Netflix Stack Learning with Sample code

Purpose of this Exercise, to learn the microservice architecture with good example from leading microservice pioneer.

The Example of this slide can be found at https://github.com/ewolff/microservice before starting the exploring further in slide , please look to get the understanding of code once, and download the code on your local PC.

It consists of three microservices:

- ✓ The **catalog** microservice that manages the information about the items.
- ✓ The **customer** microservice that stores the data of the customers.
- ✓ The **order** microservice that can accept new orders by using the catalog and the customer microservice.

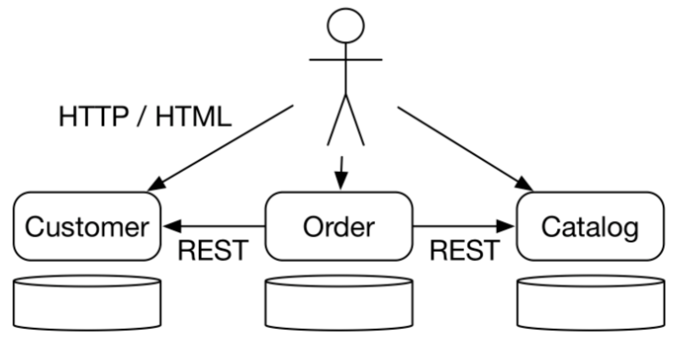
Architecture of the above example shown in right-side top in Pict-01

- ✓ Each of the microservices has its own web interface with which users can interact.
- ✓ Among each other, the microservices communicate via REST.
- ✓ The order microservice requires information about customers and items from the other two microservices.

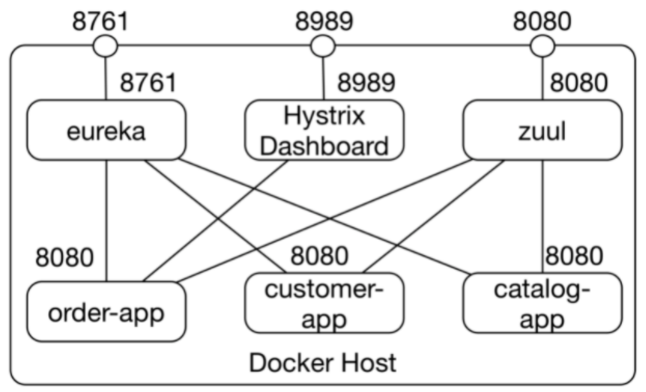
In addition to the microservices, there is a Java application that displays the Hystrix dashboard were monitoring the Hystrix circuit breakers is visualized.

Pict-02 referring The Docker containers communicate via an internal network. Some Docker containers can also be used via a port on the Docker host. The **Docker host** is the computer on which the Docker containers run.

The three microservices **order**, **customer**, and **catalog** each run in their own Docker containers. Access to the Docker containers is only possible within the Docker network.



Pict- 01



Pict- 02

Netflix Stack Learning with Sample code

Let's dive little more in the Exercise code.

Routing via Zuul (Load balancer)

In this example and the right Pict-02 of Docker host, you can refer that Services from the outside get connected to the Services like (Order, Customer, Catalog) using Zuul routing service on port 8080.

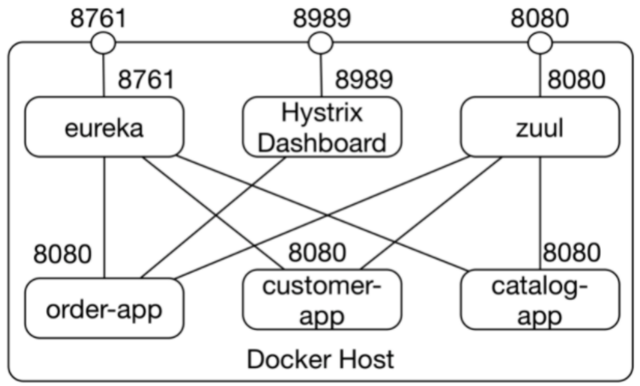
- ✓ The Zuul container can be accessed from outside under port 8080 and forwards requests to the microservices.
- ✓ If the Docker containers are running locally, the URL is **http://localhost:8080**.
- ✓ At this URL, there is also a web page available which includes links to all microservices, Eureka, and the Hystrix dashboard.

I just introduced Eureka from Netflix in Pict-03. So let me explain the Eureka which is primarily the Service Discovery for the multiple service.

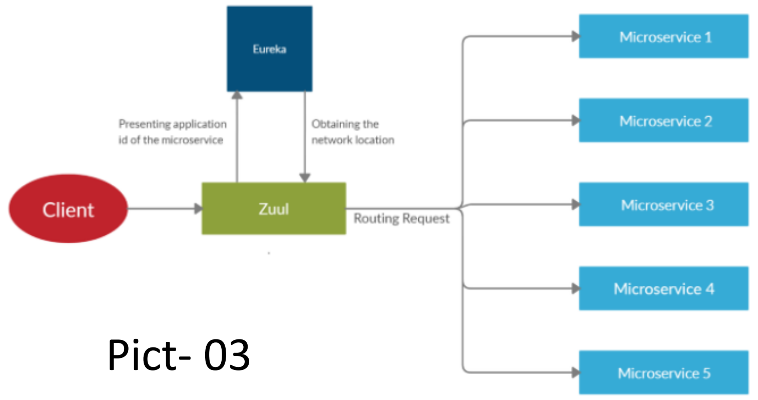
Eureka serves as a service discovery solution.

- ✓ The dashboard is available at port 8761.
- ✓ This port is also accessible at the Docker host.
- ✓ For a local Docker installation, the URL is **http://localhost:8761**.

Hystrix dashboard
Finally, the Hystrix dashboard runs in its own Docker container that can also be accessed under port 8989 on the Docker host, for example at http://localhost:8989.



Pict- 02



Pict- 03

Eureka : Service Discovery

Eureka
Eureka implements service discovery.

For synchronous communication, microservices must find out at which port and IP address other microservices can be accessed.

Let's discuss some essential characteristics of Eureka.

Eureka has a REST interface.

Microservices can use this interface to **register** or **request information** about other microservices.

Eureka supports replication.

The information from the Eureka servers is distributed to other servers enabling the system to compensate for the failure of Eureka servers.

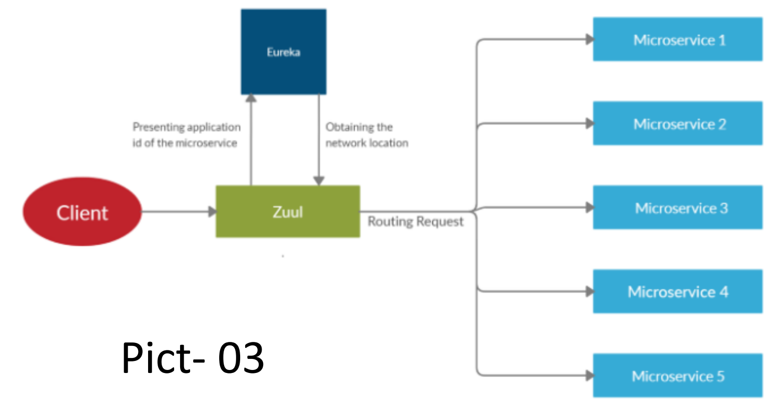
Eureka is Architected in the way; client can do caching which improves the overall experience at client side.

The Eureka server only sends information to the client about new or deleted microservices and not information about all registered services, communication very efficient.

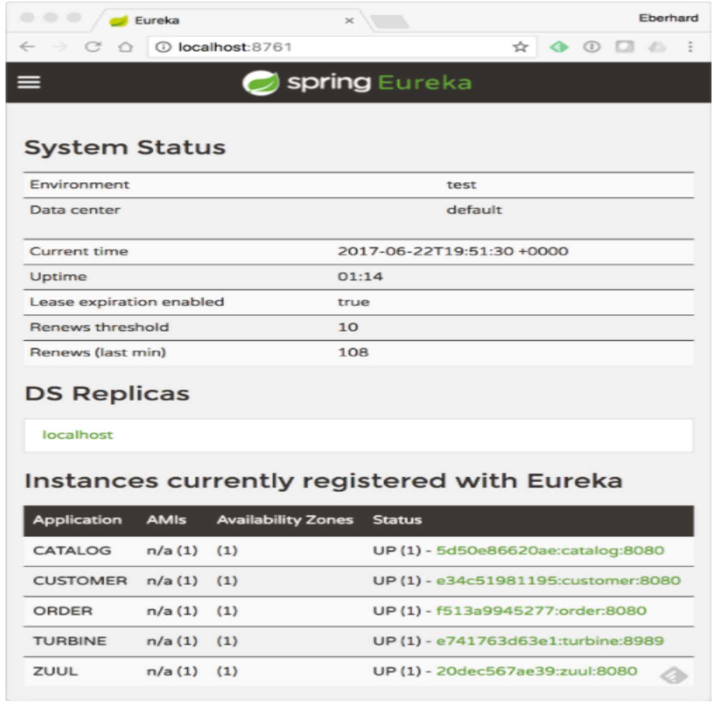
So, by caching at client-side Eureka performance & Scale also improved.

Eureka expects the microservices to regularly send heartbeats.

In this way, Eureka detects crashed instances and excludes them from the system. This increases the probability that Eureka will return service instances that are available.



Pict- 03



System Status

Environment	test
Data center	default

Current time	2017-06-22T19:51:30 +0000
Uptime	01:14
Lease expiration enabled	true
Renews threshold	10
Renews (last min)	108

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CATALOG	n/a (1)	(1)	UP (1) - 5d50e86620ae:catalog:8080
CUSTOMER	n/a (1)	(1)	UP (1) - e34c51981195:customer:8080
ORDER	n/a (1)	(1)	UP (1) - f513a9945277:order:8080
TURBINE	n/a (1)	(1)	UP (1) - e741763d63e1:turbine:8989
ZUUL	n/a (1)	(1)	UP (1) - 20dec567ae39:zuul:8080

Eureka : Service Discovery

Eureka

The Netflix Eureka project is available for download at GitHub (https://github.com/Netflix/eureka/)

You can build the project and get both the server and the client.

Eureka provides a dashboard, see the screenshot above.

- ✓ It displays an overview of the microservices which are registered with Eureka.
    - This includes the names of the microservices and the URLs at which they can be accessed.
- ✓ However, the URLs only work in the Docker internal network meaning the links in the dashboard do not work.
- ✓ The dashboard is accessible on the Docker host at port 8761 i.e., http://localhost:8761/ if the example runs locally.

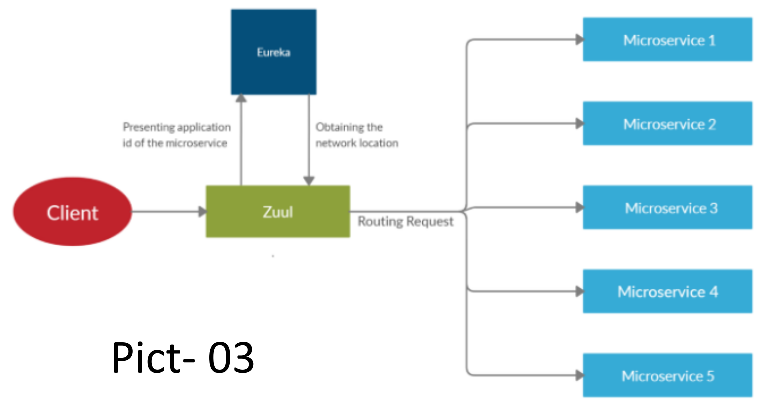
Registration
See the example of Catalog service registration

```
spring.application.name=catalog
eureka.client.serviceUrl.defaultZone=http://eureka:8761/eureka/
eureka.instance.leaseRenewalIntervalInSeconds=5
eureka.instance.metadataMap.instanceId=${spring.application.name}:${random.value}
eureka.instance.preferIpAddress=true
```

There is always a question that does this Eureka only supports Java or other programming language also possible.

The Answer is, Yes it supports the other language but for other language there is a concept of **sidecar.**

A sidecar is an application written in Java that uses the Java libraries to talk to the Netflix infrastructure.



Pict- 03

So, for other programming language written client can interact with Netflix Infra using sidecar but integration is putting additional layer and some of the resource consumed by this layer also.

Netflix itself offers Prana as a sidecar. Spring Cloud also provides an implementation of such a sidecar.

Access to other services

In the example in this article, **Ribbon** implements the access to other **services in order to implement load balancing across** multiple instances.

Thus, **the Eureka API is only used via Ribbon** to find information about other microservices.

Ribbon : Understanding of Load balancing

Load balancing with Ribbon

Microservices have the advantage that each microservice can be scaled independently of the other microservices.

But to achieve this flexibility, you need some central Entity who can distribute the load for that specific microservice call. Here you note that, central entity is called Load Balancer.

What if this Central Entity Failed? There is a possibility of Failure so what if it fails. Basically, this is the question to an architect that did you architected system with Enough Resiliency or not?

The load balancer is also a single point of failure. If the load balancer fails, all network traffic stops functioning and the entire microservices system fails.

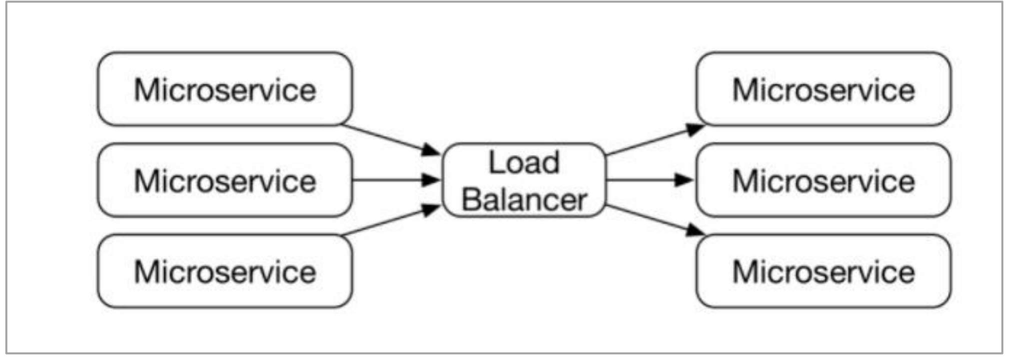
So, what is the solution here. Decentralized Load Balancing.

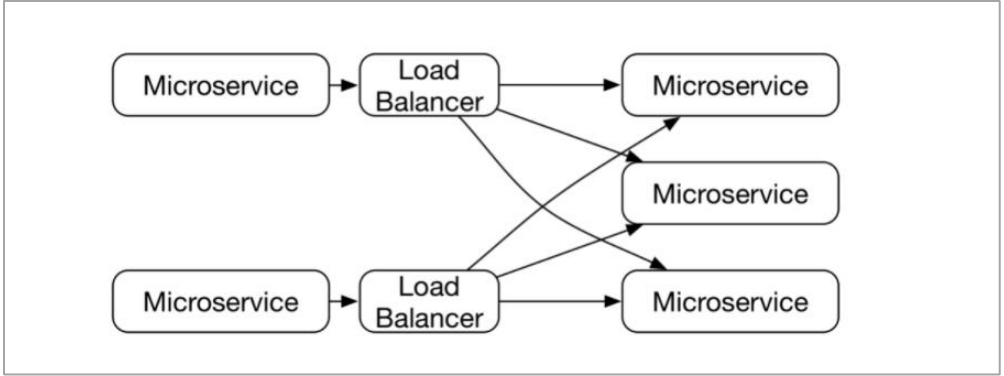
For this, each microservice must have its own load balancer. If such a load balancer fails, only one microservice will fail.

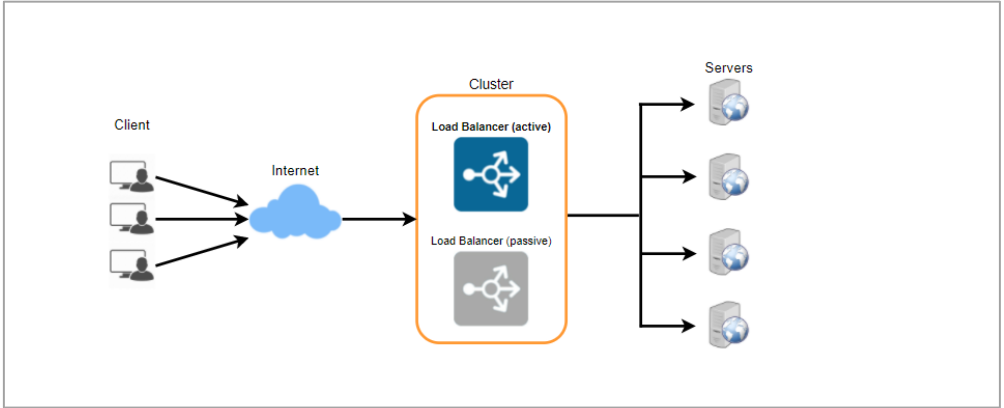
Another option as an improvement of this exercise is to manage multiple copy of Load balancer and your DNS / router can be used for checking the Load balancer Heartbit and accordingly direct the traffic to the live load balancer.

But this kind of system, becoming complex architecture and required in production system.

It is also possible to write a library that distributes requests to other microservices to different instances. This library must read the currently available microservice instances from the service discovery and then, for each request, select one of the instances. This is how Ribbon works.







Hystrix : Understanding of Resilience

Resilience with Hystrix

With synchronous communication between microservices, it is important that the failure of one microservice does not cause other microservices to fail as well.
The microservices may return errors because they cannot deliver reasonable results due to a failed microservice.
However, it must not happen that a microservice waits for the result of another microservice for an infinite period and thereby becomes unavailable itself.

So, In a nutshell there are few standard Resilience patterns that would be part of Hystrix system.

✓ **Timeout**
A timeout prevents a microservice from waiting too long for another microservice.

Hystrix executes a request in a separate thread pool. Hystrix controls these threads and can terminate the request to implement the timeout.

✓ **Fail fast**
Fail Fast describes a similar pattern. It is better to generate an error as quickly as possible.
The code can check at the beginning of an operation whether all necessary resources are available.
If this is not the case, the request can be terminated immediately with an error. This reduces the time that the caller must block a thread or other resources.

✓ **Bulkhead**
Hystrix can use its own thread pool for each type of request. For example, a separate thread pool can be set up for each called microservice.
If the call of a particular microservice takes too long, only the thread pool for that microservice is emptied, while the others still contain threads.
This will limit the impact of the problem and is called a **bulkhead**.

✓ **Circuit breaker**
This is a fuse analogous to the ones used in the electrical system of a house. There, a circuit breaker is used to cut off the current flow if there is a short circuit, preventing a fire from breaking out.

The Hystrix circuit breaker has a different approach. If a system call results in an error, the circuit breaker is opened and does not allow any calls to pass through.

After some time, a call is allowed to pass through again. Only when this call is successful, is the circuit breaker closed again. This prevents a faulty microservice from being called. This saves resources and avoids blocked threads.

These Patterns are part of the Hystrix implementation of Netflix. So, I am going to explain you little more about Hystrix Implementation in next page.

Hystrix : Implementation

Hystrix Implementation

Implementation
Hystrix offers an implementation of most resilience patterns as a Java library.

The Hystrix API requires command objects instead of simple method calls. These classes supplement the method call with the necessary Hystrix functionalities.

When using Hystrix with Spring Cloud, it is not necessary to implement commands. Instead, the methods are annotated with @HystrixCommand. It activates Hystrix for this method and the attributes of the annotation configure Hystrix.

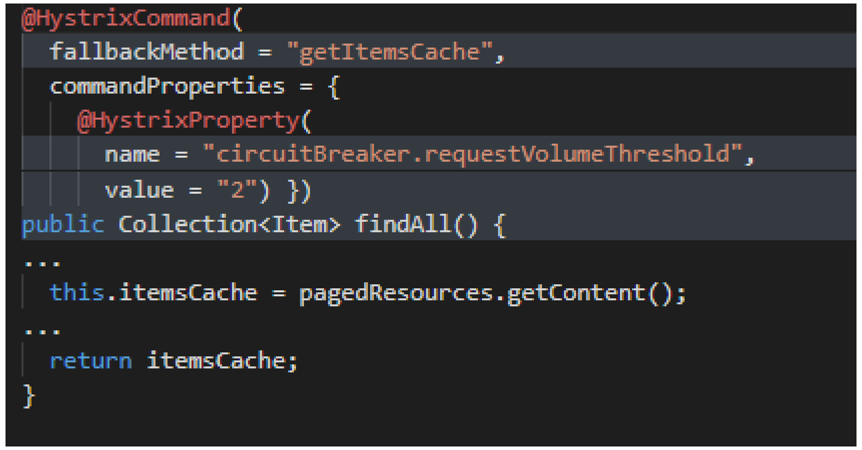
Some Improvements as Variation in discussion

There are various alternatives to the technologies of the Netflix stack.

An alternative might be Zuul2. It is based on asynchronous I/O, so it consumes less resources and is more stable. However, Spring Cloud won't support Zuul2.

Netflix does not invest in Hystrix that much anymore. They suggest using resilience4j instead, which is a very similar Java library that supports typical resilience patterns.

Consul supports DNS and can handle any programming language as implemented in the Consul DNS example. Consul Template offers the possibility to configure services with Consul by filling a configuration file template with the data from Consul. In the example, Apache httpd is configured this way.



```
@HystrixCommand(
  fallbackMethod = "getItemsCache",
  commandProperties = {
    @HystrixProperty(
      name = "circuitBreaker.requestVolumeThreshold",
      value = "2") })
public Collection<Item> findAll() {
...
  this.itemsCache = pagedResources.getContent();
...
  return itemsCache;
}
```