**MicroServices**

**Monoliths**

Traditionally we are building large enterprise applications in modularised fashion (??!!??) but finally deploy them together as a single deployment unit (EAR or WAR). These are called Monolithic applications.

**Dis advantages of Monolith Services:**

Large codebases become mess over the time

It is not possible to scale up only certain parts of the application

Technology updates/rewrites become complex and expensive tasks

However, IMHO, it is relatively easy to deploy and monitor Monoliths compared to MicroServices.

**Advantages of MicroServices**

Comprehending smaller codebase is easy

Can independently scale up highly used services

Technology updates/rewrites become simpler

**MicroServices**

**Definition:** Small autonomous services that work together.

A MicroService is a service built around a specific business capability which can be independently deployed. So, to build large enterprise applications we can identify the sub-domains of our main business domain and build each sub-domain as a MicroService using Domain Driven Design (DDD) techniques. But in the end, we need to make all these MicroServices work together to serve the end user as if it is a single application.

**Advantages of MicroServices:**

* Smaller codebase is easy to maintain.
* Can independently scale up highly used services
* Each team can focus on one (or few) MicroService(s)
* Technology updates/rewrites become simpler

**Challenges with MicroServices:**

* Getting correct sub-domain boundaries, in the beginning, is hard
* Need more skilled developers to handle distributed application complexities
* Managing MicroServices based applications without proper DevOps culture is next to impossible
* Local developer environment setup might become complex to test cross-service communications. Though using Docker/Kubernetes this can be mitigated to some extent.

**Spring Cloud** is essentially an implementation of various design patterns to be followed while building Cloud Native applications. Instead of reinventing the wheel, we can simply take advantage of various Spring Cloud modules and focus on our main business problem than worrying about infrastructural concerns.

Following are just a few of Spring Cloud modules that can be used to address distributed application concerns:

**Spring Cloud Config Server:** To externalize configuration of applications in a central config server with the ability to update the configuration values without requiring to restart the applications. We can use Spring Cloud Config Server with git or Consul or ZooKeeper as config repository.

**Service Registry and Discovery:** As there could be many services and we need the ability to scale up or down dynamically, we need Service Registry and Discovery mechanism so that service-to-service communication should not depend on hard-coded hostnames and port numbers. Spring Cloud provides Netflix Eureka-based Service Registry and Discovery support with just minimal configuration. We can also use Consul or ZooKeeper for Service Registry and Discovery.

**Advantages:**

1. No need to hard URL and Port.
2. Dynamically increase and decrease the instances based on load.

When a service is registered with Eureka Server it keeps sending heartbeats for certain interval. If Eureka server didn’t receive heartbeat from any service instance it will assume service instance is down and take it out from the pool.

**Circuit Breaker:** In microservices based architecture, one service might depend on another service and if one service goes down then failures may cascade to other services as well. Spring Cloud provides Netflix Hystrix based Circuit Breaker to handle these kinds of issues.

**Spring Cloud Data Streams:** These days we may need to work with huge volumes of data streams using Kafka or Spark etc. Spring Cloud Data Streams provides higher-level abstractions to use those frameworks in an easier manner.

**Spring Cloud Security:** Some of the microservices needs to be accessible to authenticated users only and most likely we might want a Single Sign-On feature to propagate the authentication context across services. Spring Cloud Security provides authentication services using OAuth2.

**Distributed Tracing:** One of the pain-point with microservices is the ability to debug issues. One simple end-user action might trigger a chain of microservice calls, there should be a mechanism to trace the related call chains. We can use Spring Cloud Sleuth with Zipkin to trace the cross-service invocations. Using Zuul Filters for cross-cutting concerns.

As Zuul act as a proxy to all our microservices, we can use Zuul service to implement some cross-cutting concerns like security, rate limiting etc. One common use-case is forwarding the Authentication headers to all the downstream services.

Typically in microservices, we will use OAuth service for authentication and authorization. Once the client is authenticated OAuth service will generate a token which should be included in the requests making to other microservices so that client need not be authenticated for every service separately. We can use Zuul filter to implement features.

<https://sivalabs.in/2018/03/microservices-part-5-spring-cloud-zuul-proxy-as-api-gateway/>

**Spring Cloud Contract:** There is a high chance that separate teams work on different microservices. There should be a mechanism for teams to agree upon API endpoint contracts so that each team can develop their APIs independently. Spring Cloud Contract helps to create such contracts and validate them by both service provider and consumer.

Spring Boot already provides a lot of options to externalize configuration properties. However, once the application is started you can’t change those property values at runtime. You need to update the properties and restart the application to take those changes into effect.

In the microservices world, there could be a large number of microservices and multiple instances of those microservices are running. Updating configuration properties and restarting all those instances manually or even with automated scripts may not be feasible. Spring Cloud Config addresses this problem.

We can create a Spring Cloud Config Server which provides the configuration values for all of our microservices. We can use GIT, SVN, database or Consul as a backend to store the configuration parameters. Then we can configure the location of Spring Cloud Config server in our microservice so that it will load all the properties when we start the application. In addition to that, whenever we update the properties we can invoke /refresh REST endpoint in our microservice so that it will reload the configuration changes without requiring to restart the application.

# Developer Notes:

1. **Configuration Management:**

* Spring cloud config server (Keeps configuration in one place, that makes easy to maintain configuration for all microservices)

1. **Dynamic Scale Up and Scale down:**

* Naming Server (Eureka)
* Ribbon (Client Side Load Balancing by using Naming server)
* Feign (Easier REST Clients)

1. **Visibility and Monitoring:**

* Zipkin distributed tracing (to trace request across multiple components)
* Netflix API gateway.

Spring cloud sleuth will assign id to request across multiple components.

1. **Fault Tolerance:**

* Hystrix

**Zuul API Gateway:**

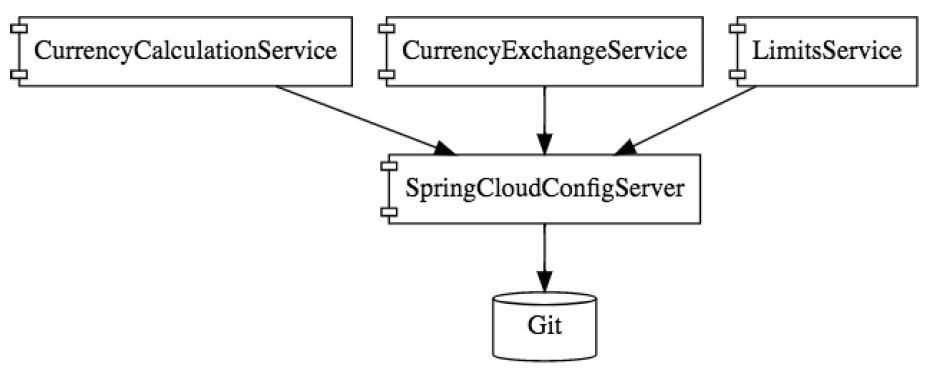
* Authentication, Authorization and Security
* Rate Limits
* Fault Toleration
* Service Aggregation

Feign is used as rest client

Ribbon is used as load balancing. (We are hard coding instance URL’s)

If you use both Feign and Ribbon then only load balancing works.

Eureka Naming Server is used to dynamically increase and decrease instances. (We no need to hard code URL’s)



**Development Notes:**

**limits-service:**

@SpringBootApplication

@EnableHystrix

**public** **class** LimitsServiceApplication {

}

**spring-cloud-starter-config:** is used as spring cloud config server server client.

**spring-cloud-starter-bus-amqp:** is used to refresh the configuration. Refresh URL is

http://localhost:2018/actuator/bus-refresh (POST method)

**spring-cloud-starter-netflix-hystrix:** is used to implement fault tolerance.

@HystrixCommand(fallbackMethod = "fallbackRetrieveConfiguration")

@EnableHystrix (at Application level)

**spring-boot-starter-actuator:** is used to monitor the application.

**spring-data-rest-hal-browser:** is UI for actuator.

**spring-boot-devtools:** Automatically pics up the changes.

* To get configuration from spring-cloud-config-server project to limits-service we need to rename application.properties to bootstrap.properties then add

spring.cloud.config.uri=http://localhost:8888/

spring.profiles.active=dev

management.endpoints.web.exposure.include=\* (This requires for bus refresh)

@ConfigurationProperties("limit-service") (Is used to read the properties files with prefix limit-service)

**spring-cloud-config-server:**

@SpringBootApplication

@EnableConfigServer

**public** **class** SpringCloudConfigServerApplication {

}

**spring-cloud-config-server:** Centralized server to serve configuration files/information. Clients connect over HTTP and retrieves their configuration. This project can run as load balancer to server all the configuration.

@EnableConfigServer (at Application level)

To connect spring-cloud-config-server to GIT repo add below property in application.properties.

spring.cloud.config.server.git.uri=file:\\E:\\GitRepo

If the active profile is DEV and key is not available in DEV properties file, then value will be picked up from default properties file.

Configuration file naming convention:

<spring.application.name>-<profile>.properties or .yml

* spring.application.name: is client app name
* profile: is client spring.profiles.active property.

Obtaining settings from config server:

* http://<server>:<port>/<spring.application.name>/<profile> (spring clients do this automatically on startup.)
* First it looks for .yml file then .properties file.

**Example file name:**

limits-service.properties

limits-service-dev.properties

limits-service-qa.properties

**eureka-naming-server:**

@SpringBootApplication

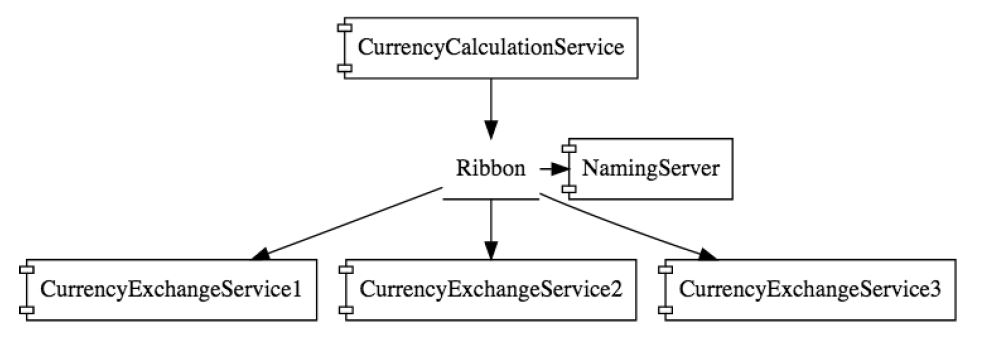
@EnableEurekaServer

**public** **class** EurekaNamingServerApplication {

}

**spring-cloud-starter-netflix-eureka-server:** is used for define eureka server.

@EnableEurekaServer (at Application level)



**currency-exchange-service:**

@SpringBootApplication

@EnableDiscoveryClient

**public** **class** CurrencyExchangeServiceApplication {

}

**spring-cloud-starter-netflix-eureka-client:** to register this project with eureka server.

@EnableDiscoveryClient (at Application level) we can user @EnableEurekaClient but best approach is to use Discovery because later Eureka server might change). Eureka client sends IP, Port and other details to eureka server, so eureka server register with it and keeps on checking health of the client, if it not getting response from client it remove the instance from eureka server.

eureka.client.service-url.default-zone=http://localhost:8761/eureka (in properties file)

**spring-cloud-starter-sleuth:** is used to assign unique id to same request.

@Bean

**public** Sampler defaultSampler() {

**return** Sampler.***ALWAYS\_SAMPLE***;

}// To trace all the requests.

**spring-cloud-sleuth-zipkin:** zipkin server.

**spring-cloud-starter-bus-amqp:** rabbit MQ.

**currency-conversion-service:**

@SpringBootApplication

@EnableFeignClients("com.spring.microservices")

@EnableDiscoveryClient

**public** **class** CurrencyConversionServiceApplication {

}

**spring-cloud-starter-netflix-eureka-client:** to register this project with eureka server.

@EnableDiscoveryClient (at Application level)

eureka.client.service-url.default-zone=http://localhost:8761/eureka (in properties file)

**spring-cloud-starter-sleuth:** is used to assign unique id to same request.

@Bean

**public** Sampler defaultSampler() {

**return** Sampler.***ALWAYS\_SAMPLE***;

}// To trace all the requests.

**spring-cloud-sleuth-zipkin:** zipkin server.

**spring-cloud-starter-bus-amqp:** rabbit MQ.

**spring-cloud-starter-openfeign:** is used to write client code to invoke service feign make it easy.

@EnableFeignClients("com.spring.microservices") (at Application lever)

@FeignClient("zuul-api-gateway-server") (we used zuul api app name here so request go through the zuul api gateway)

**public** **interface** CurrencyExchangeServiceProxy {

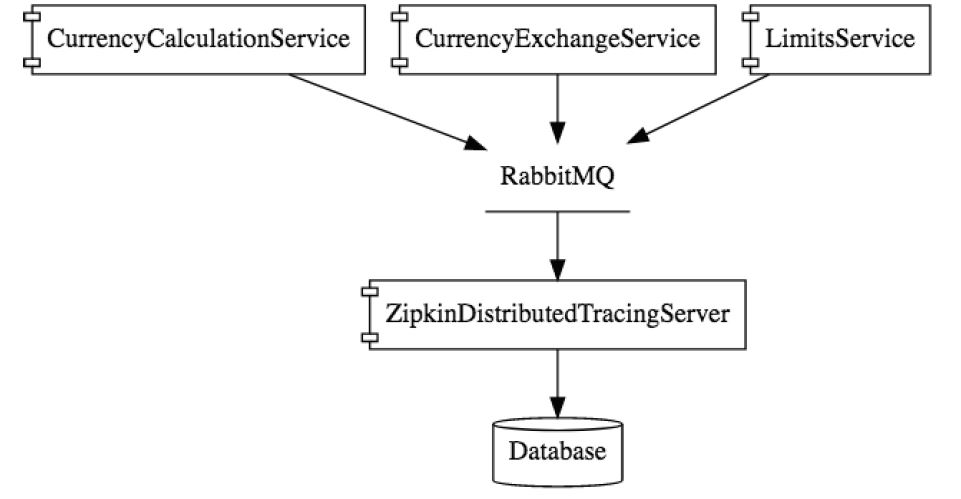
}

**spring-cloud-starter-netflix-ribbon:** it enables client side load balancing.

@RibbonClient("currency-exchange-service")

**public** **interface** CurrencyExchangeServiceProxy {

}



**zuul-api-gateway-server:**

@SpringBootApplication

@EnableZuulProxy

@EnableDiscoveryClient

**public** **class** ZuulApiGatewayServerApplication {

}

**spring-cloud-starter-netflix-zuul:** zuul api gateway server.

@EnableZuulProxy (at Application lever)

**spring-cloud-starter-netflix-eureka-client:** to register this project with eureka server.

@EnableDiscoveryClient (at Application level)

eureka.client.service-url.default-zone=http://localhost:8761/eureka (in properties file)

**spring-cloud-starter-sleuth:** is used to assign unique id to same request.

@Bean

**public** Sampler defaultSampler() {

**return** Sampler.***ALWAYS\_SAMPLE***;

}// To trace all the requests.

**spring-cloud-sleuth-zipkin:** zipkin server.

**spring-cloud-starter-bus-amqp:** rabbit MQ.

@Component

**public** **class** ZuulLoggingFilter **extends** ZuulFilter {

}