March 28, 2024

Spring Boot Microservices Tutorial - Part 1

Introduction

In this **Spring Boot Microservices Tutorial** series, you will learn how to develop applications with Microservices Architecture using Spring Boot and Spring Cloud and deploy them using Docker and Kubernetes.

We will cover several concepts and Microservices Architectural Patterns as part of this tutorial series, here are the topics we are going to cover in each part:

- Part -1 covers building REST-based applications using **Spring Boot 3** and following several best practices.
- Part -2 of this tutorial series covers, the Synchronous Inter-Service Communication Pattern using **Spring Cloud Open Feign**
- Part - 3 covers the Service Discovery Pattern using **Spring Cloud Netflix Eureka**
- Part - 4 covers the API Gateway Pattern using **Spring Cloud Gateway**
- Part - 5 covers the Microservices Security using Keycloak

- Part - 6 covers the Circuit Breaker Pattern using **Spring Cloud CircuitBreaker** with **Resilience4J**
- Part - 7 covers the Event Driven Architecture Pattern using Kafka
- Part - 8 covers the Observability Pattern, and we will be implementing Distributed Tracing using **Open Telemetry** and **Grafana Tempo**, we will be implementing the Log Aggregation Pattern to view the logs of our services using **Grafana Loki**, and we will be using **Prometheus** to collect the Metrics and **Grafana** to visualize the metrics in a dashboard.
- In Part - 9, we will be containerizing all our applications using **Docker**. We will see how to run our applications using Docker Compose
- In Part - 10, we will migrate our Docker Compose Workloads to Kubernetes

Application Overview

We will be building a simple e-commerce application where customers can order products. Our application contains the following services:

- Product Service
- Order Service
- Inventory Service
- Notification Service

To focus on the principles of Spring Cloud and Microservices, we will develop services with essential functionality rather than creating fully-featured e-commerce services.

Download Source Code

You can download the source code of this project through Github – https://github.com/SaiUpadhyayula/spring-boot-microservices/tree/initial-setup

Architecture Diagram of the Project

Here is the architecture diagram of the project we are going to cover in this tutorial series


Architecture Diagram for Spring Boot Microservices Project

Creating our First Microservice: Product Service

Let's start creating our first microservice (Product Service). As discussed before, we will keep this service simple and only include the most important features.

We are going to expose a REST API endpoint that will CREATE and READ products.

Service Operation	HTTP METHOD	Service End point
CREATE PRODUCT	POST	/api/product/
READ ALL PRODUCTS	GET	/api/product/

Product Service REST Operations

To create the project, let's go to start.spring.io and create our project based on the following configuration:


Start.Spring.IO Configuration for Product Service

Here are the dependencies you need to add:

- Lombok
- Spring Web
- Test Containers
- Spring Data MongoDB
- Java 21
- Maven as the build tool

We are going to use **MongoDB** as the database backing our **Product Service**

After adding the necessary configuration, click on the **Generate** button, and the source code should be downloaded to your machine.

Unzip the source code and open it in your favorite IDE.

After opening the project, run the below command to build the project:

```
mvn clean verify
```

The application should be built successfully without any errors.

Download MongoDB using Docker and Docker Compose

We will be using Docker to install the necessary software like Databases, Message Queues, and other required software for this project.

If you don't have Docker installed on your machine, you can download it at this link: https://docs.docker.com/get-docker/

Once Docker is installed, create a file called **docker-compose.yml** in the root folder:

```
version: '4'
services:
  mongo:
    image: mongo:7.0.5
    container_name: mongo
    ports:
      - "27017:27017"
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: password
      MONGO_INITDB_DATABASE: product-service
    volumes:
      - ./docker/mongodb/data:/data/db
```

We have to configure the MongoDB URI Details inside the **application.properties** file:

```
spring.data.mongodb.uri=mongodb://root:password@localhost:27017/produ
```

> *If you are not aware of how to work with MongoDB and Spring Boot, have a look at the Spring Boot MongoDB REST API Tutorial*

Creating the Create and Read Endpoints

Let's create the below model class which acts as the domain for the Products.

Product.java

```
package com.programmingtechie.productservice.model;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.math.BigDecimal;

@Document(value = "product")
@AllArgsConstructor
@NoArgsConstructor
@Builder
@Data
public class Product {

    @Id
    private String id;
    private String name;
    private String description;
    private BigDecimal price;
}
```

Next, let's create the Spring Data MongoDB interface for the Product class - ProductRepository.java

ProductRepository.java

```
package com.programming.techie.productservice.repository;

import com.programming.techie.productservice.model.Product;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface ProductRepository extends MongoRepository<Product, S
}
```

Now let's create the service class - ProductService.java, which contains the actual business logic of our product-service, that is responsible for creating and reading the products from the database.

ProductService.java

```
package com.programmingtechie.productservice.service;

import com.programmingtechie.productservice.dto.ProductRequest;
import com.programmingtechie.productservice.dto.ProductResponse;
import com.programmingtechie.productservice.model.Product;
import com.programmingtechie.productservice.repository.ProductReposit
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Service;
```

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

import java.util.List;

@Service
@RequiredArgsConstructor
@Slf4j
public class ProductService {

    private final ProductRepository productRepository;

    public void createProduct(ProductRequest productRequest) {
        Product product = Product.builder()
                .name(productRequest.name())
                .description(productRequest.description())
                .price(productRequest.price())
                .build();

        productRepository.save(product);
        log.info("Product {} is saved", product.getId());
    }

    public List<ProductResponse> getAllProducts() {
        List<Product> products = productRepository.findAll();

        return products.stream().map(this::mapToProductResponse).toLi
    }

    private ProductResponse mapToProductResponse(Product product) {
        return new ProductResponse(product.getId(), product.getName()
                product.getDescription(), product.getPrice());
    }
}
```

Next, we need the Controller class that exposes the POST and GET endpoint to create and read the products.

ProductRestController.java

```
package com.programmingtechie.productservice.controller;

import com.programmingtechie.productservice.dto.ProductRequest;
import com.programmingtechie.productservice.dto.ProductResponse;
import com.programmingtechie.productservice.service.ProductService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/product")
@RequiredArgsConstructor
public class ProductController {

    private final ProductService productService;

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public void createProduct(@RequestBody ProductRequest productRequ
        productService.createProduct(productRequest);
    }

    @GetMapping
    @ResponseStatus(HttpStatus.OK)
    public List<ProductResponse> getAllProducts() {
```

```
            return productService.getAllProducts();
        }

    }
```

The ProductController class uses ProductRequest and ProductResponse as the DTOs, let's also create those records

ProductRequest.java

```
package com.programmingtechie.productservice.dto;

import java.math.BigDecimal;

public record ProductRequest(String name, String description, BigDeci
}
```

ProductResponse.java

```
package com.programmingtechie.productservice.dto;

import java.math.BigDecimal;

public record ProductResponse(String id, String name, String descript
}
```

Testing the Product Service APIs

Let's start the application and test our two Endpoints

We will start by creating a product, by calling the URL http://localhost:8080/api/product with HTTP Method POST, this REST call should return a status 201.


Create Product Test from Postman

Now let's make a GET call to the URL - http://localhost:8080/api/product to test whether the created product is returned as a response or not.


Get All Products Test from Postman

Write Integration Tests for Product Service

Let's write a couple of Integration Tests to test our Create Product and Get Products Endpoints, for the integration test, as we need a real Mongo database, we will be using TestContainers to spin up a MongoDB Container as part of the test.

If you are unaware of Testcontainers, you can read more about it here: https://testcontainers.com/

Before writing our tests, we need to add one dependency to our **pom.xml** file:

```
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured</artifactId>
    <version>5.3.2</version>
```

```
        </dependency>
```

We added the rest-assured dependency as we need a real HTTP Client to call the endpoints while running the Integration Tests.

Let's create the integration test with the below code:

ProductServiceApplicationTests.java

```
package com.programmingtechie.productservice;

import com.programmingtechie.productservice.dto.ProductRequest;
import io.restassured.RestAssured;
import org.hamcrest.Matchers;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.server.LocalServerPort;
import org.springframework.boot.testcontainers.service.connection.Ser
import org.testcontainers.containers.MongoDBContainer;

import java.math.BigDecimal;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM
class ProductServiceApplicationTests {

    @ServiceConnection
    static MongoDBContainer mongoDBContainer = new MongoDBContainer("
    @LocalServerPort
    private Integer port;

    @BeforeEach
```

```
    void setup() {
        RestAssured.baseURI = "http://localhost";
        RestAssured.port = port;
    }

    static {
        mongoDBContainer.start();
    }

    @Test
    void shouldCreateProduct() throws Exception {
        ProductRequest productRequest = getProductRequest();

        RestAssured.given()
                .contentType("application/json")
                .body(productRequest)
                .when()
                .post("/api/product")
                .then()
                .log().all()
                .statusCode(201)
                .body("id", Matchers.notNullValue())
                .body("name", Matchers.equalTo(productRequest.name())
                .body("description", Matchers.equalTo(productRequest.
                .body("price", Matchers.is(productRequest.price().int
    }

    private ProductRequest getProductRequest() {
        return new ProductRequest("iPhone 13", "iPhone 13", BigDecima
    }

}
```

Create Second Microservice - Order Service

Now let's create our 2nd Microservice, the order service, this service contains only one endpoint, to submit an order.

Service Operation	Endpoint Method	Service Endpoint
PLACE ORDER	POST	/api/order

Operations for Order Service

Let's create the project, by visiting the site start.spring.io

Create the project with below dependencies:

- Spring Web
- Lombok
- Spring Data JPA
- MySQL Driver
- Flyway Migration
- Testcontainers
- We will be using Java 21 also for this service and Maven as the build tool.

 Order Service Starter Configuratione

In Order Service, we are going to use **MySQL** Database, so let's go ahead and download **MySQL** using docker-compose.

Create a docker-compose.yml file with the below contents:

```
version: '4'
services:
  mysql:
    image: mysql:8.3.0
    container_name: mysql
    ports:
      - "3306:3306"
    environment:
      MYSQL_ROOT_PASSWORD: mysql
    volumes:
      - ./mysql/init.sql:/docker-entrypoint-initdb.d/init.sql
      - ./docker/mysql/data:/var/lib/mysql
```

We need to create the database schema during the start-up of our MySQL Database, for that we added the line **./mysql/init.sql:/docker-entrypoint-initdb.d/init.sql** which asks docker to copy the SQL file from the folder '**mysql**' into the **docker-entrypoint-initdb.d** location and executes the SQL file.

If we don't add the above step, then we need to manually create the database.

Now let's configure our project to use MySQL by adding below properties in the **application.properties** file:

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

```
spring.datasource.password=mysql
spring.jpa.hibernate.ddl-auto=none
server.port=8081
```

- We are using the **spring.jpa.hibernate.ddl-auto** property as **none** because we don't want Hibernate to create the database tables and manage migrations, we will be handling that using the Flyway library.
- Notice that we are running the order-service application on port 8081, as product-service is already running on port 8080

Database Migrations with Flyway

As mentioned before, we will be using Flyway to execute database migrations, the necessary dependencies for it are already added in the generated project. Here are the dependencies for Flyway:

```
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-core</artifactId>
</dependency>
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-mysql</artifactId>
</dependency>
```

By using Flyway, we can provide the necessary SQL scripts that will be executed whenever we need to change our database schema. We need to provide these scripts under the **src/main/resources/db/migration** folder.

Flyway will look for the scripts under this particular folder, and Flyway will also follow a particular naming convention to identify the SQL scripts, we need to name the files like below:

V<Number>__file-name.sql

Example: V1__init.sql, V2__add_products.sql, etc.

Note that the number, inside the name of the SQL file, needs to be incremented for each database migration you want to run.

Let's create the below file to create the Order table

V1__init.sql

```
CREATE TABLE `t_orders`
(
    `id`          bigint(20) NOT NULL AUTO_INCREMENT,
    `order_number` varchar(255) DEFAULT NULL,
    `sku_code`  varchar(255),
    `price`    decimal(19, 2),
    `quantity` int(11),
    PRIMARY KEY (`id`)
);
```

Before running the migrations, let's create the necessary Model classes and the Submit Order Endpoint.

NOTE: I simplified some logic and the table structure recently. I removed the OrderLineItems table and the related logic to make the who logic simple. So you may find some discrepancies compared to the first version of the article which

contains references to the OrderLineItems table.

Order.java

```
package com.programmingtechie.orderservice.model;

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

import java.math.BigDecimal;

@Entity
@Table(name = "t_orders")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String orderNumber;
    private String skuCode;
    private BigDecimal price;
    private Integer quantity;
}
```

OrderRepository.java

OrderRepository.java

```
package com.programmingtechie.orderservice.repository;

import com.programmingtechie.orderservice.model.Order;
import org.springframework.data.jpa.repository.JpaRepository;

public interface OrderRepository extends JpaRepository<Order, Long> {
}
```

OrderService.java

```
package com.programmingtechie.orderservice.service;

import com.programmingtechie.orderservice.dto.OrderRequest;
import com.programmingtechie.orderservice.model.Order;
import com.programmingtechie.orderservice.repository.OrderRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.UUID;

@Service
@RequiredArgsConstructor
@Transactional
public class OrderService {

    private final OrderRepository orderRepository;
```

```
    public void placeOrder(OrderRequest orderRequest) {
        var order = mapToOrder(orderRequest);
        orderRepository.save(order);
    }

    private static Order mapToOrder(OrderRequest orderRequest) {
        Order order = new Order();
        order.setOrderNumber(UUID.randomUUID().toString());
        order.setPrice(orderRequest.price());
        order.setQuantity(orderRequest.quantity());
        order.setSkuCode(orderRequest.skuCode());
        return order;
    }
}
```

OrderController.java

```
package com.programmingtechie.orderservice.controller;

import com.programmingtechie.orderservice.dto.OrderRequest;
import com.programmingtechie.orderservice.service.OrderService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/order")
@RequiredArgsConstructor
public class OrderController {
```

```
    private final OrderService orderService;

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public String placeOrder(@RequestBody OrderRequest orderRequest)
        orderService.placeOrder(orderRequest);
        return "Order Placed Successfully";
    }
}
```

OrderRequest.java

```
package com.programmingtechie.orderservice.dto;

import java.math.BigDecimal;

public record OrderRequest(Long id, String skuCode, BigDecimal price,
}
```

Testing the Application through Postman

Now Let's test our endpoints using Postman, before that let's start our application by running the **OrderServiceApplication.java** class

Let's make a POST request to the URL http://localhost:8081/api/order as seen in the below screenshot:


Testing Order Service through Postman

The request should be successful with HTTP Status 201 Created and the response body should have the text "Order Placed Successfully".

Writing Integration Tests for Order Service

Let's write the integration tests also for the OrderService.

OrderServiceApplicationTests.java

```
package com.programmingtechie.orderservice;

import io.restassured.RestAssured;
import org.hamcrest.Matchers;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.server.LocalServerPort;
import org.springframework.boot.testcontainers.service.connection.Ser
import org.testcontainers.containers.MySQLContainer;

import static org.hamcrest.MatcherAssert.assertThat;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM
class OrderServiceApplicationTests {

    @ServiceConnection
    static MySQLContainer mySQLContainer = new MySQLContainer("mysql:
```

```
    @LocalServerPort
    private Integer port;

    @BeforeEach
    void setup() {
        RestAssured.baseURI = "http://localhost";
        RestAssured.port = port;
    }

    static {
        mySQLContainer.start();
    }

    @Test
    void shouldSubmitOrder() {
        String submitOrderJson = """
                {
                    "skuCode": "iphone_15",
                    "price": 1000,
                    "quantity": 1
                }
                """;


        var responseBodyString = RestAssured.given()
                .contentType("application/json")
                .body(submitOrderJson)
                .when()
                .post("/api/order")
                .then()
                .log().all()
                .statusCode(201)
                .extract()
                .body().asString();
```

```
            assertThat(responseBodyString, Matchers.is("Order Placed Succ
        }
    }
```

Creating Third Microservice - Inventory Service

Now let's create our 3rd microservice the Inventory Service. Go to start.spring.io and select the below dependencies:

- Spring Web
- Spring Data JPA
- Lombok
- Flyway
- MySQL JDBC Driver
- Test Containers
- Java 21 and Maven as Build tool

The Inventory Service exposes only 1 endpoint, similar to the Order Service, here is a brief overview of the endpoint:

Service Operation	Endpoint Method	Service Endpoint
GET Inventory	GET	/api/inventory

REST Operations for Inventory Service

As we are using MySQL Database also for the inventory service, we need to first update the mysql/init.sql file with the SQL commands to create the inventory database.

mysql/init.sql

```
CREATE DATABASE IF NOT EXISTS order_service;
CREATE DATABASE IF NOT EXISTS inventory_service;
```

Now let's configure the **application.properties** file with the relevant Spring Data JPA and Hibernate properties to interact with MySQL Database:

application.yml

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/inventory_service
spring.datasource.username=root
spring.datasource.password=mysql
spring.jpa.hibernate.ddl-auto=none
server.port=8082
```

We are using almost the same configuration as the Order Service, the only difference is we will be running the Inventory Service on port 8082.

Let's also create the Flyway migration scripts under the **src/main/resources/db/migration** folder, here we will be creating 2 scripts:

- V1__init.sql
- V2__add_inventory.sql

The V1__init.sql file as the name suggests, creates the t_inventory table

V1__init.sql

```
CREATE TABLE `t_inventory`
(
    `id`       bigint(20) NOT NULL AUTO_INCREMENT,
    `sku_code`  varchar(255) DEFAULT NULL,
    `quantity` int(11)      DEFAULT NULL,
    PRIMARY KEY (`id`)
);
```

V2__add_inventory.sql

```
insert into t_inventory (quantity, sku_code)
values (100, 'iphone_15'),
       (100, 'pixel_8'),
       (100, 'galaxy_24'),
       (100, 'oneplus_12');
```

Now let's go ahead and create the necessary code for implementing the Get Inventory endpoint.

Inventory.java

```
package com.programmingtechie.inventoryservice.model;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import jakarta.persistence.*;

@Entity
@Table(name = "t_inventory")
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class Inventory {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String skuCode;
    private Integer quantity;
}
```

InventoryRepository.java

```
package com.programmingtechie.inventoryservice.repository;

import com.programmingtechie.inventoryservice.model.Inventory;
import org.springframework.data.jpa.repository.JpaRepository;
```

```
public interface InventoryRepository extends JpaRepository<Inventory,
    boolean existsBySkuCodeAndQuantityIsGreaterThanEqual(String skuCo
}
```

InventoryService.java

```
package com.programmingtechie.inventoryservice.service;

import com.programmingtechie.inventoryservice.repository.InventoryRep
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
@RequiredArgsConstructor
public class InventoryService {

    private final InventoryRepository inventoryRepository;

    @Transactional(readOnly = true)
    public boolean isInStock(String skuCode, Integer quantity) {
        return inventoryRepository.existsBySkuCodeAndQuantityIsGreate
    }
}
```

InventoryController.java

```
package com.programmingtechie.inventoryservice.controller;

import com.programmingtechie.inventoryservice.service.InventoryServic
import lombok.RequiredArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/inventory")
@RequiredArgsConstructor
public class InventoryController {

    private final InventoryService inventoryService;

    @GetMapping
    @ResponseStatus(HttpStatus.OK)
    public boolean isInStock(@RequestParam String skuCode, @RequestPa
        return inventoryService.isInStock(skuCode, quantity);
    }
}
```

Now let's start the application by running the InventoryServiceApplication.class, and you should see the below logs, indicating that the database migrations are executed successfully.

```
Successfully applied 2 migrations to schema `inventory_service`, now
```

Testing using Postman

Now let's open Postman and call the http://localhost:8082/api/inventory?skuCode=iphone_15&quantity=100 endpoint, notice that we are passing multiple SKUCodes in the Request Params.


Testing Inventory Service through Postman

Writing Integration Tests

Let's write integration tests for the Inventory Service.

InventoryServiceApplicationTests.java

```
package com.programmingtechie.inventoryservice;

import com.jayway.jsonpath.JsonPath;
import io.restassured.RestAssured;
import org.hamcrest.Matchers;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.server.LocalServerPort;
import org.springframework.boot.testcontainers.service.connection.Ser
import org.testcontainers.containers.MySQLContainer;
```

```
import static org.hamcrest.Matchers.is;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertTrue;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM
class InventoryServiceApplicationTests {

    @ServiceConnection
    static MySQLContainer mySQLContainer = new MySQLContainer("mysql:
    @LocalServerPort
    private Integer port;

    @BeforeEach
    void setup() {
        RestAssured.baseURI = "http://localhost";
        RestAssured.port = port;
    }

    static {
        mySQLContainer.start();
    }

    @Test
    void shouldReadInventory() {
        var response = RestAssured.given()
                .when()
                .get("/api/inventory?skuCode=iphone_15&quantity=1")
                .then()
                .log().all()
                .statusCode(200)
                .extract().response().as(Boolean.class);
        assertTrue(response);

        var negativeResponse = RestAssured.given()
                .when()
```

```
                    .get("/api/inventory?skuCode=iphone_15&quantity=1000"
                    .then()
                    .log().all()
                    .statusCode(200)
                    .extract().response().as(Boolean.class);
            assertFalse(negativeResponse);

        }

    }
```

Conclusion

That's it for the first part of the **Spring Boot Microservices Tutorial** Series, we create 3 services for our application, and from the next part, we will be concentrating on applying the Microservice Design Patterns to our application.

In the next part, we will learn about Synchronous Inter-Service Communication Pattern using Spring Cloud OpenFeign. Until then, Happy Coding Techies!