Microservices Architecture

Table of Contents

1. Microservices and API-Driven Development

Microservices architecture is a method of developing applications as a suite of small services, each running independently and communicating over lightweight mechanisms like HTTP or messaging queues. These services are organized around business capabilities and can be developed, deployed, and scaled independently.

API-driven development ensures that services communicate through well-defined APIs (Application Programming Interfaces). This approach provides flexibility and allows each service to evolve independently.

2. Implementing RESTful APIs for Microservices

RESTful APIs (Representational State Transfer) are a popular architectural style for creating web services. They use HTTP methods like GET, POST, PUT, and DELETE to enable communication between microservices and clients.
In Java, the Spring Boot framework simplifies the development of RESTful APIs by providing pre-configured setup and reducing boilerplate code. Below are code snippets of the Application class as well as the controller class of the Product Service Restful API.

Sample Code: Java Spring Boot REST API

```java
package microservice.product;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication
@EnableDiscoveryClient
public class ProductApplication {

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

}


package microservice.product.controllers;

import microservice.product.dto.ProductDto;
import microservice.product.services.ProductService;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/products")
public class ProductController {
    private ProductService productService;
    public ProductController(ProductService productService){
        this.productService=productService;
    }

    @GetMapping("getProduct/{id}")
    public ResponseEntity<ProductDto> getProduct(@PathVariable long
id){
        return ResponseEntity.ok(productService.getProductById(id));
    }

    @PostMapping("addProduct/")
    public ResponseEntity<ProductDto> addProduct(@RequestBody
ProductDto productDto){
        return
ResponseEntity.ok(productService.addProduct(productDto));
    }
}
```

3. API Gateway for Managing API Traffic and Routing

An API Gateway is a vital component in microservices architecture, providing a single entry point for all clients. It helps in managing, routing, and securing API requests across multiple microservices. This improves performance, simplifies client interactions, and helps manage cross-cutting concerns like authentication, logging, and rate limiting.

In Java, Spring Cloud Gateway can be used to implement an API Gateway. It is a library that routes requests and provides features such as load balancing, security, and monitoring. Below are code snippets of the application class as well as the gateway routing configuration class.

Sample Code: Spring Cloud Gateway

```
package microservice.apigateway;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication
@EnableDiscoveryClient
public class ApiGatewayApplication {

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


package microservice.apigateway.config;
import org.springframework.cloud.gateway.route.RouteLocator;
import
org.springframework.cloud.gateway.route.builder.RouteLocatorBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class GatewayConfig {

    @Bean
    public RouteLocator gatewayRoutes(RouteLocatorBuilder builder) {
        return builder.routes()
                // Route for the product service
                .route("product", r -> r.path("/products/**")  //
Matches all paths starting with /products/
                        .uri("lb://PRODUCT")) // Uses load balancer to
route to PRODUCT-SERVICE
                // Route for the order service
                .route("order", r -> r.path("/orders/**")  // Matches
all paths starting with /orders/
                        .uri("lb://ORDER")) // Uses load balancer to
route to ORDER-SERVICE
                .build();
    }
}
```

4. gRPC for Efficient Inter-Service Communication

gRPC is a modern, open-source RPC (Remote Procedure Call) framework that allows services to communicate efficiently. It uses Protocol Buffers for serialization and supports multiple programming languages, making it a great choice for inter-service communication in a microservices architecture.

gRPC is faster and more efficient than REST when it comes to binary data, making it ideal for high-performance microservices.

Below are code snippets of the proto file of my product service as well as the grpc service implementation for fetching response to the order service upon request.

Sample Code: gRPC Service Definition (Proto)

```
syntax = "proto3";
```

```
//option java_multiple_files = true;

option java_package = "microservice.proto.grpc";

option java_outer_classname = "ProductProto";

message Product{
  int64 id = 1;
  string productName = 2;
  string productDescription = 3;
  int32 productPrice = 4;
  int32  productQuantity = 5;
}

service ProductService{
  rpc GetProductById(ProductRequest) returns (ProductResponse);
}

message ProductRequest{
  int64 id = 1;
}

message ProductResponse{
  Product product = 1;
}
```

Sample Code: Java gRPC Server

```
package microservice.product.services;

import io.grpc.Status;
import io.grpc.stub.StreamObserver;
import microservice.product.ProductRepository.ProductRepository;
import microservice.product.entities.ProductEntity;
import microservice.proto.grpc.ProductProto;
import microservice.proto.grpc.ProductServiceGrpc;
import net.devh.boot.grpc.server.service.GrpcService;
import org.springframework.stereotype.Service;

@GrpcService
public class ProductServiceGRPC extends
ProductServiceGrpc.ProductServiceImplBase {
    private final ProductRepository productRepository;

    public ProductServiceGRPC(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    @Override
    public void getProductById(ProductProto.ProductRequest request,
StreamObserver<ProductProto.ProductResponse> responseObserver) {
        try {
            long productId = request.getId();
```

```java
            ProductEntity productEntity =
productRepository.findById(productId)
                    .orElseThrow(() -> new RuntimeException("Product
with ID " + productId + " doesn't exist"));

            ProductProto.Product product =
ProductProto.Product.newBuilder()
                    .setId(productEntity.getId())
                    .setProductName(productEntity.getProductName())

.setProductDescription(productEntity.getProductDescription())
                    .setProductPrice(productEntity.getProductPrice())

.setProductQuantity(productEntity.getProductQuantity())
                    .build();

            ProductProto.ProductResponse response =
ProductProto.ProductResponse.newBuilder()
                    .setProduct(product)
                    .build();

            responseObserver.onNext(response);
            responseObserver.onCompleted();
        } catch (Exception e) {
            responseObserver.onError(Status.INTERNAL
                    .withDescription("Error retrieving product: " +
e.getMessage())
                    .asRuntimeException());
        }
    }
}
```