GeekBrains

Разработка интернет-магазина на Spring Framework

Spring Cloud

Spring Cloud. Spring Cloud Config. Spring Cloud Netflix.

Оглавление

Spring Cloud

Введение

Spring Cloud основывается на Spring Boot и предоставляет множество полезных библиотек. Использование модели PaaS избавляет разработчика (и эксплуатанта) от необходимости устанавливать, настраивать, поддерживать аппаратную часть, операционную систему и платформу для развертывания приложения, в нашем случае — Java Virtual Machine. Этот подход позволяет ограничиться отправкой кода приложения в удаленный репозиторий. Унификация и широкое распространение данной модели дает возможность ценой незначительных усилий создавать географически распределенные отказоустойчивые сервисы.

Вы можете воспользоваться базовым поведением по умолчанию, чтобы начать работу быстро. А затем, когда потребуется, настроить или расширить конфигурацию, чтобы создать собственное решение.

Spring Cloud абстрагирует соединение к облачному сервису и дает возможность устанавливать одни и те же приложения на разных облачных платформах минимальными усилиями.

Spring Cloud для типичных сценариев использования в облачной инфраструктуре предоставляет разработчику такие возможности, как:

- конфигурация распределенных версий;
- регистрация и открытие службы маршрутизации;
- служебные вызовы;
- балансировка нагрузки;
- автоматические выключатели;
- глобальные защелки;
- кластеры;
- распределенная передача сообщений.

Spring Cloud Config

Spring Cloud Config позволяет отделить конфигурацию приложения от него самого в месте развертывания. Предоставление конфигурации превращается в конфигурируемый сервис поверх стандартных протоколов HTTP и Git.

Создадим модуль конфигурационного сервера на основе Spring Boot приложения:

```
@EnableConfigServer
@SpringBootApplication
public class ConfigServerApplication {

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

Аннотация **@EnableConfigServer** делает все дело, превращая простейшее Spring Boot приложение в сервер раздачи конфигураций. Под капотом **@EnableConfigServer** скрывается то, как зависимость подтягивает **Tomcat** (с возможностью выбора другого контейнера) и сервлет, отдающий конфигурации по HTTP-протоколу.

Нам потребуются следующие зависимости:

```
<dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

В свойствах модуля необходимо указать версию Spring Cloud:

```
<properties>
            ...
      <spring-cloud.version>Dalston.SR3</spring-cloud.version>
</properties>
```

Файл свойств модуля (**application.properties**):

```
# по умолчанию клиент обращается 8888 порт
server.port=8888

# локальный вариант хранения (без Git-репозитория)
# spring.profiles.active=native
# путь к ней внутри classpath.
# реально конфиги будут деплоиться в облако вместе с сервисом
# просто и понятно, но оборотной стороной будет
# потребность в редеплое приложения при изменениях настроек
# spring.cloud.config.server.native.search-locations=classpath:/config

# используем Git, в облаке так удобнее, не требуется редеплой, только рестарт
spring.cloud.config.server.git.uri=${GIT_URI}
spring.cloud.config.server.git.search-paths=${GIT_SEARCH_PATHS}
```

В Git-репозитории (локальном или удаленном) необходимо разместить (**add**) и закоммитить (**commit**) файл конфигурации:

```
# указываем конфигурационный параметр приложения, в данном случае — строку
config.name=Config Server App From Git application.properties
```

Запускаем сервер конфигураций.

Клиент Spring Cloud Configuration

Создадим еще одно простейшее Spring Boot приложение. Оно будет получать конфигурационный параметр **config.name** от конфигурационного сервера **Spring Cloud Configuration Server**:

```
@EnableAutoConfiguration
@SpringBootApplication
@RestController
public class ConfigClientApplication {

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

	    // переменная, значение которой будет
	    // подниматься с сервера конфигурирования
	@Value("${config.name}")
	String name = "World";

	@RequestMapping("/")
	public String home() {
		return "Hello " + name;
	}
}
```

Нам потребуются следующие зависимости:

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

	<dependency>
		<groupId>org.springframework.cloud</groupId>
		<artifactId>spring-cloud-starter-config</artifactId>
	</dependency>
```

По умолчанию, если не указано иного в **application.properties**, клиент конфигурирования будет пытаться найти соответствующий сервер по адресу http://localhost: 8888.

Проверим результат:

```
$ curl localhost:8080
Hello Config Server App From Git application.properties
```

Сервер конфигурирования может поддерживать множество **.properties**-файлов для разных приложений:

Добавим .properties-файл для «именованного» приложения: создадим и закоммитим в репозитории файл **alabama.properties**:

```
config.name=Config Server App From Git alabama.properties
```

А также укажем имя приложения в .properties-файле клиентского приложения (**application.properties**):

```
# имя приложения
spring.application.name=alabama

# адрес конфигурационного сервера, по умолчанию localhost
spring.cloud.config.uri=https://config.service.uri
```

файл **bootstrap.properties**

Теперь при запуске клиентское приложение получит набор параметров из файла **alabama.properties**, находящегося в Git-репозитории.

```
$ curl localhost:8080
Hello Config Serer App From Git alabama.properties
```

Если попытка найти в репозитории **.properties** соответствующее имя будет неудачной, будут выданы настройки из файла **application.properties**:

```
$ curl localhost:80
Hello Config Serer App From Git application.properties
```

На клиенте можно указать конкретный URL конфигурационного сервера, отличный от установленного по умолчанию:

```
spring.cloud.config.uri=http://localhost:8888
```

Можно обеспечить безопасность сервера, раздающего конфигурации, с помощью стандартных средств библиотеки Spring Security на стороне сервера конфигурирования.

Spring Cloud Netflix

Spring Cloud Netflix — это набор компонентов, разработанный компанией Netflix для поддержки ее инфраструктуры, которая развернута поверх распределенной сети дата-центров облачного провайдера Amazon. В дальнейшем был открыт исходный код некоторых компонентов, используемых компанией. Данный проект прошел «проверку боем» в реальных сервисах от Netflix. Команда Spring решила использовать готовые наработки Netflix — результат этой адаптации известен как Spring Cloud Netflix.

Каждый из компонентов Netflix является одной из реализаций паттернов построения распределенных систем и служит для решения конкретной задачи, часто во взаимодействии с другими компонентами.

Паттерны, реализуемые Netflix:

- **Service Discovery (Eureka)** — экземпляры Eureka могут быть зарегистрированы, а клиенты могут обнаруживать их с использованием Spring. Сервер может быть создан с декларативной конфигурацией;
- паттерны **маршрутизатор** и **фильтр (Zuul)** — простой декларативный подход к фильтрации трафика и созданию прокси;
- **Circuit Breaker (Hystrix)** — при возникновении в системе неработающего сервиса блокирует распространение отказа, перенаправляя запросы на заранее подготовленную заглушку;
- декларативный REST-клиент (Feign) — декларативное описание REST-клиента и балансировка нагрузки на стороне клиента.

Маршрутизатор и фильтр (Zuul)

Маршрутизация

Допустим, путь **/** сопоставлен с вашим веб-приложением, **/api/users** — с микросервисом пользователей, а **/api/shop** — с микросервисом магазина.

Zuul — это маршрутизатор на основе JVM. Может использоваться для:

- маршрутизации;
- тестирования и переходных сценариев;
- управления трафиком.

Рассмотрим маршрутизацию более подробно. Подключение — стандартное:

```
<dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
```

Добавьте аннотацию **@EnableZuulProxy** к основному классу Spring Boot приложения маршрутизатора.

```
@EnableZuulProxy
@SpringBootApplication
public class ZuulApplication {
...
```

Это перенаправит весь входящий трафик приложения на прокси Zuul.

Далее настраиваем «таблицу маршрутизации» — доступно несколько вариантов:

1) файл **application.properties**:

```
zuul.routes.discus.path=/discus/**
zuul.routes.discus.url=http://discus.example.com:8081/discus
```

2) или **.yml**:

```
zuul:
  routes:
    news: # имя конечной точки
      path: /news/**
      url: http://news-service.example.com:8082/
   admin:
      path: /admin/**
      url: http://admin-service.example.com:8083/
```

3) программно, файл **ApplicationConfiguration.java**:

```
@Bean
public PatternServiceRouteMapper serviceRouteMapper() {
    return new PatternServiceRouteMapper(
        "(?<name>^.+)-(?<version>v.+$)",
        "${version}/${name}");
}
```

Используя Zuul, можно постепенно дробить монолитное приложение на отдельные микросервисы:

```
zuul:
 routes:
   first:
     path: /first/**
     url: http://first-service.example.com
   second:
     path: /second/**
     url: forward:/second
   legacy:
     path: /**
     url: http://legacy-app.example.com
```

Здесь мы оставляем унаследованному приложению все запросы, которые не соответствуют одному из других шаблонов. Запросы к **/first/**** были извлечены в новую службу с внешним URL-адресом. Запросы к **/second/**** перенаправляются в локальное приложение (несущее прокси), но по модифицированному пути его можно обработать локально, используя **@RequestMapping("/second")**. Запросы, не подпадающие ни под одно из правил, не игнорируются — они просто не обрабатываются прокси-сервером и передаются на обработку локальному приложению без изменений.

Фильтры

Фильтры способны выполнять ряд действий во время маршрутизации HTTP-запросов и ответов.

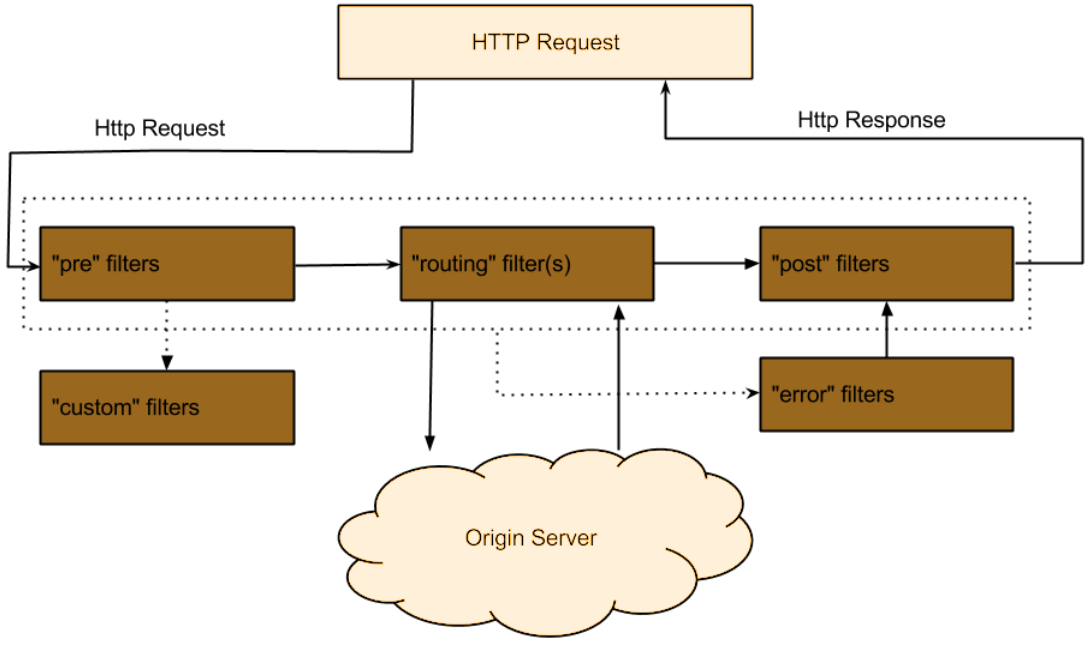
Характеристики фильтров

- **Тип** — чаще всего определяет этап во время потока маршрутизации, на котором применяется фильтр. См. метод **filterType()** ниже.
- **Execution Order** — применяется в домене типа, определяет порядок выполнения для нескольких фильтров. См. метод **filterOrder()** ниже.
- **Критерии** — условия, необходимые для выполнения фильтра. См. метод **shouldFilter()** ниже.
- **Действие** — действие, которое должно выполняться, если соблюдается критерий. Zuul обеспечивает структуру для динамического чтения, компиляции и запуска этих фильтров. См. метод **run()** в примере ниже.

Фильтры не взаимодействуют друг с другом напрямую — вместо этого они делят состояние через **RequestContext**, который уникален для каждого запроса.

Существует несколько стандартных типов фильтров, которые соответствуют типичному жизненному циклу запроса.

- **PRE-фильтры** выполняются перед маршрутизацией. Применяются для проверки подлинности запроса, выбора целевого сервера или регистрации информации об отладке.
- **ROUTING-фильтры** обрабатывают запрос на происхождение. Здесь создается HTTP-запрос источника, который отправляется с использованием Apache HttpClient.
- **POST-фильтры** выполняются после того, как запрос был направлен целевому серверу. Обычно POST-фильтры занимаются добавлением HTTP-заголовков в ответ, сбором статистики, а также передачей ответа от источника клиенту.
- **ERROR-фильтры** выполняются, когда возникает ошибка во время прохождения запроса.



Жизненный цикл запроса

источник: https://github.com/Netflix/zuul/wiki/How-it-Works

Пример:

```
@Component
public class CustomZuulFilter extends ZuulFilter {
    @Override
    public Object run() {
        RequestContext context = RequestContext.getCurrentContext();
        context.addZuulRequestHeader("X-zuul-filter-example", "test value");
        return null;
    }

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

    @Override
    public String filterType() {
        // PRE-Фильтры выполняются перед маршрутизацией.
        // тип фильтра PRE,
        return FilterConstants.PRE_TYPE;
    }

    @Override
    public int filterOrder() {
        // =1 порядок запуска фильтра
        return FilterConstants.DEBUG_FILTER_ORDER ;
    }
}
```

В данном примере фильтр добавляет специфический заголовок к запросу.

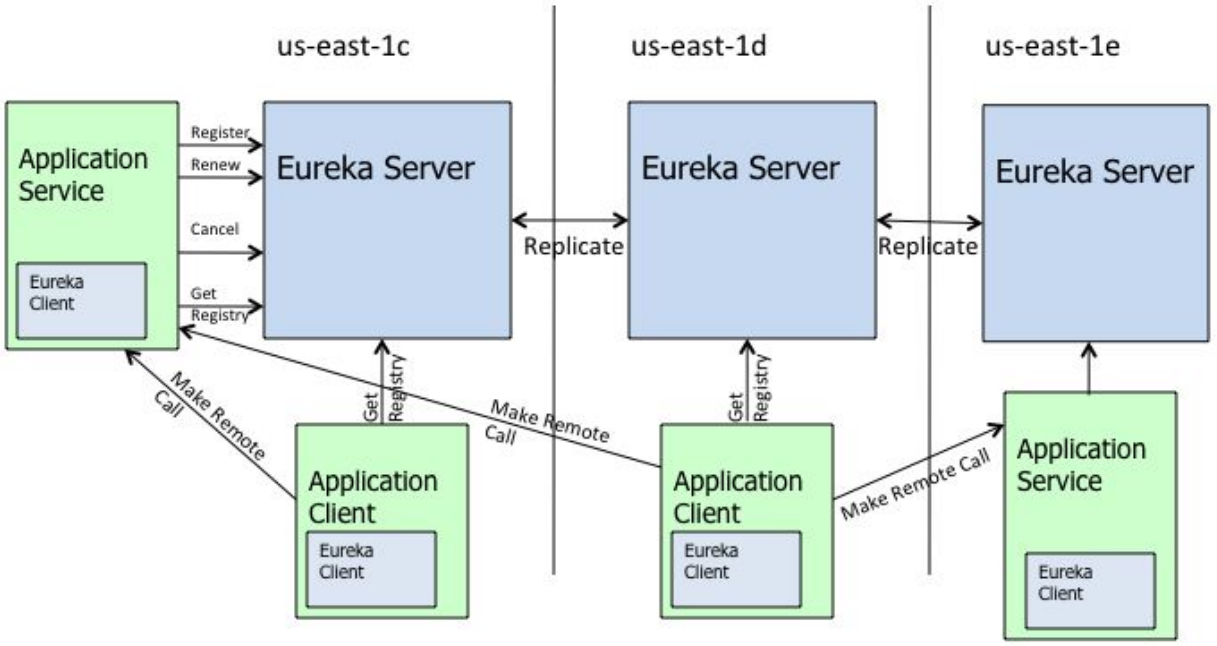
Service Discovery

Еще один широко известный паттерн для распределенных систем. Service Discovery позволяет автоматически определять URI доступных (функционирующих) инстансов приложений, которые могут динамически изменяться из-за масштабирования, падений и обновлений.

Рассмотрим пример: в системе есть некоторое (иногда достаточно большое) количество взаимосвязанных приложений (микросервисов). Без использования Service Discovery значительное число взаимосвязанных приложений приведет к образованию многих жестких взаимосвязей между ресурсами. При изменении адреса одного из приложений придется вносить правки во все зависимые от него приложения и перезапускать их, что крайне нежелательно. Кроме того, в реальном мире приложения запускаются и завершаются, падают, но система в целом должна сохранять работоспособность.

Для решения этих проблем служит паттерн Service Discovery. В системе есть распределенный сервис (несколько взаимосвязанных узлов, выполняющих одну общую функцию), на котором приложение регистрируется при старте, сообщая сведения о себе: имя, хост, порт и прочие параметры. Узлы распределенного сервиса обмениваются этой информацией. Во время работы приложение отправляет на сервис регистрации хартбит-сообщения, свидетельствующие о его жизнеспособности. Если хартбит-сообщения не поступают в течение некоторого времени, информация о данном инстансе удаляется. Клиентская часть Service Discovery, выполняющаяся в рамках приложения, также участвует в репликации реестра приложений как клиент. Поэтому внутри одного приложения можно использовать имена других при обращении к их ресурсам, а не их фактический адрес.

Например, в системе есть сервис с именем **config**, доступный по адресу https://config-app.hpherokuapp.com/…. Все остальные компоненты системы для указания на этот ресурс внутри приложения используют ссылку вида https://config/… Клиентская часть Service Discovery должна будет получить реальный URI **config-**сервиса. Далее мы можем перенести **config-**сервис на другой провайдер (возможно, поменяется его адрес) или захотим запустить несколько инстансов этого сервиса. Когда в системе появятся новые инстансы, они будут доступны запрашиваемым приложениям, а завершенные сервисы перестают быть доступными. Самое главное, что все это происходит с «серверной стороны» Service Discovery и никак **не влияет на работу остальных сервисов системы,** пользующихся ресурсами **config** сервиса — ведь они находятся с «клиентской стороны» и изолированы от перипетий реальной жизни обслуживающих их сервисов.



Eureka — пример **client-side service discovery** паттерна. Это означает, что клиент должен запросить адреса доступных инстансов и осуществлять балансировку между ними самостоятельно.

Чтобы превратить Spring Boot приложение в Registry server, достаточно добавить зависимость:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

А еще аннотацию **@EnableEurekaServer** и указать соседние узлы репликации. Для вырожденного случая, состоящего из одного узла, указываем себя.

```
# app name
spring.application.name=registry
# Client Side in Standalone Mode. Не использовать в реальных развертываниях
eureka.instance.hostname=localhost
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
# список соседей. в standalone указываем, фактически, самое себя.
eureka.client.service-url.default-zone=http://${eureka.instance.hostname}:${serv
er.port}/eureka/
```

*файл **application.properties***

На стороне клиентов — зависимость:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

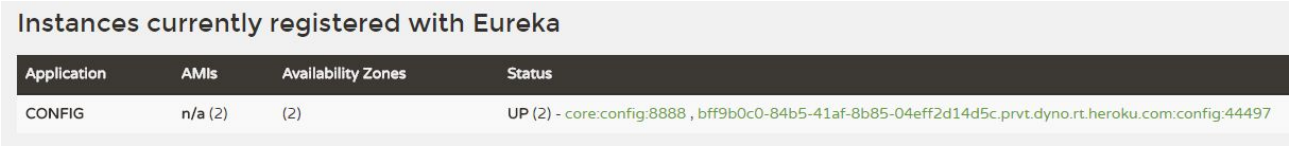
```
@EnableEurekaServer
@SpringBootApplication
public class RegistryApplication {
      public static void main(String[] args) {
            SpringApplication.run(RegistryApplication.class, args);
      }
}
```

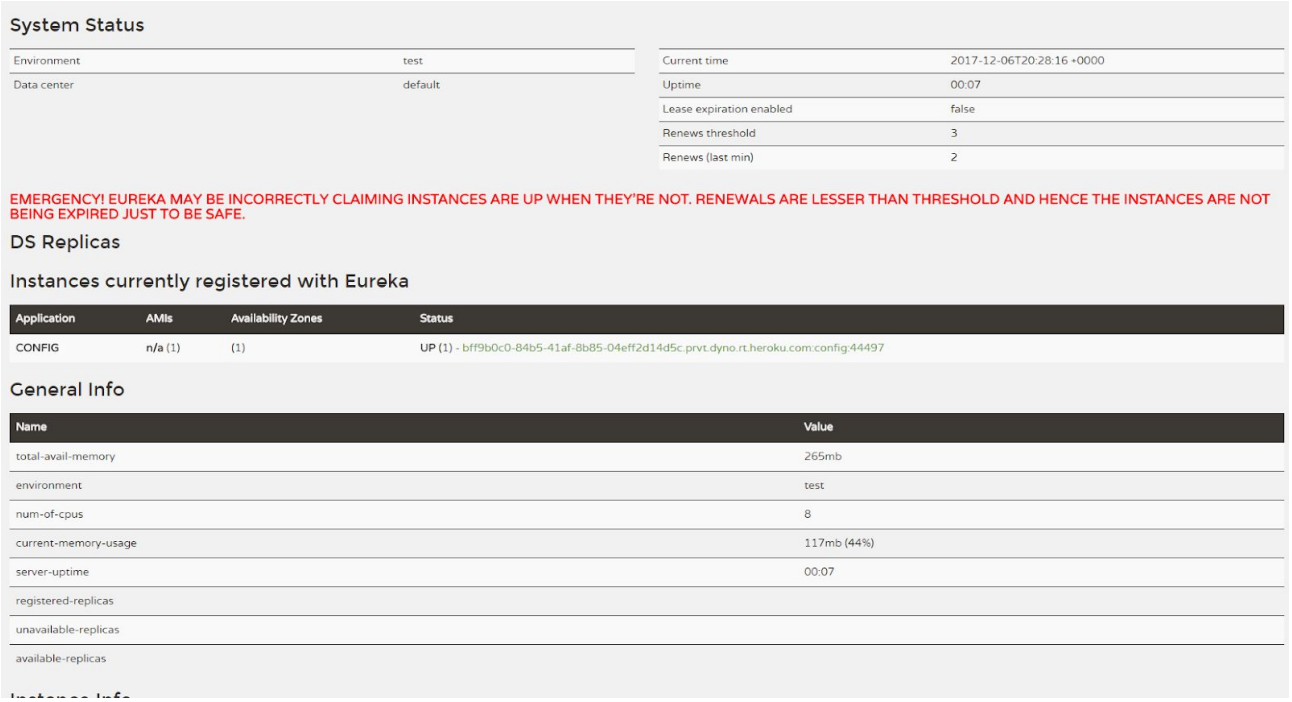
Аннотация **@EnableDiscoveryClient**, имя приложения (**serviceId**) и список реальных URI-серверов Service Discovery:

```
# имя приложения
spring.application.name=gateway
# указываем путь до Discovery Server
eureka.client.service-url.defaultZone=http://first.discovery-app.com,
http://second.discovery-app.com, ...
```

*файл **bootstrap.properties***

Теперь инстанс приложения при старте будет регистрироваться на сервисе Eureka, предоставляя метаданные (хост, порт и прочее). Eureka принимает хартбит-сообщения, и если их нет в течение сконфигурированного времени, инстанс будет удален из реестра. Кроме того, Eureka предоставляет дашборд, на котором видны зарегистрированные приложения с количеством инстансов и другая техническая информация. Ниже мы видим две копии микросервиса **CONFIG**:





Здесь — прочая техническая информация, в том числе предупреждение о вырожденности standalone-конфигурации.

Использование на стороне клиента в конфигурационных файлах

Другие компоненты Netflix автоматически используют возможности, предоставляемые клиентской частью Service Discovery.

Например, в процессе конфигурирования адреса, чтобы забрать конфигурацию с сервера при использовании модели **registry first** (сначала регистрируемся, потом забираем конфигурацию):

```
# забрать конфиг из облака — URI указываем, учитывая 'registry first'
spring.cloud.config.uri=https://config
# registry first
spring.cloud.config.discovery.enabled=true
spring.cloud.config.discovery.serviceId=config
```

*файл **bootstrap.properties***

Или при конфигурировании роутера Zuul:

```
zuul:
  routes:
    news-service:
        path: /news/**
        serviceId: news-service
        ### нет необходимости указывать url: http://news-service.example.com
```

файл конфигурации роутера ***gateway.yml***

Здесь Zuul будет маршрутизировать по символическому имени сервиса, а не по физическому адресу одного из инстансов микросервиса.

Использование на стороне клиента в коде

Ручное получение URI конкретного сервиса:

```
    @Autowired
    DiscoveryClient client;

    public URI getServiceUri(String serviceId) {
        List<ServiceInstance> instances = client.getInstances(serviceId);
        if (instances != null && instances.size() > 0)
            return instances.get(0).getUri();
        else
            return null;
    }
```

Практическое задание

Создать отдельное микросервисное веб-приложение, в котором есть:

- Сервис, отвечающий за работу с продуктами: получение списка продуктов из базы, добавление/удаление продуктов, но при этом сервис не имеет фронтенда.
- Сервис для отображения этих продуктов на веб-странице с формой для добавления товара, и кнопками удаления товаров.

То есть один сервис только работает с продуктами, а другой может их отображать. Реализовать это приложение с использованием Spring Cloud.

Дополнительные материалы

1. Netflix Zuul.
2. Пример конфигурирования.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Официальная документация Spring.
2. Документация от Netflix.