Liveliness and readiness probe.

Spring boot actuator provides by default liveliness and readiness probes.

readinessProbe:

httpGet:

path: /actuator/health/readiness

port: 8080

initialDelaySeconds: 5

periodSeconds: 10

timeoutSeconds: 2

successThreshold: 1

failureThreshold: 3

livenessProbe:

httpGet:

path: /actuator/health/liveness

port: 8080

initialDelaySeconds: 15

periodSeconds: 20

timeoutSeconds: 2

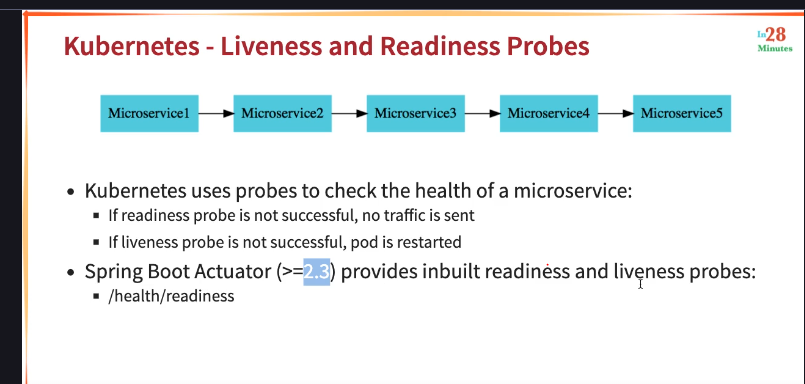
successThreshold: 1

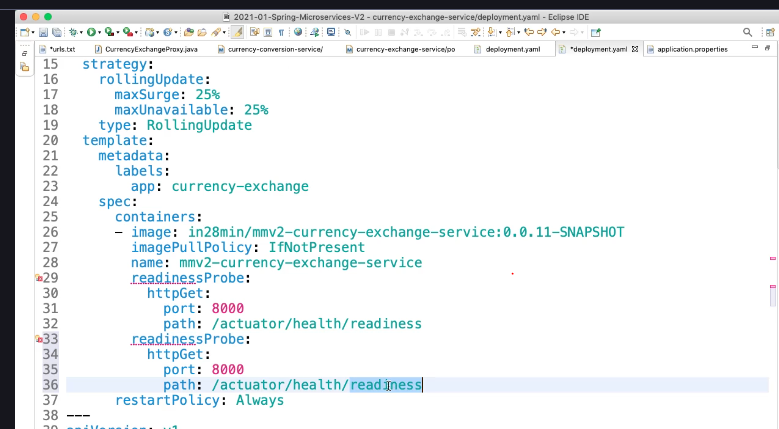
failureThreshold: 3

application.yml

management.endpoint.health.probes.enabled=true

management.endpoints.web.exposure.include=health





## 1. ****Liveness Probe**** – Is the app alive?

🔹 Purpose: Tells Kubernetes **whether the app is stuck or dead**.

📌 **If this fails**, Kubernetes will **restart the pod**.

### 🧪 Example:

Your app has a memory leak and gets stuck.  
It’s still running, but it’s non-responsive.  
The liveness probe fails, so Kubernetes **kills and restarts the container**.

yaml

CopyEdit

livenessProbe:

httpGet:

path: /healthz

port: 8080

initialDelaySeconds: 5

periodSeconds: 10

## ✅ 2. ****Readiness Probe**** – Is the app ready to serve traffic?

🔹 Purpose: Tells Kubernetes **whether the app is ready to receive requests**.

📌 **If this fails**, Kubernetes **removes the pod from the Service endpoint** — it won’t send traffic until it’s ready again.

### 🧪 Example:

Your app takes 30 seconds to load data from the DB.  
During this time:

* The readiness probe fails → no traffic is sent.
* Once the app is ready → probe passes → Kubernetes routes traffic to it.

yaml

CopyEdit

readinessProbe:

httpGet:

path: /ready

port: 8080

initialDelaySeconds: 5

periodSeconds: 10

## 🔁 Difference Between Liveness vs Readiness

| **Feature** | **Liveness Probe** | **Readiness Probe** |
| --- | --- | --- |
| Purpose | Check if app is alive | Check if app is ready for traffic |
| Failure action | Pod will be restarted | Pod is **removed from load balancer** |
| Used for | Crash detection, self-healing | Load balancing, startup control |
| Recovers automatically? | No (needs restart) | Yes (pod stays running) |

## 🧠 Real-Life Analogy

Imagine a web server:

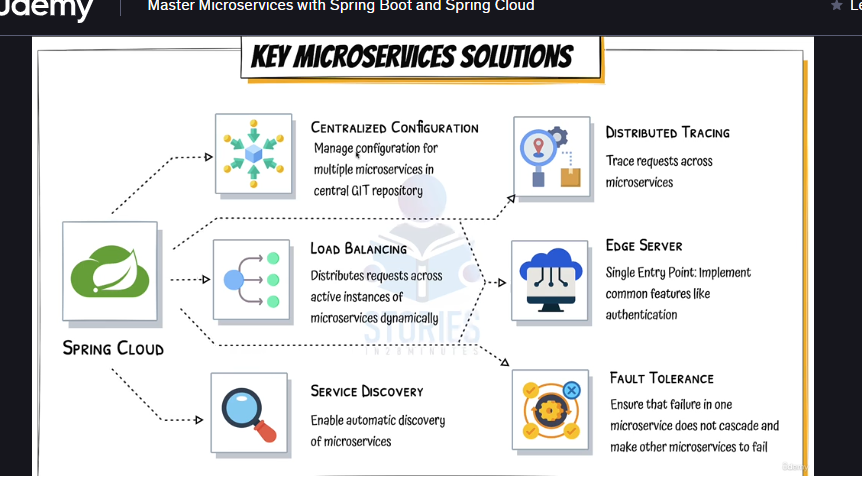
* **Liveness** is like checking if the server is plugged in and turned on.
* **Readiness** is like checking if the web page has loaded and is ready to accept users.

## 🛠️ Bonus: You Can Use Different Types of Probes

* httpGet: hits a URL (e.g. /health)
* tcpSocket: checks if port is open
* exec: runs a command (e.g., curl localhost:8080/ping)

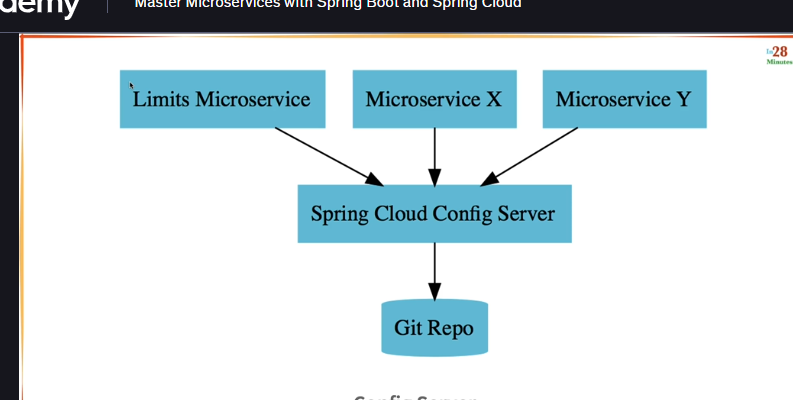
## ✅ Summary

| **Probe Type** | **When Used** | **What Happens on Failure** |
| --- | --- | --- |
| readinessProbe | During startup and after restarts | Pod temporarily removed from service |
| livenessProbe | Throughout lifecycle | Pod is killed and restarted |

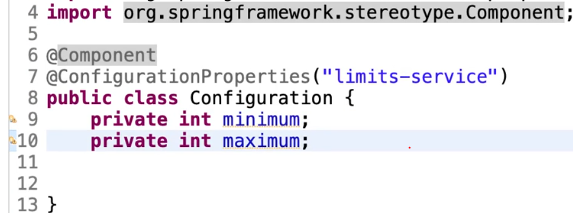


Centralized configurations:

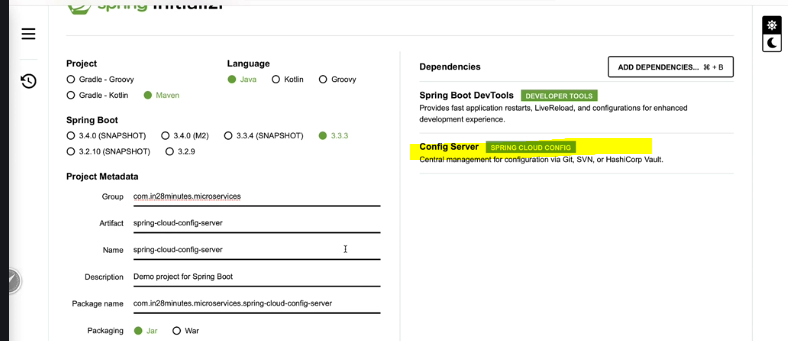
Spring config server.



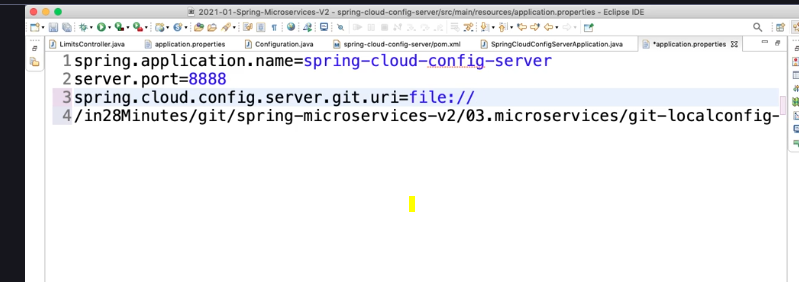
How to get values from properties file. By using @Value at property level and @ConfigurationProperties at the Class level.

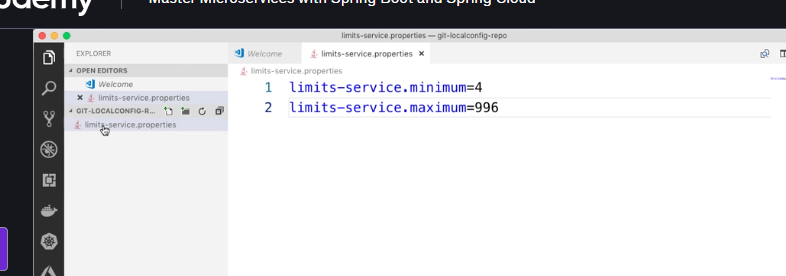


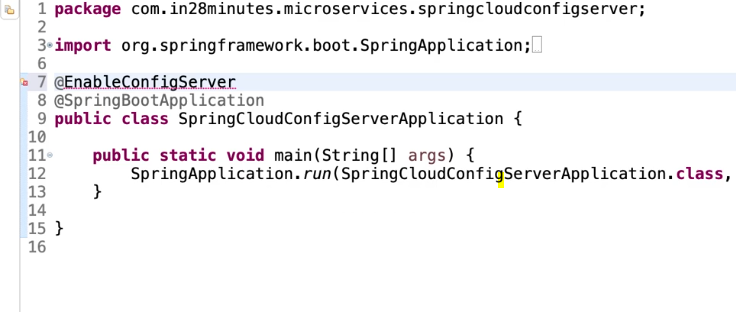
How to get the values from Config server?



In Config server application dependency is Config server while as in the Limit server one of the client is dependency is Config client.







@EnableConfigServer

@SpringBootApplication

public class SpringCloudConfigServerApplication {

public static void main(String[] args) {

SpringApplication.run(SpringCloudConfigServerApplication.class, args);

}

}

spring.application.name=spring-cloud-config-server

server.port=8888

spring.cloud.config.server.git.uri=file:///Ranga/git/01.udemy-course-repos/spring-micro-services/03.microservices/git-localconfig-repo

localhost:8888/limits-service/default

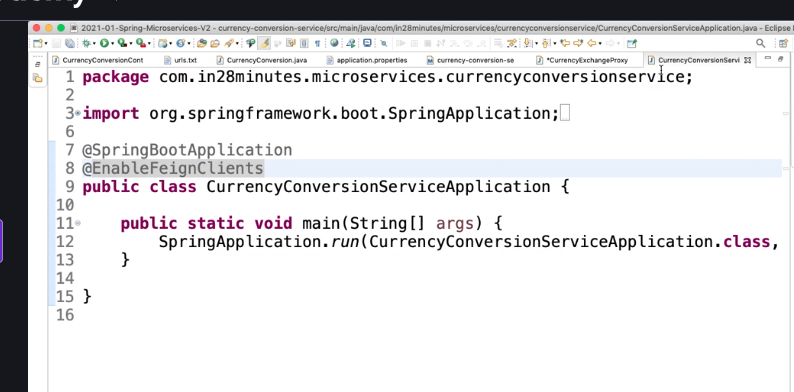
Profiles in Spring config server.

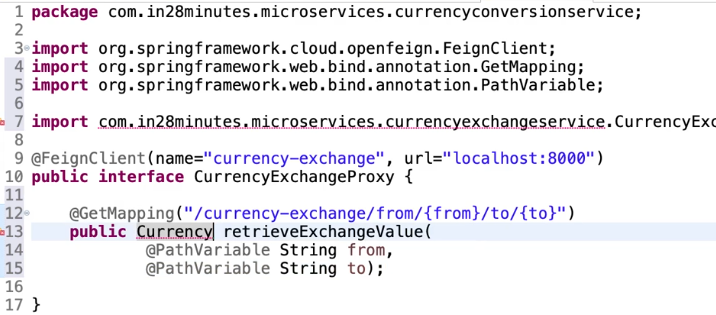
Limits-service-dev.properties

Limits-service.prod.properties

<http://localhost:8888/your-app-name/default>

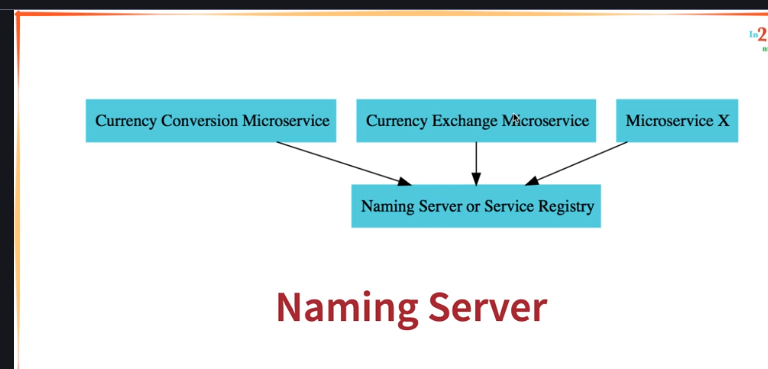
How to enable feign clients?

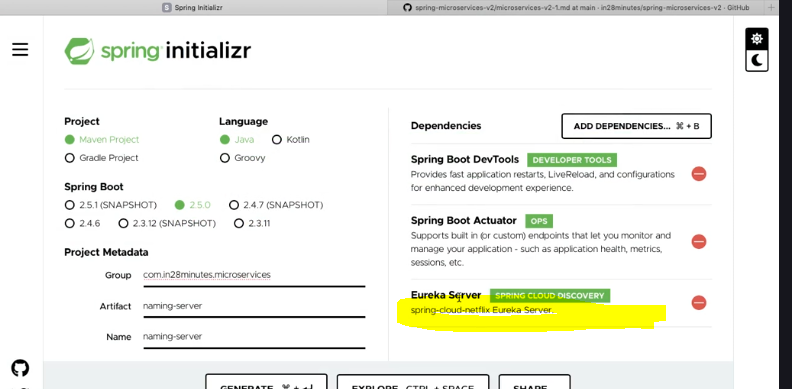


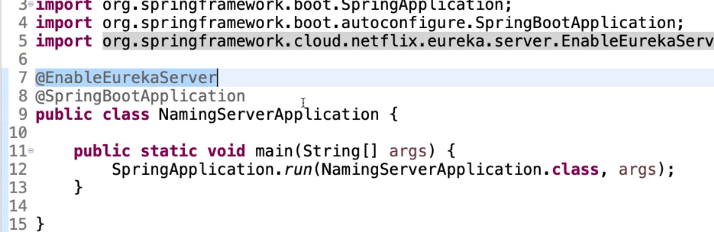


Why do we need Naming server or Service registry

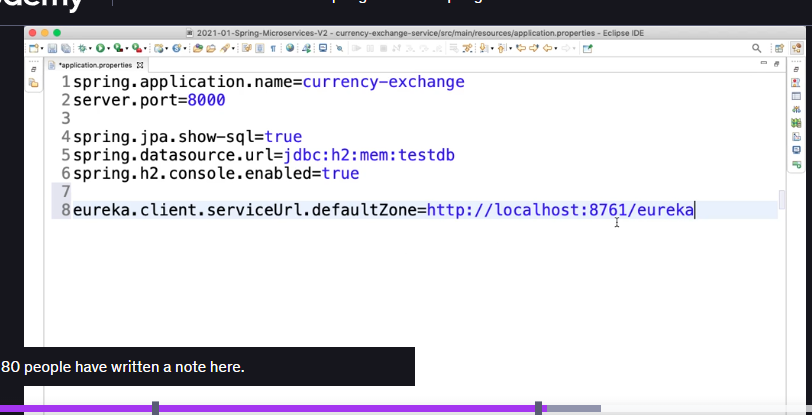
While using open feign in url we mention the host url, when the host url changes we cant keep on modifying the host url so for hat we use service registry.





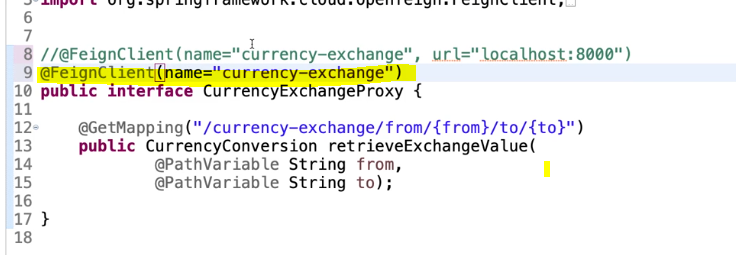


In the currency-exchange:

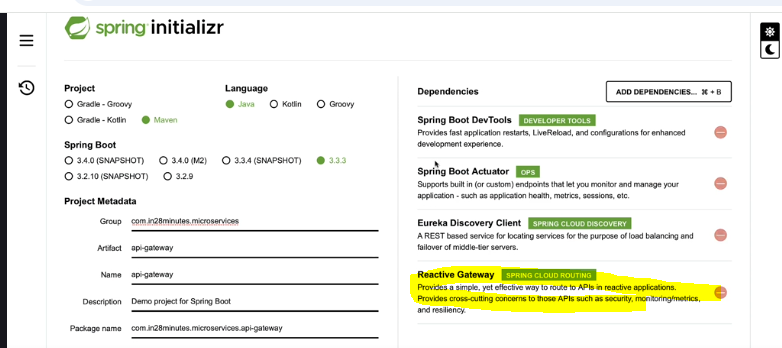


Load Balancing: By default it uses spring cloud load balancing.

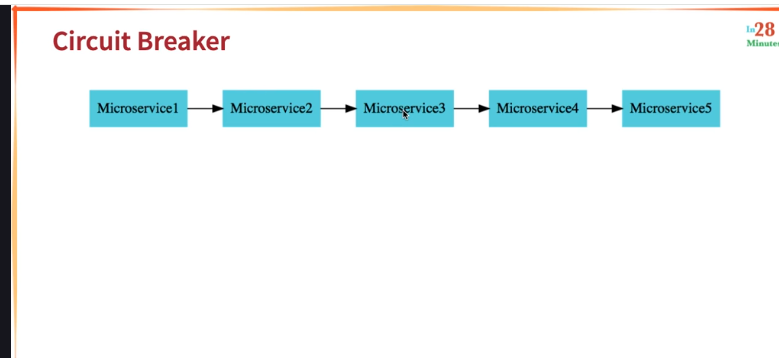
Just by adding name only feign talk to eureka and gets the currency-exchange service instance and sends the request.

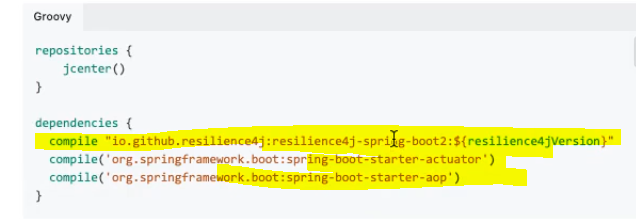


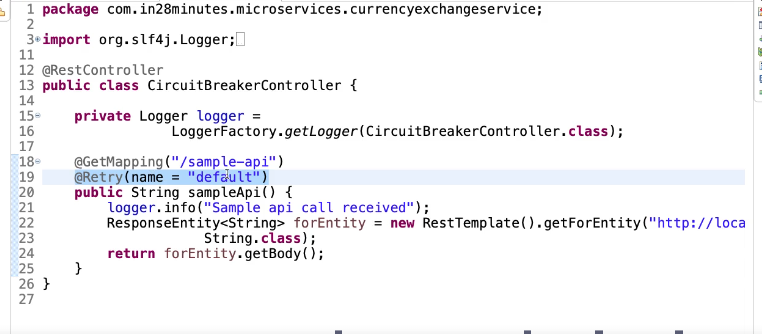
Spring API Gateway: Reactive gateway



Circuit breaker:







**import io.github.resilience4j.retry.annotation.Retry;**

**@RestController**

**public class ProductController {**

**@GetMapping("/products")**

**@Retry(name = "productService", fallbackMethod = "fallbackProducts")**

**public String getProducts() {**

**if (Math.random() < 0.7) {**

**throw new RuntimeException("Remote service failed");**

**}**

**return "Products fetched!";**

**}**

**public String fallbackProducts(Throwable t) {**

**return "Fallback: Cannot fetch products now.";**

**}**

**}**

**resilience4j.retry:**

**instances:**

**productService:**

**maxAttempts: 3**

**waitDuration: 1s**

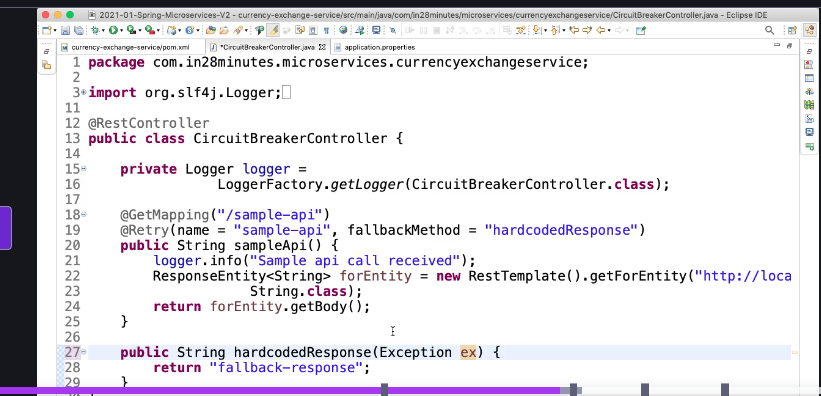
**retryExceptions:**

**- java.io.IOException**

**- java.util.concurrent.TimeoutException**

**ignoreExceptions:**

**- com.example.CustomBusinessException**



**import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;**

**@RestController**

**public class OrderController {**

**@GetMapping("/order")**

**@CircuitBreaker(name = "orderService", fallbackMethod = "fallbackOrder")**

**public String getOrder() {**

**// Simulate remote call**

**if (Math.random() < 0.5) {**

**throw new RuntimeException("Remote service failed");**

**}**

**return "Order placed";**

**}**

**public String fallbackOrder(Throwable t) {**

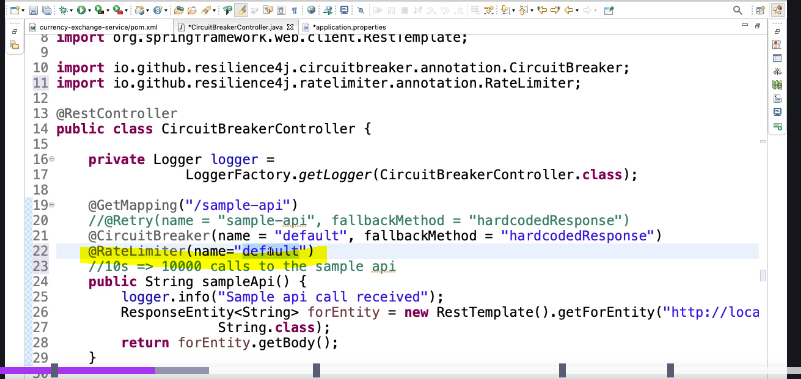
**return "Fallback: Unable to place order at the moment";**

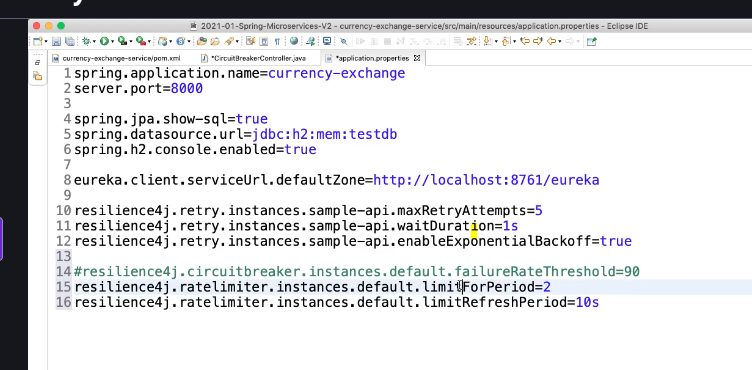
**}**

**}**

curl -X GET <http://localhost:8080/api/data>

How to configure RateLimit:





**What is bulk head in micro services?**

**How many concurrent calls we can allow at time for a specific end point.**

**How to implement the bulk head design pattern between Order and the inventory.**

**Client → Order Service → Inventory Bulkhead**

**| └─ max 5 concurrent calls**

**└─ If full → fallback method or error**

**@Bulkhead(name = "inventoryBulkhead", type = Bulkhead.Type.SEMAPHORE, fallbackMethod = "inventoryFallback")**

**@GetMapping("/check-stock/{itemId}")**

**public String checkStock(@PathVariable String itemId) {**

**return inventoryService.checkStock(itemId);**

**}**

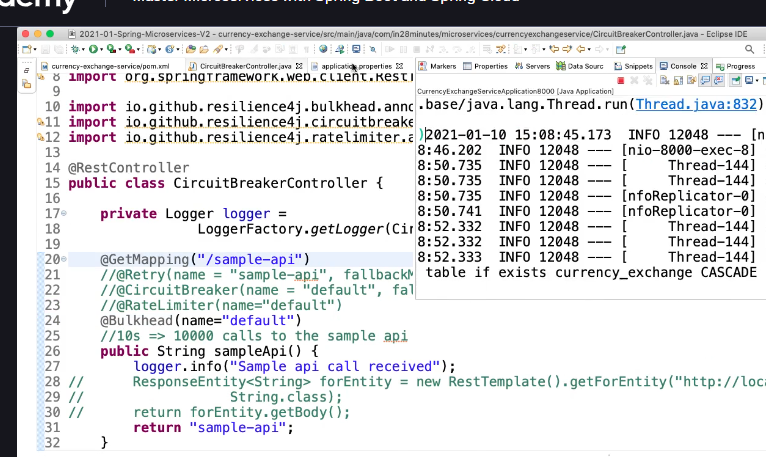
**resilience4j.bulkhead:**

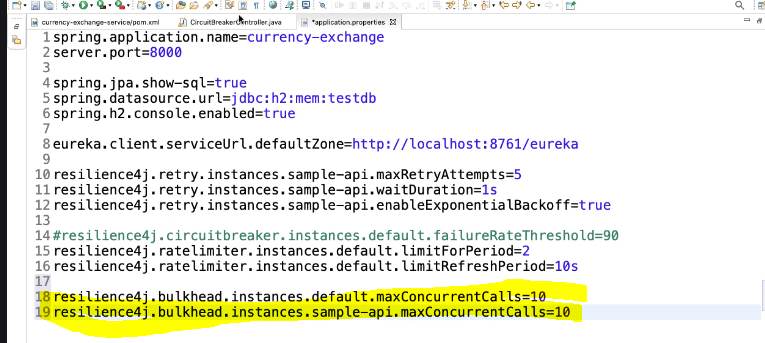
**instances:**

**inventoryBulkhead:**

**maxConcurrentCalls: 5**

**maxWaitDuration: 0**





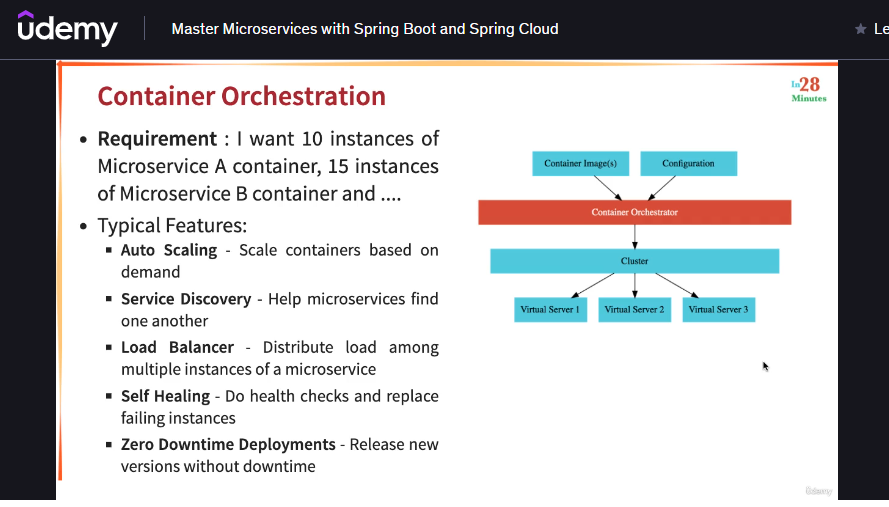
You're referencing Spring Cloud Sleuth — which is used for **distributed tracing** in microservices.

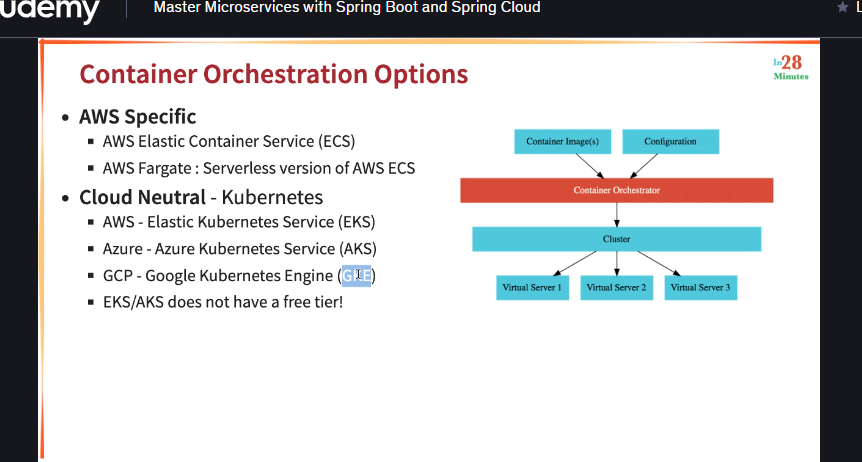
However, **Spring Cloud Sleuth is deprecated as of Spring Cloud 2022.x** and has been replaced by **Micrometer Tracing** + **Brave** or **OpenTelemetry**.

……………………………………………………………………………………..

Micro services with GCP

https://github.com/in28minutes/spring-microservices-v3/tree/main/05.kubernetes





Commands used in this project for AKS

1. gcloud config set project my-kubernetes-project-304910
2. gcloud container clusters get-credentials my-cluster --zone us-central1-c --project my-kubernetes-project-304910
4. kubectl create deployment hello-world-rest-api --image=in28min/hello-world-rest-api:0.0.1.RELEASE
5. kubectl get deployment
7. kubectl expose deployment hello-world-rest-api --type=LoadBalancer --port=8080
8. kubectl get services
9. kubectl get services --watch
10. press ctrl+c
11. curl 35.184.204.214:8080/hello-world
13. kubectl scale deployment hello-world-rest-api --replicas=3
15. gcloud container clusters resize my-cluster --node-pool default-pool --num-nodes=2 --zone=us-central1-c
17. kubectl autoscale deployment hello-world-rest-api --max=4 --cpu-percent=70
18. kubectl get hpa
20. kubectl create configmap hello-world-config --from-literal=RDS\_DB\_NAME=todos
21. kubectl get configmap
22. kubectl describe configmap hello-world-config
24. kubectl create secret generic hello-world-secrets-1 --from-literal=RDS\_PASSWORD=dummytodos
25. kubectl get secret
26. kubectl describe secret hello-world-secrets-1
28. kubectl apply -f deployment.yaml
30. gcloud container node-pools list --zone=us-central1-c --cluster=my-cluster
31. kubectl get pods -o wide
33. kubectl set image deployment hello-world-rest-api hello-world-rest-api=in28min/hello-world-rest-api:0.0.2.RELEASE
35. kubectl get services
36. kubectl get replicasets
37. kubectl get pods
39. kubectl delete pod hello-world-rest-api-58dc9d7fcc-8pv7r
41. kubectl scale deployment hello-world-rest-api --replicas=1
43. kubectl get replicasets
45. gcloud projects list
46. kubectl delete service hello-world-rest-api
47. kubectl delete deployment hello-world-rest-api
49. gcloud container clusters delete my-cluster --zone us-central1-c

What is deployment?

Deployemnt.yml will be having all the details like from where to get image, all the required details of image like credentials ..etc.

Finally all the required properties for service will be present in the deployment.yml

why service is required ?

By using service only we can expose our deployment to internet.

Different types of services for deployments?

different types of services type: ?

In Kubernetes, the spec.type field in a **Service** defines **how the service is exposed**.

Here are the **4 main types** of services:

### 🔹 1. ClusterIP (default)

* **Use case**: Internal communication between services within the cluster.
* **External access**: ❌ No
* **Example**:

yaml

CopyEdit

type: ClusterIP

#### 🔧 Example:

apiVersion: v1

kind: Service

metadata:

name: my-internal-service

spec:

type: ClusterIP

selector:

app: my-app

ports:

- port: 80

targetPort: 8080

### 🔹 2. NodePort

* **Use case**: Exposes the service on a static port on each node.
* **External access**: ✅ Yes (via <NodeIP>:<NodePort>)
* **Range**: Ports between 30000–32767
* **Example**:

type: NodePort

#### 🔧 Example:

apiVersion: v1

kind: Service

metadata:

name: my-nodeport-service

spec:

type: NodePort

selector:

app: my-app

ports:

- port: 80

targetPort: 8080

nodePort: 30036

### 🔹 3. LoadBalancer

* **Use case**: Exposes the service externally using a cloud provider's load balancer.
* **External access**: ✅ Yes (public IP)
* **Automatically creates**: A ClusterIP + cloud LB
* **Used in**: AKS, EKS, GKE, etc.
* **Example**:

yaml

CopyEdit

type: LoadBalancer

#### 🔧 Example:

apiVersion: v1

kind: Service

metadata:

name: my-loadbalancer-service

spec:

type: LoadBalancer

selector:

app: my-app

ports:

- port: 80

targetPort: 8080

### 🔹 4. ExternalName

* **Use case**: Maps a service to an external DNS name (like a legacy database or external API).
* **No selector**, no pods involved.
* **Creates a CNAME DNS record.**
* **Example**:

type: ExternalName

#### 🔧 Example:

apiVersion: v1

kind: Service

metadata:

name: external-db

spec:

type: ExternalName

externalName: mydb.example.com

### ✅ Summary Table

| **Service Type** | **Accessible From** | **Use Case** | **Creates Public IP?** |
| --- | --- | --- | --- |
| ClusterIP | Inside cluster | Default, for internal services | ❌ |
| NodePort | External via node IP & port | For simple external testing or use with Ingress | ❌ (access via NodeIP) |
| LoadBalancer | Internet | Public access to services in cloud | ✅ |
| ExternalName | Outside cluster | Redirect to external DNS names | ❌ |

………………………………………………………………………………………………………………………………………………………….

When to use service and when to use ingress?

My aks cluster has nodes 4 but i want to increase one of service in aks cluster to 100 how can i do that?

To increase **one of the services in your AKS cluster to run 100 replicas**, you simply need to scale the corresponding **Deployment**.

AKS (Azure Kubernetes Service) runs Kubernetes under the hood, so you scale using standard Kubernetes methods.

### ✅ Step-by-Step: Scale a Deployment to 100 Replicas

#### 🔧 Option 1: ****kubectl scale****

If you already have a deployment named, for example, my-app-deployment, run:

kubectl scale deployment my-app-deployment --replicas=100

This will instruct Kubernetes to maintain **100 replicas** of the pods.

#### 🔧 Option 2: ****Update the YAML****

Edit your deployment YAML (deployment.yaml) like this:

apiVersion: apps/v1

kind: Deployment

metadata:

name: my-app-deployment

spec:

replicas: 100

selector:

matchLabels:

app: my-app

template:

metadata:

labels:

app: my-app

spec:

containers:

- name: my-app-container

image: your-image

ports:

- containerPort: 8080

Then apply it:

bash

CopyEdit

kubectl apply -f deployment.yaml

### ⚠️ Important: What If You Have Only 4 Nodes?

* If 4 nodes **don’t have enough CPU/RAM**, some of the 100 pods **will stay in Pending state**.
* You have two options:

### 🚀 Option A: ****Enable Cluster Autoscaler****

AKS supports autoscaling your node pool automatically.

Enable it using the Azure CLI:

bash

CopyEdit

az aks nodepool update \

--resource-group myResourceGroup \

--cluster-name myAKSCluster \

--name nodepool1 \

--enable-cluster-autoscaler \

--min-count 4 \

--max-count 10

Now, when pods can’t be scheduled due to resource shortage, AKS will add more nodes automatically.

### 🚀 Option B: ****Manually Scale Node Pool****

You can also manually scale the node count:

bash

CopyEdit

az aks nodepool scale \

--resource-group myResourceGroup \

--cluster-name myAKSCluster \

--name nodepool1 \

--node-count 10

### ✅ After Scaling, Check Status

bash

CopyEdit

kubectl get pods -l app=my-app

kubectl get deployment my-app-deployment

kubectl get nodes

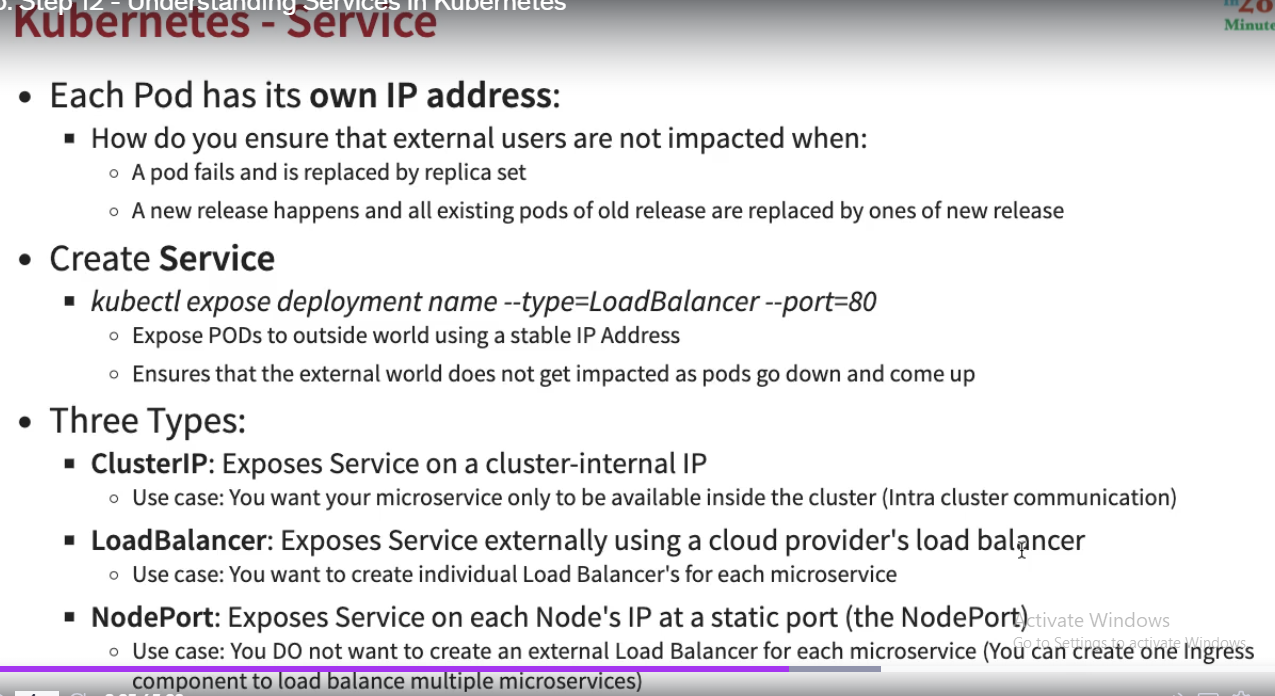
### 🧠 Summary

| **Action** | **Command / Tool** |
| --- | --- |
| Scale Deployment to 100 | kubectl scale deployment ... |
| Add nodes manually | az aks nodepool scale |
| Enable autoscaler | az aks nodepool update ... --enable-cluster-autoscaler |
| Check pod status | kubectl get pods |

……………………………………………………………………

Autoscaling, configmaps and Secrets.

1. kubectl autoscale deployment hello-world-rest-api --max=4 --cpu-percent=70
2. kubectl get hpa 🡪 **Horizontal Pod Autoscaler**.



Kubernetes components:

## CONTROL PLANE COMPONENTS

### 🔹 1. ****kube-apiserver****

📌 "Front door of the cluster"

#### ✅ Example Data:

* Receives this request:

bash

CopyEdit

kubectl create deployment spring-app --image=myacr.azurecr.io/spring:1.0

* Receives a patch update:

bash

CopyEdit

kubectl patch deployment spring-app -p '{"spec": {"replicas": 5}}'

#### 🧠 Role:

It talks to all other components, validates requests, and updates etcd with new cluster state.

### 🔹 2. ****etcd****

📌 "Cluster memory (key-value store)"

#### ✅ Example Data Stored in etcd:

* Deployment state:

json

CopyEdit

{"deployment": "spring-app", "replicas": 3, "image": "spring:1.0"}

* Node info:

json

CopyEdit

{"node1": {"cpu": "8", "memory": "16Gi", "status": "Ready"}}

#### 🧠 Role:

Holds the desired and actual state of everything in the cluster — services, pods, config maps, etc.

### 🔹 3. ****kube-scheduler****

📌 "The decision-maker"

#### ✅ Example Decisions:

* Schedules pod spring-app-12345 to node-2 based on:

json

CopyEdit

{"cpuAvailable": "2", "memoryAvailable": "3Gi"}

* Skips node-1 because:

json

CopyEdit

{"taints": ["NoSchedule"], "memoryAvailable": "100Mi"}

#### 🧠 Role:

Analyzes all nodes and picks the most suitable one for each new pod.

### 🔹 4. ****kube-controller-manager****

📌 "The enforcer"

#### ✅ Example Actions:

* Sees only 2 pods running instead of desired 3:

json

CopyEdit

{"desired": 3, "available": 2}

→ Creates a new pod.

* Detects a crashed pod and recreates:

json

CopyEdit

{"podName": "spring-app-xyz", "status": "CrashLoopBackOff"}

#### 🧠 Role:

Maintains desired state — scales replicas, recreates failed pods, updates endpoints.

## 🖥️ WORKER NODE COMPONENTS

### 🔹 5. ****kubelet****

📌 "The node manager"

#### ✅ Example Work:

* Starts this container:

json

CopyEdit

{"image": "myacr.azurecr.io/spring:1.0", "pod": "spring-app-12345"}

* Reports pod status to API server:

json

CopyEdit

{"pod": "spring-app-12345", "status": "Running"}

#### 🧠 Role:

Receives pod definitions from API server and ensures containers are running and healthy.

### 🔹 6. ****kube-proxy****

📌 "The traffic director"

#### ✅ Example Routing Rules:

* Forwards traffic to pod IPs:

json

CopyEdit

{"service": "spring-service", "targets": ["10.244.0.5", "10.244.0.6"]}

* Handles port mapping:

json

CopyEdit

{"servicePort": 80, "targetPort": 8080}

#### 🧠 Role:

Handles network rules so services and pods can talk to each other (load balances requests).

### 🔹 7. ****container runtime**** (e.g., containerd)

📌 "The engine that runs your containers"

#### ✅ Example Actions:

* Pulls image:

bash

CopyEdit

docker pull myacr.azurecr.io/spring:1.0

* Starts container:

json

CopyEdit

{"containerId": "abc123", "image": "spring:1.0", "state": "Running"}

#### 🧠 Role:

Runs your application containers as instructed by the kubelet.

## ✅ Summary Table with Real Data Examples

| **Component** | **Description** | **Real Data Examples** |
| --- | --- | --- |
| **kube-apiserver** | API for cluster | kubectl apply, patch request |
| **etcd** | Cluster state DB | replicas: 3, node status: Ready |
| **kube-scheduler** | Assigns pods to nodes | node: node-2, cpuAvailable: 2 |
| **kube-controller-manager** | Maintains desired state | available: 2 vs desired: 3, CrashLoopBackOff |
| **kubelet** | Runs pods on node | image: spring:1.0, status: Running |
| **kube-proxy** | Routes traffic | targets: [10.244.0.5], port: 8080 |
| **container runtime** | Pulls & runs containers | docker pull, container state: Running |

…………………………………………………………………………………….

ConfigMap: Centarlized configuration in kuberenetes.

### ****How do you secure microservices?****

**Answer:**

* Use **OAuth2**, **JWT** for authentication.
* Use **mTLS** for secure communication.
* Enforce **role-based access control (RBAC)**.

CAP Theorem:

Consistency

Availability

Partition tolerance

