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:

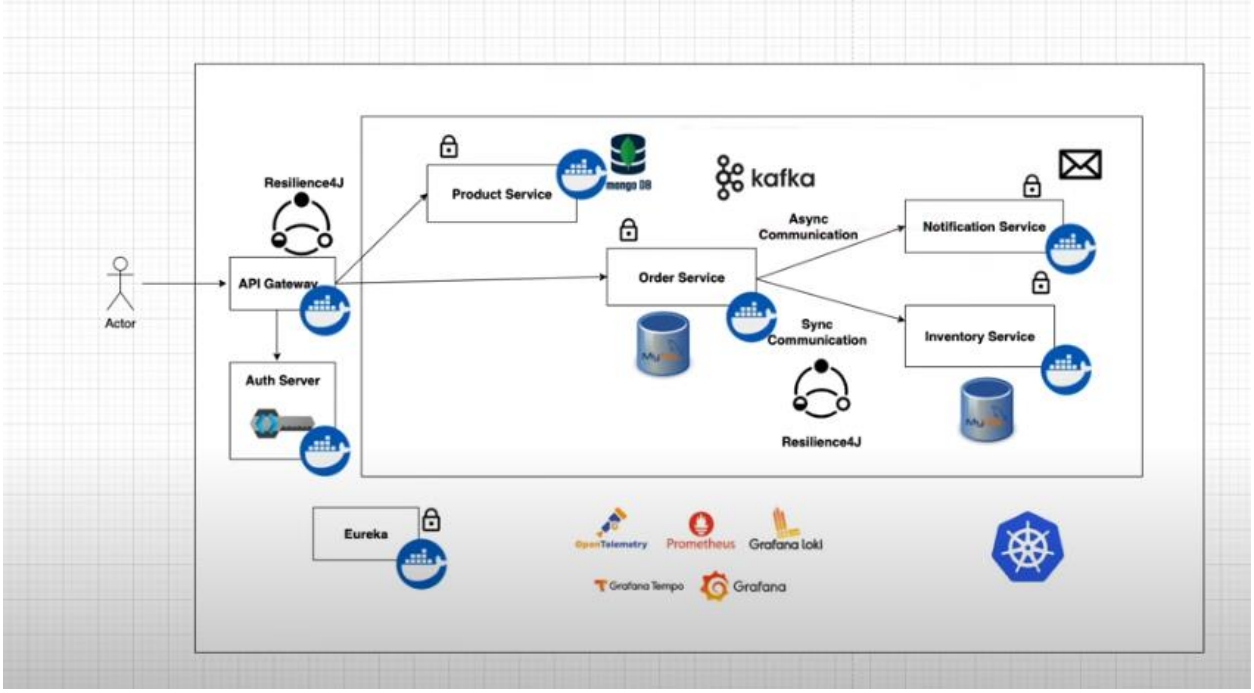
- Product Service
- Order Service

- 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.

Architecture Diagram of the Project

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



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:

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:
  mongodb:
    image: mongo:8.0.3
    container_name: mongodb
    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
```

version: '4' ***—> Specifies the Docker Compose file format version.***

services —> Defines the services (containers) to run. Here, there's a single service: mongo.

image: mongo:7.0.5 —> Specifies the Docker image for MongoDB, version `7.0.5`.

If this image isn't on your system, Docker will pull it from Docker Hub.

container_name: mongo —> Sets the name of the container to `mongo`

ports: "27017:27017" —> Maps the host machine's port `27017` to the container's port `27017`

environment —> Sets environment variables used to configure MongoDB during startup:

- `MONGO_INITDB_ROOT_USERNAME`: Username for the root user (`root`).
- `MONGO_INITDB_ROOT_PASSWORD`: Password for the root user (`password`).
- `MONGO_INITDB_DATABASE`: Creates an initial database named `product-service`.

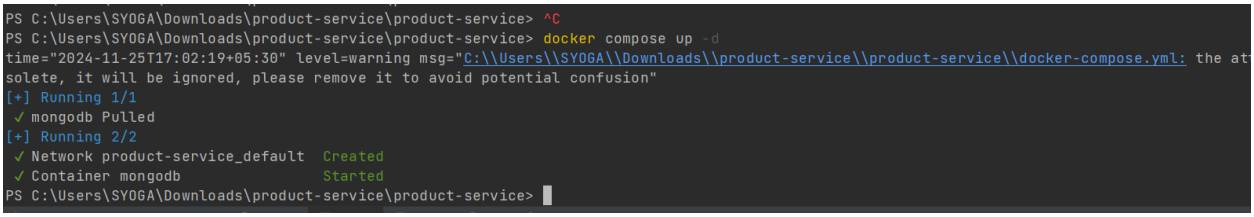
volumes —> Maps a directory on the host machine to a directory inside the container.

- `./docker/mongodb/data`: Path on the host machine.
- `/data/db`: Path inside the container where MongoDB stores its data.

- it will map what ever meta data stored inside /data/db inside our project on host machine(product-service).

docker compose up -d :

- Pull the MongoDB image (`mongo:8.0.3`) if it's not already downloaded.
- Create and start the `mongo` container as defined in the `docker-compose.yml`.
- Automatically create the `./docker/mongodb/data` directory on your host machine for persistent storage.



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

`spring.data.mongodb.uri=mongodb://root:password@localhost:27017/product-service?authSource=admin`

`mongodb://`:

- Specifies the protocol for connecting to MongoDB.

`root : password`:

- username : root
- password : password

`@localhost`:

- `localhost`: The **host** where the MongoDB server is running. `localhost` indicates that the MongoDB instance is running on the same machine as the application.
- **Alternative**: If MongoDB were hosted remotely, you would replace `localhost` with the IP address or hostname of the server.

`:27017`:

- **Port**: The port number (`27017`) where MongoDB listens for incoming connections. This is the default port for MongoDB.

`/product-service:`

- **Database Name**: The name of the **MongoDB database**.
- If the database doesn't exist, MongoDB will create it automatically when the first document is inserted.

`?authSource=admin:`

- `authSource`: Specifies the **database** that MongoDB uses to authenticate the user. In this case, the `admin` database is used for authentication.

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;
```

```
}
```

`@Document(value = "product"):`

- This annotation marks the class as a **MongoDB document.**
- `value = "product"` specifies the collection name in MongoDB. If not specified, MongoDB would default to using the class name (`Product`) as the collection name.

`@Id:`

Marks the `id` field as the **identifier** for the MongoDB document.

`@Builder:`

Here's how the builder works:

- `Product.builder()`:

This method returns a **builder instance** for the `Product` class.

- `.name("Product Name")`:

This method sets the `name` field in the builder.

- `.description("Product Description")`:

Sets the `description` field in the builder.

- `.price(new BigDecimal("99.99"))`:

Sets the `price` field in the builder.

- `.build()`:

After all the necessary fields are set, the `build()` method constructs and returns the `Product` object.

When to Use Setters vs. Builders

Scenario	Use Setters	Use Builders
Simple Objects	✅ Yes	❌ No
Complex Objects with Many Fields	❌ No	✅ Yes
Immutable Objects	❌ No	✅ Yes
Fluent API and Readability Required	❌ No	✅ Yes
Validation During Construction	❌ No	✅ Yes
Optional Fields	❌ Awkward	✅ Easy
Thread-Safe Construction	❌ No	✅ Yes

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, String> {
}
```

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.ProductRepository;
```

```
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
@RequiredArgsConstructor
@Slf4j
public class ProductService {

    private final ProductRepository productRepository;

    public ProductResponse 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");
        return new ProductResponse(product.getId(), product.getName(),
product.getDescription(), product.getPrice());
    }


    public List<ProductResponse> getAllProducts() {
        return productRepository.findAll().stream().map(Product -> new
ProductResponse(Product.getId(), Product.getName(), Product.getDescription(),
Product.getPrice()) ).toList();

    }
}
```

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 productRequest) {
        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.

`@ResponseStatus` is an annotation in Spring used to set the **HTTP response status code** for a specific controller method or exception handler.

`@RequiredArgsConstructor:` Automatically generates a constructor for **all** `final` fields and **fields annotated with** `@NonNull`.

ProductRequest.java

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

import java.math.BigDecimal;

public record ProductRequest(String name, String description, BigDecimal
price) {
}
```

A **record** is a special kind of class in Java designed to represent **immutable data** with minimal boilerplate. It's used when the primary purpose of a class is to store data without the need for explicit getters, setters, constructors, or `toString()`.

Fields in a `record` are implicitly `private` and `final`.

ProductResponse.java

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

import java.math.BigDecimal;

public record ProductResponse(String id, String name, String description,
BigDecimal price) {
}
```

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.

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.

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/

Test Containers:

Testcontainers is a **Java library** that helps you run **Docker containers** during testing. It is mainly used for **integration testing** when your application depends on external tools like databases or message brokers.

Instead of using fake or mocked services, you can test with real tools like PostgreSQL, MySQL, or Kafka.

Each test gets its own temporary, clean container, so tests don't affect each other.

Containers automatically stop after the test is done.

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. (like get, post, put... methods)

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.ServiceConnection;
import org.testcontainers.containers.MongoDBContainer;

import java.math.BigDecimal;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class ProductServiceApplicationTests {

    @ServiceConnection
    static MongoDBContainer mongoDBContainer = new
MongoDBContainer("mongo:7.0.7");
    @LocalServerPort
    private Integer port;

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

    static {
        mongoDBContainer.start();
    }

    @Test
    void shouldCreateProduct()  {
        String requestBody = """
                        {
                        "name":"MSI",
                        "description":"laptop 1TB SSD, 16GB RAM, 4GB
dedicated GP",
                        "price":"300000"
                        }""";

            RestAssured.given()
                        .contentType("application/json")
                        .body(requestBody)
                        .when()
                        .post("/api/product")
                        .then()
                        .statusCode(201)
                        .body("id", Matchers.notNullValue())
                        .body("name", Matchers.equalTo("MSI"))
                        .body("description", Matchers.equalTo("laptop 1TB
SSD, 16GB RAM, 4GB dedicated GP"))
                        .body("price", Matchers.equalTo(300000));
```

```
}
```

`@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)` —> starts the full springboot application in a random port for testing.

Why Random port? —> Avoids conflicts with other applications running on fixed ports.

`MongoDBContainer:` —> Creates and manages a temporary MongoDB instance using Docker.

`ServiceConnection:` —> Automatically connects this container to your Spring application as its MongoDB service.

`static { mongoDBContainer.start(); }`

Starts the MongoDB container before any tests run.

`@LocalServerPort private Integer port;`

- **Injects** the random port used by the Spring Boot application.
- Used to configure Rest-Assured to send requests to the correct port

`@BeforeEach void setup()`

- Sets up **Rest-Assured** before each test:
  - `RestAssured.baseURI`: The base URL for requests (`http://localhost`).
  - `RestAssured.port`: Configures the port to the one Spring Boot is using.

`@BeforeEach` —> The method annotated with `@BeforeEach` runs once before **each test method** in the class.

Why Before Each Test? —> Ensures that each test runs with the correct URI and port setup.

`given()`: Sets up the request.

- `contentType("application/json")`: Specifies that the request body is in JSON format.
- `.body(requestBody)`: Attaches the request body.

`when()`: Indicates the action to perform (HTTP POST to `/api/product`).

`body()`:

- Validates specific fields in the JSON response:
  - `"id"`: Ensures the server returns a non-null product ID.
  - `"name"`, `"description"`, `"price"`: Checks if the response matches the input values.

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.

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.

normally it maps the directories or files from host machine to the directories from container.

1. `- ./mysql/init.sql:/docker-entrypoint-initdb.d/init.sql`

This volume mapping does the following:

- **Host path**: `./mysql/init.sql` – This refers to a file named `init.sql` located in the `mysql` directory on the host machine. The `./` prefix means it is relative to the directory where the `docker-compose.yml` file is being executed.
- **Container path**: `/docker-entrypoint-initdb.d/init.sql` – This is the path inside the container. The `/docker-entrypoint-initdb.d/` directory is special for MySQL Docker images. Any `.sql` file placed here will be automatically executed when the container starts, initializing the MySQL database with the contents of the file.

2. `- ./docker/mysql/data:/var/lib/mysql`

This volume mapping does the following:

- **Host path**: `./docker/mysql/data` – This is a directory on the host machine (relative to the current directory).
- **Container path**: `/var/lib/mysql` – This is the default directory inside the container where MySQL stores its database files, logs, and other data.

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

- `spring.jpa.hibernate.ddl-auto=none` -→ If we give create it will destroy all old data and create new one. so we given none. **No changes** will be made to your database tables (like creating, updating, or validating). You are responsible for managing the database schema yourself, either by writing SQL scripts or using a tool like Flyway or Liquibase.
- 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

After running application, we will get unknown database error. so first we have to create database. we can install work bench and create. or we can do in our project like create new folder(docker) from root directory, and inside that again create one more folder(mysql), and inside mysql create init.sql file.

As we are using Intellij community edition, we can't handle sql from editor. we can use outside tool Mysql workbench.

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.(this folder present in our project only)

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.

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

```
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){

        Order order = new Order();
        order.setOrderNumber(UUID.randomUUID().toString());
        order.setPrice(orderRequest.price());
        order.setSkuCode(orderRequest.skuCode());
        order.setQuantity(orderRequest.quantity());
        orderRepository.save(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;
```

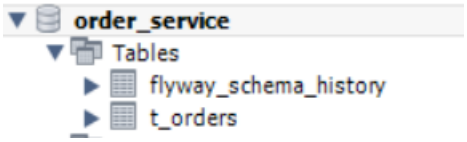
```
import org.springframework.web.bind.annotation.*;

@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, Integer
quantity) {
}
```

After running above Application, two tables will be created.

—>`flyway_schema_history`: it maintains all the history of the scripts that was executed.



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:

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.ServiceConnection;
import org.testcontainers.containers.MySQLContainer;

import static org.hamcrest.MatcherAssert.assertThat;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class OrderServiceApplicationTests {

    @ServiceConnection
    static MySQLContainer mySQLContainer = new MySQLContainer("mysql:8.3.0");
    @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
Successfully"));
    }
}
```

`.log().all()`: This logs the **full request and response** (including headers, body, etc.) for debugging purposes.

`.extract()`: This extracts the response from the API call.

`.body()`: This specifically refers to the **body** of the response.

`.asString()`: This converts the **body of the response** into a plain **String**.

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 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.properties:

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3307/inventory_service
spring.datasource.username=root
spring.datasource.password=password
```

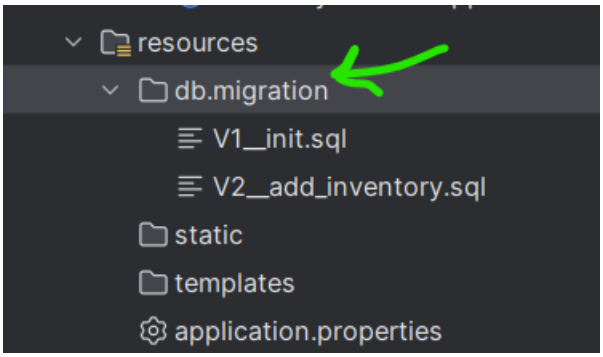
```
spring.jpa.hibernate.ddl-auto=none
server.port=8082
```

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.

docker-compose.yml:

```
version: '4'
services:
  mysql:
    image: mysql:8.0.40
    container_name: mysql
    ports:
      - "3307:3306"
    environment:
      MYSQL_ROOT_PASSWORD: mysql
    volumes:
      - ./mysql/init.sql:/docker-entrypoint-initdb.d/init.sql
      - ./docker/mysql/data:/var/lib/mysql
```

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, Long> {
    boolean existsBySkuCodeAndQuantityIsGreaterThanEqual(String skuCode, int
quantity);
}
```

InventoryService.java

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

import com.programmingtechie.inventoryservice.repository.InventoryRepository;
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.existsBySkuCodeAndQuantityIsGreaterThanEqual(skuCode,
quantity);
    }
}
```

existsBySkuCodeAndQuanitityIsGreaterThanEqual —> this method we dont need to give implementation. JPA repository only will create this method and provide necessary logic for that method.

InventoryController.java

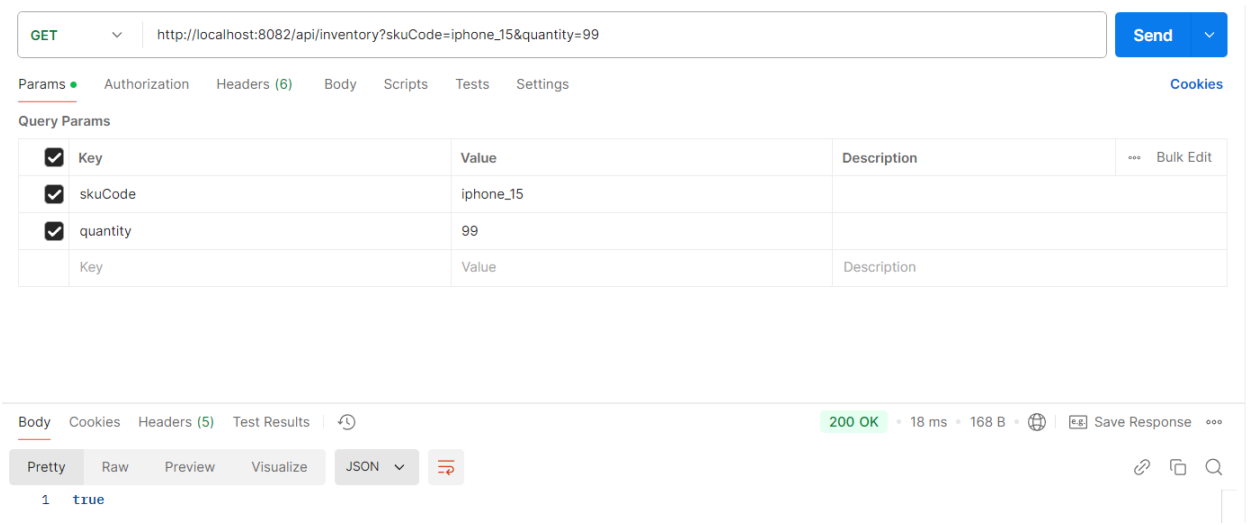
```
package com.programmingtechie.inventoryservice.controller;

import com.programmingtechie.inventoryservice.service.InventoryService;
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, @RequestParam Integer
quantity) {
        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
at version v2 (execution time 00:00.033s)
```

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.



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.ServiceConnection;
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_PORT)
class InventoryServiceApplicationTests {

    @ServiceConnection
    static MySQLContainer mySQLContainer = new MySQLContainer("mysql:8.3.0");
```

```
    @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);

    }
}
```

Request:

- This sends a **GET** request to the endpoint `/api/inventory?skuCode=iphone_15&quantity=1`.
- This request is to check the availability of 1 unit of the product with the SKU code `iphone_15`.

Assertions and Validations:

- The `.then()` method is used to perform assertions after the request is made.
- `.log().all()` logs the entire response (including headers, body, status code, etc.) for inspection.
- `.statusCode(200)` asserts that the response status code is **200**, indicating a successful request.
- `.extract().response().as(Boolean.class)` extracts the response body and converts it to a `Boolean` value. This assumes that the API returns a `true` or `false` boolean value.

Assertion:

- `assertTrue(response)` checks that the response is `true`. If the response is `true`, the test passes, indicating that the requested quantity (1) of the product is available.

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.