Netflix microservices

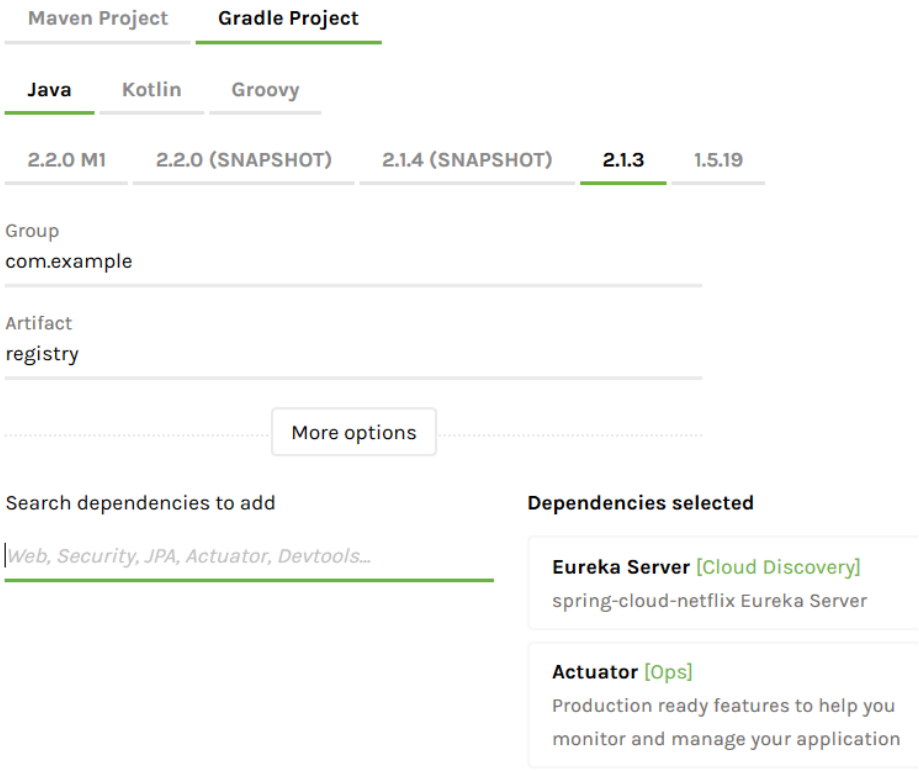
Spring cloud Netflix (https://spring.io/projects/spring-cloud-netflix)

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)..

Service registration and discovery

The registry

Use the Spring initializer to create a Gradle project with Eureka Server and Actuator as dependency:



Then, unzip the project and use Eclipse or Intellij:

- Intellij: simply open the project
- Eclipse: import the project as a Gradle project

Add a Yaml file named application.yml in src/main/resources containing the registry configuration:

```
spring:
  freemarker:
    template-loader-path: classpath:/templates/
    prefer-file-system-access: false
```

```yaml
server:
   port: 8761
```

Notes:

- Only one configuration file should be present and if the file application.yml is present, you should delete application.properties.
- Take care at the format of the yml file: respect the indentation at the beginning of each line (use double white spaces or tabulation). Take care also at unexpected invisible characters if you copy paste the content from a pdf file.

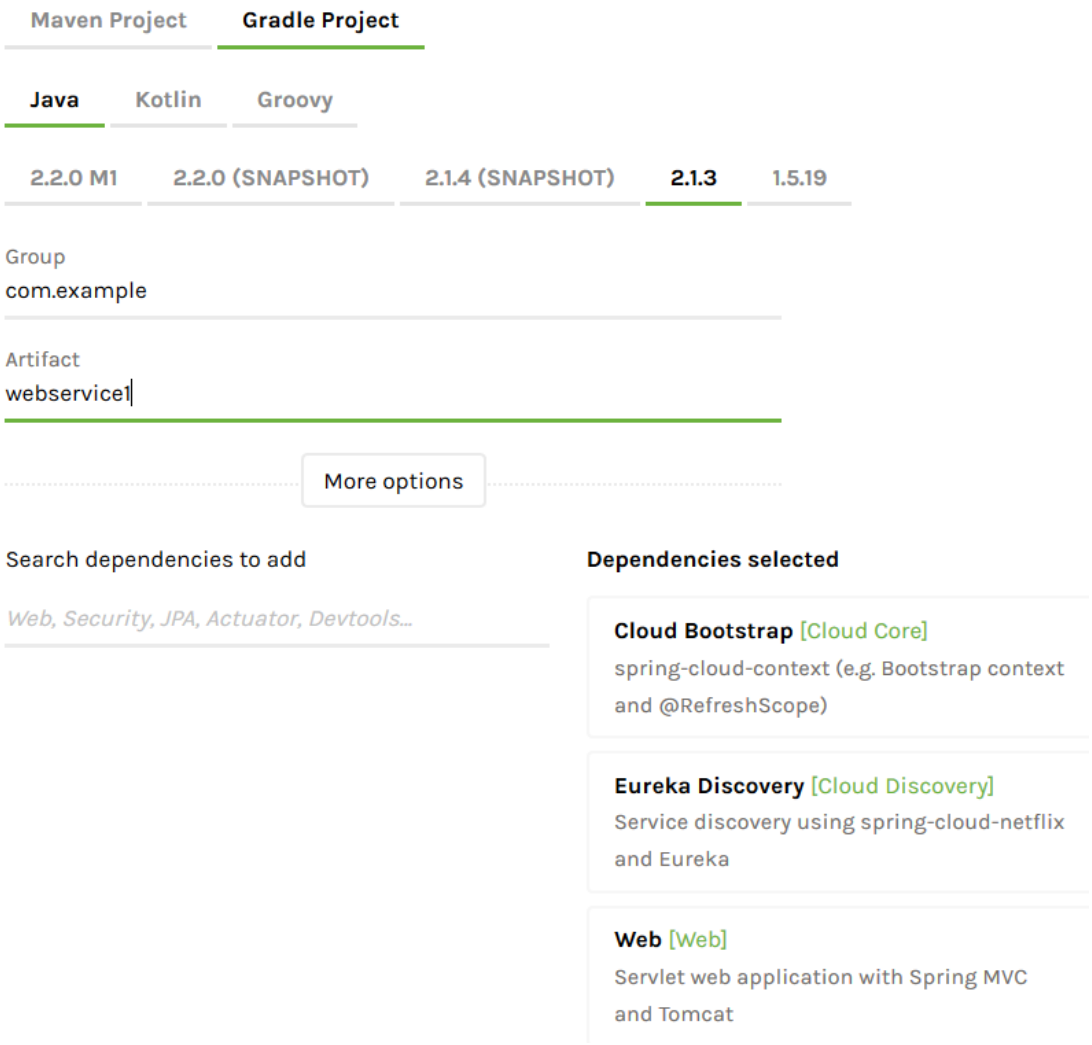
Simply add EnableEurekaServer to your main program then start it.

```java
package com.example.registry;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class RegistryApplication {

    public static void main(String[] args) {
        SpringApplication.run(RegistryApplication.class, args);
    }

}
```

Then check the Eureka console at: http://localhost:8761/

Registration of a web service in the Eureka registry

Create a new project using Spring initializer:

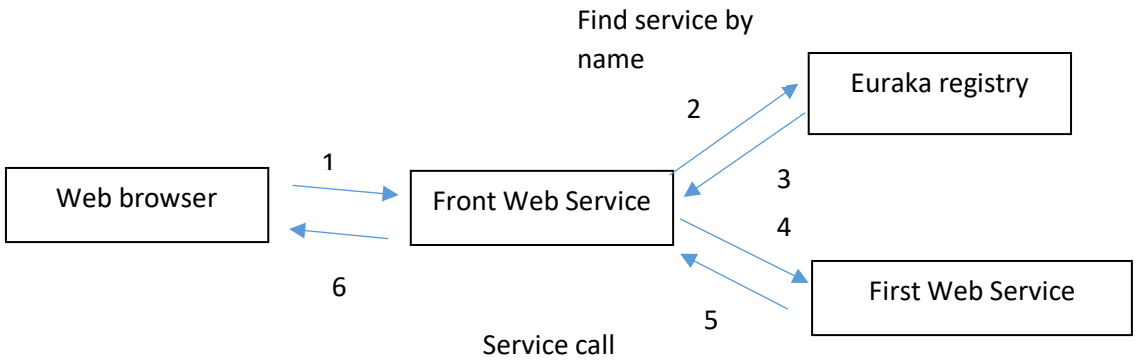


Then code a rest web service (don't forget to add methods with http mapping like get, put, post or delete).

The following configuration (application.yml) sets a name (`name-of-the-microservice1`) for this web service and register it into the Eureka registry:

```
spring:
  application:
    name: name-of-the-microservice1
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
  healthcheck:
    enabled: true
```

Service discovery



Create a third application (front web service) that communicates with the first web service:

- this last web service should also be registered into Eureka
- it should also discover the first web service using

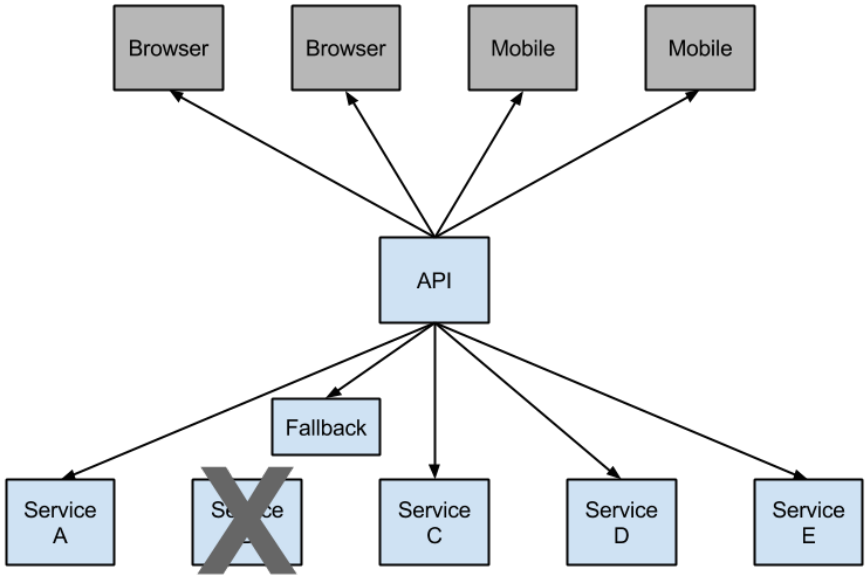
```
@RestController
public class FrontwebserviceApplication {
  @Autowired
  DiscoveryClient discoveryClient;
  @GetMapping("/")
  public String hello() {
    List<ServiceInstance> instances = discoveryClient.getInstances("name-of-the- microservice1");
    ServiceInstance test = instances.get(0);
    String hostname = test.getHost();
    int port = test.getPort();

   RestTemplate restTemplate = new RestTemplate();
   String microservice1Address = "http://" + hostname + ":" + port;
   ResponseEntity<String> response =
   restTemplate.getForEntity(microservice1Address, String.class);
   String s = response.getBody();

   Return s;
  }
}
```

Where `DiscoveryClient` comes from the Spring framework.

- then it should call the first web service using RestTemplate (https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html)

Circuit breaker (https://cloud.spring.io/spring-cloud-netflix/spring-cloud-netflix.html#_circuit_breaker_hystrix_clients)

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.



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.

How to?

Update the dependencies of the front office web service (build.gradle file) with hystrix:

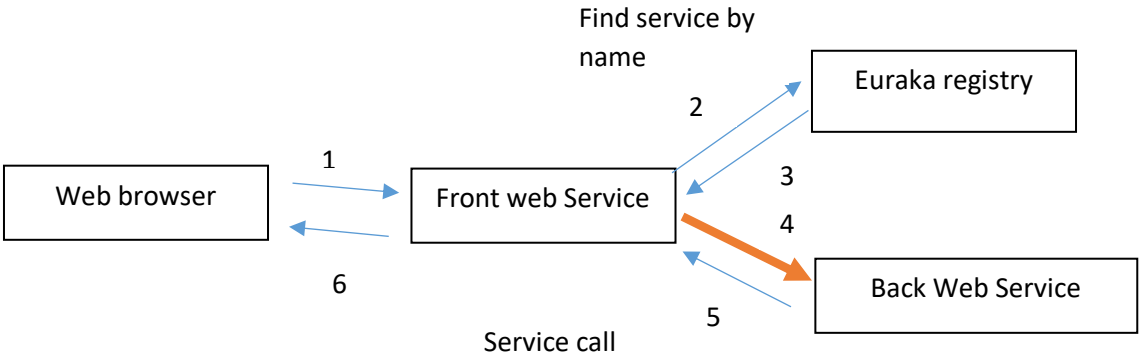
```
dependencies {
        implementation 'org.springframework.boot:spring-boot-starter-web'
        implementation 'org.springframework.boot:spring-boot-starter-actuator'
        runtimeOnly 'org.springframework.boot:spring-boot-devtools'
        implementation 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'
        implementation 'org.springframework.cloud:spring-cloud-starter-netflix-hystrix'
        testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

Unable the circuit breaker in the main program:

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;

@SpringBootApplication
```

```
@EnableCircuitBreaker
public class Microservice2Application {

	public static void main(String[] args) {
		SpringApplication.run(Microservice2Application.class, args);
	}

}
```

Add the circuit breaker between the front and the front and the back web service (red arrow)



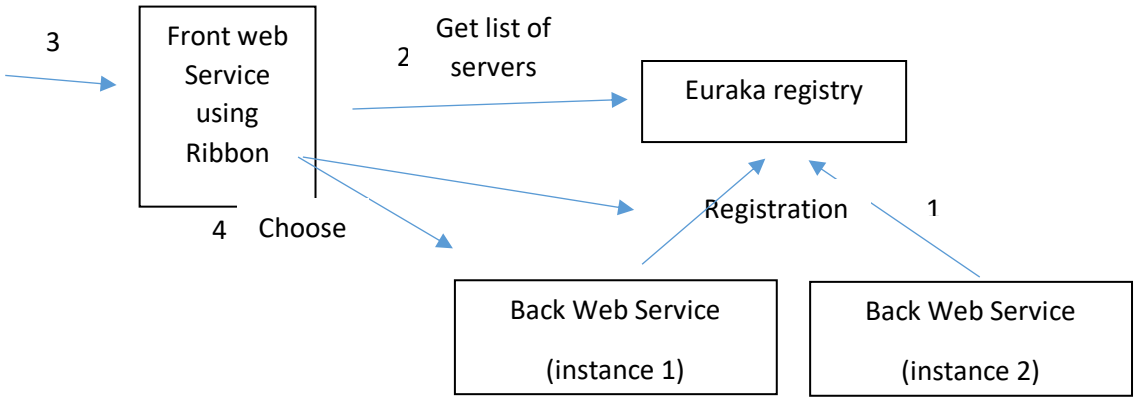
so that in the case of a failure an alternative response is sent:

```
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;


@HystrixCommand(fallbackMethod = "defaultMessage")
@GetMapping(…)
public String hello() {
	…
}

public String defaultMessage() {
	return "Salut !";
}
```

Load balancer (https://cloud.spring.io/spring-cloud-netflix/spring-cloud-netflix.html#spring-cloud-ribbon)

Ribbon is a client-side load balancer that gives you a lot of control over the behavior of HTTP and TCP clients.

It can retrieve a list of servers and then choose among then the one to be used:



How to?

Create another application for Ribbon based on the following dependencies:

```
dependencies {
	implementation 'org.springframework.boot:spring-boot-starter-web'
	implementation 'org.springframework.boot:spring-boot-starter-actuator'
	runtimeOnly 'org.springframework.boot:spring-boot-devtools'
	implementation 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'
	implementation 'org.springframework.cloud:spring-cloud-starter-netflix-ribbon'
	testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

Then it is possible to choose among the server the one given by its name:

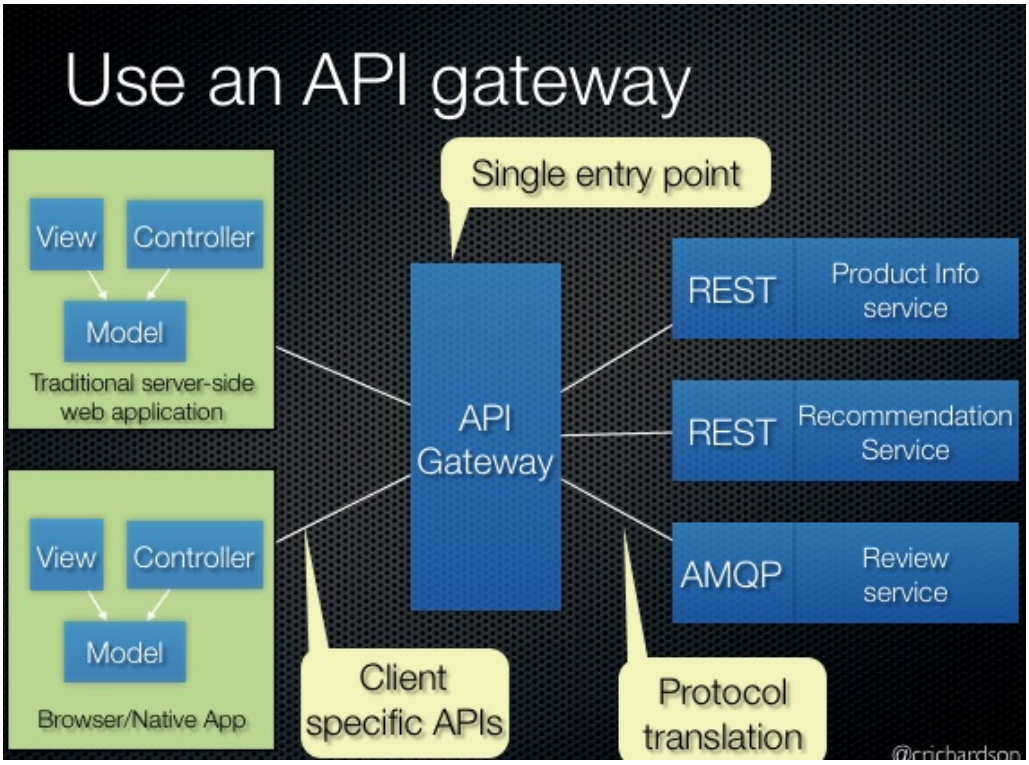
```
@RestController
public class Microservice3 {


	@Autowired
	private LoadBalancerClient loadBalancer;

	@GetMapping(…)
	public void method() {
		ServiceInstance serviceInstance = loadBalancer.choose("name-of-the-microservice1");
		System.out.println(serviceInstance.getUri());
	}
```

}

Zuul (https://cloud.spring.io/spring-cloud-netflix/spring-cloud-netflix.html#_router_and_filter_zuul)

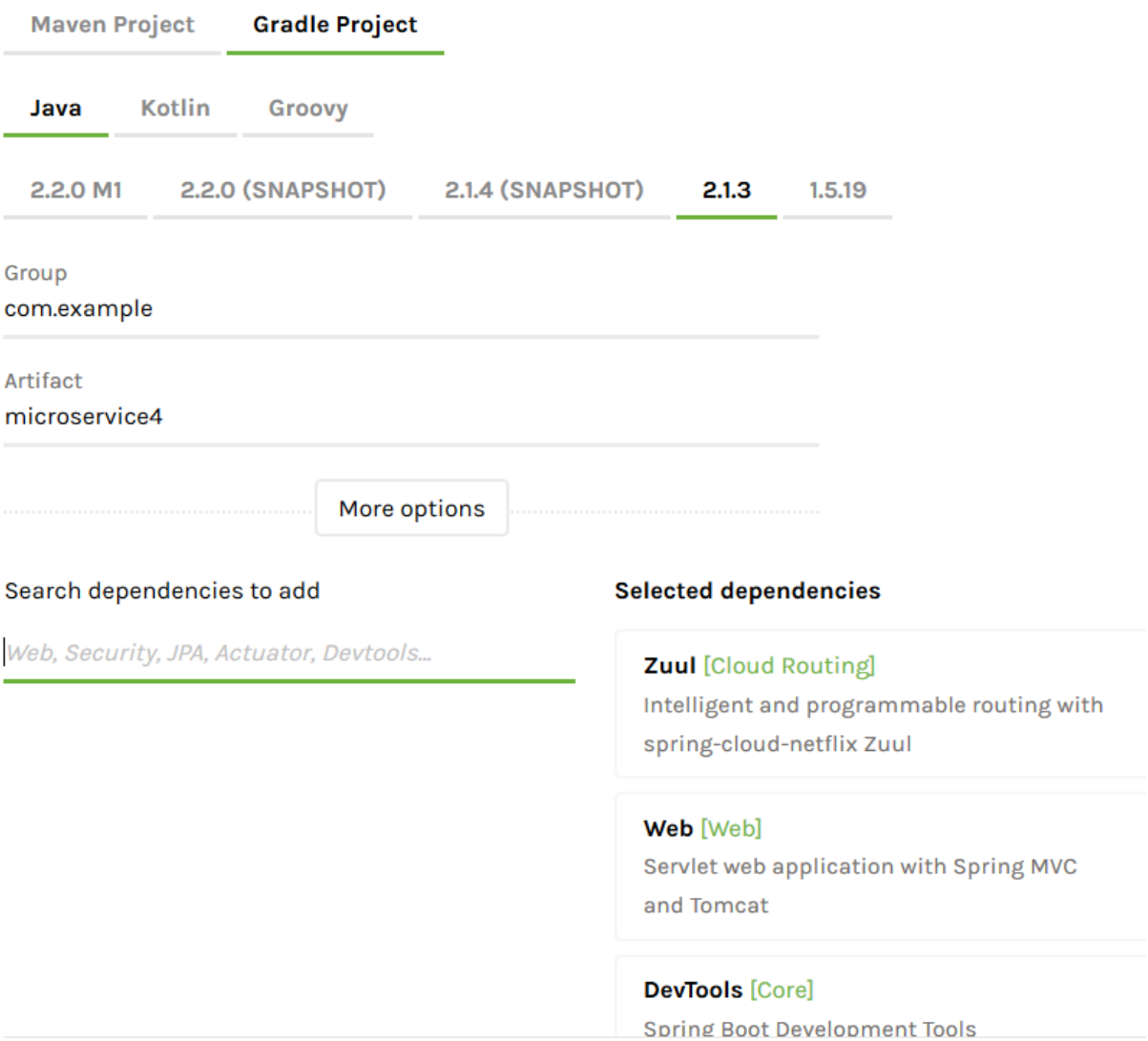
Zuul is a gateway service that provides dynamic routing, monitoring, resiliency, security, and more.

An API Gateway is a single entry point for all the micro-services (https://microservices.io/patterns/apigateway.html):



How to?

Create another application for Zuul based on the following dependencies:



Enable Zuul in the main program:

```
package com.example.microservice4;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

@SpringBootApplication
@EnableZuulProxy
public class Microservice4Application {

	public static void main(String[] args) {
		SpringApplication.run(Microservice4Application.class, args);
	}
```

```
}
```

Then configure this application in a application.yml file:

```
server:
  port: 8484
zuul:
  routes:
    foos:
      path: /foos/**
      url: http://localhost:8080/
```

The application will start on port 8484 and each time a request comes from a client on the URL foos it will be redirected to localhost:8080 (which is the address of the one of the micro-services).

Enable security

Spring Security can do authentication and more. It provides default login mechanism.

Simply add this starter in the dependencies part of to the configuration file build.gradle:

```
implementation 'org.springframework.boot:spring-boot-starter-security'
```

Update your project (gradlew build and gradlew eclipse if necessary).

Then set a default login and password:

```
package com.example.microservice4;

import org.springframework.context.annotation.Bean;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity
;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;

@EnableWebSecurity
public class WebSecurityConfig {

	@Bean
	public UserDetailsService userDetailsService() throws Exception {
		UserDetails user =
User.withDefaultPasswordEncoder().username("user").password("password").roles("USE
R").build();
		return new InMemoryUserDetailsManager(user);
	}

}
```

Finally try to access the main page: http://localhost:8484/foos

The default logout page is accessible at: http://localhost:8484/login?logout