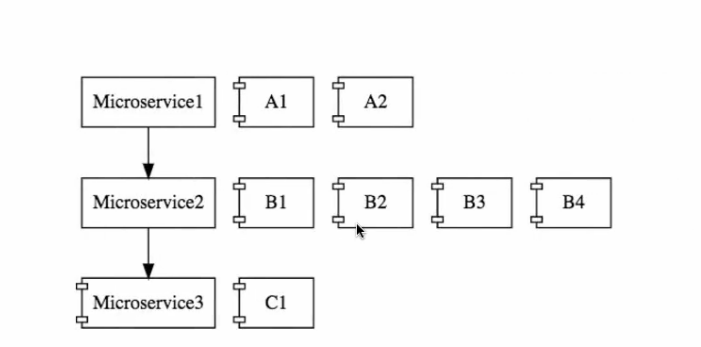
**Micro Services**

Develop a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms (HTTP resource API).

Independently deployable by automated deployment.

* REST
* Small deployable units which are cloud enabled.



Multiple instances running in cloud environment i.e M1 has two instance, M2 has 4 instance

**Challenges in Micro Service  
  
1. Bounded Context –** Identify the boundary of micro service. It is an evolutionary approach.

**2. Configuration Management -** There are multiple instance and environments and it becomes difficult to manage

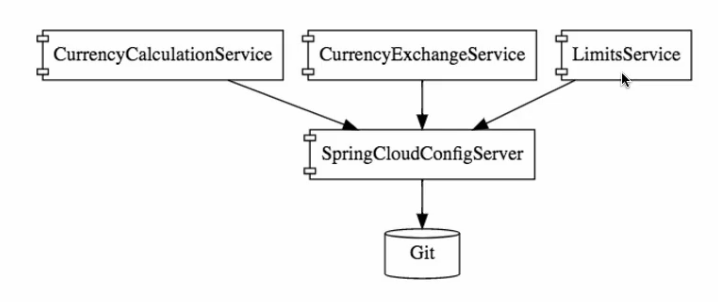
**3. Dynamic Scale up and Scale down –** Ability to bring new instance and distribute the load among new instance too

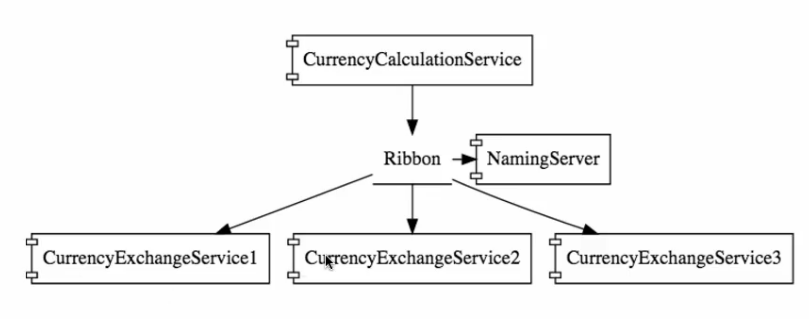
**4. Visibility –** to identify bugs, running services.

**5.** **Pack of cards**- should have fault tolerance in micro services.

**Spring Cloud**

* **Spring cloud config server** = all the config of different env of different micro services and spring-cloud-config-server helps in keeping configs at one place.



* **Dynamic Scale up and Scale down -**

**Naming server** = Eureka (Service registration)

**Ribbon** = Client side load balancing (Service discovery)  
CurrencyCalculationService uses ribbon in client side for load balancing.  
**Feign** = Easier RESTFUL Clients

* **Visibility and tracking**

**Zipkin Distributed tracing –** to track id across multiple components.

**Netflix API Gateway**

* **Fault tolerance using Hysterix**

**Advantages of Microservices:** New process and technology can be accommodated, Dynamic Scaling (load balancing), faster release cycles.

**Steps to create a micro service and deploy on cloud**

**Services Exposed:**

* Limits-service
* Currency-exchange-service
* Currency-calculator-service

1. **Limits-service:** maintains the config to have maximum and minimum value  
     
   **Dependencies**: spring-web, DevTools, Actuator, Spring-cloud-config-client.  
   Assign spring.application.name and server.port in applications.properties.

**Read the values from application.properties using spring-boot**

#limits-service.minimum = 1

#limits-service.maximum = 1000

* Create a new class Configuration.java and make as @component to make spring manage it.
* Add @ConfigurationProperties(“limits-service”) to configuration.java and make a getters and setters for the same.

1. **Setting up Spring-cloud-config-server  
   Dependencies:** spring-cloud-config-server, DevTools

* Assign spring.application.name and server.port in applications.properties.
* Create a local git repository using mkdir and do a **git init** to make it as git repo.
* Add external folder as source link using link source and create a new text file named **“limits-service”** which stores configuration of limits-service micro service.
* Git add – a
* Git commit –m “first commit”

1. **Connecting spring-cloud-config –server to local git**

* Find out the folder path of local git repo.
* Make change in spring-cloud-config-server -> application.properties  
  spring.cloud.config.server.git.uri=file://C:/Codebase/git\_repo/git-local-config-repo

url: localhost:8088/limits-service/default  
@EnableConfigServer to be added in main class.

1. **Configuring for multiple Environments**

* Create limits-service.properties for each environment.
* Now we can access the config based on environment using below url

url: localhost:8088/limits-service/dev

1. **Connect limit-service to spring-cloud-config-server**

* Pick the config from spring-cloud-config-server.
* Rename the properties file to bootstrap.properties.
* Specify the url of the spring cloud config server in bootstrap.properties  
  spring.cloud.config.uri = http://localhost:8088
* Specify the environment if you want to point to specific env else default properties will be used. This can also be configured in vm arguments or path variable.  
  spring.profiles.active=uat

1. **CurrencyExchangeService –** to talk to the database to retrieve exchange values for different currency combination  
     
   Variables in URL: **two currency values** (use @PathVariable) **Dependencies:** web, DevTools, actuators, spring-cloud-config-client  
   **Port**: 8000

* Assign spring.application.name and server.port in applications.properties.
* Create a simple controller with uri and path variable
* Create a pojo class to get values and set values
* To read the values from database, make the POJO class as **@Entity**
* Create a data.sql file to insert value to currencyExchangeValue.
* To read the port from environment use,

@Autowired

**private** Environment environment;  
  
environment.getProperty("local.server.port")

* To make two instance of currencyExchangeService running, set it in vm arguments.  
  Run configuration -> arguments -> -Dserver.port=8001
* Enable h2 console

spring.jpa.show-sql=true

spring.h2.console.enabled=true

* Add @Id, @Column(name=””) annotation.

1. **CurrencyControllerService** – to fetch value from currencyExchangeService and calculate the amount.  
   Variables in URL: **two currency values, quantity** (use @PathVariable) **Dependencies:** web, DevTools, actuators, spring-cloud-config-client  
   **Port**: 8100

* Assign spring.application.name and server.port in applications.properties.
* Create a simple controller with uri and path variable
* Create a pojo class to get values and set values
* Use **RestTemplate** to get the values from the currencyExchangeService service.

ResponseEntity<CurrencyConversionBean> responseEntity = **new** RestTemplate().

getForEntity("http://localhost:8001/currency-exchange/from/{from}/to/{to}", CurrencyConversionBean.**class**, uriVariables );

* Get the values from the currencyExchangeService and add it to currencyConversionServiceBean.

1. **Avoiding use of RestTemplate and use Feign to overcome the problem**

* **Dependencies:** openfeign, Netflix-ribbion
* Enable the feign configuration in the main class @EnableFeignClients("com.spring.currencyconversionservice")
* Create a proxy class interface to get the values from currencyExchangeService  
  @FeignClient(name="currency-exchange-service" , url="localhost:8000")

@GetMapping("/currency-exchange/from/{from}/to/{to}")

**public** CurrencyConversionBean returnCurrencyValue(@PathVariable String from, @PathVariable String to ) ;

* We can use Ribbon for load balancing.

@RibbonClient(name="currency-exchange-service")  
Add the below config in the application.properties  
currency-exchange-service.ribbon.listOfServers = http://localhost:8000, http://localhost:8001, http://localhost:8002, <http://localhost:8003>

1. **Naming server** – to dynamically increase or decrease the service based on load.

* No hardcoding of the currency-exchange-service in the application.properties in currency-calculation-service.
* **Service registry -** All instance of micro services will be registered in Eureka naming server.
* **Service discovery** – asking for instance of currency exchange service.

1. **Create a component for naming server**

* **Dependencies:** Netflix-eureka-server
* Add the configs in application.properties

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

* Add the annotation @EnableEurekaServer
* In the client side limits-service, currency-exchange-service, currency-conversion-service add the annotation @EnableDiscoveryClient
* Add the below property to refer to eureka server

eureka.client.service-url.default-zone = <http://localhost:8761/eurkea>

1. **API Gateways: Useful for logging calls, debug and analytics  
   all calls between a micro services should pass through this gateway.**

* Authentication, authorization and security
* Rate limits
* Fault tolerance- Default responses when services are down.
* Service Aggregation – external consumer wants to aggregate 15 service and give as one service.
* Dependencies: **netfilx-zuul, eureka-client**
* Setup zullu component
* What should it do when it intercepts the request
* All requests are configured in the gateway.
* Create a class ZuulLoggingFilter extends ZuulFilter and add unimplemented methods

Now, every request should flow to zuul api gateway, to do this we need to modify our CurrencyExchangeServiceProxy.

* Make the @FeignClient refer to zuulAPIGatewayServer instead of currencyExchangeService.
* @FeignClient(name="netflix-zuul-api-gateway-server"
* Modify @GetMapping("currency-exchange-service/….

Url: <http://localhost:8002/currency-exchange/from/EUR/to/INR/>  
Modified URL: <http://localhost:8765/currency-exchange-service/currency-exchange/from/USD/to/INR>

URL: <http://localhost:8101/currency-exchange-feign/from/USD/to/INR/quantity/1000>

Modified URL: <http://localhost:8765/currency-conversion-service/currency-exchange-feign/from/USD/to/INR/quantity/1000>

1. **Spring Cloud sleuth: Foe enabling tracing id while logging.**

* **Add the dependency** spring-starter-sleuth
* Also, add a bean in the main class to have the tracing id for every request.

@Bean

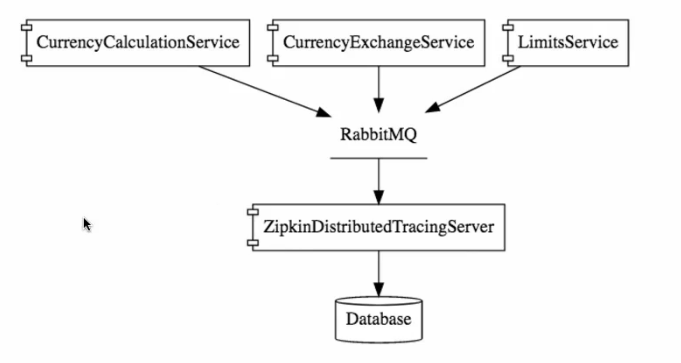
**public** Sampler defaultSampler() {

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

}

1. **Distributed tracing in zipkin using centralized log**

* ELK stack, Elastic search, Kibana are few centralized logging.
* **ZipkinDistributedTracingServer**



* All the micro services will be putting the logging into the Rabbit MQ and the ZDTS will be listening it and pick it from queue.
* ZDTS is connected to a database.   
    
  To download Zipkin server:

<https://search.maven.org/remote_content?g=io.zipkin.java&a=zipkin-server&v=LATEST&c=exec>  
  
Start the Rabbit MQ service first and run the below commands.  
Set RABBIT\_URI=amqp://localhost  
java –jar zipkin-server-2.12.9-exec.jar

URL: <http://localhost:9411/zipkin/>

1. **To connect Rabbit MQ with Micro services**

Dependencies: spring-cloud-sleuth-zipkin, spring-cloud-starter-bus-amqp  
  
Start all the service and monitor the health of the service in zipkin server.

1. **Need of Spring Cloud bus**

If we add or update in config and you want the config to reflect, then we manually do a post request

<http://localhost:8080/application/referesh>  
  
To avoid refreshing the url every time once we make config change, we can use spring cloud bus such as Kafka, Rabbit MQ.

Dependency: amqp   
  
<http://localhost:8088/actuator/bus-refresh>

All instances registered in bus. When we need any config change to be done spring cloud bus will propagate to all the instance to update the config.

1. Fault tolerance in Micro service.

If any of the service is down , all other micro services goes down with it. To avoid this behaviour, we can implement fault tolerance behaviour i.e default behaviour.

Dependency: spring-cloud-starter-hystrix

* Add the annotation in main class @EnableHystrix
* To all the controller you can add a fallback method @HystrixCommand(fallbackMethod = )