**Microservice Architecture Overview**

We’ll develop three independent microservices that communicate with each other:

1. **Product Service**: Manages product data.
2. **Order Service**: Manages customer orders.
3. **Inventory Service**: Manages stock levels for products.

Each service will expose REST APIs for communication, use an embedded H2 database for simplicity, and will register with Eureka for service discovery.

**1. Product Service**

This service will handle CRUD operations for products in the catalog.

**Product Service API Endpoints:**

* **POST /products**: Add a new product.
* **GET /products/{id}**: Get details of a product by ID.
* **GET /products**: List all products.
* **PUT /products/{id}**: Update a product by ID.
* **DELETE /products/{id}**: Delete a product by ID.

**Spring Boot Code Example:**

java

Copy code

@RestController

@RequestMapping("/products")

public class ProductController {

@Autowired

private ProductService productService;

@PostMapping

public ResponseEntity<Product> createProduct(@RequestBody Product product) {

return new ResponseEntity<>(productService.createProduct(product), HttpStatus.CREATED);

}

@GetMapping("/{id}")

public ResponseEntity<Product> getProductById(@PathVariable Long id) {

return new ResponseEntity<>(productService.getProductById(id), HttpStatus.OK);

}

@GetMapping

public ResponseEntity<List<Product>> getAllProducts() {

return new ResponseEntity<>(productService.getAllProducts(), HttpStatus.OK);

}

@PutMapping("/{id}")

public ResponseEntity<Product> updateProduct(@PathVariable Long id, @RequestBody Product product) {

return new ResponseEntity<>(productService.updateProduct(id, product), HttpStatus.OK);

}

@DeleteMapping("/{id}")

public ResponseEntity<Void> deleteProduct(@PathVariable Long id) {

productService.deleteProduct(id);

return new ResponseEntity<>(HttpStatus.NO\_CONTENT);

}

}

**2. Order Service**

This service manages customer orders. It interacts with both the **Product Service** and **Inventory Service** to validate if the product exists and is in stock.

**Order Service API Endpoints:**

* **POST /orders**: Place a new order.
* **GET /orders/{id}**: Retrieve an order by ID.

**Spring Boot Code Example:**

java

Copy code

@RestController

@RequestMapping("/orders")

public class OrderController {

@Autowired

private OrderService orderService;

@PostMapping

public ResponseEntity<Order> createOrder(@RequestBody OrderRequest orderRequest) {

return new ResponseEntity<>(orderService.createOrder(orderRequest), HttpStatus.CREATED);

}

@GetMapping("/{id}")

public ResponseEntity<Order> getOrderById(@PathVariable Long id) {

return new ResponseEntity<>(orderService.getOrderById(id), HttpStatus.OK);

}

}

**OrderService Interaction Logic:**

* Check with the **Product Service** if the requested product exists.
* Check with the **Inventory Service** if the product is available in stock.
* If valid, create and return the order.

**3. Inventory Service**

This service manages the stock levels for products.

**Inventory Service API Endpoints:**

* **GET /inventory/{productId}**: Check stock availability for a specific product.

**Spring Boot Code Example:**

java

Copy code

@RestController

@RequestMapping("/inventory")

public class InventoryController {

@Autowired

private InventoryService inventoryService;

@GetMapping("/{productId}")

public ResponseEntity<Inventory> getStockByProductId(@PathVariable Long productId) {

return new ResponseEntity<>(inventoryService.getStockByProductId(productId), HttpStatus.OK);

}

}

**4. Service Discovery (Eureka Server)**

We will use **Eureka** to enable service discovery, allowing the microservices to register themselves and discover others.

**Eureka Server Setup:**

In the **Eureka Server** application (application.yml):

yaml

Copy code

server:

port: 8761

eureka:

client:

register-with-eureka: false

fetch-registry: false

Each microservice will include the Eureka client dependency and configure it to register with Eureka:

yaml

Copy code

eureka:

client:

service-url:

defaultZone: http://localhost:8761/eureka/

**5. Communication Between Services**

Microservices communicate via REST. For example, when placing an order, the **Order Service** can call the **Product Service** and **Inventory Service** as follows:

**RestTemplate Example in OrderService:**

java

Copy code

@Autowired

private RestTemplate restTemplate;

public boolean checkProductExists(Long productId) {

String productUrl = "http://PRODUCT-SERVICE/products/" + productId;

ResponseEntity<Product> response = restTemplate.getForEntity(productUrl, Product.class);

return response.getStatusCode() == HttpStatus.OK;

}

public boolean checkInventory(Long productId) {

String inventoryUrl = "http://INVENTORY-SERVICE/inventory/" + productId;

ResponseEntity<Inventory> response = restTemplate.getForEntity(inventoryUrl, Inventory.class);

return response.getBody().getAvailableStock() > 0;

}

**6. Testing the Microservice Architecture**

Once the microservices are up and running:

1. Register **Product Service**, **Order Service**, and **Inventory Service** with Eureka.
2. Use a tool like **Postman** to send requests to each service.
3. Ensure that orders can only be placed if the product exists and is in stock.

**Conclusion**

This case study demonstrates how you can develop a microservice architecture using Spring Boot. By breaking down functionalities into services like **Product**, **Order**, and **Inventory**, each service becomes independent, maintainable, and scalable. Using **Eureka**, services can discover each other and communicate through REST endpoints.