### Spring Cloud

**Spring Cloud** provides tools for developers to quickly build some of the common pattern in distributed system.

While building distributed system, there are a few problems that are encountered. They are configuration management, Service Discovery, Circuit Breakers, and distributed sessions.

Spring cloud is a collection of tools that provides solutions to commonly encountered problem.

**Service Registration and discovery**

When we start project, we usually have all the configuration in the properties file.

As more and more services are developed and deployed, adding and modifying these properties become more complex. Some services might go down, while some of the location might change. This manual changing of properties may create issues.

Eureka service registration and discovery helps in such scenarios.

As all services are registered to the Eureka Server and lookup done by calling the Eureka server, any change in service locations need not be handled and is taken care of.

Eureka is alternatively known as the Netflix Service Discovery.

**Include Eureka in your project**

* To include the Eureka Client in your project, use spring-cloud-starter-eureka-client dependency and @EnableEurekaClient annotation on the Spring Boot Application class.

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

* To include the Eureka server in your project, use spring-cloud-starter-eureka-server dependency and @EnableEurekaServer annotation on the Spring Boot Application class.

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

**Load balancing**

Load balancing improves the distribution of work loads across multiple computing resources.

Load balancing aims to optimize resource use, maximise throughput, minimise the response time and avoid overload of any single resource.

Load balancing usually involves dedicated software or hardware , such as a multilayer switch or a Domain Name System server process.

For example, we have two services such as Account Service, Customer Service. The load at Customer service is high.

To deal with this, we deploy multiple instances of customer service. Now we will have to use a load balancer to route any incoming requests to either of these two services.

**Client-Side Load Balancing**

Netflix Ribbon performing load balancing. It will call either of the customer service instance.

**spring-cloud-starter-ribbon**

@Autowired

private LoadBalancerClient loadBalancer;

public void getEmployee() throws RestClientException, IOException {

ServiceInstance serviceInstance=loadBalancer.choose("employee-producer");

System.out.println(serviceInstance.getUri());

String baseUrl=serviceInstance.getUri().toString();

baseUrl=baseUrl+"/employee";

}

**Circuit Breaker design pattern**

The Circuit Breaker design pattern is used to stop the process of request and response if a service is not working.

**Problem**:

A service generally calls other services to retrieve data, there is always the possibility that the other service is unavailable or may be down.

There are two problem with this.

1. First, the request will keep going to the down service, exhausting network resources, and slowing performance.

2. Second, the user experience will be bad and unpredictable.

**Solution**:

The consumer should invoke a remote service via a proxy that behaves in a similar fashion to an electrical circuit breaker.

When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period, all attempts to invoke the remote service will fail immediately.

After the timeout expires the circuit breaker allows a limited number of test requests to pass through.

If those requests succeed the circuit breaker resumes normal operation. Otherwise, if there is a failure the timeout period begins again.

Netflix Hystrix is a good implementation of the circuit breaker pattern. It also helps you to define a fallback mechanism which can be used when the circuit breaker trips. That provides a better user experience.

**Hystrix**

Hystrix is an error tolerance and latency library developed by Netflix that implements Circuit Breaker Pattern.

The main purpose of Hystrix is isolates the access points between services, stop cascading failures across them and provides the fall-back options.

In a microservice architecture, it is common to have multiple layers of service calls, i.e one microservice can call multiple downstream microservices. A service failure in any one of the lower level services can cause cascading failure all the way up to the user.

Circuit Breaker pattern provides a fallback mechanism, which avoids cascading of failures up to the user.

Fallback executes any alternative flow if it exists in case remote service 's call is failing.

Circuit Breakers stops client to call remote service if remote service is repeatedly failing.

### include Hystrix in your project

<dependency>

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

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

</dependency>

To include Hystrix in your project, use **spring-cloud-starter-netflix-hystrix** dependency and @EnableCircuitBreaker annotation on the Spring Boot Application class.

Use the @HystrixCommand annotation on the method for which fallback method has to be applied.

@Service

**public** **class** StudentServiceDelegate {

@Autowired

RestTemplate restTemplate;

@HystrixCommand(fallbackMethod = "callStudentServiceAndGetData\_Fallback")

**public** String callStudentServiceAndGetData(String schoolname) {

System.***out***.println("Getting School details for " + schoolname);

String response = restTemplate

.exchange("http://localhost:8300/getSchoolDetails/{schoolName}"

, HttpMethod.***GET***

, **null**

, **new** ParameterizedTypeReference<String>() {

}, schoolname).getBody();

System.***out***.println("Response Received as " + response + " - " + **new** Date());

**return** "NORMAL FLOW !!! - School Name - " + schoolname + " ::: " +

" Student Details " + response + " - " + **new** Date();

}

**private** String callStudentServiceAndGetData\_Fallback(String schoolname) {

System.***out***.println("Student Service is down!!! fallback route enabled...");

**return** "CIRCUIT BREAKER ENABLED!!! No Response From Student Service at this moment. " +" Service will be back shortly - " + **new** Date();

}

@Bean

**public** RestTemplate restTemplate() {

**return** **new** RestTemplate();

}

}

@HystrixCommand(fallbackMethod = "fallback\_hello", commandProperties = {

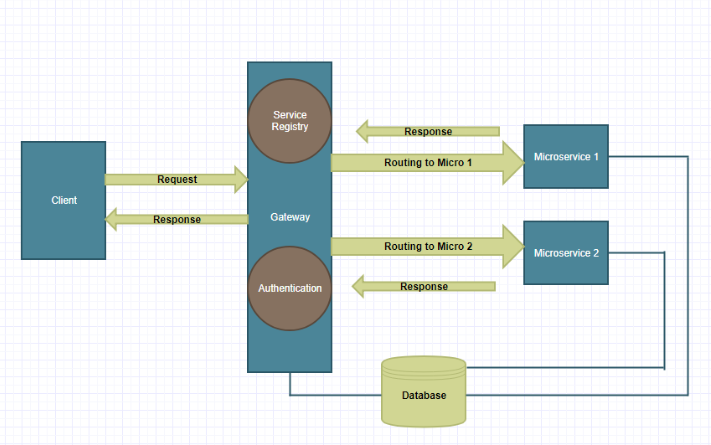
@HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "1000")

})

execution.isolation.semaphore.maxConcurrentRequests – Default value – 10

**API gate way**

* The [API Gateway pattern](https://microservices.io/patterns/apigateway.html) defines how clients access the services in a microservice architecture.
* API gateway is single entry point for all clients.
* The API gateway handles requests in one of two ways.
  + Some requests are simply proxied/routed to the appropriate service.
  + It handles other requests by fanning out (spread) to multiple services.
* The API gateway might also implement security, e.g. verify that the client is authorized to perform the request.
* The API Gateway is a server. It is a single-entry point into a system. API Gateway encapsulates the internal system architecture. It provides an API that is tailored to each client. It also has other responsibilities such as **authentication, monitoring, load balancing, caching, request shaping and management,**and **static response handling**.
* All the requests made by the client go through the API Gateway. After that, the API Gateway routes requests to the appropriate microservice.
* The API Gateway can provide each client with a custom API. It also translates between two protocols, such as **HTTP,** **WebSocket,** and **Web-Unfriendly** protocols that are used internally.



## **Working of API Gateway**

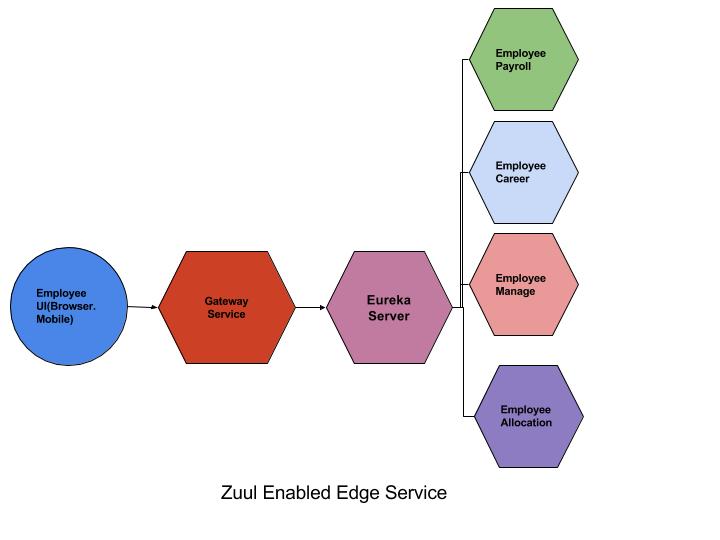
In microservices, we route all the requests through an API. We can implement common features like **authentication, routing, caching, versioning, auditing,**and **logging** in the API Gateway.

## **Advantages of API Gateway**

* + The most important advantage of API Gateway is that it encapsulates the internal structure of the application.
  + Rather than invoking the specific service, the client directly talks to the API Gateway.
  + It reduces the number of round trips between client and application.
  + It simplifies the client code.
  + It reduces coding efforts, makes the application more efficient, decreases errors all at the same time.
  + It provides each kind of client with a specific API.

## **Disadvantages**

* + It requires routing rules.
  + There is a possibility of a single point of failure.
  + Risk of complexity due to all the API rules are in one place.



* API Gateway based on [Netflix Zuul](https://github.com/Netflix/zuul) which will perform the task of filtering, routing etc.
* Service discovery server which will maintain the records for each microservice present in the system. This server will be based on [Spring Eureka](https://projects.spring.io/spring-cloud/)
* Finally, a microservice which can be accessed via the API Gateway

Spring Cloud provides Zuul proxy that can be used to create API gateway.

Zuul server is a gateway application that handles all requests and does dynamic routing of microservices application.

To include Zuul in your project, use the spring-cloud-starter-netflix-zuul dependency.

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

zuul.ignored-services=\*

zuul.routes.gallery-service.path=/gallery/\*\*

zuul.routes.gallery-service.service-id=gallery-service

**Server-side load balancing**

The server-side load balancing can be done by using Netflix Zuul. It is also known as a JVM based router.

**Different kinds of filters provided by Zuul**

Zuul provides the following filter types that correspond to the lifecycle of a request.

1. PRE Filters - Filters that execute before routing to the origin server.

2.ROUTING Filters - Filters that handle routing the request to an origin. Builds HTTP Request and calls the Origin server using Apache HttpClient or Netflix Ribbon.

3. POST Filters - Filters that execute after the request has been routed to the origin.

4. ERROR Filters - Filters that execute when an error occurs during any one of the phases.

<https://www.javatpoint.com/introduction-to-api-gateways>

<https://medium.com/@arjunac009/spring-boot-microservice-with-centralized-authentication-zuul-eureka-jwt-5719e05fde29>