Microservices & API Integrations

Amo Samuel

This project demonstrates a microservices architecture utilizing gRPC, Docker, and Spring Boot, with PostgreSQL for persistence. It consists of four core microservices:

- **Gateway Service**: Routes incoming API requests.
- **Product Service**: Manages product information and communicates with other services via gRPC.
- **Order Service**: Handles product ordering using gRPC for service communication.
- **Shipping Service**: Retrieves order details via Feign clients and manages shipping.

A key part of this project is how the services communicate using gRPC, and how .proto files are used to define the communication contract between services.

gRPC and Proto Files

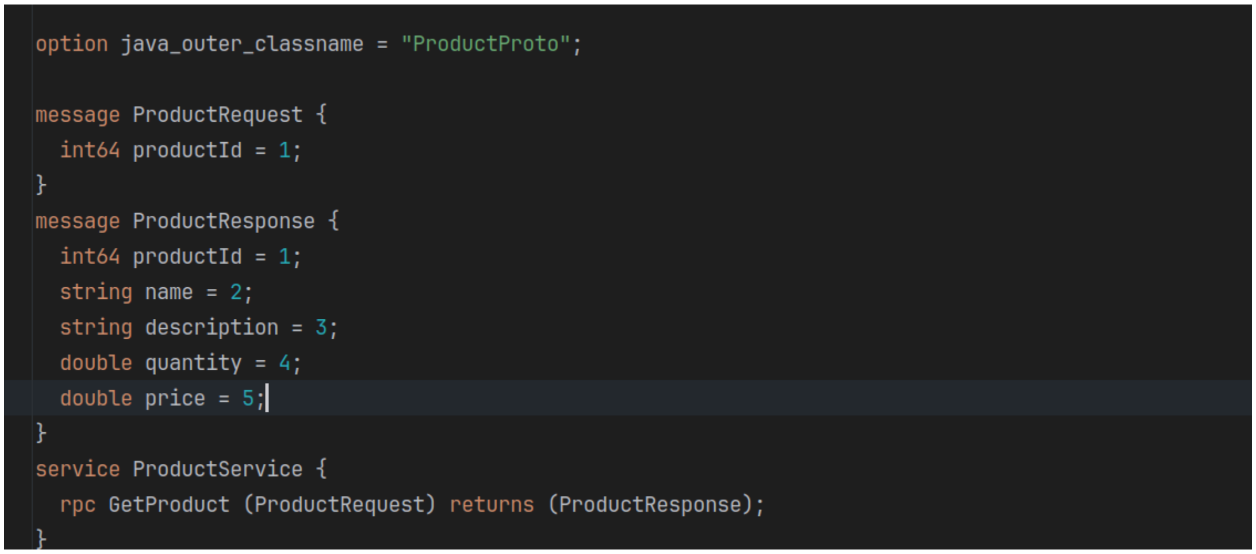
gRPC is used for inter-service communication between **Order Service** and **Product Service**. To define the communication contract, I used **Protocol Buffers (protobuf)**. These .proto files describe the services and messages for gRPC communication, and from these, code is auto-generated to handle serialization, deserialization, and service stubs.

Steps to Generate Proto Files

1. **Defined the .proto file**: The .proto file specifies the service methods and message types.
2. **Compiled the proto file** using the protoc compiler, which generates the Java gRPC code used by the microservices.
3. **Used the generated stubs** in the Product Service and Order Service for inter-service communication.

Proto File Example

Let's look at an example .proto file that defines a product service.



```
option java_outer_classname = "ProductProto";

message ProductRequest {
  int64 productId = 1;
}
message ProductResponse {
  int64 productId = 1;
  string name = 2;
  string description = 3;
  double quantity = 4;
  double price = 5;
}
service ProductService {
  rpc GetProduct (ProductRequest) returns (ProductResponse);
}
```

In this file:

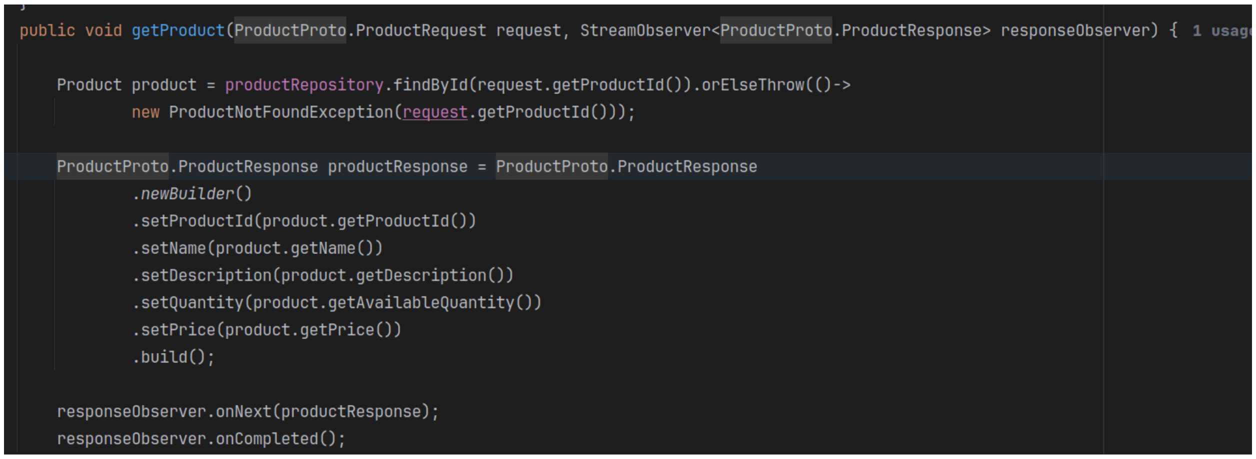
- ProductService is a gRPC service that exposes the method GetProduct.
- ProductRequest is the message sent by the client with a product_id.
- ProductResponse is the message returned by the server, containing product details such as name, description, quantity, and price

When you run the build (e.g., mvn clean install), this plugin generates the necessary Java classes for the gRPC service and message types. These generated classes include:

- ProductServiceGrpc: The base class for gRPC client and server.
- ProductProto.ProductRequest and ProductProto.ProductResponse: Java classes for the request and response message types.

Product Service Implementation (gRPC Server)

Once the proto files were compiled, I implemented the service in the Product Service by extending ProductServiceGrpc.ProductServiceImplBase. Here's how the ProductService is implemented:

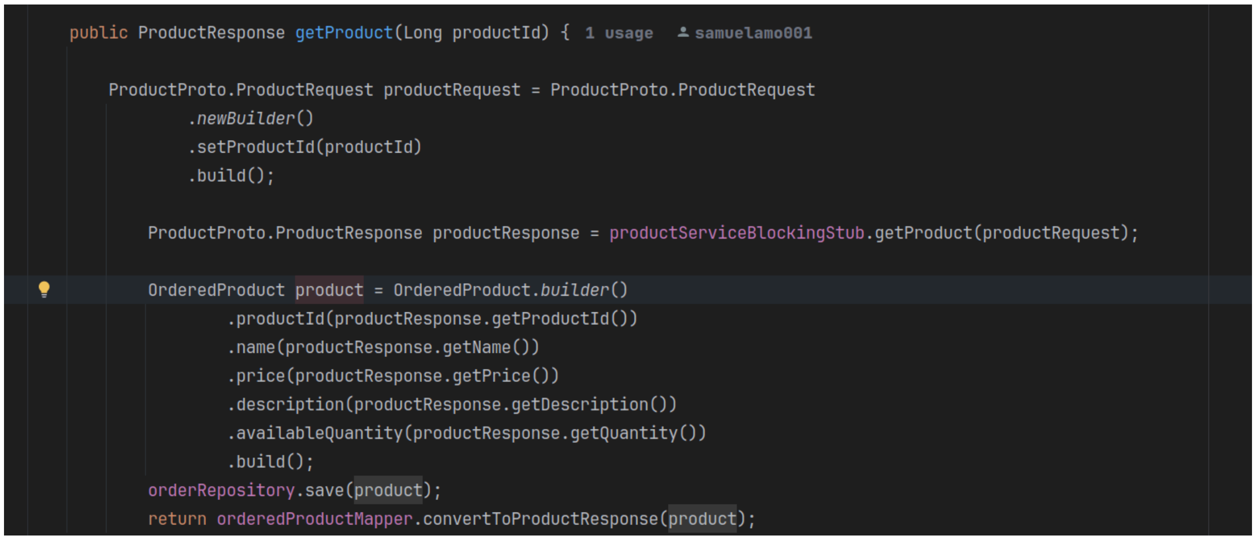


```java
public void getProduct(ProductProto.ProductRequest request, StreamObserver<ProductProto.ProductResponse> responseObserver) {  1 usag

    Product product = productRepository.findById(request.getProductId()).orElseThrow(()->
            new ProductNotFoundException(request.getProductId()));

    ProductProto.ProductResponse productResponse = ProductProto.ProductResponse
            .newBuilder()
            .setProductId(product.getProductId())
            .setName(product.getName())
            .setDescription(product.getDescription())
            .setQuantity(product.getAvailableQuantity())
            .setPrice(product.getPrice())
            .build();

    responseObserver.onNext(productResponse);
    responseObserver.onCompleted();
```

The getProduct method receives a ProductRequest, queries the database for the product, and returns a ProductResponse

Order Service Implementation (gRPC Client)

In the Order Service, I used the generated gRPC client stubs to call the Product Service. Here's how the Order Service calls the GetProduct method from the Product Service:



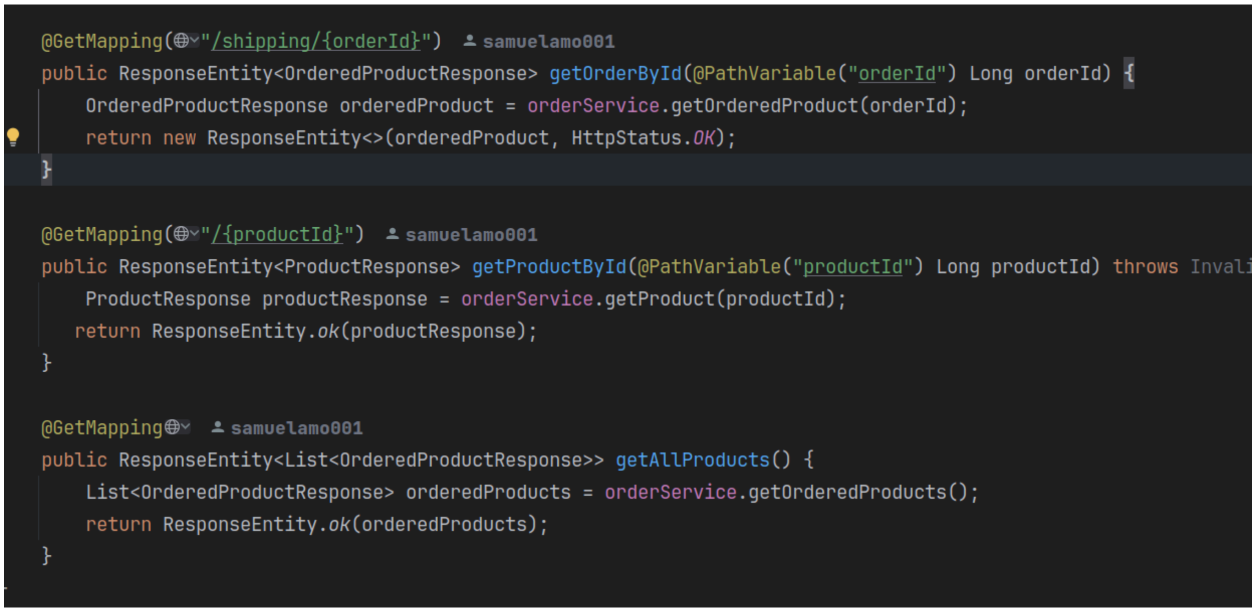
```java
public ProductResponse getProduct(Long productId) {  1 usage  samuelamo001

    ProductProto.ProductRequest productRequest = ProductProto.ProductRequest
            .newBuilder()
            .setProductId(productId)
            .build();

        ProductProto.ProductResponse productResponse = productServiceBlockingStub.getProduct(productRequest);

        OrderedProduct product = OrderedProduct.builder()
                .productId(productResponse.getProductId())
                .name(productResponse.getName())
                .price(productResponse.getPrice())
                .description(productResponse.getDescription())
                .availableQuantity(productResponse.getQuantity())
                .build();
        orderRepository.save(product);
        return orderedProductMapper.convertToProductResponse(product);
```

ProductServiceGrpc.ProductServiceBlockingStub: This is the client stub auto-generated from the proto file.

getProduct method sends a ProductRequest to the Product Service using gRPC and handles the response.

OrderController in the Order Microservice

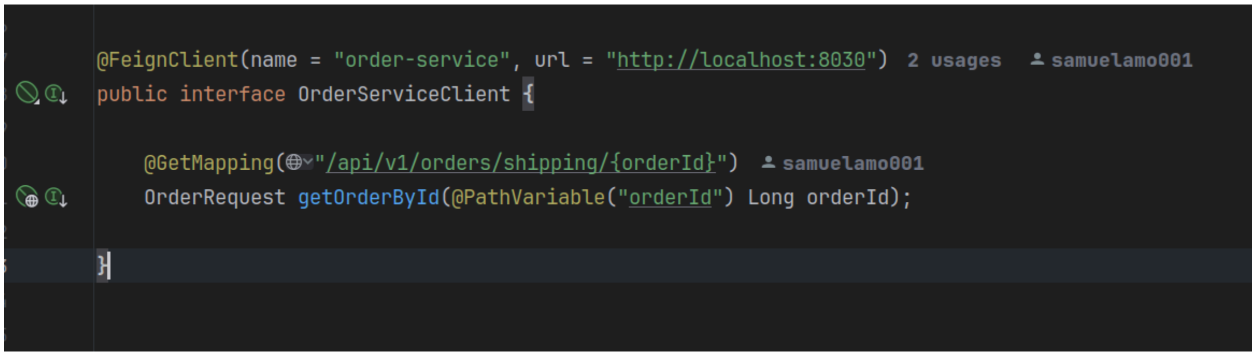
The OrderController is the main entry point for the **Order Service** that handles incoming HTTP requests. It is responsible for exposing REST endpoints that allow clients to perform operations related to orders and products. The controller interacts with the OrderService class to retrieve or manipulate the necessary data. Here's a brief explanation of each endpoint:



```
@GetMapping("/shipping/{orderId}")  samuelamo001
public ResponseEntity<OrderedProductResponse> getOrderById(@PathVariable("orderId") Long orderId) {
    OrderedProductResponse orderedProduct = orderService.getOrderedProduct(orderId);
    return new ResponseEntity<>(orderedProduct, HttpStatus.OK);
}

@GetMapping("/{productId}")  samuelamo001
public ResponseEntity<ProductResponse> getProductById(@PathVariable("productId") Long productId) throws Invali
    ProductResponse productResponse = orderService.getProduct(productId);
   return ResponseEntity.ok(productResponse);
}

@GetMapping  samuelamo001
public ResponseEntity<List<OrderedProductResponse>> getAllProducts() {
    List<OrderedProductResponse> orderedProducts = orderService.getOrderedProducts();
    return ResponseEntity.ok(orderedProducts);
}
```

Shipping Service

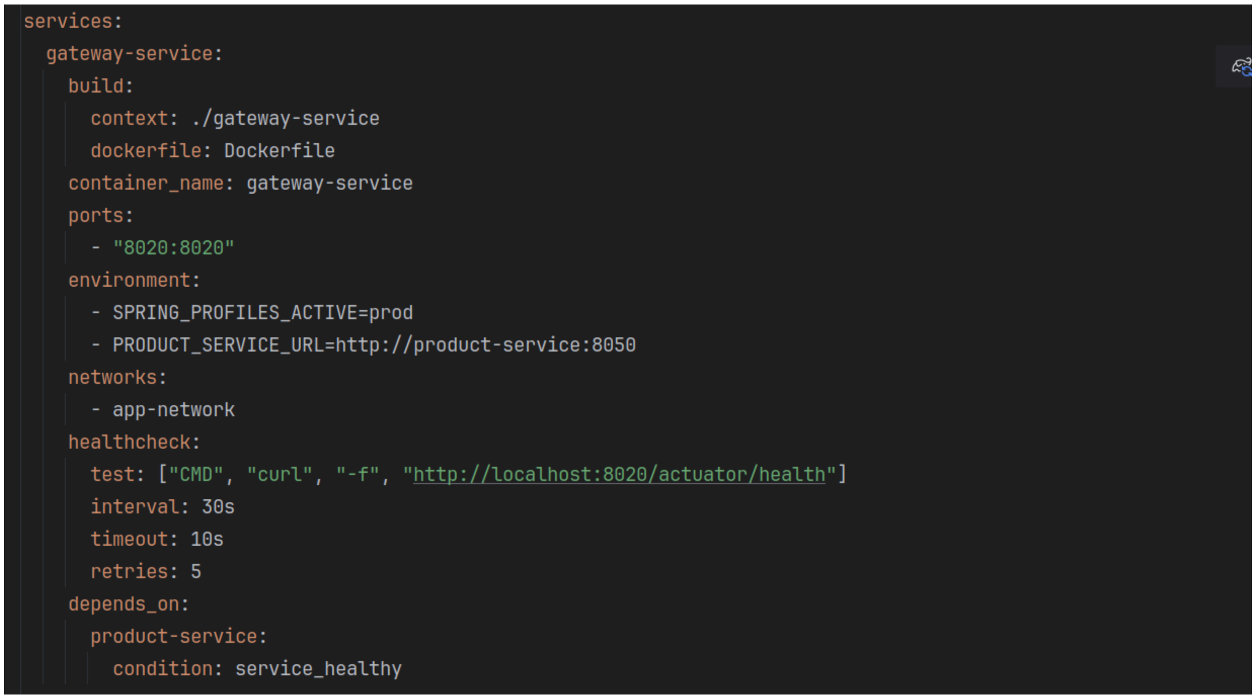
The Shipping Service retrieves order details from the Order Service using Feign clients. Below is a snippet showcasing this interaction.



```
@FeignClient(name = "order-service", url = "http://localhost:8030")  2 usages  samuelamo001
public interface OrderServiceClient {

    @GetMapping("/api/v1/orders/shipping/{orderId}")  samuelamo001
    OrderRequest getOrderById(@PathVariable("orderId") Long orderId);

}
```

Dockerization

Each microservice is containerized using Docker. The following is a part of the docker-compose.yml for deploying these services.

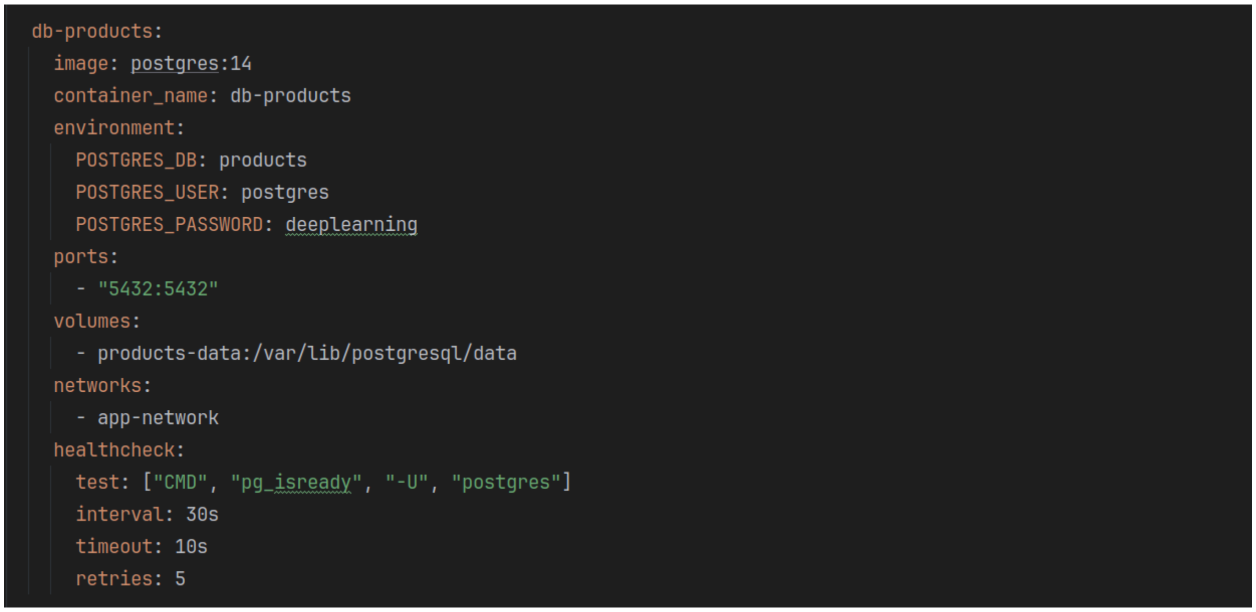


```yaml
services:
  gateway-service:
    build:
      context: ./gateway-service
      dockerfile: Dockerfile
    container_name: gateway-service
    ports:
      - "8020:8020"
    environment:
      - SPRING_PROFILES_ACTIVE=prod
      - PRODUCT_SERVICE_URL=http://product-service:8050
    networks:
      - app-network
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8020/actuator/health"]
      interval: 30s
      timeout: 10s
      retries: 5
    depends_on:
      product-service:
        condition: service_healthy
```

This configuration sets up four containers: gateway-service, product-service, order-service, and shipping-service. All services are connected via the app-network bridge.

Databases

Each microservice has a dedicated PostgreSQL database. The example below demonstrates the configuration for the Order Service.



```yaml
db-products:
  image: postgres:14
  container_name: db-products
  environment:
    POSTGRES_DB: products
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: deeplearning
  ports:
    - "5432:5432"
  volumes:
    - products-data:/var/lib/postgresql/data
  networks:
    - app-network
  healthcheck:
    test: ["CMD", "pg_isready", "-U", "postgres"]
    interval: 30s
    timeout: 10s
    retries: 5
```

Conclusion

This project demonstrates a microservice architecture with multiple services communicating via gRPC, Feign clients, and RESTful APIs. Each service is isolated in its container and manages its own data via PostgreSQL. The API Gateway handles the routing, and health checks ensure service reliability. The solution provides a scalable, resilient architecture that is ready for deployment.