice.dto.ResponseDto;
import com.example.userservice.entity.User;
import com.example.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

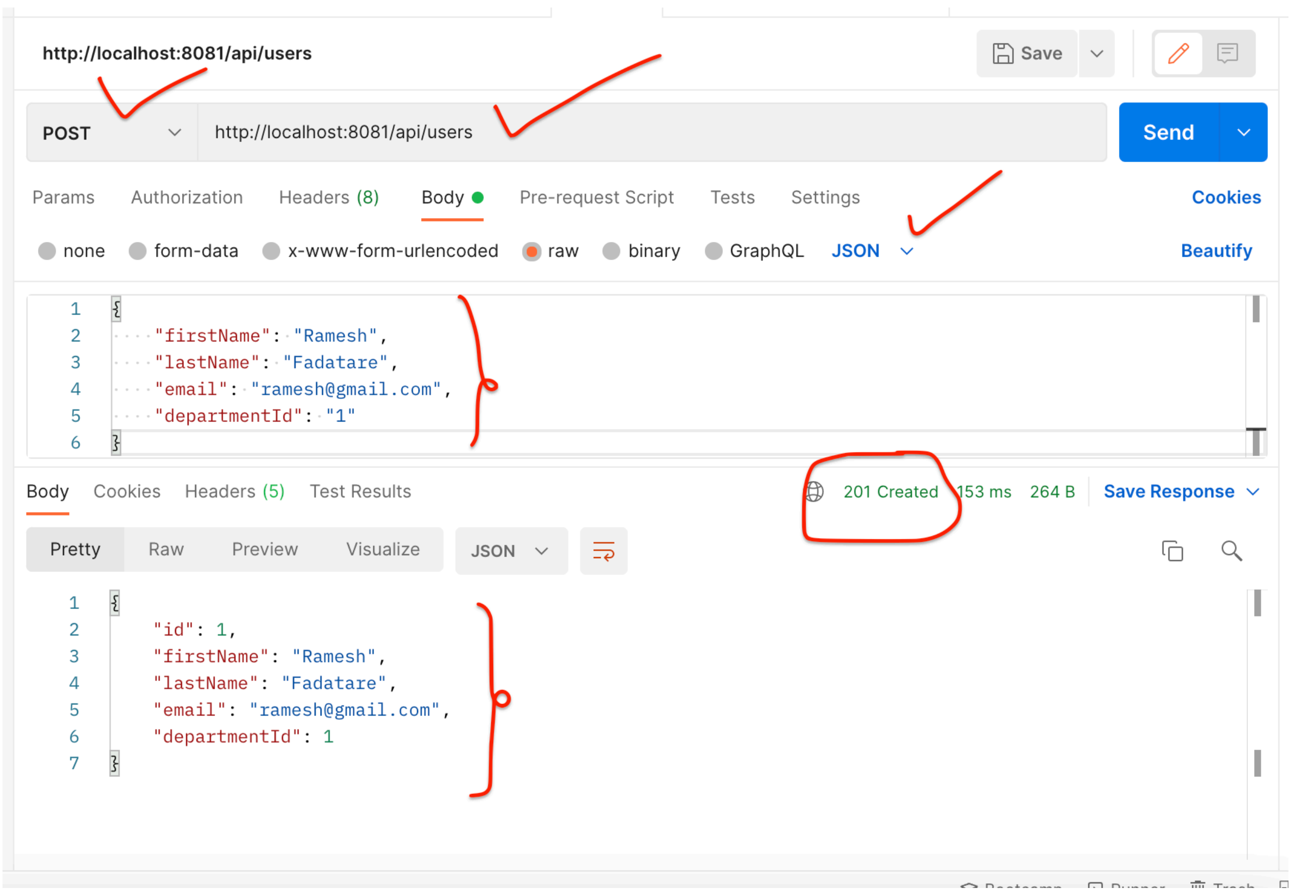
Two ways we can start the standalone Spring boot application.
1. From the root directory of the application and type the following command to run it -

```
$ 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/**.

UserService - Test REST APIs using Postman Client

Save User REST API:



http://localhost:8081/api/users

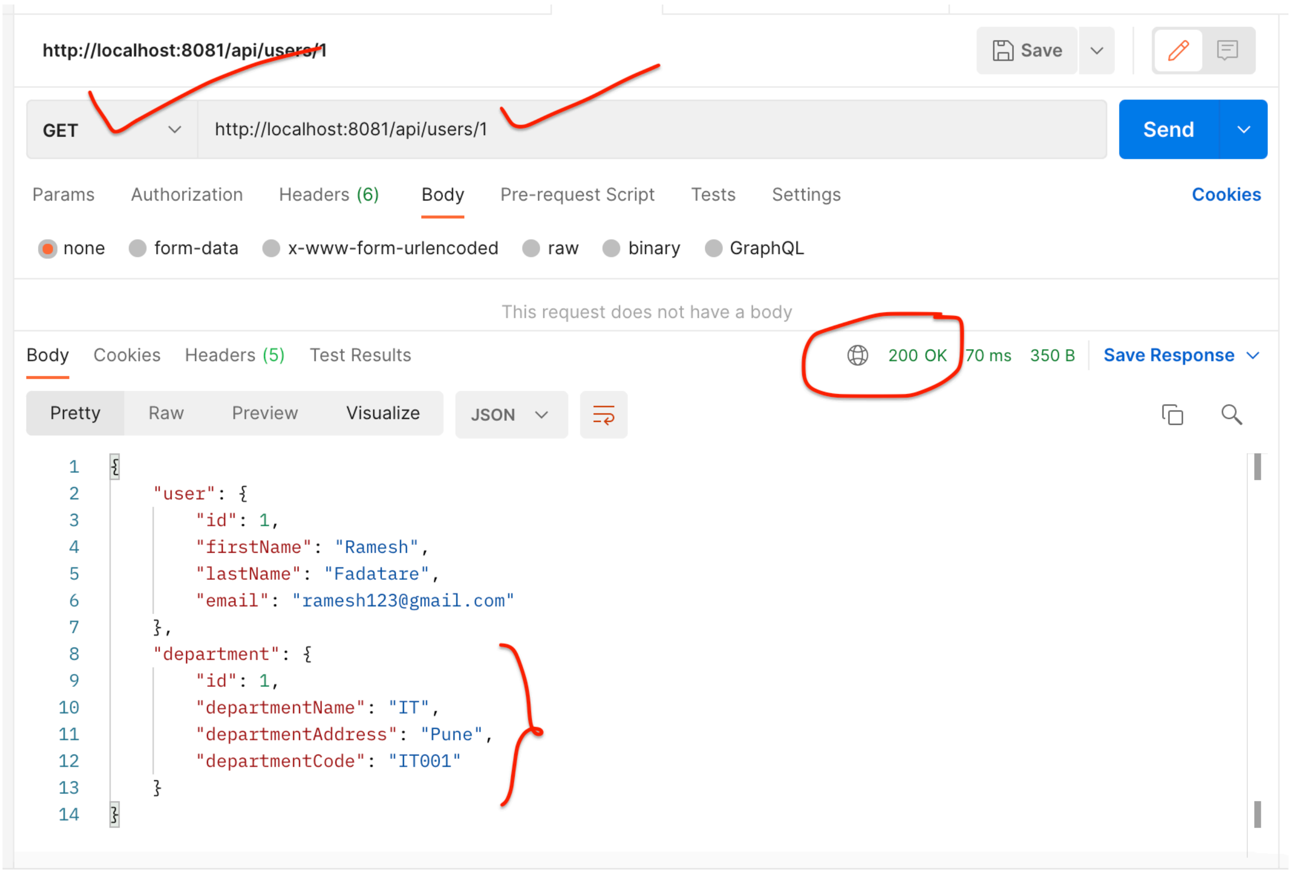
POST http://localhost:8081/api/users

```
{
    "firstName": "Ramesh",
    "lastName": "Fadatare",
    "email": "ramesh@gmail.com",
    "departmentId": "1"
}
```

201 Created 153 ms 264 B

```
{
    "id": 1,
    "firstName": "Ramesh",
    "lastName": "Fadatare",
    "email": "ramesh@gmail.com",
    "departmentId": 1
}
```

Get User REST API:



http://localhost:8081/api/users/1

GET http://localhost:8081/api/users/1

Body: none

This request does not have a body

200 OK 70 ms 350 B

```
{
    "user": {
        "id": 1,
        "firstName": "Ramesh",
        "lastName": "Fadatare",
        "email": "ramesh123@gmail.com"
    },
    "department": {
        "id": 1,
        "departmentName": "IT",
        "departmentAddress": "Pune",
        "departmentCode": "IT001"
    }
}
```

Note that the response contains a Department for a User. This demonstrates that we have successfully made a REST API call from UserService to DepartmentService.

Conclusion

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