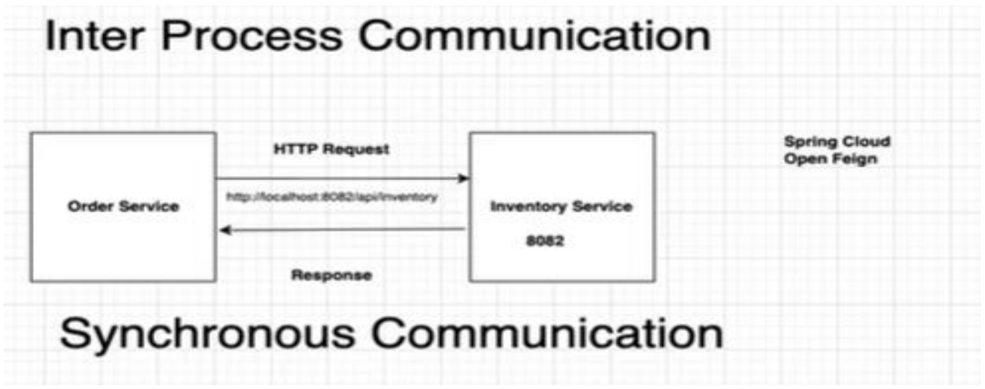
Spring Boot Microservices Tutorial - Part 2

In Part 2 of this **Spring Boot Microservices Tutorial** series, we will implement Synchronous Communication between our Order Service and Inventory Service using Spring Cloud OpenFeign Library.

Spring Cloud OpenFeign library uses that provides OpenFeign integrations with Spring Boot and Spring Cloud. It provides a declarative REST Client that makes consuming REST Endpoints in our code easy.


Inter Process Communication
Order Service
HTTP Request
http://localhost:8082/api/inventory
Response
Inventory Service
8082
Spring Cloud
Open Feign
Synchronous Communication

Synchronous Communication:

synchronized in time. In this mode, the sender waits for the receiver to acknowledge or respond before proceeding with the next action. Both parties must be actively involved at the same time, meaning the sender cannot move on until the receiver responds.

Example: A phone call or a real-time chat where both parties are engaged in the conversation at the same time.

We will implement Synchronous Communication between Order Service and Inventory Service using the Spring Cloud OpenFeign library.

Asynchronous Communication:

The sender sends a message and doesn't wait for an immediate response. The receiver can respond at a later time, allowing both parties to communicate independently without requiring them to be active at the same time.

Example: Email, text messages, or API calls where the sender doesn't need to wait for a reply immediately.

Add Spring Cloud OpenFeign to Order Service

To get started, let's add the Spring Cloud OpenFeign Starter to the pom.xml file of the Order Service.

pom.xml

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

We also need to add the spring-cloud-dependencies pom dependency to the `<dependencyManagement>` section in the pom.xml file.

```xml
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

This is how your pom.xml should look like at the end:

Pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```xml
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>microservices-new</artifactId>
        <groupId>com.programming.techie</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>order-service</artifactId>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.flywaydb</groupId>
            <artifactId>flyway-core</artifactId>
        </dependency>
        <dependency>
            <groupId>org.flywaydb</groupId>
            <artifactId>flyway-mysql</artifactId>
        </dependency>
        <dependency>
            <groupId>com.mysql</groupId>
            <artifactId>mysql-connector-j</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-openfeign</artifactId>
        </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>
        <dependency>
```

```xml
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-testcontainers</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.testcontainers</groupId>
            <artifactId>junit-jupiter</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.testcontainers</groupId>
            <artifactId>mysql</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>io.rest-assured</groupId>
            <artifactId>rest-assured</artifactId>
            <version>5.3.2</version>
        </dependency>
    </dependencies>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
</project>
```

Create FeignClient for Inventory Service

As we will be calling Inventory Service from Order Service, we need to create a interface called InventoryClient.java inside the client package inside the order-service.

client/InventoryClient.java

```java
package com.programmingtechie.orderservice.client;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
```

```java
@FeignClient(value = "inventory", url = "${inventory.url}")
public interface InventoryClient {
    @RequestMapping(method = RequestMethod.GET, value = "/api/inventory")
    boolean isInStock(@RequestParam String skuCode, @RequestParam Integer quantity);
}
```

Notice that the @FeignClient annotation has an attribute called URL that is pointing to the inventory.url property in the application.properties file

```
inventory.url=http://localhost:8082
```

By externalizing this property we can replace it dynamically in tests or during startup time.

Coming to the method, we have the @RequestMapping annotation that is calling the path - /api/inventory.

Now we have to call the **isInStock()** method from the placeOrder() method of the Order Service.

If the client returns true, then we will place the order and save it to the database successfully, or else, we will throw a Runtime Exception

Here's how the OrderService class looks like with the final logic.

OrderService.java

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

import com.programmingtechie.orderservice.client.InventoryClient;
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;
    private final InventoryClient inventoryClient;

    public void placeOrder(OrderRequest orderRequest){

        if(inventoryClient.isInStock(orderRequest.skuCode(), orderRequest.quantity())) {
            Order order = new Order();
            order.setOrderNumber(UUID.randomUUID().toString());
            order.setPrice(orderRequest.price());
            order.setSkuCode(orderRequest.skuCode());
            order.setQuantity(orderRequest.quantity());
            orderRepository.save(order);
        }
        else{
            throw  new RuntimeException("product with skuCode "+orderRequest.skuCode() +"
not in stock");
        }
    }
}
```

Before we go ahead and test our implementation, we have to add the @EnableFeignClients annotation to enable Feign Client Capabilities

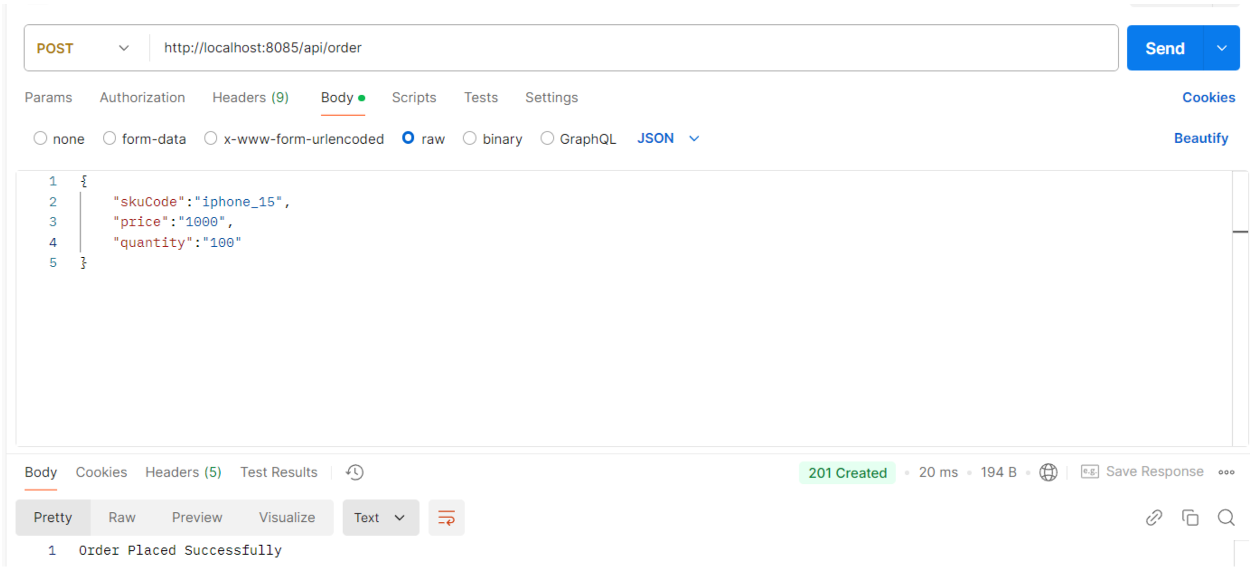
OrderServiceApplication.java

```
package com.programmingtechie.orderservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.openfeign.EnableFeignClients;

@SpringBootApplication
@EnableFeignClients
public class OrderServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(OrderServiceApplication.class, args);
    }

}
```

Manual Testing using Postman

Now it's time to test our implementation using Postman, make sure you start both the Order Service as well as the Inventory Service and call the Place Order Endpoint of Order Service.

Let's order the skuCode iphone_15, with a quantity of 100, as in Part -1 we initialized all skuCodes with quantity 100, this product should be in stock, and our Order should go through.



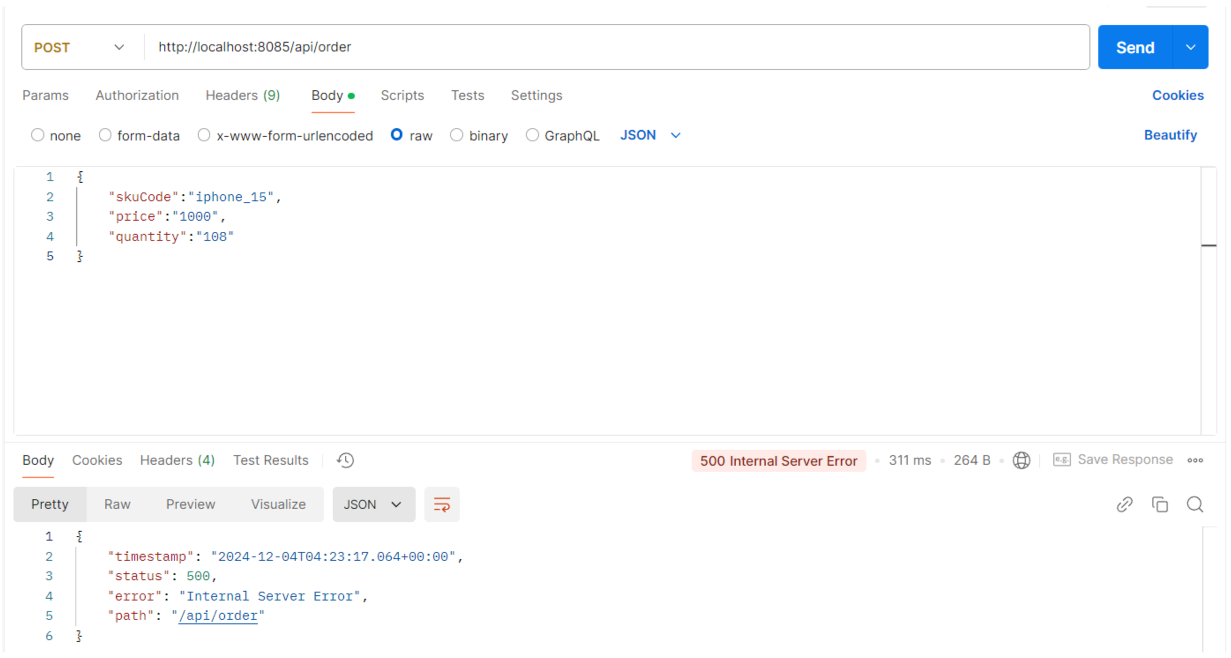
http://localhost:8085/api/order

```
{
    "skuCode":"iphone_15",
    "price":"1000",
    "quantity":"100"
}
```

201 Created • 20 ms • 194 B

```
Order Placed Successfully
```

Now let's change the quantity to 101, and this time our Order call should fail with a 500 error.


POST
Send
Params Authorization Headers (9) Body Scripts Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Beautify
Body Cookies Headers (4) Test Results
500 Internal Server Error 311 ms 264 B
Save Response
Pretty Raw Preview Visualize JSON

```
http://localhost:8085/api/order
```

```
{
    "skuCode":"iphone_15",
    "price":"1000",
    "quantity":"108"
}
```

```
{
    "timestamp": "2024-12-04T04:23:17.064+00:00",
    "status": 500,
    "error": "Internal Server Error",
    "path": "/api/order"
}
```

If you observe logs, then you should see the below exception message:

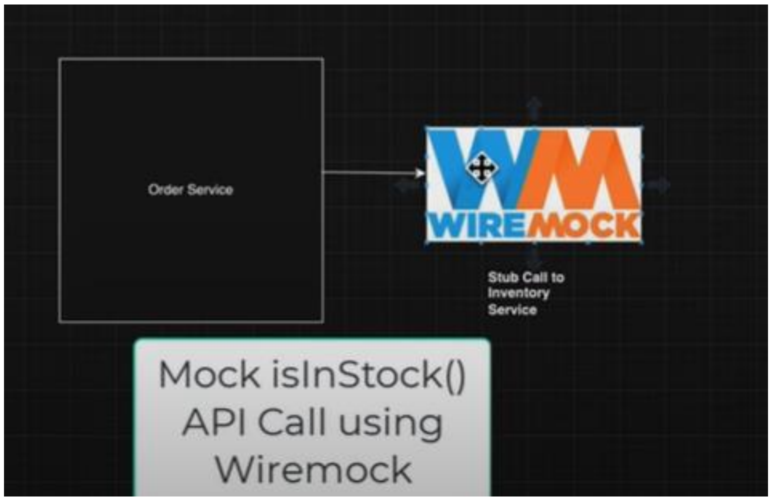
```
java.lang.RuntimeException: Product with Skucode iphone_15is not in stock
```

Updating the Integration Tests

Now if you run our Integration Tests in the order service, you will notice that they no longer run successfully as we are calling the Inventory Service.

To make these test successful, we have to use a library called Wiremock that provides a mock server environment to test our Order Service by making some mock HTTP calls.

By using Wiremock, we can verify if our Order Service is calling the inventory service with correct URL Params/Request Body/ Path Variables or not. We can also stub the response and test how our service is responding for various scenarios.


Order Service
WIREMOCK
Stub Call to Inventory Service
Mock isInStock() API Call using Wiremock

To enable wiremock, we need to add the following dependency to our pom.xml file of Order Service

pom.xml

```xml
    <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
    <scope>test</scope>
</dependency>
```

Here's how the update Integration Test looks like:

OrderServiceApplicationTests.java

```java
package com.programmingtechie.orderservice;

import com.programmingtechie.orderservice.stub.InventoryStubs;
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.springframework.cloud.contract.wiremock.AutoConfigureWireMock;
import org.testcontainers.containers.MySQLContainer;

import static org.hamcrest.MatcherAssert.assertThat;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
```

```java
@AutoConfigureWireMock(port = 0)
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
                }
                """;

        InventoryStubs.stubInventoryCall("iphone_15", 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"));
    }
}
```

@AutoConfigureWireMock(port = 0)→ Annotation for test classes that want to start a WireMock server as part of the Spring Application Context.

port=0 means springboot will pick the random port and assign that port to wiremock.

InventoryStubs.stubInventoryCall("iphone_15", 1);

- This line is calling the `stubInventoryCall` method from `InventoryStubs` class that you defined earlier.
- It sets up a mock response for the `/api/inventory?skuCode=iphone_15&quantity=1` API call.
- WireMock will now return a **200 OK** status with a body `"true"` when the API `/api/inventory?skuCode=iphone_15&quantity=1` is called, simulating that the product `iphone_15` with quantity `1` is available in stock.

application.properties

```
inventory.url=http://localhost:${wiremock.server.port}
```

InventoryStubs.java

```
package com.programmingtechie.orderservice.stub;

import lombok.experimental.UtilityClass;

import static com.github.tomakehurst.wiremock.client.WireMock.*;

public class InventoryStubs {

    public void stubInventoryCall(String skuCode, Integer quantity) {
        stubFor(get(urlEqualTo("/api/inventory?skuCode=" + skuCode + "&quantity=" +
quantity))
                .willReturn(aResponse()
                        .withStatus(200)
                        .withHeader("Content-Type", "application/json")
                        .withBody("true")));
    }
}
```

- The `stubInventoryCall` method mocks the response of an external inventory service.
- `stubFor(get(...))`: This method defines the HTTP request that WireMock should mock. Here, it simulates a GET request to the URL `/api/inventory` with query parameters `skuCode` and `quantity`.
    - `urlEqualTo("/api/inventory?skuCode=" + skuCode + "&quantity=" + quantity)`: This specifies that the request URL must exactly match `/api/inventory?skuCode=<skuCode>&quantity=<quantity>`.
- `.willReturn(aResponse()...)`: This defines what WireMock should respond with when the above request is received. The response:
    - **Status**: `200` (indicating a successful HTTP response).
    - **Header:** `Content-Type: application/json` (to specify that the response body is JSON).
    - **Body**: `"true"` (the content of the response, in this case, a string representing `true`, indicating that the product is in stock).