

# Microservices Discovering and Orchestration

It is a car-info application built of microservices, orchestrated by the Netflix family services:

- Eureka server as a service registry.
- Zuul is a router for requests from devices and websites to the backend of streaming applications.
- Feign is a discovery-aware REST services client
- Hystrix is a circuit-breaker, responsible for error handling.
- Ribbon is a load balancer.

You will create three microservices:

1. Car Catalog
2. Car Selector
3. Eureka microservices discovery server

The car selector will request and consume data from the car catalog, using discovery services from Eureka and support from Feign.

Ribbon will take care of replication and balanced loading.

Hystrix will react in case of cascading errors.

## MS1: Microservices Discovery Server

First, create and run the server. From the start it will discover nothing, as no other MS will be running.

It allows individual microservices to register themselves at deployment and running. When a client requires a particular service, it can obtain a list of running instances of the service from the discovery tool and then to request one.

1. Create normal Spring Boot/maven application, initialized with **Spring Cloud Discovery / Eureka Server** dependency.
2. Add annotation `@EnableEurekaServer` at the top of the main class to make it a microservices registry.
3. Add parameters to `application.properties` file

```
spring.application.name = eureka-server
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

(the last configuration says Eureka shouldn't discover itself)

4. Run and observe the dashboard at **localhost:8761**. No microservice is registered yet.

## MS2: Car Catalog Microservice

This microservice will manage a H2 database **Cars**.

1. It is a normal web service, Spring Boot/maven application, initialized with the following eight dependencies:
  - a) **DevTools**: to auto-reload the application when files change
  - b) **Lombok**: to reduce the boilerplate code
  - c) **Web**: to be built as web app and get embedded Tomcat web server

- d) **Rest Repositories**: to expose data repositories as REST endpoints at localhost
- e) **Eureka Discovery Client**: to register at Eureka and enable discovering
- f) **H2**: to create in-memory SQL database
- g) **JPA**: to convert data from a database to an object
- h) **Actuator**: from Ops, to create monitoring endpoints and help you monitor your application

## 2. Create and configure a data source **Car**

- a) Add parameters to **application.properties** file

```
server.port=8090
spring.application.name=car-catalog
eureka.client.serviceUrl.defaultZone=${EUREKA_SERVER:http://localhost:8761/eureka}

# H2 database
spring.datasource.url=jdbc:h2:mem:car
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=sa
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
# Enabling H2 Console
spring.h2.console.enabled=true
# Custom H2 Console URL
spring.h2.console.path=/h2

# JPA Configuration
spring.jpa.show-sql = true
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=update
```

- b) create resource file **schema.sql**

```
DROP TABLE IF EXISTS Car;

CREATE TABLE Cars (
  id INT AUTO_INCREMENT PRIMARY KEY,
  brand VARCHAR(50) NOT NULL,
  year INT NOT NULL,
  km INT NOT NULL
);
```

- c) create resource file **data.sql**

```
INSERT INTO Car (brand, year, km) VALUES
('Ferrari', 2009, 250000),
('Lamborghini', 1985, 50000),
('Bugatti', 2017, 35000),
('Renault', 2005, 150000),
('Mini', 2014, 75000);
```

- 3. Add annotation **@EnableDiscoveryClient** at the top of the main class to enable the microservice to register itself.
- 4. Create REST API - **model** and **controller** components of the microservice.

## MS3: Car Gateway Microservice

Create another microservice, which will communicate with the first one over the discovery server.

Ensure dependencies, as follow:

- **Eureka Discovery:** for service registration
- **OpenFeign:** for routing
- **Rest Repositories:** to expose JPA repositories as REST endpoints
- **Web:** Web app and embedded Tomcat web server
- **Hystrix and Hystrix Dashboard:** a circuit breaker to stop cascading failure and enable resilience
- **Actuator:** to monitor the processes
- **Lombok:** to reduce boilerplate code

This microservice exposes a `/mycars` endpoint that filters out some cars I want to see.

Add parameters to `application.properties` file

```
server.port=8080
spring.application.name=car-gateway
eureka.client.serviceUrl.defaultZone=${EUREKA_SERVER:http://localhost:8761/eureka}
feign.hystrix.enabled=true
hystrix.shareSecurityContext=true

management.endpoints.web.exposure.include=hystrix.stream
management.endpoints.web.exposure.include=env,info,health,loggers,mappings
management.endpoints.web.exposure.include=httptrace, metrics, caches

car-catalog.ribbon.listOfServers=localhost:8090,localhost:8092
server.ribbon.eureka.enabled=true
server.ribbon.ServerListRefreshInterval=1000
```

## Hystrix and Error Recovery Strategy

After setting up the microservices, they need to communicate with each other over the network. API calls to a microservice could easily fail due to a variety of reasons. In such situations, you can use **Hystrix** to provide some level of fault tolerance.

Hystrix lets you define a fallback method that gets invoked if your network calls to another microservice fails. It reverts back to normal behavior once the service is available again.

Our microservice `car-gateway` uses Feign and Hystrix to talk to the `car-catalog` microservice and fallback to a `fallback()` method if it's unavailable

Check for the following dependencies in the `pom.xml` file:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
  <version>2.0.0.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
  <version>2.0.0.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

## Ribbon and Load Balancing

**Ribbon** is a load balancer, which can be configured to automatically obtain a list of instances of a service from Eureka and query the service while balancing the load.

All of queries to a microservice are routed through Ribbon, which talks with Eureka to find the actual address of the microservice instance that you need to query.

For testing the load balancing, create 3 instances of car-catalog microservice that run on 3 different ports.

You can use

```
java -jar -Dserver.port=XXXX <path-to-jar>.yyyy.jar
```