# Microservices with Java Spring Boot and Spring Cloud

# C M Abdullah Khan

Senior Software Engineer
BJIT Ltd

# Agenda

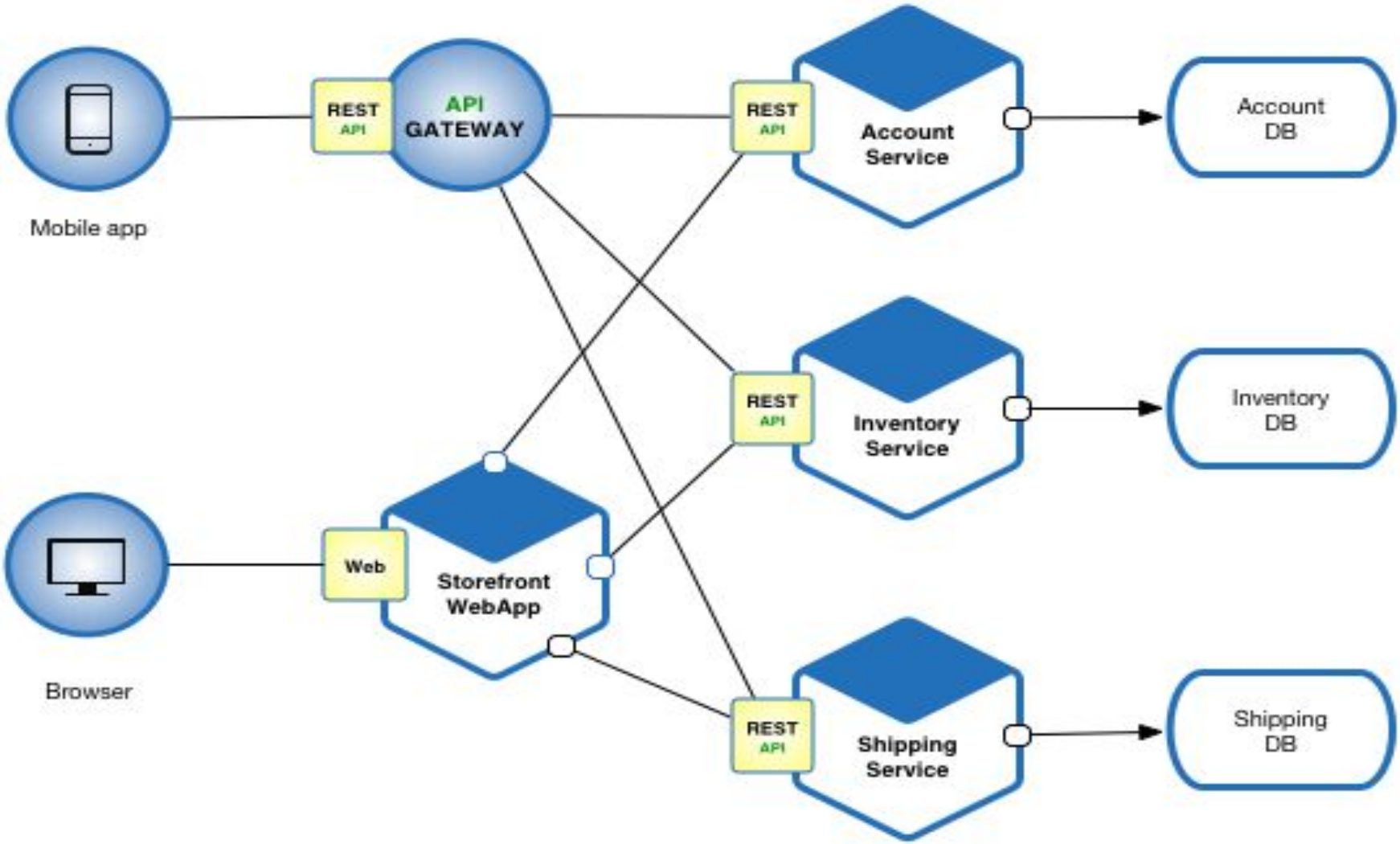
- MicroServices Fundamental
- Monolith vs N-Layer MicroService
- Microservice Architecture
- Microservice Advantage/Limitations and when to use!
- Framework and tools
- Intra service communications

# Day 1

a. Introduction to Microservice Architecture

b. Implementation of MicroServices & service discovery

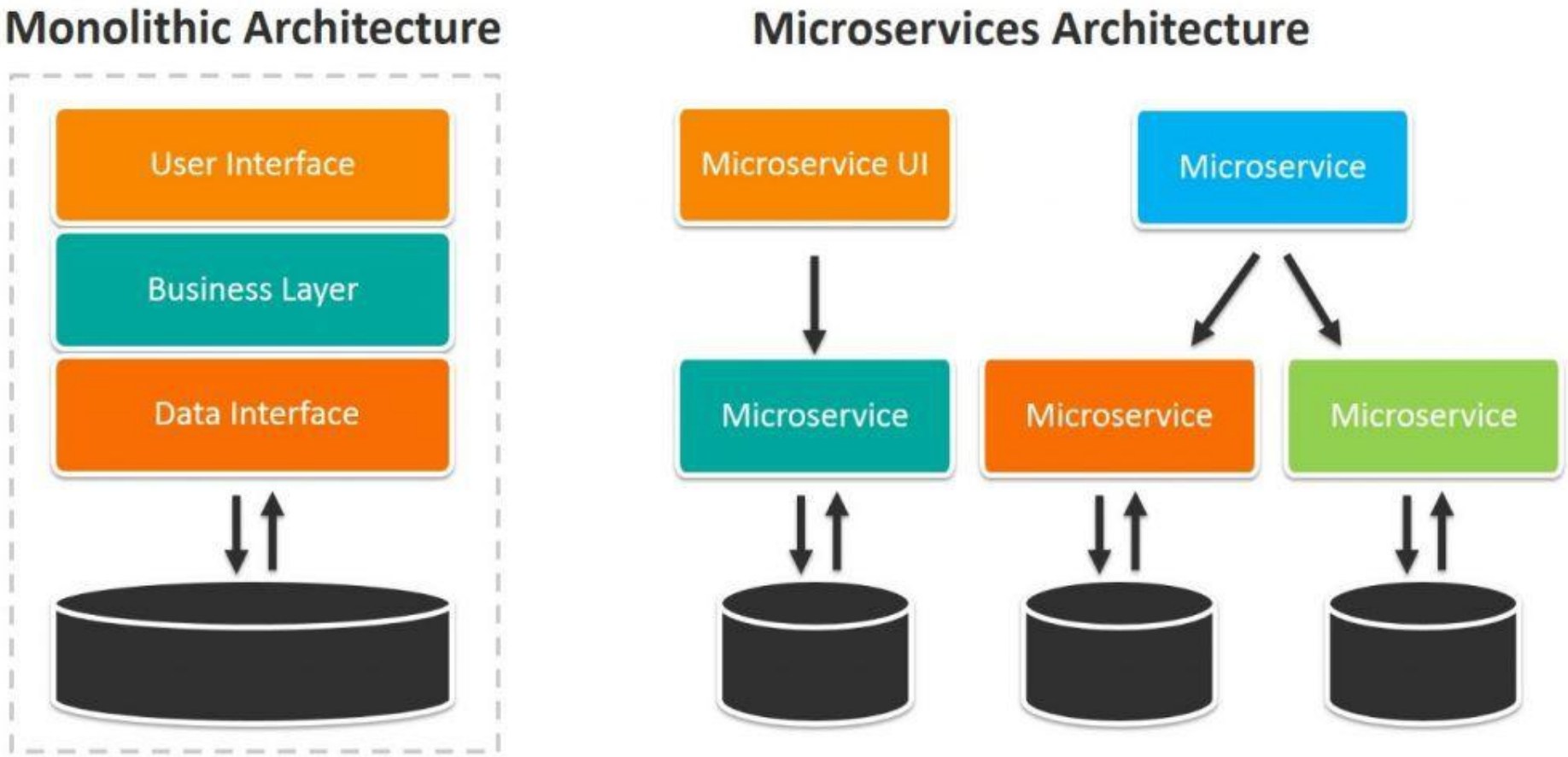
# MicroServices Fundamental



# MicroServices Fundamental

- Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are
  I. Loosely coupled multiple services
  II. Independently deployable
  III. Organized around business capabilities
  IV. Owned by a small team
  V. Highly maintainable and testable
  VI. Ex: Rakuten, Amazon

# Monolith vs N-Layer MicroService



# Monolith vs N-Layer MicroService

- The monolithic architecture pattern has been the architectural style used in the past, pre-Kubernetes and cloud services days.
- In a monolithic architecture, the software is a single application distributed in a single place
- Characteristics of a monolithic architecture:
  - Changes are slow
  - Changes are costly
  - Hard to adapt to a specific, or changing, product line
  - Monolithic structures make changes to the application extremely slow. Modifying just a small section of code can require a completely rebuilt and deployed version of software.

# Use case

1. per product application can serve 250 queries per second

4 core cpu, 1gb ram with 3 gb heap memory

2. per order application can serve 250 queries per second

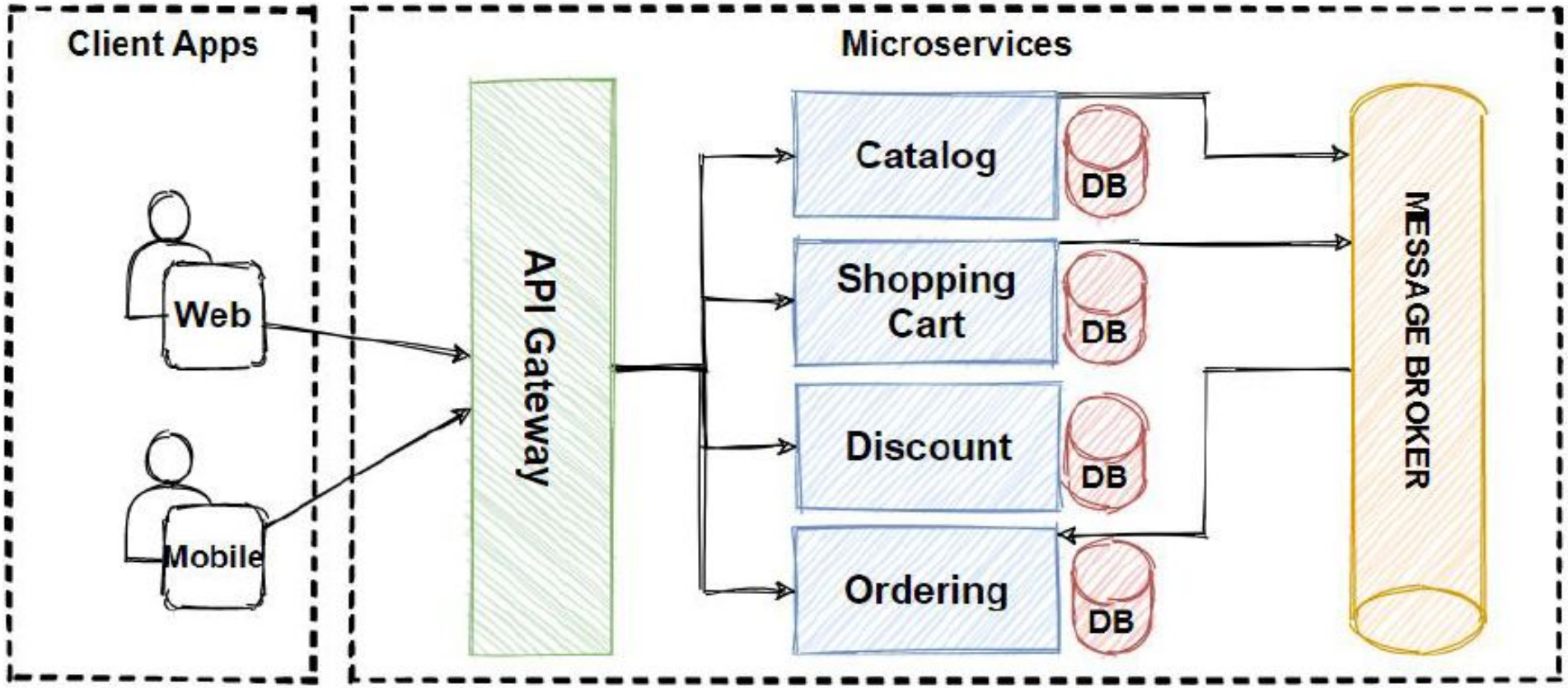
4 core cpu, 1gb ram with 6 gb heap memory

/product/list-> 900M -> 10,416.66 queries/S
/order/submit -> 270M

system accepts 30% of orders corresponding to products view

**count the number of instances.**
**count the number of instances at the peak time.**

# Microservice Architecture



# Advantages to Microservices

- Applications built as a set of independent modular components.
- easier to test, maintain, and understand.
- Decrease the amount of time to build
- Developer independence
- Isolation & resilience
- Scalability

# Disadvantages of microservices

- Each microservice's size
- Optimal boundaries and connection points between microservices
- The right framework to integrate services
- More broadly, microservices have these drawbacks:
  - Increased complexity
  - More expensive
  - Greater security risks

# Framework and tools

- Spring Boot with Spring Cloud
- Data Management — Spring have various modules to easily integrate with popular databases. Spring JDBC, **Spring JPA**.
- Observability — Spring Boot Actuator are powerful, it provides **health check**, view logs, **Metrix**.
- Security — Spring Security, good support for OAuth2, Session management, possible to build stateful and stateless services.

- Distributed config management — **Spring Cloud Config**

- Service Discovery — **Client Side**, **Server Side**

- Performance — **Catching support**, **Load Balancing**, **Clustering with Spring Cloud Cluster**
- Communication Data Format — **JSON**, XML

- Testing — Spring Testing module, **Mocking**, Profiling

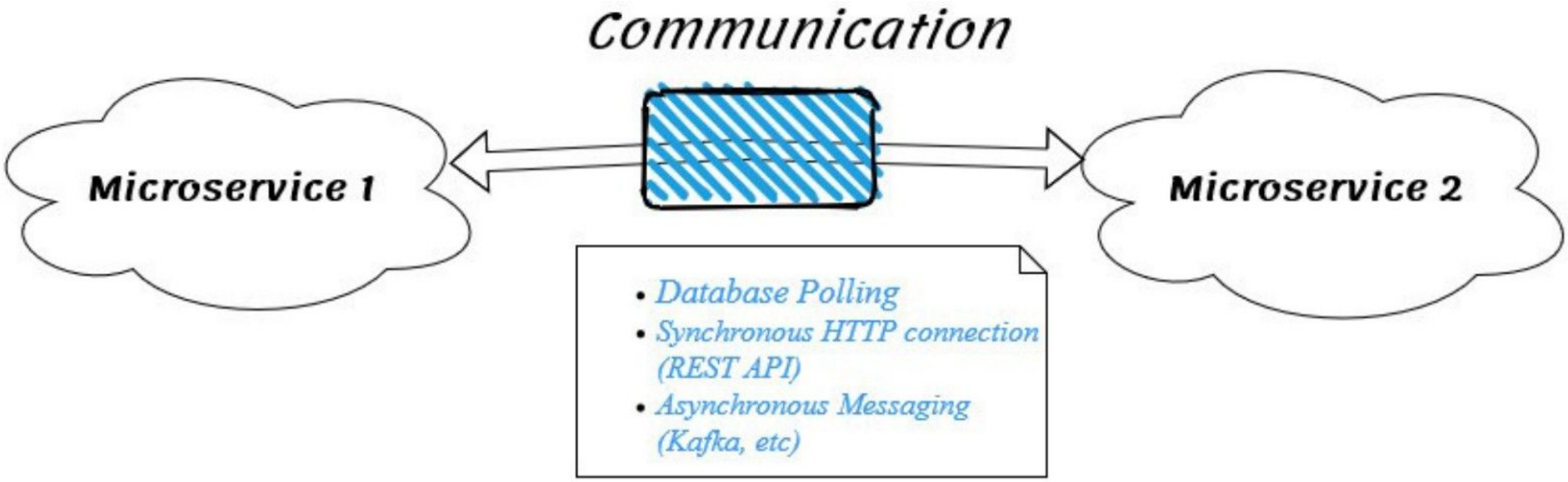
# Framework and tools

- Communication Style — Spring have capabilities to **build reactive app**. It is super easy to build rest API with the help of Spring MVC. Spring is easy to integrate with **MQ to build asynchronous style services**.

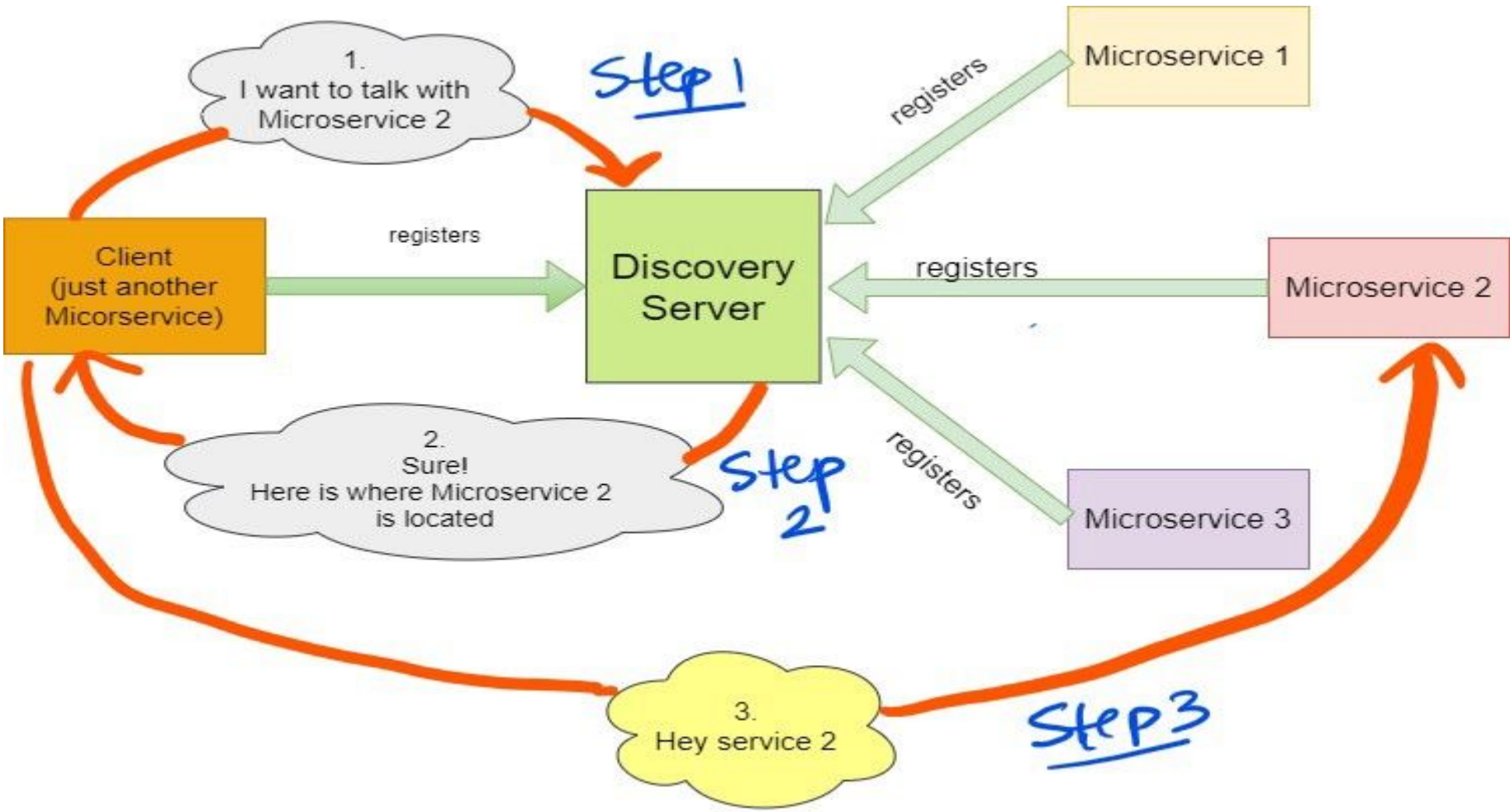
- Middle-tier Integration — With the help of various module Spring provide abstraction to integrate with **Apache Kafka**, RabbitMQ

- Integration with Tools —**Prometheus**, **Grafana**

# Orders to Products service communication



# Service Discovery



# @EnableEurekaServer

```
package phoenix;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class NetflixEurekaNamingServerApplication {

  public static void main(String[] args) {

SpringApplication.run(NetflixEurekaNamingServerApplication.class,
args);
  }

}
```

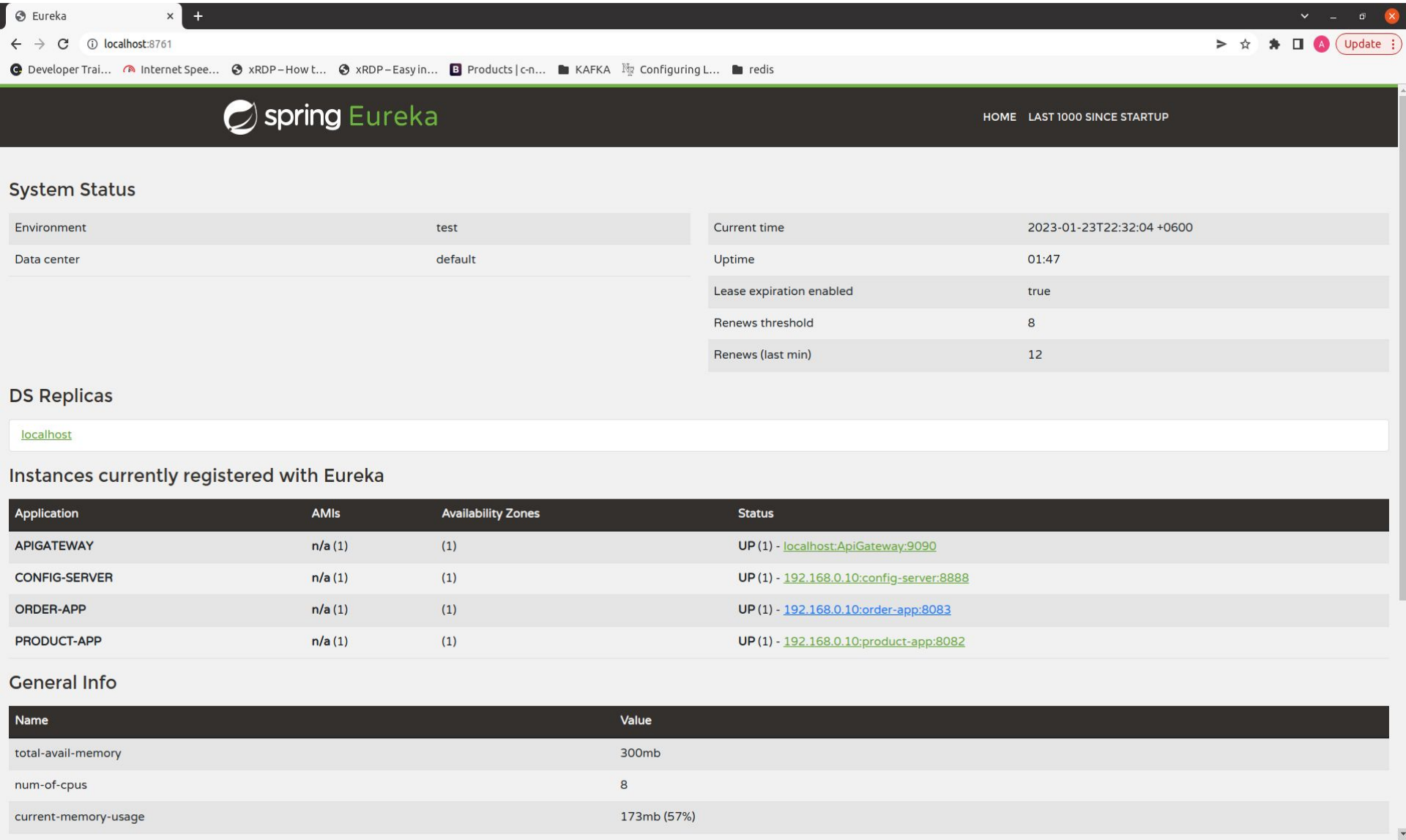
# spring-cloud-starter-netflix-eureka-server

```
dependencies {

    //actuator
    implementation
'org.springframework.boot:spring-boot-starter-actuator'

    //eureka-server
    implementation
'org.springframework.cloud:spring-cloud-starter-netflix-eureka-serv
er'

    //lombok
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation
'org.springframework.boot:spring-boot-starter-test'
}
```

# Eureka Server Configuration

```
spring.application.name=EurekaNamingServer
server.port=8761

eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
#application.properties
```

# Eureka Dashboard



# Client-side service discovery

Client-side service discovery allows services to find and communicate with each other without hard-coding the hostname and port. The only ‘fixed point' in such an architecture is the service registry, with which each service has to register.

# Day2

a. Introduction of Spring Cloud config & API Gateway

b. Implementation of Spring Cloud config & API Gateway

c. Assignment on a hands-on task

# Eureka Client Config

***need to register with eureka server***

```
eureka.client.service-url.default-zone=http://localhost:8761/eureka
```

***need to add dependency as config client***

```
implementation
'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'
```

# Spring Cloud Config

Spring Cloud Config provides server-side and client-side support for externalized configuration in a distributed system. With the Config Server, you have a central place to manage external properties for applications across all environments.

# @EnableConfigServer

```
package phoenix;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {

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

# Set the repository Path

```
spring.application.name=config-server
#spring.cloud.config.server.git.uri=file:///home/bjit/Documents/ECSG/MicroserviceEssentials/configRepo
spring.cloud.config.server.git.uri=file:///home/abdullah/Documents/workspace/MicroserviceEssentials/configRepo
server.port=8888
eureka.client.service-url.default-zone=http://localhost:8761/eureka
#set this config in application.properties
```

# Need to add config server dependency

```
dependencies {

    //actuator
    implementation
'org.springframework.boot:spring-boot-starter-actuator'

    //eureka-client
    implementation
'org.springframework.cloud:spring-cloud-starter-netflix-eureka
-client'

    //config server
    implementation
'org.springframework.cloud:spring-cloud-config-server'

    testImplementation
'org.springframework.boot:spring-boot-starter-test'
}
```

# Config Server Url End-point

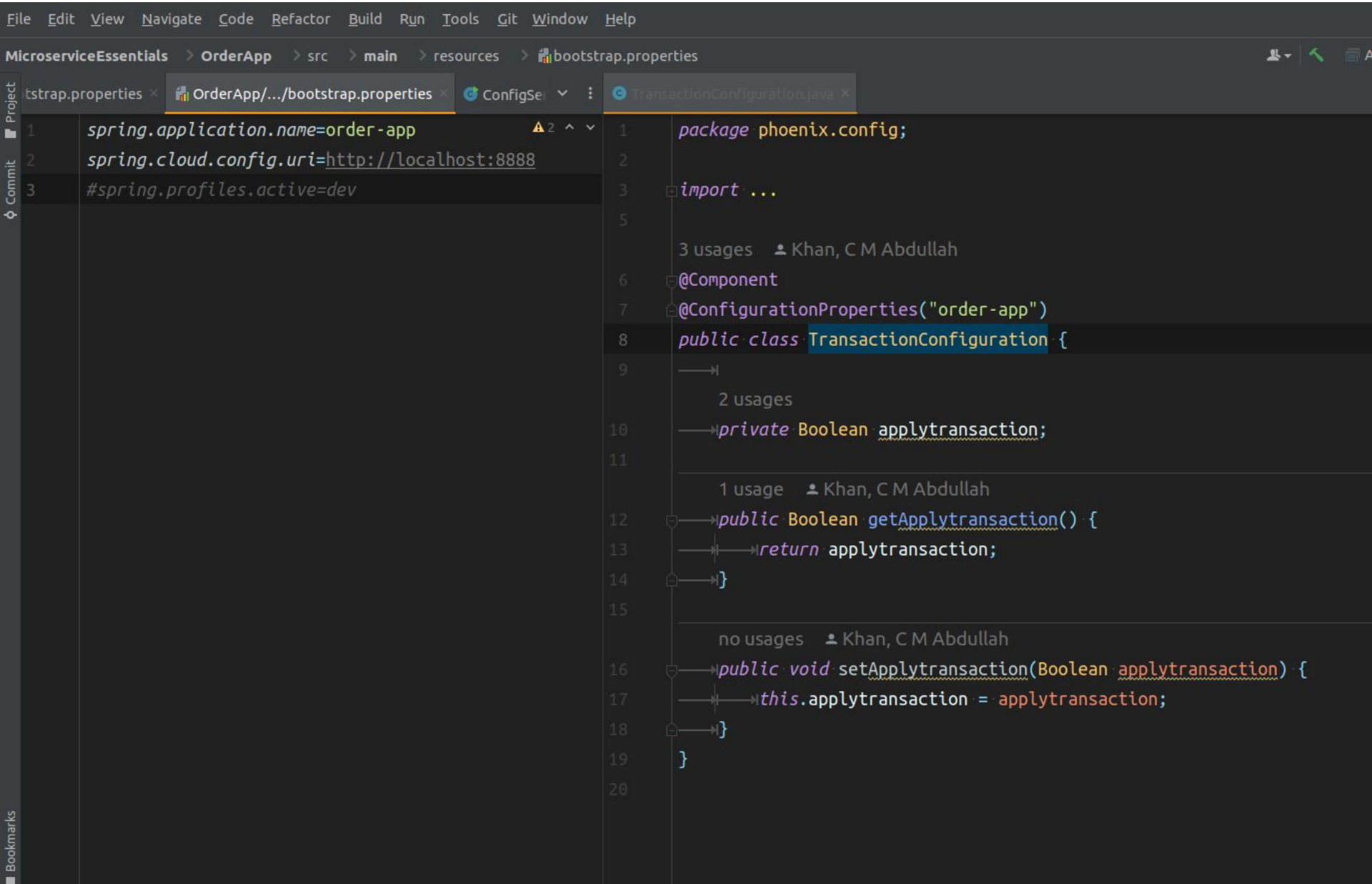
- curl --location --request GET 'http://localhost:8888/order-app/default'
- curl --location --request GET 'http://localhost:8888/config-server/default'

# Read this properties from config Client

order-app.properties

- `order-app.applytransaction=true`

**Need to make sure application name and config name must be same**


File Edit View Navigate Code Refactor Build Run Tools Git Window Help
MicroserviceEssentials > OrderApp > src > main > resources > bootstrap.properties
tstrap.properties
OrderApp/.../bootstrap.properties
ConfigSe
TransactionConfiguration.java
spring.application.name=order-app
spring.cloud.config.uri=http://localhost:8888
#spring.profiles.active=dev
package phoenix.config;
import ...
3 usages Khan, C M Abdullah
@Component
@ConfigurationProperties("order-app")
public class TransactionConfiguration {
2 usages
private Boolean applytransaction;
1 usage Khan, C M Abdullah
public Boolean getApplytransaction() {
return applytransaction;
}
no usages Khan, C M Abdullah
public void setApplytransaction(Boolean applytransaction) {
this.applytransaction = applytransaction;
}
}
Project
Commit
Bookmarks

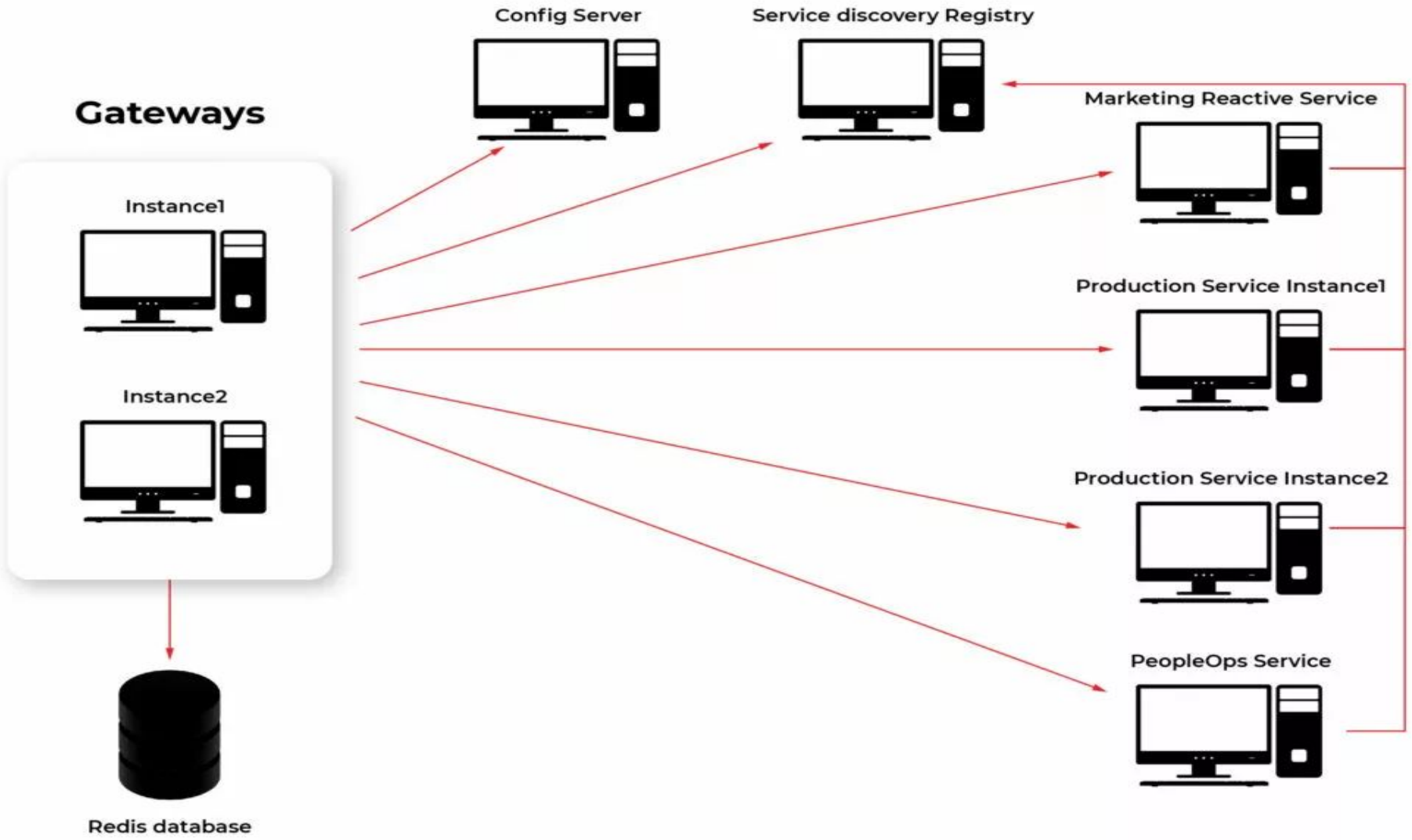
# API Gateway

Spring Cloud Gateway aims to provide a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as: security, monitoring/metrics, and resiliency.

# spring-cloud-starter-gateway

```
dependencies {
    implementation
'org.springframework.cloud:spring-cloud-starter-gateway'
    implementation
'org.springframework.cloud:spring-cloud-starter-netflix-eure
ka-client'

    //lombok
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation
'org.springframework.boot:spring-boot-starter-test'
}
```

# Request Routing



# Global Filter

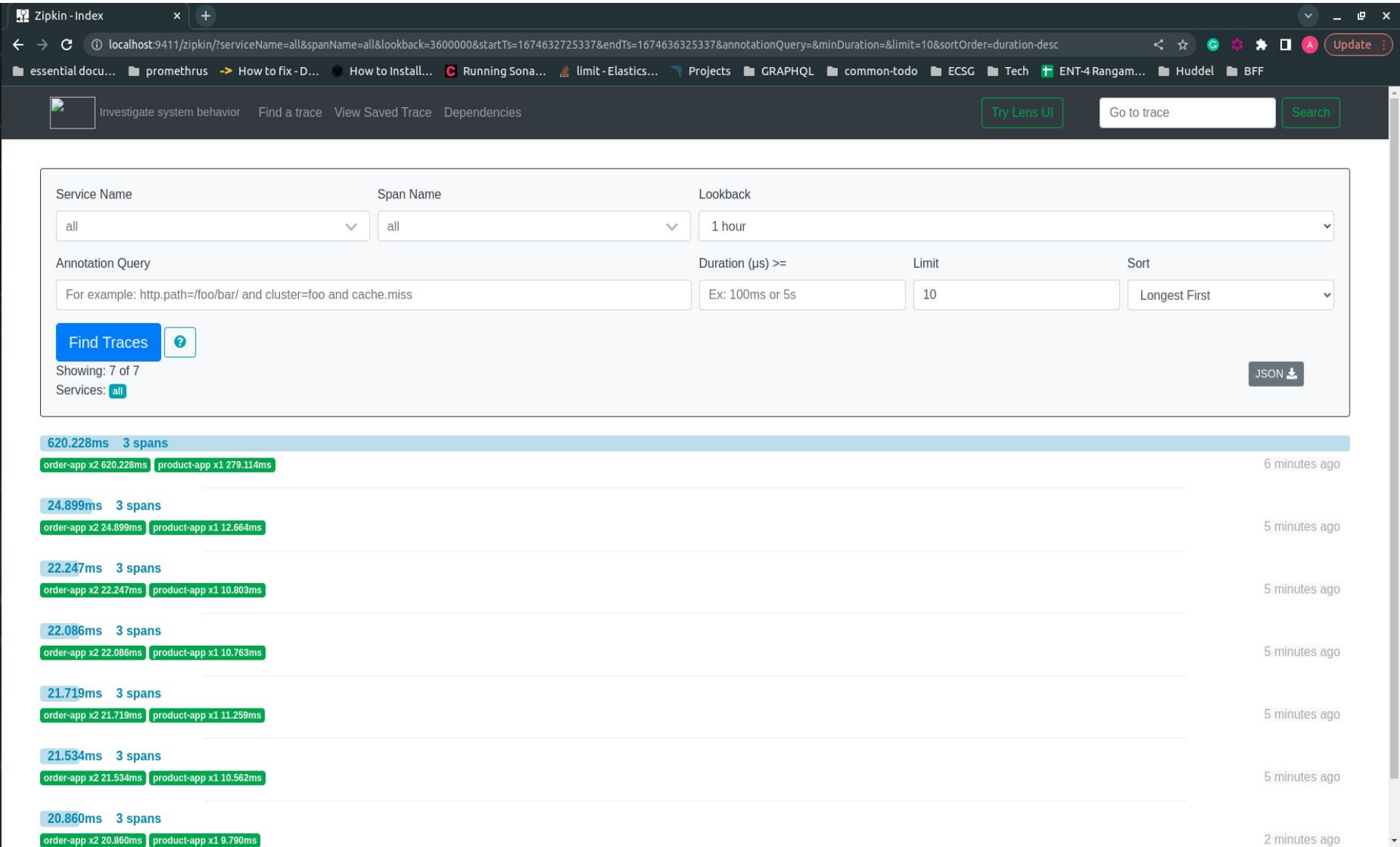
```
@Configuration
public class APIGatewayCustomFilter implements GlobalFilter
{
    Logger logger =
LoggerFactory.getLogger(APIGatewayCustomFilter.class);
    @Override
    public Mono<Void> filter(ServerWebExchange exchange,
GatewayFilterChain chain) {
        logger.info("Global Pre Filter executed ");
        logger.info("Request Headers = " +
exchange.getRequest().getHeaders());
        //return chain.filter(exchange);
        return
chain.filter(exchange).then(Mono.fromRunnable(() -> {
            logger.info("Global Post Filter executed: " +
exchange.getResponse().getStatusCode());
        }));
    }
}
```

# Distributed Tracing

Add a Unique ID to our logs

- Trace Id: unique id is the same on the entire request
- Span Id: one trace id will contain multiple span IDs, by which we can identify individual services.
-

# Aggregate logs through Zipkin

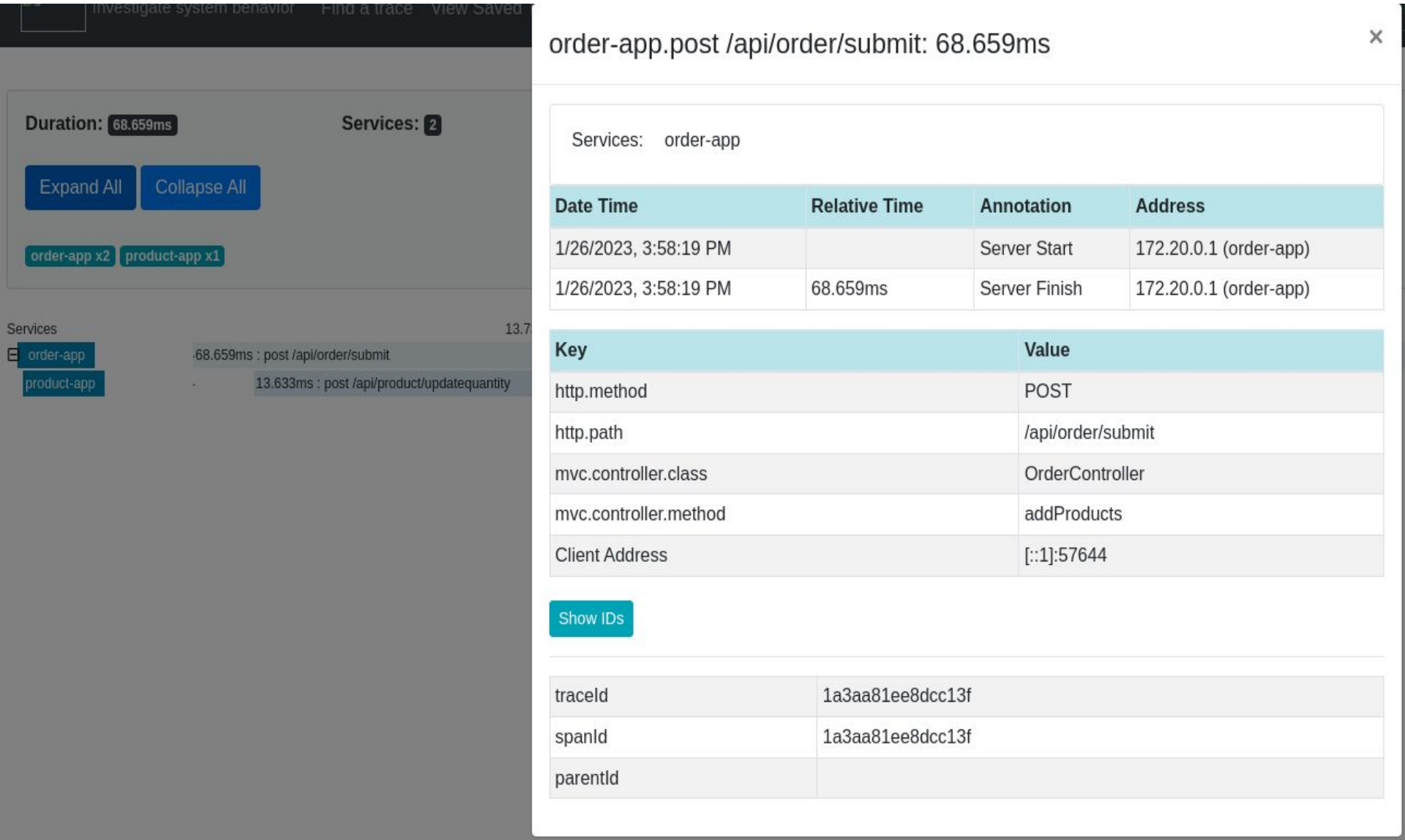


# Order Success

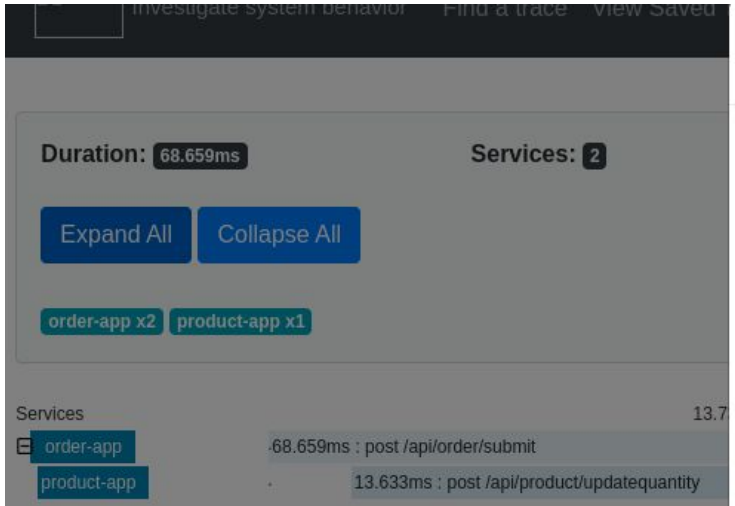
traceId list

| Order App | | Product App | |
|---|---|---|---|
| traceId | A | traceId | A |
| spanId | A | spanId | B |
| parentId | | parentId | A |

# Order App call



# Product-App call




product-app.post /api/product/updatequantity: 13.633ms

Services: order-app,product-app

| Date Time | Relative Time | Annotation | Address |
|---|---|---|---|
| 1/26/2023, 3:58:19 PM | 2.393ms | Client Start | 172.20.0.1 (order-app) |
| 1/26/2023, 3:58:19 PM | 5.175ms | Server Start | 172.20.0.1 (product-app) |
| 1/26/2023, 3:58:19 PM | 16.026ms | Client Finish | 172.20.0.1 (order-app) |
| 1/26/2023, 3:58:19 PM | 16.093ms | Server Finish | 172.20.0.1 (product-app) |

| Key | Value |
|---|---|
| http.method | POST |
| http.path | /api/product/updateQuantity |
| mvc.controller.class | ProductController |
| mvc.controller.method | updateProductsQuantity |
| Client Address | 192.168.96.217:45132 |

Show IDs

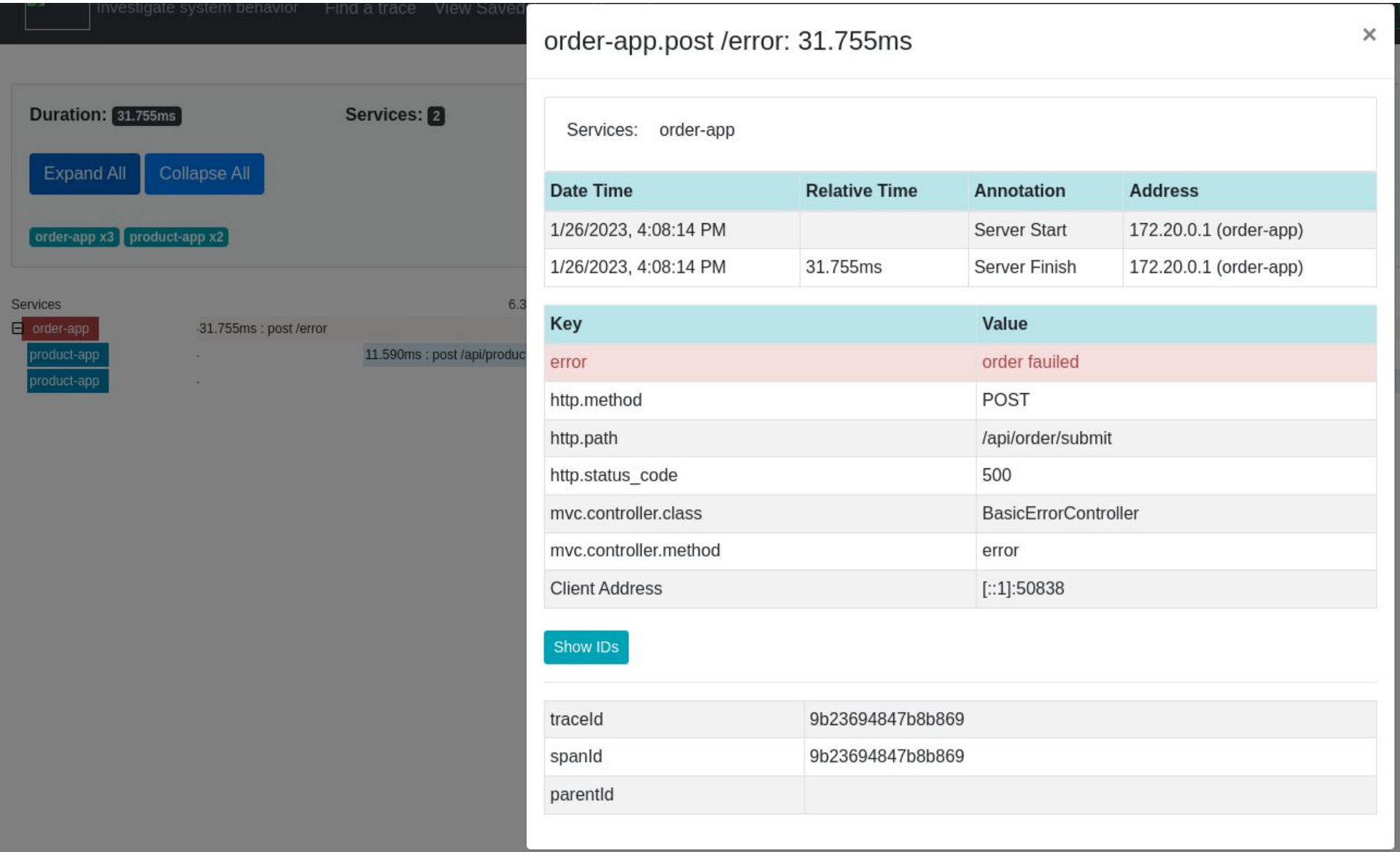
| | |
|---|---|
| traceId | 1a3aa81ee8dcc13f |
| spanId | 4c7f23a840ee1136 |
| parentId | 1a3aa81ee8dcc13f |

# Order Failed

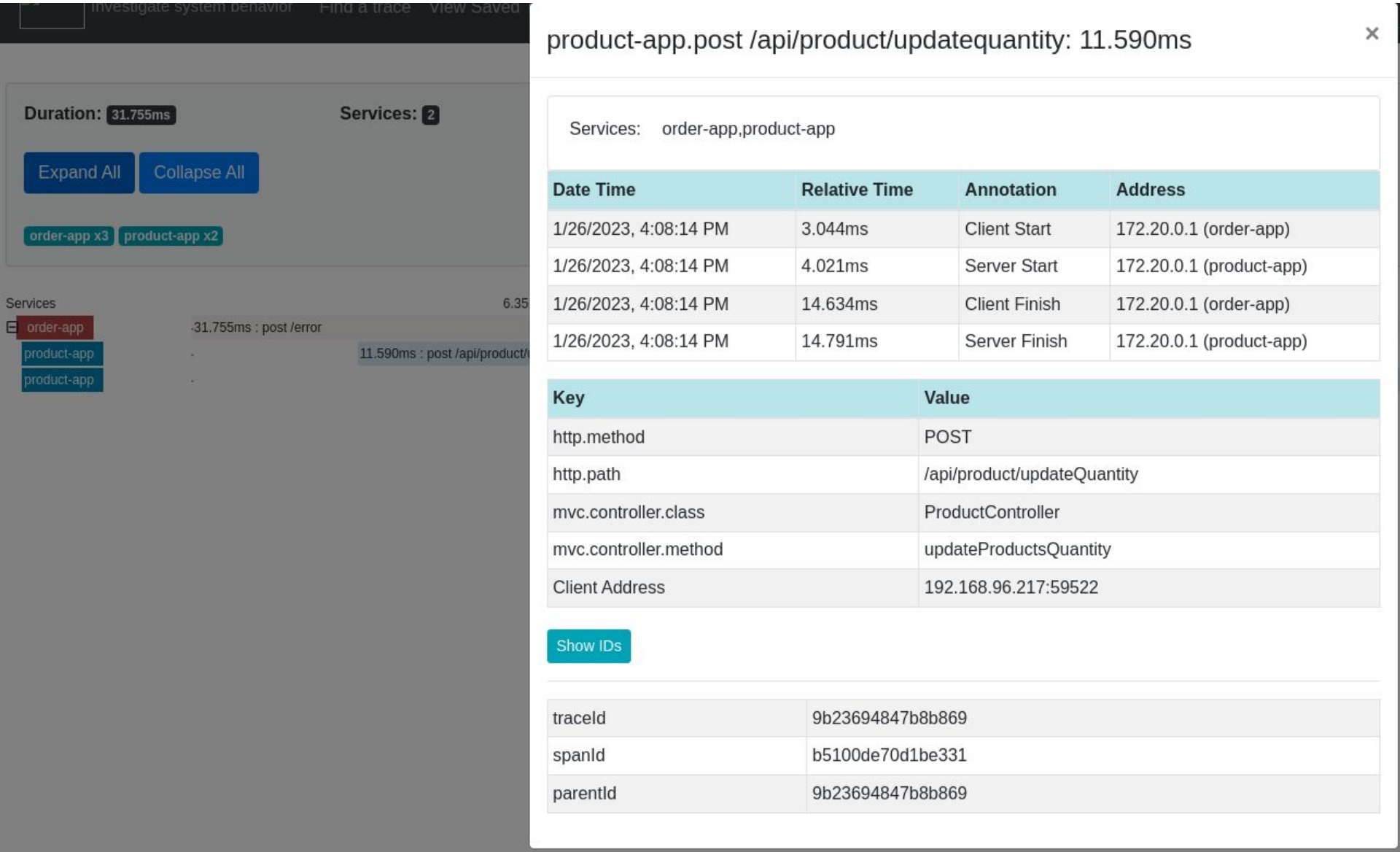
traceId list

| Order App [/api/order/submit] | | Product App [/api/product/updateQuantity] | | Product App [/api/product/revokeQuantity] | |
|---|---|---|---|---|---|
| traceId | A | traceId | A | traceId | A |
| spanId | A | spanId | B | spanId | C |
| parentId | | parentId | A | parentId | A |

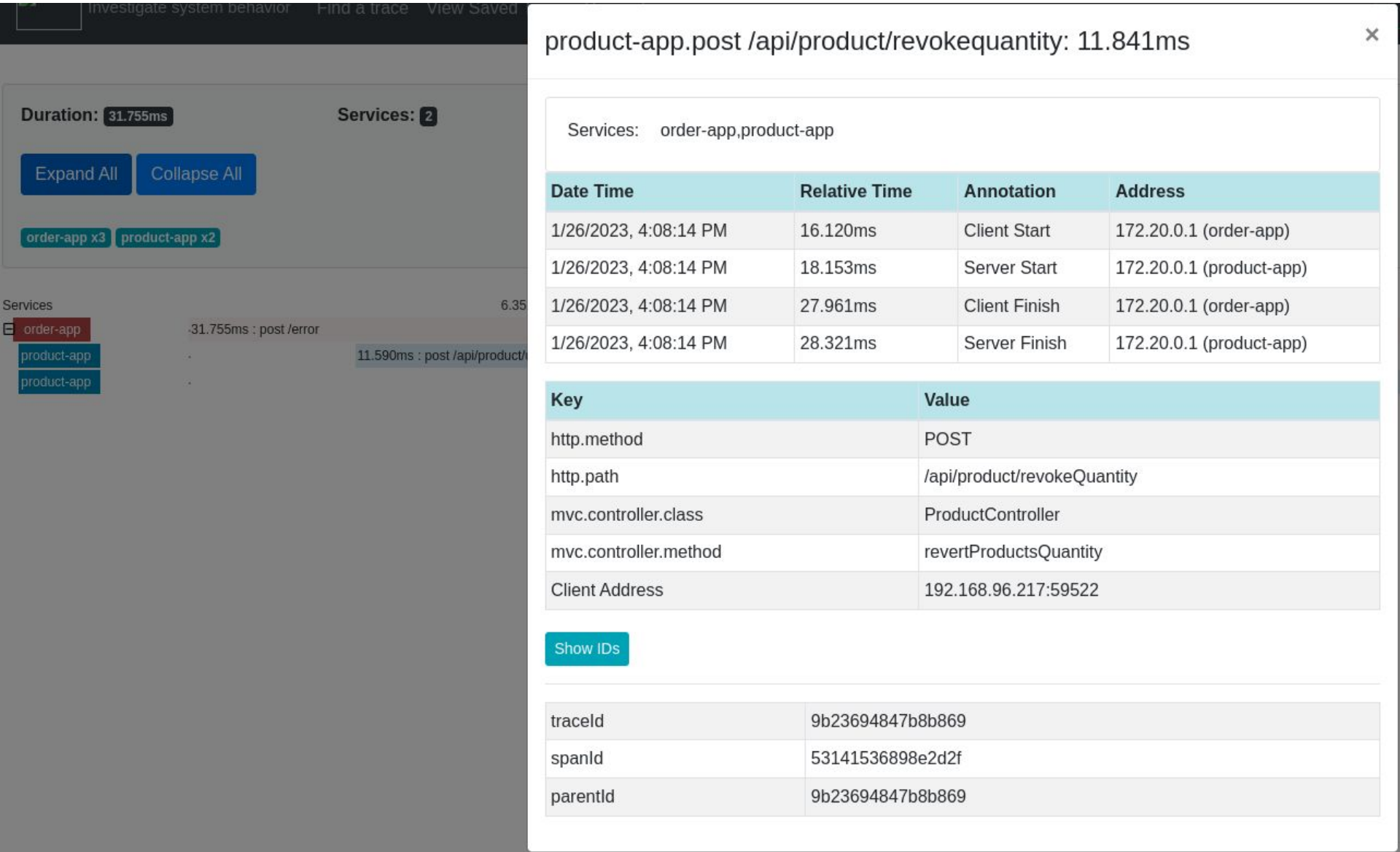
# When Order Failed



# Product app Update call /api/product/updateQuantity



# Product app Update call /api/product/revokeQuantity



# Task

write a microservice named **transaction** service it will be called with specific parameters,

1. transaction service will return

true or false by reading config properties from the config server.

2. after that order will be placed successfully or failed.
3. transaction service will be registered with service discovery server

```
        //call product service
        ResponseEntity<List<ProductResponseDto>> apiResponse =
                productFeignClient.updateProductsQuantity(productOrderRequestDto);
        List<ProductResponseDto> requestedProductList =
                apiResponse.getBody() != null ? apiResponse.getBody() : Collections.emptyList();
//      callProductService.callProductService(productOrderRequestDto,
//              "http://localhost:8082/api/product/updateQuantity");


        // call payment app
        // if success then return
        //else revert as failure
        if (transaction()) {
            //transaction success
```

# Day3

a. Various Profile management using Spring Cloud config

b. Follow up hands-on task

c. Q & A

- Ref:
- https://www.bmc.com/blogs/microservices-architecture/
- https://microservices.io/
- https://medium.com/microservices-architecture/top-10-microservices-framework-for-2020-eefb5e66d1a2
- https://www.baeldung.com/spring-cloud-netflix-eureka
- https://grapeup.com/blog/big-picture-of-spring-cloud-gateway/#
- https://www.baeldung.com/spring-cloud-custom-gateway-filters