[**https://www.youtube.com/watch?v=QrPvGdpcjXo&list=PL0zysOflRCelb2Y4WOVckFC6B050BzV0D**](https://www.youtube.com/watch?v=QrPvGdpcjXo&list=PL0zysOflRCelb2Y4WOVckFC6B050BzV0D)

**MicroService**

-----------------------------------------------------------------------

-> Monolithic Architecture

Single code base.

Deployed in Single bundle

One service deployment need to deploy complete application

Building problems

Problem in scale

Difficult to achieve the different code base.

-> MicroService

Divide large applications into smaller parts.

Different code base, tech stack.

Modules are managed independently.

Handle microservices are complex

Example: Hotel Review System

- Hotel Service - store hotel information - use database1

- User Service - store the users information - use database2

- Rating Service - store the rating data - use database3

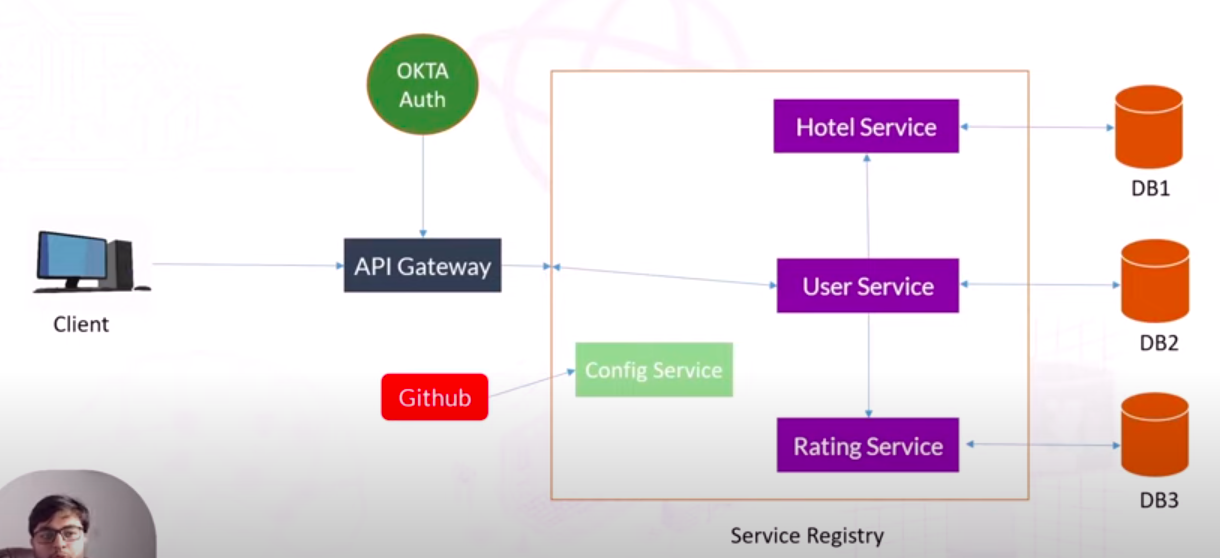
- Config Service - use to store the configuration - store in the github

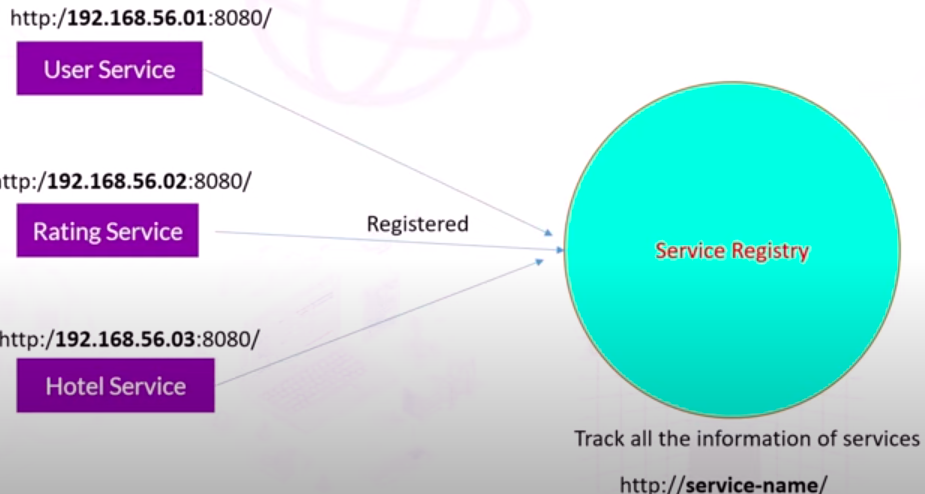
- API Gateway - That is the middle point to navigate the request from client to service.

Ex: client request for the Users details -> APIGateway -> User Service

- Authentication: User Okta outh in then API Gateway.

- Service Registory/Service Discovery: Need to register all the services. (Eureka Server/Client)





UserServiceDatabase in MySQL

HotelServiceDatabase in Postgres SQL

RatingServiceDatabase in MongoDB

@SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })

Application.properties:

spring:

application:

name: RATING-SERVICE

data:

mongodb:

uri: mongodb://localhost:27017

database: ratingmicroService

eureka:

instance:

prefer-ip-address: true

client:

fetch-registry: true

register-with-eureka: true

service-url:

defaultZone: http://localhost:8761/eureka

We can call the Rating and Hotel service from the User service using some HTTP client.

* RestTemplate
* FeignClient

Make the Host and Port Replace with the Service Name:

Need to annotation @LoadBalanced on RestTemplate Configuration.

**->** **FeignClient:**

Http Web Client developed by Netflix

Declarative approach.

Create Interface and annotate with the **@FeignClient**

Add dependency of Feign client in the project.

<dependency>

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

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

Add **@EnableFeignClients** in Main Application file.

**@FeignClient(name = "HOTEL\_SERVICE")**

**Interface XYZService{**

**@GetMapping(path = "/hotels/{id}")**

**Hotel getHotelById(@PathVariable("id") Long id);**

**}**

**API GATEWAY:**

Use spring cloud gateway.

Dependencies:

Cloud Bootstrap, Gateway, Spring Reactive Web, Eureka Discovery Client

**Config Server:**

Store the configuration globally.

Create one repository in git and app the url in the config service property file.

<http://192.168.12.183:8085/application/default>

1. Create one Config service that import the configuration form the git repository.
2. Add the Config client repository in the application that we wanted to use configurations.

<dependency>

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

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

Add property **spring.config .import** in properties file.

spring:

config:

Import: optional:configserver:<http://localhost:8085>

Optional is used for the optional parameter.

**Fault Tolerance:**

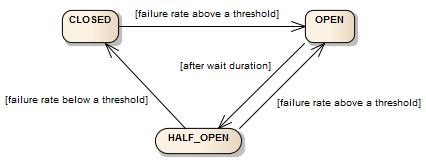
How to know which dependant API is breaked.

Circuit Breaker:

States of the Circuit breaker: close, open, halfopen, disable and forced\_open

Open Senario: Call api for number of threshold if threshold reached and response not return then API is down and circuit is open.

HalfOpen: all api for number of threshold if threshold reached and response comes for the 5 time in 10 succeeding time then circuit is HalfOpen.



Hystrix and Resilience 4J is the libraries to use circuit breaker.

Currently Resilience 4J library is used. Before that we use the Hystrix.

<!-- Depencenies for the Foult tollerence -->

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>

</dependency>

<!-- Depencenies for the Foult tollerence -->

* Add circut breaker annotation above the API that call another api

@CircuitBreaker(name = "ratingHotelCirciteBreaker", fallbackMethod = "**ratingHotelCirciteBreakerHandler**")

Create fallbackMethod that call when the service is down on another API.

public ResponseEntity<User> ratingHotelCirciteBreakerHandler(String userId, Exception e) {

log.error("Error is : " + e.getMessage());

User user = new User().builder().about("dummy").email("dummuEmail").name("dummy").ratings(null).userId(0L)

.build();

return new ResponseEntity<User>(user, HttpStatus.BAD\_REQUEST);

}

**Make the Configuration for the circut breaker (#Resilience4j Configurations)**

#acctuator

management:

health:

circuitbreakers:

enabled: true

endpoints:

web:

exposure:

include: health

endpoint:

health:

show-details: always

#Resilience4j

resilience4j:

circuitbreaker:

instances:

ratingHotelCirciteBreakers:

register-health-indicator: true

event-consumer-buffer-size: 10

failure-rate-threshold: 50

minimum-number-of-calls: 5

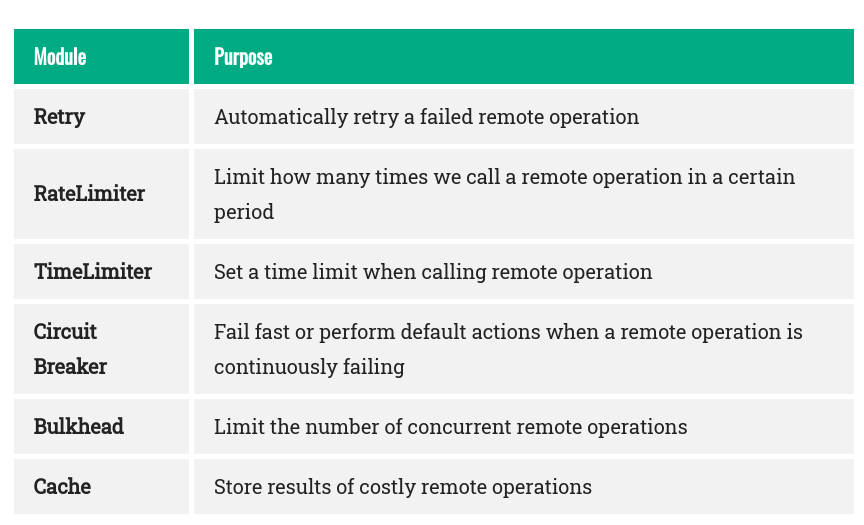
automatic-transition-from-open-to-half-open-enabled: true

wait-duration-in-open-state: 6s

permitted-number-of-calls-in-half-open-state: 3

sliding-window-size: 10

sliding-window-type: count-based



**Bulkhead**: limit the number of concurrent execution.

Restrict user to perform limitated process in the threading environment.

**Retry:**

Rety is used when there is some network issue at Another API side.

It the API is down then we don’t need to retry. We use retry when the API is slow network or we are not sure for the API.

@Retry(name = "ratingHotelRetry", fallbackMethod = "**ratingHotelRetryHandler**")

retry:

instances:

ratingHotelRetry:

max-attempts: 3

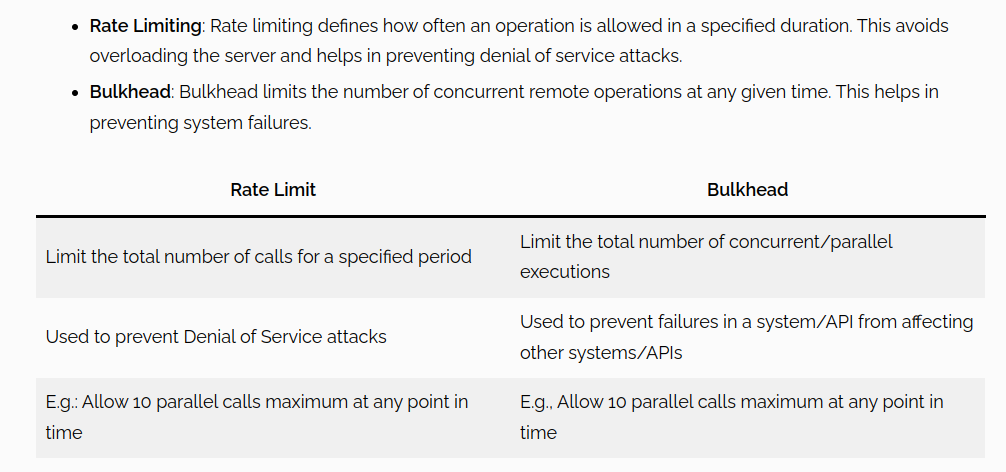
wait-duration: 5s

**Rate Limiter:**

Limiting the access to some services.

Limit the number of call we could process in some window.

We improve the Performance = restrict the call based on the server limit and Security= restrict the DOS Attect(Many request to the APIs) by using the Rate Limiter.



**Time Limiter:**

Example: time limit of 2s for the flight search call. In other words, if the call doesn’t complete within 2s, we want to be notified through an error.

resilience4j:

instances:

basicExample:

timeoutDuration: 2s

@TimeLimiter(name = "basicExample", fallbackMethod=”TimeLimiterException”)

Three ways:

Request per second, Request per minutes, Request per Hours

@RateLimiter(name = "userRateLimiter", fallbackMethod = "rateLimitorHandler")

**Authentication In the MicroService:**

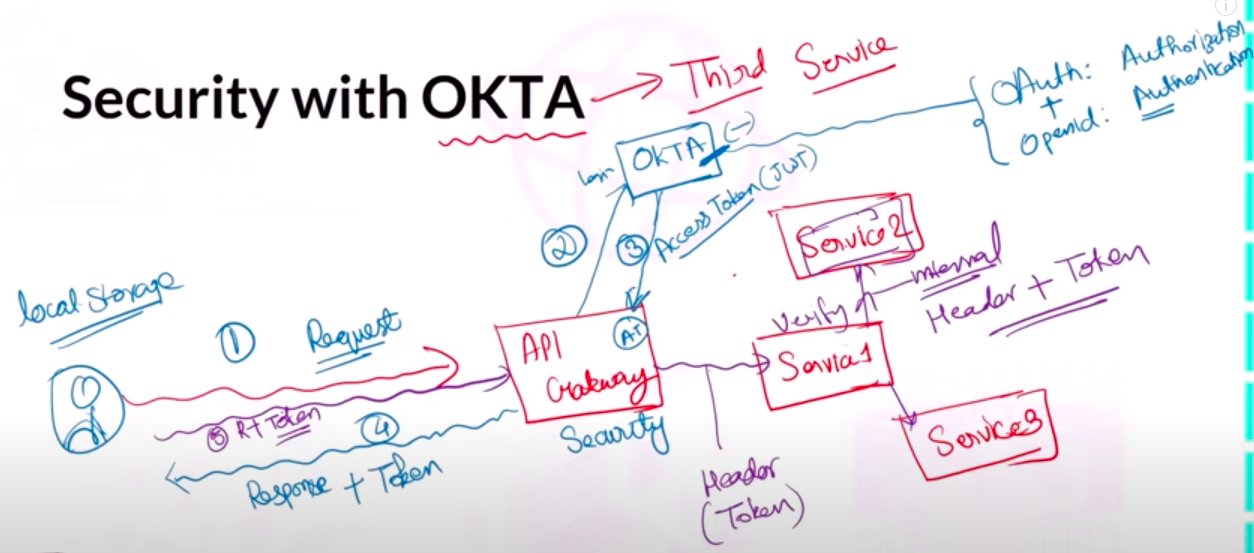
JWT Tokenbased authentication: pass the token in header and validate that. (JWT TOKEN = Header + payload + Signature)

OKTA: Third party service to validate the request.

OKTA = Oauth + openId

Oauth: Authorization (Which service user eligible to access)

openId: Authentication (User is valid to access the service)



OKTA

ClinetId: 0oa97emmkrthqJjaA5d7

Secret Key: YeJTvNHsVMA\_Cd5LsyISXw5c2in2Fy\_UIj27dGmk

Create Users, Groups, Assign the Users and Groups to the Application.

Inside API add the scopes, claims for the Internal API calling

Add properties in the application.properties file of the api gateway, user service

Security for calling the API form the another API not called directly.

@PreAuthorized

**CACHING:**

Caching store the frequently access data, objects and images near to the database to perform fast access and reduce the cost for access that.

**WHERE TO USE:**

Used when the repeated data data return the same output.

Used when there is the multiple Hierarchical APIs in microservice. (Ex: service1 -> serivce2 -> service3 …. Return response …… -> service1).

Used when we call the External services like user validation paid services. We can validate user once and store in the cache.

**Differente cache providers:**

**Ehcache 3**

Add two dependencies:

<dependency>

<groupId>javax.cache</groupId>

<artifactId>cache-api</artifactId>

</dependency>

<dependency>

<groupId>org.ehcache</groupId>

<artifactId>ehcache</artifactId>

<version>3.10.8</version>

</dependency>

@Bean

public CacheManager EhcacheManager() {

CacheConfiguration<Long, User> cachecConfig = CacheConfigurationBuilder

.newCacheConfigurationBuilder(Long.class, User.class,

ResourcePoolsBuilder.newResourcePoolsBuilder().offheap(10, MemoryUnit.MB).build())

.withExpiry(ExpiryPolicyBuilder.timeToIdleExpiration(Duration.ofSeconds(10))).build();

CachingProvider cachingProvider = Caching.getCachingProvider();

CacheManager cacheManager = cachingProvider.getCacheManager();

javax.cache.configuration.Configuration<Long, User> configuration = Eh107Configuration

.fromEhcacheCacheConfiguration(cachecConfig);

cacheManager.createCache("user", configuration);

return cacheManager;

}

Hazelcast

Infinispan

Couchbase

Redis:

Add new dependencies:

<dependency>

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

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

</dependency>

Application.yml

spring:

cache:

redis:

cache-null-values: true

use-key-prefix: true

key-prefix: usercache

time-to-live: 60000

data:

redis:

host: localhost

port: 6379

timeout: 10000

Caffeine

Pivotal GemFire

Caching size is limited so we have the different stretorgy to stoer the data in the caching.

-> Least Recently Used, Least Frequently Used.

What the Things we need to do For the Caching Enable.

* Enable the Cahing in Application
* Add the data in the Cache.
* When there is a Update the data we need to update that data in the cache.
* Evict data: When we do not require the data we need to clean that data.

Annotation Used in the Spring Boot:

@EnableCaching = at class level,

@Cacheable= used with the method that are cacheble,

@Cacheable({"user","hotel","rating"}): give the multiple names to the cache

@Cacheable(cacheNames = "user", key = "#id", condition = "#id > 0"): give cache name key and condition where we need to store the cache.

**Synchronization Caching**: user when same cache used in multiple methods simultaneously.

@Cacheable(cacheNames = "user", **sync = true**):

**Cache Manager:**

**Cache Resolver:**

@CachePut: used to update the cache. Usevaly used when update call is made in entity.

@CachePut(cacheNames = "user", key = "#id")

**Different between @Cacheable and @CachePut is,**

If the find result are in the cache then Cacheable not execute the method.

In CachePut it always executes the method is in cache or not and update the info in cache.

@CacheEvict = clears the cache values from the cache storage,

@CacheEvict(cacheNames= “user”, allEntries = true)

allEntries is remove all the entries in the cache.

@Caching= used to support multiple caching annotation with the same type. Ex: @CacheEvit or @CachePut.

Like we need to delete from two cache and update in the two cache.

@CacheConfig: it is the class level annotation, it used when we need to implement the same caching mechanism for all the methods that are available in that class.

Different cache provider

JDK-ConcurrentMap Based Cache

Ehcache-based cache

Caffeine cache

Setting Eviction/TTL Policy: Use diectly from the cache provider.

Cache abstration is and abstraction not cache implementation.

**Customizing Spring Boot Cache Manager**

Using creating the custom “CacheManagerCustomizer”.

**Ehcahce:**

Open source, Commercial support by terracota, robust full feature and also integrate with the other libraries as well.

First Level Cache: By defaul available in the Hibernet. (Session Object)

Second Level Cache: Need to implement using the Ehcahce. (SessionFactory: we can user that in any where in the application. Session Factory contains multiple session object). If on session have

Cache hit: find data find in the cache.

Cache miss: find data don’t find in the cache.

System of record: database, third party services and provides.

**Factors that affect the efficiency of Cache:**

Liveliness: data is not present in the cache then how frequently the data is synced.

Portion (How much data are cached) of data caches

Shape of usage distribution (80/20): 20% data are found and 80% data are not in cached when request.

Read/Write ratio.

**Topology:**

Standalone: same server store in the Standalone

Distributed: Different server for storing the caching data

Replicated: suppose we have the three application then all the data is replicate to the all the application cache.

**Storage tier:**

Memory Store: Subject to GC

offHeap Store: Available to RAM

Disk Store: store in the file system

**Redis data caching:**

Redis means Remote Dlctionary Server.

Open Source

In Memory Data Structure: String, Hash, Set, Sorted set.

Build in Replication and LRU (Least Recently Used) algorithm, transaction, disk-level persistence.

Redis is used as database, caching, Message Brokers.

**MicroSevice Architeture:**

**Challenge face in the MicroService:** bigger challenge in the micro services are monitoring the service. If we have downone service then it is difficult to monitor the performance of the APIs.

Actuator is there to monitoring the performance.

**Semantic monitoring in microservice:** (a.k.a. synthetic monitoring) runs a subset of an application's automated tests against the live production system on a regular basis. The results are pushed into the monitoring service, which triggers alerts in case of failures

**Distributed transaction: (Not Possible to achieve the ACID Properties)**

It is the sequence of the transaction are need to perfore to complete on transaction.

Two phase locking stretorgy is used to achieve the ACID properties.

**Transaction States:** Active, Partially Commit, Commit, terminate, abort, failed