Spring Cloud Netflix

Service Discovery: Eureka Server

By default, every Eureka server is also a Eureka client and requires (at least one) service URL to locate a peer. If you do not provide it, the service runs and works, but it fills your logs with a lot of noise about not being able to register with the peer.

Standalone Mode

The combination of the two caches (client and server) and the heartbeats make a standalone Eureka server fairly resilient to failure, as long as there is some sort of monitor or elastic runtime (such as Cloud Foundry) keeping it alive. In standalone mode, you might prefer to switch off the client side behavior so that it does not keep trying and failing to reach its peers. The following example shows how to switch off the client-side behavior:

application.yml (Standalone Eureka Server).

```
server:
  port: 8761

eureka:
  instance:
    hostname: localhost
  client:
    registerWithEureka: false
    fetchRegistry: false
    serviceUrl:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
```

Notice that the serviceUrl is pointing to the same host as the local instance.

Peer Awareness

Eureka can be made even more resilient and available by running multiple instances and asking them to register with each other. In fact, this is the default behavior, so all you need to do to make it work is add a valid serviceUrl to a peer, as shown in the following example:

application.yml (Two Peer Aware Eureka Servers).

```
---
spring:
  profiles: peer1
eureka:
  instance:
    hostname: peer1
  client:
    serviceUrl:
      defaultZone: http://peer2/eureka/

---
```

```
spring:
  profiles: peer2
eureka:
  instance:
    hostname: peer2
  client:
    serviceUrl:
      defaultZone: http://peer1/eureka/
```

In the preceding example, we have a YAML file that can be used to run the same server on two hosts (peer1 and peer2) by running it in different Spring profiles. You could use this configuration to test the peer awareness on a single host (there is not much value in doing that in production) by manipulating /etc/hosts to resolve the host names. In fact, the eureka.instance.hostname is not needed if you are running on a machine that knows its own hostname (by default, it is looked up by using java.net.InetAddress).

You can add multiple peers to a system, and, as long as they are all connected to each other by at least one edge, they synchronize the registrations amongst themselves. If the peers are physically separated (inside a data center or between multiple data centers), then the system can, in principle, survive "split-brain" type failures. You can add multiple peers to a system, and as long as they are all directly connected to each other, they will synchronize the registrations amongst themselves.

application.yml (Three Peer Aware 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
```

When to Prefer IP Address
In some cases, it is preferable for Eureka to advertise the IP addresses of services rather than the hostname. Set eureka.instance.preferIpAddress to true and, when the application registers with eureka, it uses its IP address rather than its hostname.

If the hostname cannot be determined by Java, then the IP address is sent to Eureka. Only explict way of setting the hostname is by setting eureka.instance.hostname property. You can set your hostname at the run-time by using an environment variable — for example, **eureka.instance.hostname=${HOST_NAME}.**

Securing The Eureka Server
You can secure your Eureka server simply by adding Spring Security to your server's classpath via **spring-boot-starter-security. By default when Spring Security is on the classpath it will require that a valid CSRF token be sent with every request to the app**. **Eureka clients will not generally possess a valid cross site request forgery (CSRF) token you will need to disable this requirement for the /eureka/** endpoints**. For example:

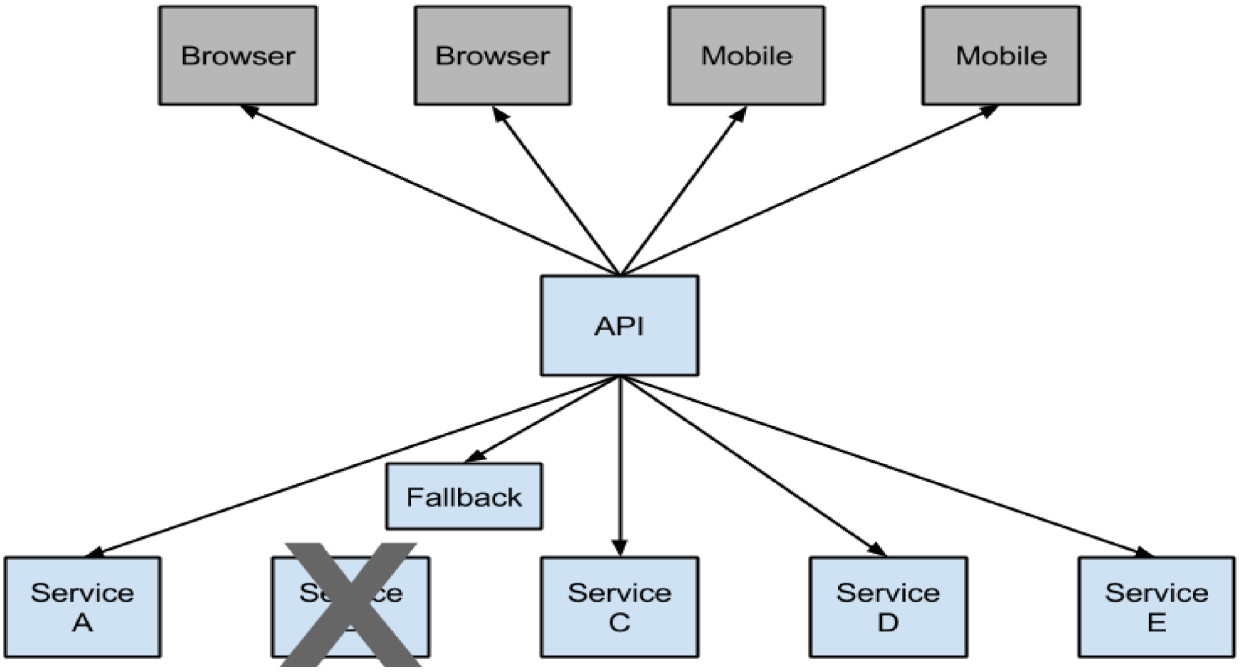
```
@EnableWebSecurity
class WebSecurityConfig extends WebSecurityConfigurerAdapter {

   @Override
   protected void configure(HttpSecurity http) throws Exception {
      http.csrf().ignoringAntMatchers("/eureka/**");
      super.configure(http);
   }
}
```

Circuit Breaker: Hystrix Clients
Netflix has created a library called Hystrix that implements the circuit breaker pattern. In a microservice architecture, it is common to have multiple layers of service calls.

A service failure in the lower level of services can cause cascading failure all the way up to the user. When calls to a particular service exceed **circuitBreaker.requestVolumeThreshold (default: 20 requests)** and the failure percentage is greater than **circuitBreaker.errorThresholdPercentage (default: >50%)** in a rolling window defined by **metrics.rollingStats.timeInMilliseconds (default: 10 seconds)**, the circuit opens and the call is not made. In cases of error and an open circuit, a fallback can be provided by the developer.

Hystrix fallback prevents cascading failures
Having an open circuit stops cascading failures and allows overwhelmed or failing services time to recover. The fallback can be another Hystrix protected call, static data, or a sensible empty value. Fallbacks may be chained so that the first fallback makes some other business call, which in turn falls back to static data.

Figure 3.2. Hystrix fallback prevents cascading failures



How to Include Hystrix
To include Hystrix in your project, use the starter with a **group ID** of **org.springframework.cloud** and a **artifact ID** of **spring-cloud-starter-netflix-hystrix**.

The following example shows a minimal Eureka server with a Hystrix circuit breaker:

```
@SpringBootApplication
@EnableCircuitBreaker
public class Application {

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }
}

@Component
public class StoreIntegration {

    @HystrixCommand(fallbackMethod = "defaultStores")
    public Object getStores(Map<String, Object> parameters) {
        //do stuff that might fail
    }

    public Object defaultStores(Map<String, Object> parameters) {
        return /* something useful */;
    }
}
```

The **@HystrixCommand** is provided by a Netflix contrib library called **"javanica".** Spring Cloud automatically wraps Spring beans with that annotation in a proxy that is connected to the Hystrix circuit breaker. The circuit breaker calculates when to open and close the circuit and what to do in case of a failure.

To configure the **@HystrixCommand** you can use the commandProperties attribute with a list of **@HystrixProperty** annotations

Propagating the Security Context or Using Spring Scopes
If you want some thread local context to propagate into a @HystrixCommand, the default declaration does not work, because it executes the command in a thread pool (in case of timeouts). You can switch Hystrix to use the same thread as the caller through configuration or directly in the annotation, by asking it to use a different "Isolation Strategy". The following example demonstrates setting the thread in the annotation:

```
@HystrixCommand(fallbackMethod = "stubMyService",
   commandProperties = {
     @HystrixProperty(name="execution.isolation.strategy", value="SEMAPHORE")
   }
)
...
```

The same thing applies if you are using @SessionScope or @RequestScope. If you encounter a runtime exception that says it cannot find the scoped context, you need to use the same thread.

You also have the option to set the **hystrix.shareSecurityContext property to true**. Doing so auto-configures a Hystrix concurrency strategy plugin hook to transfer the SecurityContext from your main thread to the one used by the Hystrix command. Hystrix does not let multiple Hystrix concurrency strategy be registered so an extension mechanism is available by declaring your own HystrixConcurrencyStrategy as a Spring bean. Spring Cloud looks for your implementation within the Spring context and wrap it inside its own plugin.

Circuit Breaker: Hystrix Dashboard
One of the main benefits of Hystrix is the set of metrics it gathers about each HystrixCommand. The Hystrix Dashboard displays the health of each circuit breaker in an efficient manner.

Router and Filter: Zuul

Routing is an integral part of a microservice architecture. For example, / may be mapped to your web application, /api/users is mapped to the user service and /api/shop is mapped to the shop service. Zuul is a JVM-based router and server-side load balancer from Netflix.

Netflix uses Zuul for the following:

- Authentication
- Insights
- Stress Testing
- Canary Testing
- Dynamic Routing
- Service Migration
- Load Shedding
- Security
- Static Response handling
- Active/Active traffic management

Zuul's rule engine lets rules and filters be written in essentially any JVM language, with built-in support for Java and Groovy.

Note

The configuration property **zuul.max.host.connections has been replaced by two new properties, zuul.host.maxTotalConnections and zuul.host.maxPerRouteConnections,** which default to **200** and **20** respectively.

The default Hystrix isolation pattern **(ExecutionIsolationStrategy)** for all routes is **SEMAPHORE**. **zuul.ribbonIsolationStrategy can be changed to THREAD** if that isolation pattern is preferred.

How to Include 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, so failures appear in Hystrix metrics. Once the circuit is open, the proxy does not try to contact the service.

Note

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: http://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: http://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), 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 serviceId and inject them into a route pattern, as shown in the following example:

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: http://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: http://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), 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 serviceId and 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
```

Note
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.

[Warning]
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.

Cookies and Sensitive Headers
You can share headers between services in the same system, but you probably do not want sensitive headers leaking downstream into external servers. You can specify a list of ignored headers as part of the route configuration. Cookies play a special role, because they have well defined semantics in browsers, and they are always to be treated as sensitive. If the consumer of your proxy is a browser, then cookies for downstream services also cause problems for the user, because they all get jumbled up together (all downstream services look like they come from the same place).

If you are careful with the design of your services, (for example, if only one of the downstream services sets cookies), you might be able to let them flow from the back end all the way up to the caller. Also, if your proxy sets cookies and all your back-end services are part of the same system, it can be natural to simply share them (and, for instance, use Spring Session to link them up to some shared state). Other than that, any cookies that get set by downstream services are likely to be not useful to the caller, so it is recommended that you make (at least) Set-Cookie and Cookie into sensitive headers for routes that are not part of your domain. Even for routes that are part of your domain, try to think carefully about what it means before letting cookies flow between them and the proxy.

The sensitive headers can be configured as a comma-separated list per route, as shown in the following

example:

application.yml.

```
zuul:
 routes:
   users:
     path: /myusers/**
     sensitiveHeaders: Cookie,Set-Cookie,Authorization
     url: https://downstream
```

This is the default value for sensitiveHeaders, so you need not set it unless you want it to be different. This is new in Spring Cloud Netflix 1.1 (in 1.0, the user had no control over headers, and all cookies flowed in both directions).

The sensitiveHeaders are a blacklist, and the default is not empty. Consequently, to make Zuul send all headers (except the ignored ones), you must explicitly set it to the empty list. Doing so is necessary if you want to pass cookie or authorization headers to your back end. The following example shows how to use sensitiveHeaders:

application.yml.

```
zuul:
 routes:
   users:
     path: /myusers/**
     sensitiveHeaders:
     url: https://downstream
```

You can also set sensitive headers, by setting zuul.sensitiveHeaders. If sensitiveHeaders is set on a route, it overrides the global sensitiveHeaders setting.

Ignored Headers

In addition to the route-sensitive headers, you can set a global value called zuul.ignoredHeaders for values (both request and response) that should be discarded during interactions with downstream services. By default, if Spring Security is not on the classpath, these are empty. Otherwise, they are initialized to a set of well known "security" headers (for example, involving caching) as specified by Spring Security. The assumption in this case is that the downstream services might add these headers, too, but we want the values from the proxy. To not discard these well known security headers when Spring Security is on the classpath, you can set zuul.ignoreSecurityHeaders to false. Doing so can be useful if you disabled the HTTP Security response headers in Spring Security and want the values provided by downstream services.

Management Endpoints

By default, if you use @EnableZuulProxy with the Spring Boot Actuator, you enable two additional endpoints:

- **Routes**
- **Filters**

Routes Endpoint

A GET to the routes endpoint at /routes returns a list of the mapped routes:

```
GET /routes.
{
  /stores/**: "http://localhost:8081"
}
```

Additional route details can be requested by adding the **?format=details** query string to **/routes**. Doing so produces the following output:

```
GET /routes/details.
{
  "/stores/**": {
    "id": "stores",
    "fullPath": "/stores/**",
    "location": "http://localhost:8081",
    "path": "/**",
    "prefix": "/stores",
    "retryable": false,
    "customSensitiveHeaders": false,
    "prefixStripped": true
  }
}
```

A POST to **/routes** forces a refresh of the existing routes (for example, when there have been changes in the service catalog). You can disable this endpoint by setting **endpoints.routes.enabled to false.**

Note
the routes should respond automatically to changes in the service catalog, but the POST to **/routes** is a way to force the change to happen immediately.

Filters Endpoint
A GET to the filters endpoint at /filters returns a map of Zuul filters by type. For each filter type in the map, you get a list of all the filters of that type, along with their details.

Strangulation Patterns and Local Forwards
A common pattern when migrating an existing application or API is to **"strangle" old endpoints,** slowly replacing them with different implementations. **The Zuul proxy is a useful tool for this because you can use it to handle all traffic from the clients of the old endpoints but redirect some of the requests to new ones.**

The following example shows the configuration details for a "strangle" scenario:

```
application.yml.
 zuul:
  routes:
    first:
      path: /first/**
      url: http://first.example.com
    second:
      path: /second/**
```

```
    url: forward:/second
  third:
    path: /third/**
    url: forward:/3rd
  legacy:
    path: /**
    url: http://legacy.example.com
```

In the preceding example, **we are strangle the "legacy" application**, which is mapped to all requests that do not match one of the other patterns. Paths in **/first/**** have been extracted into a new service with an external URL. Paths in **/second/**** are forwarded so that they can be handled locally (for example, with a normal Spring @RequestMapping). Paths in **/third/**** are also forwarded but with a different prefix (/third/foo is forwarded to /3rd/foo).

Note
The ignored patterns aren't completely ignored, they just are not handled by the proxy (so they are also effectively forwarded locally)

Uploading Files through Zuul
If you use @EnableZuulProxy, you can use the proxy paths to upload files and it should work, so long as the files are small. For large files there is an alternative path that bypasses the Spring DispatcherServlet (to avoid multipart processing) in "/zuul/*". In other words, if you have zuul.routes.customers=/customers/**, then you can POST large files to /zuul/customers/*. The servlet path is externalized via zuul.servletPath. If the proxy route takes you through a Ribbon load balancer, extremely large files also require elevated timeout settings, as shown in the following example:

application.yml.

```
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 60000
ribbon:
  ConnectTimeout: 3000
  ReadTimeout: 60000
```

Note that, for streaming to work with large files, you need to use chunked encoding in the request (which some browsers do not do by default), as shown in the following example:

```
$ curl -v -H "Transfer-Encoding: chunked" \
    -F "file=@mylarge.iso" localhost:9999/zuul/simple/file
```

Query String Encoding
When processing the incoming request, query params are decoded so that they can be available for possible modifications in Zuul filters. They are then re-encoded the back end request is rebuilt in the route filters. The result can be different than the original input if (for example) it was encoded with Javascript's encodeURIComponent() method. While this causes no issues in most cases, some web servers can be picky with the encoding of complex query string.

To force the original encoding of the query string, it is possible to pass a special flag to ZuulProperties so that the query string is taken as is with the HttpServletRequest::getQueryString method, as shown in the following example:

application.yml.
```
zuul:
  forceOriginalQueryStringEncoding: true
```

Note
This special flag works only with SimpleHostRoutingFilter. Also, you loose the ability to easily override query parameters with **RequestContext.getCurrentContext().setRequestQueryParams(someOverriddenParameters),** because the query string is now fetched directly on the original HttpServletRequest.

Request URI Encoding
When processing the incoming request, request URI is decoded before matching them to routes. The request URI is then re-encoded when the back end request is rebuilt in the route filters. This can cause some unexpected behavior if your URI includes the encoded "/" character.

To use the original request URI, it is possible to pass a special flag to 'ZuulProperties' so that the URI will be taken as is with the HttpServletRequest::getRequestURI method, as shown in the following example:

application.yml.
```
zuul:
  decodeUrl: false
```

Note
If you are overriding request URI using requestURI RequestContext attribute and this flag is set to false, then the URL set in the request context will not be encoded. It will be your responsibility to make sure the URL is already encoded.

Plain Embedded Zuul
If you use **@EnableZuulServer (instead of @EnableZuulProxy),** you can also run a Zuul server without proxying or selectively switch on parts of the proxying platform. Any beans that you add to the application of type ZuulFilter are installed automatically (as they are with @EnableZuulProxy) but without any of the proxy filters being added automatically.

In that case, the routes into the Zuul server are still specified by configuring "zuul.routes.*", but there is no service discovery and no proxying. Consequently, the "serviceId" and "url" settings are ignored. The following example maps all paths in "/api/**" to the Zuul filter chain:

application.yml.
```
zuul:
  routes:
    api: /api/**
```

Disable Zuul Filters
Zuul for Spring Cloud comes with a number of ZuulFilter beans enabled by default in both proxy and server mode. If you want to disable one, set **zuul.<SimpleClassName>.<filterType>.disable=true.** By convention, the package after filters is the Zuul filter type. For example to disable **org.springframework.cloud.netflix.zuul.filters.post.SendResponseFilter,** set

zuul.SendResponseFilter.post.disable=true.

Providing Hystrix Fallbacks For Routes

When a circuit for a given route in Zuul is tripped, you can provide a fallback response by creating a bean of type FallbackProvider. Within this bean, you need to specify the route ID the fallback is for and provide a ClientHttpResponse to return as a fallback. The following example shows a relatively simple FallbackProvider implementation:

```
class MyFallbackProvider implements FallbackProvider {

   @Override
   public String getRoute() {
      return "customers";
   }

   @Override
   public ClientHttpResponse fallbackResponse(String route, final Throwable cause) {
      if (cause instanceof HystrixTimeoutException) {
         return response(HttpStatus.GATEWAY_TIMEOUT);
      } else {
         return response(HttpStatus.INTERNAL_SERVER_ERROR);
      }
   }

   private ClientHttpResponse response(final HttpStatus status) {
      return new ClientHttpResponse() {
         @Override
         public HttpStatus getStatusCode() throws IOException {
            return status;
         }

         @Override
         public int getRawStatusCode() throws IOException {
            return status.value();
         }

         @Override
         public String getStatusText() throws IOException {
            return status.getReasonPhrase();
         }

         @Override
         public void close() {
         }

         @Override
         public InputStream getBody() throws IOException {
            return new ByteArrayInputStream("fallback".getBytes());
         }
```

```
            @Override
            public HttpHeaders getHeaders() {
                HttpHeaders headers = new HttpHeaders();
                headers.setContentType(MediaType.APPLICATION_JSON);
                return headers;
            }
        };
    }
}
```

The following example shows how the route configuration for the previous example might appear:

```
zuul:
  routes:
    customers: /customers/**
```

If you would like to provide a default fallback for all routes, you can create a bean of type FallbackProvider and have the getRoute method return * or null, as shown in the following example:

```
class MyFallbackProvider implements FallbackProvider {
    @Override
    public String getRoute() {
        return "*";
    }

    @Override
    public ClientHttpResponse fallbackResponse(String route, Throwable throwable) {
        return new ClientHttpResponse() {
            @Override
            public HttpStatus getStatusCode() throws IOException {
                return HttpStatus.OK;
            }

            @Override
            public int getRawStatusCode() throws IOException {
                return 200;
            }

            @Override
            public String getStatusText() throws IOException {
                return "OK";
            }

            @Override
            public void close() {

            }
```

```
        @Override
        public InputStream getBody() throws IOException {
            return new ByteArrayInputStream("fallback".getBytes());
        }

        @Override
        public HttpHeaders getHeaders() {
            HttpHeaders headers = new HttpHeaders();
            headers.setContentType(MediaType.APPLICATION_JSON);
            return headers;
        }
    };
  }
}
```

Zuul Timeouts

If you want to configure the socket timeouts and read timeouts for requests proxied through Zuul, you have two options, based on your configuration:

- If Zuul uses service discovery, you need to configure these timeouts with the ribbon.ReadTimeout and ribbon.SocketTimeout Ribbon properties.

If you have configured Zuul routes by specifying URLs, you need to use zuul.host.connect-timeout-millis and zuul.host.socket-timeout-millis.

Rewriting the Location header

If Zuul is fronting a web application, you may need to re-write the Location header when the web application redirects through a HTTP status code of 3XX. Otherwise, the browser redirects to the web application's URL instead of the Zuul URL. You can configure a LocationRewriteFilter Zuul filter to re-write the Location header to the Zuul's URL. It also adds back the stripped global and route-specific prefixes. The following example adds a filter by using a Spring Configuration file:

```
import org.springframework.cloud.netflix.zuul.filters.post.LocationRewriteFilter;
...

@Configuration
@EnableZuulProxy
public class ZuulConfig {
  @Bean
  public LocationRewriteFilter locationRewriteFilter() {
    return new LocationRewriteFilter();
  }
}
```

[Caution] Caution

Use this filter carefully. The filter acts on the Location header of ALL 3XX response codes, which may not be appropriate in all scenarios, such as when redirecting the user to an external URL.

Enabling Cross Origin Requests
By default Zuul routes all **Cross Origin requests (CORS)** to the services. If you want instead Zuul to handle these requests it can be done by providing custom WebMvcConfigurer bean:

```
@Bean
public WebMvcConfigurer corsConfigurer() {
   return new WebMvcConfigurer() {
      public void addCorsMappings(CorsRegistry registry) {
         registry.addMapping("/path-1/**")
               .allowedOrigins("http://allowed-origin.com")
               .allowedMethods("GET", "POST");
      }
   };
}
```

In the example above, we allow GET and POST methods from http://allowed-origin.com to send cross-origin requests to the endpoints starting with path-1. You can apply CORS configuration to a specific path pattern or globally for the whole application, using /** mapping. You can customize properties: allowedOrigins,allowedMethods,allowedHeaders,exposedHeaders,allowCredentials and maxAge via this configuration.

Zuul RequestContext
To pass information between filters, Zuul uses a RequestContext. Its data is held in a ThreadLocal specific to each request. Information about where to route requests, errors, and the actual HttpServletRequest and HttpServletResponse are stored there. The RequestContext extends ConcurrentHashMap, so anything can be stored in the context. FilterConstants contains the keys used by the filters installed by Spring Cloud Netflix

@EnableZuulProxy vs. @EnableZuulServer
Spring Cloud Netflix installs a number of filters, depending on which annotation was used to enable Zuul. **@EnableZuulProxy is a superset of @EnableZuulServer**. **In other words, @EnableZuulProxy contains all the filters installed by @EnableZuulServer. The additional filters in the "proxy" enable routing functionality. If you want a "blank" Zuul, you should use @EnableZuulServer.**

@EnableZuulServer Filters
@EnableZuulServer creates a **SimpleRouteLocator** that loads route definitions from Spring Boot configuration files.

The following filters are installed (as normal Spring Beans):

- **Pre filters:**
  - **ServletDetectionFilter:** Detects whether the request is through the Spring Dispatcher. Sets a boolean with a key of FilterConstants.IS_DISPATCHER_SERVLET_REQUEST_KEY.
  - **FormBodyWrapperFilter:** Parses form data and re-encodes it for downstream requests.
  - **DebugFilter:** If the debug request parameter is set, sets RequestContext.setDebugRouting() and RequestContext.setDebugRequest() to true. *Route filters:
  - **SendForwardFilter:** Forwards requests by using the Servlet RequestDispatcher. The forwarding location is stored in the RequestContext attribute,

FilterConstants.FORWARD_TO_KEY. This is useful for forwarding to endpoints in the current application.

- **Post filters:**
  - **SendResponseFilter:** Writes responses from proxied requests to the current response.

- **Error filters:**
  - **SendErrorFilter:** Forwards to /error (by default) if RequestContext.getThrowable() is not null. You can change the default forwarding path (/error) by setting the error.path property.

@EnableZuulProxy Filters
Creates a **DiscoveryClientRouteLocator** that loads route definitions from a DiscoveryClient (such as Eureka) as well as from properties. A route is created for each serviceId from the DiscoveryClient. As new services are added, the routes are refreshed.

In addition to the filters described earlier, the following filters are installed (as normal Spring Beans):

- **Pre filters:**
  - PreDecorationFilter: Determines where and how to route, depending on the supplied RouteLocator. It also sets various proxy-related headers for downstream requests.

- **Route filters:**
  - **RibbonRoutingFilter:** Uses Ribbon, Hystrix, and pluggable HTTP clients to send requests. Service IDs are found in the RequestContext attribute, FilterConstants.SERVICE_ID_KEY. This filter can use different HTTP clients:
    - **Apache HttpClient:** The default client.
    - **Squareup OkHttpClient v3:** Enabled by having the com.squareup.okhttp3:okhttp library on the classpath and setting ribbon.okhttp.enabled=true.
    - **Netflix Ribbon HTTP client:** Enabled by setting ribbon.restclient.enabled=true. This client has limitations, including that it does not support the PATCH method, but it also has built-in retry.
  - **SimpleHostRoutingFilter:** Sends requests to predetermined URLs through an Apache HttpClient. URLs are found in RequestContext.getRouteHost().

Custom Zuul Filter Examples

Most of the following "How to Write" examples below are included Sample Zuul Filters project. There are also examples of manipulating the request or response body in that repository.

How to Write a Pre Filter

Pre filters set up data in the RequestContext for use in filters downstream. The main use case is to set information required for route filters. The following example shows a Zuul pre filter:

```
public class QueryParamPreFilter extends ZuulFilter {
        @Override
        public int filterOrder() {
                return PRE_DECORATION_FILTER_ORDER - 1; // run before PreDecoration
        }

        @Override
        public String filterType() {
                return PRE_TYPE;
        }

        @Override
        public boolean shouldFilter() {
                RequestContext ctx = RequestContext.getCurrentContext();
                return !ctx.containsKey(FORWARD_TO_KEY) // a filter has already forwarded
                                && !ctx.containsKey(SERVICE_ID_KEY); // a filter has already
determined serviceId
        }
   @Override
   public Object run() {
      RequestContext ctx = RequestContext.getCurrentContext();
                HttpServletRequest request = ctx.getRequest();
                if (request.getParameter("sample") != null) {
                   // put the serviceId in `RequestContext`
                ctx.put(SERVICE_ID_KEY, request.getParameter("foo"));
        }
      return null;
   }
}
```

The preceding filter populates SERVICE_ID_KEY from the sample request parameter. In practice, you should not do that kind of direct mapping. Instead, the service ID should be looked up from the value of sample instead.

Now that SERVICE_ID_KEY is populated, PreDecorationFilter does not run and RibbonRoutingFilter runs.

Tip
If you want to route to a full URL, call ctx.setRouteHost(url) instead.

To modify the path to which routing filters forward, set the REQUEST_URI_KEY.

How to Write a Route Filter

Route filters run after pre filters and make requests to other services. Much of the work here is to translate request and response data to and from the model required by the client. The following example shows a Zuul route filter:

```
public class OkHttpRoutingFilter extends ZuulFilter {
	@Autowired
	private ProxyRequestHelper helper;

	@Override
	public String filterType() {
		return ROUTE_TYPE;
	}

	@Override
	public int filterOrder() {
		return SIMPLE_HOST_ROUTING_FILTER_ORDER - 1;
	}

	@Override
	public boolean shouldFilter() {
		return RequestContext.getCurrentContext().getRouteHost() != null
				&& RequestContext.getCurrentContext().sendZuulResponse();
	}

  @Override
  public Object run() {
		OkHttpClient httpClient = new OkHttpClient.Builder()
				// customize
				.build();

		RequestContext context = RequestContext.getCurrentContext();
		HttpServletRequest request = context.getRequest();

		String method = request.getMethod();

		String uri = this.helper.buildZuulRequestURI(request);

		Headers.Builder headers = new Headers.Builder();
		Enumeration<String> headerNames = request.getHeaderNames();
		while (headerNames.hasMoreElements()) {
			String name = headerNames.nextElement();
			Enumeration<String> values = request.getHeaders(name);

			while (values.hasMoreElements()) {
				String value = values.nextElement();
				headers.add(name, value);
			}
		}
```

```
            InputStream inputStream = request.getInputStream();

            RequestBody requestBody = null;
            if (inputStream != null && HttpMethod.permitsRequestBody(method)) {
                MediaType mediaType = null;
                if (headers.get("Content-Type") != null) {
                    mediaType = MediaType.parse(headers.get("Content-Type"));
                }
                requestBody = RequestBody.create(mediaType,
StreamUtils.copyToByteArray(inputStream));
            }

            Request.Builder builder = new Request.Builder()
                    .headers(headers.build())
                    .url(uri)
                    .method(method, requestBody);

            Response response = httpClient.newCall(builder.build()).execute();

            LinkedMultiValueMap<String, String> responseHeaders = new
LinkedMultiValueMap<>();

            for (Map.Entry<String, List<String>> entry :
response.headers().toMultimap().entrySet()) {
                responseHeaders.put(entry.getKey(), entry.getValue());
            }

            this.helper.setResponse(response.code(), response.body().byteStream(),
                    responseHeaders);
            context.setRouteHost(null); // prevent SimpleHostRoutingFilter from running
            return null;
    }
}
```

The preceding filter translates Servlet request information into OkHttp3 request information, executes an HTTP request, and translates OkHttp3 response information to the Servlet response.

How to Write a Post Filter

Post filters typically manipulate the response. The following filter adds a random UUID as the X-Sample header:

```
public class AddResponseHeaderFilter extends ZuulFilter {
    @Override
    public String filterType() {
        return POST_TYPE;
    }

    @Override
```

```
	public int filterOrder() {
		return SEND_RESPONSE_FILTER_ORDER - 1;
	}

	@Override
	public boolean shouldFilter() {
		return true;
	}

	@Override
	public Object run() {
		RequestContext context = RequestContext.getCurrentContext();
	HttpServletResponse servletResponse = context.getResponse();
		servletResponse.addHeader("X-Sample", UUID.randomUUID().toString());
		return null;
	}
}
```

[Note]

Other manipulations, such as transforming the response body, are much more complex and computationally intensive.

How Zuul Errors Work

If an exception is thrown during any portion of the Zuul filter lifecycle, the error filters are executed. The **SendErrorFilter** is only run if **RequestContext.getThrowable() is not null**. It then sets specific **javax.servlet.error.*** attributes in the request and forwards the request to the Spring Boot error page.

Zuul Eager Application Context Loading

Zuul internally uses Ribbon for calling the remote URLs. By default, Ribbon clients are lazily loaded by Spring Cloud on first call. This behavior can be changed for Zuul by using the following configuration, which results eager loading of the child Ribbon related Application contexts at application startup time. The following example shows how to enable eager loading:

application.yml.

```
zuul:
  ribbon:
    eager-load:
      enabled: true
```

Retrying Failed Requests

Spring Cloud Netflix offers a variety of ways to make HTTP requests. **You can use a load balanced RestTemplate, Ribbon, or Feign**. No matter how you choose to create your HTTP requests, there is always a chance that a request may fail. **When a request fails, you may want to have the request be retried automatically. To do so when using Sping Cloud Netflix, you need to include Spring Retry on your application's classpath. When Spring Retry is present, load-balanced RestTemplates, Feign, and Zuul automatically retry any failed requests (assuming your configuration allows doing so).**

BackOff Policies

By default, no backoff policy is used when retrying requests. If you would like to configure a backoff policy, you need to create a bean of type **LoadBalancedRetryFactory** and override the **createBackOffPolicy** method for a given service, as shown in the following example:

```
@Configuration
public class MyConfiguration {
   @Bean
   LoadBalancedRetryFactory retryFactory() {
      return new LoadBalancedRetryFactory() {
         @Override
         public BackOffPolicy createBackOffPolicy(String service) {
            return new ExponentialBackOffPolicy();
         }
      };
   }
}
```

Configuration

When you use Ribbon with Spring Retry, you can control the retry functionality by configuring certain Ribbon properties. To do so, set the **client.ribbon.MaxAutoRetries, client.ribbon.MaxAutoRetriesNextServer**, and **client.ribbon.OkToRetryOnAllOperations properties**.

Enabling **client.ribbon.OkToRetryOnAllOperations** includes retrying POST requests, which can have an impact on the server's resources, due to the buffering of the request body.

In addition, you may want to retry requests when certain status codes are returned in the response. You can list the response codes you would like the Ribbon client to retry by setting the clientName.ribbon.retryableStatusCodes property, as shown in the following example:

```
clientName:
  ribbon:
    retryableStatusCodes: 404,502
```

You can also create a bean of type LoadBalancedRetryPolicy and implement the retryableStatusCode method to retry a request given the status code.

Zuul
You can turn off Zuul's retry functionality by setting zuul.retryable to false. You can also disable retry functionality on a route-by-route basis by setting zuul.routes.routename.retryable to false.

HTTP Clients
Spring Cloud Netflix automatically creates the HTTP client used by Ribbon, Feign, and Zuul for you. However, you can also provide your own HTTP clients customized as you need them to be. To do so, you can create a bean of type ClosableHttpClient if you are using the Apache Http Cient or OkHttpClient if you are using OK HTTP.

Note
When you create your own HTTP client, you are also responsible for implementing the correct connection management strategies for these clients. Doing so improperly can result in resource management issues.