

Building MicroServices using Spring Cloud

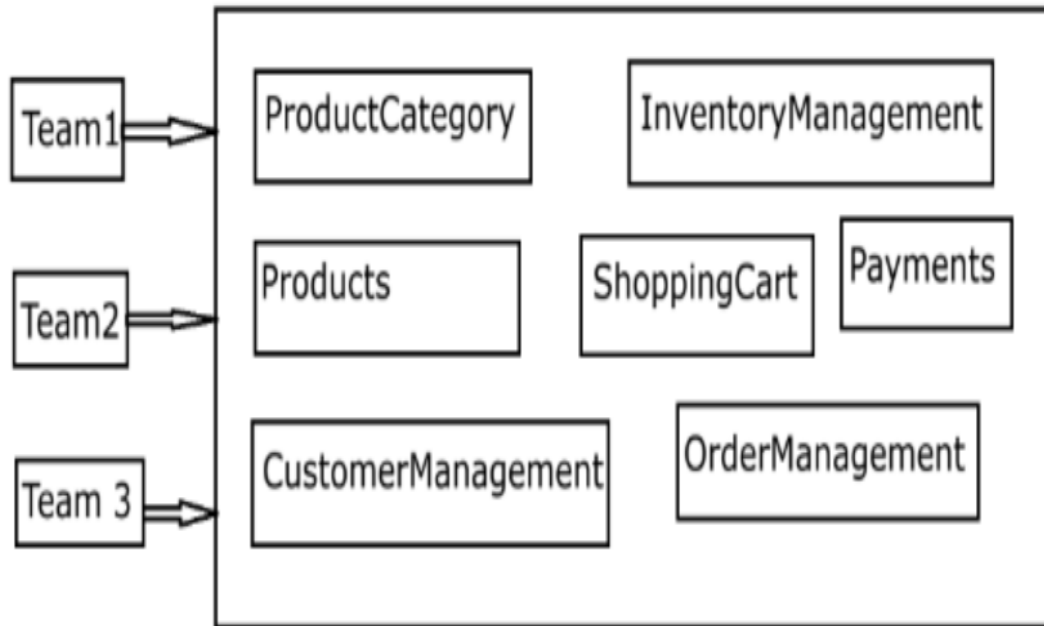
From Basics To Advanced

Before Microservices

- **Monolithic Architecture**
- What is it ?
- Disadvantages
 - **Brittle architecture**
 - Scalability costly
 - Single Technology Stack
 - Huge Codebase deployed in single artifact
 - Tougher re deployment
 - Tight coupling

Visualization of Monolith Architecture

Monolithic Architecture for the typical ECommerce App



Single Huge codebase

Deployed as single artifact

Single Tech Stack

Must redeploy on every single smallest change

Enter Microservices

- What is it ?
 - It is a smaller (micro) , independent application(RESTful web service) , which can be developed , tested , deployed and scaled independently on separate technologies.
 - Has its own Database (SQL / NO SQL based)

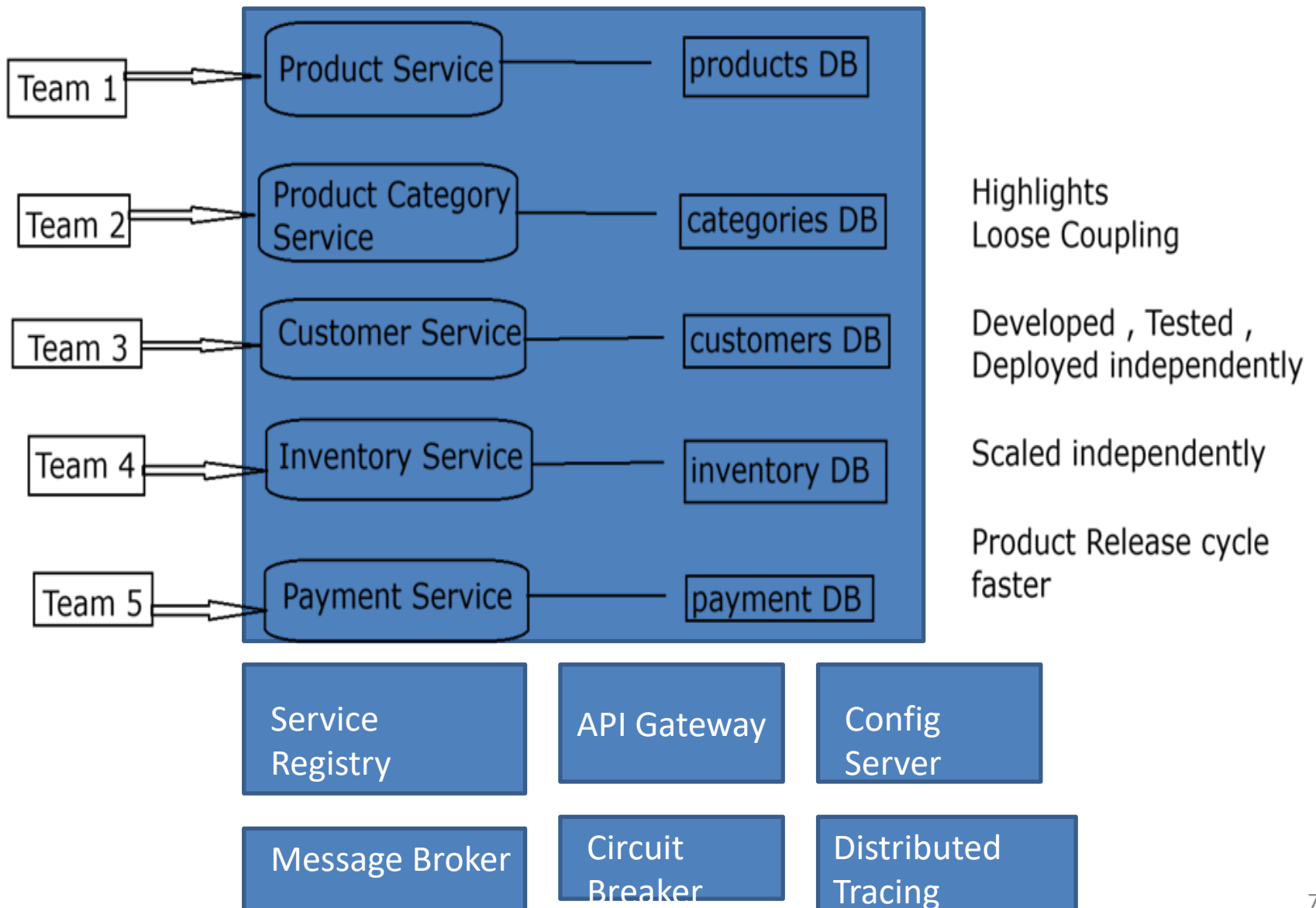
More Details About Microservices

- It is an architectural approach to building software applications that helps both developers and businesses
 - to achieve faster delivery cycles, faster innovation, improve scalability, and help teams to work independently.
- The entire application is broken into a smaller independent services
- Each of this service is self-contained that performs a dedicated function and links with other services through APIs and protocols.
- Different programming languages, frameworks, and technologies are used to develop these services.

Benefits of Microservices

- 30% Higher customer satisfaction/retention
- 29% Faster time to market/ responding to changes in the marketplace
- 28% Improved application quality/performance
- 29% Better security of company/customer data

Microservices Based Architecture



Challenges in Designing Microservices based application

- How to identify the break down of the entire application ?
- How many separate services to create ?
- How to keep track of different services, their host names and port numbers ?
- How to communicate between different services ?
- Debugging , monitoring and many more concerns....

Best Practices

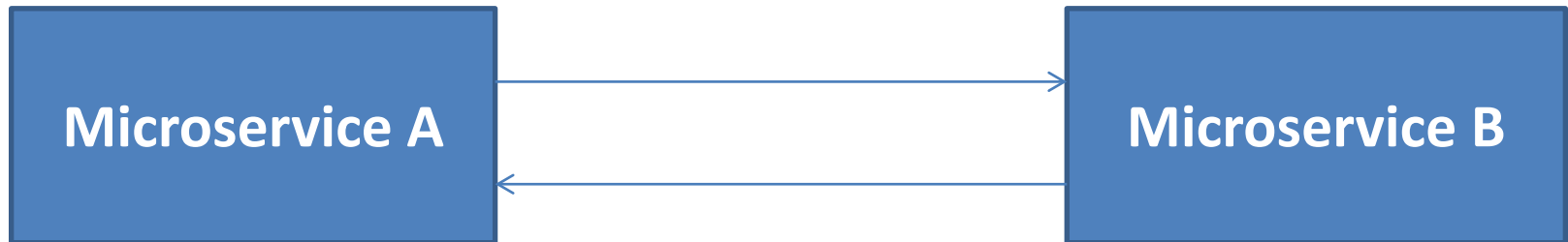
- Each Microservice has its own database.
- Client (API or Frontend) interacts only through the API gateway,
 - Do not have direct access to the services.
- Each service will be registered to the discovery server. The discovery server keeps track of all the Microservices available in the system.
- Configuration server contains all the configurations for the Microservices.
- Each Microservice is built as a separate Spring Boot application, which can be developed, deployed, and scaled independently.

Communication between Services

- 2 ways of communication
 - Synchronous communication
 - HTTP client sends a request and awaits its response , causing blocking of the invoker thread.
 - Can use RestTemplate or Web Client or Open Feign client
 - Asynchronous communication
 - The client sends a request to the Middleman (message broker) and does not wait for the response
 - Can use RabbitMq or Apache Kafka

Synchronous Communication

- HTTP Request and Response based communication



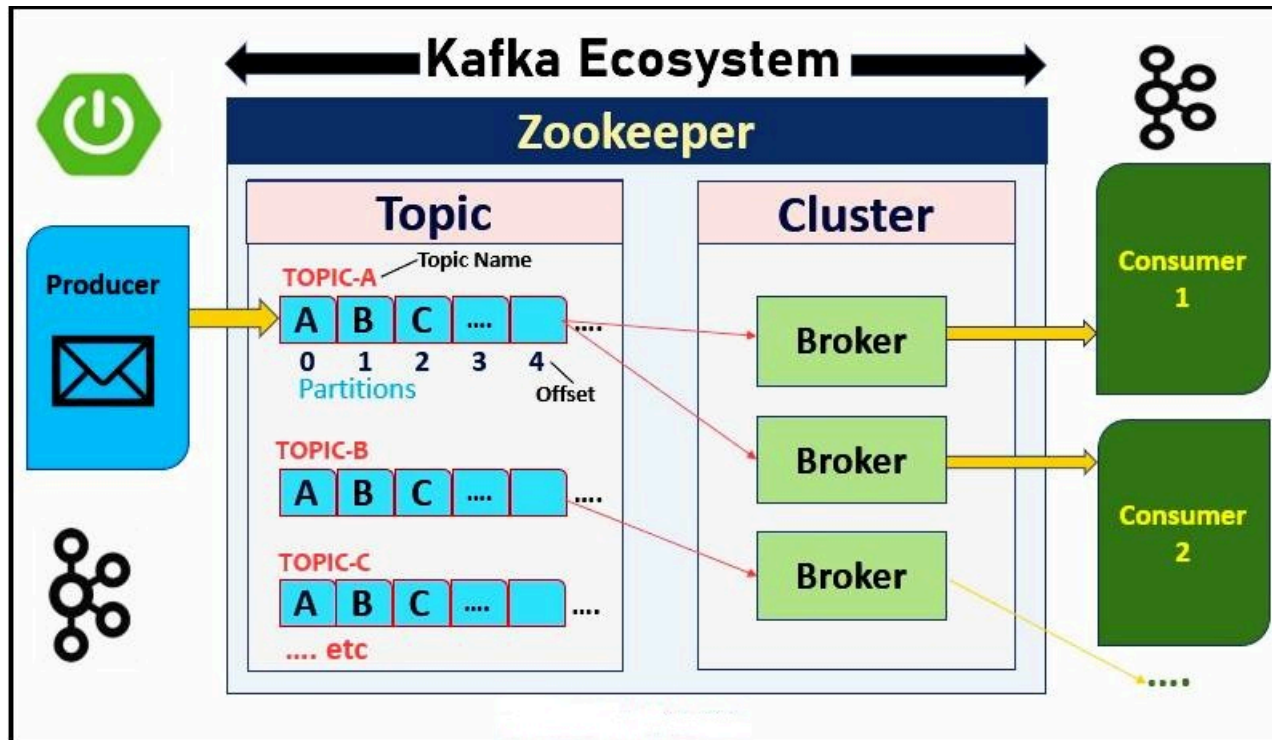
- REST API call (Blocking)

Asynchronous Communication

- Complete de coupling of producer from the consumer
- Based upon MoM (Message Oriented Middleware)



Kafka Based Asynchronous communication



More challenges and their solutions

- Client has to remember the host name and port numbers of different micro services and its instances.
- Solution
 - Design common entry point , where the client sends all the requests. This entry point will further delegate the requests to actual microservice.
 - Design Pattern – **API Gateway**

Continued with Problem and Solution

- Problem
 - In case of increased or reduced load , individual micro services will be scaled up or down . How to dynamically locate these services in a distributed environment ?
- Solution
 - Different services can register to centralized registry and can discover each other dynamically
- Service Registry and Discovery Pattern

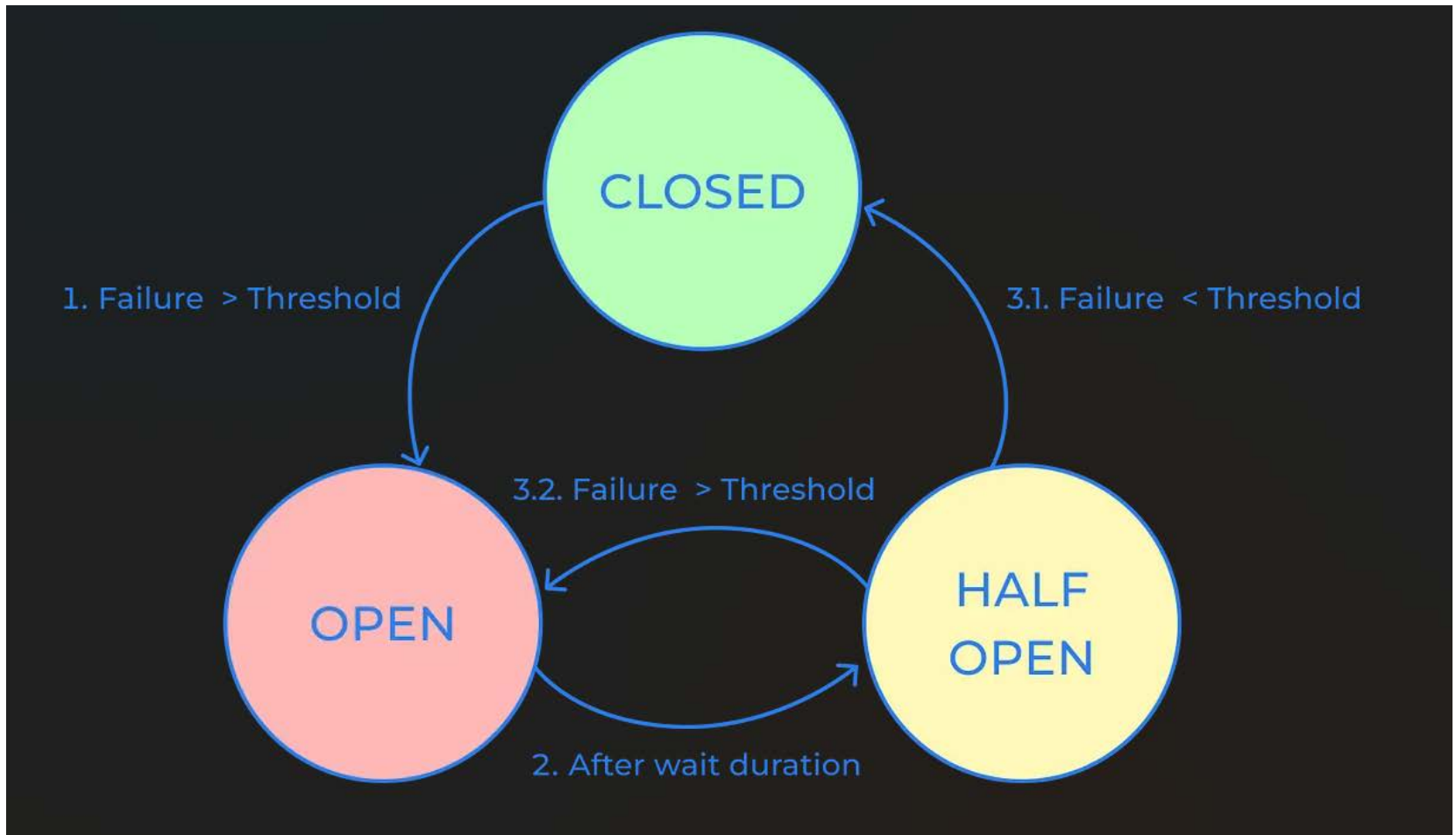
Continued

- In case of one micro service instance down or busy , the call should be redirected to another instance
- Solution
 - Load Balancing
- Server side or Client Side Load Balancing

Continued

- In case of synchronous communication between the services, when a particular service is showing high latency or is completely unresponsive, then that failure is spread across multiple systems causing Cascading of failures.
- To build fault toleant system
- **Circuit breaker design pattern**

Circuit Breaker Pattern



Problem and solution

- Different micro services will have their separate configuration. If its tightly coupled within the service , anytime config changes , all of the instances will have to be restarted
- Solution
 - Externalize the configuration
- **Centralized Configuration Pattern**

Continued

- In a distributed application where , multiple services communicate with each other over the network , it challenging to diagnose and debug issues when they arise.
- Solution
- Distributed Tracing Pattern

Spring Cloud

- Provides ready made implementation for earlier mentioned design patterns
- **Example Patterns and Implementations**
- *The API Gateway Pattern - **Spring Cloud Gateway***
- *The Circuit Breaker Pattern - **Resilience4j Integration***
- *The Service Registry and Discovery Pattern – **Spring Cloud Eureka Server n Client***
- *The Config Server Pattern – **Spring Cloud Config , Spring Cloud Bus***
- *Additionally Saga Pattern , Event Driven architecture pattern n many more..*

Case Study

- Food Ordering Distributed Application
- Microservices used
 - Restaurant Service
 - Restaurant food menu Service
 - User Service
 - FoodOrder Service

Food Ordering App Requirements

- Admin can add restaurants
- Admin can add food items (menu) for specific restaurant
- Customers can sign up n sign in
- Customer can get list of all restaurants
- Customer can get specific restaurant n its menu details
- Get specific customer details by its ID
- Customer can place food order , from specific restaurant , containing list of food items

Restaurant Use Case

- Add New Restaurant
 - **URL** <http://localhost:8080/restaurants>
 - **Method** POST
 - **Payload** RestaurantDTO
 - Name , address , city , description
 - **Response Status** 201
 - **Response Body** Restaurant DTO

Restaurant Use Case

- Fetch Details of All Restaurants
 - **URL** <http://localhost:8080/restaurants>
 - **Method** GET
 - **Payload** none
 - **Response Status** 204 (empty response) or 200
 - **Response Body** List<Restaurant DTO>

Restaurant Use Case

- Fetch Details of specific Restaurant by ID
 - **URL**
<http://localhost:8080/restaurants/{restaurantId}>
 - **Method** GET
 - **Payload path variable** restaurantId
 - **Response Status** 404 (with error message) or 200
 - **Response Body** Restaurant DTO

Development steps for Restaurant Service

- Create Spring boot project
 - STS 4 , Spring Boot 3.4.1 ,Jakarta EE 9
- Add Dependencies
 - Web , JPA , MySql , swagger , model mapper , validation,lombok
- Configure application.properties
- Create Entity , Repository , Service and controller.
- DTO , custom exceptions and global exception handler

Testing of Restaurant MS

- Either use swagger
 - <http://localhost:8080/swagger-ui/index.html>
- OR import postman collection
 - Food Ordering MS
 - Run the end points from the folder
 - Food Ordering MS /restaurants

Restaurant Menu Use Case (Uses **MS communication using RestTemplate**)

- Add New Food Item to the Menu
 - **URL** `http://localhost:7070/menu`
 - **Method** POST
 - **Payload** FoodItemDTO
 - Food item Name , description, is veg , price , restaurant id
 - **Response Status** 201 or 400 (in case of invalid restaurant id)
 - **Response Body** FoodItemDTO or error message
- **Highlight** – RestTemplate based communication between MS , for validating restaurant

Restaurant Menu Use Cases

- Fetch Details of specific Food Item by ID
 - **URL** <http://localhost:7070/menu/{foodItemId}>
 - **Method** GET
 - **Payload path variable** foodItemId
 - **Response Status** 404 (with error message) or 200
 - **Response Body** FoodItem DTO , with ID

Restaurant Menu Use Case (Uses **MS communication using RestTemplate**)

- Get Restaurant n menu details
 - **URL** `http://localhost:7070/menu /restaurant/{restaurantId}`
 - **Method** GET
 - **Payload** path variable restaurantId
 - **Response Status** 200 or 400 (in case of invalid restaurant id)
 - **Response Body** RestaurantMenuDetails DTO or error message
 - RestaurantMenuDetails – RestaurantDTO + List<FoodItemDTO>
- **Highlight** – RestTemplate based communication between MS , for getting restaurant details
- (no load balancing yet)

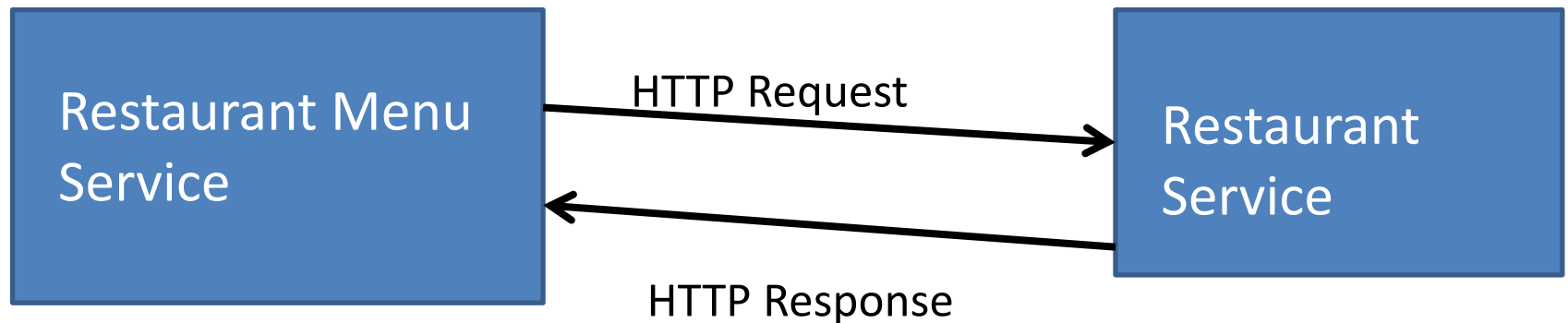
Development steps for RestaurantMenu Service

- Create Spring boot project
 - STS 4 , Spring Boot 3.4.1 ,Jakarta EE 9
- Add Dependencies
 - Web , JPA , MySql , swagger , model mapper , validation,lombok
- Configure application.properties , with additional property for getting Restaurant details
 - restaurant.get=http://localhost:8080/restaurants/{restaurantId}
- Create Entity , Repository , Service and controller, DTO , custom exceptions and global exception handler

Inter communication between Micro services using RestTemplate

Validate | Get Restaurant Details by ID

REST Client



Dev steps for using RestTemplate in RestaurantMenu Service

- Configure RestTemplate bean in Spring boot application class
 - Represents the abstraction of synchronous REST client
 - Not yet using load balancer !
- Inject value of GET URL , using SpEL , in Service class
 - `@Value("${restaurant.get}")`
private String url;

Microservice Communication using RestTemplate

- Inject RestTemplate in FoodMenuService implementation class
- Make REST call , using RestTemplate API
<T> T getForObject(String url, Class<T> responseTypeCls, Object... uriVariables) throws RestClientException
- It gets representation of the resources by doing a GET call on the specified URL. The response is converted and returned.
- Can pass URI variables , if needed

Testing of RestaurantMenu MS

- Either use swagger
 - <http://localhost:7070/swagger-ui/index.html>
- OR import postman collection n Run the end points from the folder , Food Ordering MS /menu
 - Add new food item
 - Get details of specific food item
 - Get restaurant n its complete menu

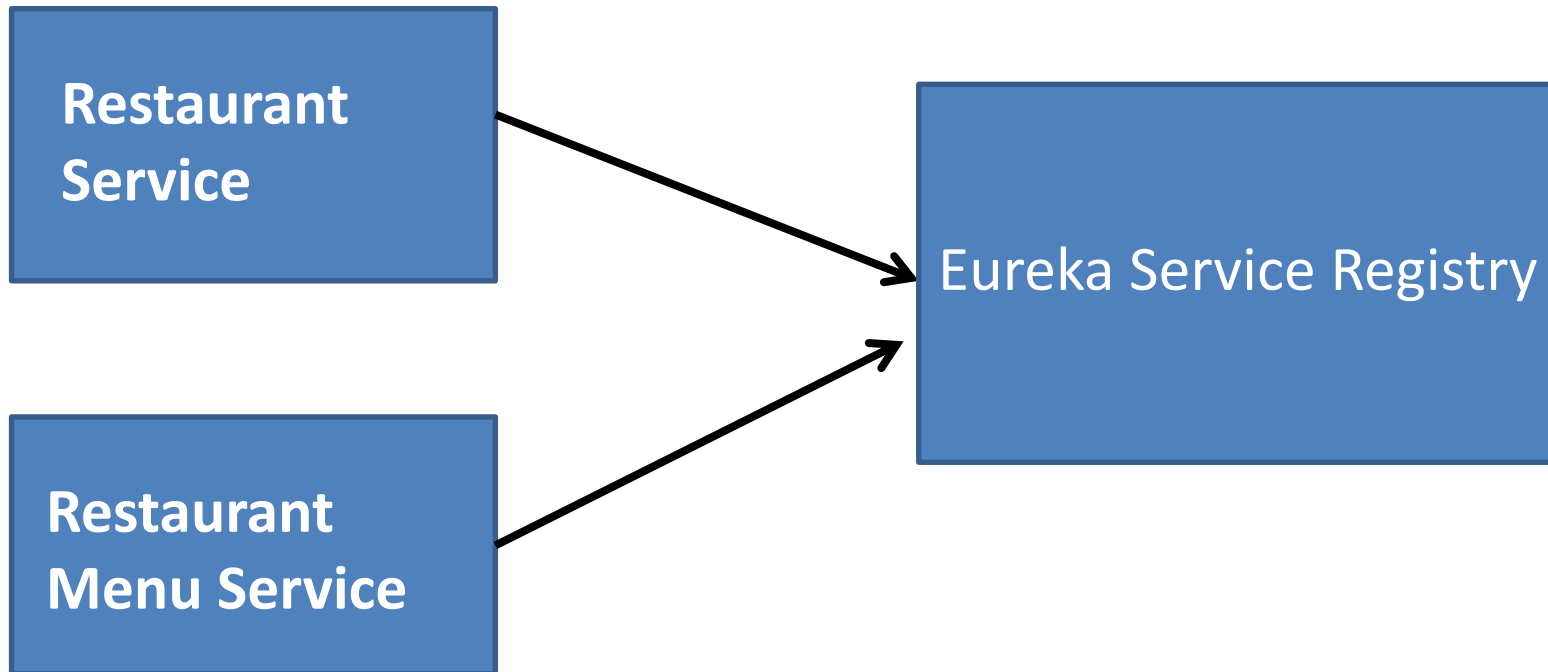
Why Service Registry and Discovery?

- Problem
 - In microservices inter communication , hard coding of hostnames , port numbers required
 - Eg GET URL for getting Restaurant details specified by id from Restaurant Menu service
 - <http://localhost:8080/restaurants/{restaurantId}>
 - Load balancing unavailable

Service Registry and Discovery Pattern

- Multiple Microservices will register to a centralized registry.
- Centralized registry maintains track of all of registered services n their instances
- While making a call from one MS to another , use Spring application name instead of hard coded details.
- Enables dynamic discovery of the available MS instance n performs the call.
- Analogy – Telephone Directory

Spring Cloud Eureka Server (Netflix)



Spring Cloud Service Registry Development Steps

- Create Spring boot project
- Add dependency
 - `spring-cloud-starter-netflix-eureka-server`
- Configure `application.properties`(or `yml`) for this Eureka Server
 - `eureka.client.fetch-registry=false`
 - Disable Fetch registry from Eureka Server
 - `eureka.client.register-with-eureka=false`
 - Disable register with Eureka
 - `server.port=8761` (Default port for Eureka Server)
- Add `@EnableEurekaServer` annotation on Spring Boot Application class

Test Service Registry

- Build (Maven Goals – clean n package) Spring Boot app
- Run Service Registry JAR from cmd prompt
 - `java -jar ServiceRegistry-0.0.1.jar`
- Launch Eureka dashboard
- From web browser
 - <http://localhost:8761>
- Should display Eureka dashboard , with no current instances registered.

Register RestaurantService As Eureka client

Development Steps

- Add Eureka Client dependency in pom.xml
- Add the property , to register Restaurant Service as Eureka client with Eureka server
 - eureka.client.serviceUrl.defaultZone=
<http://localhost:8761/eureka/>
- Build n run Spring Boot app.

Register Restaurant Menu Service As Eureka client

Development Steps

- Add Eureka Client dependency in pom.xml
- Add the property , to register Restaurant Menu Service as Eureka client with Eureka server
 - eureka.client.serviceUrl.defaultZone=
<http://localhost:8761/eureka/>
- Build n run Spring Boot app.

Testing Service Registry

- Refresh Eureka Console
- Should display 2 instances registered with Eureka
 - Restaurant-Service
 - Restaurant-Menu-Service

Eureka Client Dynamic Discovery

- Add `@LoadBalanced` over `RestTemplate Bean` , in spring boot application class
 - It enables client side load balancing
- Replace hard coded host name n port no by Service Name , in the configuration file
 - `restaurant.get`=`http://Restaurant-service/restaurants/{restaurantId}`
- Re build , run n test microservice communication as earlier , using swagger or postman
- Eg Get Restaurant n List of Food Item Details by restaurant id (postman collection)

Customer Sign Up use Case

- Add New Customer
 - **URL** http://localhost:8080/users/signup
 - **Method** POST
 - **Payload** UserReqDTO
 - Name , email, password, dob
 - **Response Status** 201 or 400 in case of validation failures or in case of dup email
 - **Response Body** error mesg or Restaurant DTO

Customer Sign in Use Case

- Customer Sign in
 - **URL** http://localhost:8080/users/signin
 - **Method** POST
 - **Payload** AuthRequest
 - Email n password
 - **Response Status** 401 or 200
 - **Response Body** error message for invalid email | password or User response DTO

User(Customer) Use Case

- Get User Details
 - **URL** http://localhost:8080/users/{userId}
 - **Method** GET
 - **Payload** path variable userId
 - **Response Status** 404 or 200
 - **Response Body** error message or User resp DTO ,
with generated User ID

Development steps for User Service

- Create Spring boot project
 - STS 4 , Spring Boot 3.4.1 ,Jakarta EE 9
- Add Dependencies
 - Web , JPA , MySql , swagger , model mapper , validation,lombok , **Eureka Client**
- Configure application.properties
 - DB , JPA n **Eureka client properties**
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/
- Create Entity , Repository , Service and controller.
- DTO , custom exceptions and global exception handler

Testing User MicroService

- Insert admin record
 - insert into users values (default,'1990-10-20','admin@gmail.com','admin','admin',
- Refresh Eureka dashboard , to confirm running of User Service instance
- Test end points either using swagger | postman collection
 - Sign up
 - Sign in
 - Get customer Details by ID

Place Food Order Use Case

- Place Food Order
 - **URL** http://localhost:8080/orders
 - **Method** POST
 - **Payload** OrderRequestDTO
 - **Response Status** 201 or 400 | 404 in case of validation failures or in invalid customer | restaurant
 - **Response Body** error mesg or OrderRespDTO

Schemas of Request DTO

- Order Request DTO
 - List<FoodOrderItem>
 - Customer ID
 - Restaurant ID
 - Delivery Address DTO
- FoodOrderItem
 - Food Item Id
 - Quantity

Schemas of Response DTO

- **Order Response DTO**

- Order ID
- Order Status (*NEW, PROCESSING, DELIVERED, CANCELLED*)
- *Order Amount (total bill)*
- *Order placing Date Time*
- *Promised delivery Date Time*
- *Delivery charges*
- List<OrderLineDTO>
- Restaurant Name
- Customer Name

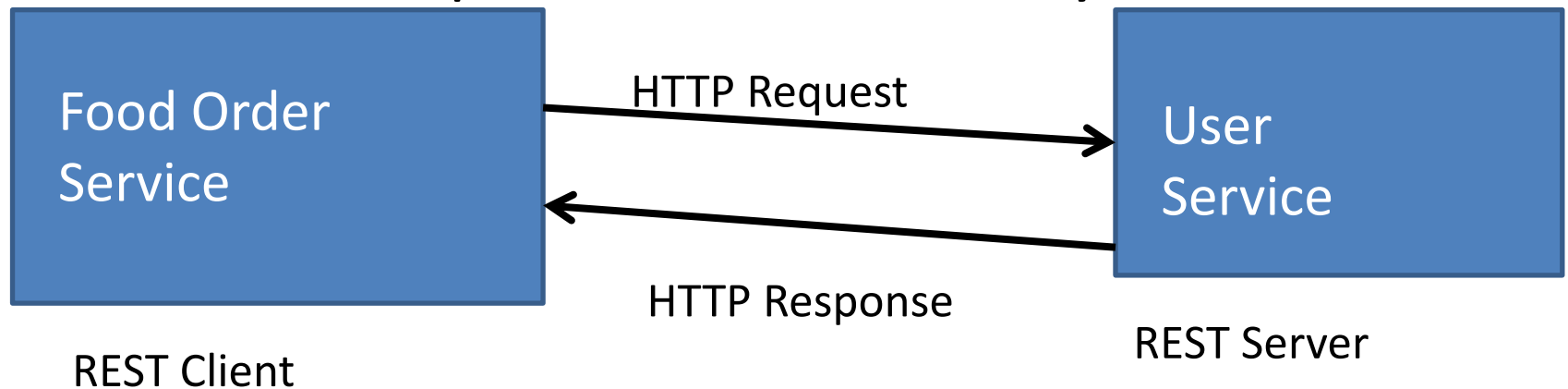
Schema of OrderLineDTO

- OrderLineDTO
 - Food Item Name
 - Quantity
 - Sub total

Inter communication between Micro services using WebClient

RestTemplate – Maintenance mode – to be
Deprecated API (Replaced by WebClient)

Validate | Get User Details by ID



Web Client Supports synchronous , asynchronous and streaming communication between micro services

Dev Steps for using Web Client

- Since WebClient is a part of Rective Spring web , add new dependency
 - spring-boot-starter-webflux
- Configure WebClient.Builder as a Spring bean , in spring boot application class
 - Add @LoadBalanced for client side load balancing
- Inject WebClient.Builder in the Service layer , build WebClient
- Use WebClient methods for synchronous communication between microservices.

Intercommunication between MicroServices using OpenFeign Client

- Why OpenFeign ?
 - With RestTemplate or WebClient , programmer has to write lot of boiler plate code for making the REST call.
- What is OpenFeign ?
 - It is an open-source project developed by Netflix and currently managed by Spring Cloud.
 - It is a declarative synchronous REST client
 - It creates a dynamic implementation of the interfaces , declared as FeignClient.
 - Lot easier than any of the earlier approaches

Development Steps in using OpenFeign Client

- Add dependency in pom.xml
 - spring-cloud-starter-openfeign
- Add `@EnableFeignClients` , to enable OpenFeign clients
- Create Client interface annotated with
 - `@FeignClient(name = "Service-Name")`
 - Declare interface methods
 - Eg - `@GetMapping("/restaurants/{restaurantId}")`
 - **public** RestaurantDTO findRestaurantById(@PathVariable Long restaurantId);
- OpenFeign library will automatically generate the implementation logic for making a REST call
- **Does it remind you of anything learnt earlier ?**

Business Logic Steps for Placing Order in Service Layer

- Validate n get Customer details by customer Id from User MicroService , via WebClient
- Validate n get Restaurant details by restaurant Id from Restaurant MicroService , via OpenFeign client
- Create Order entity , with order status – NEW
- Establish uni directional association between Order and DeliveryAddress
 - Order HAS-A Delivery Address

Steps Continued

- Assign customer id n restaurant id in Order Entity
- Set Promised delivery time (30 minutes , after placing the order)
- Save order entity , using Order Repository
- Create order lines using ordered food items
- Assign order lines to current Order
 - Establish bi directional association between Order 1 \leftrightarrow * OrderLine
 - Single Order HAS-A multiple OrderLines
- Generate Order Response DTO with
 - Order details
 - Order line details
 - Restaurant name
 - Customer name

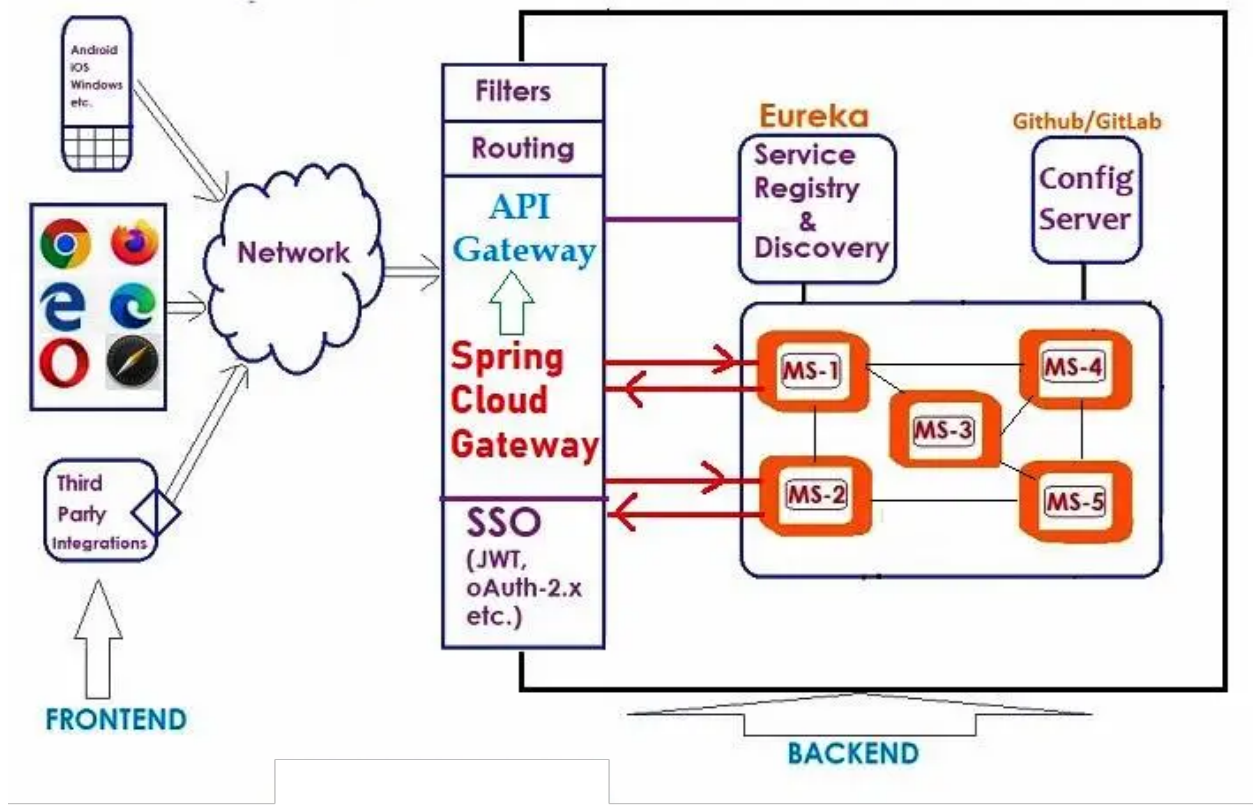
Testing Of Place Order

- Launch postman collection Food Order MS / orders n run
 - Place Food Order By Specified Customer for Specified Restaurant
- Confirm from the orders DB , record insertion in
 - orders , order_lines n delivery_address tables.

API Gateway

- Why ?
 - It simplifies the communication between a client and a service (i.e frontend application and the backend application)
 - In its absence Client will have to know details about each n every microservice running in the back end.
 - **It acts as a single entry point to access any micro service from the distributed application.**
 - It is a non-blocking and reactive gateway that provides several features like routing, filtering, load balancing, circuit breaking n cross cutting concerns like Security

Spring Cloud API Gateway



API Gateway Implementation Steps

- Create Spring boot project with dependencies
 - spring-cloud-starter-gateway
 - spring-cloud-starter-netflix-eureka-client
- Register API Gateway as Eureka client with Eureka server
- Configure routes for individual micro service
 - Id – unique ID
 - uri - lb://Service-Name
 - Predicates : Path – URL pattern of the micro service

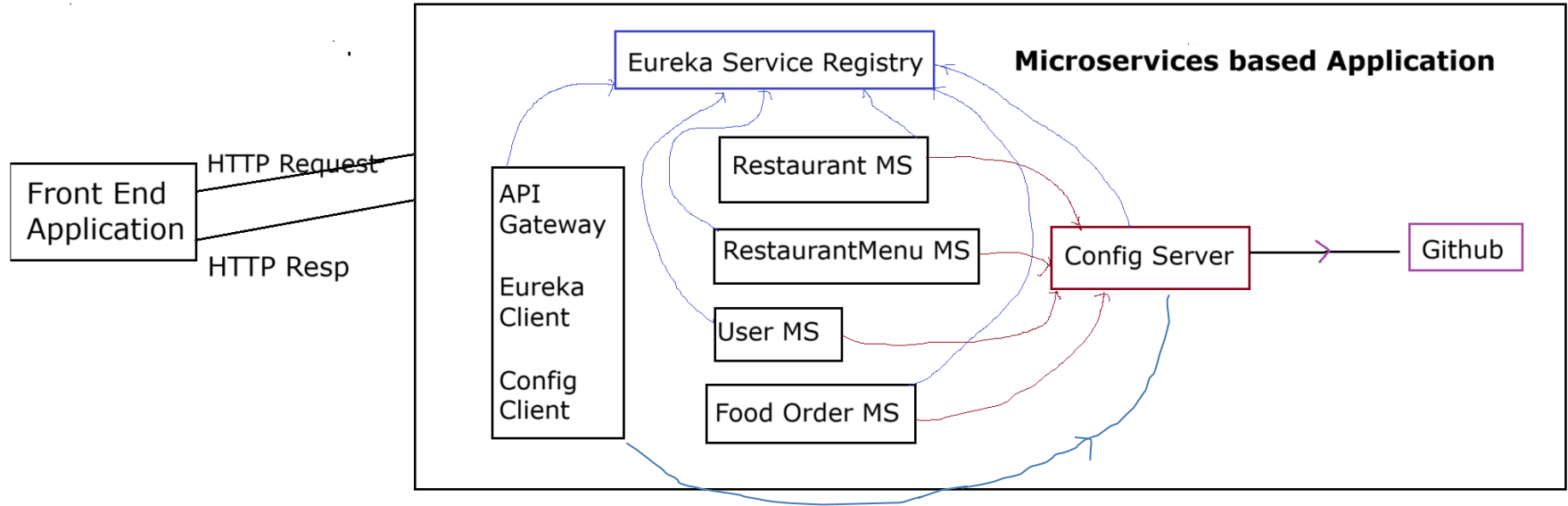
Testing

- Run Api Gateway Service
- Run postman client , for
 - Placing Food Order
 - Get Restaurant n Menu Details
- Refer to postman collection
- Concludes Microservices V3 here

Spring Cloud Config Server

- Why ?
 - Managing configuration in case of large number of micro services based architecture is tedious.
 - Hard coding configurations or maintaining separate configuration files for each service locally , leads to inconsistencies and potential errors.
- Enter **Spring Cloud Config**
 - A project under Spring cloud eco system to provide support for externalized configuration in a distributed system.
 - Supports automatic refreshes.

Block Diagram



Spring Cloud Config Server Steps

- Create Spring Boot application for Config Server Micro service
- Add dependencies
 - spring-cloud-config-server
 - spring-cloud-starter-netflix-eureka-client
 - spring-boot-starter-actuator
- Add annotation `@EnableConfigServer` on the Spring boot application class

Steps Continued

- Log in to your Git Hub account
 - Create one , if it doesn't exist
- Create new repository (public)
 - Eg – config-server-repo
- Copy its URL
- Configure the properties in the configuration file (application.properties) of Config Server Microservice

Configuration properties of Config Server

- Register Config Server as Eureka Client
 - `eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/`
- Configure Git repository location for the Config server
 - `spring.cloud.config.server.git.uri=https://github.com/username/repoName`
- To clone the Git Repo at the application start up
 - `spring.cloud.config.server.git.clone-on-start=true`
- Specify the branch of the Git remote repository
 - `spring.cloud.config.server.git.default-label=main`

Testing

- Build (clean compile | package) Config Server Project
- Run it from IDE or cmd prompt
 - `java -jar ConfigServer-0.0.1-SNAPSHOT.jar`
- Confirm with Eureka Server dashboard

Configure RestaurantService as Config Client

- Add the dependency in pom.xml
 - spring-cloud-starter-config
 - spring-boot-starter-actuator (if not added previously)
- Log in to your Git Hub account
- Create properties file for storing micro service configuration , with same name
 - Eg – restaurant-service.properties
- Copy all the properties from the local property file (application.properties) in the remote Repo
- To specify Config Server location (in optional manner)
 - spring.config.import=optional:configserver:http://localhost:9191 (port no of the config server)
- Enable Actuator's refresh end point
 - management.endpoints.web.exposure.include=refresh

Configure Other Microservices As Config Clients

- Repeat the steps of Restaurant Service for
 - API Gateway
 - User Service
 - Restaurant Menu Service
 - Food Order Service
- All of these 5 services are now Eureka Client and Config Server's client
- For checking automatic refresh
 - In Restaurant Menu Service , add @RefreshScope over the Service layer (for refreshing of properties)

Testing Config Server , Client and Refresh end point

- Edit restaurant-menu-service.properties
 - Inject purposefully wrong value of restaurant.get URL
 - Run all the Services , in the correct sequence
 - Service Registry
 - Config Server
 - API Gateway
 - Restaurant Service
 - RestaurantMenu Service
 - FoodOrder Service

Testing Steps Continued

- Run the HTTP request
 - Get Restaurant n List of Food Item Details by restaurant id
- **Expected Result** – Failure
 - Can't reach Restaurant Service from the Menu Service
- Correct the property in Git repo configuration file
- Run its actuator refresh end point
 - Eg <http://localhost:7070/actuator/refresh> , method=POST
 - 7070 : port no of Restaurant Menu Service
- Without restarting the service , try to Get Restaurant n List of Food Item Details by restaurant id
 - **Expected Result** – Success (indicating refreshing the properties , without the restart)

The Config flow known so far ...

- Application's configuration are stored in a Git repository , in a centralized manner
- Job of Spring Cloud Config Server's
 - To serve these configuration properties to your microservices (Config Clients).
- The Config Server
 - Is Configured to point to the Git repository.
 - It fetches the configuration files from Git.
- Spring Cloud Config Clients
 - Configured with the address of the Config Server.
 - When Config Client starts, it contacts the Config Server to retrieve its configuration.

Problem And Solution

- Problem
- In case of multiple micro services n their multiple instances
 - To reload the changes in configuration , invoking actuator/refresh end point on every instance
 - Feasible ?
- Solution
 - Spring cloud Bus.

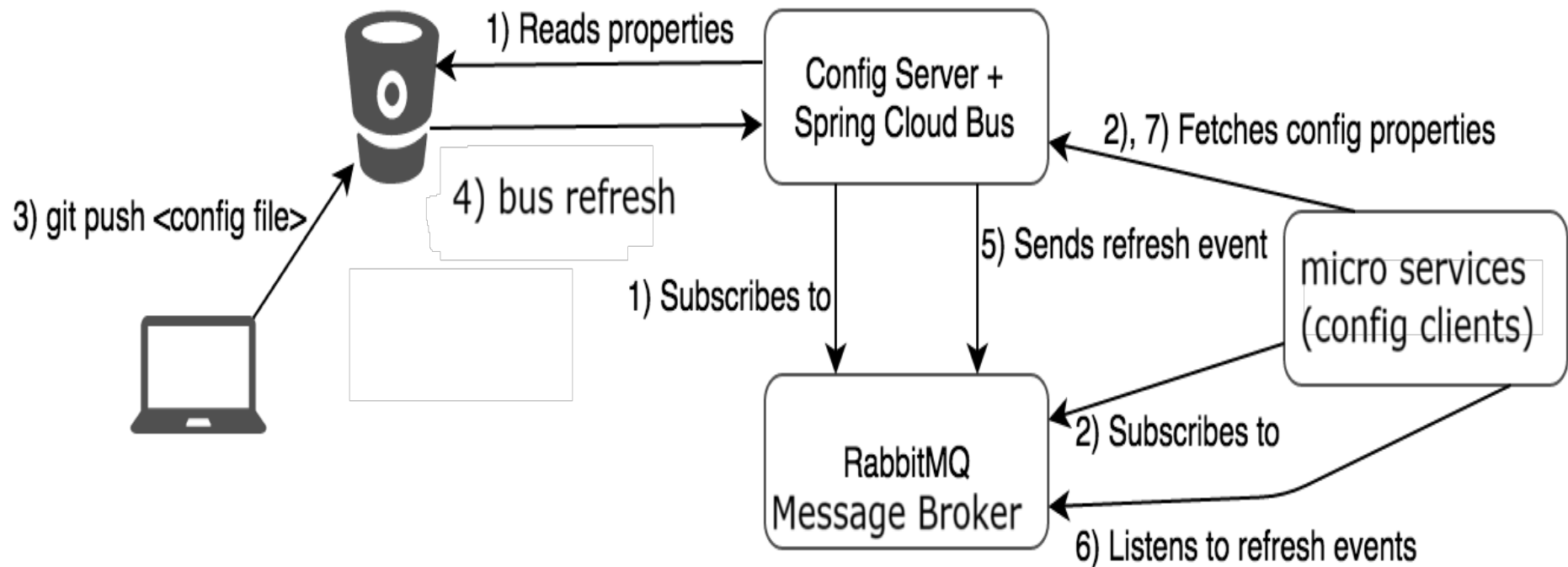
Spring Cloud Bus

- **What is it ?**
 - A project within the Spring Cloud ecosystem.
 - An event bus that uses message brokers like **RabbitMQ** and **Kafka** to communicate across distributed systems.
 - Useful for broadcasting events like **configuration updates** across a network of applications.
 - A way to dynamically refresh or update components in your microservices architecture **without requiring a complete restart** of each service.

Important Terms

- **Message Broker**
 - Spring Cloud Bus uses a message broker
 - e.g. RabbitMQ or Kafka
 - All services that want to participate in the bus connect to this broker.
- **Event Propagation**
 - When a change occurs , an event is published to the message broker.
- **Service Listeners**
 - The services that are connected to the message broker are also listening for these events.
- **Refresh Event**
 - When any service receives a RefreshEvent , it can trigger a refresh involving reloading its configuration properties.

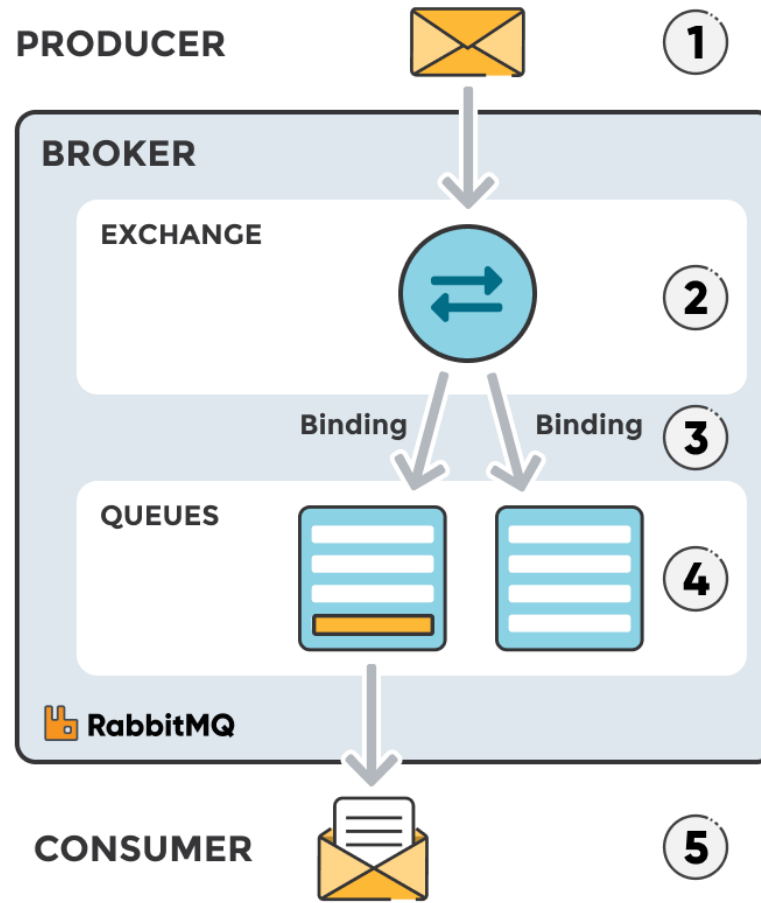
Spring Cloud Bus Architecture



AMQP and RabbitMQ

- **AMQP (Advanced Message Queuing Protocol)**
- Specification for how message queuing systems
- Defines a set of rules for how messages should be sent, received, and managed between different systems.
- Before AMQP, switching from one message broker to another was extremely difficult because they all had their own proprietary protocols
- Eg – Switching from ActiveMQ to RabbitMQ
- **RabbitMQ**
- A popular **message broker** that implements the AMQP standard.
- Acts as an intermediary between applications that send messages (producers) and applications that receive messages (consumers).
- Provides message persistence, message delivery guarantees, and different exchange types for flexible message routing.

RabbitMQ Architecture



Download , Install and Run RabbitMQ

- Download n install Docker desktop
- Launch Docker Desktop
- Docker commands (from command prompt)
 - `docker pull rabbitmq:4.0.5`
 - `docker run --rm -it -p 5672:5672 rabbitmq:4.0.5`

Implementation Steps

- **Add Spring Cloud Bus AMQP Dependency to Config Server n Config clients applications**
- **Configure Message Broker (RabbitMQ) properties**
 - `spring.rabbitmq.host=localhost`
 - `spring.rabbitmq.port=5672`
 - `spring.rabbitmq.username=guest`
 - `spring.rabbitmq.password=guest`
- **Currently using these as default values , so no need to configure explicitly.**
- **After changing the properties , in Git remote Repo**
- **Trigger Refresh on Config Server:**
 - <http://localhost:9191/actuator/busrefresh>

Explanation of the Flow

- You make a change to the configuration in your Git repository.
- You send a POST request to the `/actuator/busrefresh` endpoint on the Config Server.
- The Config Server receives the request and publishes a `RefreshEvent` to the message broker.
- All Config Clients that are connected to the message broker receive the `RefreshEvent`.
- The Config Clients then contact the Config Server to get the latest configuration.
- The Config Clients update their configurations and refresh any `@RefreshScope` beans.

Testing

1. Hit the end points (User Service n Restaurant Service)
 - Check original property values of the message
2. Edit the properties in multiple property files from the Got repo.
 - Eg – user-service.properties n restaurant-service.properties
3. Hit the end point
 - Config server actuator bus refresh end point
 - <http://localhost:9191/actuator/busrefresh> , POST
4. Repeat step 1 again
 - Check for modified values !