**Zuul and Eureka:**

A common challenge when building microservices is providing a unified interface to the consumers of your system. You can integrate **Zuul** with other Netflix projects like Hystrix for fault tolerance and **Eureka** for service discovery, or use it to manage routing rules, filters, and load balancing across your system.

**Spring** Boot - **Zuul** Proxy Server and Routing. Advertisements. **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.

**Zuul** acts as an **API gateway** or Edge service. ... So, we have to create a brand new microservice which is **Zuul**-enabled, and this service sits on top of all other microservices. It acts as an Edge service or client-facing service. Its Service **API**should be exposed to the client/UI.

**Zuul** uses a range of different types of filters that enables us to quickly and nimbly apply functionality to our edge service. These filters help us **perform** the following functions: Authentication and Security - identifying authentication requirements for each resource and rejecting requests that **do** not satisfy them.

**Eureka** Server is an application that holds the information about all client-service applications. Every Micro service will register into the **Eureka** server and **Eureka** server knows all the client applications running on each port and IP address. **Eureka** Server is also known as Discovery Server.

**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.

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.

Spring Cloud Netflix provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms. With a few simple annotations you can quickly enable and configure the common patterns inside your application and build large distributed systems with battle-tested Netflix components. The patterns provided include Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon).

**Getting Started with Eureka**

As long as Spring Cloud Netflix and Eureka Core are on the classpath any Spring Boot application with @EnableEurekaClient will try to contact a Eureka server on [http://localhost:8761](http://localhost:8761/) (the default value of eureka.client.serviceUrl.defaultZone):

@SpringBootApplication

@EnableEurekaClient

@RestController

public class Application {

@RequestMapping("/")

public String home() {

return "Hello World";

}

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

To run your own server use the spring-cloud-starter-netflix-eureka-server dependency and @EnableEurekaServer.

**Getting Started with Zuul**

To include Zuul in your project, use the starter with a group ID of org.springframework.cloud and a artifact ID of spring-cloud-starter-netflix-zuul.

**Embedded Zuul Reverse Proxy**

Spring Cloud has created an embedded Zuul proxy to ease the development of a common use case where a UI application wants to make proxy calls to one or more back end services. This feature is useful for a user interface to proxy to the back end services it requires, avoiding the need to manage CORS and authentication concerns independently for all the back ends.

To enable it, annotate a Spring Boot main class with @EnableZuulProxy. Doing so causes local calls to be forwarded to the appropriate service. By convention, a service with an ID of users receives requests from the proxy located at /users (with the prefix stripped). The proxy uses Ribbon to locate an instance to which to forward through discovery. All requests are executed in a [hystrix command](https://cloud.spring.io/spring-cloud-netflix/multi/multi__router_and_filter_zuul.html), so failures appear in Hystrix metrics. Once the circuit is open, the proxy does not try to contact the service.

the Zuul starter does not include a discovery client, so, for routes based on service IDs, you need to provide one of those on the classpath as well (Eureka is one choice).

To skip having a service automatically added, set zuul.ignored-services to a list of service ID patterns. If a service matches a pattern that is ignored but is also included in the explicitly configured routes map, it is unignored, as shown in the following example:

**application.yml.**

zuul:

ignoredServices: '\*'

routes:

users: /myusers/\*\*

In the preceding example, all services are ignored, **except** for users.

To augment or change the proxy routes, you can add external configuration, as follows:

**application.yml.**

zuul:

routes:

users: /myusers/\*\*

The preceding example means that HTTP calls to /myusers get forwarded to the users service (for example /myusers/101 is forwarded to /101).

To get more fine-grained control over a route, you can specify the path and the serviceId independently, as follows:

**application.yml.**

zuul:

routes:

users:

path: /myusers/\*\*

serviceId: users\_service

The preceding example means that HTTP calls to /myusers get forwarded to the users\_service service. The route must have a path that can be specified as an ant-style pattern, so /myusers/\* only matches one level, but /myusers/\*\* matches hierarchically.

The location of the back end can be specified as either a serviceId (for a service from discovery) or a url (for a physical location), as shown in the following example:

**application.yml.**

zuul:

routes:

users:

path: /myusers/\*\*

url: https://example.com/users\_service

These simple url-routes do not get executed as a HystrixCommand, nor do they load-balance multiple URLs with Ribbon. To achieve those goals, you can specify a serviceId with a static list of servers, as follows:

**application.yml.**

zuul:

routes:

echo:

path: /myusers/\*\*

serviceId: myusers-service

stripPrefix: **true**

hystrix:

command:

myusers-service:

execution:

isolation:

thread:

timeoutInMilliseconds: ...

myusers-service:

ribbon:

NIWSServerListClassName: com.netflix.loadbalancer.ConfigurationBasedServerList

listOfServers: https://example1.com,http://example2.com

ConnectTimeout: 1000

ReadTimeout: 3000

MaxTotalHttpConnections: 500

MaxConnectionsPerHost: 100

Another method is specifiying a service-route and configuring a Ribbon client for the serviceId (doing so requires disabling Eureka support in Ribbon — see [above for more information](https://cloud.spring.io/spring-cloud-netflix/multi/multi_spring-cloud-ribbon.html#spring-cloud-ribbon-without-eureka)), as shown in the following example:

**application.yml.**

zuul:

routes:

users:

path: /myusers/\*\*

serviceId: users

ribbon:

eureka:

enabled: **false**

users:

ribbon:

listOfServers: example.com,google.com

You can provide a convention between serviceId and routes by using regexmapper. It uses regular-expression named groups to extract variables from serviceIdand inject them into a route pattern, as shown in the following example:

**ApplicationConfiguration.java.**

*@Bean*

**public** PatternServiceRouteMapper serviceRouteMapper() {

**return** **new** PatternServiceRouteMapper(

"(?<name>^.+)-(?<version>v.+$)",

"${version}/${name}");

}

The preceding example means that a serviceId of myusers-v1 is mapped to route /v1/myusers/\*\*. Any regular expression is accepted, but all named groups must be present in both servicePattern and routePattern. If servicePattern does not match a serviceId, the default behavior is used. In the preceding example, a serviceId of myusers is mapped to the "/myusers/\*\*" route (with no version detected). This feature is disabled by default and only applies to discovered services.

To add a prefix to all mappings, set zuul.prefix to a value, such as /api. By default, the proxy prefix is stripped from the request before the request is forwarded by (you can switch this behavior off with zuul.stripPrefix=false). You can also switch off the stripping of the service-specific prefix from individual routes, as shown in the following example:

**application.yml.**

zuul:

routes:

users:

path: /myusers/\*\*

stripPrefix: **false**

zuul.stripPrefix only applies to the prefix set in zuul.prefix. It does not have any effect on prefixes defined within a given route’s path.

In the preceding example, requests to /myusers/101 are forwarded to /myusers/101 on the users service.

The zuul.routes entries actually bind to an object of type ZuulProperties. If you look at the properties of that object, you can see that it also has a retryable flag. Set that flag to true to have the Ribbon client automatically retry failed requests. You can also set that flag to true when you need to modify the parameters of the retry operations that use the Ribbon client configuration.

By default, the X-Forwarded-Host header is added to the forwarded requests. To turn it off, set zuul.addProxyHeaders = false. By default, the prefix path is stripped, and the request to the back end picks up a X-Forwarded-Prefix header (/myusers in the examples shown earlier).

If you set a default route (/), an application with @EnableZuulProxy could act as a standalone server. For example, zuul.route.home: / would route all traffic ("/\*\*") to the "home" service.

If more fine-grained ignoring is needed, you can specify specific patterns to ignore. These patterns are evaluated at the start of the route location process, which means prefixes should be included in the pattern to warrant a match. Ignored patterns span all services and supersede any other route specification. The following example shows how to create ignored patterns:

**application.yml.**

zuul:

ignoredPatterns: /\*\*/admin/\*\*

routes:

users: /myusers/\*\*

The preceding example means that all calls (such as /myusers/101) are forwarded to /101 on the users service. However, calls including /admin/ do not resolve.

If you need your routes to have their order preserved, you need to use a YAML file, as the ordering is lost when using a properties file. The following example shows such a YAML file:

**application.yml.**

zuul:

routes:

users:

path: /myusers/\*\*

legacy:

path: /\*\*

If you were to use a properties file, the legacy path might end up in front of the users path, rendering the users path unreachable.

**Zuul Http Client**

The default HTTP client used by Zuul is now backed by the Apache HTTP Client instead of the deprecated Ribbon RestClient. To use RestClient or okhttp3.OkHttpClient, set ribbon.restclient.enabled=true or ribbon.okhttp.enabled=true, respectively. If you would like to customize the Apache HTTP client or the OK HTTP client, provide a bean of type ClosableHttpClient or OkHttpClient.

**Reference Guide:**

<https://spring.io/projects/spring-cloud>