



## Spring Cloud API Gateway

В этой статье продолжим дорабатывать предыдущий  с Eureka и client-side load balancing — добавим в него Spring Cloud API Gateway.

### ① Что такое Spring Cloud API Gateway

Это отдельное Spring Boot приложение, через которое проходят все запросы, реализация . То есть микросервисы не знают друг о друге, а обращаются к прокси. Внешнему пользователю тоже известен только прокси. Прокси, в свою очередь, анализирует запрос, перенаправляет его к нужному микросервису и возвращает ответ обратно. Ниже мы рассмотрим, как в прокси задать условия — какой запрос к какому микросервису направить.

### ② Зачем нужен Spring Cloud API Gateway

Например для того, чтобы зафиксировать REST API. Представьте, что вы разрабатываете микросервисы и обсуждаете с фронтенд-разработчиком REST API. У вас постоянно что-то меняется: сегодня микросервис выдает данные по такому url, завтра — по-другому. А то и вовсе данные будут выдаваться новым

микросервисом.

Можно зафиксировать REST API на прокси и менять внутреннюю структуру как угодно. Снаружи ничего не поменяется, просто прокси будет обращаться по другим адресам. А обращения к самому прокси останутся прежними.

### 3 Spring Cloud API Gateway vs. Zuul

Пример сделан на новом Spring Cloud API Gateway.

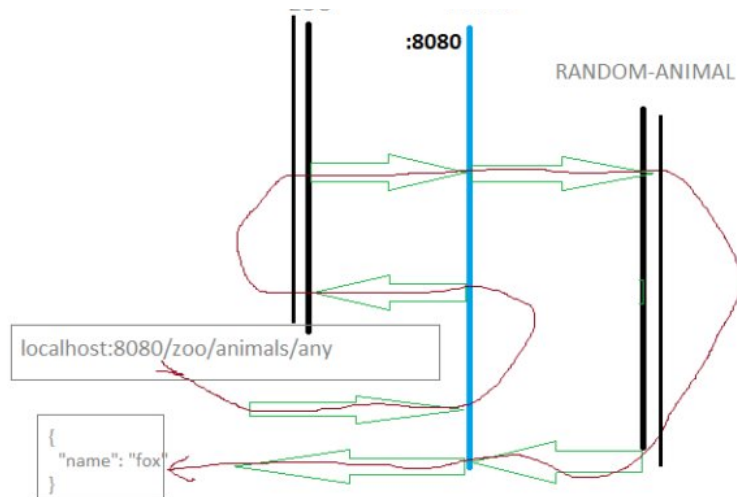
Zuul имеет примерно ту же функциональность, но Zuul 1.x не реактивный. Zuul 2.0 реактивный, но Spring его поддерживает хуже, чем Zuul 1.x. Поэтому если у нас реактивный стек, то Spring Cloud API Gateway — лучший выбор.

### 4 Наша структура

Мы продолжим разрабатывать [REDACTED]. Есть микросервис Zoo — он выдает случайное животное. Раньше пользователь в браузере обращался к нему, но теперь Zoo будет запущен в двух экземплярах (вот, еще одно преимущество), а пользователь будет обращаться к прокси, запущенному на порту 8080. **Это прокси и есть наш Spring Cloud API Gateway.**

Есть также микросервис Random Animal — к нему обращался Zoo, чтобы получить это животное, но **теперь Zoo будет обращаться тоже к прокси**. А прокси, в свою очередь, к Random Animal. Random Animal тоже запущен в двух экземплярах (но можно запустить сколько угодно, пример будет работать).

В общем картина такая:



Запросы проходят через прокси

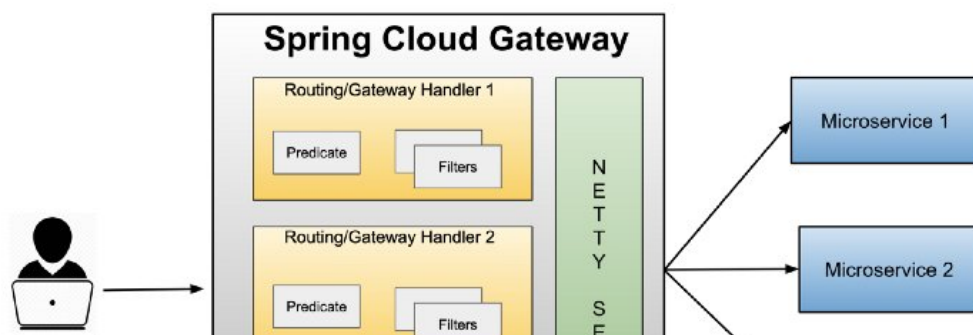
Проксу должен как-то решать, к какому микросервису направлять пришедший запрос.

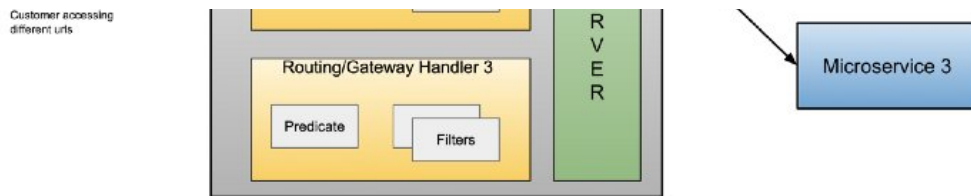
Например, пользователь в браузере обращается к прокси с запросом `localhost:8080/zoo/animals/any`, прокси решает, к какому микросервису перенаправить запрос, и выбирает Zoo. Потом Zoo обращается к прокси с другим запросом, и прокси решает перенаправить его к Random-Animal.

Эта логика задается с помощью элементов, перечисленных ниже.

## 5 Элементы Spring Cloud API Gateway

(Spring Cloud API Gateway работает на сервере Netty.)





Spring Cloud API Gateway

Чтобы сопоставить входной url (идуший в Spring Cloud Gateway) выходному url (идушему к микросервису), нужно задать три пункта:

1. **Predicate**: условие, при котором запрос перенаправляется (например, если *url* соответствует такому-то шаблону).
2. **URI**: содержит *uri* куда перенаправляем запрос (к какому микросервису).
3. **Filter** (необязательно): как модифицировать запрос (на пути туда или обратно).

Эти три пункта (еще идентификатор Route-a) объединены в **Route** — основной строительный блок. Из нескольких таких блоков и состоит настройка.

Два **Route**-а мы настроим ниже. Но сначала добавим в приложение Maven-зависимость и зададим ему имя (для Eureka).

## 6 Maven-зависимость

Чтобы сделать Spring Boot приложение API Gateway-ем, добавим зависимость:

```
1. <dependency>
2.     <groupId>org.springframework.cloud</groupId>
3.     <artifactId>spring-cloud-starter-
   gateway</artifactId>
4. </dependency>
```

Также (поскольку мы уже включили в предыдущий пример

Eureka), чтобы другие микросервисы могли обращаться к API Gateway-ю по имени, а не по адресу с портом (типа *localhost:8080*), сделаем Spring Cloud API Gateway клиентом Eureka:

```
1. <dependency>
2.     <groupId>org.springframework.cloud</groupId>
3.     <artifactId>spring-cloud-starter-netflix-
   eureka-client</artifactId>
4. </dependency>
```

А обращаться они будут к нему по имени *проху*, что и зададим ниже.

## 7 Имя приложения API Gateway

В нашей системе API Gateway будет обнаруживаться по имени *проху* — так мы назовем наше приложение:

```
1. server:
2.     port: 8080
3. spring:
4.     application:
5.         name: proxy
```

А запущен он будет на порту *8080*.

**i** В примере обращаться к прокси по имени *проху* будет микросервис Zoo. (Пользователь в браузере для обращения к прокси вводит обычный адрес *localhost:8080/...*, где запущен прокси).

## 8 Сопоставление адресов: настройка Route-ов: URI, Predicate и Filter

Итак, зададим, что если *url* обращения к прокси (который у нас на порту 8080) начинается с */zoo*:

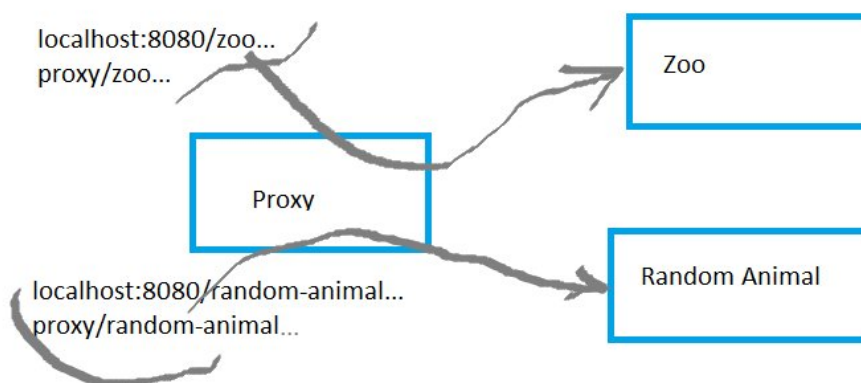
```
1. localhost:8080/zoo/....
```

то прокси переводит обращение на микросервис Zoo.

А если с */random-animal*:

```
1. localhost:8080/random-animal/...
```

то на микросервис Random Animal:



Какие запросы на какой микросервис идут



Еще раз обращаю внимание, что адрес до первого слеша / может быть как *localhost:8080*, так и просто имя *проху*. Это зависит от того, кто обращается. Микросервисы обращаются по имени *проху* благодаря Eureka, из браузера так нельзя.

Вообще настроить Spring Cloud API Gateway можно как в коде, так и в *application.yml*.

## Настройка в коде

Видно, что в настройке фигурируют два Route (как на картинке выше). Для каждого задан Uri, Predicate и Filter:

```
1.  @Configuration
2.  class ProxyConfig {
3.
4.      @Bean
5.      RouteLocator
6.      customRouteLocator(RouteLocatorBuilder builder) {
7.          return builder.routes()
8.              .route("random_animal_route",
9.                  route ->
10.                  route.path("/random-animal/**")
11.                      .and()
12.                      .method(HttpMethod.GET)
13.                      .filters(filter ->
14.                          filter.stripPrefix(1)
15.                      )
16.                      .uri("lb://random-
17.                          animal"))
18.              .route("zoo_route",
19.                  route ->
20.                  route.path("/zoo/**")
21.                      .filters(filter ->
22.                          filter.stripPrefix(1)
23.                      )
24.                      .uri("lb://zoo"))
25.              .build();
26.      }
27.  }
```

- Uri

В *uri()* задан микросервис, куда идет перенаправление: *zoo* или *random-animal*. Важно не пытаться прописать тут вложенные пути — не сработает. Только имя микросервиса (под которым он зарегистрирован в Eureka). Префикс *lb* говорит о том, что обращение к микросервису должно проходить через балансировщик нагрузки — то есть заодно еще автоматически будет принято решение о том, к какому именно экземпляру микросервиса обратиться.

- Predicate

Здесь могут быть заданы любые условия, по которым запрос отбирается. Как уже говорилось, мы отбираем запрос по *path()* — задаем шаблон «*url* начинается с того-то». Еще сказано, что это должен быть метод *GET* (просто для демонстрации возможностей). Все эти заданные условия и есть предикат.

- Filter

В фильтре мы отбрасываем вот эту начальную часть *url*, по которой выбрали запрос (*/zoo/* и */random-animal/*). В контроллер микросервиса запрос пойдет без этой части.

Аргумент *1* в методе:

```
1. filter.stripPrefix(1)
```

означает, что именно одну часть отбрасываем. Если бы мы отбирали запрос по двум начальным частям, например *localhost:8080/zoo/part2/...*, то отбросили бы две части, чтобы не тянуть их в контроллер.

## 9 Проверка

Контроллер в микросервисе *Zoo* у нас такой:

```
1. @RestController
2. public class ZooController {
3.     @Autowired
4.     private RandomAnimalClient randomAnimalClient;
5.
6.     @GetMapping("/animals/any")
7.     ResponseEntity<Animal> seeAnyAnimal() {
8.         return randomAnimalClient.random();
9.     }
10. }
```

Обращение к нему через прокси будет таким:





Обращение через прокси

`/zoo` отбрасывается, в контроллер идет `/animals/any`.

А внутри контроллера в `Zoo` обращение ко второму микросервису уже по имени:

```
1.  @Component
2.  public class RandomAnimalClient {
3.      private static final Logger LOGGER = LoggerFactory
4.          .getLogger(RandomAnimalClient.class);
5.      private final RestTemplate loadBalancedTemplate;
6.
7.      RandomAnimalClient(@LoadBalanced RestTemplate
8.          loadBalancedTemplate,
9.          DiscoveryClient
10.         discoveryClient) {
11.         this.loadBalancedTemplate =
12.             loadBalancedTemplate;
13.     }
14.
15.     public ResponseEntity<Animal> random1() {
16.         LOGGER.debug("Sending request for animal
17.             {}");
18.         return
19.             loadBalancedTemplate.getForEntity("http://proxy/random-
20.                 animal/random",
21.                 Animal.class);
22.     }
23. }
```

В запросе

```
1.  http://proxy/random-animal/random
```

использовано имя нашего API Gateway в Eureka — `proxy`.

Обращение сделано через *@LoadBalanced RestTemplate* — это значит, что экземпляров *proxy* вообще может быть несколько (можно запустить API Gateway на нескольких портах — и запрос будет работать).

Часть url */random-animal/* служит для того, чтобы отобрать запрос для направления в микросервис *Random Animal*, а затем отбрасывается. В микросервис идет только */random*.

Соответственно контроллер в микросервисе *Random Animal* принимает запросы, начинающиеся с */random*:

```
1.  @RestController
2.  public class RandomAnimalController {
3.
4.      private final AnimalDao animalDao;
5.
6.      public RandomAnimalController (AnimalDao
animalDao) {
7.          this.animalDao=animalDao;
8.      }
9.
10.     @GetMapping("/random")
11.     public Animal randomAnimal() {
12.         Animal animal=animalDao.random();
13.         System.out.println(animal);
14.         return animal;
15.     }
16. }
```

## 10 Настройка в application.yml

Настройку из кода можно перенести в файл настроек:

```
1.  spring:
2.      cloud:
3.          gateway:
4.              routes:
5.                  - id: random_animal_route
6.                    uri: lb://random-animal
7.                    predicates:
8.                        - Path=/random-animal/**
9.                    filters:
```

```
10.         - StripPrefix=1
11.     - id: zoo_route
12.       uri: lb://zoo
13.       predicates:
14.         - Path=/zoo/**
15.       filters:
16.         - StripPrefix=1
```

## 11 Итоги

Пример можно скачать [здесь](#). Все три части — Zoo, Random Animal и Proxy — можно запускать на любом количестве портов.

[Здесь](#) рассмотрим Spring Cloud Configuration Server.

sysout / 30.12.2020 / Spring, Spring Cloud

---

## Spring Cloud API Gateway: 9 комментариев

---

**Андрей**

16.01.2021 в 05:18

Добрый день,

Спасибо за статью, очень хорошо описано.

Единственное не понял один момент: так что всё таки что мне надо ввести в адресную строку браузера со стороны клиента чтобы вернулось рандомальное животное?. В начале статьи вы пишете что клиент как и любой другой микросервис обращается только к прокси. Но далее указываете запрос

[здесь](#) то есть тут прокси почему то нет. И если такой запрос ввести в адресную строку браузера, то я проверил будет 500-я ошибка, и в стектрейсте прокси-сервиса будет что то такое:

java.net.UnknownHostException: failed to resolve 'DESKTOP-EHPRONG' after 2 queries  
at  
io.netty.resolver.dns.DnsResolveContext.finishResolve(DnsResolveContext.java:1013) ~[netty-resolver-dns-4.1.58.Final.jar:4.1.58.Final]  
Suppressed:  
reactor.core.publisher.FluxOnAssembly\$OnAssemblyException:  
Error has been observed at the following site(s):  
|\_ checkpoint --->  
org.springframework.cloud.gateway.filter.WeightCalculatorWebFilter  
[DefaultWebFilterChain]  
|\_ checkpoint --->  
org.springframework.boot.actuate.metrics.web.reactive.server.Metrics  
WebFilter [DefaultWebFilterChain]  
|\_ checkpoint ---> HTTP GET «/zoo/animals/any»  
[ExceptionHandlerWebHandler]  
Подскажите, что я делаю не правильно?

Ответить

---

**sysout** 👤

16.01.2021 в 10:34

Прокси в примере — это то, что запущено на порту 8080. По имени *проху* мы можем обращаться к нему из других микросервисов (только благодаря Eureka), но не из браузера. (Zoo в примере запущен на *8081*, из браузера обращение <http://localhost:8080/zoo/animals/any> есть обращение к прокси, а не к Zoo).

Перепроверка примера ошибок не показала. Но убедитесь, что запускаете Eureka ДО запуска остальных микросервисов. Чтобы микросервис смог зарегистрироваться в Eureka, Eureka должна быть предварительно запущена. Тогда обращение по имени <http://proxy/random-animal/random> (проху вместо

localhost:8080) сработает, а иначе нет. Посмотрите [пример](#) по Eureka.

Ответить

---

**Андрей**

16.01.2021 в 14:43

Спасибо, да работает, видимо чего то не доглядел в первый раз

Ответить

---

**sysout** 

16.01.2021 в 10:38

По адресу Еврики <http://localhost:8761/> можно проверить список зарегистрированных микросервисов.

Ответить

---

**Артём**

05.04.2021 в 10:02

Добрый день!

Спасибо за статью, очень ценный и полезный материал.

Один вопрос, как запретить прямой доступ к сервисам zoo и random?

А, то получается, что весь смысл PROXY сервиса теряется из-за того, что можно напрямую обращаться к сервисам zoo и random, указав полный путь к сервису, например, вот так:

[REDACTED], или вот так:

[REDACTED]

Ответить

---

**sysout** 👤

05.04.2021 в 13:17

Настройками сети — я так понимаю, это уж дело сисадмина.

Ответить

---

**Артём**

05.04.2021 в 14:59

Zuul Proxy, насколько я помню из коробки закрывал доступ к микросервисам и достигаться до них не было возможно. Если найду решение, обязательно поделюсь. Еще раз спасибо за статью!

Ответить

---

**Александр**

31.10.2022 в 15:53

меня тоже интересует этот вопрос -как закрыть доступ к остальным сервисам  
Получилось найти какое-то решение?

Ответить

---

**javastream**

24.05.2021 в 18:45

Одно из самых понятных объяснений, которые я нашел в сети.  
Спасибо вам огромное!

Из application.yml убрал эту запись, не хотело взлетать:

eureka:

client:

healthcheck:

enabled: true

Ответить

---

## Добавить комментарий

Ваш адрес email не будет опубликован. Обязательные поля помечены \*

КОММЕНТАРИЙ \*

ИМЯ \*

EMAIL \*

САЙТ

ОТПРАВИТЬ КОММЕНТАРИЙ

---

НАЗАД

## Микросервисы: Eureka и client-side Load Balancing

---

ДАЛЕЕ

## Spring Cloud Configuration Server

---

*Прошу прощения: на комментарии временно не отвечаю.*

### СВЕЖИЕ КОММЕНТАРИИ

- sysout к записи [Отношение OneToMany в Hibernate и Spring](#)
- Сергей Акопов к записи [Отношение OneToMany в Hibernate и Spring](#)
- Дмитрий к записи [N+1 проблема в Hibernate](#)
- Дмитрий к записи [Spring Custom Login Form](#)
- Руслан к записи [Отказоустойчивость микросервисов: шаблон](#)



## Circuit Breaker

- Николай к записи [Spring Custom Login Form](#)
- Денис к записи [@Primary, @Qualifier и внедрение списка](#)
- Михаил к записи [Работа с IoC-контейнером в Spring](#)
- Арсен к записи [Как работает Flush в Hibernate](#)
- Сергей к записи [Spring Data JDBC: CrudRepository и Query methods](#)
- Yustas к записи [Пример приложения с JWT-токеном](#)
- sysout к записи [Пример приложения с JWT-токеном](#)
- Михаил к записи [Пример приложения с JWT-токеном](#)
- Андрей к записи [Преобразование Entity в DTO с помощью ModelMapper](#)

---

## Core Java

---

### Spring



---

## Java Libraries

---

## Git

---