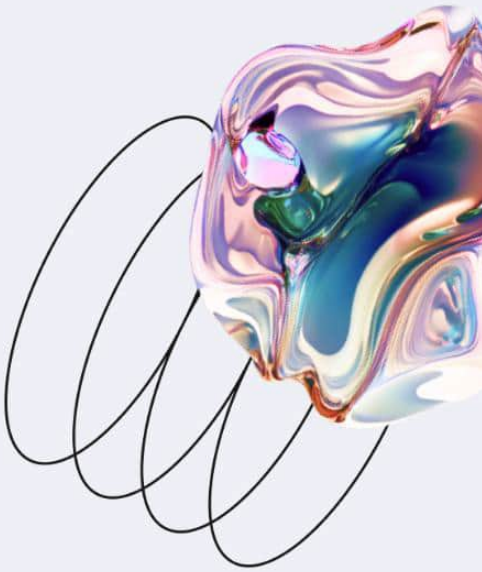
GeekBrains



Spring Cloud. Микросервисная архитектура

Фреймворк Spring

Оглавление

Введение

Привет, ребята! Сегодня мы погрузимся в удивительный мир Spring Cloud и микросервисов. Помните наш последний урок про Spring AOP? Он был о том, как элегантно решать задачи аспектно-ориентированного программирования. Но мир IT не стоит на месте, и сегодня мы поговорим о следующем этапе в эволюции разработки – переходе от монолитных приложений к микросервисам.

Возможно, вы слышали слово “монолит”, которое описывает приложение, состоящее из одного гигантского блока кода. По сути, все компоненты, функционал и слои в таком приложении находятся в одном месте. Именно так работали многие проекты ещё несколько лет назад. Но прогресс не стоит на месте, и появилась потребность в более гибких, масштабируемых и удобных в обслуживании системах. И тут на помощь приходят микросервисы!

Давайте представим, что вы строите город. Если бы город был монолитом, то это как если бы все горожане жили бы в одном огромном доме. Один кран сломался? Весь дом без воды. Хотите добавить несколько новых комнат? Весь дом нужно перестраивать!

А теперь представьте, что вместо одного большого дома у нас есть множество маленьких домиков — каждый со своей системой отопления, водоснабжения и т. д. Это и есть микросервисы.

Spring Cloud предоставляет нам инструменты для создания таких систем на основе микросервисов, но давайте не спешить и узнаем о нем побольше в следующих разделах.

А теперь представьте, что вместо одного большого дома у нас есть множество маленьких домиков — каждый со своей системой отопления, водоснабжения и т. д. Это и есть микросервисы.

Теперь к преимуществам:

- **Масштабируемость:** Вернемся к нашим домикам. Если в одном районе стало много жителей, можно легко построить еще один такой же домик рядом. В мире микросервисов, если одна часть системы испытывает нагрузку, мы можем просто добавить еще один такой же микросервис, чтобы поделить эту нагрузку.

- **Устойчивость:** Вспомните про кран. Если в одном домике сломается кран, это не затронет другие домики. Так же и с микросервисами: если один сервис “падает”, другие продолжают работать.

- **Независимость компонентов:** Каждый домик может быть построен из разных материалов, в разном стиле и даже иметь свою собственную систему безопасности. Так и с микросервисами: каждый может быть написан на разных языках программирования, иметь свою базу данных и обновляться независимо от других.

По сути, микросервисы предлагают нам гибкость. Вместо того чтобы зависеть от одного “огромного дома”, мы управляем множеством маленьких, что делает жизнь намного проще и удобнее.

Термины, используемые в лекции

Микросервис — Это подход к разработке программного обеспечения, при котором приложение состоит из мелких, независимых компонентов, работающих вместе.

Eureka Server — Сервис от Netflix, интегрированный в Spring Cloud, который предоставляет сервис обнаружения, где микросервисы могут регистрироваться и находить друг друга.

Zuul Proxy — Используется в Spring Cloud в качестве маршрутизатора и прокси-сервера между клиентом и микросервисами.

Spring Cloud Config Server — Централизованный сервер для управления конфигурациями микросервисов через все окружения.

Load Balancing — Распределение входящего трафика между множеством серверов или микросервисов.

Feign Client — Декларативный web-клиент от Netflix, интегрированный в Spring Cloud, который упрощает написание кода для взаимодействия с другими микросервисами.

Hystrix — Библиотека от Netflix, интегрированная в Spring Cloud, предоставляющая латентное и отказоустойчивое выполнение.

Circuit Breaker Pattern — Подход, который позволяет системе продолжать работу, даже когда часть необходимых сервисов временно недоступна.

Spring Cloud Bus — Механизм, который соединяет различные узлы микросервисного комплекса с легкостью и доступностью облачного средства.

Service Mesh — Подход к управлению и контролю трафика между микросервисами.

API Gateway — Сервер, который является точкой входа в микросервисную архитектуру и маршрутизирует запросы к соответствующим службам.

Service Registry & Discovery — Механизм для автоматической регистрации и обнаружения микросервисов в среде.

Distributed Tracing — Подход к мониторингу запросов в микросервисной архитектуре, позволяющий отслеживать выполнение запроса через все микросервисы.

Spring Cloud Stream — Фреймворк для создания приложений для обработки потоковых данных с использованием Spring Boot.

Spring Cloud Data Flow — Инструмент для оркестровки приложений потоковых данных на основе Spring Cloud Stream

Популярность микросервисов

Хм, представьте, что у вас есть пазл из тысячи кусочков. Вы точно знаете, что каждый кусочек имеет своё место, и в случае необходимости вы можете заменить один кусочек, не разбирая всю картину. Таким образом, микросервисы - это как кусочки этого пазла, позволяющие создавать гибкие и адаптивные системы. Но давайте поговорим, почему именно микросервисы стали такими популярными.

1. **Быстрый тайм ту маркет:** Крупные компании и стартапы хотят запустить новые продукты и функции как можно быстрее. С микросервисами, они могут разрабатывать, тестировать и выпускать отдельные части системы без необходимости ждать остальную часть. Это как если бы вы улучшали одну комнату в доме, не перестраивая весь дом.

2. **Талантливые команды:** В мире разработки всё больше талантливых специалистов с разными навыками. Микросервисы позволяют командам использовать разные технологии и языки программирования для разных сервисов. Это как если бы на стройке каждая бригада строила свою часть дома, используя свои предпочтения в материалах и инструментах.

3. **Масштабирование при минимальных затратах:** Крупные компании сталкиваются с проблемой обработки большого количества данных и запросов. Микросервисы позволяют масштабировать только те части системы, которые это действительно нужно, сэкономив ресурсы и деньги.

4. **Разделение ответственности:** В крупных организациях, где много команд, микросервисы обеспечивают четкое разделение ответственности. Каждая команда может заниматься своим микросервисом, не беспокоясь о том, что ее изменения повлияют на работу других команд.

5. **Инновации и эксперименты:** Стартапы, известные своими инновациями, любят экспериментировать. С микросервисами они могут быстро создавать новые функции, экспериментировать с ними, и если что-то идет не так, просто отключить или заменить этот микросервис без риска для всей системы.

В итоге, микросервисы предоставляют гибкость, масштабируемость и возможность быстрого внедрения изменений, что делает их идеальным выбором для современных крупных компаний и динамичных стартапов.

Отказоустойчивость и самовосстановление

Вспомните ощущение, когда вы играете в домино, и одна фишка случайно падает, опрокидывая остальные за собой. В мире микросервисов, если один сервис “падает”, это может вызвать цепную реакцию сбоев по всей системе. Однако, в отличие от домино, нам не нужно, чтобы наши сервисы ломали друг друга!

Итак, почему в микросервисной архитектуре так важна отказоустойчивость и быстрое восстановление?

1. **Множество сервисов:** Когда у вас много маленьких сервисов, вероятность того, что один из них в какой-то момент “упадет”, гораздо выше, чем если бы у вас было одно монолитное приложение. Поэтому нам нужно быть готовыми к таким ситуациям.

2. **Независимость:** Хотя каждый микросервис независим, они часто взаимодействуют друг с другом. Если один сервис не работает, это может затруднить работу других. Поэтому быстрое восстановление после сбоев критично для общей производительности системы.

3. **Пользовательский опыт:** Никто не любит, когда сайт или приложение не работает. Быстрое восстановление сервиса обеспечивает лучший пользовательский опыт и удерживает клиентов.

И тут на помощь приходит Spring Cloud! Этот инструмент предоставляет ряд решений, специально разработанных для микросервисной архитектуры, чтобы сделать наши приложения более устойчивыми к сбоям. Он предлагает такие функции, как автоматическое масштабирование, балансировка нагрузки, обнаружение сервисов и даже обработку сбоев. Таким образом, даже если один из наших микросервисов “падает”, Spring Cloud поможет быстро восстановить его или перераспределить трафик, чтобы минимизировать воздействие на пользователей.

В общем, живя в мире микросервисов, мы должны быть всегда готовы к неожиданным проблемам. Но с правильными инструментами, такими как Spring Cloud, мы можем быть уверены в том, что наша система будет работать плавно и эффективно, даже в самых сложных условиях.

Spring Cloud

Представьте себе швейцарский армейский нож для разработчика микросервисов. Ну или, если не любите такие аналогии, представьте себе магический инструментарий, который делает жизнь разработчика гораздо проще. Вот это и есть Spring Cloud!

Spring Cloud — это не просто единственный инструмент или библиотека. Это целый набор инструментов и решений, предоставляемых командой Spring, чтобы упростить разработку, управление и масштабирование микросервисных приложений. Да, вы можете думать о нем как о наборе суперсил, которые позволяют вашим микросервисам взаимодействовать друг с другом, восстанавливаться после сбоев, балансировать нагрузку и делать множество других вещей без лишних головных болей.

Так что же делает Spring Cloud таким особенным?

1. **Упрощение разработки:** Когда речь идет о создании микросервисов, есть много повторяющихся задач и проблем, с которыми сталкиваются разработчики. Spring Cloud предлагает готовые решения для большинства из них, позволяя вам сконцентрироваться на бизнес-логике, а не на “изобретении велосипеда”.

2. **Бесшовное взаимодействие:** В мире микросервисов сервисы постоянно общаются друг с другом. Spring Cloud предоставляет механизмы для упрощения этого взаимодействия, будь то обнаружение сервисов, конфигурация или обработка сбоев.

3. **Отказоустойчивость:** Как мы уже говорили ранее, быстрое восстановление после сбоев критически важно в микросервисной архитектуре. Spring Cloud включает в себя решения для мониторинга, обнаружения проблем и автоматического восстановления сервисов.

В общем, Spring Cloud — это как мост, соединяющий все микросервисы в единую, управляемую и надежную систему. С его помощью разработчики могут спать спокойно, зная, что многие сложные аспекты управления микросервисами уже решены за них.

Компоненты Spring Cloud

Spring Cloud, как вы уже, наверное, догадались, не просто один инструмент. Это целый “набор инструментов” для разработчиков. Представьте себе конструктор LEGO: у вас есть множество разных деталей, каждая из которых имеет свое назначение. Но когда вы собираете их вместе, вы можете создать что-то действительно крутое!

Давайте рассмотрим некоторые ключевые компоненты Spring Cloud:

1. **Spring Cloud Config:** Этот компонент управляет внешними настройками для приложений во всех окружениях. Представьте, что у вас есть универсальный пульт управления для всех настроек ваших микросервисов!
2. **Spring Cloud Netflix:** На самом деле это набор подпроектов, вдохновленных Netflix OSS. К нему относятся:
    - **Eureka** для обнаружения сервисов.
    - **Hystrix** для контроля над временем ожидания между микросервисами и обработки сбоев.
    - **Zuul** для маршрутизации и фильтрации на уровне API.
3. **Spring Cloud Gateway:** Это более современный маршрутизатор на основе Spring, который можно использовать в качестве альтернативы Zuul.
4. **Spring Cloud Bus:** Этот компонент использует легковесные сообщения (часто с помощью RabbitMQ или Kafka) для обмена информацией между различными частями системы.

Так как же все эти компоненты могут работать вместе?

Представьте сценарий, где у вас есть несколько микросервисов. Вместо того, чтобы каждый из них имел свои отдельные настройки, вы используете **Spring Cloud Config** для централизованного управления конфигурацией. Теперь, когда один из микросервисов хочет общаться с другим, он использует **Eureka** для обнаружения этого сервиса. Затем, чтобы гарантировать надежное взаимодействие, он применяет **Hystrix** для обработки сбоев и контроля времени ожидания. Весь входящий трафик может проходить через **Zuul** или **Spring Cloud Gateway**, которые решают, к какому микросервису направить запрос. И в случае любых изменений в системе, **Spring Cloud Bus** может распространять эти изменения между всеми компонентами.

Таким образом, все эти “кусочки” Spring Cloud соединяются вместе, создавая надежную, масштабируемую и устойчивую микросервисную архитектуру.

Eureka

Давайте начнем с вопроса: когда вы хотите позвонить своему другу, что вы делаете? Вероятно, вы открываете свои контакты и ищете его имя, чтобы узнать номер телефона, правильно? В мире микросервисов есть что-то похожее, и это называется “обнаружение сервисов”. Eureka от Spring Cloud Netflix — это одно из самых популярных решений для этой задачи.

Что такое Eureka?

Eureka — это система обнаружения сервисов, которую можно представить как “телефонную книгу” для ваших микросервисов. Когда микросервис запускается, он регистрируется в Eureka, сообщая ей: “Привет, я здесь, и вот мой адрес!”. Теперь, когда другой микросервис хочет общаться с ним, он просто спрашивает Eureka: “Где мой друг, микросервис X?” Eureka, в свою очередь, предоставляет адрес нужного микросервиса, и взаимодействие может начаться.

Зачем это нужно?

1. **Динамичность:** В современных облачных средах микросервисы могут постоянно перезапускаться, менять свое местоположение и масштабироваться. Без системы обнаружения сервисов было бы сложно отслеживать их все.

2. **Балансировка нагрузки:** Eureka не только помогает обнаруживать сервисы, но и может управлять трафиком, распределяя его между различными экземплярами одного и того же микросервиса.
3. **Отказоустойчивость:** Если один из микросервисов становится недоступным, Eureka знает об этом и перенаправляет трафик на другие, работающие экземпляры этого сервиса.

Как это работает?

Eureka состоит из двух основных компонентов:

1. **Eureka Server:** Это центральное место, где хранится информация обо всех зарегистрированных микросервисах. Можно представить его как “центральную телефонную книгу”.
2. **Eureka Client:** Это библиотека, которую включают в каждый микросервис. Она позволяет микросервису регистрироваться в Eureka Server и запрашивать информацию о других сервисах.

В итоге Eureka создает удобную, централизованную и автоматизированную систему для обнаружения и управления микросервисами в вашей архитектуре.

Давайте рассмотрим, как легко начать работать с Eureka на практике.

1. Настройка Eureka Server

Для начала нам нужно настроить сервер Eureka.

1.1. Добавьте зависимости в ваш pom.xml (если вы используете Maven):

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

1.2. В вашем главном классе приложения добавьте аннотацию @EnableEurekaServer:

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {
    public static void main(String[] args) {
```

```
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}
```

1.3. Настройте application.properties или application.yml:

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

Теперь, запустив приложение, у вас будет работающий сервер Eureka на порту 8761.

2. Регистрация микросервиса как Eureka Client

Чтобы ваш микросервис мог регистрироваться на сервере Eureka, выполните следующие шаги:

2.1. Добавьте зависимость в pom.xml:

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

2.2. В вашем главном классе приложения добавьте аннотацию @EnableEurekaClient:

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

2.3. В конфигурационном файле укажите адрес сервера Eureka:

```
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
```

Теперь, каждый раз когда вы запускаете свой микросервис, он автоматически регистрируется на сервере Eureka.

Теперь у вас есть базовое понимание того, как настроить сервер Eureka и зарегистрировать на нем свой микросервис. После этих действий вы можете

посетить веб-интерфейс Eureka Server на http://localhost:8761 и увидеть свой микросервис в списке.

Zuul

Друзья, представьте себе величественные ворота, стоящие перед замком. Чтобы войти внутрь замка, вам нужно пройти через эти ворота, и у них есть страж, который решает, кто может пройти, а кто нет. В мире микросервисов у нас есть свой “страж ворот” – и это Zuul от Spring Cloud Netflix.

Что такое Zuul?

Zuul — это API Gateway, что в переводе можно назвать “шлюзом для API”. Его задача — быть первой точкой взаимодействия между пользователями (или другими сервисами) и вашими микросервисами. Почему это полезно?

1. **Маршрутизация:** Zuul определяет, к какому микросервису направить приходящий запрос. Это как навигатор, который знает кратчайший путь к нужному месту в замке.
2. **Фильтрация:** Zuul может обрабатывать запросы, прежде чем они достигнут микросервиса. Это может включать в себя задачи аутентификации (проверка личности), авторизации (определение, что вы можете делать) или модификации запроса.

3. **Защита:** С помощью Zuul вы можете защитить свои микросервисы от вредоносных атак или нежелательного трафика, блокируя или ограничивая некоторые запросы.

4. **Отказоустойчивость:** Zuul может автоматически повторять запросы к другим экземплярам микросервиса, если первый экземпляр не отвечает. Это как если бы у вас было несколько дверей в одной комнате, и если одна дверь заблокирована, вы просто пользуетесь другой.

Как это работает?

В основе работы Zuul лежат фильтры. Есть несколько типов фильтров:

1. **Пред-фильтры (Pre Filters):**

   Пред-фильтры Zuul активируются до того, как запрос будет направлен к своему конечному месту назначения — к вашему микросервису. Они идеально подходят для задач, связанных с предварительной обработкой запросов.

   Например, представьте, что каждый приходящий запрос — это посетитель клуба. Пред-фильтр может действовать как барьер, проверяющий правильность учетных данных или токена авторизации запроса, прежде чем разрешать ему вход. Это немного похоже на проверку билета или удостоверения личности перед входом.

   Кроме того, пред-фильтры могут играть роль регистратора, записывая в логи информацию о приходящем запросе — его тип, URL, заголовки и так далее. Это немного похоже на то, как дверной страж регистрирует вход каждого гостя.

   Также эти фильтры могут изменять запросы, добавляя, изменяя или удаляя параметры и заголовки. Это можно сравнить с тем, как дверной охранник может дать посетителю специальный браслет или метку для доступа к определенным зонам клуба.

   Важно помнить, что пред-фильтры Zuul выполняются в определенном порядке. Если один из фильтров решает, что запрос не должен быть обработан, остальные фильтры могут быть пропущены, и запрос будет немедленно отклонен.

В общем, пред-фильтры Zuul — это первая линия обороны, которая гарантирует, что только нужные и безопасные запросы проходят дальше. Они делают систему надежной, обеспечивая ее безопасность и корректное функционирование.

2. **Фильтры маршрутизации (Route Filters):**

После того как охранниик (пред-фильтр) проверил и допустил посетителя в клуб, у нас есть новая задача — куда направить этого посетителя? В VIP-зону? На танцпол? К барной стойке? Вот где в игру вступают фильтры маршрутизации Zuul.

Фильтры маршрутизации в Zuul, по сути, определяют путь, по которому будет направлен ваш запрос после того, как он был предварительно обработан. Их основная задача — прокладывать маршрут для каждого запроса к соответствующему микросервису или конечной точке.

Вернемся к нашему клубу. Посетитель, которого дверной страж только что пропустил, — это ваш запрос. Фильтр маршрутизации — это дружелюбный хост или гид, который берет посетителя за руку и направляет его к нужной локации в клубе на основе его билета, браслета или другой информации.

Если посетитель пришел на конкретное мероприятие, гид направит его к нужной сцене. Если же у посетителя VIP-билет, гид проведет его в VIP-зону. В мире микросервисов фильтры маршрутизации анализируют запрос, чтобы определить, к какому сервису он должен быть направлен, и перенаправляют его соответственно.

Таким образом, фильтры маршрутизации играют ключевую роль в управлении трафиком внутри вашей системы на основе микросервисов. Они гарантируют, что каждый запрос достигает своей цели быстро и эффективно, что, в свою очередь, помогает системе работать плавно и эффективно.

3. **Пост-фильтры (Post Filters):**

Так, ваш посетитель нашел свое место в клубе благодаря фильтру маршрутизации. Он танцует, слушает музыку, заказывает напитки. Но когда приходит время уходить, что происходит? Он, возможно, проходит через выходную проверку, где ему могут дать листовку о следующем мероприятии

или сувенир в память о ночи. В нашем мире микросервисов и Zuul эта последняя стадия обработки представлена пост-фильтрами.

Пост-фильтры в Zuul активируются после того, как запрос был обработан и получил ответ от микросервиса. Эти фильтры обычно используются для различных "постобработок" ответа перед его отправкой пользователю.

Вернемся к клубу: представьте, что пост-фильтр — это охранник у выхода, который ставит печать на вашей руке, благодаря которой вы сможете вернуться в клуб без дополнительной проверки. Или это тот дружелюбный работник, который дает вам скидочный купон на следующее мероприятие.

В контексте микросервисов, пост-фильтры могут, например, добавлять к ответу некоторые HTTP-заголовки, логировать информацию о запросе или изменять формат ответа.

Таким образом, пост-фильтры гарантируют, что после того, как запрос был обработан, он подготавливается и оптимизируется для окончательной передачи пользователю. Они играют ключевую роль в обеспечении того, чтобы ваш пользователь получил лучший возможный опыт, уделяя внимание деталям и завершающим штрихам.

И вот так, благодаря команде пред-, маршрутизационных и пост-фильтров, каждый запрос в мире Zuul и микросервисов обрабатывается аккуратно, эффективно и с учетом всех нюансов.

4. **Фильтры ошибок (Error Filters):**

Итак, представьте себе, что вы возвращаетесь в тот же клуб, но на этот раз что-то идет не так. Возможно, ваш QR-код на входе не сканируется, или у вас возникает конфликт с другим посетителем. Что будет в этом случае? Наверняка появится администратор или охранник, чтобы помочь разрешить ситуацию. В мире микросервисов и Zuul этот "администратор" представлен фильтрами ошибок.

Фильтры ошибок в Zuul срабатывают, когда в процессе выполнения запроса происходит исключение или другая проблема. Эти фильтры дают возможность централизованно обрабатывать ошибки, логировать их и отправлять соответствующий ответ пользователю.

Вернемся к аналогии с клубом. Представьте, что фильтр ошибок — это администратор, который спешит на помощь, когда что-то идет не так. Он

может предложить вам бесплатный напиток в качестве извинений за неудачное сканирование вашего QR-кода или помочь разрешить конфликт, обеспечивая безопасность и комфорт для всех гостей.

Таким же образом, в мире Zuul, если происходит исключение на этапе пред-фильтрации, маршрутизации или даже после выполнения запроса, фильтр ошибок "перехватывает" эту проблему. Он может логировать ошибку для дальнейшего анализа и, возможно, отправлять пользователю дружелюбное сообщение об ошибке.

Таким образом, благодаря фильтрам ошибок, система может гарантировать, что даже в случае непредвиденных проблем пользователь получит корректный и информативный ответ. Это улучшает общий опыт взаимодействия и дает разработчикам инструменты для быстрого выявления и устранения проблем.

Zuul — это мощный инструмент в мире микросервисов. Он позволяет не только управлять трафиком, но и обеспечивает безопасность, масштабируемость и отказоустойчивость вашего приложения. С его помощью вы можете быть уверены, что ваши микросервисы защищены и работают эффективно.

Теперь закатываем рукава и приступаем к практической части.

1. Настройка Zuul как API Gateway

1.1. Для начала добавим необходимые зависимости в pom.xml (если вы используете Maven):

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

1.2. Активируем Zuul в вашем главном классе приложения, добавив аннотацию @EnableZuulProxy:

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

1.3. В конфигурационном файле укажите адрес сервера Eureka и задайте маршруты для Zuul:

```
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/

zuul.routes.myservice.url=http://localhost:8080
```

Теперь, любой запрос, приходящий на ваш Zuul Gateway по пути /myservice/**, будет перенаправлен на микросервис, который слушает на порту 8080.

2. Применение фильтров в Zuul

Создадим простой пред-фильтр, который логирует информацию о приходящем запросе.

2.1. Создайте новый класс фильтра:

```
@Component
public class SimplePreFilter extends ZuulFilter {

    private static final Logger log =
LoggerFactory.getLogger(SimplePreFilter.class);

    @Override
    public String filterType() {
        return "pre";
    }

    @Override
    public int filterOrder() {
        return 1;
    }
```

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

    @Override
    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        HttpServletRequest request = ctx.getRequest();

        log.info(String.format("%s request to %s", request.getMethod(),
request.getRequestURL().toString()));

        return null;
    }
}
```

Теперь, каждый раз когда кто-то обращается к вашему API Gateway, этот фильтр будет записывать метод и URL запроса.

Вуаля! Теперь у вас настроен API Gateway с Zuul, который умеет перенаправлять трафик к нужным микросервисам и применять к запросам фильтры. Запустите приложение, и попробуйте обратиться к любому из ваших микросервисов через Zuul – и вы увидите, как все это волшебство работает!

Spring Cloud Config

Друзья, представьте себе библиотеку. Но вместо книг на ее полках лежат разные секреты, настройки и параметры для вашего приложения. Такова роль Spring Cloud Config в микросервисном мире.

Что такое Spring Cloud Config?

Spring Cloud Config предоставляет сервер и клиентский компоненты для централизованного управления внешними конфигурациями в распределенной системе.

Как это работает?

Централизованное хранилище: Все ваши конфигурации (настройки для разных окружений, параметры, секреты) хранятся в одном месте, обычно в Git-репозитории. Таким образом, вы можете отслеживать изменения, вносить правки и даже использовать разные версии конфигурации.

Динамическое обновление: Помните те дни, когда при любом изменении настройки нужно было перезапускать приложение? Забудьте об этом! Spring Cloud Config позволяет динамически обновлять конфигурации в ваших приложениях на лету.

Безопасность: Ваши секреты и конфигурации важны, верно? Spring Cloud Config обеспечивает безопасное и шифрованное хранение ваших данных.

Почему это так ценно?

1. **Упрощение управления:** Вместо того чтобы бороздить множество микросервисов в поисках нужной настройки, у вас есть одно место, где все хранится. Это упрощает управление и уменьшает вероятность ошибок.
2. **Быстрое внедрение изменений:** Нужно срочно внести изменение в конфигурацию? Никаких проблем! Сделайте это в централизованном хранилище, и все ваши микросервисы получат это обновление.
3. **Масштабируемость и единообразие:** При добавлении новых микросервисов вам не придется думать о том, где и как хранить их конфигурации. Они все будут единообразными и находиться в одном месте.

Spring Cloud Config – это как ваша личная библиотека настроек, где каждый параметр, каждая настройка и каждый секрет аккуратно каталогизированы, хранятся безопасно и готовы к быстрому доступу. Это ключевой компонент для успешного и гибкого управления микросервисами.

Возьмите ваш кодовый меч, и давайте мастерить!

1. Настройка Config Server

1.1. Начнем с добавления зависимостей в pom.xml:

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

1.2. В главном классе приложения активируем Config Server с помощью аннотации:

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

1.3. Укажем местоположение нашего Git-репозитория в application.properties или application.yml:

spring.cloud.config.server.git.uri=URL_ВАШЕГО_РЕПОЗИТОРИЯ

2. Использование Config Client

Чтобы ваш микросервис мог получать настройки из Config Server, выполните следующие шаги:

2.1. Добавьте зависимость в ваш pom.xml:

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

2.2. Укажите в bootstrap.properties или bootstrap.yml местоположение вашего Config Server:

spring.cloud.config.uri=http://АДРЕС_ВАШЕГО_CONFIG_SERVER

Теперь, при старте, ваш микросервис будет подключаться к Config Server и получать необходимые настройки.

3. Обновление конфигурации в реальном времени

3.1. Для этого нам понадобится еще одна зависимость:

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

3.2. Теперь, когда вы измените что-то в вашем Git-репозитории с конфигурациями, достаточно отправить POST-запрос на /actuator/bus-refresh вашего микросервиса, и он подхватит новые настройки без перезапуска.

Заметка: Для использования этого эндпоинта вам также понадобится настроить Spring Cloud Bus и, например, RabbitMQ.

Мы научились настраивать централизованный сервер конфигураций, подключать к нему клиентов и обновлять конфигурации на лету. Это, безусловно, делает управление микросервисами гибким и устойчивым к изменениям.

Hystrix

Давай представим, что ваше приложение — это рыцарь, который идет в битву. В бою может случиться что угодно: ваш меч может сломаться, конь может упасть, или даже доспехи могут не защитить от стрелы. Но что, если у вас будет магический щит, который поможет избежать многих неприятностей и даст время для опоминания? Вот таким щитом в мире микросервисов является Hystrix!

Что это такое?

Hystrix — это библиотека из мира Spring Cloud, которая предоставляет “обертку” вокруг ваших внешних вызовов. Эта обертка предотвращает сбои и предоставляет резервные стратегии. Если что-то идет не так, Hystrix может “переключить” вас на резервный план, чтобы ваше приложение продолжало работать.

Каковы его основные функции?

1. **Обнаружение ошибок:** Hystrix быстро определяет, когда внешний сервис начинает “хромать” или не отвечать.
2. **Обход ошибок:** Если внешний сервис не работает, Hystrix может автоматически переключить вас на “резервный” метод.
3. **Ограничение потоков:** Чтобы избежать перегрузки, Hystrix может ограничивать количество одновременных запросов к внешнему сервису.
4. **Резервные стратегии:** Hystrix позволяет настроить альтернативное поведение для сценариев, когда внешний сервис недоступен.
5. **Мониторинг и метрики:** С помощью Hystrix Dashboard вы можете отслеживать состояние всех ваших вызовов в реальном времени.

Почему это так важно?

Когда вы работаете с микросервисами, зависимости между сервисами становятся критическими. Если один микросервис “падает”, он может “потянуть за собой” другие сервисы. Hystrix помогает избежать эффекта домино и убедиться, что ваше приложение останется устойчивым даже при непредвиденных проблемах.

Давай разберёмся с магией щита Hystrix на практике!

1. Подключение Hystrix

1.1. Добавь зависимость в pom.xml:

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

2. Активация Hystrix

В главном классе приложения добавь аннотацию @EnableCircuitBreaker:

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

```
    }
}
```

3. Защита метода с помощью Hystrix

Допустим, у нас есть метод, который вызывает внешний сервис:

```
public String fetchData() {
    // ... вызов внешнего сервиса
    return "Data from external service";
}
```

Чтобы защитить этот метод, используем аннотацию @HystrixCommand:

```
@HystrixCommand(fallbackMethod = "defaultData")
public String fetchData() {
    // ... вызов внешнего сервиса
    return "Data from external service";
}

public String defaultData() {
    return "Default data";
}
```

Если вызов fetchData будет длиться слишком долго или завершится ошибкой, Hystrix автоматически переключится на defaultData.

4. Настройка Hystrix

Hystrix предоставляет множество настроек для тонкой настройки. Например, в application.properties:

```
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=2000
```

Здесь мы устанавливаем тайм-аут для команд Hystrix по умолчанию в 2 секунды.

5. Мониторинг с Hystrix Dashboard

Для мониторинга состояния ваших “цепей” Hystrix, можно использовать Hystrix Dashboard.

5.1. Добавь зависимость:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
```

```
    <artifactId>spring-cloud-starter-netflix-hystrix-
dashboard</artifactId>
</dependency>
```

5.2. Активируй Hystrix Dashboard:

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

Теперь вы можете открывать панель управления Hystrix по адресу /hystrix.

С помощью Hystrix вы не только защищаете ваше приложение от сбоев, но и получаете инструменты для мониторинга и диагностики проблем в реальном времени. Это делает вашу систему более устойчивой и надежной.

Заключение

Итак, друзья, давайте подведем итоги нашего урока!

Мы живем в замечательное время, когда технологии развиваются стремительными темпами. Микросервисная архитектура не просто новый тренд или модная “фишка”. Это настоящий революционный подход к разработке программного обеспечения. Помните монолиты? Огромные, сложные, неповоротливые... С приходом микросервисов всё изменилось:

1. **Устойчивость:** Микросервисы работают независимо. Один может “упасть”, но это не остановит остальные. Вспомните наш разговор о Hystrix - это всего лишь один из механизмов, позволяющих сделать систему еще более устойчивой к сбоям.
2. **Масштабируемость:** Нужно больше мощности для определенной части вашего приложения? Просто добавьте еще один инстанс того микросервиса! Без необходимости масштабировать всё приложение целиком.

3. **Гибкость:** Разработка, тестирование и развертывание микросервисов происходит независимо. Это дает возможность быстро вносить изменения, экспериментировать и выпускать новые версии.

И вот здесь на помощь приходит **Spring Cloud**. Этот мощный набор инструментов делает разработку микросервисов еще проще и удобнее. Поддержка обнаружения сервисов с Eureka, маршрутизация с Zuul, централизованная конфигурация с Spring Cloud Config и множество других инструментов позволяют строить надежные, эффективные и масштабируемые системы.

Таким образом, объединив микросервисы и Spring Cloud, мы получаем возможность создавать современные, гибкие и надежные приложения, которые могут легко масштабироваться и адаптироваться к постоянно меняющимся требованиям бизнеса.

В следующем уроке мы погрузимся в мир **Spring Testing**. Если говорить простыми словами, то это - ваша страховка от ошибок и нежданчиков. Мы научимся писать тесты для вашего Spring приложения, используя два мощных инструмента: **JUnit** и **Mockito**.

Спасибо, что были со мной на этом уроке! Надеюсь, информация была полезной, и вы вдохновлены применять новые знания на практике.

Что можно почитать еще?

1. Изучение Spring Boot 2.0. Грег Тернквист
2. Освоение Spring Boot 2.0. Динеш Раджпут

Используемая литература

1. Spring Boot в действии. Крейг Уоллс.
2. Spring Microservices в действии. Джон Карнелл
3. Cloud Native Java. Джош Лонг и Кенни Бастани