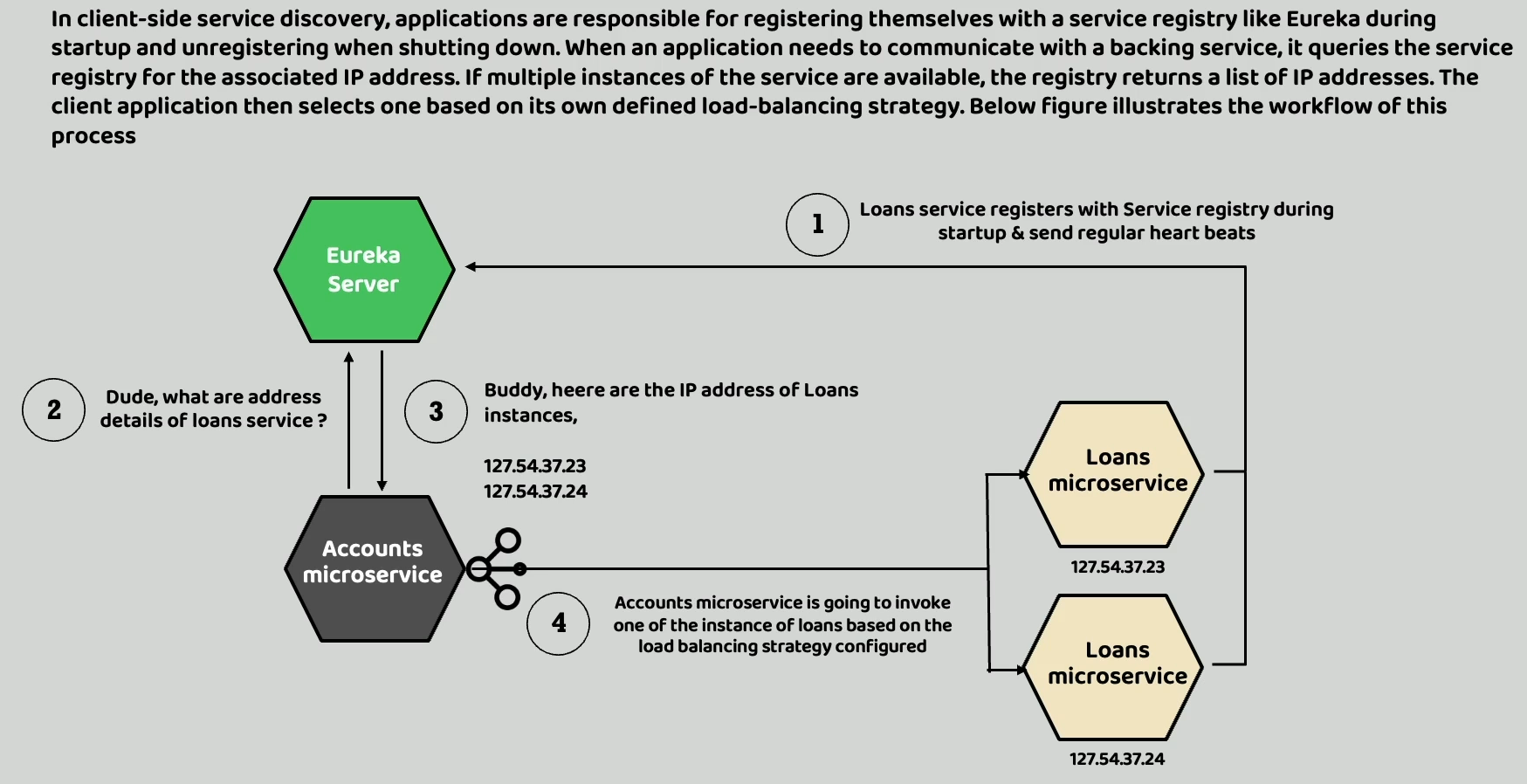
Introduction to Service-Side service discovery and load balancing



Long back when we are discussing about Eureka Server, we discussed about client-side discovery and

the load balancing that is going to happen at the client side.

Inside this approach, all the microservices inside our microservice network, they are responsible

for registering themselves with the service registry like Eureka during the startup.

And similarly, they should also unregister whenever they are shutting down.

And apart from registering and unregistering my microservices, they should also send a regular heartbeat to the Eureka Server to prove their healthy status.

In this kind of setup, our environment. whenever a microservice wants to communicate with another microservice within your microservice network, then in such scenarios the microservice, which is trying to communicate with other microservice, it must first connect with the Eureka Server to get the all the running instances of other microservice.

Think like my accounts microservice want to communicate with the loans microservice. In this scenario my accounts microservice will ask the Eureka server what are the details of the loan service.

How my Eureka server will know the details of the loans Microservice because the **2 loans microservice instances** during the startup, they will register themselves with the Eureka server along with their details like host name, port number and other instance details.

So, this is going to happen in the step one like you can see now in the **Step 1** where my accounts microservice is trying to ask Eureka Server like please give me the details of the loan service as a response to the **Step 2** inside the **Step 3**, my Eureka server is going to give you all the IP address or the instance details of the loans microservice and think like my Eureka server has given two IP address details where my loans microservice is available.

So, with these two instance details of loans microservice, my accounts microservice is going to have

a problem.

The problem is it must decide which instance, it must forward the request.

So that is why it is going to do the load balancing with the help of spring cloud load balancer and after the process of load balancing, it is going to choose one of the instance details and to the same the request is going to be sent in the **Step 4**.

So, inside this approach you can see the client microservice or the accounts microservice is itself is responsible for the load balancing.

That is why we call this approach as client-side load balancing and client-side service discovery.

We can follow this approach, but this approach has some disadvantages.

The disadvantage is the developers they must manually maintain the Eureka server.

They must create a spring boot application.

They must convert that as a Eureka server post that they should also make changes in all the microservices to connect with the Eureka Server, to register their details with the Eureka Server.

So, they must make some good number of configurations.

So, this extra burden on the developers can be avoided if we follow the server-side service discovery

and server-side load balancing.

But this approach has an advantage which is with the help of client-side load balancing the client application is going to have a complete control on the load balancing.

Inside Spring Cloud Load Balancer, there are many strategies that the client applications can follow to perform the load balancing, whereas if you go with the server-side service discovery and server-side load balancing, you are not going to have any control on the load balancing.

But the server-side discovery and load balancing are going to provide an advantage, which is you do not have to make any changes inside your microservices.

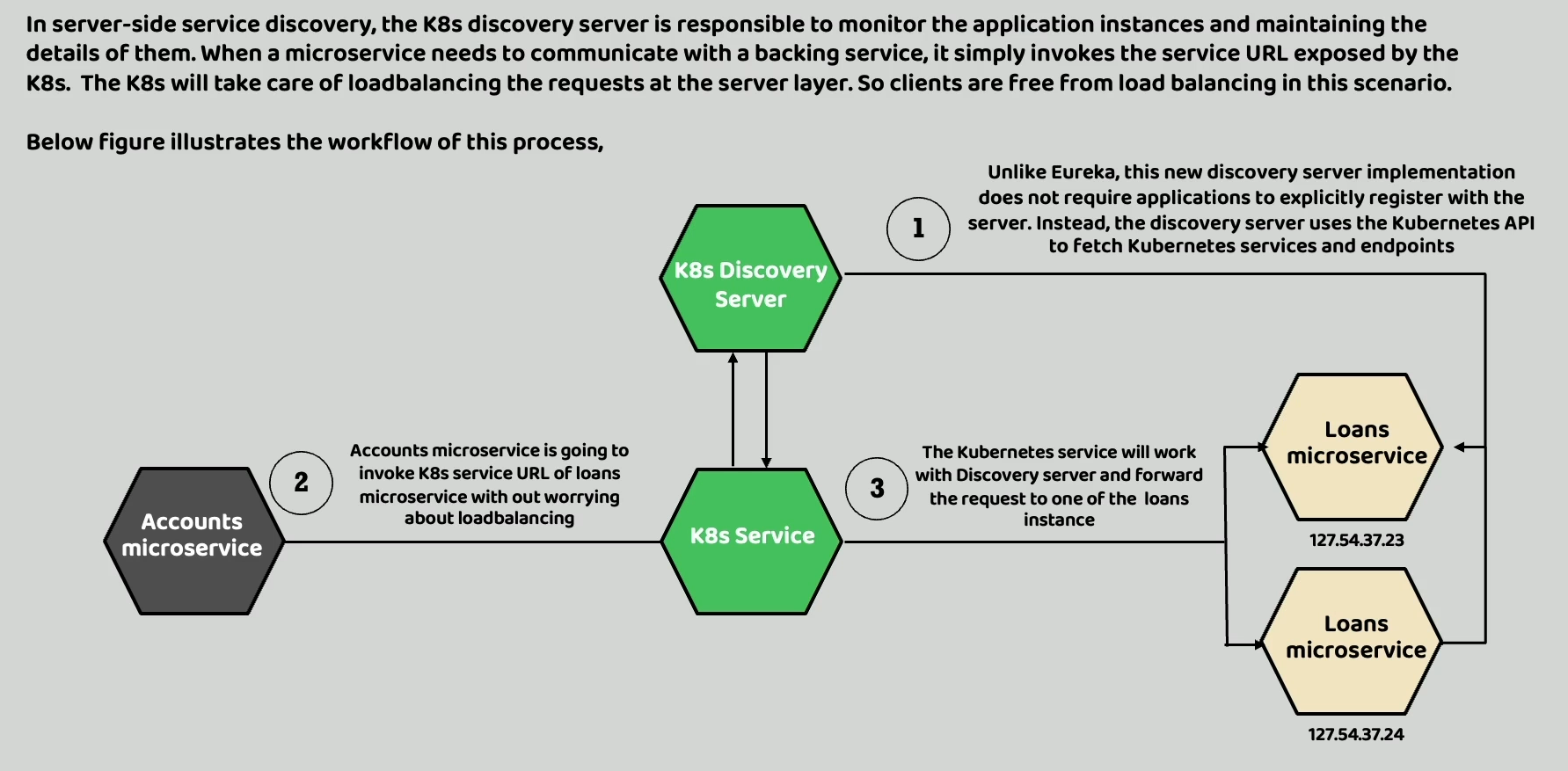
They do not have to register themselves with the service discovery like Eureka Server.

So, let us try to understand more details about the server-side service discovery and the load balancing.

We can follow the server-side service discovery and load balancing only if we are using Kubernetes cluster for our microservices deployment.

If you are not deploying your microservices into Kubernetes, then you cannot use this approach.

Inside this approach, there will be a service discovery inside your Kubernetes cluster, which is responsible to monitor all the application instances and maintaining the details of them.



Like you can see here in the **Step 1**, my Kubernetes discovery server, unlike Eureka, this server does not require the applications to explicitly register with the server.

Instead, the Discovery server itself uses the Kubernetes APIs to fetch details about the Kubernetes

services and endpoints of all the microservices.

So, inside this scenario I have given loans example.

So, my Kubernetes Discovery server is going to query Kubernetes API to fetch all the details of the loans microservice instances.

Now coming to the client-side story like think like the accounts microservice want to communicate with the loans microservice in this scenario.

So, whenever we are following server-side service discovery and load balancing the client application which is the accounts microservice, they do not have to connect with any discovery server.

They can simply send the request to the Kubernetes service using the service name at the host name or DNS name exposed by the Kubernetes.

Behind the scenes, this Kubernetes service, which you have created with the help of **cluster or load**

**balancer service type**s, it is going to work with the Discovery server.

Both the Discovery Server and Kubernetes service, they are going to decide on the load balancing and choose one of the instances of loans microservice to which the request from the accounts microservice is going to be forwarded.

So here it is very clear that the load balancing is going to happen at the server side, which is inside

the Kubernetes cluster and the client application at the client service, which is accounts microservice, it does not bother about any load balancing and it don't have to register or it don't have to connect with any discovery server like we followed in the Eureka Server approach.

So, this approach has some advantages and disadvantages.

The advantage is the developers are freed from all the maintenance of Eureka Server manually and at the same time making configuration changes or establishing connection between all the microservices with the Eureka Server.

So, all those changes the developer do not have to do because right now the responsibility will be with the Discovery server to fetch the details of the running microservices instances.

So that is one of the advantages that we have coming to the disadvantage the only disadvantage with this approach is, the client application or the developers will not have any control on the load balancing.

We as developers are the client microservice.

We can simply forward the request to the service URL and the load balancing strategy

or algorithm is completely decided by the Kubernetes cluster, so I hope you are clear about the server-side service discovery and load balancing.

Inside this section, I am going to show you how to establish server-side service discovery and load

balancer inside our Microservice network.

In the same process, we can get rid of Eureka Server inside our microservice network.

So, there are no good and bad approaches.

How to setup discovery server in K8s cluster using spring cloud Kubernetes

To get started with the server-side service discovery and load balancing inside Kubernetes cluster.

We need to take help from one of the spring cloud projects.

The project is Spring Cloud Kubernetes.

<https://spring.io/projects/spring-cloud-kubernetes>

So, inside this page you should be able to see a project with the name Spring Cloud Kubernetes.

You can click on that.

And this is the project which is going to help you to set up Discovery Server whenever you are using

Kubernetes inside your production deployments.

We are going to use libraries from this project only to set up the server-side service discovery and

load balancing.

By default, your Kubernetes cluster is not going to have any service discovery and service registration server.

So, it is our responsibility to set up some service Discovery server inside Kubernetes, just like how

we did for Eureka Server.

So, to get started around this, there is a blog which is written by the Spring Cloud Kubernetes team

itself.

They have written this somewhere in 2021 when they first released these service discovery and registration capabilities inside this spring Cloud Kubernetes project.

<https://spring.io/blog/2021/10/26/new-features-for-spring-cloud-kubernetes-in-spring-cloud-2021-0-0-m3>

Inside this blog, they clearly highlighted what are the steps that we need to follow to set up that

Discovery server inside Kubernetes cluster.

So here you can see they have given a Kubernetes manifest file that we can use to set up this Discovery server inside the Kubernetes cluster.

So, I am going to use the same and I am going to make few modifications to this manifest file before I

try to install these inside my local Kubernetes cluster.

Inside this Kubernetes folder, I am going to create a Kubernetes manifest file that is going to help

us to set up the Discovery server inside Kubernetes cluster.

https://github.com/eazybytes/microservices/blob/3.2.3/section\_17/kubernetes/kubernetes-discoveryserver.yml

Let me create the new file by going to the terminal.

\eazybank-microservice-application-k8s-helm\kubernetes\kubernetes-discoveryserver.yml

But if you try to install this Kubernetes manifest file, you may face some issues.

The reason is by default, your readiness probe and liveness probe they are going to check for these health details within a very short period, like 10s or 20s.

Within that short period of time, if you are Discovery server is not started, your Kubernetes is going

to attempt a restart and every time it will try to restart and within the short period of time, if

your application is not starting, it will again try to restart.

So, this will be a continuous loop.

That is why to be on a safer side, we need to provide more configurations here.

readinessProbe:

              httpGet:

                port: 8761

                path: /actuator/health/readiness

              initialDelaySeconds: 100

              periodSeconds: 30

            livenessProbe:

              httpGet:

                port: 8761

                path: /actuator/health/liveness

              initialDelaySeconds: 100

              periodSeconds: 30

Install spring cloud Kubernetes discovery server in K8s cluster

Inside this lecture, let me try to set up Discovery Server with the help of these Kubernetes manifest

file.

**So here you may have a question like why we are not using helm chat.**

There are two reasons on why I am not using helm chat here.

The very first one is Discovery Server is a one-time setup inside your Kubernetes cluster.

You do not have to do multiple changes every time.

With that reason, manually running these manifest files should be fine because this is only one time

activity.

And the second reason is as of now there is no helm chat developed by the helm chat community, including bitnami for this discovery server.

So, if we try to integrate this manifest file into our eazybank helm chat, then it is going to need a lot of efforts which I felt like it is not worth since this is only going to be one time activity to run this and fine running the manifest file manually.

I hope you are also fine with that.

So, to install the discovery server from the terminal, we need to run the command which is

\eazybank-microservice-application-k8s-helm\helm\environments> helm uninstall dev-env

release "dev-env" uninstalled

\eazybank-microservice-application-k8s-helm\kubernetes> kubectl apply -f kubernetes-discoveryserver.yml

I can go to the Kubernetes dashboard and make sure that Discovery server is started successfully.

**Now we are going to make some changes inside our helm charts and services also because we need to remove the details related to the Eureka Server inside our helm charts and services.**

Making Kubernetes Discovery Client changes in microservices – Part 1

**Now we are going to make some changes inside our helm charts and services also because we need to remove the details related to the Eureka Server inside our helm charts and services.**

Change the tag name to s17 in all microservices except eureka microservice

Update accounts microservice pom.xml

        <!-- <dependency>

            <groupId>org.springframework.cloud</groupId>

            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

        </dependency> -->

        <dependency>

            <groupId>org.springframework.cloud</groupId>

            <artifactId>spring-cloud-starter-kubernetes-discoveryclient</artifactId>

        </dependency>

D:\Experiments\Microservices\eazybank-microservice-application-k8s-helm\accounts\src\main\java\com\eazybytes\accounts\controller\CustomerController.java

// import org.apache.http.HttpStatus;

import org.apache.hc.core5.http.HttpStatus;

D:\Experiments\Microservices\eazybank-microservice-application-k8s-helm\accounts\src\main\java\com\eazybytes\accounts\AccountsApplication.java

@EnableDiscoveryClient

D:\Experiments\Microservices\eazybank-microservice-application-k8s-helm\accounts\src\main\resources\application.yml

Delete all eureka configuration

eureka:

  instance:

    preferIpAddress: true

  client:

    fetchRegistry: true

    registerWithEureka: true

    serviceUrl:

      defaultZone: http://localhost:8070/eureka/

D:\Experiments\Microservices\eazybank-microservice-application-k8s-helm\accounts\src\main\resources\application.yml

...

  config:

    import: "optional:configserver:http://localhost:8071/"

  cloud:

    kubernetes:

      discovery:

        all-namespaces: true

    openfeign:

      circuitbreaker:

...

D:\Experiments\Microservices\eazybank-microservice-application-k8s-helm\accounts\src\main\java\com\eazybytes\accounts\service\client\LoansFeignClient.java

@FeignClient(name="loans", url = "http://loans:8090", fallback = LoansFallback.class)

D:\Experiments\Microservices\eazybank-microservice-application-k8s-helm\accounts\src\main\java\com\eazybytes\accounts\service\client\CardsFeignClient.java

@FeignClient(name="cards", url = "http://cards:9000", fallback = CardsFallback.class)

D:\Experiments\Microservices\eazybank-microservice-application-k8s-helm\cards\pom.xml

and

D:\Experiments\Microservices\eazybank-microservice-application-k8s-helm\loans\pom.xml

        <!-- <dependency>

            <groupId>org.springframework.cloud</groupId>

            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

        </dependency> -->

        <dependency>

            <groupId>org.springframework.cloud</groupId>

            <artifactId>spring-cloud-starter-kubernetes-discoveryclient</artifactId>

        </dependency>

D:\Experiments\Microservices\eazybank-microservice-application-k8s-helm\cards\src\main\java\com\eazybytes\cards\CardsApplication.java

and  
D:\Experiments\Microservices\eazybank-microservice-application-k8s-helm\loans\src\main\java\com\eazybytes\loans\LoansApplication.java

@SpringBootApplication

@EnableDiscoveryClient

D:\Experiments\Microservices\eazybank-microservice-application-k8s-helm\cards\src\main\resources\application.yml

and

D:\Experiments\Microservices\eazybank-microservice-application-k8s-helm\loans\src\main\resources\application.yml

  config:

    import: "optional:configserver:http://localhost:8071/"

  cloud:

    kubernetes:

      discovery:

        all-namespaces: true

# eureka:

#  instance:

#    preferIpAddress: true

#  client:

#    fetchRegistry: true

#    registerWithEureka: true

#    serviceUrl:

#      defaultZone: http://localhost:8070/eureka/

Making Kubernetes Discovery Client changes in microservices – Part 2

Updating Helm charts for Kubernetes Discovery Server changes

Demo of Server-side service discovery and load balancing