What are microservices?

An architectural style that structures an application as a collection of services that are

1. Highly maintainable

2. Loosely coupled

3. Independently deployable

4. Organized around business capabilities

The goal of the microservice architecture is to accelerate software development by enabling continuous delivery/deployment & deliver Viable Product.

Monolithic Architecture

The Monolithic application describes a N-tiered software application in which different components combined into a single program from a single platform.

Explain how EAR & WAR files are packed (View , Serivce , DAO, RestAPI , Email) for one single platform.

Advantages :

Simple to develop

Simple to deploy

Simple to scale

Simple to test (testing UI will cover all the back end modules as well)

Incident Analysis is very simple. As there is no need for log aggregation.

DownSide :

Scaling the application can be difficult (can only scale in one dimension, With a monolithic architecture we cannot scale each component independently)

Continuous deployment is difficult (As team size grows, frequent deployment may cause lot of downtime). Build itself has dependecy on multiple team.

Smaller change requires the entire App to be brought down & deployment.

CICD is very difficult.

Micro Service :

Advantage :

Better deployability - services can be deployed independently without the entire app downtime.

Autonomous teams. (independent team increase fast delivery)

Experment different tech stack in micro-service (JAVA API & NODE API)

Fast Delivery.

DisAdvantges:

Developers must deal with the additional complexity of creating a distributed system

Developers must implement the inter-service communication mechanism and deal with partial failure

maintaing transaction accross mutiple service call & failure secarios.

Exception tracking (Service calls A -> Service Call B -> Service C thrown Exception)

Distributed/Centralized Session Handling

End to End Testing & testing the interactions between services is more difficult (Will have mock every other dependent service)

Implementing requests that span multiple services requires careful coordination between the teams

Service Aggregation ( aggregate result from multiple DB schema & return restult to consumer)

testing of App will be difficult. (test individual API & Consumer UI)

Decomposing technique:

Decompose by business capability : Define services corresponding to business capabilities. (Product Service, User Service, Enrollment, Formulary Service)

Deployment patterns :

Multiple service instances per host (i'm using this pattern)

Service instance per host

Service instance per VM

Service instance per Container

Cross cutting concerns:

API gateway::::

is the single entry point for all clients. The API gateway handles requests in one of two ways.

the API gateway can expose a different API for each client.

Advatage :

Insulates the clients from the problem of determining the locations of service instances

Reduces the number of requests/roundtrips from Client prospective.

Insulates the clients from how the application is partitioned into microservices

Best Place for implemt cross cuttinig concern like, logging, security,

DisAdvantages :

Sinle Point Failure.

Externalized configuration - Maintain All the Property in One Single Code base Using Sprinig Cloud Config.

Dedicated Service which tells other API how to connect to the external/3rd party services. & what are the business configurations.

It enables API to change the configuration without downtime.

Things to check : ensure that when an application is deployed the supplied configuration matches what is expected.

Encrypt the password in config file.

Service discovery

In Cloud environment, IP will be assigned dynamically. In frequent tear down, client will have to be aware of the location of the service.

Service discovery pattern enable to client to be aware of the location & API.

Client-side discovery - When making a request to a service, the client obtains the location of a service instance by querying a Service Registry,

which knows the locations of all service instances.

(There is no router involved in client side discovery)

Server-side discovery - When making a request to a service, the client makes a request via a router (a.k.a load balancer)

that runs at a well known location. The router queries a service registry, which might be built into the router,

and forwards the request to an available service instance.

Self registration - A service instance is responsible for registering itself with the service registry.

On startup the service instance registers itself (host and IP address) with the service registry and makes itself available for discovery.

The client must typically periodically renew its registration so that the registry knows it is still alive. On shutdown, the service instance unregisters itself from the service registry.

Reliability

Circuit Breaker - When one service synchronously invokes another there is always the possibility that the other service is unavailable or

is exhibiting such high latency it is essentially unusable.This might lead to resource exhaustion, which would make the calling service unable to handle other requests.The failure of one service can potentially cascade to other services throughout the application.

Data management:

Database per Service - For each service, there will be a schema. (Service will access the its own database. if the service requires data in Schema B, it should call Service B instead of directly calling it)

Event Sourcing - (Event sourcing persists the state of a business entity such an Order or a Customer as a sequence of state-changing events. Whenever the state of a business entity changes, a new event is appended to the list of events. Since saving an event is a single operation, it is inherently atomic. The application reconstructs an entity’s current state by replaying the events.)

CQRS - Command Query Responsibility Segregations

Security

Access Token\*

Observability

Health check API - Dedicated API for checking the application/API status. API Status includes - DB health, MQ Server Health, Other dependent service health.

Log aggregation -

Distributed tracing -

Log aggregation -

Testing

Service Component Test

Consumer-side contract test

CLOUD-BASED COMPUTING TYPES :

Infrastructure as a Service (IaaS)

Platform as a Service (PaaS)

Software as a Service (SaaS)

======================================================================================================================================================================

======================================================================================================================================================================

Spring Cloud Config

::::::::::::::::::::::::::::::Config API Impl:::::::::::::::

@SpringBootApplication

@EnableConfigServer

public class ConfigServerApplication {

public static void main(String[] args) {

SpringApplication.run(ConfigServerApplication.class, args);

}

}

server:

port: 8888

spring:

application:

name: licensingservice

profiles:

active: native

cloud:

config:

server:

native:

searchLocations: file:///Users/johncarnell1/book/native\_cloud\_apps/ch4-config-managment/confsvr/src/main/resources/config/licensingservice

uri: http://localhost:8888

repos:

appFoo:

pattern: app-foo

uri: https://github.com/solivaf/config-properties-bar

::::::::::::::::::::::::::::::Client Impl::::::::::::::::::: (–Dspring.profiles.active=dev)

server:

port: 8888

spring:

profiles:

active: native

cloud:

config:uri: http://localhost:8888 Specify the location of the Spring Cloud Config server

searchPaths: licensingservice,organizationservice

username: native-cloud-apps

password: 0ffended

No Specifi annotation required at the client side - on Main Class.

@Value("${example.property}") - Annotate the class with @RefreshScope

private String exampleProperty;

Secure Configuration.

1. export ENCRYPT\_KEY=IMSYMMETRIC

2. Use built in endpoint to encrypt/decrypt

3. Configure the encrypted values in config.(spring.datasource.password:"{cipher}858201e10fe3c9513e1d28b33ff417a66e8c8411dcff3077c53cf53d8a1be360")

4. Configure Spring Cloud Config to not decrypt properties on the server side. (spring.cloud.config.server.encrypt.enabled: false)

Secure Client Config

1 Set the symmetric key on the licensing server.

2 Add the spring-security-rsa JARs to the licensing services pom.xml file

======================================================================================================================================================================

======================================================================================================================================================================

Service Discovery Using Eureka

::::::::::::::::::::Eureka Service Registry API:::::::::::::::::::::::::::::

How Service Discovery Works

1. Service registration - Whenever the service is bootstrapping(even after a recovery), they’ll register their physical location, path, and port

that they can be accessed by with one or more service discovery instances. While each

instance of a service will have a unique IP address and port, each service instance that

comes up will register under the same service ID. A service ID is nothing more than a

key that uniquely identifies a group of the same service instances.

2. Client lookup of service address -

A service will usually only register with one service discovery service instance. Most

service discovery implementations use a peer-to-peer model of data propagation

where the data around each service instance is communicated to all the other nodes

in the cluster.

3. Information sharing - -

4. Health monitoring - Any services failing to return a good health check will be removed from the pool of available service instances.

Eureka requires three consecutive heartbeat pings from the service spaced 10 seconds apart before it will say the service is ready for use.

Application ID : Used to represent the grooup of application( app id is configured using the prop spring.application.name in bootstrap.yml)

Instance ID : instance ID will be a random number meant to represent a single service instance (Random Number + App ID + Port Number)

Word on Client Side Load Balancing:::::

Client-side load balancing involves having the client look up all of a service’s individual instances from a service discovery agent (like Netflix Eureka) and then caching the physical location of said service instances.

Each time a client wants to call the service, the service consumer will look up

the location information for the service from the cache. Usually client-side

caching will use a simple load balancing algorithm like the “round-robin” load

balancing algorithm to ensure that service calls are spread across multiple service

instances.

The client will then periodically contact the service discovery service and

refresh its cache of service instances. The client cache is eventually consistent,

but there’s always a risk that between when the client contacts the service discovery

instance for a refresh and calls are made, calls might be directed to a service

instance that isn’t healthy.

:::::: In Eureka Server API

#Default port is 8761

server:

port: 8761

eureka:

client:

registerWithEureka: false

fetchRegistry: false

server:

waitTimeInMsWhenSyncEmpty: 30

serviceUrl:

defaultZone: http://localhost:8761

---

spring:

profiles: peer1

eureka:

instance:

hostname: peer1

---

spring:

profiles: peer2

eureka:

instance:

hostname: peer2

eureka.instance.preferIpAddress property tells Eureka that you want to register the service’s IP address to Eureka rather than its hostname.

NOTE : (eureka.instance.preferIpAddress = true)

@SpringBootApplication

@EnableEurekaServer

public class EurekaServerApplication {

public static void main(String[] args) {

SpringApplication.run(EurekaServerApplication.class, args);

}

}

:::::::: In Eureka Client

eureka:

instance:

preferIpAddress: true

client:

registerWithEureka: true

fetchRegistry: true

serviceUrl:

defaultZone: http://localhost:8761/eureka/

@SpringBootApplication

@EnableDiscoveryClient // required based on the client types.

@EnableFeignClients // @EnableFeignClients application aren’t needed when using the Ribbon backed RestTemplate and can be removed.

public class Application {

@LoadBalanced //Ribbon backed rest client. (Type 1 - ribbon backed client, no annoation required on Main Class)

@Bean

public RestTemplate getRestTemplate(){

return new RestTemplate();

}

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

--------------- (Type 2 - DiscoveryClient )

@Autowired

private DiscoveryClient discoveryClient;

public Organization getOrganization(String organizationId) {

RestTemplate restTemplate = new RestTemplate();

List<ServiceInstance> instances = discoveryClient.getInstances("organizationservice");

if (instances.size()==0) return null;

String serviceUri = String.format("%s/v1/organizations/%s", instances.get(0).getUri().toString(), organizationId);

ResponseEntity< Organization > restExchange = restTemplate.exchange( serviceUri, HttpMethod.GET, null, Organization.class, organizationId);

return restExchange.getBody();

}

--------------- // @EnableFeignClients

@FeignClient("organizationservice")

public interface OrganizationFeignClient {

@RequestMapping(method= RequestMethod.GET,value="/v1/organizations/{organizationId}",consumes="application/json")

Organization getOrganization(@PathVariable("organizationId") String organizationId);

}

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

===================================================================================================================================================================================

===================================================================================================================================================================================

client-side resiliency patterns

1 Client-side load balancing (Covered above)

2 Circuit breakers - Monitor the service calls for given period of time and try to detect failure. If the number of failures reach the threshold, inteercept the service call and throw error for given period of buffer time. this buffer time is for application resource to recover from resource outage.

3 Fallbacks - when a remote service call fails, rather than generating an exception, the service consumer will execute an alternative code path and try to carry

out an action through another means.

4 Bulkheads -

you can break the calls to remote resources into their own thread pools and reduce the risk that a problem with one slow remote resource call will take down the entire application.

Benefits : Fail fast, Fail gracefully, Recover seamlessly

Hystrix :

@SpringBootApplication

@EnableCircuitBreaker

@EnableEurekaClient

public class Application {

@LoadBalanced //Ribbon backed rest client. (Type 1 - ribbon backed client, no annoation required on Main Class)

@Bean

public RestTemplate getRestTemplate(){

return new RestTemplate();

}

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

@HystrixCommand

public List<License> getLicensesByOrg(String organizationId){

return licenseRepository.findByOrganizationId(organizationId);

}

When the Spring framework sees the @HystrixCommand, it will dynamically generate a proxy that will wrapper the method and manage all calls to that method through a thread pool of threads specifically set aside to handle remote calls.

By default, when you specify a @HystrixCommand annotation without properties, the annotation will place all remote service calls under the same thread pool.

@HystrixCommand(fallbackMethod = "buildFallbackLicenseList",

threadPoolKey = "licenseByOrgThreadPool",

threadPoolProperties = {@HystrixProperty(name = "coreSize",value="30"),@HystrixProperty(name="maxQueueSize", value="10")},

commandProperties={@HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds",value="12000")},

commandPoolProperties ={@HystrixProperty(name="circuitBreaker.requestVolumeThreshold", value="10"),

@HystrixProperty(name="circuitBreaker.errorThresholdPercentage", value="75"),

@HystrixProperty(name="circuitBreaker.sleepWindowInMilliseconds",value="7000"),

@HystrixProperty(name="metrics.rollingStats.timeInMilliseconds", value="15000")

@HystrixProperty(name="metrics.rollingStats.numBuckets", value="5")})

public List<License> getLicensesByOrg(String organizationId){

randomlyRunLong();

return licenseRepository.findByOrganizationId(organizationId);

}

private List<License> buildFallbackLicenseList(String organizationId){

return ListofLicense;

}

execution.isolation.thread.timeoutInMilliseconds is used to set the length of the timeout

circuitBreaker.requestVolumeTheshold, controls the amount of consecutive calls that must occur within a 10-second window before Hystrix will consider tripping the circuit breaker for the call.

circuitBreaker.errorThresholdPercentage, is the percentage of calls that must fail (due to timeouts, an exception being thrown, or a HTTP 500 being returned) after the circuitBreaker.requestVolumeThreshold value has been passed before the circuit breaker it tripped.

circuitBreaker.sleepWindowInMilliseconds, is the amount of time Hystrix will sleep once the circuit breaker is tripped before Hystrix will allow another call through to see if the service is healthy again.

metrics.rollingStats.timeInMilliseconds, is used to control the size of the window that will be used by Hystrix to monitor for problems with a service call.

metrics.rollingStats.numBuckets, controls the number of times statistics are collected in the window you’ve defined.

Hystrix supports two isolation models: THREAD and SEMAPHORE.

 Hystrix’s default isolation model, THREAD, completely isolates a Hystrix protected

call, but doesn’t propagate the parent thread’s context to the Hystrix

managed thread.

 Hystrix’s other isolation model, SEMAPHORE, doesn’t use a separate thread to

make a Hystrix call. While this is more efficient, it also exposes the service to

unpredictable behavior if Hystrix interrupts the call.

 Hystrix does allow you to inject the parent thread context into a Hystrix managed

Thread through a custom HystrixConcurrencyStrategy implementation.

======================================================================================================================================================================

======================================================================================================================================================================

Gateway

The API gateway handles requests in one of two ways. Requests are simply proxied/routed to the appropriate service.

Need for Gateway :

Insulates the clients from the problem of determining the locations of service instances

Load Balancing through gateway. (As service are registred in Service Registry, Zuul will automatically spread out the load to different instance of the same API request.)

Might also implement security, e.g. verify that the client is authorized to perform the request

Might also implement centralized logging/Auditing.

A service gateway acts as an intermediary between the service client and a service being invoked. The service client talks only to a single URL managed by the service gateway.

Static routing—A service gateway places all service calls behind a URL and API route.

Dynamic routing—A service gateway can inspect incoming service requests and, based on data from the incoming request, perform intelligent routing based on

who the service caller is.

Authentication and authorization -

Metric collection and logging -

Routing in Zuul:

 Automated mapping of routes via service discovery

Zuul can automatically route requests based on their service IDs(service IDs are loaded from Service registry) with zero configuration. If you don’t specify any routes, Zuul will automatically use the Eureka service ID of the service being called and map it.

map the service instance id & client request URL (exclude the first part of URL. first part of the URL is for resolving the service ID).

 Manual mapping of routes using service discovery

zuul:

ignored-services: 'organizationservice' -> this will remove the automated mapping from Service registry

prefix: /api -> all service will be prefixed with /api

routes:

organizationservice: /organization/\*\*

licensingservice: /licensing/\*\* -> (licensingservice = service ID, /api/licensing/ -> /licensingservice/\*\* )

 Manual mapping of routes using static URLs

manual mappinig used to route services that aren’t managed by Eureka.

zuul:

routes:

licensestatic:

path: /licensestatic/\*\*

url: http://licenseservice-static:8081

organizationstatic:

path: /licensestatic/\*\*

serviceId: organizationstatic -> Defines a service ID that will be used to look up the service in Ribbon

ribbon:

eureka:

enabled: false

organizationstatic: -> Configuration for service id

ribbon:

listOfServers: http://organizationsservice-static1:8081,http://organizationsservice-static2:8082

Zuul Gateway: Router and server-side load balancer

@SpringBootApplication

@EnableZuulProxy

public class ZuulServerApplication {

public static void main(String[] args) {

SpringApplication.run(ZuulServerApplication.class,args);

}

}

eureka:

instance:

preferIpAddress: true

client:

registerWithEureka: true

fetchRegistry: true

serviceUrl:

defaultZone: http://localhost:8761/eureka/

Zuul Filter :

Pre-filters—A pre-filter is invoked before the actual request to the target destination occurs with Zuul.

i.e auth/ token expiry.

Post filters—A post filter is invoked after the target service has been invoked and a response is being sent back to the client.

logging, metrics

Route filters—The route filter is used to intercept the call before the target service is invoked.

Usually a route filter is used to determine if some level of dynamic routing needs to take place.

dynamic routing based on the incoming request.

All type are implemented using ZuulFilter interface.

( method filterType, tell what the actual filter type is -POST\_TYPE/ROUTE\_TYPE/PRE\_TYPE)

public class TrackingFilter extends ZuulFilter{

@Override

public String filterType() {

return FilterUtils.PRE\_FILTER\_TYPE;

}

@Override

public int filterOrder() {

return someNumber;

}

@Override

public boolean shouldFilter() {

return SHOULD\_FILTER;

}

@Override

public Object run() {

}

}

EnableZuulProxy VS EnableZuulServer

@EnableZuulProxy is a superset of @EnableZuulServer

@EnableZuulProxy - contains all the filters installed by @EnableZuulServer. The additional filters in the “proxy” enable routing functionality

Provides integaration with Eureaka & Ribbon load balancing.

EnableZuulServer - enableZuulServer is used when you want to build your own routing service and not use any Zuul prebuilt capabilities.

load routing from application.yaml file. does not provide any integration with Eureka.

In zuul, we can specify a list of ignored headers as part of the route configuration.

zuul:

routes:

users:

path: /myusers/\*\*

sensitiveHeaders: Cookie,Set-Cookie,Authorization

url: https://downstreams