**Micro-services**

* Why micro-services

Here you 1st need to understand what is a Monolithic application

* Monolithic application
* All services, modules deployed in to one server, or use the common infra, e.g. app-servers, messaging middleware, dbase servers etc
* As you add new features, size of Monolithic application increases
* Code base becomes more complex
* Deployment and testing time-lines increase
* You cannot scale individual modules or feature, need to scale the whole module

In recent times, big 2-commerence companies like Netflix and Amazon, has addressed their scalability issues using micro-services approach

* Attributes of a micro-services app
* Product is divided into small business areas, that share nothing with each other
* Each business area is an independently testable n deployable piece of code
* Each module has its own infra, dbase, middle-ware, etc
* These micro-services can then be developed using different tech stacks, the one that best suits the need
* These micro-services can communicate with each-other via

1. Rest or thrift (sync way)
2. JMS, messaging (async way)

* Disadvantges of the micro-services architecture
* Need to define standard ways to development n deployment
* Need better and more communication between different teams
* Need more investment on infra and other components

The key distinction between a monolitch and micro-serivce is what gets deployed, you may have lot of modules(maven modules) i.e. independent projects while you develop, but what gets deployed is a single monolicth application, it scales as a mono

Soa vs micro-services

Soa is about building re-usable services, like doa layer, every time to need a database related operation re-use this service.

Micro-service via Spring Boot

* Each micro-service is a spring boot application,
* RestController: you need to specify via @RestController annotation and pass it the requestMapping url iva @RequestMapping(“/{endpoint}”), so this controller will get called on every hit of the end point and will be re-directed to the serving method accordingly
* Later you can propagate to the method using same @RequestMapping annotion
* Use the application.properties file to run services on different port, 8080 being the default one, vai ***server.port=8081***
* Path variable vs request param
* @Bean @Autowire

@bean is a producer, like you have created some object and if any one wants to re-use @bean tell the spring contianier to give it them

@autowire, is like consuming, e.g. consume a bean that some-one has declared or written earlier

* RestTemplate: its an easy way to make rest-api calls,

RestTemplate.getForObject(“<url>”,”returnType.class”,)

Rest-template is a synchronous way to interact between micro-services, to use asynch way, you can use WebClient.

* WebClient: is a reactive way for programing, reactive –web.
* Service-discovery/Eureka,
* Need to avoid hard-coding of urls, we need to discover where a particular service is running, can be done in 2 ways

1. Client –side, the discovery server, hand-shakes the client and the service. Spring-cloud/boot use client-side service discovery
2. Server-side, calling service interacts via this discovery server

* Create a eureka server-
* Its just another spring-boot application, with eureka server dependency
* @Enable ServiceDiscovery, this annotation tells spring-boot that’s its an eureka server
* By default eureka server will run on port 8761
* Add 2 properties to avoid eureka from registering itself

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

* Now for eureka server to discover the micro-service, each micr-service needs to add the eureka client/service-discovery client dependency in its classpath i.e maven entry
* Now if you re-start the eureka server, this micro-service will be shown, by its service-name defined in the application.properties

i.e. spring.application.name=”abc”

- optinally to make things more explicit, you can add the @Enable EurekaClient annotation to the micro-service main class

* Consuming the registered services, via eureka server
* @LoadBalance is the answer, typically we need ot annototat the restTemplate with this, and now you can pass the service-discovery name, instead of the hard-coded url
* i.e. http://<service-discovery-name>/endpoints/enpoints
* Fault Tolerance & Resilience via Hystrix

1. Circuit-breaker pattern:

* Detect the faulty micro-service
* Avoid sending request to this faulty service for a certain time
* Hystrix is an open source library by Netflix, it implements the circuit breaker pattern
* All we need to do is pass the parameters
* Enable @HystrixCommand annotation on the method that is suspectable to fail due to circuit breaking
* There are a lot of paramets and configs, need to read more
* Also you can spcify the fallback method, so this method get called in case a ciritu break happens
* Bulk head pattern: the basic idea is to separate the fulty service from effecting the good service
* To config a bulk-ahead, we use the @HyxtrixCommand annotation and use the threadPoolKey property (it sort of a thread-pool specific to this mthod), so once the thread-pool is exhausted , it will resort ot the fallback method
* Zuul server/ Api Gateway