О клиенте и сервере в микросервисной архитектуре

Согласно устоявшемуся в индустрии мнению, работа старших разработчиков и архитекторов ПО во многом состоит из поиска компромиссов между преимуществами и недостатками тех или иных решений и выделения "достаточно хороших решений" для поставленных задач.

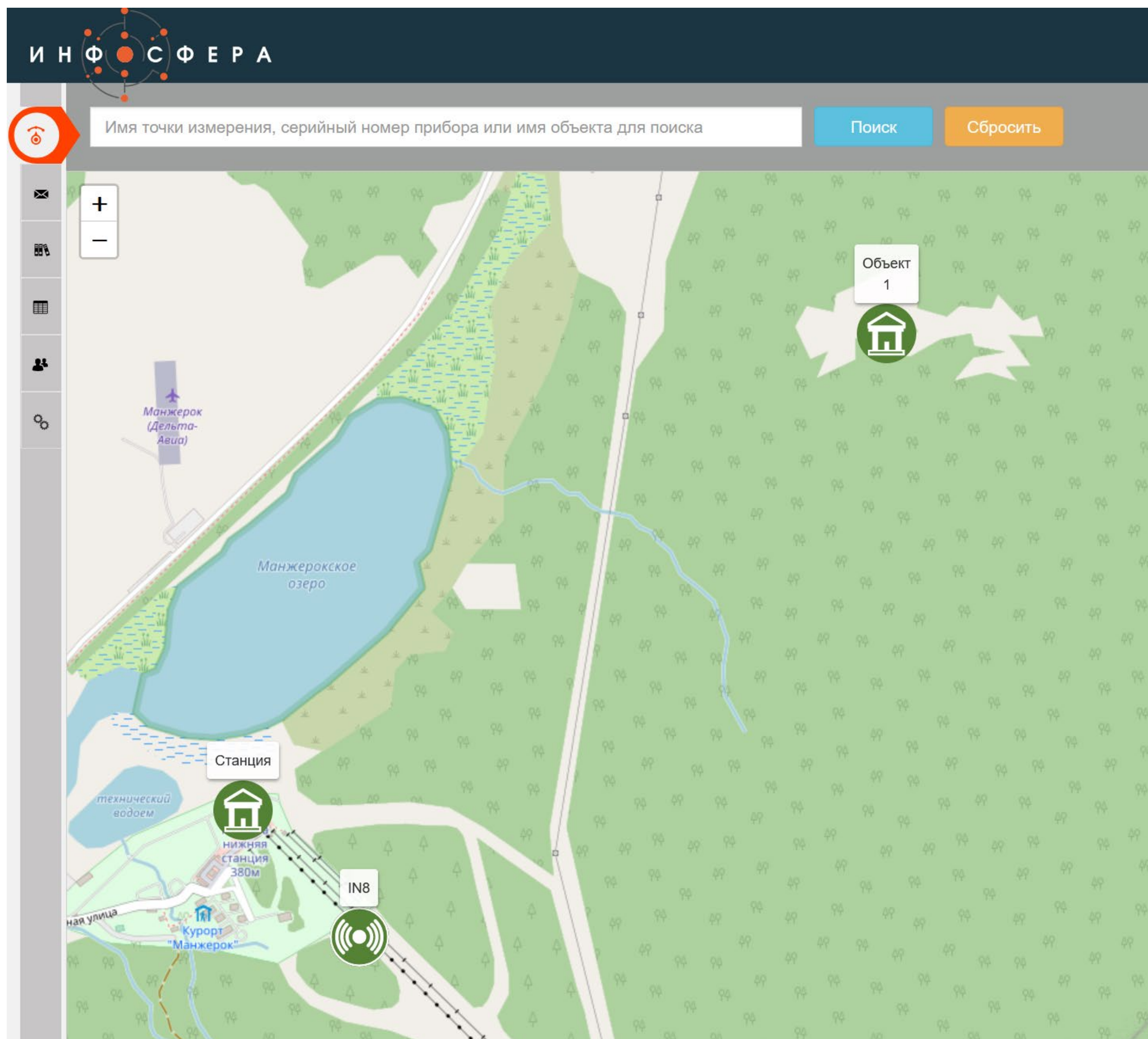
Когда мы задались вопросом перехода на микросервисную архитектуру, мы столкнулись с некоторым количеством подобных трейд-оффов. Проведя ряд экспериментов и отвязавшись от специфических для нашего продукта бизнес-требований, мы попытались сформулировать вопросы, которые могут встать перед любой командой разработки, безотносительно к требованиям к продукту. Ну и, конечно, дать на них ответы - никто не любит вопросы без ответов.

В качестве прикладного дополнения к рассуждениям мы разработаем несколько Proof of Concept, сопроводим их разработку краткими пояснениями и приложим исходный код PoC.

Для нас "родным" стеком является Java 8 и Spring Boot 2, так что приложенные демки будут написаны на основе этих технологий с изрядными вкраплениями Spring Cloud. Мы также постараемся предложить несколько обособленных типовых решений для задач, которые, как нам показалось, могут в перспективе возникнуть перед разработчиками.

О нас

Мы - подразделение группы компаний "Миландр", занимающееся разработкой и поддержкой IoT-платформы "ИНФОСФЕРА". Этот продукт включает в себя комплекс решений для ЖКХ, умного дома, электросетевой энергетики и промышленных предприятий. Мы собираем данные с различных приборов (счетчиков, датчиков, камер, домофонов, умных устройств), позволяем клиентам оперировать этими данными (в том числе, с использованием алгоритмов ML), настраивать пользовательские сценарии автоматизации для обработки данных, а также осуществлять удаленное управление приборами.



Интерфейс "ИНФОСФЕРА Диспетчерская"

Для кого эта статья

1. Для новичков, желающих ознакомиться с основными компонентами Spring Cloud и принципами, лежащими в основе микросервисной архитектуры.
2. Для разработчиков и архитекторов, планирующих переход к микросервисной архитектуре.
3. Для разработчиков и архитекторов, желающих пополнить коллекцию типовых решений и почитать про грабли, на которые можно наступить при первом использовании Spring Cloud.

Про микросервисы

Микросервисам и Spring Cloud на Хабре уже посвящено множество статей, которыми, отчасти, мы вдохновлялись и которые хотели бы дополнить в тех моментах, где чувствовали недостаточное раскрытие темы.

Концепция микросервисной архитектуры достаточно молода, но мы уже успели за 5-6 лет посмотреть, как она прошла все этапы Gartner Hype Cycle.

Мы видели, как в 2015-м году люди восторгались новым подходом, видели и отторжение технологии в 2016-м, и теперь, после дозревания технологии и отношения к ней, можем видеть прагматические рассуждения, с трезвыми рассуждениями и подробным описанием плюсов и минусов.

Что касается необходимости использования такой архитектуры, для нас при реализации нескольких новых проектов возникла необходимость в адаптации к возрастающим нагрузкам. Возможность выборочного масштабирования отдельных компонентов системы и послужила главным аргументом для рассмотрения возможности перехода на новую архитектуру.

Про Spring Cloud

Spring Cloud - набор инструментов, позволяющих организовать работу Spring-based приложений в соответствии с принципами микросервисной архитектуры.

Статьи, описывающие использование Spring Cloud

- Отличная статья от @sqshq с уклоном в практические аспекты создания микросервисного приложения, с подробными описаниями действий и отличными примерами.
- Простейшее демо, показывающее, как запустить Eureka Server и подключить к нему клиенты.

Небольшое уточнение для первой работы со Spring Cloud

Обращаем внимание на не совсем привычный механизм указания зависимостей от Spring Cloud. После создания проекта с зависимостями из Spring Cloud через Spring Initializr в глаза сразу бросается, что starters из Spring Cloud не наследуются из родительского pom-файла, как это происходит со стартерами Spring Boot, а приносятся через dependency management:

```
<properties>

        <spring-cloud.version>2020.0.3</spring-cloud.version>

</properties>
```

```
...

<dependencyManagement>

        <dependencies>

                <dependency>

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

                        <artifactId>spring-cloud-dependencies</artifactId>

                        <version>${spring-cloud.version}</version>

                        <type>pom</type>

                        <scope>import</scope>

                </dependency>

        </dependencies>

</dependencyManagement>
```

1. Так что же за вопросы?

В нашем случае на первый план вышли вопросы, касающиеся принципов организации клиентского приложения и наладки его взаимодействия с серверной стороной. Сложности добавляет тот факт, что все эти вопросы достаточно плотно переплетены между собой. Будем разбираться.

Вопрос	Варианты решения
Выбор принципа рендеринга веб-страниц	Client-side rendering, Server-side rendering, Mixed
Хранение и выдача интерфейса на клиент	Единая точка, микрофронтенды, несколько точек-страниц

Протоколы взаимодействия клиента и сервера, а также сервисов друг с другом	Брокер, HTTP, WS, смесь
Необходимость Service Discovery	Зависит от использования HTTP

Мы намеренно оставляем за пределами подробного рассмотрения (ограничившись упоминаниями, по крайней мере, в рамках данной статьи) следующие вопросы:

- Использование Spring Security для обеспечения безопасности использования приложения,
- Распределенные транзакции для обеспечения согласованности баз данных разных микросервисов,
- Балансировка нагрузки для адаптируемости отдельных компонентов к возрастанию нагрузки,
- Боевое развертывание, пока говорим преимущественно о процессе разработки. Некоторые недостатки предлагаемых подходов могут быть нивелированы за счет грамотного проектирования и администрирования.

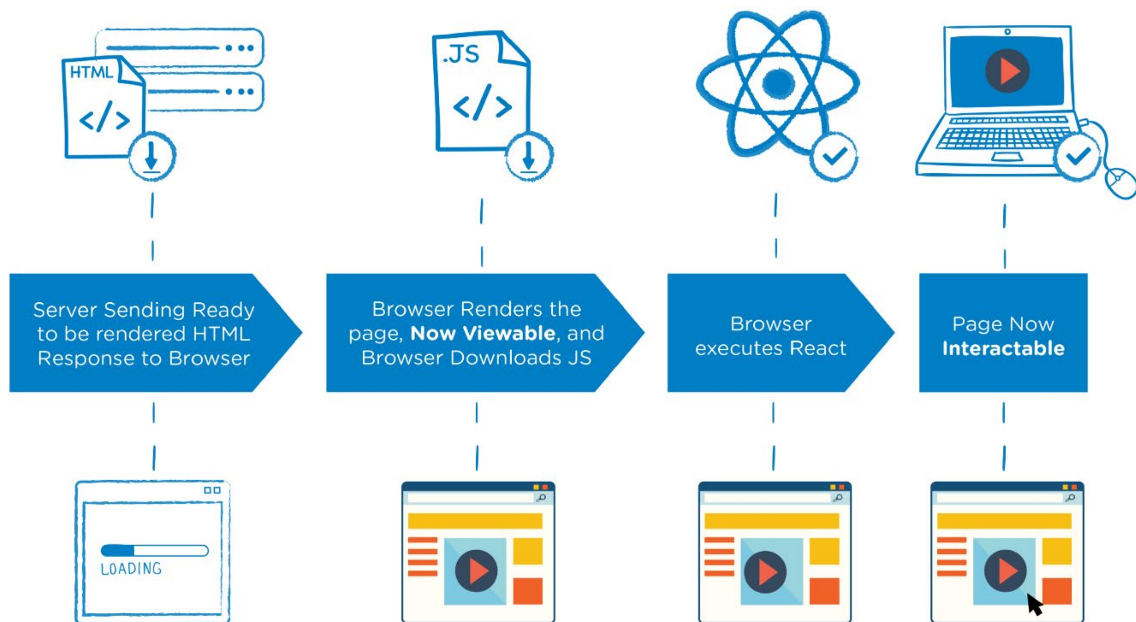
1.1 Рендеринг страниц

Первый вопрос, который хотелось бы рассмотреть, относится к выбору подхода к рендерингу страниц.

Под этими словами подразумевается, где будет формироваться HTML-документ для дальнейшего его использования браузером. На текущем этапе развития технологий в основном предлагается использовать клиентский рендеринг (Client-side rendering, CSR) и серверный рендеринг (Server-side rendering, SSR).

Рассмотрим коротко суть каждого из подходов.

Серверный рендеринг



Данный подход предполагает формирование HTML-страницы (возможно, с каким-то изначальным набором данных, предоставляемых в рамках запроса пользователя к странице) на сервере. Дальнейшее взаимодействие пользователя с интерфейсом может осуществляться посредством исполнения загружаемого JS-кода или полным обновлением всей HTML-страницы при выполнении того или иного действия.

Преимущества

- С точки зрения пользователя это позволяет быстро получить отрисованную страницу (и даже **содержательную** отрисовку, если сервер перед выдачей страницы сразу подставляет в нее данные).

Недостатки

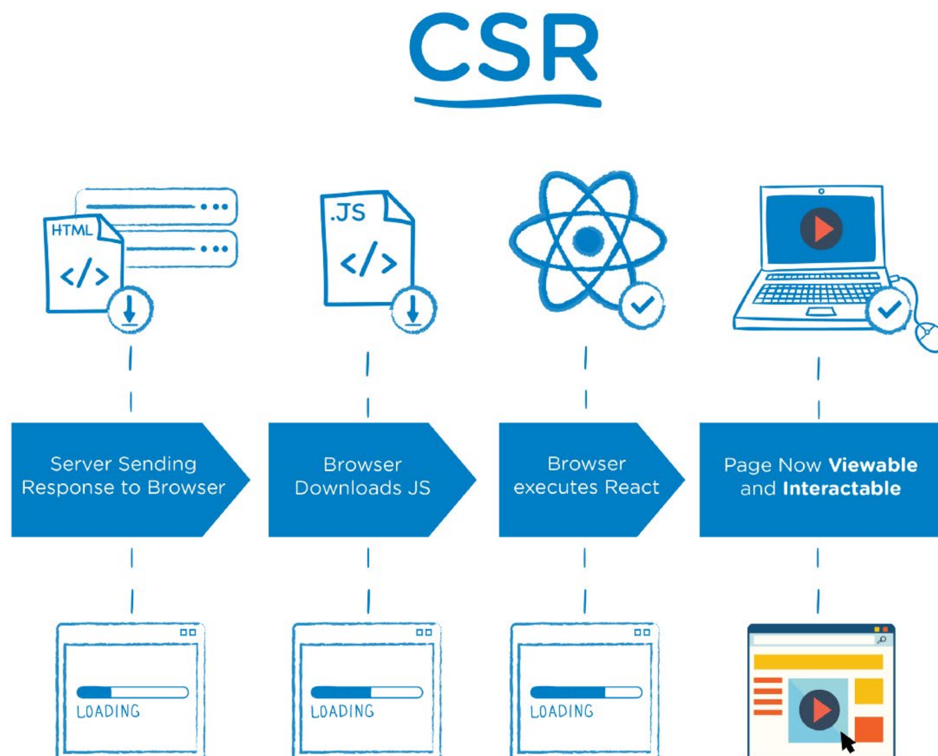
- После отрисовки страница может еще некоторое время находиться в неоперабельном состоянии, до подготовки JS-кода к исполнению;
- Без ухищрений в клиентском коде может потребоваться полное обновление страницы для обновления данных.

Технологии:

- JSP;

- Thymeleaf**;**
- Mustache;
- + ваши любимые **JS-библиотеки;**

Клиентский рендеринг



Данный подход, напротив, предполагает формирование страницы на клиенте посредством исполнения JS-кода. Например, так поступает React при работе через JSX, формируя содержимое страницы из кусков разметки, возвращаемой из компонентов.

Преимущества

- Почти одновременно происходит отрисовка страницы и наступает готовность к взаимодействию.

Недостатки

- В скорости отрисовки проигрывает серверному рендерингу, поскольку JS необходимо загрузить, а затем исполнить на клиенте. Страница будет отрисована и готова к работе только по окончании работы всего необходимого для отрисовки JS-кода.

Ссылки

Хорошая обзорная статья (картинки взяты из нее же)

1.2 Протоколы взаимодействия клиента и сервера, а также сервисов друг с другом

Для микросервисной архитектуры в качестве механизма обмена данными между сервисами зачастую рекомендуют использовать событийно-ориентированную архитектуру

Данный подход к разработке информационных систем предписывает обеспечивать общение сервисов друг с другом посредством обмена сообщения через специализированный middleware-софт, предназначенный для работы по принципам PUB/SUB. PUB/SUB обеспечивает слабую связанность приложений-источников сообщений и приложений-потребителей сообщений.

Мы и раньше использовали механизм обмена событиями между приложениями, но в весьма ограниченном круге задач: так передавались показания от приборов и отправлялись команды на приборы.

Здесь мы вплотную подошли к вопросу: как следует организовать взаимодействие сервисов друг с другом? Как определить, следует ли в конкретной ситуации использовать событийный подход или следует использовать типичные для HTTP запрос-ответ? Порассуждаем.

Событийный подход хорошо показывает себя, когда:

- стороне, публикующей событие, не требуется ответ (реакция на это событие);
- есть несколько сервисов, заинтересованных в получении события;
- существует потребность в сохранении события в топике (для случая использования Kafka или другого персистентного брокера сообщений) для возможности потом его прочитать из топика.

С другой стороны, HTTP лучше подходит в ситуациях:

- когда требуется получить ответ на запрос;
- когда требуется выполнить синхронную операцию;
- когда не нужно привносить дополнительную сложность в систему, ограничившись синхронным обращением одного сервиса к другому.

Из этого набора фактов следует, что комбинировать события и HTTP-запросы - уместная практика. При этом нужно четко осознавать, для каких задач какой тип взаимодействия использовать.

Случай “внешнего” клиента

Тут хотелось бы обсудить вопрос адресации при обращении к сервисам из внешнего клиента. Для выполнения HTTP-запроса к сервису клиенту необходимо знать адрес этого сервиса. В целом, существует несколько типовых решений этой проблемы:

1. Хардкод адресов в клиентском коде (негибко и очень сложно поддерживать в боевом окружении, отметаем);
2. Параметризация клиентского кода (негибко, отметаем);
3. Использование паттерна Service Discovery для клиента (подробнее о Service Discovery можно прочесть ниже) и отправка HTTP запросов напрямую к сервисам;
4. Использование паттерна API gateway для проксирования всех запросов к backend-сервисам через единую точку входа. Подробнее об API gateway можно прочесть ниже;
5. Если же основное взаимодействие между сервисами строится на основе событий и брокера сообщений, можно попытаться подключить клиентское приложение напрямую к брокеру. Хотя современные технологии и позволяют это сделать, такая архитектура привносит серьезные проблемы с безопасностью, и потому далее не рассматривается.

Из пяти вариантов для рассмотрения остаются два:

1. Service Discovery;
2. API Gateway.

Service Discovery для клиентских приложений - жизнеспособный подход, обеспечивающий возможность менять конфигурацию адресов сервисов на лету. Проблема в том, что он предполагает прямые обращения к сервисам, что вынуждает выставлять все сервера, с которыми взаимодействует клиент, наружу из защищенных сетей. Такой подход приводит к возникновению большого количества направлений атаки, поэтому противоречит хорошим практикам безопасности.

Еще одна сложность может возникнуть вследствие недопустимости прямого доступа клиента к брокеру сообщений. Если для обмена данными между сервисами будет использоваться исключительно событийная модель, сервисам все равно потребуется HTTP API, чтобы внешний клиент мог взаимодействовать с ними.

С другой стороны, API Gateway выступает единой точкой входа клиентов в защищенную сеть, поэтому, будучи единственным сервисом, доступным снаружи, выступает единственным направлением атаки для злоумышленника, не проникшего в защищенную сеть. Следует отметить, что по тем же причинам API

Gateway также является и единой точкой отказа, выход которой из строя сделает невозможным взаимодействие клиентов с сервером.

Другая сильная сторона API Gateway заключается в возможности выступать шлюзом для внешних запросов даже в том случае, когда внутри сервисной сети используется обмен данными только через брокер. Нам не удалось найти готовых реализаций инструментов, которые могли бы производить "перекладывание" запроса к backend в брокер, но подобная логика может быть без труда реализована и посредством обычных Web-контроллеров.

Существует еще один вариант архитектуры - введение дополнительного слоя сервисов 1-го уровня, способных принимать запросы от API Gateway по HTTP и перекладывать их в брокер. Такой подход позволяет и не вводить HTTP API для всех сервисов и сохранить “чистоту” API Gateway, не привнося туда логику работы с брокером.

Еще одно преимущество API Gateway над подходом Service Discovery может быть получено при использовании [WebSocket]() для полнодуплексного обмена данными между клиентом и сервером. Например, может возникнуть задача уведомлять клиентское приложение о происходящих в системе событиях. Источниками таких событий могут выступать несколько backend-сервисов. Вместо поддержания нескольких WS-соединений с такими сервисами клиент может поддерживать одно соединение с одной точкой, предоставляющей возможность пересылки таких уведомлений через WS. API Gateway - неплохой кандидат для решения этой задачи.

Исходя из приведенных рассуждений, мы сформулировали для себя следующие выводы:

1. Смешение событий и HTTP-запросов - допустимая практика;
2. Запрос от клиента к API Gateway следует направлять по HTTP;
3. Во внутренней сети для поиска адресов сервисов для последующего выполнения HTTP-запросов следует использовать Service Discovery. Он же может служить заделом для масштабирования;
4. Использовать HTTP для межсервисного взаимодействия на ранней стадии переработки системы - допустимая практика, обеспечивающая простоту и быстрое построение работоспособной системы.

Spring Cloud Gateway

Ссылки

1. Концепт API Gateway
2. Официальный гайд по Spring Cloud Gateway
3. Гайд от Baeldung по Spring Cloud Gateway

Документация по java-конфигурации Spring Cloud Gateway:

1. PredicateSpec
2. UriSpec

Spring Cloud Gateway принимает входящие внешние запросы, проверяет свои правила маршрутизации и, в случае нахождения подходящего для пришедшего запроса правила, перенаправляет запрос к одному из сервисов.

Данный сервис предназначен для упрощения взаимодействия клиента, находящегося во внешней сети, с сервисами, расположенными во внутренней backend-сети.

Использование

По сути, для начала работы со Spring Cloud Gateway нужно сделать лишь две вещи:

1. Добавить зависимость в ClassPath;
2. А Определить бин типа org.springframework.cloud.gateway.route.RouteLocator и добавить его конфигурацию;
   ИЛИ
   Б Прописать конфигурацию в файле application.properties/yaml
   Пример

На этапе разработки мы решили остановиться на java-конфигурации. Это позволило опробовать более тонкие варианты конфигурации API-gateway. Однако, вместе с тем, в промышленной эксплуатации этот вариант привносит и ограничения, не позволяя изменять конфигурацию "на лету" (например, с использованием Spring Cloud Config), вынуждая менять код приложения, исполняемого в боевом окружении, и, как следствие, требуя пересборки приложения при необходимости внести изменения в конфигурацию шлюза.

Мы попытались предложить несколько типовых решений для конфигурации Spring Cloud Gateway, которые могли бы быть полезны сообществу.

Пример 1: Меняем request path при помощи регулярного выражения. Запрос /google/hello приведет к обращению по адресу с параметром http://google.com/search?q=hello.

```
@Bean
```

```java
public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {

  return builder

        .routes()

                .route(r -> r

                        .path("/google/**")

                        .filters(gatewayFilterSpec ->
gatewayFilterSpec.rewritePath("/google/(?<appendix>.*)", "/search?q=${appendix}"))

                        .uri("http://google.com"))

        .build();

}
```

Пример 2: Отрезаем от path 1 блок. Запрос /yandex/hello/123 будет перенаправлен на http://yandex.ru/hello/123

```java
@Bean

public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {

  return builder

        .routes()

                .route(r -> r

      .path("/yandex/**")

      .filters(gatewayFilterSpec -> gatewayFilterSpec.stripPrefix(1))

      .uri("[http://yandex.ru](http://yandex.ru)"))
```

```
        .build();

}
```

Пример 3: Формируем URI для перенаправления, вычленяя адрес сервиса из path запроса.

```
@Bean

public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {

  return builder

        .routes()

          .route(r -> r

      .path("/service/**")

      .filters(gatewayFilterSpec ->
gatewayFilterSpec.changeRequestUri(serverWebExchange -> {

          ServerHttpRequest originalRequest = serverWebExchange.getRequest();

          URI oldUri = serverWebExchange.getRequest().getURI();

          UriComponentsBuilder newUri = UriComponentsBuilder.fromUri(oldUri)

          .host(originalRequest.getPath().subPath(3, 4).toString() + ".com") //
0,1,2,3 - /service/<serviceName>,

          .port(null)

          .replacePath(originalRequest.getPath().subPath(4).toString());

          return Optional.of(newUri.build().toUri());

        }))
```

```
      .uri("http://ignored-URI")) // Этот URI игнорируется
        .build();
}
```

Пример 4: Прописываем route с минимальным приоритетом. Запросы по адресам, не предусмотренным в других route, будут пересылаться по адресу localhost:8090.

```
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
  return builder
        .routes()
          .route(r -> r
          .order(Integer.MAX_VALUE)
          .path("/**")
          .uri("[http://localhost:8090](http://localhost:8090)"))
        .build();
}
```

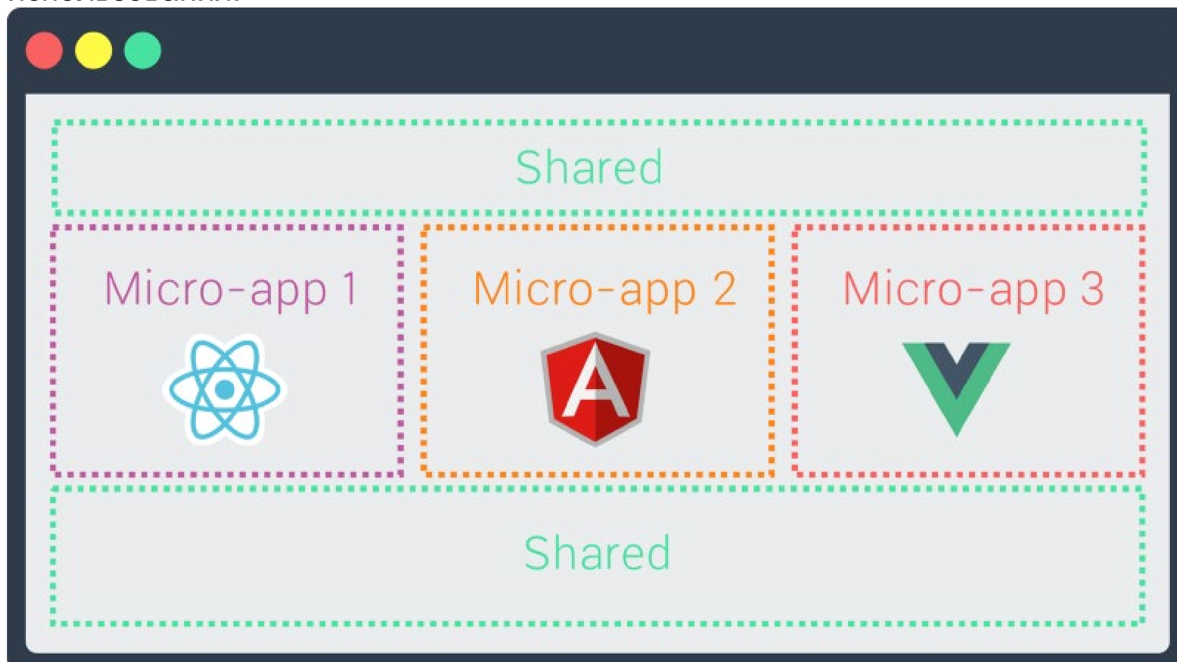
1.3 Хранение интерфейса и его выдача на клиент

В рамках решения задачи передачи интерфейса для исполнения на клиент также существует несколько вариантов действия.

Микрофронтенды

По сути - это продолжение идеи микросервисов и на UI. Некоторые микросервисы, помимо выполнения серверных задач, также имеют в своей зоне ответственности

части пользовательского интерфейса, которые могут быть отданы на клиент для использования.



Оригинальное изображение - https://techrocks.ru/2018/09/19/prepare-your-skill-set-for-web-developer-interviews-2/microfrontends/

Объединение компонентов может осуществляться как на клиенте, так и на сервере.

- Компоновка на клиенте
- Компоновка на сервере

Преимущества

- Гибко - позволяет развивать UI разных компонентов независимо друг от друга.
- Продолжение идеи независимого развертывания - позволяет исключать микросервисы из продакшена, что будет приводить к исчезновению лишь отдельных компонентов со страницы. Например, упал сервис N, его кусочки интерфейса недоступны, а все остальное - живее живых. Круто.
- Возможность использовать разные технологии для разных блоков интерфейса. Если не уходить в занудство на тему однообразия технологий и простоты поддержки, следует признать, что это все же выглядит скорее преимуществом.

Недостатки

- Сложно организовать совместную работу с UI. Подход требует серьезной дисциплины, например, в части выдерживания единых стилей. Как следствие, нужны компетенции в клиентской разработке.

- Накладные расходы на формирование интерфейса из кусочков. Как трудовые, так и вычислительные.
- Подход молодой, и его нужно глубоко и аккуратно изучать.

Технологии
- Single SPA
- Bit

Единая точка хранения интерфейса

Следующий вариант предписывает хранение всего, что нужно для работы пользовательского интерфейса, в одном сервисе. В качестве такого сервиса может существовать как обособленный сервис, так и сервис, совмещенный по функционалу с другим сервисом.

Вариант 1 - Выделенный UI-сервис

В backend-сети выделяется отдельный сервис, единственная задача которого - хранить и отдавать по запросу пользовательский интерфейс.

Преимущества
- Логическая обособленность интерфейсного кода и страниц от остального кода приложения.

Недостатки
- Необходимость создания и администрирования отдельного сервиса.

Технологии
- Spring Web + JSP/Thymeleaf как простейший пример для минимальной реализации.

Вариант 2 - UI Gateway

Выше мы уже упоминали паттерн API Gateway. Существует практика расширения этого шлюза до **UI Gateway** - шлюза с функционалом хранения и раздачи страниц и клиентского кода.

Преимущества
- Меньшее количество сервисов, чуть большая простота администрирования, не нужно прописывать дополнительные роутинги на UI.

Недостатки

- Как мы помним, API Gateway - единая точка отказа системы. Даже при наличии его реплицированных экземпляров в эксплуатации обновление такого шлюза для обновления UI выглядит достаточно рискованной операцией.

Технологии

Spring Web + JSP/Thymeleaf как простейший пример для минимальной реализации.

Вариант 3 - Page UI-сервис

Концепция похожа на предыдущую с той лишь разницей, что единый UI-сервис в данном подходе разбит на несколько сервисов, каждый из которых отвечает за одну или несколько страниц.

Преимущества

- При необходимости обновить код одной из страниц достаточно перезапустить только тот UI-сервис, который его содержит.
- Меньший объем кодовой базы в каждом сервисе.

Недостатки

- Сложно организовать совместную работу с UI, определенно требует серьезной дисциплины, например, в части выдерживания единых стилей на разных страницах.

Технологии

Spring Web + JSP/Thymeleaf как простейший пример для минимальной реализации.

1.4 Необходимость Service Discovery

Ссылки

- Server-Side Discovery
- Client-Side Discovery

Service Discovery - обобщенное название для подходов, позволяющих сервисам (и, возможно, клиентам) обнаруживать друг друга в сети. В контексте данной статьи Service Discovery реализуется посредством использования Service Registry

- реестра сервисов. Когда очередной сервис запускается, он регистрируется в реестре сервисов. Когда у клиента (внешнего или сервиса) возникает необходимость обратиться к одному из сервисов по его имени, он обращается к реестру за информацией о точном адресе сервиса. Получив его, клиент отправляет запрос.

Spring Cloud Eureka

Ссылки

- Официальное руководство от Spring
- Англоязычная статья с описанием Eureka
- Русскоязычная статья с описанием Eureka

Eureka Server - приложение, реализующее паттерн Service Discovery и основанное на Service Registry. Eureka развивается в рамках экосистемы Spring Cloud. Наряду с Eureka существует несколько других реализаций для Java, например Consul и Zookeeper.

Перезапуск Eureka-server приводит к очищению его реестра, однако наличие локальных копий реестра в приложениях позволяет клиентам какое-то время обходиться без Eureka-сервера, а наличие heartbeat-сообщений от клиентов позволит оперативно восстановить содержимое реестра после перезапуска.

Использование

В простейшем случае конфигурация клиентов и сервера достаточно проста, что свойственно всей инфраструктуре Spring Cloud.

Важно: настройки, начинающиеся с `eureka.server`, следует задавать в приложении-сервере, а `eureka.instance` и `eureka.client` - в том приложении, которое мы хотим сконфигурировать как клиент.

Для сервера: после добавления зависимости на Eureka-сервер

```
<dependency>

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

        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
```

```
</dependency>
```

2. Указать `server.port = 8888` (по умолчанию - 8761)
3. Отключить попытки Eureka-сервера зарегистрироваться в своем собственном реестре и попытки загружать этот реестр `eureka.client.register-with-eureka=false`, `eureka.client.fetch-registry=false`
4. Включить использование Eureka-сервера, добавив аннотацию `@EnableEurekaServer` на Main-класс приложения.

Для клиента: после добавления зависимости на Eureka-клиент:

```
<dependency>

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

        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>
```

2. Указать URL Eureka-сервера: `eureka.client.serviceUrl.defaultZone = http://localhost:8888/eureka/`
3. Определить имя, с которым приложение будет регистрироваться в реестре сервисов. Это можно сделать одним из двух способов:
4. 1. Указать имя приложения через настройку `spring.application.name`, и тогда несколько инстансов одного приложения будут объединены в группу по имени приложения, что позволит в будущем использовать эту группу для распределения нагрузки.
5. Указать имя **экземпляра** приложения через настройку `eureka.instance.instanceId`. InstanceId - имя экземпляра приложения в реестре сервисов. **Важно:** instanceId должен быть уникальным для обеспечения Eureka возможности поиска уникального экземпляра приложения для точной адресации. Регистрация приложения с именем, указанным через eureka.instance.instanceId, усложняет дальнейшее внедрение балансировки нагрузки, поэтому мы рекомендуем использовать spring.application.name. Если не указать ни одну из этих двух настроек, приложение зарегистрируется в реестре с именем UNKNOWN.
6. Включить использование Eureka-клиента, добавив аннотацию `@EnableEurekaClient` на Main-класс приложения.

Использование Eureka для Service Discovery возможно и в API Gateway. После включения `@EnableEurekaClient` в API Gateway, достаточно URI перенаправления прописать в формате `lb://service-name` (lb - load balancing, балансировка нагрузки, задел на масштабирование нагрузки по инстансам, известным Service Registry).

Примечание: Поскольку к API Gateway ни один из сервисов, использующих Service Discovery, скорее всего, обращаться не будет, можно отключить попытки шлюза зарегистрироваться в реестре, указав настройку `eureka.client.register-with-eureka=false`

Запустив приложение, выступающее Eureka-сервером, и перейдя по его базовому адресу (хост + порт), можно увидеть интерфейс service-registry, в котором перечислены зарегистрировавшиеся в реестре приложения.

Грабли: Healthcheck в Eureka и ClassNotFoundException

Проведем границу. **Heartbeat**-сообщения - сообщения, сигнализирующие о том, что сервис существует (без детализации его состояния). **Healthcheck** = Heartbeat + детализация состояния (посредством использования статуса Spring Boot Actuator).

Для поддержания реестра в актуальном состоянии Eureka позволяет включить функционал проверки доступности сервисов в реестре посредством обмена healthcheck-сообщениями. По умолчанию данный функционал не используется.

Чтобы включить healthcheck, необходимо:

1. Добавить Eureka-клиент в зависимости проекта.
2. Включить `eureka.client.healthcheck.enabled = true`.
3. Приложение перестало запускаться `Caused by: java.lang.ClassNotFoundException: org.springframework.boot.actuate.health.SimpleStatusAggregator`. Тут все легко: лечится добавлением зависимости на Spring Boot Actuator.

Хитрости: Self-Preservation and renewal

Спустя какое-то время после запуска в интерфейсе Eureka появляется надпись: **EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.**. Попробуем разобраться, что это такое.

Eureka-сервер считает, что для того, чтобы запись в реестре считалась релевантной, зарегистрированный инстанс должен присылать heartbeat-сообщения с определенной периодичностью.

Эта периодичность определяется настройкой `eureka.server.expected-client-renewal-interval-seconds` в Eureka-сервере (по умолчанию имеет значение 30)

Когда периодичность получения heartbeat-сообщений нарушается, Eureka-сервер начинает переживать насчет правомерности наличия этого сервиса в реестре. Спустя время `eureka.instance.lease-expiration-duration-in-seconds` (по умолчанию - 90) от последнего полученного heartbeat сервис начинает считаться кандидатом на удаление.

Когда происходит очередное срабатывание джобы, `eureka.server.eviction-interval-timer-in-ms` (период по умолчанию - одна минута, отсчет идет от старта Eureka-сервера), кандидаты на удаление удаляются из реестра и исчезают из интерфейса eureka-сервера.

Self-preservation - механизм Eureka, который предотвращает удаление инстансов из реестра, даже если heartbeats от них приходят реже положенного. Использованием этого механизма разработчики Spring Cloud предлагают защититься от низкого качества соединения между клиентом и SD-сервером (в то время как соединение между двумя клиентами может быть в порядке).

По умолчанию режим self-preservation активен. Для отключения следует использовать `eureka.server.enable-self-preservation = false`.

Правила перехода в self-preservation регулируются настройками `eureka.server.renewalPercentThreshold` и `eureka.instance.leaseRenewalIntervalInSeconds` и неплохо описаны по ссылкам:

1. https://stackoverflow.com/a/41217806/7757009
2. https://dzone.com/articles/the-mystery-of-eurekas-self-preservation
3. https://www.baeldung.com/eureka-self-preservation-renewal

2. Варианты реализации

В данном разделе мы предлагаем несколько типовых решений, которые планируется использовать для дальнейшего анализа и развития.

PoC 1: API Gateway + SSR + HTTP + SD + UI-service

[Ссылка на репозиторий с демо]

Задача:

1. Показать страницу интерфейса;
2. Обеспечить сборный интерфейс из элементов, предоставляемых разными сервисами.

В соответствии с намеченной архитектурой и требованиями, перед нами вырисовывается следующий набор сервисов:

Имя сервиса	Адрес
service-registry	localhost:8888
api-gateway	localhost:8080
ui-service	localhost:8090
api-hello-service	localhost:8081
api-goodbye-service	localhost:8082

Сконфигурируем API Gateway. Будем направлять запросы /view к UI-сервису, /api - по соответствующим сервисам.

```
@Bean

public RouteLocator customRoutes(RouteLocatorBuilder builder) {

 return builder.routes()

  .route(r -> r

          .path("/view/**")

          .uri("lb://ui-service/"))

  .route(r -> r

          .path("/api/hello/**")

          .uri("lb://api-hello-service/"))
```

```
  .route(r -> r
          .path("/api/goodbye/**")
          .uri("lb://api-goodbye-service/"))
  .build();
}
```

В UI-сервисе определяем контроллер, который будет возвращать по запросам соответствующие вьюшки

```
@Controller
@RequestMapping("view")
public class ViewController {


 @GetMapping("hello_view")
 public String helloView() {
        return "hello_view";
 }


 @GetMapping("goodbye_view")
 public String goodbyeView() {
        return "goodbye_view";
 }
```

```
}
```

Что происходит во время работы

1. В браузере переходим по адресу localhost:8080/view/hello_view;
2. Клиент выполняет запрос о получении одного из view;
3. API Gateway перенаправляет запрос к UI-сервису;
4. UI-сервис возвращает view;
5. В браузере клиента исполняется JS-код, выполняющий запрос к первому или второму API-сервису.
6. API Gateway перенаправляет запрос к API-сервису;
7. API-сервис возвращает ответ в формате JSON;
8. В браузере клиента производится обработка ответа, данные из ответа API отображаются в пользовательском интерфейсе.

Схоже работает и вторая страница, на которую можно перейти по ссылке со страницы 1.

PoC 2: API Gateway + Microfrontends + SD + UI-service

Ссылка на репозиторий с демо

В данном демо мы изобразили принципы работы, лежащие в основе микрофронтендного подхода

Имя сервиса	Адрес
service-registry	localhost:8888
api-gateway	localhost:8080
ui-service	localhost:8090
hello-service	localhost:8081
goodbye-service	localhost:8082

Что происходит во время работы

1. В браузере переходим по адресу localhost:8080/view;
2. Клиент выполняет запрос о получении одного единственного view;
3. API Gateway перенаправляет запрос к UI-сервису;
4. UI-сервис возвращает view;

5. В браузере клиента исполняется JS-код, выполняющий запросы о двух кусках интерфейса;l
6. API Gateway перенаправляет запрос к сервисам hello и goodbye;
7. Сервисы возвращают ответы в формате "строка, подготовленная как часть HTML";
8. В браузере клиента производится обработка ответа, данные из ответа API отображаются в пользовательском интерфейсе.

Какие еще сценарии могут быть опробованы

1. Использование WebSocket;
2. Использование Kafka в backend-сети.

Резюме

В данной статье мы:

1. Рассмотрели ряд вопросов, возникающих при переходе на микросервисную архитектуру в клиент-серверном приложении.
2. Изучили архитектурные концепции, стоящие за возникшими вопросами.
3. Изучили особенности решения возникших вопросов на программном стеке Java + Spring Boot + Spring Cloud, попутно пройдясь по граблям и предложив несколько типовых решений.

Если у вас есть предложения на тему того, как можно улучшить содержание статьи на благо сообщества, просим не стесняться высказываться в комментах.

Что почитать или посмотреть еще

Если хочется поглубже изучить явление микросервисов и то, каким принципам рекомендуется следовать при разработке и эксплуатации систем, соответствующих микросервисной архитектуре:

1. Сэм Ньюмен, "Создание Микросервисов. Есть вариант на русском (перевод от издательства "Питер");
2. Сайт за авторством Chris Richardson с короткими и емкими практическими кейсами использования микросервисов.

Если хочется получить больше понимания об устройстве Spring Cloud:

1. Конечно, официальные документация и гайды Spring Cloud;
2. Официальный YouTube-канал SpringDeveloper;
3. Выступления с конференций и их текстовые расшифровки. Из того, что мелькало на хабре, вспоминается доклад с Joker 2017.

Для тех, кто еще не знаком с Kafka - конечно же, официальные документация и гайды.

Для лучшего изучения темы статьи на смежные темы:

1. В статье приводится практическое описание настройки стека Spring Cloud на основе Kotlin. Помимо описанных в текущей статье технологий, используются Ribbon, Hystrix, OpenFeign, Sleuth, предпочтение отдается конфигурации API gateway на основе файла .properties, включен Spring Security.
2. Рассуждения с высокого уровня и точки зрения процессов (хотя со всеми тезисами согласиться трудно).
3. Статья-предостережение от использования подхода Entity Service.
4. Еще одна статья-предостережение, на этот раз от бездумного использования микросервисов по поводу и без него. Предлагаются взвешенные аргументы, несмотря на категоричный заголовок.

Гайды и руководства:

1. Практический пример использования Eureka для Service Discovery.

Теги:АрхитектураSpringSpring CloudМикросервисы

Хабы:Блог компании МиландрJavaМикросервисы