# Spring Cloud Netflix

# Spring Cloud Netflix provides Netflix OSS integrations for Spring Boot apps through auto configuration and binding to the Spring Environment and other Spring programming model idioms.

## ****Eureka****

Eureka is a REST (Representational State Transfer) based service that is primarily used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers. We call this service, the Eureka Server. Eureka also comes with a Java-based client component, the Eureka Client, which makes interactions with the service much easier. The client also has a built-in load balancer that does basic round-robin load balancing.

Multiple peer of Eureka Servers:

eureka:

client:

serviceUrl:

defaultZone: http://peer1/eureka/,http://peer2/eureka/,http://peer3/eureka/

---

spring:

profiles: peer1

eureka:

instance:

hostname: peer1

---

spring:

profiles: peer2

eureka:

instance:

hostname: peer2

---

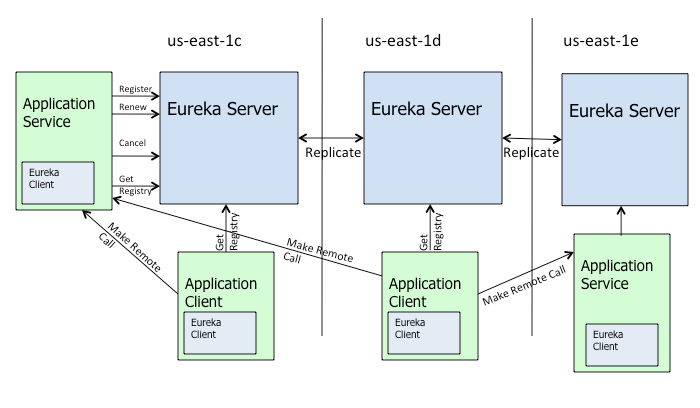
spring:

profiles: peer3

eureka:

instance:

hostname: peer3

Client-side service discovery allows services to find and communicate with each other without hard coding hostname and port. The only ‘fixed point’ in such an architecture consists of a service registry with which each service has to register. A drawback is that all clients must implement a certain logic to interact with this fixed point. This assumes an additional network round trip before the actual request.

# With Netflix Eureka each client can simultaneously act as a server, to replicate its status to a connected peer. In other words, a client retrieves a list of all connected peers of a service registry and makes all further requests to any other services through a load-balancing algorithm. To be informed about the presence of a client, they have to send a heartbeat signal to the registry.

To achieve the goal of this write-up, we will implement three micro services:

* a service registry (Eureka Server),
* a REST service which registers itself at the registry (Eureka Client) and
* a web-application, which is consuming the REST service as a registry-aware client (Spring Cloud Netflix Feign Client).

## ****Eureka Server****

To implement a *Eureka Server* for using as service registry is as easy as: adding [spring-cloud-starter-netflix-eureka-server](https://search.maven.org/search?q=spring-cloud-starter-netflix-eureka-server) to the dependencies, enable the Eureka Server in a [*@SpringBootApplication*](https://www.baeldung.com/spring-boot-application-configuration) per annotate it with *@EnableEurekaServer* and configure some properties.

Configuration

server:

  port: 8761

eureka:

  client:

    registerWithEureka: false

    fetchRegistry: false

We are telling the built-in Eureka Client not to register with ‘itself’, because our application should be acting as a server.

## ****Eureka Client****

For a @SpringBootApplication to be discovery-aware, we have to include some Spring Discovery Client (for example [spring-cloud-starter-netflix-eureka-client](https://search.maven.org/search?q=spring-cloud-starter-netflix-eureka-client)) into our classpath.

Then we need to annotate a @Configuration with either @EnableDiscoveryClient or @EnableEurekaClient – note that this annotation is optional if we have the spring-cloud-starter-netflix-eureka-client dependency on the classpath.

The latter tells Spring Boot to use Spring Netflix Eureka for service discovery explicitly. To fill our client application with some sample-life, we’ll also include the [spring-boot-starter-web](https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.boot%22%20AND%20a%3A%22spring-boot-starter-web%22) package in the pom.xml and implement a REST controller.

Configuration:

spring:

  application:

    name: spring-cloud-eureka-client

server:

  port: 0

eureka:

  client:

    serviceUrl:

      defaultZone: ${EUREKA\_URI:<http://localhost:8761/eureka>}

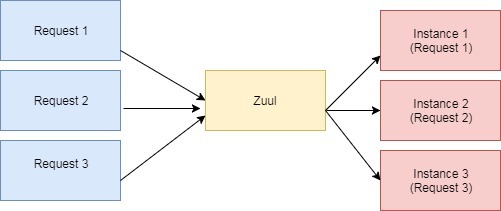
  instance:

    preferIpAddress: true

## Load Balancing with Zuul

**When Zuul receives a request, it picks up one of the physical locations available and forwards requests to the actual service instance.** The whole process of caching the location of the service instances and forwarding the request to the actual location is provided out of the box with no additional configurations needed.

Here, we can see how Zuul is encapsulating three different instances of the same service:



Internally, Zuul uses Netflix Ribbon to look up for all instances of the service from the service discovery (Eureka Server).

# Zuul Proxy:

Zuul Server is a gateway application that handles all the requests and does the dynamic routing of microservice applications. The Zuul Server is also known as Edge Server.

<dependency>

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

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

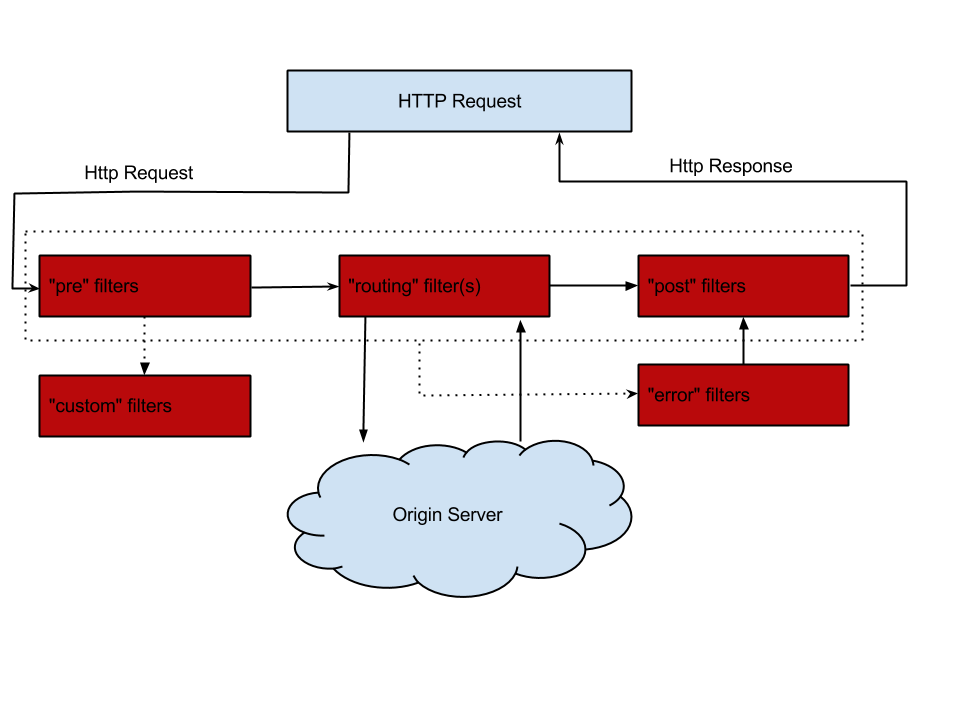
</dependency>

Application.properties

zuul.routes.products.path = /api/demo/\*\*

zuul.routes.products.url = http://localhost:8080/

The @EnableZuulProxy annotation is used to make your Spring Boot application act as a Zuul Proxy server.



**Netflix** [**Ribbon**](https://github.com/Netflix/ribbon)

Netflix [Ribbon](https://github.com/Netflix/ribbon) is an Inter Process Communication (IPC) cloud library. Ribbon primarily provides client-side load balancing algorithms.

Apart from the client-side load balancing algorithms, Ribbon provides also other features:

* **Service Discovery Integration** – Ribbon load balancers provide service discovery in dynamic environments like a cloud. Integration with Eureka and Netflix service discovery component is included in the ribbon library
* **Fault Tolerance** – the Ribbon API can dynamically determine whether the servers are up and running in a live environment and can detect those servers that are down
* **Configurable load-balancing rules** – Ribbon supports RoundRobinRule, AvailabilityFilteringRule, WeightedResponseTimeRule out of the box and also supports defining custom rules

Ribbon API works based on the concept called “Named Client”. While configuring Ribbon in our application configuration file we provide a name for the list of servers included for the load balancing.

Ribbon API enables us to configure the following components of the load balancer:

* Rule – Logic component which specifies the load balancing rule we are using in our application
* Ping – A Component which specifies the mechanism we use to determine the server’s availability in real-time
* ServerList – can be dynamic or static. In our case, we are using a static list of servers and hence we are defining them in the application configuration file directly

Notice how we used the *WeightedResponseTimeRule* rule to determine the server and *PingUrl* mechanism to determine the server’s availability in real-time. According to this rule, each server is given a weight according to its average response time, lesser the response time gives lesser the weight. This rule randomly selects a server where the possibility is determined by server’s weight. And the *PingUrl* will ping every URL to determine the server’s availability.

## ****Feign Client****

To finalize our project with three dependent microservices, we will now implement a REST-consuming web application using Spring Netflix Feign Client.

Think of Feign as **discovery-aware** [**Spring RestTemplate**](https://www.baeldung.com/rest-template) **using interfaces to communicate with endpoints**. This interfaces will be automatically implemented at runtime and instead of service-urls, it is using service-names.

Without Feign we would have to autowire an instance of EurekaClient into our controller with which we could receive a service-information by service-name as an Application object.

We would use this Application to get a list of all instances of this service, pick a suitable one and use this InstanceInfo to get hostname and port. With this, we could do a standard request with any http client.

The Feign Client is located in the [*spring-cloud-starter-feign*](https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%22spring-cloud-starter-feign%22) package. To enable it, we have to annotate a @Configuration with @EnableFeignClients. To use it, we simply annotate an interface with @FeignClient(“service-name”) and auto-wire it into a controller.

@FeignClient("spring-cloud-eureka-client")

public interface GreetingClient {

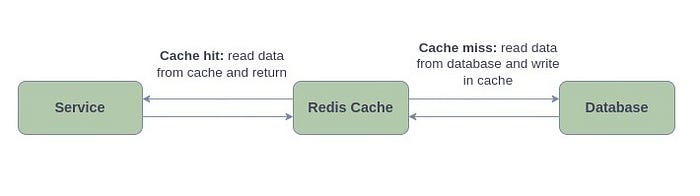
    @RequestMapping("/greeting")

    String greeting();

}

Remote Dictionary Server, aka Redis

* Add the @EnableCaching annotation to the configuration file.
* Add the required cache library to the classpath.
* Add the cache provider configuration file to the root classpath.



<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-redis</artifactId>  
 <version>3.1.2</version>  
</dependency>

Redis-related libraries are part of the Spring Data Package, which provides easy access to databases, relational and non-relational databases, map-reduce frameworks, and cloud-based data services. The spring-boot-starter-data-redis will have all the necessary dependencies prepared.

spring:  
 cache:  
 type: redis  
 host: localhost  
 port: 6379  
 redis:  
 time-to-live: 60000

1. **@Cacheable** is employed to fetch data from the database, storing it in the cache. Upon future invocations, the method retrieves the cached value directly, eliminating the need to execute the method again.

@GetMapping("/product/{id}")   
@Cacheable(value = "product", key = "#id")  
public Product getProductById(@PathVariable long id) {...}

The value attribute establishes a cache with a specific name, while the key attribute permits the use of Spring Expression Language to compute the key dynamically. Consequently, the method result is stored in the ‘product’ cache, where respective ‘product\_id’ serves as the unique key. This approach optimizes caching by associating each result with a distinct key.

1. **@CachePut** is used to update data in the cache when there is any update in the source database.

@PutMapping("/product/{id}")  
@CachePut(cacheNames = "product", key = "#id")  
public Product editProduct(@PathVariable long id, @RequestBody Product product) {...}

The cacheNames attribute is an alias for value, and can be used in a similar manner.

**3. @CacheEvict** is used for removing stale or unused data from the cache.

@DeleteMapping("/product/{id}")  
@CacheEvict(cacheNames = "product", key = "#id", beforeInvocation = true)  
public String removeProductById(@PathVariable long id) {...}

**Kafka:**

<dependency>  
 <groupId>org.springframework.kafka</groupId>  
 <artifactId>spring-kafka</artifactId>  
 </dependency>  
 <dependency>

**spring.kafka.bootstrap-servers=localhost:9092**

**spring.kafka.consumer.group-id=my-group-id**

@Component  
public class MessageProducer {  
  
 @Autowired  
 private KafkaTemplate<String, String> kafkaTemplate;  
  
 public void sendMessage(String topic, String message) {  
 kafkaTemplate.send(topic, message);  
 }  
  
}

<dependency>  
 <groupId>com.amazonaws</groupId>  
 <artifactId>aws-java-sdk-sqs</artifactId>  
 <version>1.12.632</version>  
</dependency>

aws.access.key=${AWS\_ACCESS\_KEY:your\_access\_key}  
aws.secret.key=${AWS\_SECRET\_KEY:your\_secret\_key}  
aws.queueName=spring-aws-queue

@Configuration  
public class SqsConfig {  
  
 @Value("${aws.access.key}")  
 private String accessKey;  
  
 @Value("${aws.secret.key}")  
 private String secretKey;  
  
 @Bean  
 public AmazonSQS amazonSQSClient() {  
 BasicAWSCredentials awsCredentials = new BasicAWSCredentials(accessKey, secretKey);  
 return AmazonSQSClientBuilder.standard()  
 .withCredentials(new AWSStaticCredentialsProvider(awsCredentials))  
 .withRegion(Regions.EU\_CENTRAL\_1)  
 .build();  
 }  
  
}