Contents

[1 Theory 2](#_Toc94459471)

[1.1 Description 2](#_Toc94459472)

[2 Load Balancer, Service Discovery 2](#_Toc94459473)

[2.1 Why not to use traditional Load Balancer in MS? 2](#_Toc94459474)

[2.2 Architecture of Service Discovery 3](#_Toc94459475)

[2.2.1 First Talk 3](#_Toc94459476)

[2.3 Client Side-Load Balancing Architecture 4](#_Toc94459477)

[3 How to implement Eureka Server 5](#_Toc94459478)

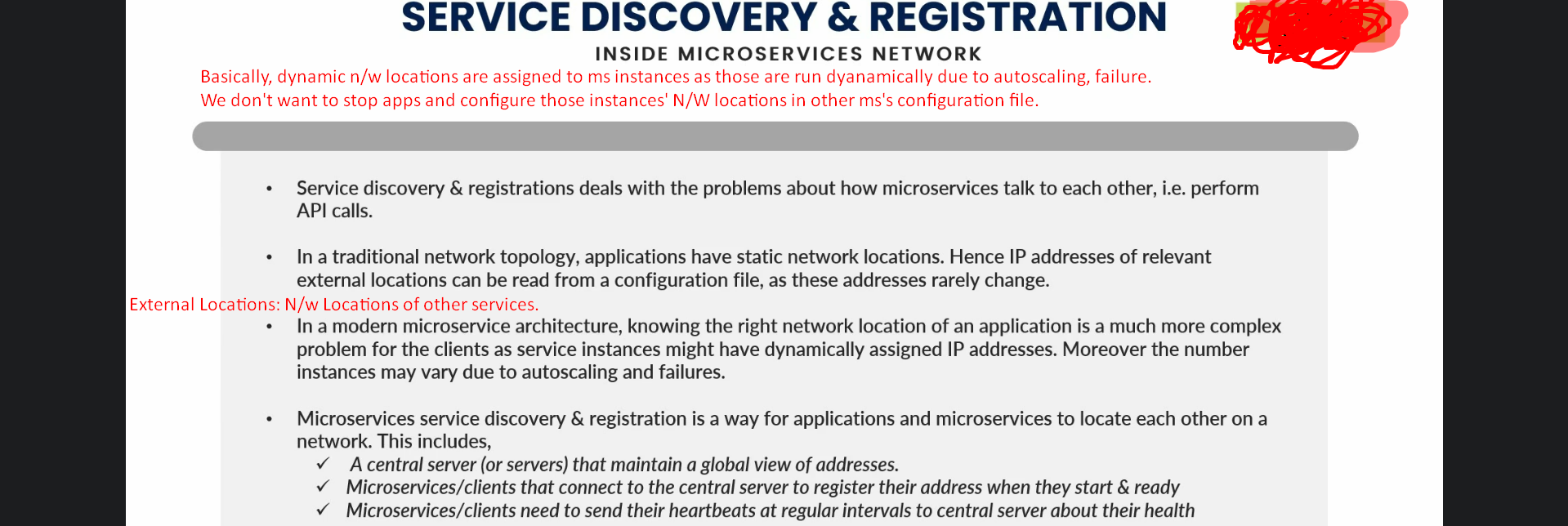
[3.1 A little Theory 5](#_Toc94459479)

[3.2 First Implementation 5](#_Toc94459480)

[3.3 Second Implementation 6](#_Toc94459481)

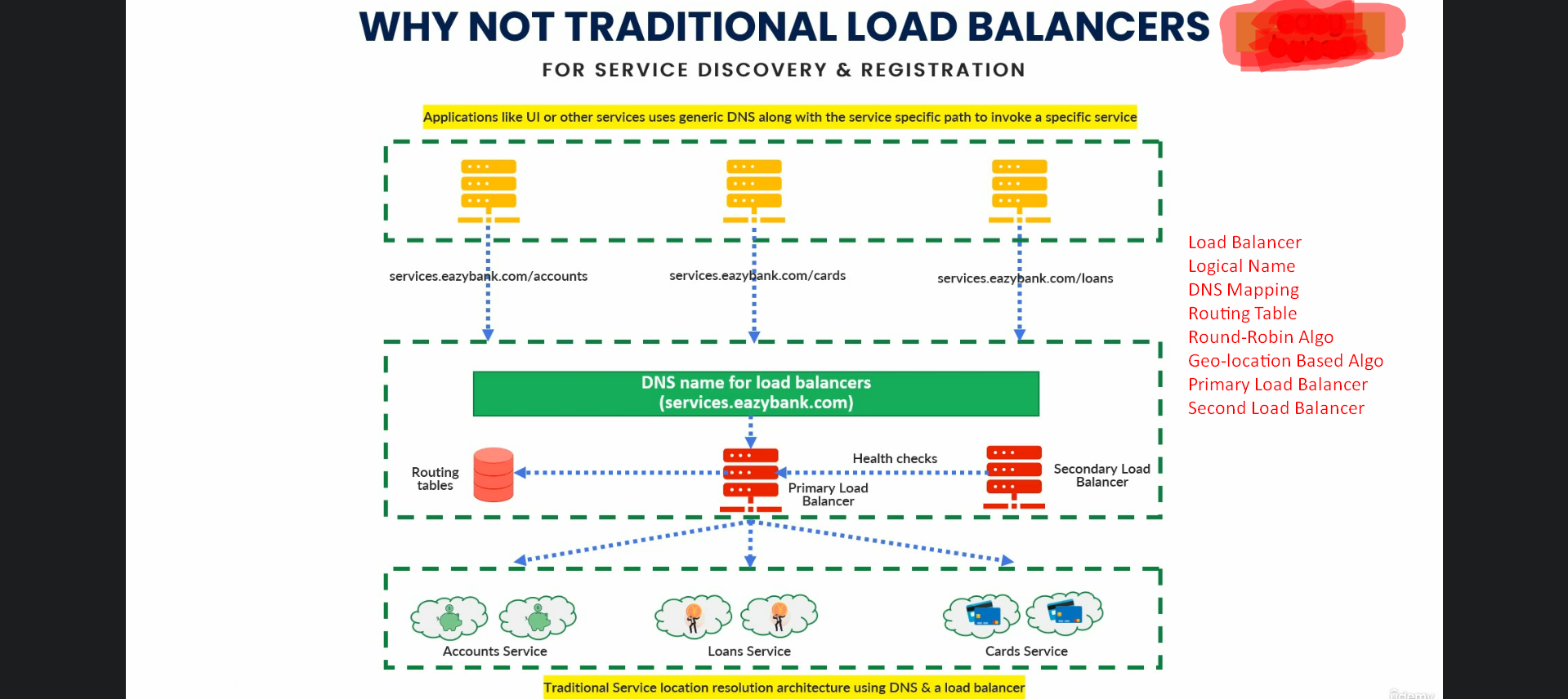
# Theory

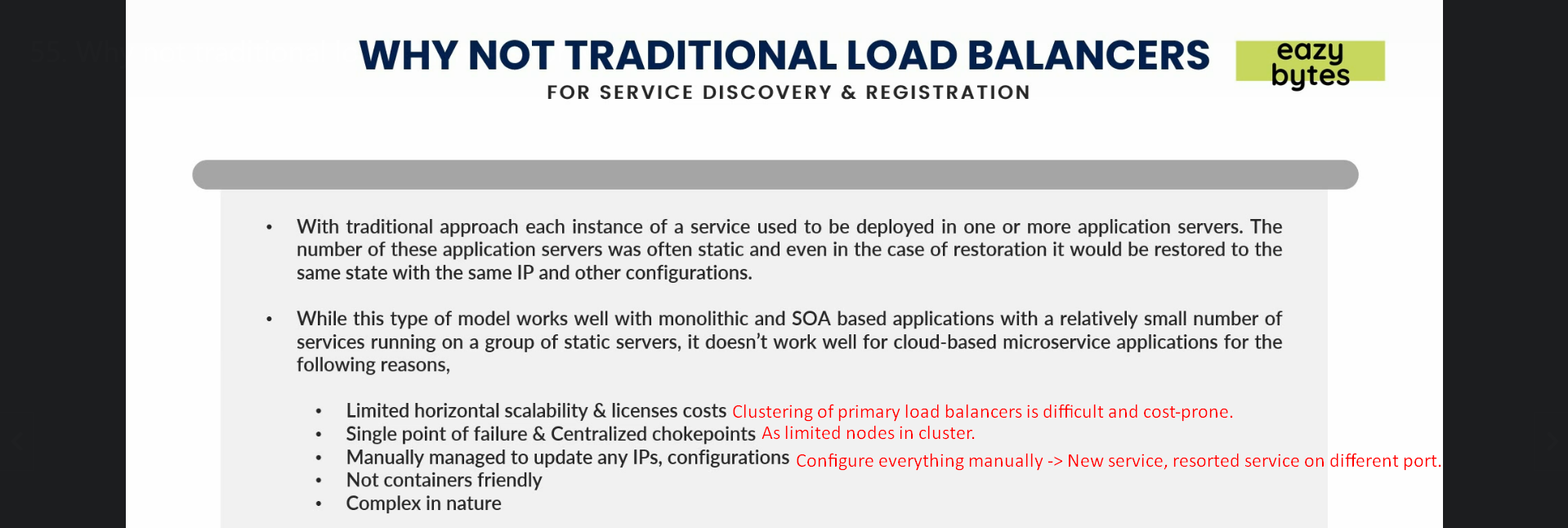
## Description

1. Eureka is Netflix Component for **Service Registry and Discovery**.
2. Why “**Service Discovery and Registration**”?  
   

# Load Balancer, Service Discovery

## Why not to use traditional Load Balancer in MS?



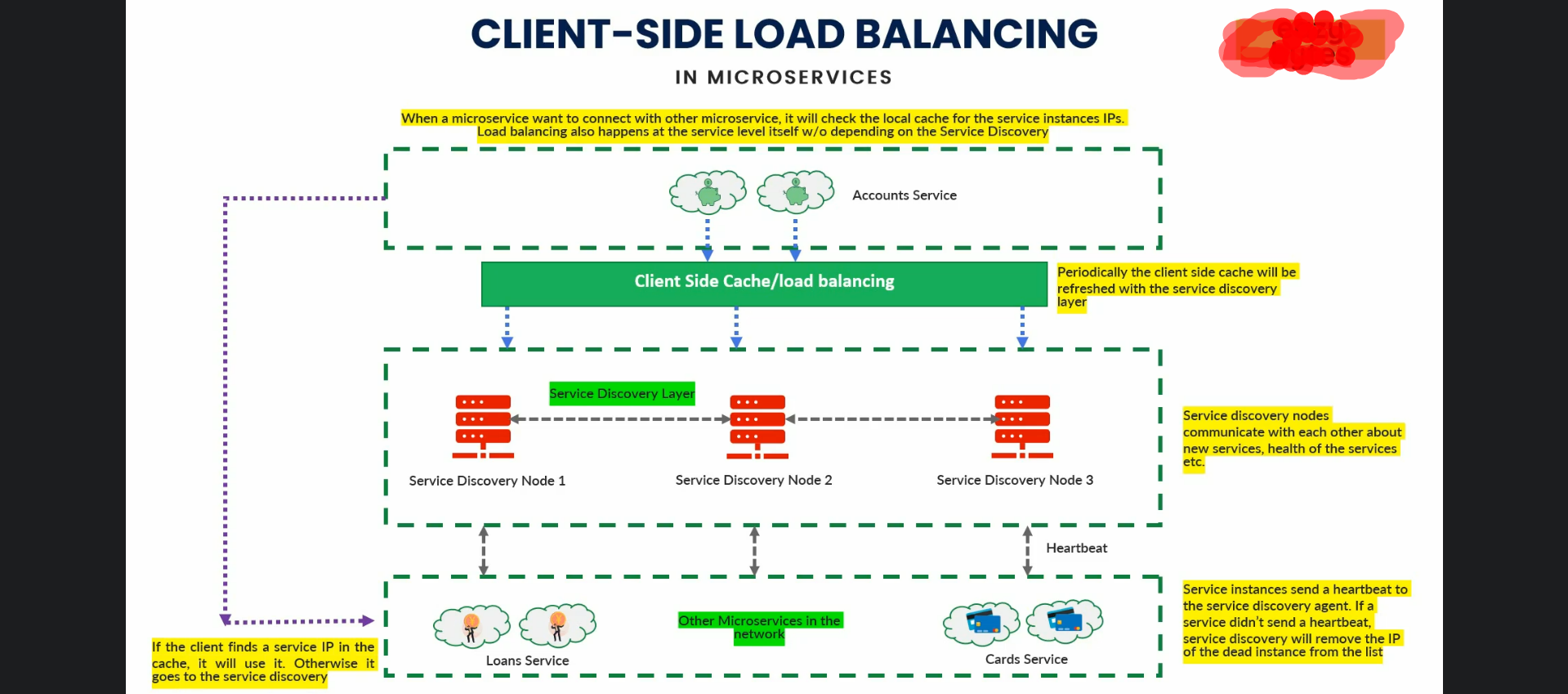
1. In traditional Load Balancer, for reach service (running app), there is a logical name and each other service (UI, Backend app) access other service using this logical name. and inside the Load Balancer, we do DNS Mapping such as **services.eazybank.com/accounts -> localhost:8080**.  
   Routing Table contains this DNS Mapping info.
2. Based on the logical name in the request, Load balancer figures out the N/w locations for all the running instances from Routing Table.
3. Then Load Balancer balances request (means route the req to a particular instance) based on some algo such as Round-Robin or Geolocation-based algo.  
   So, Load Balancer makes the actual call to a particular service.
4. In the figure, you can see a **Primary Load Balancer** where each original request comes so great dependency on PLB. If it is not working, no one can invoke services.
5. **Solution**: **Secondary Load Balancer** in the architecture as you can see in the figure which is in **standby mode** which keeps on sending **health checks** to Primary Load Balancer.  
   If Primary Load Balancer not working, Secondary will take over the responsibility to accept request.
6. **Other UI or Backend apps invoke other services using logical name for two reasons**:
   1. If we expose N/w location, others can hack the apps.
   2. If in future we change N/w for some service, we need to convey this info to our consumers and they have to change their code.
7. **Why we can’t use the same approach with MS architecture?**
   1. In traditional approach, each service is deployed on max one or two servers and if you try to restore a service, you do on the same server. So, no dynamic N/w.   
      This is static approach where we don’t have many instances of a service and their N/ws are fixed.  
      While this approach works fine for monolithic and SOA based apps but will not work for Cloud Based MS.
   2. **Reasons**:  
      
      1. **Limited horizontal scalabilities & license costs**: Limited options to scale horizontally up. Tough and cost involvement in clustering primary load balancers.
      2. **Single point of failure & Centralized chokepoints**: Suppose we go for very big jumbo server for load balancer where a big number of requests are coming which may choke our centralized location and in case of server is not working, it becomes Single-Point of Failure.
      3. **Manually managed to update any Ips, Configuration**: Manual configuration for all the services in Routing Table otherwise Load Balancer will not know about it when new instances are deployed (scale up or down) or already existing ones are deployed on different ports after restoration from a failure or health issues.
      4. **Not Containers friendly**:
   3. So, with all these issues, we will not be using traditional load balancer for registration and discovery of services inside ms architecture.

## Architecture of Service Discovery

### First Talk

1. Service Discovery Pattern resolves the followings:
   1. How a ms registers itself and come into a network topology so that other MSs can reach out to it?
   2. How one ms identifies other services in the same network?
   3. How to do load balancing and info sharing among these MSs?
2. Let’s start with **Service Discovery Layer**.
3. As a developer, I’ve to make sure to build **middle layer** of Service Discovery Layer with agents/nodes.  
   You have to just develop a microservice for node focusing service discovery functionality.
4. Service Discovery Layer maintains the followings:
   1. N/w locations for each microservice.
   2. Info about microservices such as version no, application name.
   3. how many total instances are running for each ms.
5. So create ms for Service Discovery Layer.  
   Then start it.  
   Then start individual services. They will register their N/w Locations along with their logical names with Service Discovery nodes.
6. Service Discovery node will expect heartbeat after a particular interval of time from each registered ms. If not receiving, it will deregister that ms.  
   Time Interval: 30sec.   
   Life Lines: 3times.   
   So, it will wait for 90sec in total. Afterwards, it will deregister.
7. When a particular node in Service Discovery receives N/w locations, logical name, app info, heartbeats, it will share with other nodes by peer to peer communication. Protocol used 🡺 Gossip/infection protocol.
8. How a ms calls other

## Client Side-Load Balancing Architecture



1. There is no change in Service Discovery architecture (2nd and 3rd Layers) when talking about client-side load balancing.
2. We have to write logic b/w the Service Discovery Layer and Client so where we cache and do the load balancing on the client side.  
   **Client**: In our network topology, suppose we have 10 microservices and each microservice is client to all other microservices.
3. Let’s see what would be procedure when one service tries to connect with other service.  
   Suppose account service trying to access loan service.
   1. As this is very first time, so account service tries to connect first with Service Discovery to **fetch and cache** all the details about loan service such as how many instances are running, their N/w locations, and app info such as app version, name etc.  
      Then account service hits the loan service.
   2. 2nd time when account tries to hit the loan service, it uses its cache.  
      **NOTE**: It also does load balancing using some algo such as Round-Robin, Demographic etc.   
      NOTE: Now Service Discovery Layer doesn’t have much load as load balancing is being done by client.  
      **NOTE**: We will use Spring Cloud Load Balancer.   
      **NOTE**: Netflix Ribbon is deprecated.
   3. **Question**: **What if a new instance is introduced or already running instance is down**?

**Answer**: Client App periodically refreshes the client-side cache by connecting with Service Discovery Layer. Maybe after 2-5min.

* 1. **Question**: **What if account service hits the loan service during that period (2-5min) and during that period, that particular load instance is down which it is going to hit**?  
     **Answer**: Definitely, the account service will get exception or timeout but it will not thrown an error/exception immediately rather it will refresh its cache and again hits the loan service
  2. **Question**: **So total possibilities when a client service refreshes its cache?**
  3. **Answer**: First periodically. Second when getting exception due to instance is down.

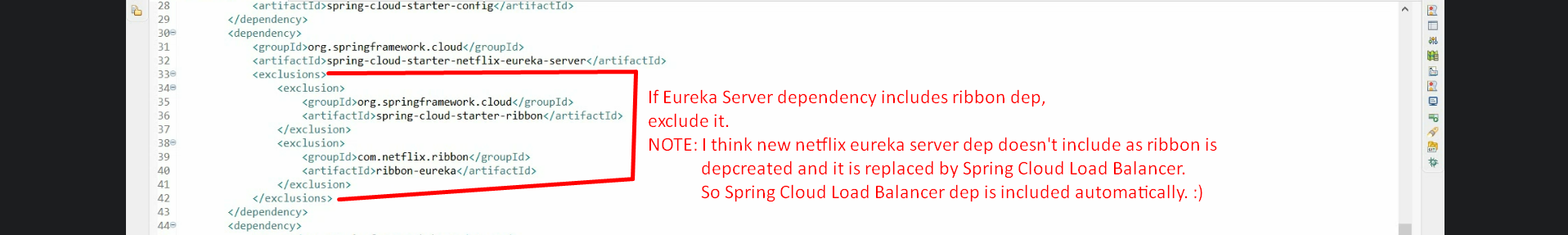
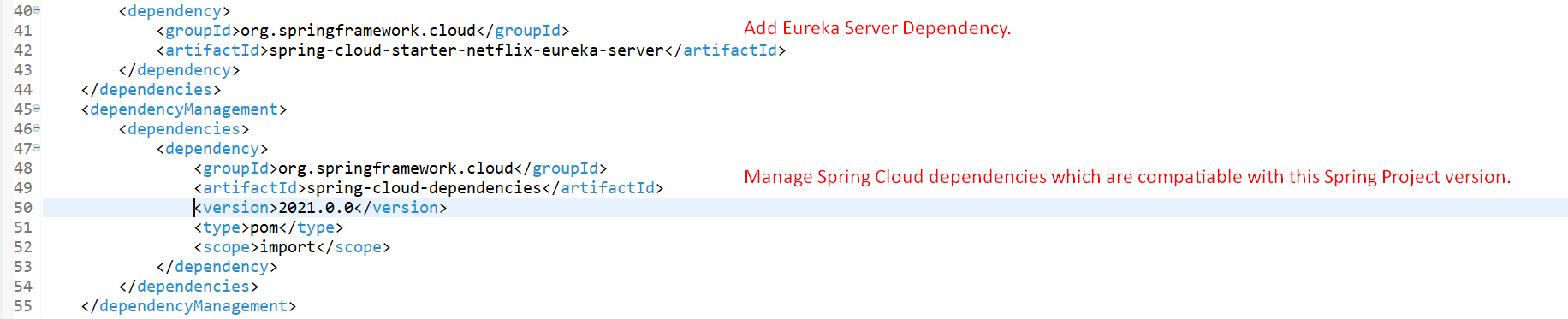
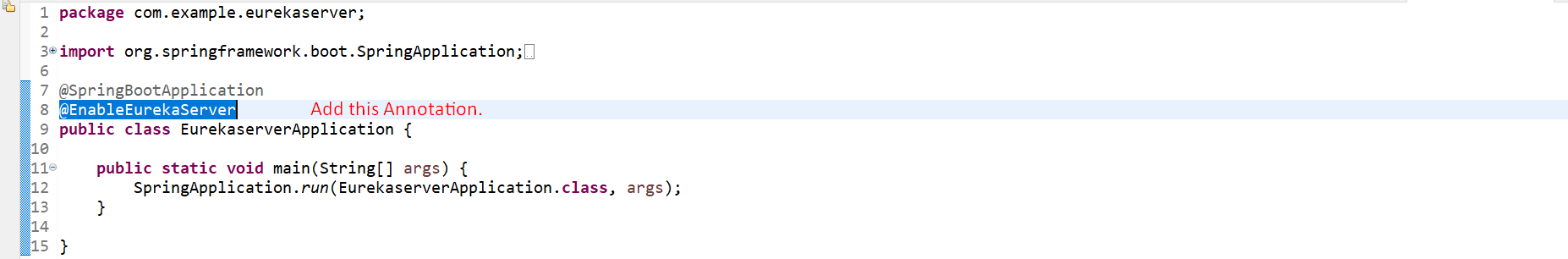
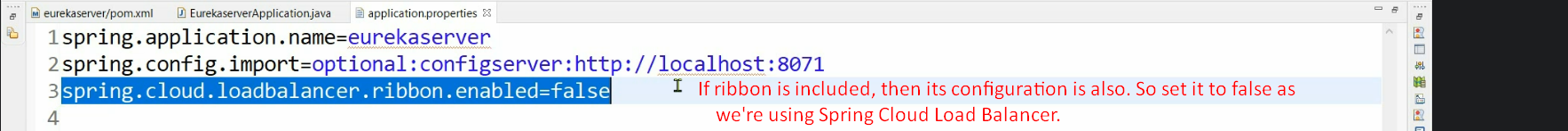
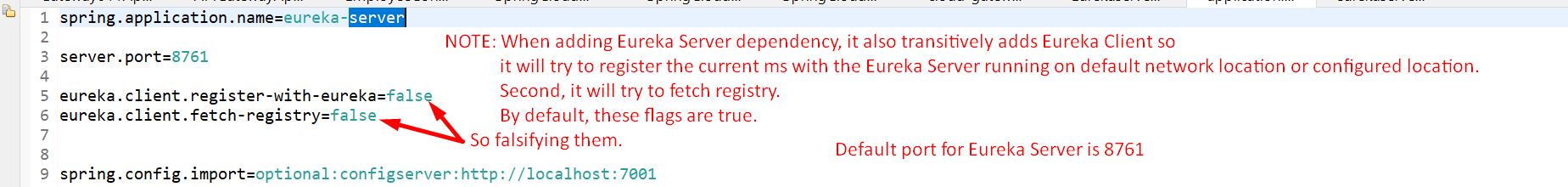
# 

# How to implement Eureka Server

## A little Theory

1. 
   1. **Spring Cloud Netflix’s Eureka Service**: It will acts as **Service Discovery Agent** also called **Eureka Server** which maintains all the **registry details** inside your network in a centralized manner.  
      This project was by Netflix and later on handed over to Spring Cloud to open source so that others can benefit from it.  
      It is **production battle tested** by many companies and stable and perfect solution to implement Service Discovery Agent/ Eureka Server.

## First Implementation

1. **Step 01: Add Netflix Eureka dependency** **Take care of the dependency management using <dependencyManagement> for spring cloud**
2. **Step 02**: **Enable** EurekaServer using annotation @EnableEurekaServer  
   
3. **Step 03:** Configure Properties in application.properties  
     
   The following property specifies URL on which Eureka Server is running.   
   I don’t have any idea about the following property.  
   eureka.client.serviceUrl.defaultZone=http://localhost:8090/eureka
4. **Step 04:** Run the app and hit the URL  
   