c

Contents

[1 Theory 2](#_Toc94232008)

[1.1 Description 2](#_Toc94232009)

[1.2 Analogy 3](#_Toc94232010)

[1.3 Why, Need, Application 3](#_Toc94232011)

[1.4 Disadvantages: 4](#_Toc94232012)

[1.5 API Gateway supports 4](#_Toc94232013)

[1.5.1 Zuul Vs Spring Cloud Gateway 4](#_Toc94232014)

[2 Spring Cloud Gateway Architecture 5](#_Toc94232015)

[2.1 Theory 5](#_Toc94232016)

[2.2 Architecture 5](#_Toc94232017)

[3 How to implement API Gateway 5](#_Toc94232018)

[4 Filters 7](#_Toc94232019)

[4.1 Theory 7](#_Toc94232020)

[4.2 Architecture 8](#_Toc94232021)

[4.3 Implementing Filters 8](#_Toc94232022)

[4.3.1 Java-Based Configuration: In-Built Filters 8](#_Toc94232023)

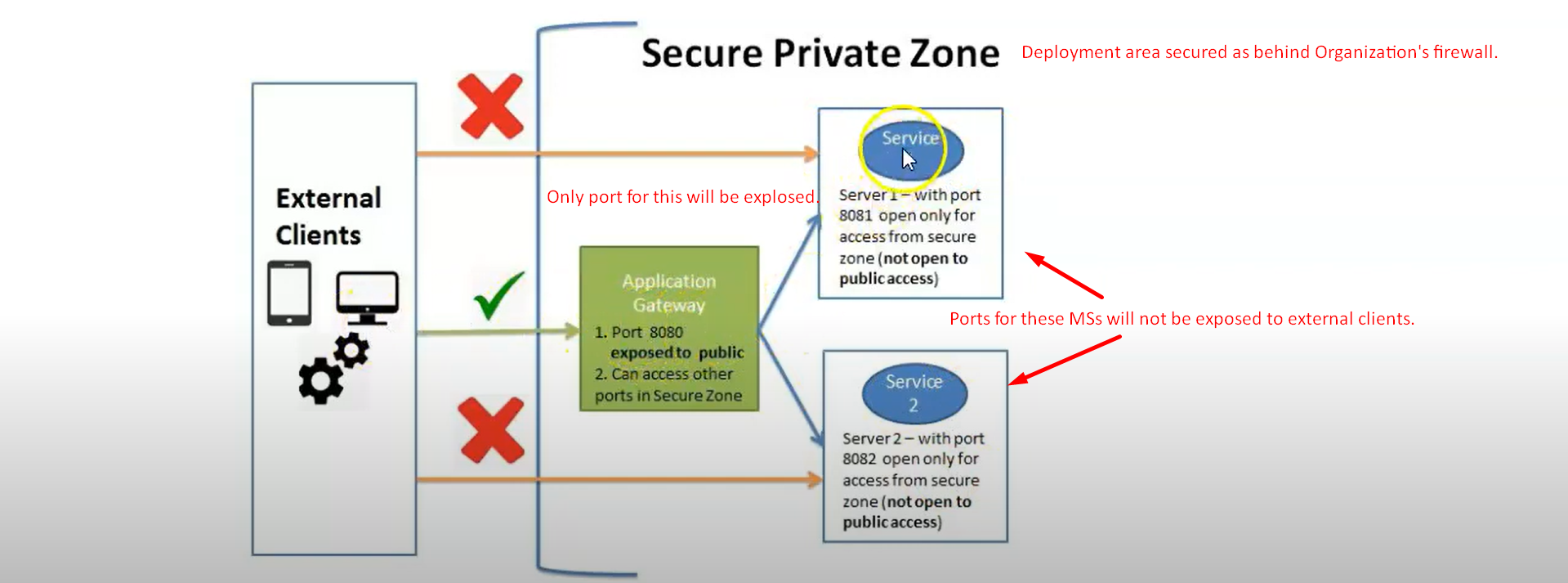
[4.3.2 Property Based Configuration : In-Built Filters 10](#_Toc94232024)

[4.3.3 Customer Filters 10](#_Toc94232025)

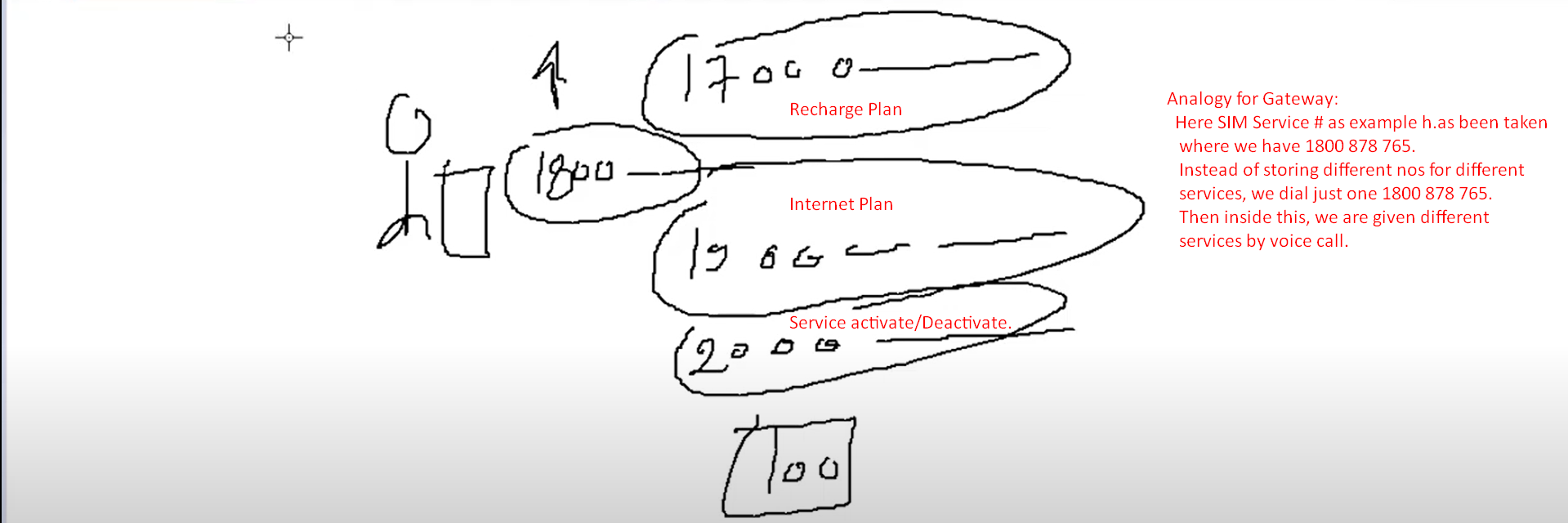
[4.3.4 Global Filter 11](#_Toc94232026)

# Theory

## Description

1. An API Gateway acts as a **single-entry point** for a collection of microservices. Any external client can’t access the ms directly but via **application gateway**.
2. **External Clients**: Categories
   1. Mobile Application.
   2. Desktop Application.
   3. External Services or 3rd party apps.
3. In any organization, there is something called “Secure Private zone” which is an deployment area behind the secured company firewall.   
   Only port for Gateway MS is exposed to outside world but all other MSs are  
   

## Analogy

1. 

## Why, Need, Application

* 1. Common features across all microservices in microservice architecture:
     1. Routing to APIs.
     2. Cross Cutting Concerns:
        1. Security:
           1. Authentication.
           2. Authorization.
        2. Logging.
        3. Rate Limiting.
        4. Monitoring/Metrics.
        5. Resiliency.
        6. Auditing.
        7. Tracing.
     3. Central Policy Enforcement Point (PEP) for all calls.
  2. **Routing to APIs**:
     1. Hundred of ms along with many instances. Some are exposed to outside world (UI, Vendors).Now you have routing requirement such as if receiving so and so path, then route to a particular microservice.
     2. Based on request parameters, you want to do dynamic routing.   
        **For example**: if receiving beta version as request parameter, redirect to a new beta ms which you’re trying to evaluate otherwise redirect to stable ms.
  3. **Cross Cutting Concerns**:
     1. Why handling such concerns in API Gateway in centralized manner rather than asking corresponding developers of a microservice?
        1. 1st each developer has its own way to implement them. So no consistency in implementation.
        2. In worst case, someone developer may not implement security correctly thus insure ms.
     2. You may think that why don’t we create a **common library (Jar)** having crosscutting logic for all MSs.
        1. This way, you’re tightly coupling your MSs with this common library.  
           Such as if you want to introduce a new change in the common library, you have to consider the followings:
           1. What would be the impact on all microservices?
           2. How will my ms react?
           3. There is no **regression issue/bug**.  
              Thus all this will result in monolithic architecture style which we’re trying to avoid with ms.  
              Thus we should have a separate entity/ms handling all these crosscutting concerns.
  4. **Latency**: Suppose one ms hitting two microservices for final results (product details and rates).   
     Gateway hits two MSs then aggregates the result which will result in network latency.   
     Create model view in Gateway so Gateway will hit two microservices, aggregate the result and returns.
  5. **Response Caching**: Instead of caching the response on each microservice, now cache on Gateway.
  6. **Retry/Circuit Breaker**: Gateway will retry, fall back to some other ms. No need to implement this on each ms.
  7. **Load Balancing:**
  8. **Query Transformation**: Adding/modifying request parameters, headers.
  9. **IP Whitelisting:** Listing which Ips are valid to use the services.
  10. **Security:** As exposing only Gateway MS to outside world and all other MSs are behind company’s firewall.
  11. **Abstraction:** Client needs not to know the internal architecture of MSs such as clients need not know the exact port and API URLs.
  12. **Simplified Interaction:** As client needs to access only single MS for all requirements (no need to manage multiple Ips and ports).

## Disadvantages:

1. Single Point of failure.

## API Gateway supports

* 1. **Zuul**: Was used in older versions of Spring Cloud.
     1. As Netflix is not supporting Zuul and is now in maintance mode.
  2. **Spring Cloud Gateway** project: By Spring Cloud.
  3. **Kong/Ambassador/Ocelot/Azure API Management**

### Zuul Vs Spring Cloud Gateway

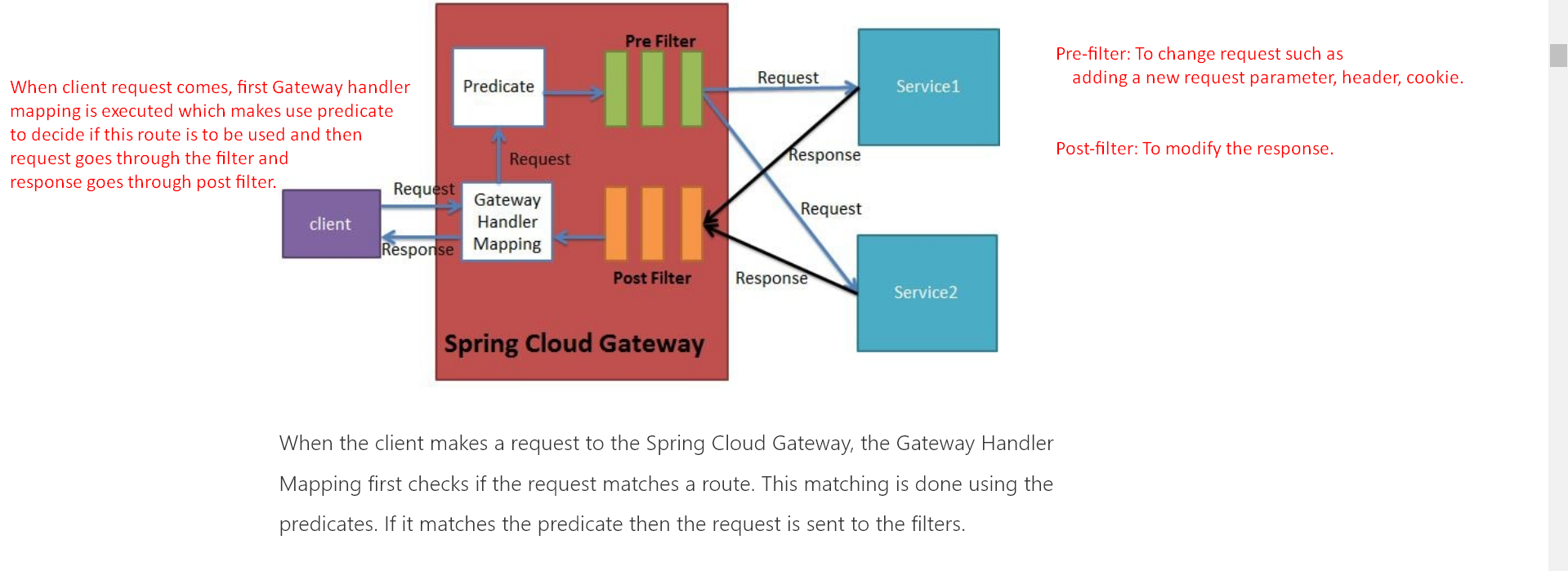
1. **Netflix Zuul** is blocking API which means for each request a new thread is created and if system is out of threads, the reached request has to wait in a queue. So, Netflix Zuul is resource-instensive.
2. **Spring Cloud Gateway** is **non-*blocking API***. When using non-blocking API, a thread is always available to process the incoming request. These requests are then processed **asynchronously** in the background and once completed the response is returned.

# Spring Cloud Gateway Architecture

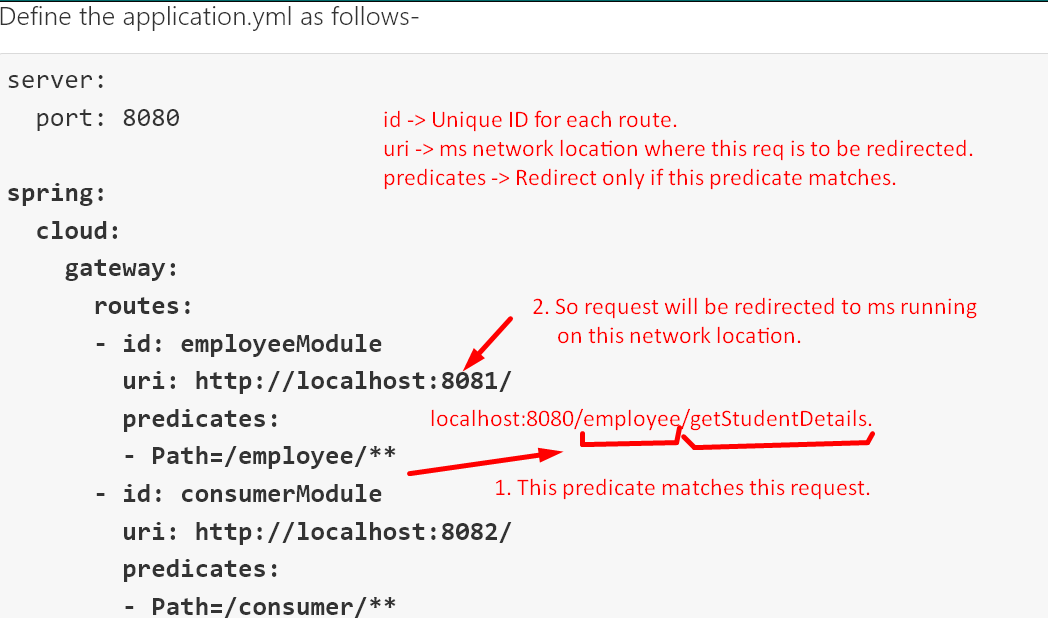
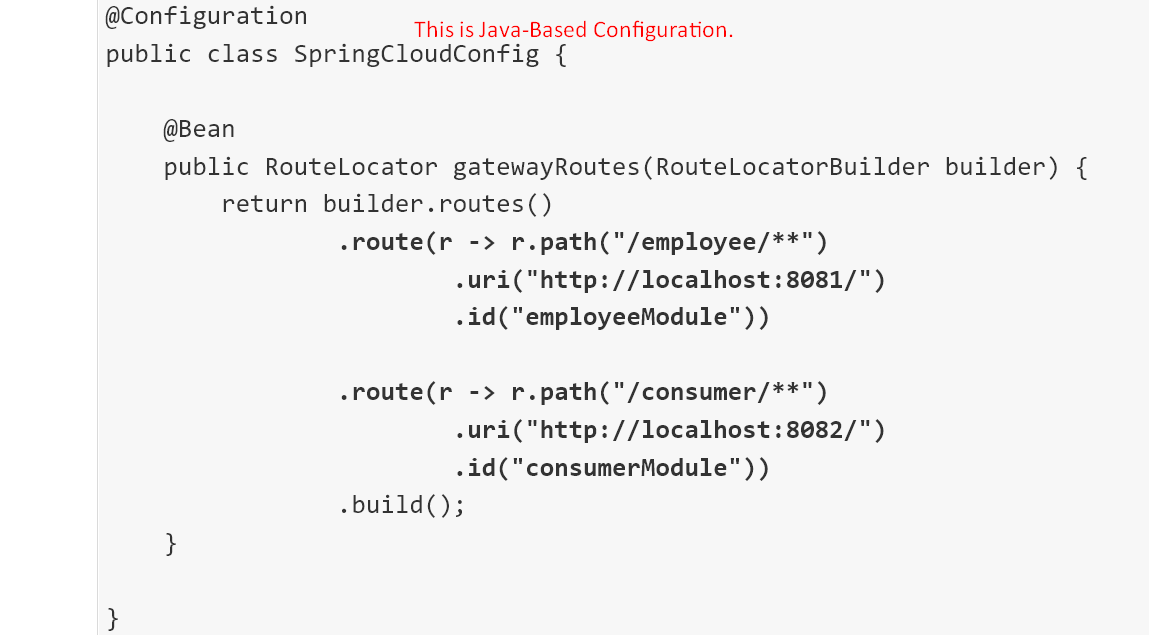
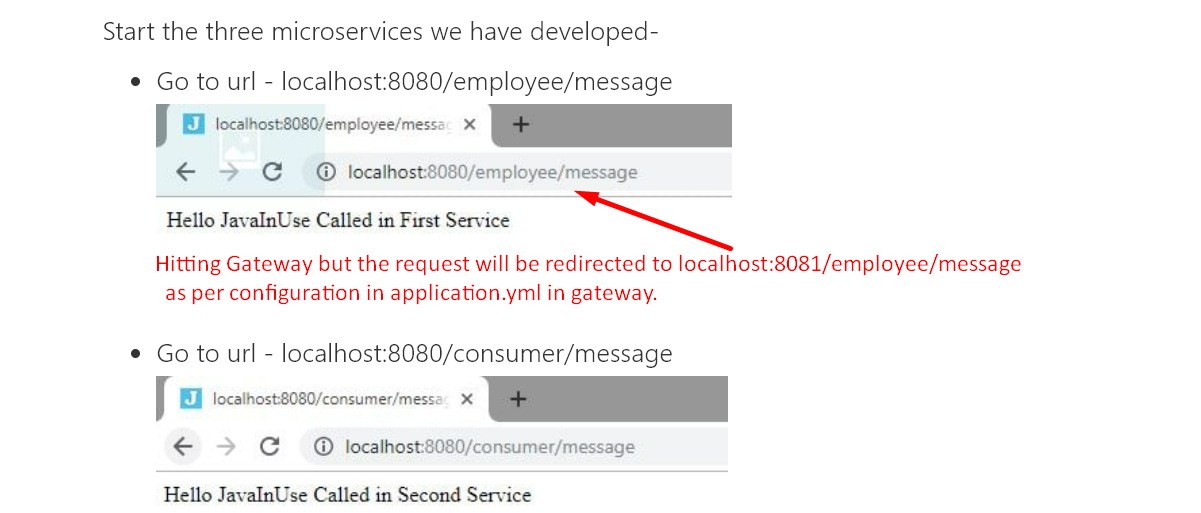
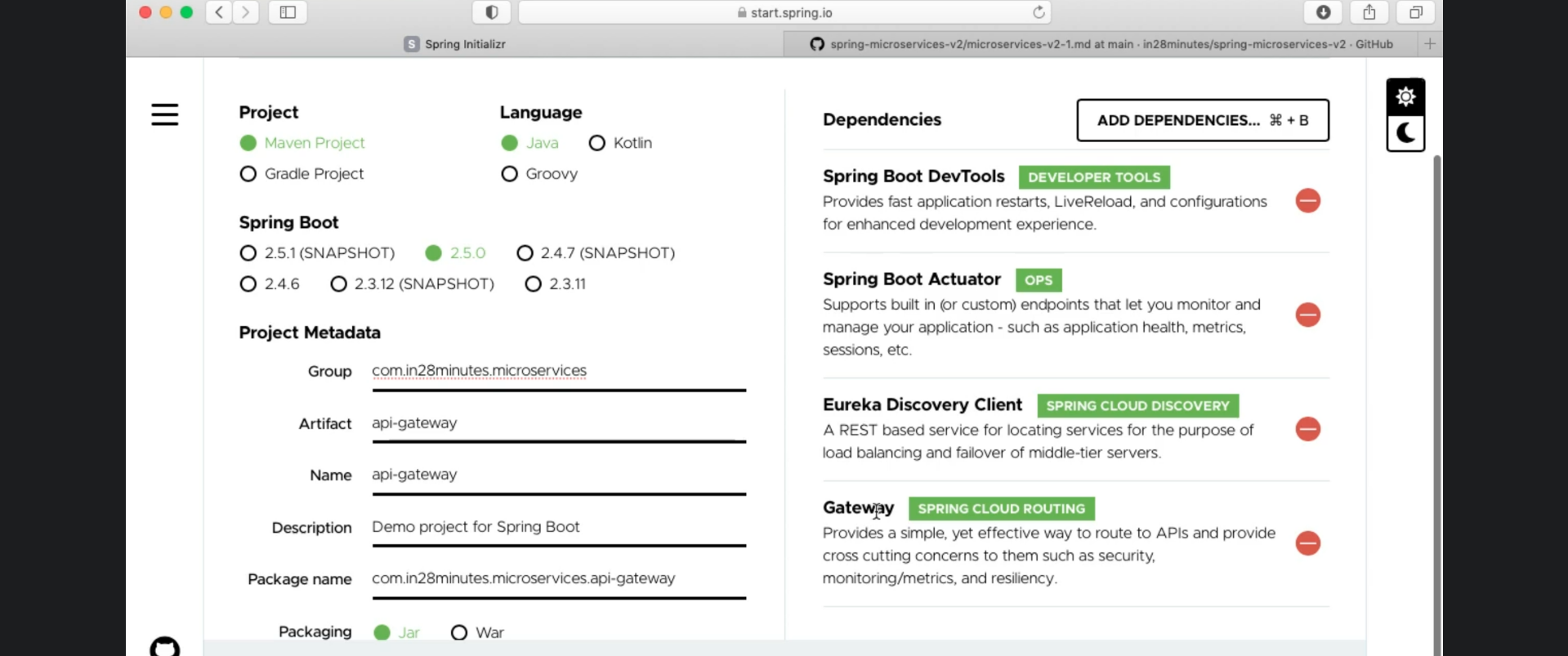
## Theory

* + 1. **Spring Cloud Gateway** is API Gateway Implementation by Spring Cloud Team on top of Spring Reactive ecosystem.  
       **Route**: It consists of the following **building blocks**.
       - 1. **ID**: Unique identification for the route.
         2. **Destination URI**: If aggregate predicates are true, the request is redirected to a particular MS.
         3. **Collection of Predicates**: If aggregate predicates are true, the request is redirected to a particular MS. Conditions can be checked based on headers, URL, cookies or parameters.
         4. **Collection of filters**: To modify the request and response (Query Transformation)

## Architecture

1. 

# How to implement API Gateway

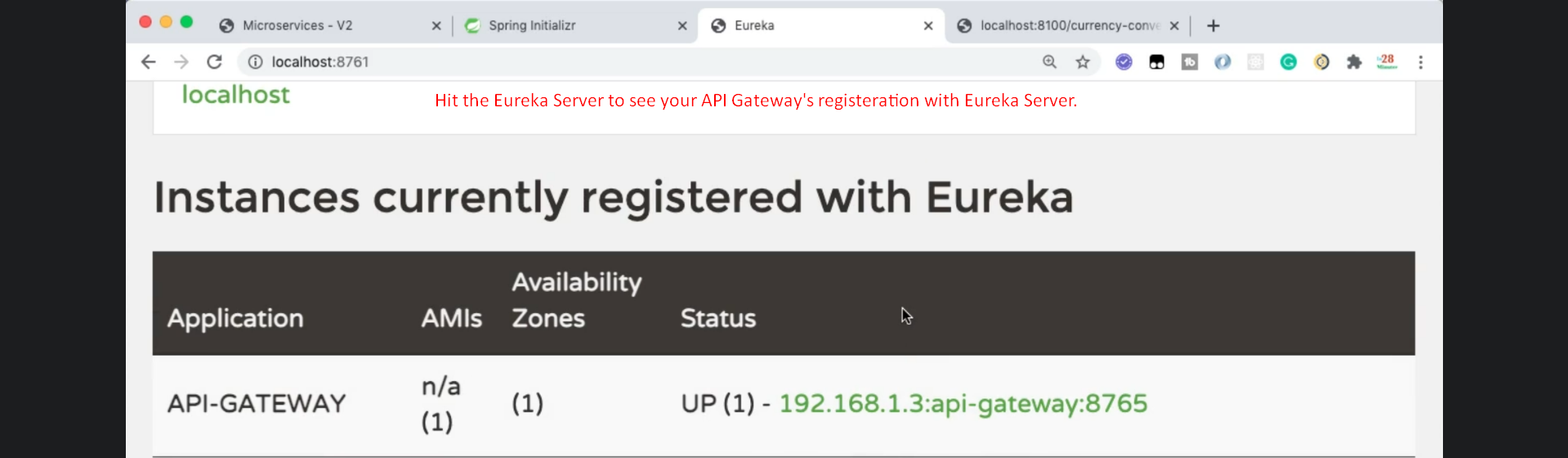
1. **Implementing only Gateway and two service**:   
   \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
   **What we’re going to do**:  
    We will create two microservices m1, m2 exposing one endpoint each.  
    Then we will create Gateway microservice which will have routing configurations for these two microservices.   
    Then through Gateway will make request to our two microservices.  
   \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
   **NOTE**: Just adding Spring Cloud Gateway dependency and adding configurations will make gateway microservice.
   1. **Step 01**: **Add dependencies**
      1. **spring-cloud-starter-gateway dependency** will also include spring web dependency and many more.
   2. **Step 02**: **Configurations in application.yml**.
      1. Using Spring Cloud Gateway, route can be configured in two ways.
         1. **Java-Based Configuration** to programmatically create routes.
         2. **Property Based Configuration** (application.properties or application.yml) to create route.  
             
   3. **Step 03**: Create two microservices m1 and m2 having **/employee/message** and **/customer/message** respectively. we created as the configurations in the application.yml in gateway.
   4. **Step 04**: Hitting Employee microservice via Gateway.  
      
2. **Implementing Gateway along with Eureka Server**.
   1. **Step 03**:
      1. **Adding dependencies**
         1. Let’s talk about dependencies:
            1. **Gateway**:

Spring Cloud Routing Project.

To convert your microservice project into Spring Cloud API Gateway.

* + - * 1. **Eureka Discovery Client**:

Gateway needs registry info about other microservices so that it can route to APIs and do load balancing. So Gateway needs to connect with Eureka Server to fetch registry info.

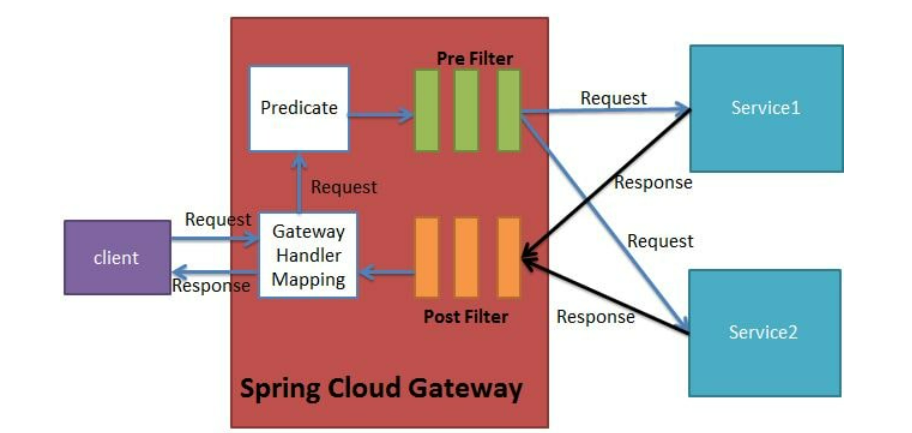
* 1. **Step 02:** Make configurations in application.properties file.
     1. 
        1. eureka.client.serviceUrl.defaultZone:
           1. Optional if Eureka server is running on the same machine as Gateway and on default port (8761).
  2. **Step 03**: Check if Gateway MS is properly registered with Eureka Server.
     1. 
  3. **Step 04**: How to access a microservice using Spring Cloud API Gateway
     1. [http://localhost:8765/<app-name>/path](http://localhost:8765/%3capp-name%3e/path)

# Filters

## Theory

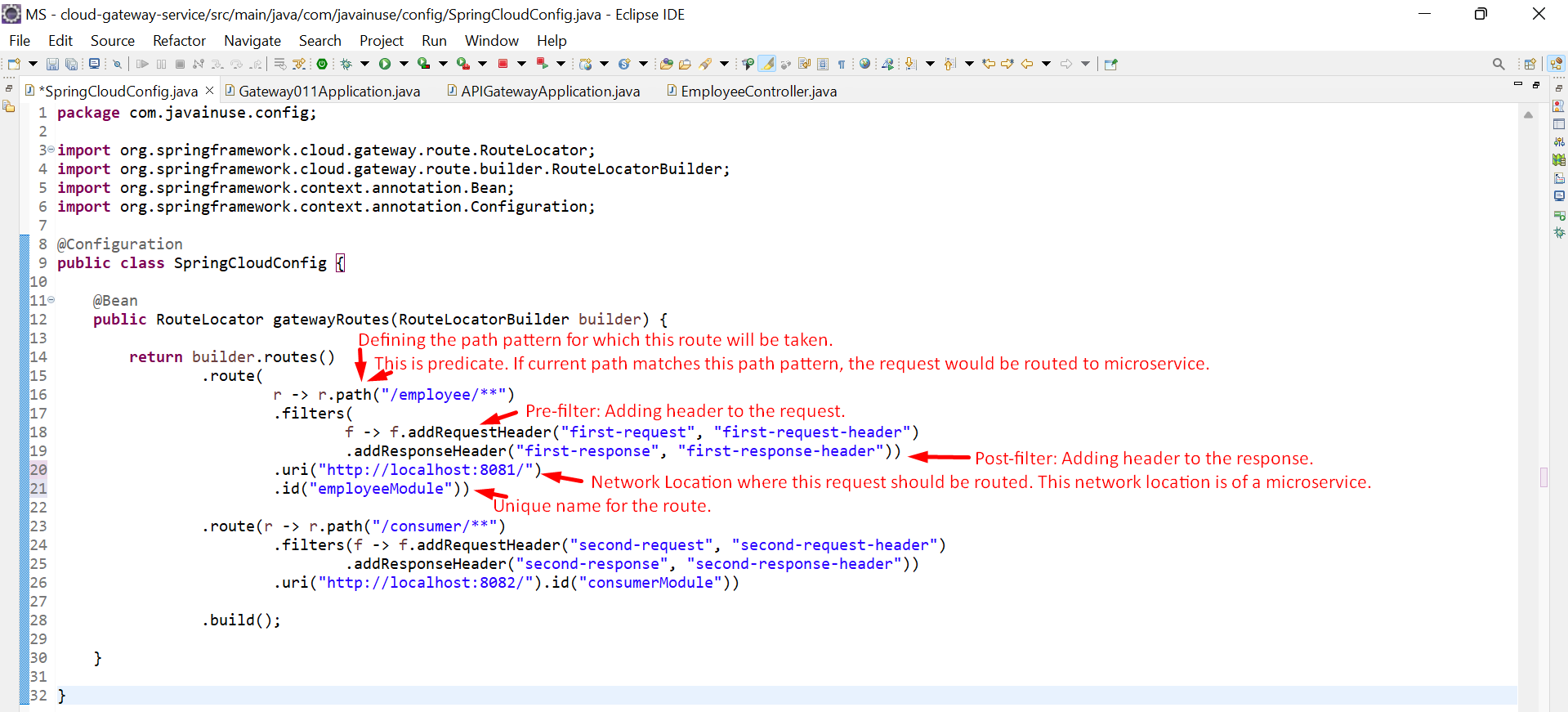
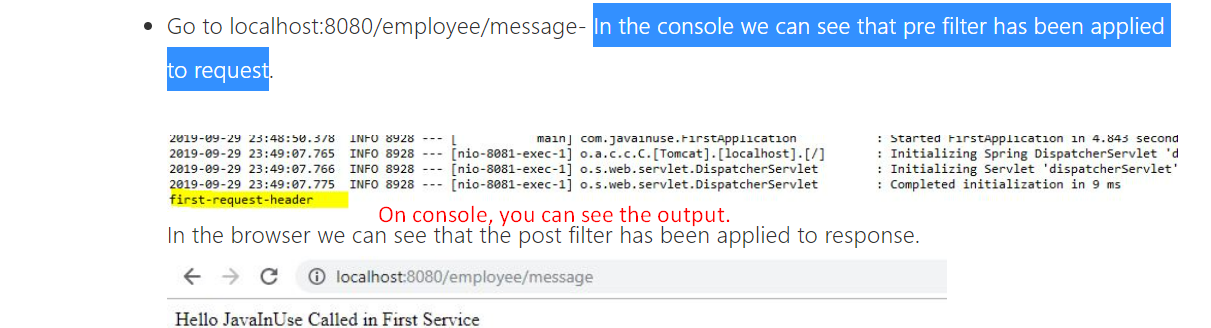
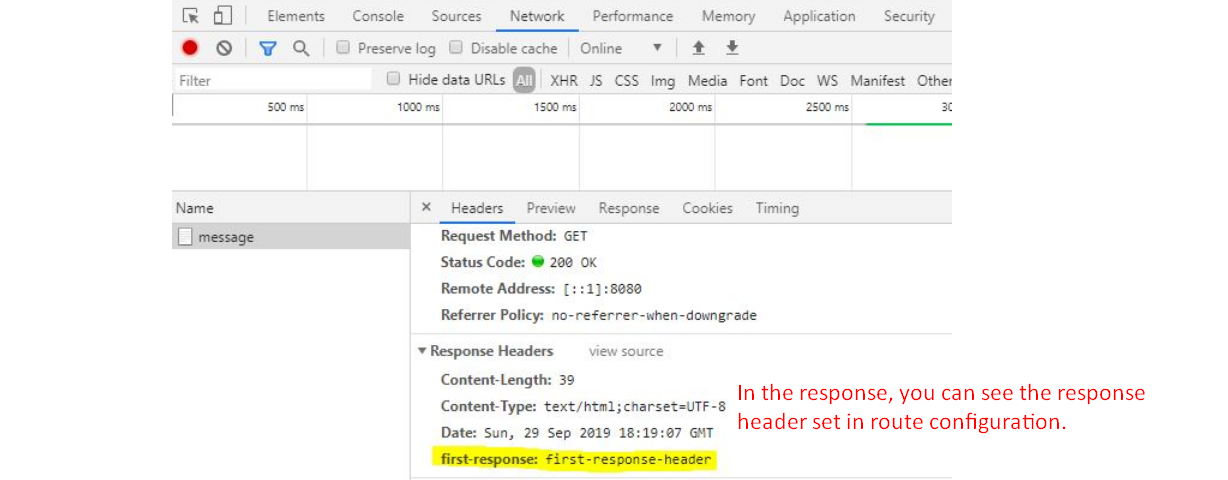
1. Two kinds of filters for Spring Cloud Gateway.
   1. **Built-in Filters**: Spring Cloud Gateway provides built-in filters.
   2. **Custom Filters**: As per our business requirement.
2. Based on **predicates**, Spring Cloud Gateway determines which **route** should get called.  
   Once decided, the request is routed to the intended microservice.
3. We routing the request we can apply some filters.
   1. **pre-filters**: Which are applied to request before routing.
   2. **post-filters**: which are applied to response after receiving response.
4. Two ways to configure filters.
   1. Property Based Configuration.
   2. Java-Based Configuration.

## Architecture

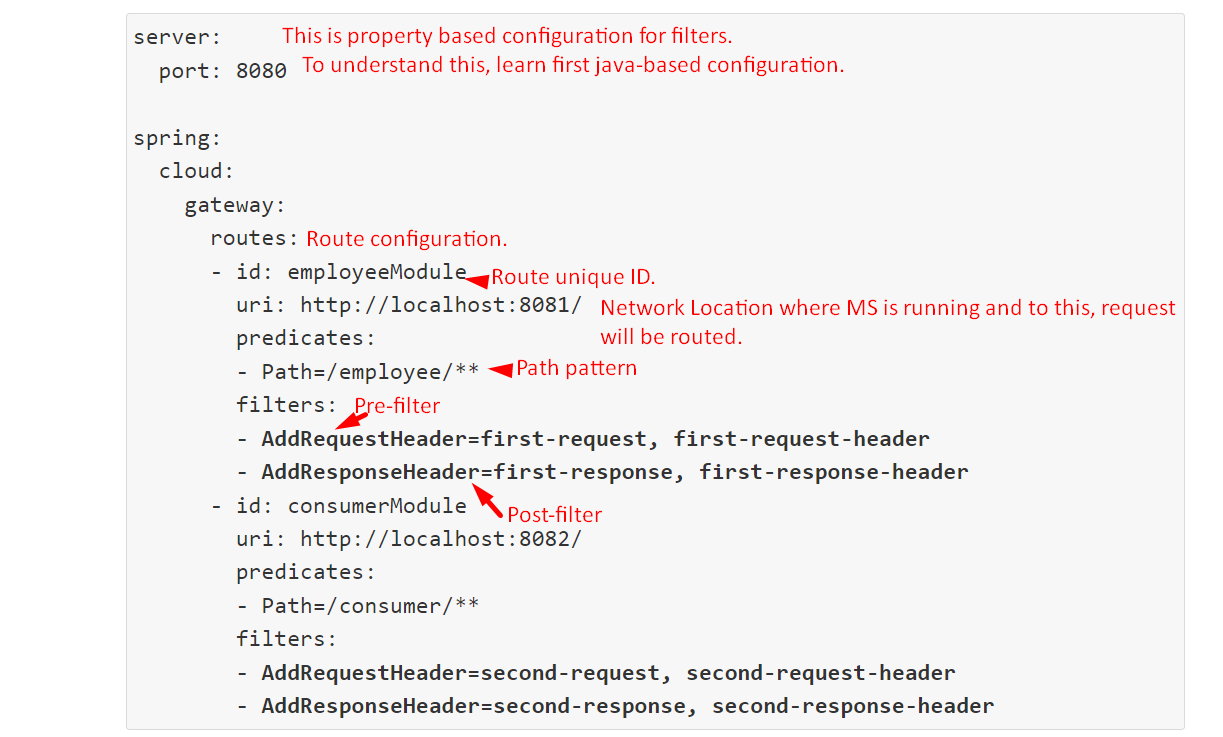
* + 1. 

## Implementing Filters

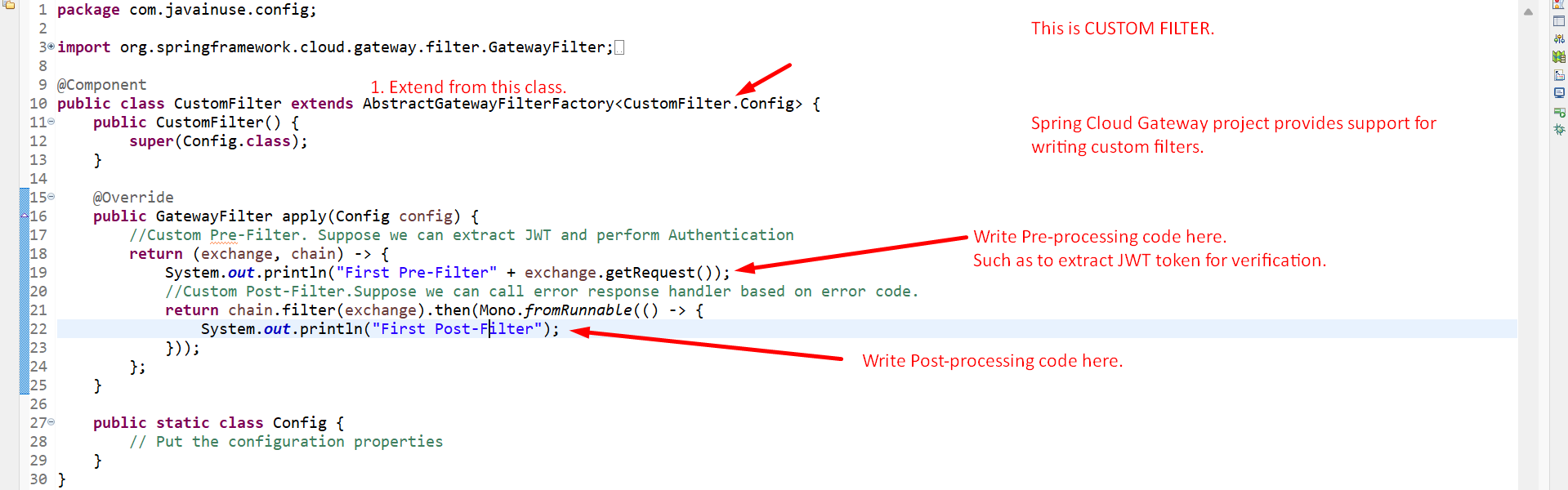
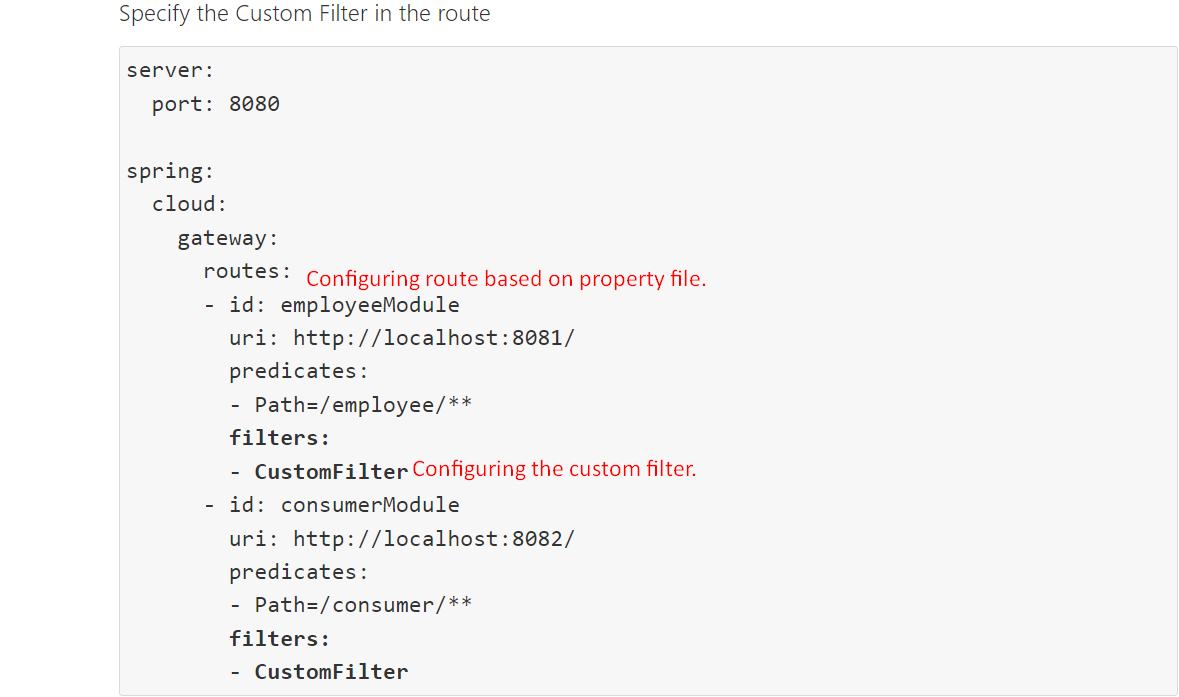
### Java-Based Configuration: In-Built Filters

1. Following is the controller in employee microservice.  
     

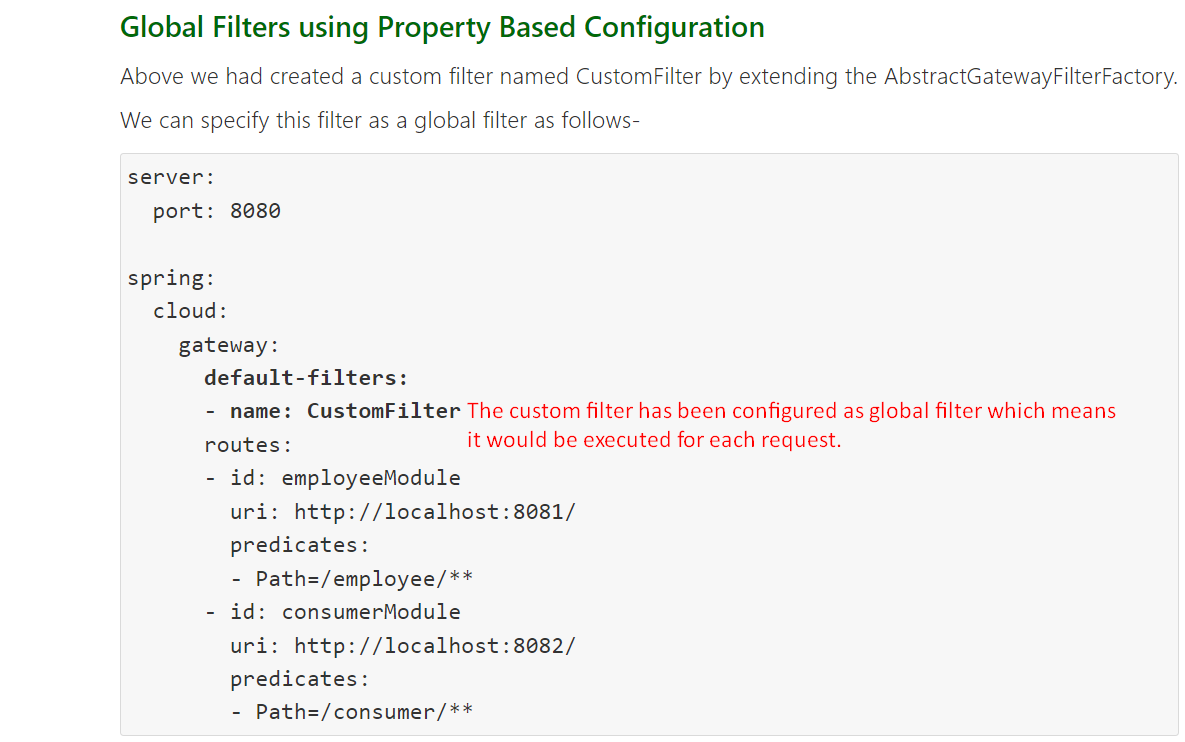
### Property Based Configuration : In-Built Filters

* + 1. 

### Customer Filters

* 1. **Steps**:
     1. **Step 01**: Create Custom Filter
        1. 
     2. **Step 02**: Configure the filter in route configuration.  
        

### Global Filter

1. Property Based Configuration:   
   If it is a custom filter that you want to configure,   
    then first create custom filter,   
    then configure that filter as global filter in yaml file.   
   
2. Java-Based Configuration: If it is custom global filter then just one step.   
   