Microservice  
with Spring Boot

short line

Your Name  
4th September, 20XX

# Microservice Characteristics

Microservices are a software architectural style that structures an application as a collection of small, loosely coupled services. Each service is focused on doing one thing well, runs in its own process, and communicates with other services through simple, well-defined interfaces (typically HTTP/REST APIs).Features for MS

* Loosely coupled
* Developed independently
* Tested independently
* Deployed independently
* Scaled independently
* Code and configuration maintainer separately
* Microservices must talk to each other
  + Synchronously- HTTP call
  + Asynchronous- Message broker Kafka/RabbitMQ

Key Component

* API gateway
* Service registry
* Config server
* Distributed tracing

## Spring boot

Spring Boot significantly helps in building microservices in several ways:

1. **Simplified Configuration**: Spring Boot's "convention over configuration" approach with auto-configuration reduces boilerplate code and setup time.
2. **Embedded Servers**: Built-in support for Tomcat, Jetty, or Undertow eliminates the need to deploy WAR files to external servers.
3. **Spring Cloud Integration**: Works seamlessly with Spring Cloud for distributed system patterns like service discovery, circuit breakers, and API gateway functionality.
4. **Actuators**: Provides Actuator endpoints exposing operational information about running applications (health, metrics, environment variables).
5. **Externalized Configuration**: Easy management of properties across environments through application.properties/yaml files.
6. **Easy Dependency Management**: Spring Boot starters simplify dependency management by bundling related dependencies.
7. **Rapid Development**: Spring Initializer and Spring Boot CLI accelerate project setup and development.
8. **Testing Support**: Comprehensive testing utilities that make unit and integration testing straightforward.
9. **RESTful APIs**: Simplifies creating RESTful services with Spring MVC and Spring Web Flux.

## Microservice Sizing

One of the most challenging aspects of building a successful microservices system is the identification of proper microservice boundaries and defining the size of each microservice. Below are the most common followed approaches in the industry

1. **Domain-Driven Sizing** Since many of our modifications or enhancements driven by the business needs, we can size/define boundaries of our microservices that are closely aligned with Domain-Driven design & Business capabilities. But this process takes lot of time and need good domain knowledge.
2. **Event Storming Sizing** Conducting an interactive fun session among various stake holder to identify the list of important events in the system like ‘Completed Payment’, ‘Search for a Product’ etc. Based on the events we can identify ‘Commands’, 'Reactions’ and can try to group them to a domain-driven services.

Other popular approaches include:

1. **Bounded Context Partitioning**: Based on DDD principles, this focuses on identifying contexts where specific models apply, with explicit boundaries between contexts becoming service boundaries.
2. **Data-Oriented Decomposition**: Services are defined around data entities they own, focusing on data cohesion and minimizing cross-service data dependencies.
3. **Transaction Boundaries**: Services are defined by transactional requirements, where operations that need ACID properties stay within a single service.
4. **Team Structure Alignment** (Conway's Law approach): Services are sized and organized to match team structures and capabilities.
5. **Strangler Pattern Implementation**: For legacy systems, gradually decomposing monoliths by identifying separable components based on feature sets or change frequency.
6. **Behavioural Analysis**: Examining user journeys and system behaviours to identify natural service boundaries.
7. **Seams Identification**: Finding natural "seams" in the application were changes rarely cross boundaries.
8. **Responsibility-Driven Design**: Focusing on single responsibility principle to define service boundaries.

## Cloud Native Applications

Cloud-native applications are software applications designed specifically to leverage cloud computing principles and take full advantage of cloud-native technologies and services. These applications are built and **optimized to run in cloud environments, utilizing the scalability, elasticity, and flexibility** offered by the cloud.

The engineering team at Heroku cloud platform introduced the 12-Factor methodology, a set of development principles aimed at guiding the design and construction of cloud-native applications. These principles are the result of their expertise and provide valuable insights into building web applications with specific characteristics:

1. **Cloud Platform Deployment:** Applications designed to be seamlessly deployed on various cloud platforms.
2. **Scalability as a Core**: Architectures that inherently support scalability.
3. **System Portability**: Applications that can run across different systems and environments.
4. **Enabling Continuous Deployment and Agility**: Facilitating rapid and agile development cycles.

These principles were developed to assist developers in building effective cloud-native applications, emphasizing the key factors that should be considered for optimal outcomes.

Subsequently, Kevin Hoffman expanded upon the original factors and introduced additional ones in his book, "Beyond the Twelve-Factor App" This revised approach, referred to as the 15-Factor methodology, refreshing the content of the original principles and incorporates three new factors.

1. **One codebase, one application** One-to-one correspondence between an application and its codebase, meaning each application has a dedicated codebase. Shared code is managed separately as a library, allowing it to be utilized as a dependency or as a standalone service, serving as a backing service for other applications. It is possible to **track each codebase in its own repository**, providing flexibility and organization.
2. **API first** This approach treats APIs as the foundation of the software, rather than building them later
3. **Dependency management** It is crucial to explicitly declare all dependencies of an application in a manifest and ensure that they are accessible to the dependency manager, which can download them from a central repository. In the case of Java applications, we are fortunate to have robust tools like Maven or Gradle that facilitate adherence to this principle.
4. **Design, build, release, run** Codebase progression from design to production deployment should follow below stages, Following the 15-Factor methodology, these stages must maintain strict separation, and runtime code modifications are prohibited to prevent mismatches with the build stage. Immutable build and release artifacts should bear unique identifiers, ensuring reproducibility.

**A diagram of a process

AI-generated content may be incorrect.**

1. **Configuration, credentials & code** to comply with this principle, configuration **should not be embedded within the code** or tracked in the same codebase, except for default configuration, which can be bundled with the application. Other configurations can still be managed using separate files, but they should be stored in a distinct repository.
2. **Logs** In a cloud-native application, **log routing and storage are not the application's concern**. Instead, applications should direct their logs to the standard output, treating them as sequentially ordered events based on time. The responsibility of log storage and rotation is now shifted to an external tool, known as a log aggregator. This tool retrieves, gathers, and provides access to the logs for inspection purposes.
3. **Disposability** Applications in the cloud are considered **ephemeral**, meaning that if a failure occurs and the application becomes unresponsive, it can be terminated and replaced with a new instance. Similarly, during high-load periods, additional instances of the application can be spun up to handle the increased workload. This concept is referred to as application disposability, where applications can be started or stopped as needed.
4. **Backing services** Backing services are external resources that an application depends on for functionality, including databases, message brokers, caching systems, SMTP servers, FTP servers, and RESTful web services. These services should be treated as attached resources that **can be modified or replaced** without changing the application code.

In the below example, we can see that a local DB can be swapped easily to a third-party DB like AWS DB without any code changes,

A diagram of a computer

AI-generated content may be incorrect.

1. **Environmental parity** aims to minimize differences between development and production environments while avoiding costly shortcuts. Containers are highlighted to promote consistent execution environments.

The factor addresses three key gaps:

* **Time gap**: Reducing deployment delays through automation and continuous deployment to shorten the time between development and production.
* **People gap**: Bridging the divide between developers who create applications and operators who handle production deployment by fostering DevOps culture and the "you build it, you run it" philosophy.
* **Tools gap**: Ensuring consistency in backing services across environments, recommending the use of identical database systems (like PostgreSQL) in both development and production instead of using different systems (like H2 locally and PostgreSQL in production).

1. **Administrative processes** Management tasks required to support applications, such as database migrations, batch jobs, or maintenance tasks, should be treated as isolated processes. Like application processes, the code for these administrative tasks should be version controlled, packaged alongside the application, and executed within the same environment. It is advisable to consider administrative tasks as independent microservices that are executed once and then discarded, or as functions configured within a stateless platform to respond to specific events. Alternatively, they can be integrated directly into the application, activated by calling a designated endpoint.
2. **Port binding** Cloud native applications, adhering to the 15-Factor methodology, should be self-contained and **expose their services through port binding**. In production environments, routing services may be employed to translate requests from public endpoints to the internally port-bound services.

An application is considered self-contained when it **doesn't rely on an external server** within the execution environment. For instance, a Java web application might typically run within a server container like Tomcat, Jetty, or Undertow. In contrast, a cloud native application does not depend on the presence of a Tomcat server in the environment; it manages the server as a dependency within itself. For example, **Spring Boot enables the usage of an embedded server**, where the application incorporates the server instead of relying on its availability in the execution environment. Consequently, each application is mapped to its own server, diverging from the traditional 6 approach of deploying multiple applications on a single server. of

The services offered by the application are then exposed through port binding. For instance, a web application binds its HTTP services to a specific port and can potentially serve as a backing service for another application. This is a common practice within cloud native systems.

1. **Stateless processes** Cloud native applications are often developed with high scalability in mind. One of the key principles to achieve scalability is designing applications as **stateless processes** and adopting share-nothing architecture. This means that no state should be shared among different instances of the application. It is important to ensure that **no data is lost if an instance of the application is destroyed and recreated**. If data loss occurs, then the application is not truly stateless.

However, it's important to note that some form of state management is necessary for applications to be functional. To address this, we design applications to be stateless and **delegate the handling and storage of state to specific stateful services**, such as data stores. In other words, a stateless application relies on a separate backing service to manage and store the required state, while the application itself remains stateless. This approach allows for better scalability and flexibility while ensuring that the necessary state is still maintained and accessible when needed.

1. **Concurrency** Scalability is not solely achieved by creating stateless applications. While statelessness is important, scalability also requires the ability to serve a larger number of users. This means that applications should support concurrent processing to handle multiple users simultaneously. According to the 15-Factor methodology, processes play a crucial role in application design. These **processes should be horizontally scalable**, distributing the workload across multiple processes on different machines. This concurrency is only feasible when applications are stateless. In Java Virtual Machine (JVM) applications, concurrency is typically managed using multiple threads, which are available from thread pools. Processes can be categorized based on their respective types. For instance, there are web processes responsible for handling HTTP requests, as well as worker processes that execute scheduled background jobs. By classifying processes and optimizing their concurrency, applications can effectively scale and handle increased workloads.
2. **Telemetry** Observability is a fundamental characteristic of cloud native applications. With the inherent complexity of managing a distributed system in the cloud, it becomes essential to have access to accurate and comprehensive data from each component of the system. This data enables remote monitoring of the system's behavior and facilitates effective management of its intricacies. Telemetry data, such as logs, metrics, traces, health status, and events, plays a vital role in providing this visibility. In Kevin Hoffman's analogy, he emphasizes the significance of telemetry by comparing applications to space probes. Just like telemetry is crucial for monitoring and controlling space probes remotely, the same concept applies to applications. To effectively monitor and control applications remotely, you need various types of telemetry data. Consider the kind of telemetry that would be necessary to ensure remote monitoring and control of your applications. This includes information such as detailed logs for troubleshooting, metrics to measure performance, traces to understand request flows, health status to assess system well-being, and events to capture significant occurrences. By gathering and utilizing these types of telemetry data, you can gain valuable insights into your applications and make informed decisions to manage them effectively from a remote location.
3. **Authentication & authorization** Security is a critical aspect of a software system, yet it often doesn't receive the necessary emphasis it deserves. To uphold a zero-trust approach, it is essential to ensure the security of every interaction within the system, encompassing architectural and infrastructural levels. While security involves more than just authentication and authorization, these aspects serve as a solid starting point. Authentication enables us to track and verify the identity of users by accessing the application. By authenticating users, we can then proceed to evaluate their permissions and determine if they have the necessary authorization to perform specific actions. Implementing identity and access management standards can greatly enhance security. Notable examples include OAuth 2.1 and OpenID Connect, which we will explore in this course.

# Configuration Management

## Config Server

With Config server we maintain configuration in external Service. We need to add config server dependency

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-config-server</artifactId>  
</dependency>

*@SpringBootApplication  
@EnableConfigServer  
public class* ConfigServerApplication {  
 *public static void* main(String[] args) {  
 SpringApplication.run(ConfigServerApplication.*class*);  
 }  
}

Config server supports multiple kinds of source for reading config

Reading from GIT

spring.profiles.active=git  
spring.cloud.config.server.git.uri=https://github.com/justamitsaha/configurations.git  
spring.cloud.config.server.git.default-label=main  
spring.cloud.config.server.git.timeout=5  
spring.cloud.config.server.git.clone-on-start=true  
spring.cloud.config.server.git.force-pull=true

Reading from classpath

spring.profiles.active=native  
spring.cloud.config.server.native.search-locations=classpath:/config

Reading from location

spring.cloud.config.server.native.search-locations="file:///Users//folder//Documents//config"

**Client-Side changes**

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-config</artifactId>  
</dependency>

If we don’t provide optional then in case the config server is not started the client service will fail and won’t start. In prod settings when client is dependent on congig server its better to not keep optional, so that developers know the issue during service startup

spring.config.import=optional:configserver:http://localhost:8071/  
spring.profiles.active=prod

If we want to provide default values of the variables we should provide them with the client application.properties, because otherwise the service will fail if it doesn’t get the values. Anyways the config server will overwrite the values from application.properties

*@Value*("${api.info}")  
*private* String buildVersion;

api.info=Product API local  
accounts.message=Welcome to Buying and Selling product Local config  
accounts.contactDetails.name=John Doe - Developer Local  
accounts.contactDetails.email=john@mailinator.com local  
accounts.onCallSupport=(555) 555-1234, (555) 523-1345

*@ConfigurationProperties*(prefix = "accounts")  
*public class* CompanyContactInfoDto {

Config server URL

Lets says we have below files

identity-service-prod.properties

identity-service.properties

<http://34.134.111.79/config/identity-service/default> will fetch default profile identity-service.properties

<http://34.134.111.79/config/identity-service/prod> will fetch prod profile + default

## Refresh information

Microservice will fetch the config information only during startup and will catch it. If we want to update it during runtime, we must use actuator

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>

management.endpoints.web.exposure.include=\*

But other microservices have already cached the information during start-up.

1. Change information in git repository.
2. Values will not update in the API <http://34.134.111.79/gateway/auth/configuration/contact-info>
3. Once we git the config server URL the values will change in config server <http://34.134.111.79/config/identity-service/prod>
4. But this will update the information in <http://34.134.111.79/gateway/auth/configuration/contact-info> which is cached
5. We can refresh the information using <http://34.134.111.79/gateway/auth/actuator/refresh> . This post endpoint will update the information and now the pod will have updated information

Config server also has a actuator endpoint. We can go to the actuator endpoint of each microservice <http://localhost:8071/actuator> . It will have many endpoints. One of them will be that of refresh <http://localhost:8071/actuator/refresh> .

Please note the auto refresh of config server doesn’t work if we use native profile or from class path, **it works only when we fetch information from GIT**

* Change the value in GIT
* Optional step, verify if the values has changed by checking in Config server endpoints like

<http://localhost:8071/identity-service/prod>

or

<http://localhost:8071/identity-service/default>

based on which profile we want to load

* Now update the config by hitting

POST <http://localhost:8080/auth/actuator/refresh>

Updating the refresh endpoint of Config server will not update it

POST <http://localhost:8071/actuator/refresh>

## Encryption Config server

Encrypt sensitive element in Github

* Set an encrypted key in config server properties

encrypt:

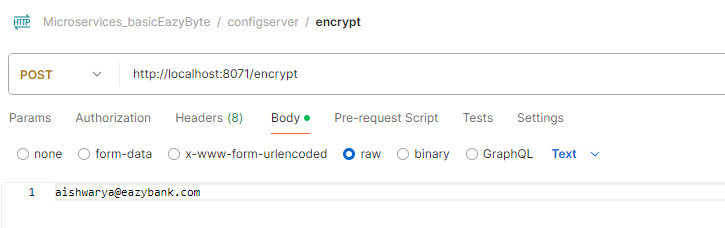
encrypt.key=45D81EC1EF61DF9AD8D3E5BB397F9

* After we set key config server will provide encrypt and decrypt URI

<http://localhost:8071/encrypt>

<http://localhost:8071/decrypt>

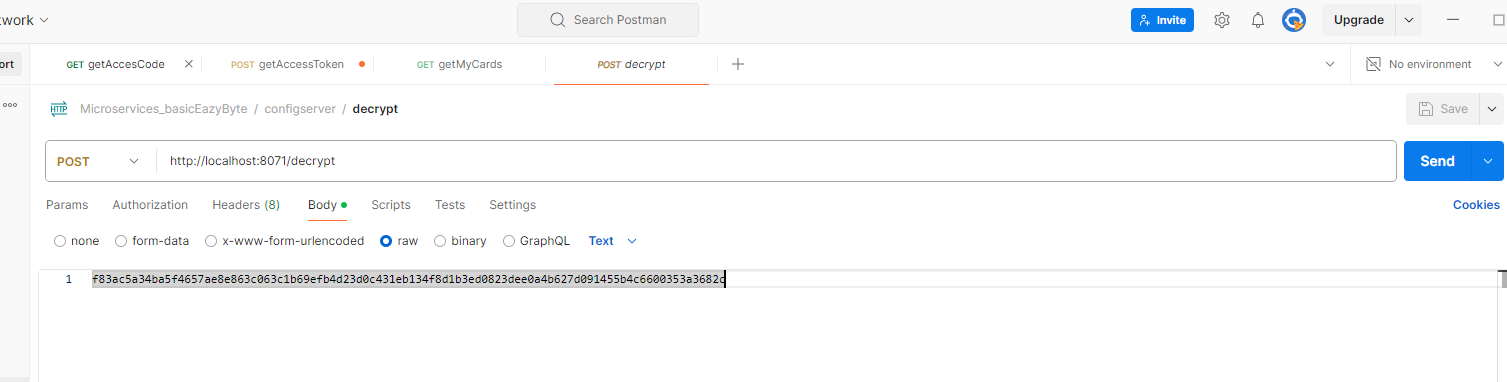
* With encrypt API we send the data we want to encrypt say email



* Once we get the encrypted value, we can set it in Github like below. To let config server know this is encrypted we pass keyword cipher

api.info=Product API Prod Config  
accounts.message=Welcome to Buying and Selling product local from config server prod  
accounts.contactDetails.name=John Doe - Developer  
accounts.contactDetails.email={cipher}f5a08fe2d89da3c88f7850a3483641bb69e7cad85890e9a051c3b07b1b11c708  
accounts.onCallSupport=(555) 555-1234, (555) 523-1345

* Decryption can be done like this



How the encryption process works.

1. Network team can prevent decrypt URL to be accessed by everyone and only microservice application can access it
2. Infra team will pass the encryption KEY as env parameter or CLI from jenkins.
3. Admin who has sensitive information encrypts using encrypted endpoints and sets the encrypted value in github.
4. The config server can decrypt the sensitive information using the secret key and pass it to microservices. This endpoint can be accessed only by microservice as the URL is restricted for other users

## Auto Refresh cache

Actuator is a good option to update configuration. But we may have hundreds of microservice, and each may have hundreds of instances. If we want to manually update the information, it may take a lot of manual work and we may miss a few pods leading to inconsistency. Also, with Kubernetes we will have pods and won’t have access to individual microservices

To overcome this challenge, we need to use a new project inside the spring cloud, which is the spring cloud bus. Spring cloud bus links all the nodes of a distributed system with a lightweight message broker. And this can be used to broadcast the state changes, for example, configuration changes or any other management instructions. So whenever you are using the spring cloud bus behind the scenes this spring cloud bus is going to interlink all your microservices instances with a lightweight message broker like Rabbitmq or Kafka. With this, the advantage is you need to invoke a bus refresh api path available against your actuator only one time for one of the instances.

If there are 500 instances running inside your production, you don't have to invoke the actuator refresh api for all your 500 instances. Instead, you can simply invoke the bus refresh API for any of the instances inside this total 500 instances. With that, the spring cloud bus will take care of communicating the changes happening on the spring cloud config server to all other nodes or the instances connected to the same message broker like Rabbitmq?

How to implement

* Install rabbit MQ using docker
* Add dependency related to Spring cloud bus and rabbit MQ in All microservice including config server, below will add both
* Then make sure actuator endpoint is enabled

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-bus-amqp</artifactId>  
</dependency>

management.endpoints.web.exposure.include=\*

* Then add the location of RabbitMQ

rabbitmq.host=localhost  
rabbitmq.port=5672  
rabbitmq.username=guest  
rabbitmq.password=guest

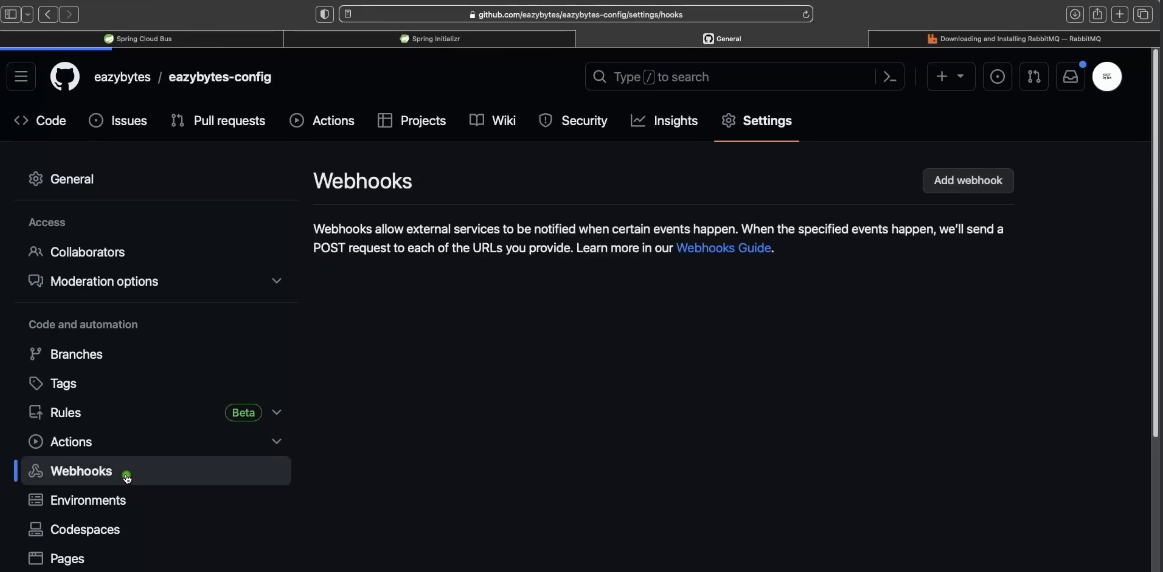
* Then we need to hit the endpoint <http://localhost:8080/actuator/busrefresh> with POST. If we hit it for one endpoint all instance of all API will get the change

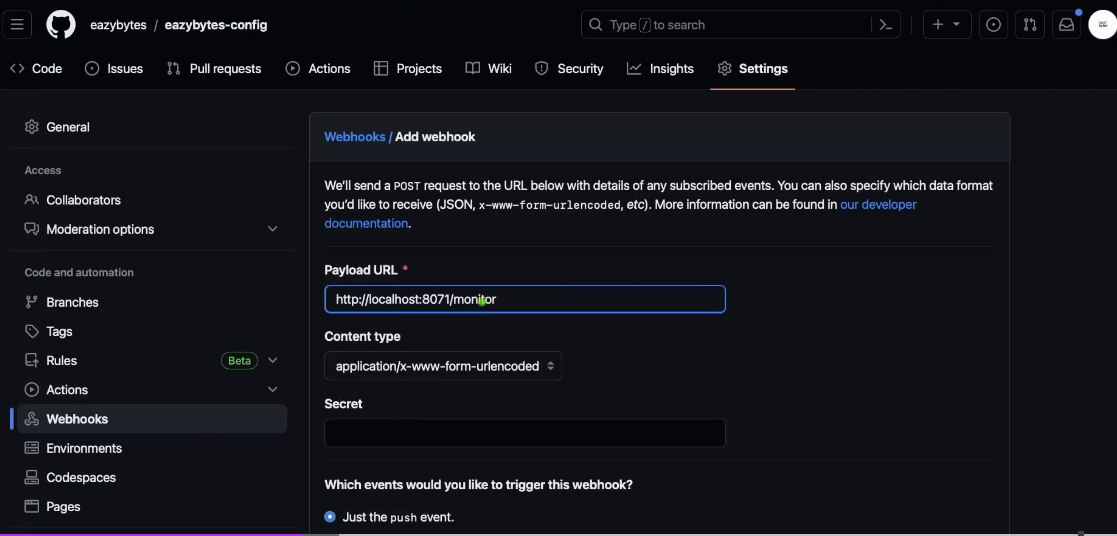
If we want no manual update and update of config on the push of code to github then we have to use a web hook. For this on top of previous change we need to add spring cloud config monitor in config server only.

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-config-monitor</artifactId>  
</dependency>

This will add a new API path /monitor. This is not part of the actuator. Using this monitor API path available inside the Spring Cloud Config server, we can create a webhook inside the GitHub repo saying that whenever a change happens inside GitHub repo, like a new property is added, please invoke the monitor API path. As soon as this monitor API path receives a webhook request from the GitHub repo, behind the scenes it is going to invoke the refresh event with the help of Spring Cloud Bus and RabbitMQ.

Changes in git hub





This will not work for local host so we can try hookdeck.com

## Liveness Readiness

When we are going to deploy our microservices, our applications as Docker containers and these containers, we don't have to manually monitor them and we don't have to manually handle the scaling or elasticity requirements. Instead, container orchestration products like Kubernetes, they are going to handle that. So to handle these containers effectively, platforms like Kubernetes or even Docker, they need to understand whether my running container is working without any issues and the health of my running container is fine. They will try to understand whether my container is running properly. If not, they will try to make some corrective steps like maybe they will try to restart the container even after restarting the container. If the health probes are not working properly, they will try to create a new container by deleting the existing one and the same.

**Liveness** Aliveness probe sends a signal that the container or application is either alive (passing) or dead (failing). If the container is alive, then no action is required because the current state is good. If the container is dead, then an attempt should be made to heal the application by restarting it. In simple words, liveness answers a true-or-false question: "Is this container alive?"

**Readiness** probe used to know whether the container or app being probed is ready to start receiving network traffic. If your container enters a state where it is still alive but cannot handle incoming network traffic (a common scenario during startup), you want the readiness probe to fail. So , traffic will not be sent to a container which isn't ready for it. If someone prematurely sends network traffic to the container, it could cause the load balancer (or router) to return a 502 error to the client and terminate the request. The client would get a "connection refused" error message. In simple words, readiness answers a true-or-false question: "Is this container ready to receive network traffic?"

Inside Spring Boot apps, the actuator gathers the "Liveness" and "Readiness" information from the Application availability interface and uses that information in dedicated health indicators: LivenessStateHealthIndicator and ReadinessState Health Indicator. These indicators are shown on the global health endpoint ("/actuator/health"). They are also exposed as separate HTTP Probes by using health groups: "/actuator/health/liveness" and "/actuator/health/readiness"

Cartoon a cartoon of a person in a boxing ring

AI-generated content may be incorrect. A cartoon of a boxer

AI-generated content may be incorrect.

We need to know if the config server is live and ready as other servers depend on it . This is done with the help of 2 actuator endpoints /actuator/health/liveness and /actuator/health/readiness. This needs to be enabled in config server properties like below

*#expose Actuator endpoints*management.endpoints.web.exposure.include=\*  
*#expose readiness*management.health.readiness-state.enabled=true  
*#expose liveness*management.health.liveness-state.enabled=true  
*#expose endpoint web URL for above*management.endpoint.health.probes.enabled=true

In order to expose the readiness and livelines we need to make below changes in the configserver section of Docker Compose. It hits the health readiness URL of the config server and if it return UP then success otherwise failure

config-server:  
 image: "justamitsaha/ms-config-server:1gateway"  
 container\_name: "config-ms"  
 ports:  
 - "8071:8071"  
*# depends\_on:  
# rabbit:  
# condition: service\_healthy* healthcheck:  
 test: "curl --fail --silent localhost:8071/actuator/health/readiness | grep UP || exit 1"  
 interval: 10s  
 timeout: 5s  
 retries: 10  
 start\_period: 10s  
 environment:  
 SPRING\_APPLICATION\_NAME: "config-server"  
 extends:  
 file: common-config.yml  
 service: microservice-base-config  
  
 discovery-service:  
 image: "justamitsaha/ms-discovery-service:1gateway"  
 container\_name: discovery-ms  
 ports:  
 - "8081:8081"  
 healthcheck:  
 test: "curl --fail --silent localhost:8081/discovery/actuator/health/readiness | grep UP || exit 1"  
 interval: 10s  
 timeout: 5s  
 retries: 10  
 start\_period: 10s  
 extends:  
 file: common-config.yml  
 service: microservice-base-config  
 environment:  
 SPRING\_APPLICATION\_NAME: "discovery-service"

## Common Config

There are a lot of common items in Docker compose which are repetitive. We can move them to a common config

services:  
 network-deploy-service:  
 networks:  
 - micro-service  
  
 microservice-base-config:  
 extends:  
 service: network-deploy-service  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
 *# environment:  
 # SPRING\_RABBITMQ\_HOST: "rabbit"* discovery-gateway-dependency:  
 extends:  
 service: microservice-base-config  
 depends\_on:  
 discovery-service:  
 condition: service\_healthy  
 environment:  
 EUREKA\_CLIENT\_SERVICEURL\_DEFAULTZONE: http://discovery-service:8081/discovery/eureka/  
  
  
 config-server-config-dependency:  
 extends:  
 service: discovery-gateway-dependency  
 depends\_on:  
 config-server:  
 condition: service\_healthy  
 environment:  
 SPRING\_PROFILES\_ACTIVE: default  
 SPRING\_CONFIG\_IMPORT: configserver:http://config-server:8071/

# Service Discovery

Discovery Server

<spring-cloud.version>2023.0.2</spring-cloud.version>

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>  
</dependency>

<dependencyManagement>  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-dependencies</artifactId>  
 <version>${spring-cloud.version}</version>  
 <type>pom</type>  
 <scope>import</scope>  
 </dependency>  
 </dependencies>  
</dependencyManagement>

Will also need actuator as it uses actuator endpoints to do health checks

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>

@SpringBootApplication  
@EnableEurekaServer  
public class DiscoveryServiceApplication {  
 public static void main(String[] args) {  
 SpringApplication.run(DiscoveryServiceApplication.class, args);  
 }  
}

spring.application.name=discovery-service  
server.port=8081  
server.servlet.context-path=/discovery  
  
eureka.instance.hostname=localhost  
eureka.client.fetchRegistry=false  
eureka.client.registerWithEureka=false  
eureka.client.serviceUrl.defaultZone=http://${eureka.instance.hostname}:${server.port}/discovery/eureka/  
  
#expose Actuator endpoints  
management.endpoints.web.exposure.include=\*  
#expose readiness  
management.health.readiness-state.enabled=true  
#expose liveness  
management.health.liveness-state.enabled=true  
#expose endpoint web URL for above  
management.endpoint.health.probes.enabled=true

On client side

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
</dependency>  
  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>

<dependencyManagement>  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-dependencies</artifactId>  
 <version>${spring-cloud.version}</version>  
 <type>pom</type>  
 <scope>import</scope>  
 </dependency>  
 </dependencies>  
</dependencyManagement>

<spring-cloud.version>2023.0.2</spring-cloud.version>

spring.application.name=discovery-service  
server.port=8081  
server.servlet.context-path=/discovery  
logging.pattern.console = ${LOGPATTERN\_CONSOLE:%green(%d{HH:mm:ss.SSS}) %blue(%-5level) %red([%thread]) %yellow(%logger{15}) - %msg%n}  
  
eureka.instance.hostname=localhost  
eureka.client.fetchRegistry=false  
eureka.client.registerWithEureka=false  
eureka.client.serviceUrl.defaultZone=http://${eureka.instance.hostname}:${server.port}/discovery/eureka/  
  
#expose Actuator endpoints  
management.endpoints.web.exposure.include=\*  
#expose readiness  
management.health.readiness-state.enabled=true  
#expose liveness  
management.health.liveness-state.enabled=true  
#expose endpoint web URL for above  
management.endpoint.health.probes.enabled=true

Important URL

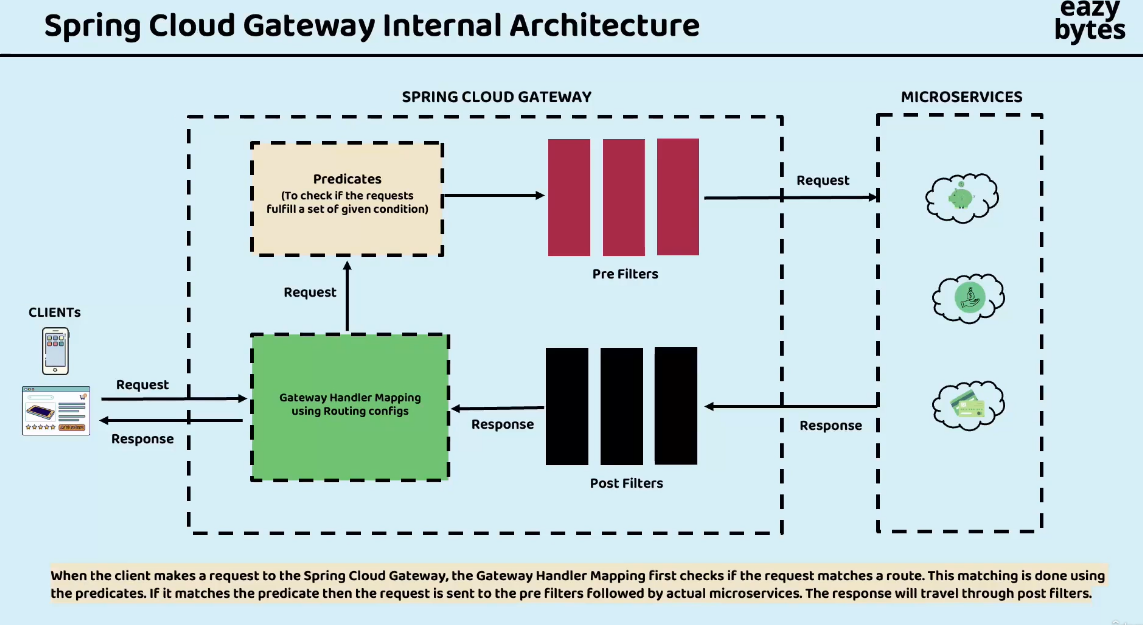
<http://localhost:8070/eureka/apps>

If we send header Accept application/json will get as JSON

As soon as my Eureka Server realizes that the majority of the microservices instances are not sending the heartbeat, then it is not going to react and remove all the service instance details from the registry. Instead, it is going to enter a self-preservation mode and once the Eureka Server enters this mode, even if it is not receiving the heartbeat signals from the service instances, it is not going to remove the service details from the service registry. This prevents the Eureka server from evicting all the instances due to some temporary network delays or temporary network glitches. Many times, the network issues can be on the internet provider, or it can be on the cloud provider side, or it can be within the microservice network regardless of where the issue is, after a few minutes or few seconds, the network related issue will automatically resolved. For these kinds of scenarios only, we have these Eureka self-preservation modes. Inside this self-preservation mode. The Eureka Server continues to serve the registered instances to client applications. It suspects that some instances are no longer available.

On a high level the summary is, Eureka Server is not going to panic whenever it is not receiving heartbeats from most of the instances. Instead, it will be calm and enter the self-preservation mode, and it will do the meditation. During the self-preservation mode or during this meditation process, it is not going to evict all the instances from the service registry. This feature is a savior where the network glitches are common and help us to handle the false positive alarms.

# Api Gateway



# Resiliency

Resilience4j is a lightweight fault tolerance library designed for functional programming. It offers the following patterns for increasing fault tolerance due to network problems or failure of any of the multiple services:

1. Circuit breaker - Used to stop making requests when a service invoked is failing
2. Fallback - Alternative paths to failing requests
3. Retry - Used to make retries when a service has temporarily failed
4. Rate limit - Limits the number of calls that a service receives in a time
5. Bulkhead - Limits the number of outgoing concurrent requests to a service to avoid overloading

## Circuit Breaker Gateway

When a microservice responds slowly or fails to function, it can lead to the depletion of resource threads on the Edge server and intermediate services. This, in turn, has a negative impact on the overall performance of the microservice network. To handle this kind of scenarios, we can use a Circuit Breaker pattern

3 states of circuit breaker

1. Closed → Will continue to send
2. Open – > Will not send request and fail
3. Half Open → In open state it will not stay forever, periodically it will send certain amount of traffic

A diagram of a circuit breaker

AI-generated content may be incorrect.

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-circuitbreaker-reactor-resilience4j</artifactId>  
</dependency>

Setting circuit breaker in Gateway. Comment out fallback URI and retry config related information. Also comment out timeout configurations in application.properties of gateway

.filters(gatewayFilterSpec -> gatewayFilterSpec  
 .rewritePath("/auth/(?<segment>.\*)", "/identity/${segment}")  
 .circuitBreaker(config -> config.setName("identity-circuit-breaker")))  
*// .circuitBreaker(config -> config.setName("identity-circuit-breaker")  
// .setFallbackUri("forward:/auth/configuration/contact-info"))  
// .retry(retryConfig -> retryConfig.setRetries(3)  
// .setMethods(HttpMethod.GET)  
// .setBackoff(Duration.ofMillis(100), Duration.ofMillis(1000), 2, true)))*

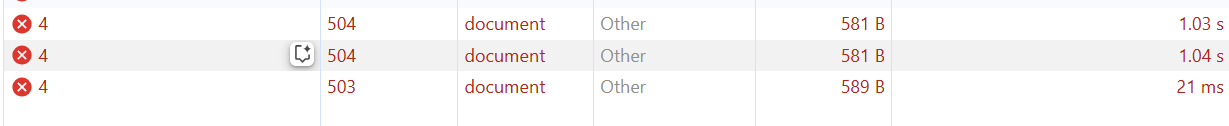
Instead of identity-circuit-breaker we can keep default and it will work for all circuit breaker in code

*#monitor 5 request before moving to close from open*resilience4j.circuitbreaker.configs.identity-circuit-breaker.slidingWindowSize=5  
*#send 2 request in half open and decide which state to go open or closed*resilience4j.circuitbreaker.configs.identity-circuit-breaker.permittedNumberOfCallsInHalfOpenState=2  
*#if 50% request are failing then go to open state*resilience4j.circuitbreaker.configs.identity-circuit-breaker.failureRateThreshold=50  
*#Will wait 10 seconds before going to half open*resilience4j.circuitbreaker.configs.identity-circuit-breaker.waitDurationInOpenState=5000

<http://localhost:8090/identity/actuator/circuitbreakerevents>

<http://localhost:8090/identity/actuator/circuitbreakers>

1. We can see the circuit breaker configuration from Actuator also <http://localhost:8080/actuator> . Where it will have many circuit-breaker endpoints <http://localhost:8080/actuator/circuitbreakers> and <http://localhost:8080/actuator/circuitbreakerevents?name=identity-circuit-breaker>
2. Initially it will have no value, it will start showing values when we hit any API for which any circuit breaker is configured. In our case something with auth. For e.g. <http://localhost:8080/auth/user/findById/4> . <http://localhost:8080/authentication/user/findById/4> is also a valid URL but it will not trigger circuit breaker since in gateway its configured with auth.
3. After we hit our endpoint and get success. We can se circuit breaker endpoint will have values, and it will be in closed status.
4. Put a debug point in <http://localhost:8080/auth/user/findById/4> and hit the API multiple times. Which will cause API to fail and trigger circuit breaker. Initially we will get 504 gateway timeouts, and it will take longer to execute.
5. After repeatedly hitting the endpoint, we will see the status changed to OPEN and response will be 503 service unavailable, and I will execute quickly.



1. In between it will allow few calls to verify if the state has changed or not and we will get 504
2. We configured a method which will help to simulate our circuit breaker

## Circuit breakers feign client

Previously we added reseliency in gateway sever. In the other api if we need to activate then we need to add below dependency as it doesn’t use spring reactor like gateway

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>  
</dependency>

spring.cloud.openfeign.circuitbreaker.enabled=true  
resilience4j.circuitbreaker.configs.default.slidingWindowSize=5  
resilience4j.circuitbreaker.configs.default.permittedNumberOfCallsInHalfOpenState=2  
resilience4j.circuitbreaker.configs.default.failureRateThreshold=50  
resilience4j.circuitbreaker.configs.default.waitDurationInOpenState=10000

We define fall back in the Feign client

*@Configuration  
@FeignClient*(name = "${product.service.feign.configuration.url}", fallback = ProductContactInfoFallback.*class*)  
*public interface* ProductFeignClient {  
  
 *@GetMapping*("/contact-info")  
 *public* ResponseEntity<CompanyContactInfoDto> getContactInfo();  
}

In case of error, it switches to fallback

*@Component  
public class* ProductContactInfoFallback *implements* ProductFeignClient {  
 *@Override  
 public* ResponseEntity<CompanyContactInfoDto> getContactInfo() {  
 *return null*;  
 }  
}

* To test circuit breaker will hit the below end point and keep product service stopped
* <http://localhost:8080/authentication/user/getUserContactInfo/2> .
* This service will fetch companyContactInfoDto from product service which is stopped. It can go to Feign client or web client, check from identity service logs
* When it goes to Feign client we can see "companyContactInfoDto": null is returned. When it goes to web client it will throw exception as the API will fail. Thus feign client helps us retrieve partial information when multiple sources are involved and there is a failure, providing graceful response to user

{

"userDto": {

"id": 2,

"name": "Shamit Saha",

"email": "shamit007@mailinator.com",

"phoneNumber": "9999999998",

"role": "SELLER"

},

"companyContactInfoDto": **null**

}

Like gateway we can see the circuit breaker for identity service also

<http://localhost:8080/auth/actuator>

<http://localhost:8080/auth/actuator/circuitbreakers>

<http://localhost:8080/auth/actuator/circuitbreakerevents?name=identity-circuit-breaker>

## Timeout:

Time out setup

//if not getting connected in 1 second we are not going to wait and we are going to kill the request.

spring.cloud.gateway.httpclient.connect-timeout=1000

//maximum time for which gateway server will wait to receive the response

spring.cloud.gateway.httpclient.response-timeout=2s

## Retry Pattern

The retry pattern will make configured multiple retry attempts when a service has temporarily failed. This pattern is very helpful in scenarios like network disruption where the client request may successful after a retry attempt. Here are some key components and considerations of implementing the Retry pattern in microservices:

* **Retry Logic**: Determine when and how many times to retry an operation. This can be based on factors such as error codes, exceptions, or response status.
* **Backoff Strategy**: Define a strategy for delaying retries to avoid overwhelming the system or exacerbating the underlying issue. This strategy can involve gradually increasing the delay between each retry, known as exponential backoff.
* **Circuit Breaker Integration**: Consider combining the Retry pattern with the Circuit Breaker pattern. If a certain number of retries fail consecutively, the circuit can be opened to prevent further attempts and preserve system resources.
* **Idempotent Operations**: Ensure that the retried operation is idempotent, meaning it produces the same result regardless of how many times it is invoked. This prevents unintended side effects or duplicate operations.

.route(p -> p  
 .path("/product/configuration/\*\*")  
 .filters(gatewayFilterSpec -> gatewayFilterSpec  
 .rewritePath("/product/configuration/(?<segment>.\*)", "/configuration/${segment}")  
 .retry(retryConfig -> retryConfig.setRetries(3)  
 .setMethods(HttpMethod.GET)  
 .setBackoff(Duration.ofMillis(100), Duration.ofMillis(1000), 2, *true*)))  
 .uri("lb://PRODUCT-SERVICE"))

We have defined a method which will return int 1/3 of time and give error 2/3 or time. When it fails it will retry, and we can see it in logs <http://localhost:8080/product/configuration/retry>

*@GetMapping*("/retry")  
*public int* getProbabilisticResponse() {  
 log.info("Inside retry pattern");  
 Random random = *new* Random();  
 *// Generate a random number between 0 and 2  
 int* randomValue = random.nextInt(3);  
  
 *// 1/3 chance of returning an integer (when randomValue is 0)  
 if* (randomValue == 0) {  
 *return* 42;  
 } *else* {  
 *// 2/3 chance of throwing an exception  
 throw new* RuntimeException("Random exception occurred");  
 }  
}

In above example we have configured retry pattern in gateway server we can configure in individual microservice also

public ResponseEntity<String> getBuildInfoFallback(Throwable throwable) {

***logger***.debug("getBuildInfoFallback() method Invoked");

return ResponseEntity

.*status*(HttpStatus.***OK***)

.body("0.9");

}

resilience4j.retry:

configs:

default:

maxRetryAttempts: 3

waitDuration: 500

enableExponentialBackoff: true

exponentialBackoffMultiplier: 2

ignoreExceptions:

- java.lang.NullPointerException

retryExceptions:

- java.util.concurrent.TimeoutException

## Rate Limiter

So based upon your requirements, you need to provide the details with the help of KeyResolver interface. There is also a default implementation of KeyResolver, which is PrincipalNameKeyResolver. If you are using spring security to secure your microservices, then with the help of these PrincipalNameKeyResolver, it is going to fetch the current logged in user name and accordingly it is going to enforce RateLimiter and by default you can see if the KeyResolver does not find a key, the requests are going to be denied if needed.

Using this Redis, we can implement these RateLimiter in gate way

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-redis-reactive</artifactId>

</dependency>

@Bean

public RedisRateLimiter redisRateLimiter() {

return new RedisRateLimiter(1, 1, 1);

}

@Bean

KeyResolver userKeyResolver() {

return exchange -> Mono.*justOrEmpty*(exchange.getRequest().getHeaders().getFirst("user"))

.defaultIfEmpty("anonymous");

}

Rate limiter in API

resilience4j.ratelimiter:

configs:

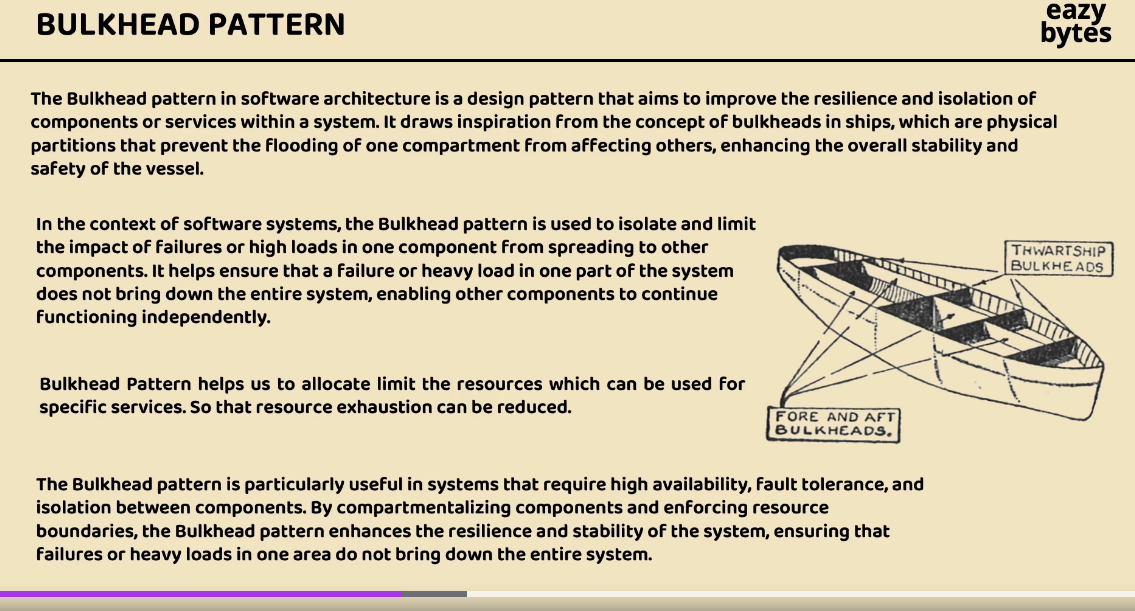
default:

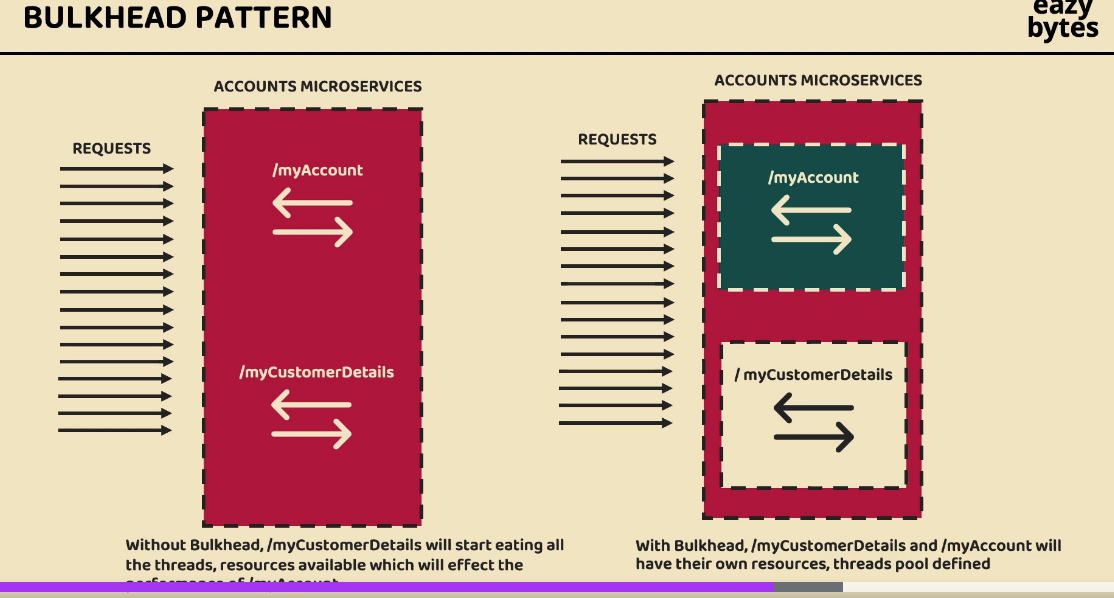
timeoutDuration: 1000

limitRefreshPeriod: 5000

limitForPeriod: 1

## BulkHead Pattern





# Helm

Helm is renowned as the "package manager of Kubernetes," aiming to enhance the management of Kubernetes projects by offering users a more efficient approach to handling the multitude of YAML files involved.

**Problem solved by Helm**

1. With out Helm we need to maintain all K8s manifest files for Deployment, Service, Config Map etc. for each microservice
2. With out Helm, the Dev-ops team members must manually apply or delete all the Kubernetes YAML manifest files using kubectl

**How it solves the problem**

The path Helm took to solve these issues is by using a packaging format called charts. A chart is a collection of files that describe a related set of Kubernetes resources. A single Helm chart might be used to deploy a simple app or something complex like a full web app stack with HTTP servers, databases, caches, and so on. A chart can have child charts and dependent charts as well. This means that Helm can install a whole dependency tree of a project with just a single command.

1. With Helm, we can a single template yaml file like shown below. Only the dynamic values will be injected during K8s services setup based on the values mentioned inside the values. yaml present inside each service/chart.
2. With the help of Helm, we can package all the YAML manifest files belongs into an application into a Chart. The same can be distributed into public or private repositories.
3. With the help of Helm, we can set up/upgrade/rollback/remove entire microservices applications into K8s cluster with just 1 command. No need to manually run kubectl apply command for each manifest file.
4. Helm automatically maintains the version history of the installed manifests. Due to that rollback of the entire K8s cluster to the previous working state is just a single command away.

**Helm structure**

* **Project name-** Folder under which helm files will get created
  + **Chart YAML-** Meta info about helm chart
  + **Values YAML-** Dynamic values for chart
  + **Charts folder-** Other chart on which current chart is dependent
  + **Templates –** Folder which contains manifests template YAML file configuration, service and deployment YAML files

Main take away from this is that if we are using helm chart, we can maintain all the dynamic values inside values.YAML file and rest will be taken care by helm.

# Monitoring and Observability

**Problem statement**

1. **Debugging :** How do we trace transactions across multiple services and containers and try to find where exactly the problem or bug is? How do we combine all the logs from multiple services into a central location where they can be indexed, searched, filtered, and grouped to find bugs that are contributing to a problem?
2. **Performance:** How can we track the path of a specific chain service call through our microservices network, and see how long it took to complete each microservice
3. **Health:** How can we easily and efficiently monitor the metrics like CPU usage, JVM metrics, etc. for all the microservices applications in our network? How can we monitor the status and health of all our microservices applications in a single place, and create alerts and notifications for any abnormal behavior of the services?

Observability and monitoring solve the challenge of identifying and resolving the above problems in microservices architectures before they cause outages.

**Observability** is the ability to understand the internal state of a system by observing its outputs. In the context of microservices, observability is achieved by collecting and analyzing data from a variety of sources, such as metrics, logs, and traces. The three pillars of observability are

1. Metrics : CPU usage, memory usage, response times
2. Logs
3. Traces: record of path taken by a request in a system

**Monitoring** microservices involves checking the telemetry data available for the application and defining alerts for known failure states. This process collects and analyzes data from a system to identify and troubleshoot problems, as well as track the health of individual microservices and the overall health of the microservices network.

Monitoring and observability can be considered as two sides of the same coin. Both rely on the same types of telemetry data to enable insight into software distributed systems. Those data types — metrics, traces, and logs — are often referred to as the three pillars of observability.

**Distributed tracing** is a technique used in microservices or cloud-native applications to understand and analyze the flow of requests as they propagate across multiple services and components. It helps in gaining insights into how requests are processed, identifying performance bottlenecks, and diagnosing issues in complex, distributed systems.

One possible solution to address this issue is to implement a straightforward approach where a unique identifier, known as a correlation ID, is generated for each request at the entry point of the system. This correlation ID can then be utilized in event logs and passed along to other relevant services involved in processing the request. By leveraging this correlation ID, we can retrieve all log messages associated with a specific transaction from multiple applications.

Distributed tracing encompasses three primary concepts:

* Tags serve as metadata that offer supplementary details about the span context, including the request URI, the username of the authenticated user, or the identifier for a specific tenant.
* A trace denotes the collection of actions tied to a request or transaction, distinguished by a trace ID. It consists of multiple spans that span across various services.
* A span represents each individual stage of request processing, encompassing start and end timestamps, and is uniquely identified by the combination of trace ID and span ID.

**Logging in Microservices**

Logging in microservices is complex. This is because each service has its own logs. This means that you need to look in multiple places to find all the logs for a particular request. To address this challenge, microservices architectures often use centralized logging. Centralized logging collects logs from all the services in the architecture and stores them in a single location. This makes it easier to find and troubleshoot problems, as you only need to look in one place

1. **Grafana** is an open-source analytics and interactive visualization web application. It provides charts, graphs, and alerts for the web when connected to supported data sources. It can be easily installed using Docker or Docker Compose. Grafana is a popular tool for visualizing metrics, logs, and traces from a variety of sources. It is used by organizations of all sizes to monitor their applications and infrastructure.
2. **Grafana Loki** is a horizontally scalable, highly available, and cost-effective log aggregation system. It is designed to be easy to use and to scale to meet the needs of even the most demanding applications.
3. **Promtail** is a lightweight log agent that ships logs from your containers to Loki. It is easy to configure and can be used to collect logs from a wide variety of sources.
4. **Grafana Tempo**  for tracing
5. **Prometheus** is an open-source systems monitoring and alerting toolkit. Just as Loki aggregates and stores event logs, Prometheus does the same with metrics. It is a component which is responsible for extracting and aggregating all the metrics from the microservices instances running inside my Microservice network. Prometheus cannot understand the metrics provided by the actuator because actuator exposes the metrics information in a Json format which we can easily understand as humans, since the same format cannot be understood by the Prometheus, we need to use micrometer.
6. **Micrometer** automatically exposes /actuator/metrics data into something your monitoring system can understand. All you need to do is include that vendor-specific micrometer dependency in your application. Micrometer automatically exposes the actuator and metrics data into a format that a particular monitoring system can understand. We, as a developer, need to simply add the vendor specific micrometer dependency inside the application and post that the micrometer is going to take care of exposing the metrics information in a format that you are monitoring system can understand.

<dependency>  
 <groupId>io.micrometer</groupId>  
 <artifactId>micrometer-registry-prometheus</artifactId>  
</dependency>

*# To group the properties exposed for this service un the application name*metrics.tags.application=${spring.application.name}

We're trying to assign a value from the property which is spring.application.name which is present in the properties file at the top So at the runtime the value accounts will be assigned to this property. With the help of this property, we are telling the micrometer and Prometheus, please group all my metrics related to the accounts microservice under the application name, which is accounts. This is going to help you to identify the metrics of each of the microservice. Otherwise, it is difficult to determine which metrics correspond to which microservice.

This will expose the list of metric at the endpoints below. Note we can’t access other microservice actuator via gateway URL as we haven’t configured routing for that in gateway routing

Gateway <http://localhost:8080/actuator/metrics>

Discovery <http://localhost:8081/discovery/actuator/metrics>

Identity <http://localhost:8090/identity/actuator/metrics>

Product <http://localhost:8091/product/actuator/metrics>

Identity <http://localhost:8071/actuator/metrics>

We get the detailed information of the metrics by putting the metric name at end. For e.g. if we want to get jvm.memory.used then we can use <http://localhost:8080/actuator/metrics/jvm.memory.used>, for eureka <http://localhost:8081/discovery/actuator/metrics/jvm.memory.used> . But we can’t go to individual metric to get all the information, as there are so many metrics so we can use <http://localhost:8080/actuator/prometheus> for eureka <http://localhost:8081/discovery/actuator/prometheus>

1. **Open Telemetry** Open Telemetry supports distributed tracing. There are other tools also like
   1. **Spring cloud sleuth** Discontinued and migrated to Micrometer tracing
   2. **Micrometer tracing**  This also supports distributed tracing. It supports integration with open zipkin and open telemetry. But it’s complicated to set up.

So, we will stick to Open telemetry.

When we work with microservices, requests often pass through multiple services, making it difficult to trace where things go wrong or where performance bottlenecks occur.

* **Open Telemetry (OTEL)** helps us solve this by automatically capturing trace and span data from our applications.
* It works by injecting **bytecode** at runtime into the application, which tracks each request and records:
  + **Traces:** End-to-end journey of a request across services.
  + **Spans:** Each individual step within a trace.
  + **Metadata:** Additional information to give context to the trace.

Think of it like adding breadcrumbs to follow a request’s journey through our services.

**How Does OpenTelemetry Work?**

* As requests hit our microservices, OpenTelemetry instruments the application to generate trace and span information.
* This data is then sent to a backend system for storage and further analysis.

**Where Does Tempo Fit In?**

Once OpenTelemetry collects the trace data, we need to **store and analyze** it. That’s where **Tempo** comes in.

* **Tempo** is part of the Grafana ecosystem and specializes in storing and indexing trace data.
* It’s **open-source, highly scalable, and cost-effective**, making it a great choice for large trace volumes.
* Just like:
  + **Loki** stores logs, and
  + **Prometheus** stores metrics,
  + **Tempo** stores and indexes trace data.

**Visualizing Trace Data with Grafana**

Now, collecting and storing trace data is useful, but we need a way to **visualize it** for easy analysis.

* **Grafana** acts as the **UI layer** that connects to different data sources (Loki, Prometheus, Tempo) and presents the data visually.
* For trace data, Grafana connects with Tempo and provides a **distributed tracing dashboard** where we can explore and analyze trace information.

# Security

**Introduction to OAuth2:**

* OAuth stands for **Open Authorization**, built on IETF standards and licensed by the **Open Web Foundation**.
* OAuth2 is the **second version** of the OAuth framework, with an upcoming version named **OAuth 2.1**.
* It enables an application to grant a third-party application access to your data **without sharing passwords**.
* This process is called **delegated authorization**.

**Key Advantages of OAuth2:**

1. **Versatile Application Support:**

* OAuth2 supports various applications such as:
  1. Backend-to-backend communication
  2. UI/mobile apps communicating with backend servers
  3. IoT devices, consoles, smart TVs, etc.
* Different **grant flows** are available for each use case.

1. **Separation of Authentication Logic:**

* OAuth2 recommends a **separate authorization server**.
* The authorization server handles:
  1. Receiving requests from clients.
  2. Issuing access tokens after successful authentication.
* Multiple applications (internal and third-party) can connect to the authorization server.
* Example: Stack Overflow uses GitHub’s auth server to authenticate users.

1. **Enhanced Security Through Access Tokens:**

* End-users do **not share credentials** with third-party applications.
* An **access token** is issued, allowing limited access to data.
* Analogy: Similar to a **hotel access card** that only opens assigned rooms.
* If compromised, access tokens can be revoked easily, maintaining security.

1. **Role-Based Privileges:**

* Tokens are issued with different privileges based on the business requirements.
* Example:
  1. A guest’s access card allows access only to their room.
  2. Housekeeping staff can access multiple rooms with broader privileges.

Oauth2 Teminology

Lets say a user wants to register for Stack overflow and he uses the option of register with git hub. Lets see what are thevarious terminologies

* **Resource Owner** - It is you the end user. In the scenario of Stack overflow, the end user who wants to use the GitHub services to get his details. In other words, the end user owns the resources (email, profile), that’s why we call him as Resource owner
* **Client**  The website, mobile app or API will be the client as it is the one which interacts with GitHub services on behalf of the resource owner/end user. In the scenario of Stack overflow, the Stack overflow website is Client
* **Authorization Server**  This is a server which knows about resource owner. In other words, resource owner should have an account on this server. In the scenario of Stack overflow, the GitHub server which has authorization logic acts as Authorization server.
* **Resource Server** This is the server where the resources that client want to consume are hosted. In the scenario of Stack Overflow, the resources like User Email and Profile details are hosted inside GitHub server. So it will act as a resource server.
* **Scopes** These are the granular permissions the Client wants, such as access to data or to perform certain actions. The Auth server can issue an access token to client with the scope of Email, READ etc.

**What is OpenID-Connect?**

* **OpenID Connect (OIDC)** is a protocol built on top of the **OAuth2 framework**.
* Many developers mistakenly believe that OpenID Connect is a **replacement** for OAuth2, but that is incorrect.
* OAuth2 primarily supports **authorization**, whereas OpenID Connect adds **authentication**.

**OAuth2 Limitations:**

* OAuth2 was originally designed only for **authorization**, not authentication.
* It allows applications to grant limited access to third-party applications using an **access token**.
* However, many organizations used OAuth2 for **authentication** by passing user details (like email) in the access token, leading to **non-standardized implementations**.

The OpenID Connect flow looks the same as OAuth. The only differences are, in the initial request, a specific scope of openid is used, and in the final exchange the client receives both an Access Token and an ID Token.

**Need for OpenID Connect:**

* Organizations needed a **standardized approach** to retrieve user identity details.
* **OIDC** was introduced as a wrapper on top of OAuth2 to address this gap.
* OIDC introduces an **ID token** that contains authenticated user information.
* This ensures a consistent and secure way to **authenticate users** and retrieve their identity details.

**How OpenID Connect Works:**

1. **Base Protocol:** OAuth2 is built on **HTTP protocol** for communication.
2. **OIDC Layer:** OIDC adds an extra layer that supports **authentication**.
3. **Scope Definition:** When using OpenID Connect, a **scope** with openid is sent.
4. **Token Issuance:**

* Access Token: For accessing protected resources.
* ID Token: For retrieving user information (identity details).

1. **Without openid Scope:** Only an **access token** is returned.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | |  | | --- | | OpenID Connect | | Oauth2.0 | | HTTP | |

**Key Advantages of OpenID Connect:**

1. **Standardized Identity Sharing:**

* OIDC provides a consistent way to share user details using scopes like:
  1. openid – Basic information
  2. profile – Profile details
  3. email – Email address
  4. address – Address details

1. **Use of JWT Standard:**

* Both **access tokens** and **ID tokens** follow the **JWT (JSON Web Token)** standard.
* Ensures secure and standardized token formats.

1. **User Info Endpoint:**

* OIDC exposes a standardized endpoint /userinfo.
* Client applications can query this endpoint to get details of the authenticated user.

**OIDC and OAuth2 Relationship:**

* OAuth2 handles **authorization**, while OpenID Connect adds **authentication**.
* Together, they enable **Identity and Access Management (IAM)** for client applications.
* OIDC is **not a replacement** but an **extension** of OAuth2, enhancing it with authentication capabilities.

**Implementation**

+-------------------+

| AUTH SERVER |

| [KEYCLOAK] |

+-------------------+

▲

|

| (1) Client gets access token

|

+-------------------+

| External API |

| [CLIENT] |

+-------------------+

|

| (2) Client sends request with access token

v

+----------------------+

| Gateway/Edge |

| [RESOURCE SERVER] |

+----------------------+

|

| (3) Gateway validates token with Auth Server

|

v

+---------------------+

| Accounts Service |

+--------------------->|---------------------|<---------------------+

| | Loans Service | |

| |---------------------| |

| | Cards Service | |

+--------------------->+---------------------+<---------------------+

|

|

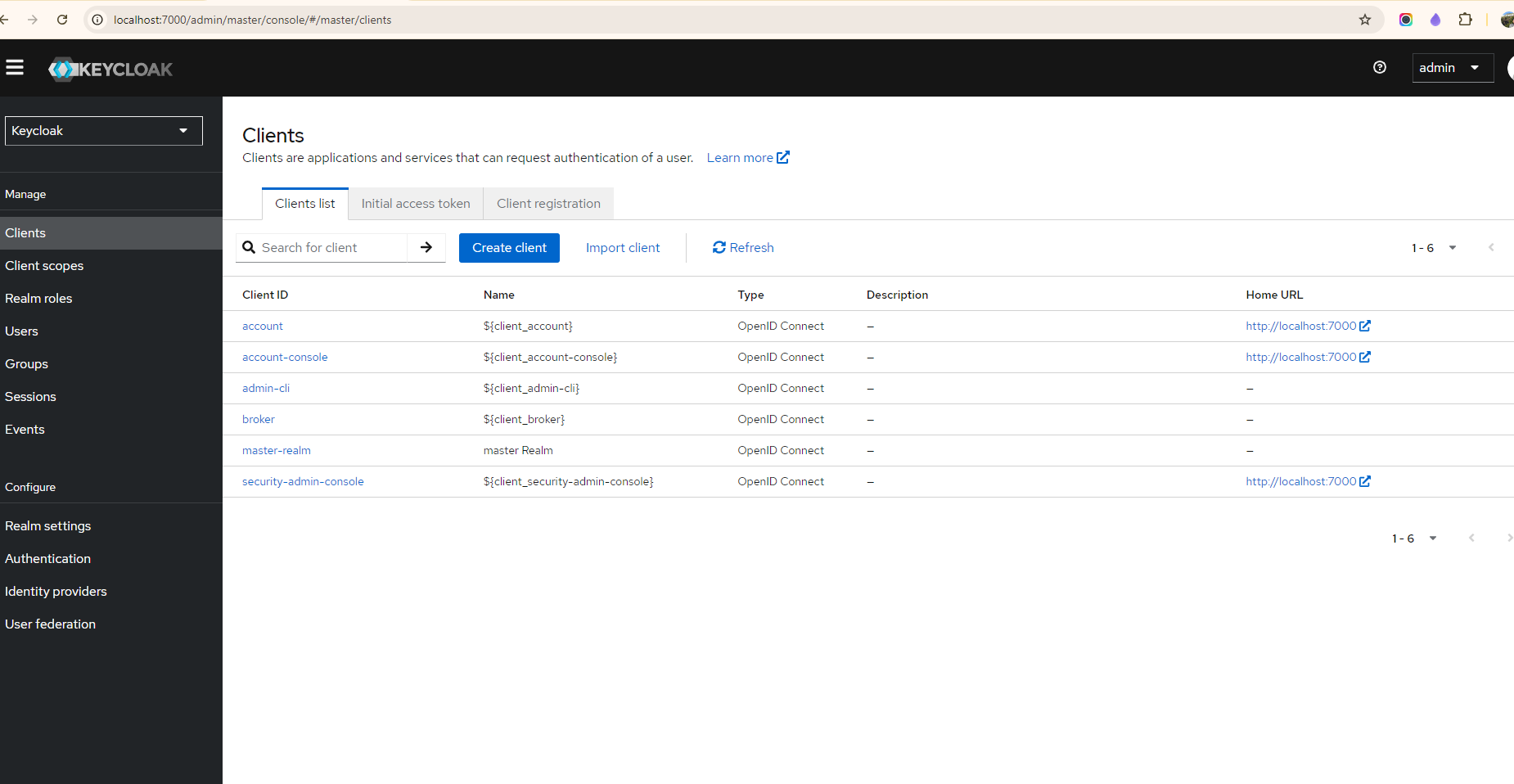
+------------------v-----------------+

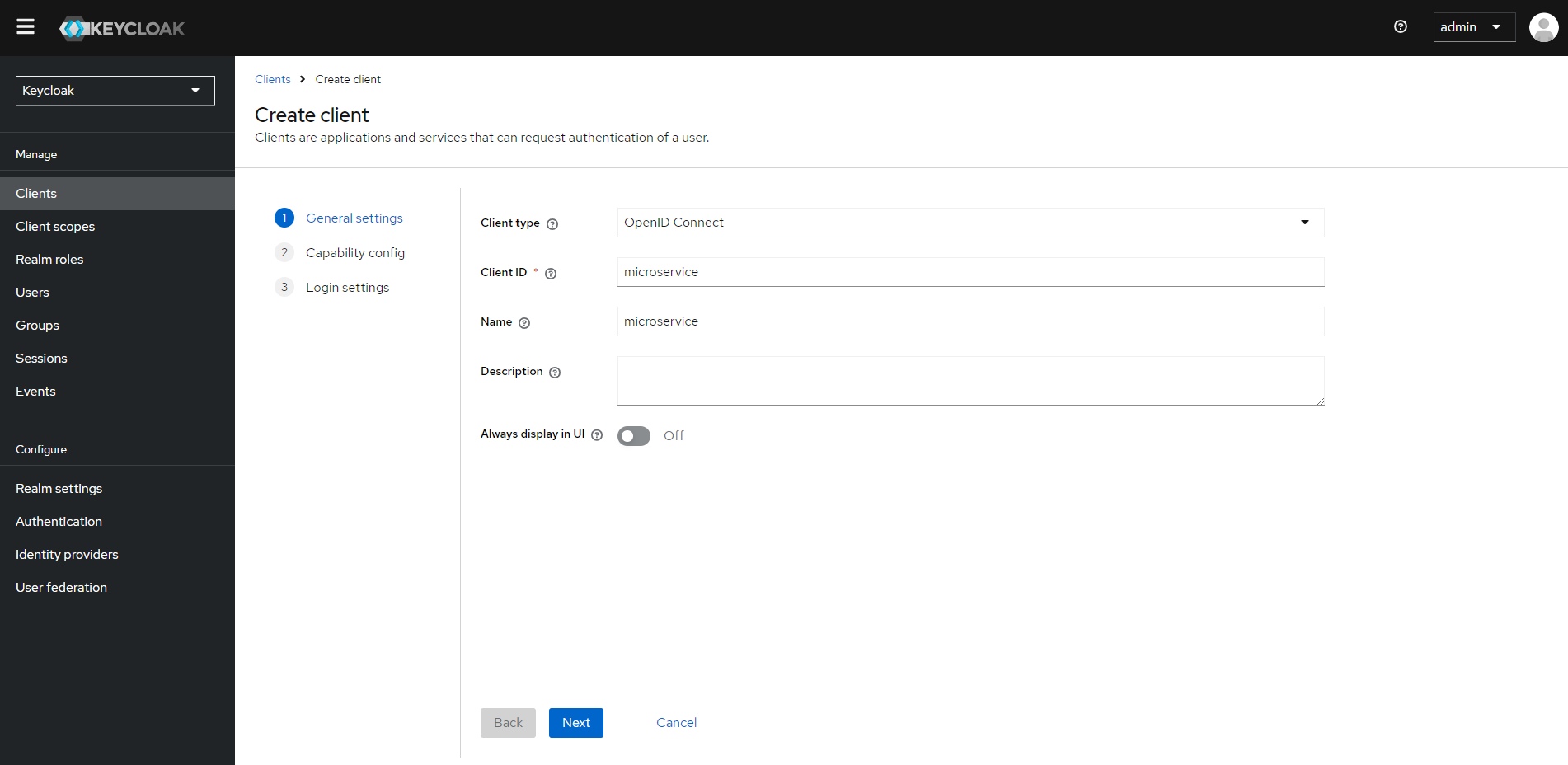
| Unsecured Services behind Firewall |

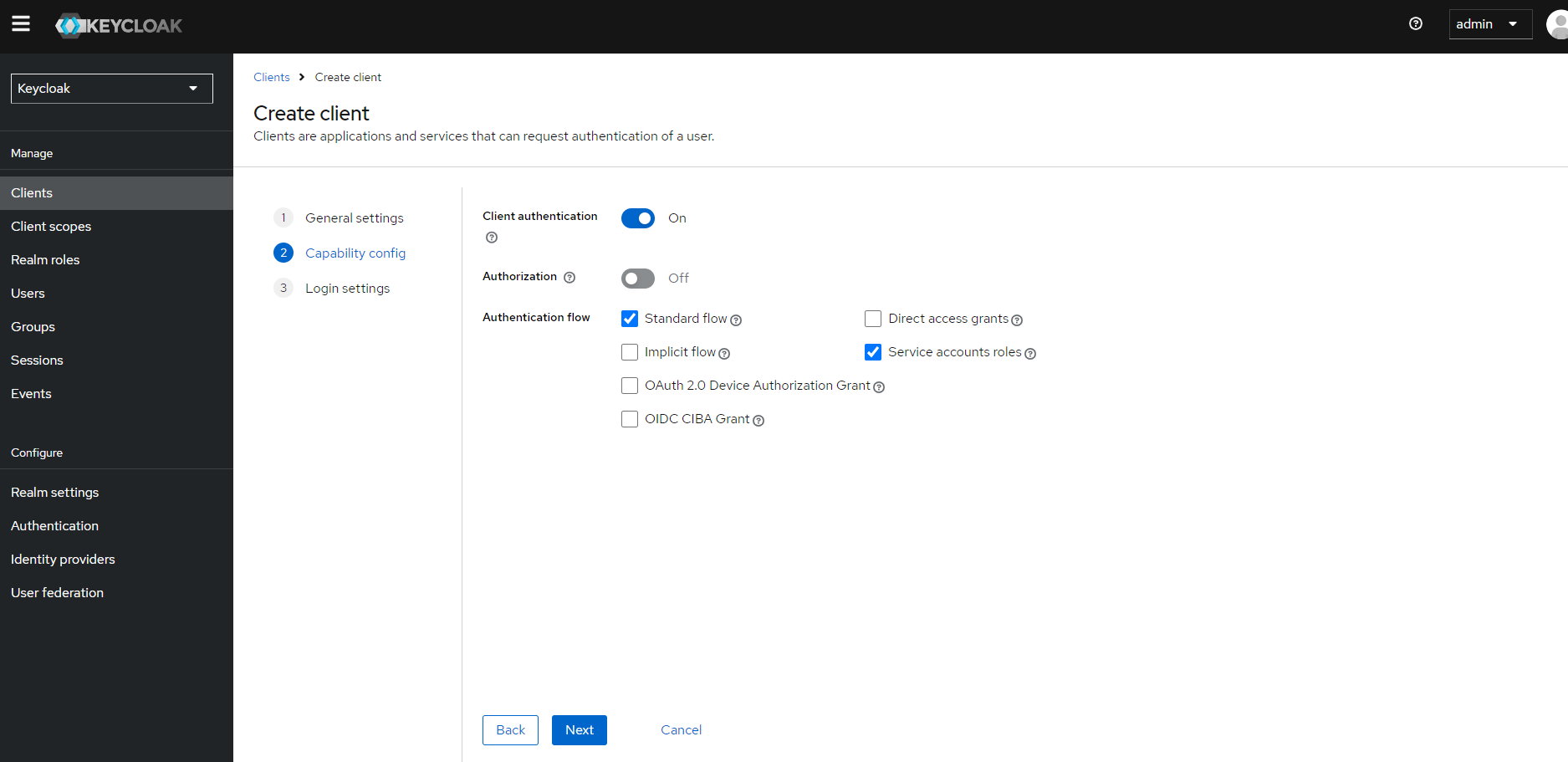
| (Docker/Kubernetes Network) |

+-------------------------------------+

1. **sClient Application**: An external service or API attempts to communicate with the microservices behind a **gateway server** (acting as the edge server).
2. **Authentication with Auth Server**: The client first interacts with an **auth server** (set up using Keycloak) to obtain an access token. Before this, the client must be registered and approved by the Keycloak admin.
3. **Gateway Server as Resource Server**: The **gateway server** is configured as a **resource server**, meaning it requires a valid access token to process incoming requests. Once the access token is received, the gateway server validates it with the auth server.
4. **Forwarding Requests to Microservices**: Once the access token is validated, the gateway server forwards the request to the appropriate microservices (e.g., accounts, loans, and cards).
5. **Securing Microservices**: To prevent direct access to individual microservices by external clients, the microservices are deployed behind a firewall or in a Docker network, accessible only via the gateway server.
6. **Why Not Resource Servers for Each Microservice?**: Making each microservice a resource server would add unnecessary complexity and reduce performance since each microservice would need to validate the access token for every internal request. Instead, securing communication between microservices can be handled with other methods, which will be covered later.
7. **Scenario**: In the **client credentials flow**, no user is involved. External applications authenticate using client credentials (client ID and secret) to get an access token. This token is then used to invoke APIs exposed by the gateway server.









Adding dependency in Gateway server

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-oauth2-resource-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-oauth2-jose</artifactId>

</dependency>

Add security configuration in Gateway

@Configuration

@EnableWebFluxSecurity

public class SecurityConfiguration {

@Bean

public SecurityWebFilterChain spSecurityWebFilterChain(ServerHttpSecurity serverHttpSecurity) {

serverHttpSecurity.authorizeExchange(authorizeExchangeSpec ->

authorizeExchangeSpec

.pathMatchers("/actuator/\*\*").permitAll()

.pathMatchers("/authentication/user/private/\*\*").authenticated()

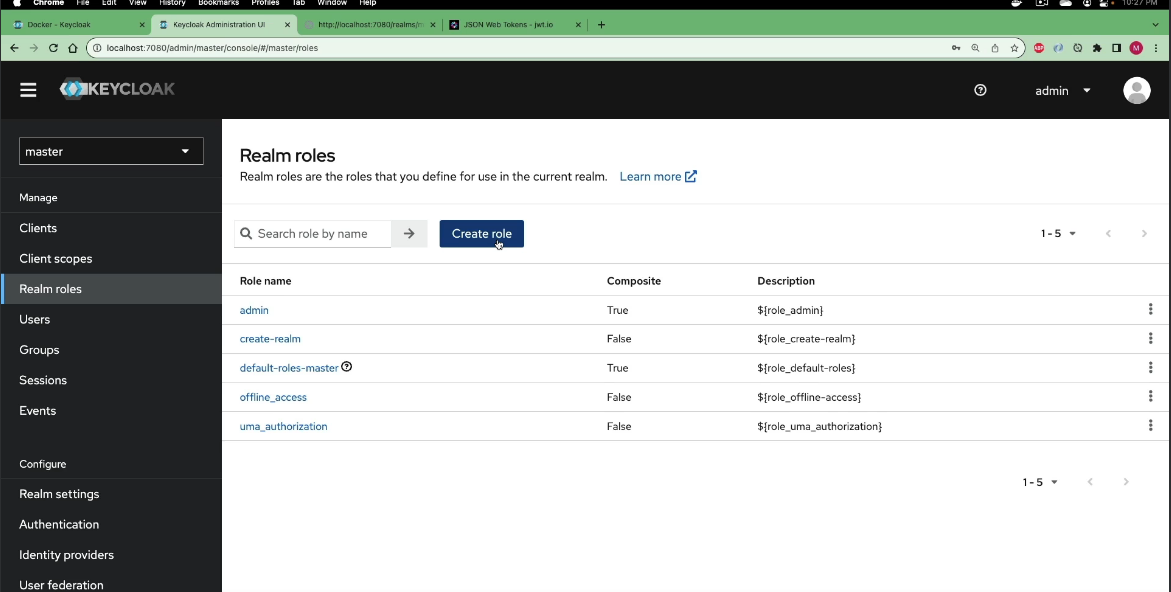
.pathMatchers("/authentication/\*\*").permitAll())

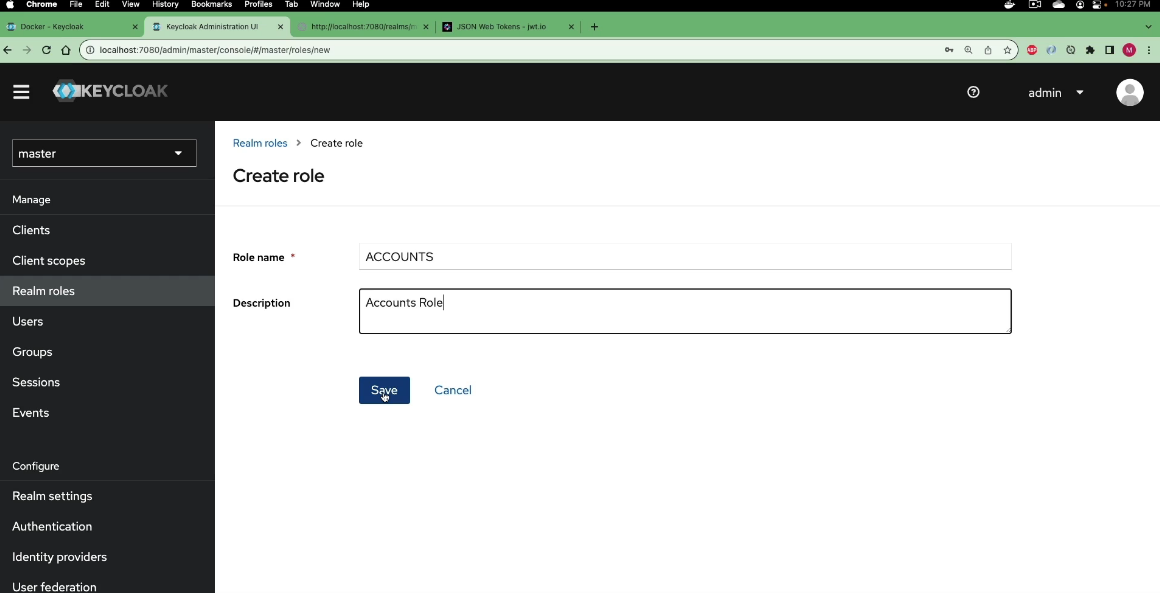
.oauth2ResourceServer(oAuth2ResourceServerSpec -> oAuth2ResourceServerSpec.jwt(Customizer.withDefaults()));

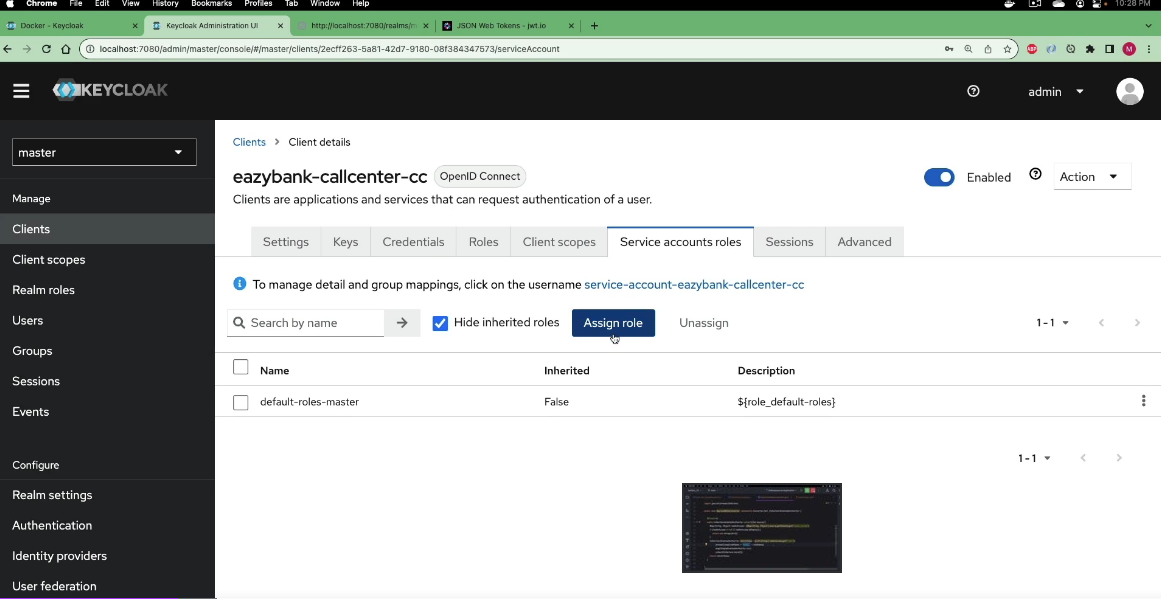
return serverHttpSecurity.build();

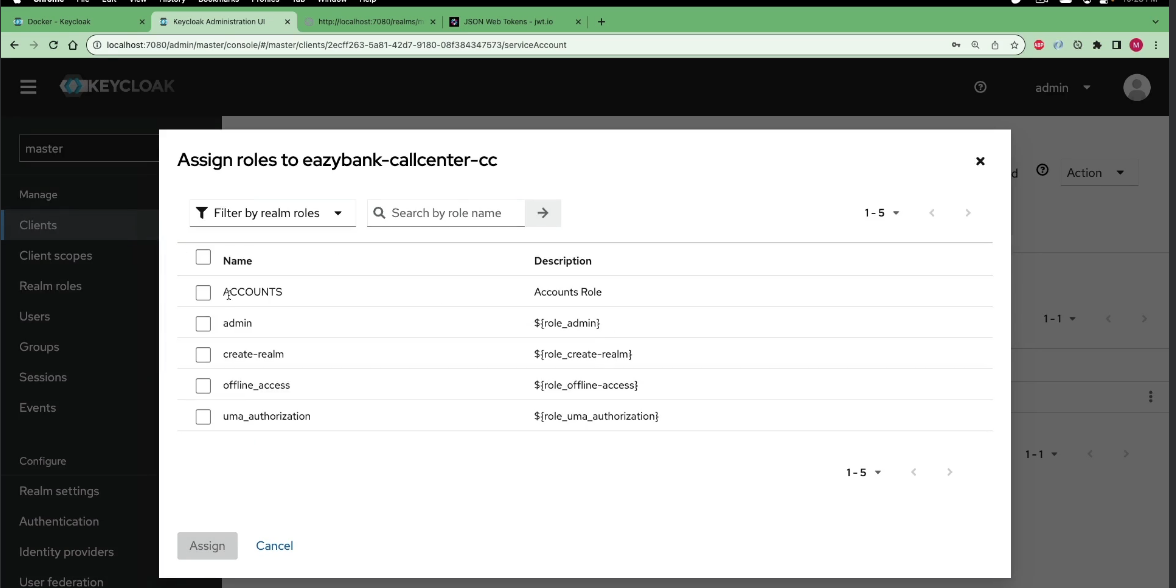
}

spring.security.oauth2.resourceserver.jwt.jwk-set-uri=http://localhost:7080/realms/master/protocol/openid-connect/certs

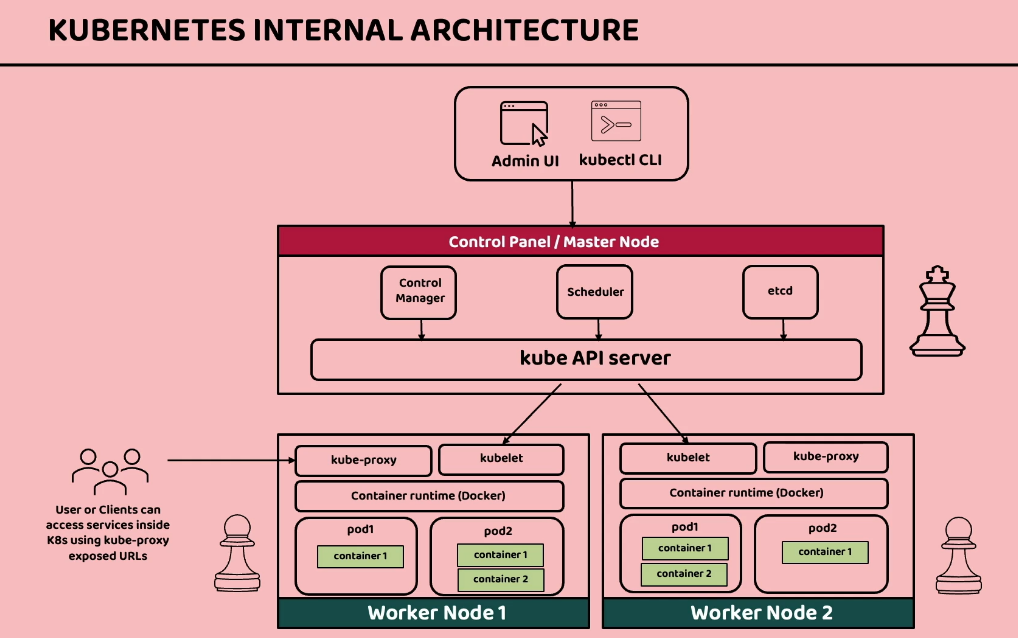








# Orchestration



Master node - For controlling worker

Kube API server- It will expose API to interact with outside world also used to interact between master and worker nodes

Kubectl CLI/ Admin UI – Provide input to API server in master node using YAML file

Scheduler - Takes input from API server which will help to identify to which worker node the changes has to be done

Control Manager - Monitor the health status of worker nodes and replace them in case of any error and kills/bring up new nodes if required

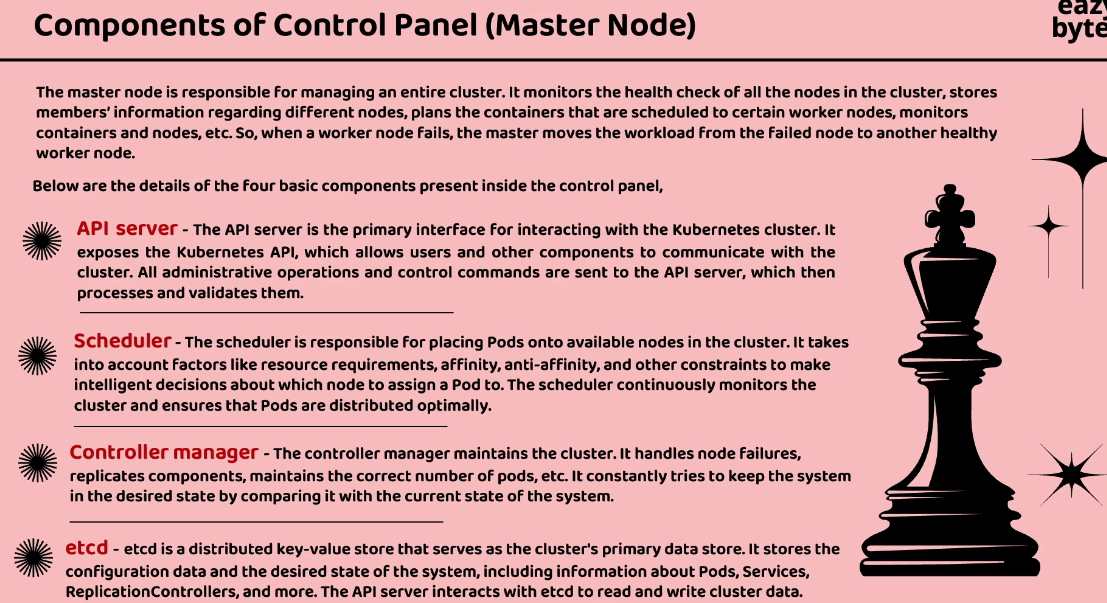
etcd - DB/Storage for the status of workers

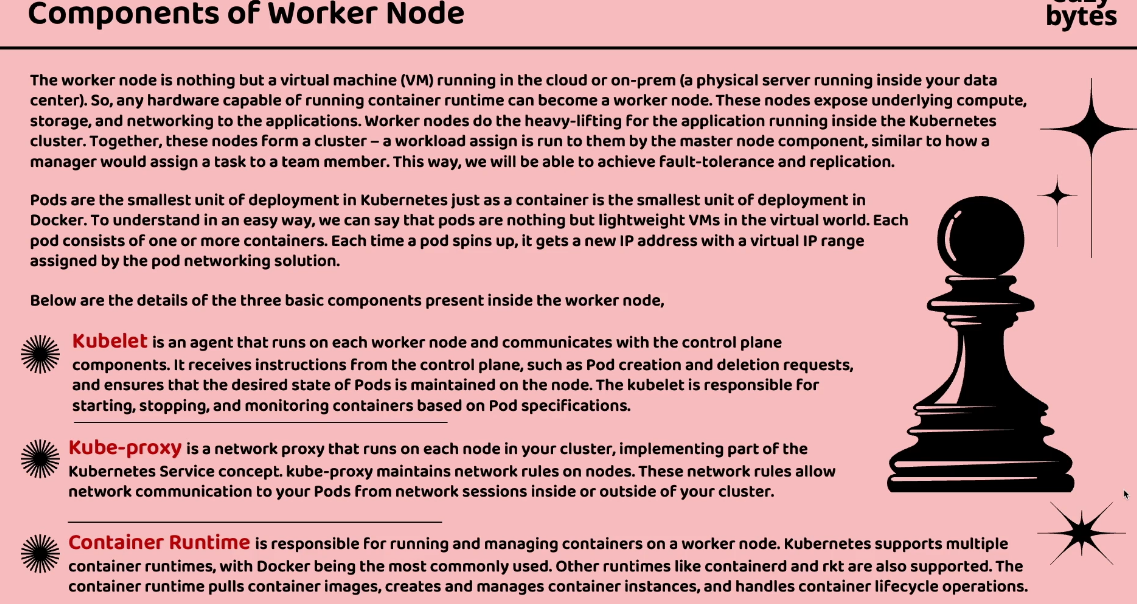
Worker node - responsible for sending traffic to microservice

Kublet - Agent running in worker through which worker connects and execute instructions

Container runtime- Docker or some docker like containerization tool

Pod - Smallest deployment unit. Always contain a single microservice unless in very rare scenario a helper container is needed





kubectl config get-contexts

kubectl config get-clusters

kubectl config use-context docker-desktop

kubectl get nodes

Kubernetes dashboard is not included by default and we need to set it up first from the helm. So install helm first then add dashboard using below commands

helm repo add kubernetes-dashboard https://kubernetes.github.io/dashboard/

helm upgrade --install kubernetes-dashboard kubernetes-dashboard/kubernetes-dashboard --create-namespace --namespace kubernetes-dashboard

Once started we have to do port forwarding

kubectl -n kubernetes-dashboard port-forward svc/kubernetes-dashboard-kong-proxy 8443:443

We can access dashboard <https://localhost:8443/#/logi>

Now need to create admin user by creating a file dashboard-adminuser.yaml

apiVersion: v1

kind: ServiceAccount

metadata:

name: admin-user

namespace: kubernetes-dashboard

And run command

kubectl apply -f dashboard-adminuser.yaml

Next have to do role binding create file and run kubectl apply -f filename.yml

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: admin-user

roleRef:

apiGroup: rbac.authorization.k8s.io

kind: ClusterRole

name: cluster-admin

subjects:

- kind: ServiceAccount

name: admin-user

namespace: kubernetes-dashboard

kubectl -n kubernetes-dashboard create token admin-user

