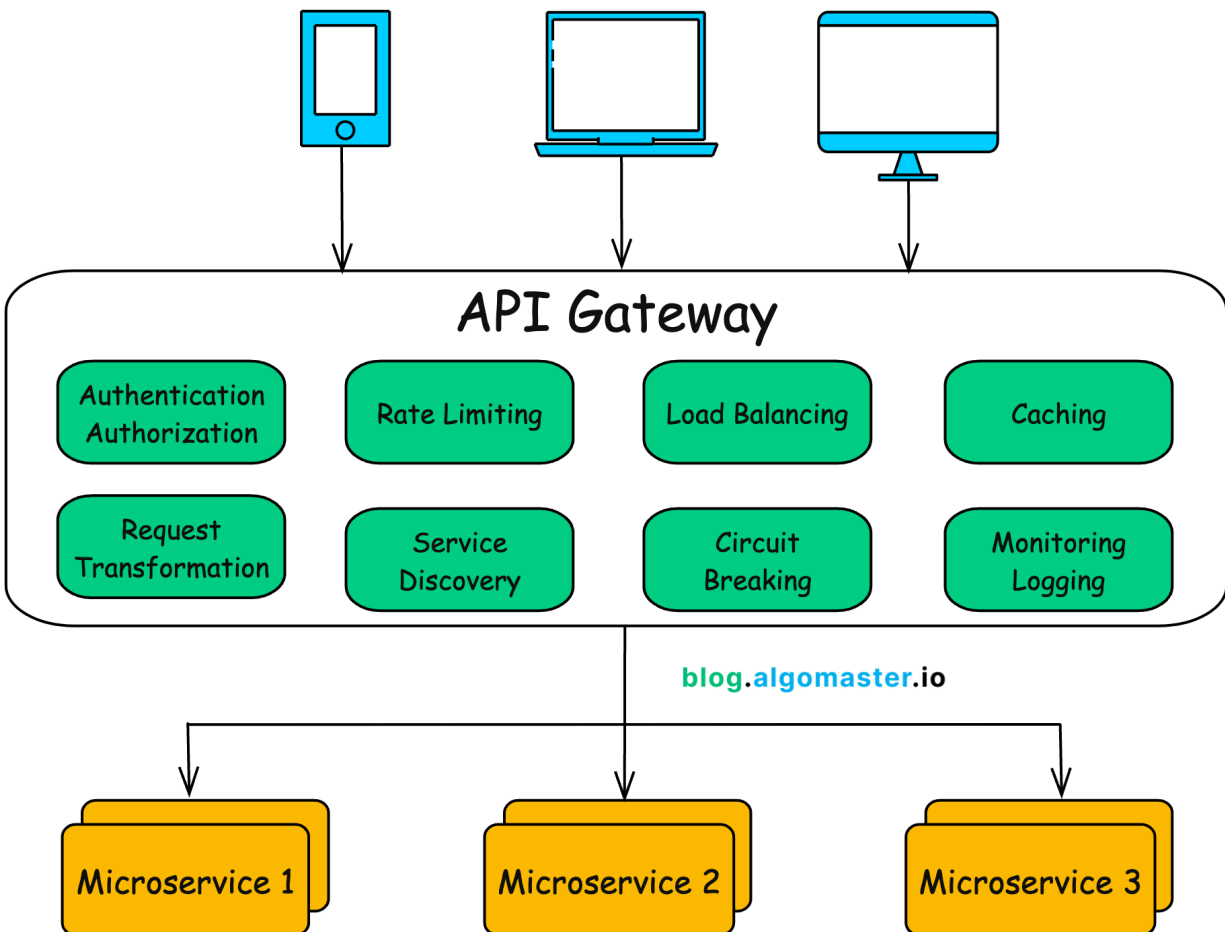


1) localhost:8085/fromsecondcontroller

2) localhost:8081/message

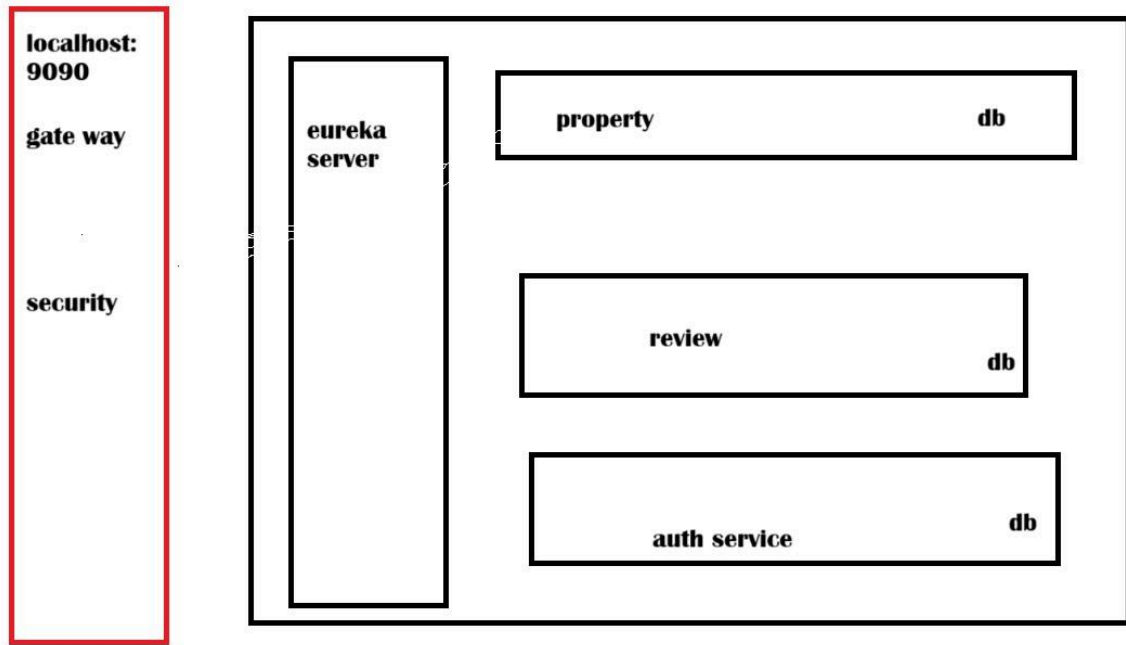
Here we have two urls maybe there are some hundreds in big project so remembering this many will not fusible for front end so we use api gate way

After api gate way : localhost:9090/fromsecondcontroller
localhost:9090/message



Fin client ,rest template and web client all can use the names of the web services that got added to eureka but they cant be accessed through front end so in order

to give uniformity same url we use api gate way



Eureka + fin client == internal server connection will happen

For internal we can use names of microservices in eureka

For client angular or front end we have to use localhost:8080 some thing

PROCESS:

Start project 🎉 add dependency

Reactive gate way

Eureka client

,

Spring dev tools

Add yml to property

```
server:
  port: 5555
  spring:
    application:
      name: api-gateway
  cloud:
    gateway:
      routes:
        - id: microservice-api-1
          uri: lb://MICROSERVICES-1
          predicates:
            - Path=/micro1/**
          filters:
            - RewritePath=/micro1/(?<segment>.*), /${segment}

        - id: microservice-api-3
          uri: lb://MICROSERVICES-3
          predicates:
            - Path=/micro3/**
          filters:
            - RewritePath=/micro3/(?<segment>.*), /${segment}
```

Here's a breakdown of the YAML configuration you provided, typically used in **Spring Cloud Gateway** to define routing rules for a microservice:

yml

CopyEdit

```
- id: microservice-api-1
  uri: lb://MICROSERVICES-1
  predicates:
    - Path=/micro1/**
  filters:
    - RewritePath=/micro1/(?<segment>.*), /${segment}
```

Explanation

✓ id: `microservice-api-1`

- A unique ID for this route definition.
- It helps in identifying this specific route among others.

✓ uri: `lb://MICROSERVICES-1`

- This tells Spring Cloud Gateway to use **load balancing** via a service registry (like Eureka).
- `lb://` means use **LoadBalancerClient** to look up the service named `MICROSERVICES-1`.

✓ predicates:

- These define **when** this route should be matched.
- `Path=/micro1/**` means:
 - Match any HTTP request whose path starts with `/micro1/`.
 - Example matches:
 - `/micro1/users`
 - `/micro1/products/123`

✓ filters:

- Filters modify the request before it reaches the backend.
- `RewritePath=/micro1/(?<segment>.*), /${segment}` means:
 - Use a **regular expression** to capture everything after `/micro1/`.

- The named group (`?<segment>.*`) captures the remaining path.
 - It rewrites `/micro1/something` → `/something` before forwarding it to the backend service.
-

✓ Example

Incoming request:

`GET /micro1/users/5`

Gateway rewrites path to:

`/users/5`

Then forwards to:

`MICROSERVICES-1 (via load balancer)`

Would you like a visual diagram to understand this routing better?

<http://localhost:5555/micro3/fromsecondcontroller>

By using this we can access the microservice 3 urls

Id we can give any thing unique

Lb means load balancer

Api gate way on separate server

Uri must match eureka

Predicate tells you about the path

What is Spring Cloud?

Spring Cloud is a set of tools and frameworks from the **Spring ecosystem** that makes it easier to build and manage **distributed systems** and **microservices** on cloud platforms.

Key Goals of Spring Cloud

- Simplify microservice development
 - Enable cloud-native patterns
 - Handle common distributed system problems like:
 - Service discovery
 - Load balancing
 - Configuration management
 - Fault tolerance
 - API gateway
 - Circuit breakers
-

Common Spring Cloud Components

Module	Purpose
Spring Cloud Config	Centralized external configuration management for all environments
Spring Cloud Netflix	Integrates Netflix OSS tools like Eureka (discovery), Ribbon (LB), etc.
Spring Cloud Gateway	Intelligent API Gateway for routing requests and applying filters
Spring Cloud Circuit Breaker	Adds resilience (like Hystrix or Resilience4J)

Spring Cloud Sleuth	Distributed tracing using Zipkin or Brave
Spring Cloud Bus	Links services via a lightweight message broker for config refresh
Spring Cloud Stream	Messaging abstraction for Apache Kafka, RabbitMQ, etc.

Typical Microservices Architecture with Spring Cloud

arduino

CopyEdit

Client → Gateway → Microservices (via Discovery) → Config Server

↓

Tracing, Circuit Breaker






Spring Cloud Example Use Case

Scenario:

You have three microservices:

- user-service
- order-service
- payment-service

You use Spring Cloud for:

-  **Eureka** to register and discover services
-  **Load Balancing** with Ribbon or Spring Cloud LoadBalancer
-  **API Gateway** to route and rewrite URLs
-  **Config Server** to manage application configs from Git
-  **Circuit Breaker** to prevent cascading failures

✓ Benefits of Using Spring Cloud

- Built on top of **Spring Boot** (easy integration)
- Battle-tested patterns for **cloud-native apps**
- Reduces boilerplate code
- Easily integrates with **cloud providers** (AWS, GCP, etc.)

Would you like a real-world microservice example with Spring Cloud config, Eureka, and Gateway?