Санкт-Петербургский государственный университет

Башкиров Александр Андреевич

Выпускная квалификационная работа

Построение диаграмм микросервисов по исходному коду в Intellij IDEA

Уровень образования: бакалавриат

Направление *09.03.04 «Программная инженерия»*

Основная образовательная программа *СВ.5080.2017 «Программная инженерия»*

Научный руководитель:
к.ф.-м.н., доц. Д.В. Луцив

Рецензент:
программист ООО «Интеллиджей Лабс», к.т.н. Н.Н. Митропольский

Санкт-Петербург
2021

Saint Petersburg State University

Alexander Bashkirov

Bachelor's Thesis

Building microservices diagrams by source code in Intellij IDEA

Education level: bachelor

Speciality *09.03.04 «Software Engineering»*

Programme *CB.5080.2017 «Software Engineering»*

Scientific supervisor:
C.Sc., docent D.V. Luciv

Reviewer:
senior software engineer at "Intellij Labs" Co. Ltd., C.Sc. N.N. Mitropolsky

Saint Petersburg
2021

Оглавление

Введение

В последнее десятилетие в контексте развития облачных вычислений одной из наиболее развивающейся и применяемой для распределённых приложений архитектурой стала микросервисная архитектура [9]. Одними из её преимуществ являются независимость разработки и развёртывания отдельных компонентов системы и высокая степень декомпозиции проекта в целом. Одни из крупнейших компаний индустрии, такие как Netflix (свыше 500 микросервисов [11]), Spotify (свыше 800 микросервисов [5]) и Uber (свыше 1000 микросервисов [13]), используют данную архитектуру для разработки своих продуктов [15]. Более формально, микросервисы представляют собой архитектурный стиль, к настоящему моменту насчитывающий уже более 28 подходов к реализации [4].

При таком количестве одновременно существующих и динамически изменяющихся модулей программного кода ощущается острая необходимость в средствах визуализации [8, 19]. Вследствие этого к настоящему моменту было создано множество инструментов визуализации микросервисной архитектуры — классификация представлена на рис. 1.

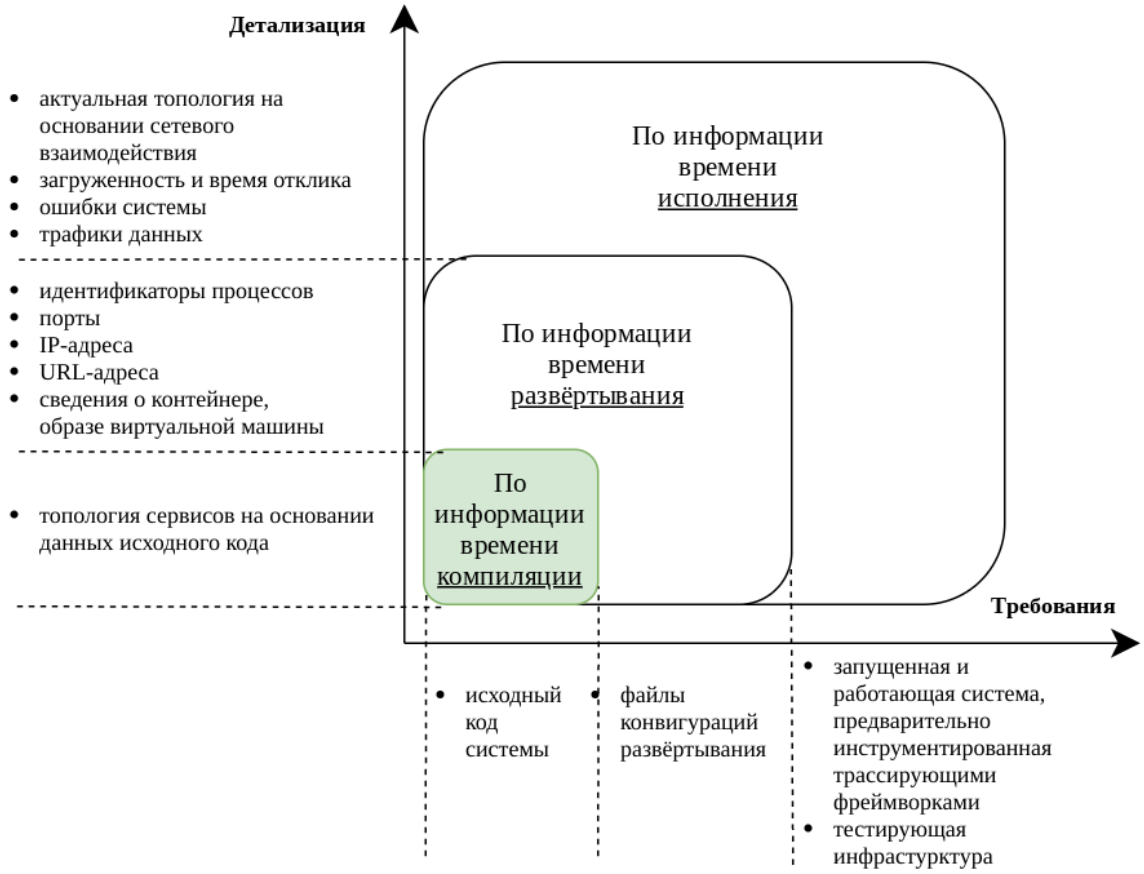


Рис. 1: Классификация средств визуализации микросервисов в соответствии с предъявляемыми требованиями и уровнем детализации

Несмотря на почти стопроцентную детализацию, трассирующие инструменты времени исполнения предъявляют чрезвычайно высокое требование в виде уже готовой к запуску системы и сервиса нагрузочного тестирования, что неприменимо в процессе создания новых сервисов. При этом внесение изменений требует длительного времени на горячую подмену всего сервиса целиком. В свою очередь для отображения подробных сведений времени развёртывания требуется топология общения сервисов, извлечь которую в общем случае возможно только из исходного кода. Поэтому ощущается необходимость в создании быстрых инструментов визуализации графа взаимодействия сервисов по исходному коду, который, как базовый каркас, впоследствии может быть дополнен сведениями о развёртывании из конфигурационных файлов.

Что же касается вручную создаваемых диаграмм, как показывает практика, таковые быстро выходят из синхронизации с активно изменяющимся исходным кодом и подходят для использования только на этапе проектирования системы, что так же свидетельствует в пользу необходимости средств автоматической генерации диаграмм по исходному коду, гарантирующих актуальность отображаемой информации. Наконец, для ускорения продуктивности программиста полезным представляется возможность быстрого переключения между узлами диаграмм и исходным кодом, им соответствующим, что подсказывает о целесообразности встраивания подобных средств в интегрированные среды разработки (далее IDE), как правило уже содержащие средства анализа исходного кода, визуализации данных и навигации по компонентам проекта.

Представленный в данной работе инструмент является частью и расширяет функциональность недавно выпущенного проприетарного плагина Endpoints [6] для IDE IntelliJ IDEA Ultimate [7], использующего уже существующие проприетарные плагины анализа микросервисных фреймворков Spring Boot[1] и Micronaut[2], а также отображения диаграмм[3]. Новая функциональность заключается в возможности генерации диаграммы микросервисов и связей между ними на основании информации из исходного кода их реализации и интерактивного переключения между кодом и диаграммой.

[1] https://www.jetbrains.com/help/idea/spring-boot.html
[2] https://www.jetbrains.com/help/idea/micronaut.html
[3] https://www.jetbrains.com/help/idea/diagrams.html

1. Постановка задачи

Целью работы является расширение проприетарного плагина Endpoints для IntelliJ IDEA Ultimate для предоставления функциональности генерации диаграмм микросервисов для исходных кодов на языках Java и Kotlin, использующих фреймворки Spring Boot и Micronaut. Для её выполнения были поставлены следующие задачи.

- Изучить предметную область и существующие решения.
- Сформулировать требования и ограничения реализации.
- Спроектировать архитектуру реализовываемого компонента.
- Реализовать функциональность генерации диаграмм микросервисов на основании данных исходного кода, отображающих сервисы и возможное взаимодействие между ними посредством
  - HTTP запросов,
  - Websocket протокола,
  - очередей сообщений.
- Выполнить апробацию на проектах с открытым исходным кодом, использующих микросервисную архитектуру.

2. Обзор предметной области

Абстрагируясь от предъявляемых требований и развивая классификацию, представленную на рис. 1, известные на данный момент средства визуализации микросервисной архитектуры можно классифицировать по сферам применимости (см. рис. 2).

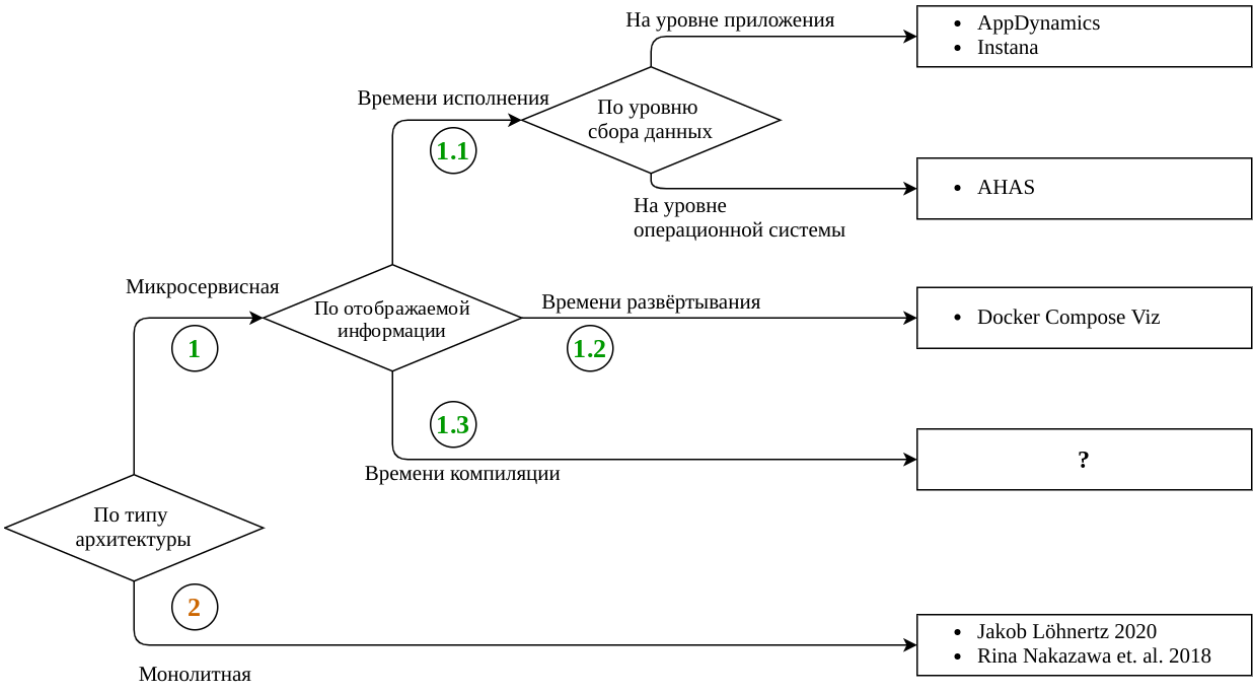


Рис. 2: Классификация средств визуализации микросервисов в соответствии со сферами применимости

Далее мы подробно разберём каждую из представленных групп.

2.1. Визуализация микросервисной архитектуры

Данный раздел соответствует ветви 1 на рис. 2 и описывает существующие средства визуализации для проектов, уже имеющих выполненную и формализованную **декомпозицию проекта на микросервисы**. Для таких инструментов, как уже упоминалось во введении, наблюдается ярко-выраженное разделение по типу используемой (и отображаемой) информации:

- *времени исполнения* — любая информация для получения которой требуется запуск системы (профили и трассы программ, сетевые запросы, сведения операционной системы о процессах и т.д.);
- *времени развёртывания* — конфигурационные файлы развёртывания системы;
- *времени компиляции* — исходные коды проекта.

2.1.1. Визуализация по данным времени исполнения

Наиболее развитыми в контексте средств визуализации микросервисов являются технологии *управления производительностью приложения* [1] (application performance management), анализирующие уже разработанные и запущенные приложения (соответствуют ветви 1.1 на рис. 2). Такие системы в частности позволяют обнаруживать, а затем в реальном времени следить за состоянием и загрузкой сервисов, а также трафиками данных между ними. Трассировка возможна на нескольких уровнях описанных далее.

- *На уровне приложения*, как это применяется, например, в системах AppDynamics[1] и Instana[2] с использованием собственных проприетарных компонентов, либо фреймворков распределённых систем трассировки с открытым кодом, таких как, например, Zipkin[3] и Jaegear[4] проекта OpenTracing[5].

- *На системном уровне*, агрегируя данные о процессах и сетевом взаимодействии, что применяется, например, в системе AlibabaCloud AHAS [18]. Также существуют компоненты с открытым исходным кодом, например Nagious[6].

Упомянутые системы имеют мощные средства визуализации в режиме реального времени — пример обзорной панели от AppDynamics представлен на рис. 3. На панели поддерживается отображение следующих данных:

1. топология *эндпоинтов* (сервисов, баз данных, очередей сообщений, распределённых кешей и др.);

2. информация о развёртывании (идентификаторы процессов, порты, IP-адреса, URL-адреса, сведения о контейнере, образе виртуальной машины и др.);

3. загруженность и время ответа;

4. ошибки системы (аварийное завершение процессов эндпоинтов, ответы на HTTP-запросы с кодами 5** и др.);

5. трафики данных.

---

[1] https://www.appdynamics.com/
[2] https://www.instana.com/
[3] https://zipkin.io/
[4] https://www.jaegertracing.io/
[5] https://opentracing.io/
[6] https://www.nagios.org/
[7] https://www.appdynamics.com/blog/product/how-ing-gains-visibility-into-their-complex-distributed-environment/

Рис. 3: Пример обзорной панели от AppDynamics[7]

Диаграммы, построенные таким образом, могут с успехом применяться для изучения тонкостей работы системы. Однако, как уже упоминалось, такой подход имеет значительное ограничение — для получения данных **необходим запуск всей или хотя бы части системы** на некоторой тестовой инфраструктуре с симуляцией запросов (топология не может быть построена без изучения трафика). Запуск и развертывание такой тестовой конфигурации потребуют либо выделенной вычислительной инфраструктуры, либо значительных ресурсов при использовании только одной машины.

2.1.2. Визуализация по данным времени развёртывания

Диаграммы на основании данных времени компиляции и развертывания (соответствуют ветвям 1.2 и 1.3 на рис. 2) могут строиться на порядок быстрее и непрерывно обновляться всё время разработки, не требуя особых вычислительных ресурсов. Вдобавок при встраивании таких средств в IDE потенциально становятся доступны уникальные для IDE функции (такие как навигация, рефакторинг и др.), что дополнительно повышает производительность программиста.

Особенностью данных времени развёртывания является наличие информации о развертывании сервисов и зависимостей между ними. Такия сведения, как правило, представлены в файлах конфигурации развертывания, например, в Docker Compose[8] `docker-compose.yml` или Kubernetes[9] `deployment.yml`, а также в файлах спецификаций Swagger OpenApi.

[8] https://docs.docker.com/compose/
[9] https://kubernetes.io/

Для Docker Compose существует небольшой инструмент[10], непосредственно визуализирующий файл спецификации `docker-compose.yml` — пример представлен на рис. 4.

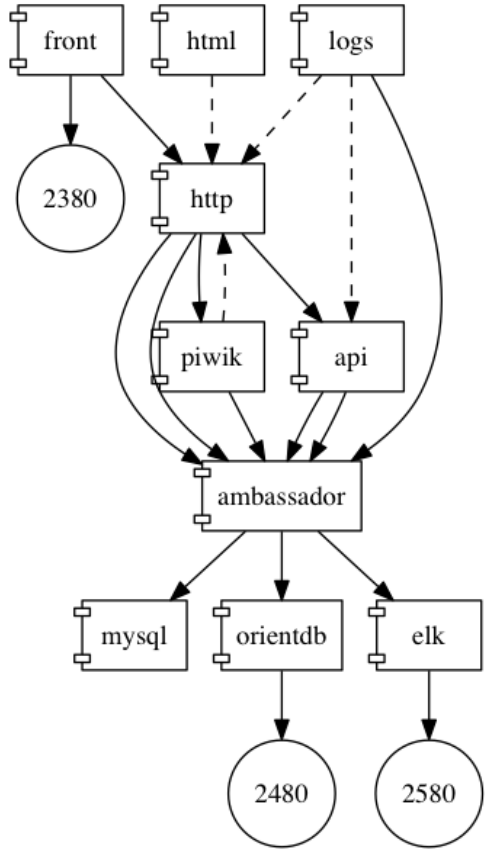


Рис. 4: Пример визуализации `docker-compose.yml` файла

Отметим, однако, что топологии времени компиляции и развёртывания в общем случае не сравнимы и в некотором смысле соотносятся как бизнес-логика и вариант её физической реализации.

- Бинарная сборка одного и того же микросервиса может быть развёрнута на нескольких вычислительных узлах, и информация об этом неизвестна во время компиляции.[11].
- Топология зависимостей времени развёртывания не обладает достаточным уровнем детализации, отображая лишь возможность взаимодействия, а не полный список возможных соединений с привязками к исходному коду.

[10] https://github.com/pmsipilot/docker-compose-viz

[11] А в случае использования балансировщика нагрузки эта информация будет доступна и вовсе только во время исполнения

Для получения максимальной детализации представляется полезным объединить оба множества сведений в одну диаграмму с максимально подробной топологией физических узлов, отображающей также все возможные виды микросервисного взаимодействия, которые удастся извлечь из исходного кода.

2.1.3. Визуализация по данным времени компиляции

Данные времени компиляции являются основополагающим но наиболее узким из рассматриваемых подмножеством данных, т.е. **не вся информация доступна** на этом этапе. В частности, возникают трудности *идентифицирования сервисов по исходному коду*: как правило исходный код сервиса не содержит никакой идентифицирующей его в сети информации. Действительно, URL-адрес, IP-адрес, порт, сведения о контейнере или образе виртуальной машины — все это данные времени развёртывания, от которых реализация сервиса абстрагирована. Поэтому для надежного установления факта меж-микросервисного взаимодействия остаётся возможным лишь опираться на знание об используемом в реализации фреймворке и конкретном способе сетевого общения.

Так существует множество способов взаимодействия микросервисов между собой:

- с помощью HTTP/Websocket протоколов на представленные REST [14] API,
- посредством очереди/брокера сообщений,
- посредством Remote Procedure Call фреймворвоков (таких как gRPC[12]).

В рамках данной работы мы **не** рассматриваем RPC-взаимодействия, оставляя это открытой задачей для реализации. В таком случае мы можем идентифицировать

- сервисы по **Path**-сегментам REST API,
- очереди сообщений по ***корзинам*** сообщений, например, темы (topics) для Apache Kafka[13] и обменные пункты (exchanges) для RabbitMQ[14].

Реализуемый REST API в исходном коде представлен в терминах используемого фреймворка. Так для фреймворка Spring это, к примеру, аннотации REST контроллера[15] (на листинге 1 представлен пример кода, использующего данные аннотации); для фреймворка Micronaut это собственные Routing аннотации[16].

[12] https://grpc.io/
[13] https://kafka.apache.org/
[14] https://www.rabbitmq.com/
[15] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html
[16] https://docs.micronaut.io/latest/guide/index.html#routing

```java
@RestController
@RequestMapping("/greetings")
public class SpringRestController {

    @GetMapping(value    = "/{name}",
                produces = MediaType.TEXT_PLAIN_VALUE)
    public ResponseEntity<?> greeting(@PathVariable String name) {
        return new ResponseEntity<>("Hello " + name, HttpStatus.OK);
    }
}
```

Листинг 1: Пример простейшего сервиса в терминах Spring REST контроллера

В этом случае факт взаимодействия в исходном коде представлен в качестве HTTP-запроса на полный URL-адрес (пример представлен на листинге 2). При этом до этапа развёртывания не известно соотношение между реализациями сервисов и URL-адресами, и остаётся только полагаться на Path-сегменты URL-адреса. Отсюда вытекает ограничение 2.1.

```java
public class Application {
    public static void main(String args[]) {
        final var greeting = new RestTemplate()
          .getForObject(
            "http://some.domain.name/greetings/{name}",
            String.class,
            "Alexander");
        System.out.println(greeting);
    }
}
```

Листинг 2: Пример запроса к микросервису на листинге 1 в терминах Spring REST клиента

Ограничение 2.1 (Идентификация сервиса)**.**

Определим $Paths(S)$ как множество обслуживаемых путей REST API сервиса S. Пусть существует путь REST API p такой, что

$$\exists\ S_1, \ldots, S_n : p \in \bigcap_{i=1}^{n} Paths(S_i) \quad \wedge \quad (\nexists\ S' : \forall i \in 1..n\ S' \neq S_i \ \wedge\ p \in Paths(S'))$$

Тогда во время компиляции для пути p невозможно достоверно определить адресата с точностью до $S_1, \ldots, S_n$.

Информация о реализуемом REST API может быть представлена в фреймворк-

независимом ключе в Swagger OpenApi-спецификации[17], однако, к сожалению, нет общепринятого способа сопоставления спецификации с исходным кодом. Поэтому IDE также приходится искать нужный файл спецификации по Path-сегментам из исходного кода, по-прежнему используя знание о фреймворке.

Наконец, по очевидным соображениям вопрос построения полностью достоверной топологии во время компиляции неразрешим ввиду необходимости анализа исходного кода. В частности, серьёзным фактором, скрывающим топологию, является наличие в системе **динамически формируемых запросов**, для которых в общем случае не удаётся определить микросервис, к которому данный запрос придёт во время исполнения. Таким образом имеем следующее ограничение 2.2.

Ограничение 2.2 (Динамически формируемые запросы)**.**

Динамически формируемые запросы, для которых на этапе компиляции неизвестно, по меньшей мере, множество возможных Path-частей, не могут быть отображены на диаграмме времени компиляции.

С интересующей нас точки зрения взаимодействие с очередью сообщений устроено абсолютно схожим образом, лишь вместо URL-адресов используется более простое имя корзины сообщений (как правило темы, или обменного пункта, специфичного для RabbitMQ). Необходимо отметить так же ограничение 2.3, проиллюстрированное на рис. 5.

Ограничение 2.3 (Идентификация очередей сообщений)**.**

В общем случае[18] *на основании одно только исходного кода и без анализа конфигурационных файлов не удаётся определить факт использования в проекте нескольких физически различных экземпляров очереди сообщений одного вендора. Анализатор будет считать, что все упоминаемые корзины сообщений принадлежат одному и только одному экземпляру.*

Что же касается существующих инструментов, нам не удалось найти ни одной реализации, визуализирующей или хотя бы предоставляющей подобную информацию. Существуют лишь небольшие скрипты[19], позволяющие извлечь упоминаемые в исходном коде URL-адреса, правда, без привязок к исходному коду и подразделению на слушаемые / запрашиваемые адреса.

2.2. Автоматическая декомпозиция монолитной архитектуры

Для создания новых информационных систем предлагаются два конкурирующих подхода: *монолит-ориентированный* [3] и *сервис-ориентированный* [16]. Первый предпо-

[17] https://swagger.io/specification/

[18] Например, для связки Micronaut и Apache Kafka

[19] Например, https://github.com/GerbenJavado/LinkFinder

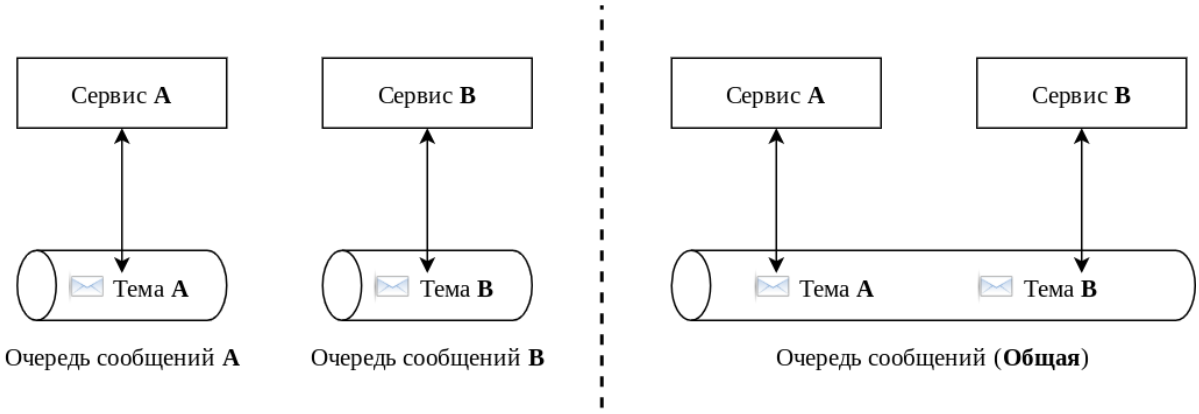


Рис. 5: Пример проявления ограничения 2.3

лагает начинать разработку проекта в виде единого модуля кода (монолита), развёртываемого на одной машине, который впоследствии только на определённом этапе разработки будет разбит на микросервисы; второй, напротив, предполагает изначально ещё на этапе проектирования декомпозировать систему на микросервисы. Для первого, весьма популярного подхода ощущается наибольшая необходимость в визуальном представлении информации ввиду столь глобального перепроектирования.

В последние годы было разработано несколько инструментов [10, 17], позволяющих выполнить анализ монолитной архитектуры, предложить разбиение проекта на микросервисы по ряду альтернативных метрик и визуализировать полученные результаты.

К текущему моменту в литературе предлагалось 4 способа проведения анализа архитектуры.

- *По контекстам вызовов, извлечённых из исходного кода* — по телу реализации метода известно, какие методы он вызывает, это далее обобщается на классы, пакеты и т.д.
- *По контекстам вызовов из данных профилировщика* — аналогично предыдущему пункту, но использует информацию о вызовах времени исполнения.
- *По семантике наименований в исходном коде* — на именованиях сущностей выполняется латентно-семантический анализ.
- *По коммитам в системе контроля версий* — утверждается, что часто близкие с точки зрения бизнес-логики сущности изменяются в исходном коде одновременно.

Результатом проведённых действий служит граф (возможно взвешенный), узлами которого являются сущности исходного кода (методы или классы в терминах

объектно-ориентированного программирования), а рёбра (возможно ориентированные, но как правило эта информация далее игнорируется) отображают извлечённые взаимодействия.

Далее на полученном графе можно запускать различные алгоритмы кластеризации и, наконец, визуализировать результаты, где каждый кластер окрашен в собственный цвет как в примере из [10] на рис. 6.

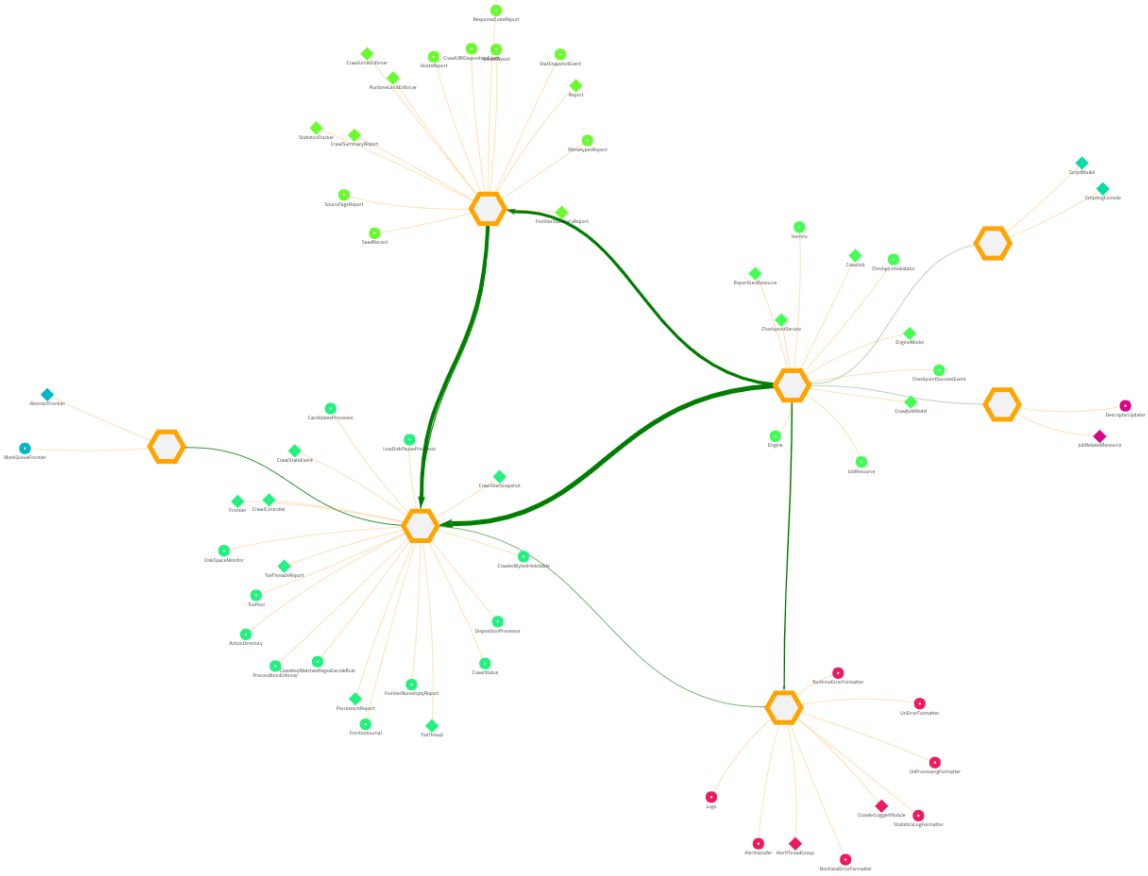


Рис. 6: Пример результатов анализа по семантике наименований и кластеризации *Clauset-Newman-Moore* [2] от [10] на проекте Heritix[20]

[1] https://github.com/internetarchive/heritrix3

3. Требования и ограничения

3.1. Функциональные требования

Разрабатываемое расширение должно удовлетворять следующим требованиям по предоставляемой функциональности:

- Плагин должен содержать следующие *действия*[1] (actions) по генерации диаграмм.
  - Визуализация всех сервисов и используемых очередей сообщений в проекте (*Show Webservices Requests...*), доступное для вызова из окна Endpoints toolwindow [6] как показано на рис. 7.

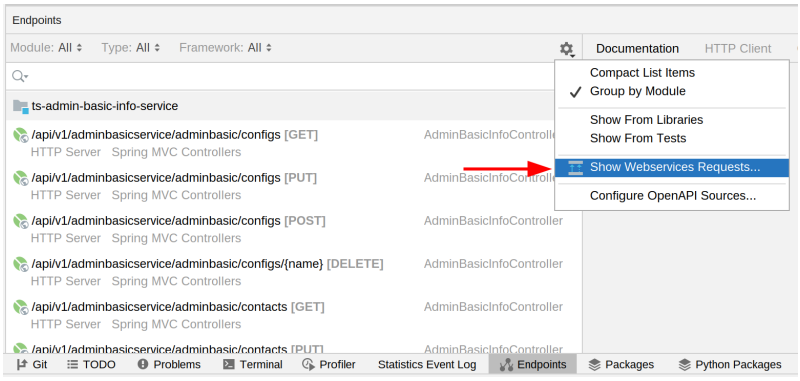


Рис. 7: Пример места вызова общей диаграммы микросервисов

  - Визуализация взаимодействия с конкретной очередью сообщений (*Show message queue connections...*), доступное в качестве пункта gutter[2] иконки вместе с другими действиями (такими как поиск мест использования) для очередей сообщений — пример представлен на рис. 8.

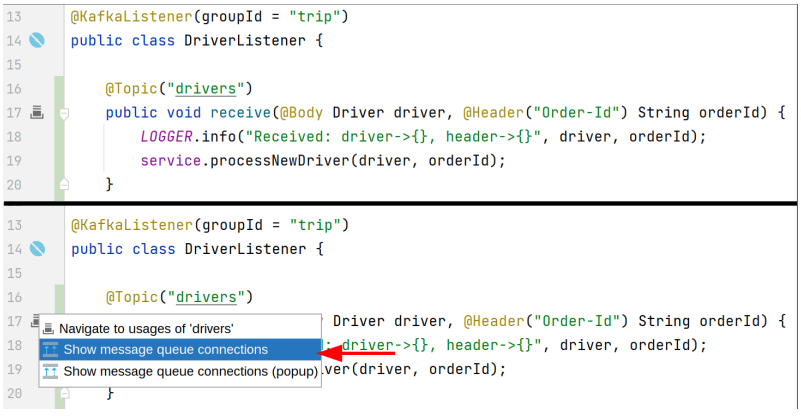


Рис. 8: Пример места вызова диаграммы запросов к очереди сообщений

[1] https://www.jetbrains.com/help/idea/searching-everywhere.html#search_actions
[2] https://www.jetbrains.com/help/idea/settings-gutter-icons.html

- Генерируемая диаграмма должна содержать следующие узлы:
  - микросервисы бизнес-логики, представленные как *модули* в терминах проектной модели IntelliJ Platform[1];
  - очереди сообщений, состоящие из корзин сообщений, сгруппированных по вендорам.
- Генерируемая диаграмма должна содержать следующие рёбра:
  - одно ребро между двумя микросервисами, группирующее все найденные варианты взаимодействий между ними;
  - направленное ребро с меткой темы от микросервиса S к очереди сообщений MQ, если S является издателем для MQ;
  - направленное ребро с меткой темы от очереди сообщений MQ к микросервису S, если S является подписчиком для MQ.
- Должны поддерживаться следующие функции навигации от диаграммы к исходному коду:
  - От модуля к его положению в Project View[2];
  - От ребра к соответствующим запросам в исходном коде с помощью Find Toolwindow[3].

3.2. Нефункциональные требования

Разрабатываемый расширение должно удовлетворять следующим требованиям по реализации:

- Плагин должен *в равной степени* предоставлять функциональность для проектов написанных с применением любых из следующих технологий:
  - языков программирования Java и/или Kotlin;
  - фреймворков для определения микросервисов Spring Boot или Micronaut;
  - очередей сообщений JMS, RabbitMQ[4] и/или Apache Kafka[5];
  - HTTP-клиентов Spring Rest Client, JDK 11 HTTP Client[6], Android Volley[7] и/или OkHttp[8].

[1] https://plugins.jetbrains.com/docs/intellij/module.html
[2] https://www.jetbrains.com/help/idea/project-tool-window.html
[3] https://www.jetbrains.com/help/idea/find-tool-window.html
[4] https://www.rabbitmq.com/jms-client.html
[5] https://kafka.apache.org/
[6] https://docs.oracle.com/en/java/javase/11/docs/api/java.net.http/java/net/http/HttpClient.html
[7] https://developer.android.com/training/volley
[8] https://square.github.io/okhttp/

Данное требование *индуцировано существующими возможностями* IntelliJ IDEA Ultimate по анализу промышленных фреймворков и технологий.

- Для представления диаграмм требуется использовать существующее в IntelliJ IDEA Ultimate API на базе проприетарной библиотеки yFiles for Java[9] версии 2.13.

3.3. Ограничения

Разрабатываемый плагин будет подчиняться следующим ограничениям.

- Плагин предназначен для IntelliJ IDEA Ultimate версии не ниже 2021.1.
- Описанные нефункциональные требования 3.2 в естественном ключе являются и соответствующими ограничениями: плагин будет поддерживать только перечисленные технологии и только указанного API для отображения диаграммы.
- Плагин подчиняется ограничению 2.1: для пересекающихся REST API взаимодействия будут отображены со всеми сервисами из пересечения.
- Плагин подчиняется ограничению 2.2: формально, в качестве анализируемых HTTP/Websocket-соединений заявлены только те, Path-части которых определяются с применением константных строк, статических неизменяемых (`final`) полей строкового типа и произвольных конкатенаций (и строковых интерполяций в случае использования языка Kotlin) предыдущих.
- Плагин подчиняется ограничению 2.3: корзины сообщений группируются по вендорам вне зависимости от возможной схемы развёртывания по нескольким экземплярам.
- Плагин **не** учитывает любую информацию времени развёртывания: Swagger OpenApi, Docker Compose, Kubernetes и иные спецификации игнорируются.
- Плагин **не** поддерживает отображение баз данных и взаимодействий с ними.
- Плагин **не** поддерживает поиск и отображение RPC взаимодействий.

[9] https://www.yworks.com/products/yfiles-for-java-2.x

4. Архитектура проекта

В данной секции предлагается рассмотреть архитектуру разрабатываемого компонента и его взаимодействие с существующими плагинами и API IntelliJ IDEA.

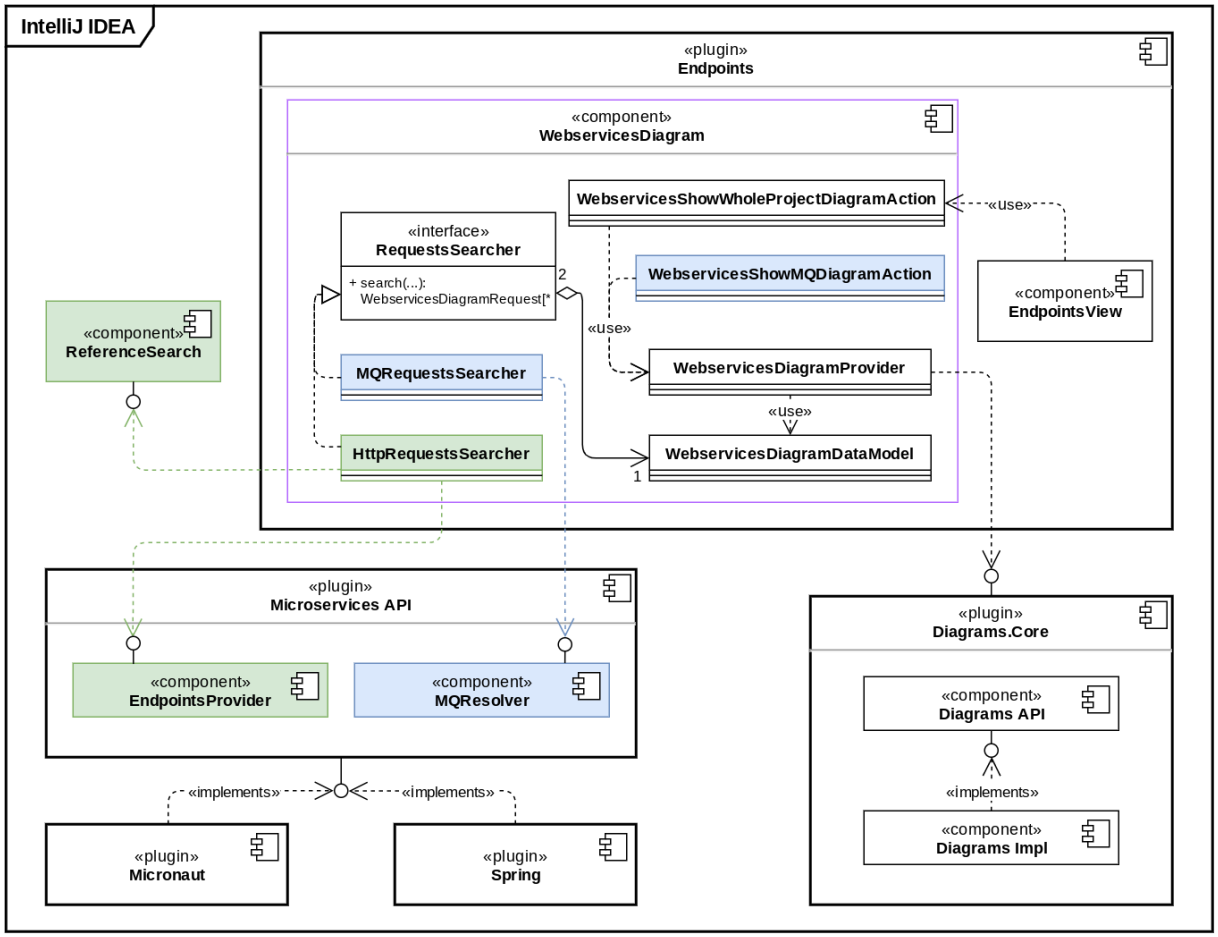


Рис. 9: Основная архитектура проекта

На рис. 9 представлены основные компоненты и классы реализации. Зелёным цветом отмечены компоненты необходимые для поиска взаимодействий микросервисов между собой посредством HTTP/Websocket протокола, синим — с очередью сообщений. Плагины `Diagrams.Core` и `Spring` реализованы на языке Java, остальные компоненты, включая реализовываемый — на языке Kotlin. В ходе работы главным образом был создан компонент `WebservicesDiagram` (выделен фиолетовым цветом), полагающийся на уже существующие плагины, компоненты и API кратко описанные далее.

- `Endpoints` плагин, предоставляющий богатые возможности поиска и просмотра всех объявленных в проекте эндпоинтов, т.е. предоставляемых сервисов точек общения с ним. Так в случае HTTP взаимодействий это множество слушаемых URL-путей[1]. Пока не поддерживается, но в скором времени планируется к реализации отображение соединений с очередями сообщений.

[1]К примеру объявленных с помощью `@org.springframework.web.bind.annotation.GetMapping`

- `Microservices API` платформы для поиска и анализа интересующих мест в коде, в частности списка эндпоинтов с помощью `EndpointsProvider` и обращений к очереди сообщений с помощью `MQResolver`.
- Плагины, предоставляющие реализации упомянутого `Microservices API` для конкретных фреймворков, такие как `Spring Boot` и `Micronaut`. Они содержат кофигурации поиска интересующих аннотаций и вызовов метода, например
  - `@org.springframework.web.bind.annotation.PostMapping`,
  - `@io.micronaut.configuration.kafka.annotation.KafkaListener`,
  - `org.springframework.jms.core.JmsTemplate#sendAndReceive()`,
  - и т.д.
- `Diagrams.Core`, непосредственно предоставляющий функциональность отображения диаграмм и взаимодействия с ними. В частности `WebservicesDiagramProvider` реализовывает определённую в нём точку расширения (extension point[2]) `DiagramProvider`.

Указанные компоненты участвуют в процессе генерации диаграммы как показано на рис. 10.

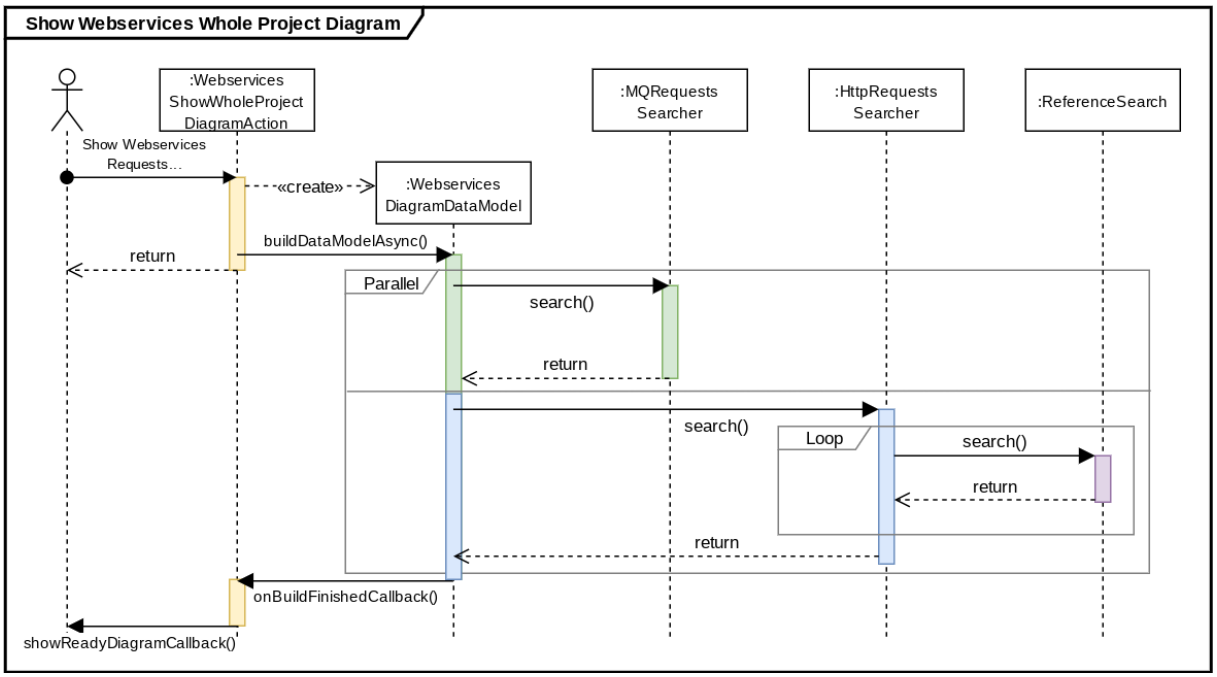


Рис. 10: Визуализации процесса генерации диаграммы взаимодействия всех микросервисов в проекте

Сам процесс является довольно естественным и линейным, и кратко описан далее.

[2] https://plugins.jetbrains.com/docs/intellij/plugin-extension-points.html

1. Из `EndpointsView` приходит запрос пользователя на генерацию диаграммы.

2. Инициируется *асинхронное* создание модели данных `WebservicesDiagramDataModel`. Поиск запросов по всему проекту является довольно тяжеловесной операцией даже при использовании мощной системы индексов[3] IntelliJ Platform, поэтому длительное построение модели данных диаграммы прячется под *строку состояния* (progress bar) и не мешает пользователю продолжать работать в IDE во время сбора данных.

3. Далее происходит поиск непосредственно самих взаимодействий в коде. При этом, к примеру, поиск мест общения с очередью сообщений и HTTP/Websocket-запросов независимы, а значит могут исполняться физически параллельно.

4. Наконец, по завершении всех поисков, на основании построенной модели данных создаются узлы и рёбра графа, и результат отражается в новой вкладке редактора.

Ясно, что ввиду использования прослойки из `Microservices API`, любой модуль или плагин, её реализующий, будет автоматически поддерживаться диаграммой, т.е. будут найдены и отображены соответствующие взаимодействия. Что же касается иных типов взаимодействий (с базами данных, посредством RPC-фреймворков и др.), то для их поддержки достаточно лишь реализовать простейший интерфейс `RequestsSearcher`, состоящий из одного метода `search`.

[3] https://plugins.jetbrains.com/docs/intellij/indexing-and-psi-stubs.html

5. Особенности реализации

5.1. Поиск HTTP/Websocket взаимодействий

Для начала стоит отметить, что поскольку, как было указано в требованиях (параграф 3.2), необходимо равно поддерживать программы как на языке Java, так и на языке Kotlin, оперирование исходным кодом напрямую не представляется удачным решением, поскольку это бы требовало дублирования всей функциональности для каждого из языков. Во избежание данной ситуации для анализа используется специальное высокоуровневое представление исходного кода Unified Abstract Syntax Tree (UAST)[1], способное выразить с достаточной точностью код как на языке Java, так и на языке Kotlin. Все упомянутые далее структуры и компоненты оперирует именно этим представлением.

Поиск взаимодействий посредством REST API подразделяется на 2 этапа.

1. Поиск слушаемых каждым сервисом эндпоинтов.

   Этот этап уже был реализован при создании `Endpoints` плагина: можно запросить список всех эндпоинтов в проекте или для каждого модуля в отдельности посредством `EndpointsProvider`.

2. Поиск мест использования слушаемого URL-пути для каждого сервиса.

   Данный этап так же уже был реализован для как действие поиска мест использования URL-адресов (Find URL reference usages action).

Здесь важно сделать замечание о технической реализации второго этапа. Общая схема алгоритма изложена на рис. 11.

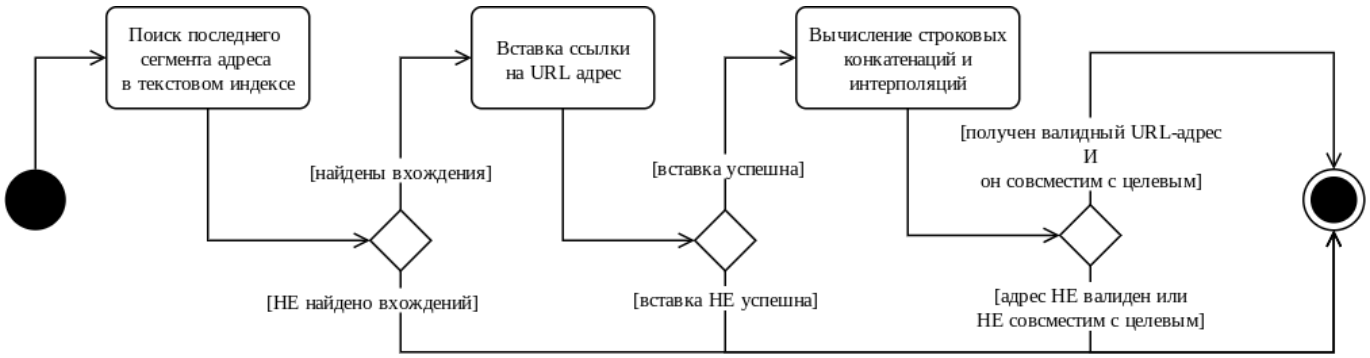


Рис. 11: Алгоритм поиска использований URL-пути

Опуская технические подробности, рассмотрим как происходит поиск мест использования URL-пути `my.domain.com/catalog/path`.

[1] https://plugins.jetbrains.com/docs/intellij/uast.html

1. В текстовом индексе проекта ищутся *окончания* (последний сегмент) адреса, т.е. `path`.

2. В найденные места пытаются встроить ссылку (reference inject) специального вида `UrlPathReference`. При этом происходят следующие действия.

   - Статически вычисляются все строковые конкатенации и интерполяции в данном выражении, подставляя значения всех используемых константных переменных.
   - Если в результате получается валидный URL-адрес с указанным окончанием `path` и он совместим с префиксом `my.domain.com/catalog`, то данное выражение считается местом использования, в противном случае отбрасывается.

Отсюда необходимо заключить следующее техническое ограничение реализации.

Ограничение 5.1 (Строковые переменных для хранения URL окончаний)**.**
Места использования URL-адреса, в которых само окончание объявлено ***не*** *литералом (а, например, с помощью неизменяемого строкового поля, т.е. константы)* ***не детектируются*** *существующим алгоритмом поиска.*

Это довольно серьёзное с точки зрения практической применимости ограничение и сейчас программистами IntelliJ Platform ведётся разработка механизма вставки ссылок не только по самим строковым литералам, но и по местам их использования в случае хранения последних в переменных (references-by-usages injection). Однако на текущий момент эта технология ещё не готова к применению.

Типы отображаемых взаимодействий представлены в таблице 1, а на рис. 12, в частности, пример визуализации HTTP-взаимодействий.

Тип	**Описание**	**Ребро на диаграмме**
HTTP	HTTP запросы	
Websocket	Общение по Websocket протоколу	

Таблица 1: Типы меж-микросервисвного взаимодействия посредством REST API

5.2. Поиск обращений к очереди сообщений

Алгоритм поиска обращений к очередям сообщений во многом похож на уже описанный вариант для детектирования HTTP-взаимодействий с оговоркой, что, как уже упоминалось в параграфе 2.1.3, различные экземпляры очередей сообщений одного вендора не отслеживаются и собираются в одну группу, а следовательно, не требуется

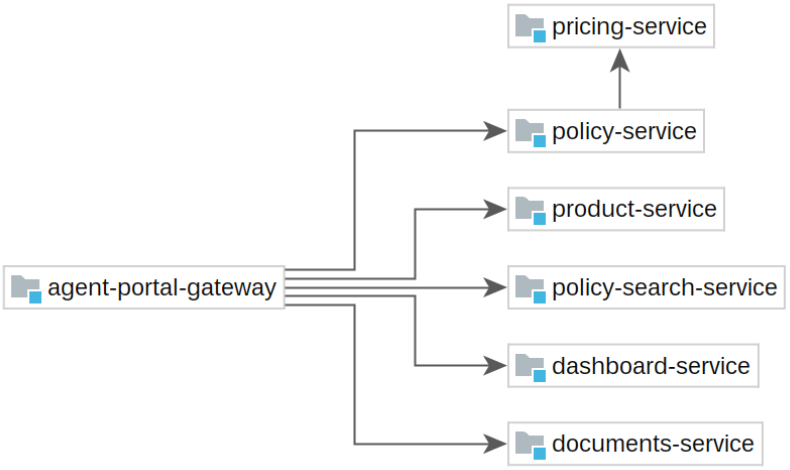


Рис. 12: Пример визуализации взаимодействия микросервисов по HTTP-протоколу

этапа поиска слушаемых эндпоинтов (т.е, в терминах очередей сообщений, множеств обслуживаемых корзин сообщений для каждого экземпляра), достаточно сразу детектировать обращения к корзинам и группировать последние по установленному вендору.

Взаимодействия с очередью сообщений подразделяются на несколько устоявшихся типов (перечисленных в таблице 2), имеющих одинаковую семантику у разных вендоров.

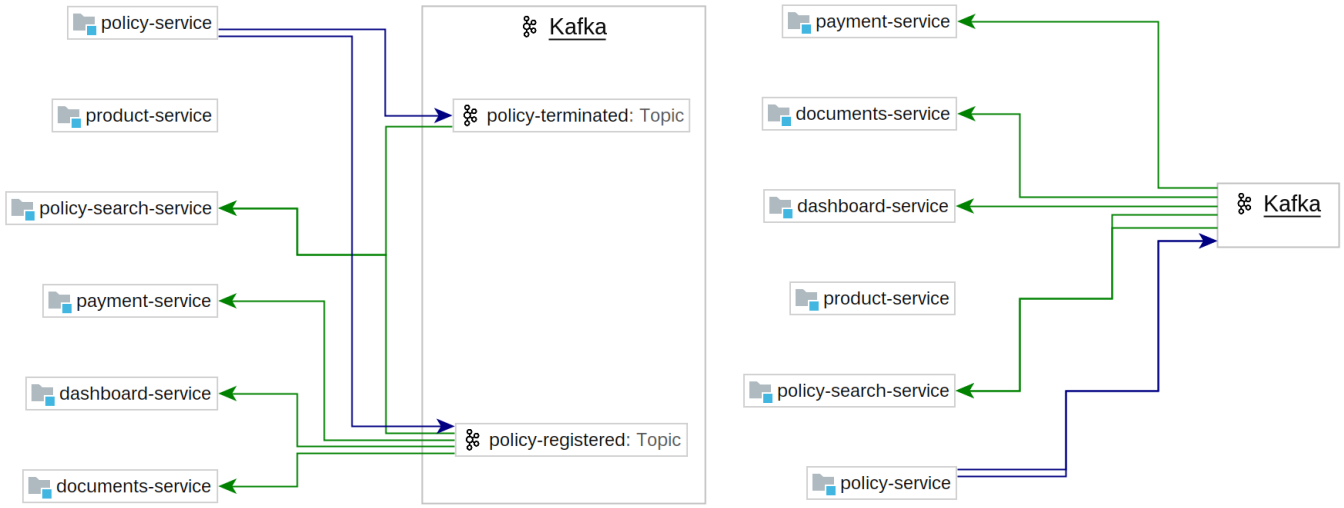
Тип	**Описание**	**Ребро на диаграмме**
Administration	Административные сервисные запросы	Service ⇢ MQueue
Receive	Чтение данных по теме	Service ← MQueue
Send	Запись данных в тему	Service → MQueue
SendAndReceive	RPC-семантика, запись с последующим блокирующим чтением	Service ↔ MQueue
Stream Forwarding	Потоковый обработчик, пересылающий данные при получении	Service ↔ MQueue
Unknown	Все остальные неразделяемые взаимодействия	Service ⇠⇢ MQueue

Таблица 2: Типы взаимодействий с очередью сообщений

Функциональность поиска в исходном коде обращений к очереди сообщений уже была реализована на момент начала работы в компоненте `MQProvider`. Однако найденные связи не разделялись по семантике, и такая дифференциация на указанные типы была добавлена в ходе данной работы.

Пример визуализации с `Send` и `Receive` типами взаимодействий представлен на

рис. 13.



(a) Узел очереди сообщений раскрыт

(b) Узел очереди сообщений сжат

Рис. 13: Пример визуализации взаимодействия микросервисов с очередью сообщений

5.3. Интерактивное взаимодействие с диаграммой

Крайне важной функциональностью при взаимодействии с любой диаграммой, сгенерированной по исходному коду, является просмотр всех мест в исходном коде, на основании которых была установлена та или иная связь — знать, что компоненты связаны, но не знать, как именно, сильно сужает область применимости диаграммы.

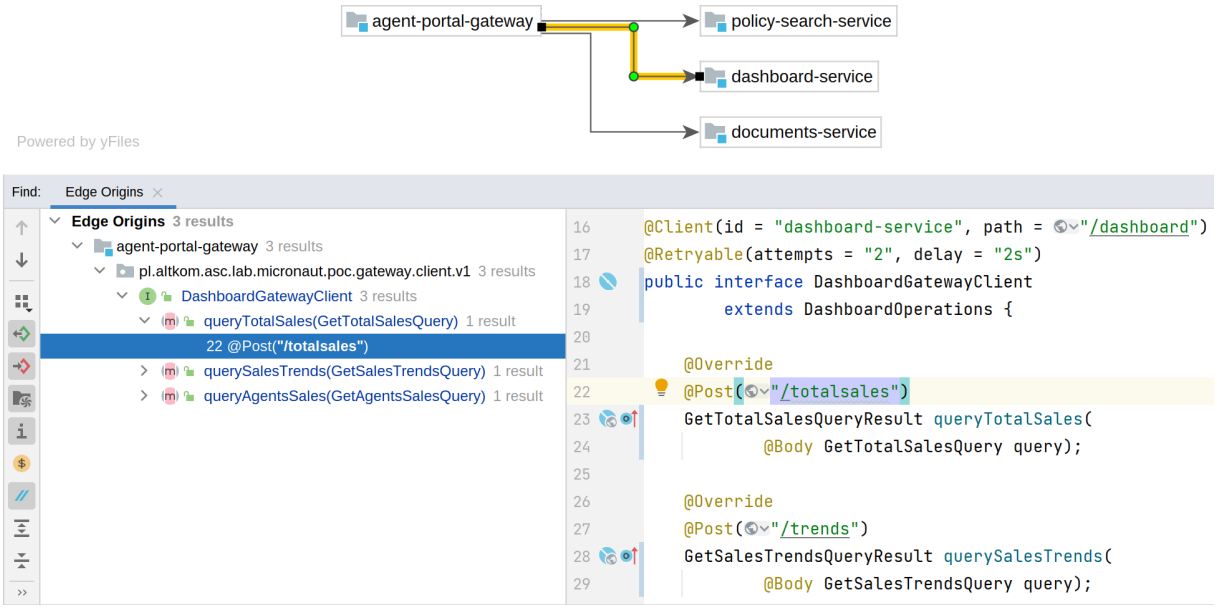


Рис. 14: Пример использования окна поиска для представления взаимодействий микросервисов в исходном коде

Для представления мест в исходном коде используется существующий, специально для этого предназначенный компонент IntelliJ Platform — *окно поиска* (Find tool window). По двойному клику на ребро открывается окно поиска со всеми взаимодействиями соответствующих микросервисов. Оно автоматически предоставляет мощную функциональность фильтрации, группировки и сортировки представляемых позиций.

Пример отображения мест определения клиентских HTTP запросов для обращения сервисом `agent-portal-gateway` к сервису `dashboard-service` представлен на рис. 14.

6. Апробация и анализ результатов

6.1. Апробация на проектах с открытым исходным кодом

Для проведения апробации были выбран ряд проектов из [12], написанных на языке Java с применением Spring фреймворка, с наибольшим числом сервисов и нетривиальной топологией. Соответствующие проекты перечислены в таблице 3.

Имя проекта	**Ссылка**	**Число сервисов**
Train Ticket	https://github.com/FudanSELab/train-ticket/	> 50
FTGO	https://github.com/microservices-patterns/ftgo-application	6
Apollo	https://github.com/ctripcorp/apollo	6
Genie	https://github.com/Netflix/genie	4

Таблица 3: Список Spring проектов отобранных для апробации

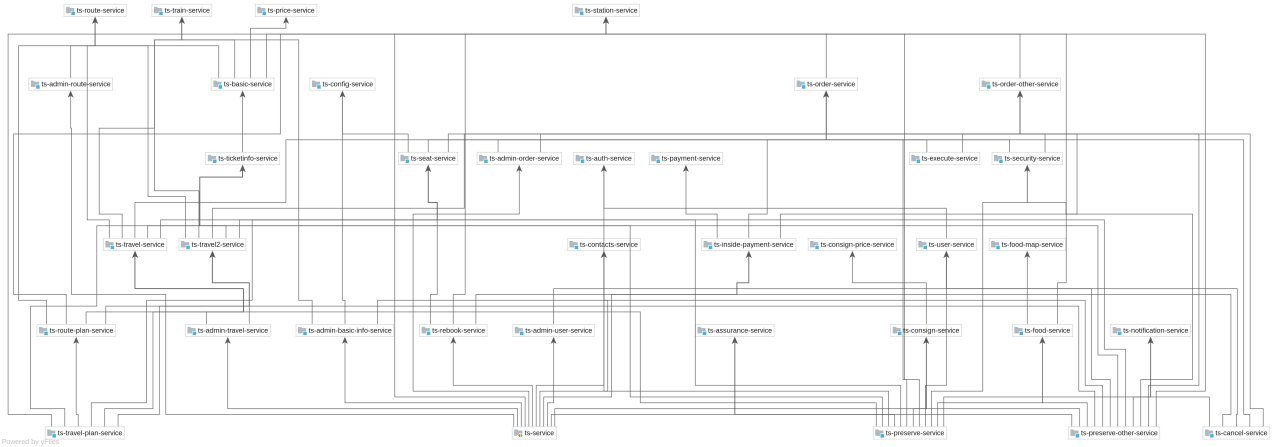
Также для проведения апробации был отобран ряд открытых проектов (перечисленных в таблице 4) на базе фреймворка Micronaut.

Имя проекта	**Ссылка**	**Число сервисов**
Asc-Lab POC	https://github.com/asc-lab/micronaut-microservices-poc	10
Piomin Kafka	https://github.com/piomin/sample-kafka-micronaut-microservices	4
Jexp Meetup	https://github.com/jexp/meetup-recommendations-micronaut	4

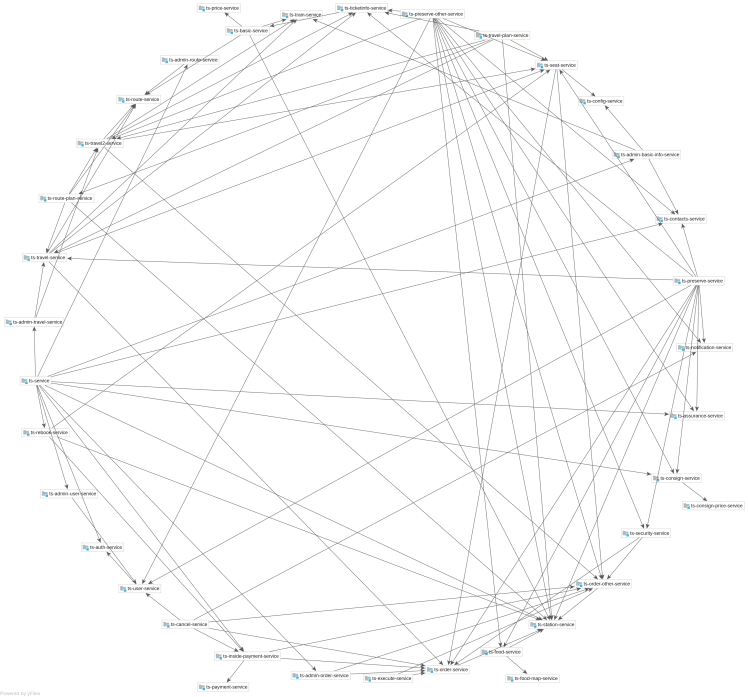
Таблица 4: Список Micronaut проектов отобранных для апробации

Наиболее видимых и значимых результатов, изображённых на рис. 15, удалось добиться для проекта Train Ticket в силу следующих особенностей.

- Большое число сервисов с нетривиальной топологией наглядным образом подтверждают практическую пользу и даже необходимость средств визуализации. Даже сама представленная диаграмма весьма сложна для восприятия, а при её отсутствии навигация и ориентирование в таком проекте по рассредоточенным по большому объёму кода связям представляют собой ощутимые трудности.
- В большинстве случаев в коде указывался полный целевой URL адрес HTTP-запросов, что является наипростейшим для поиска случаем.



(a) Иерархическая раскладка узлов

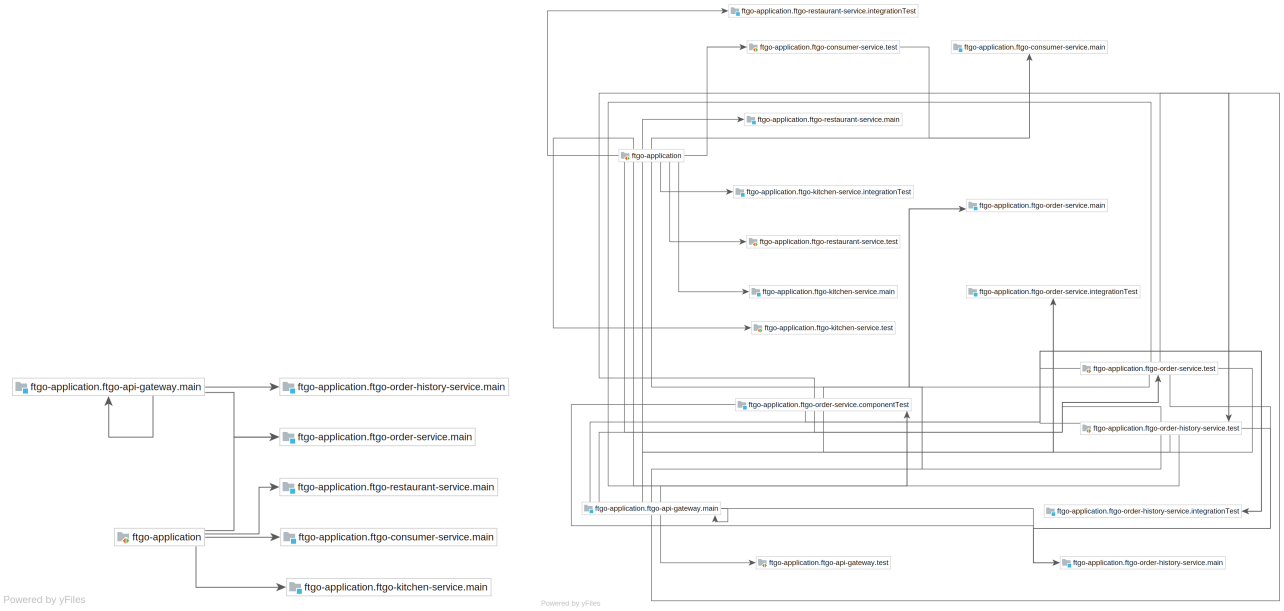


(b) Круговая раскладка узлов

Рис. 15: Визуализация проекта Train Ticket

Апробирование на проекте FTGO (результаты представлены на рис. 16) проявило следующие важные наблюдения.

- Возможности визулизации данного проекта во много лишь удачное стечение обстоятельств (а именно стиля кодирования) — большинство представленных в исходном коде запросов написаны на языке JavaScript, причём окончания запросов хранятся в переменных, что нарушает указанное ограничение 5.1, однако наименования переменных литерально совпадают с самими окончаниями, что нивелирует нарушение. Этот проект демонстрирует острую **необходимость устранения соответствующего ограничения**.



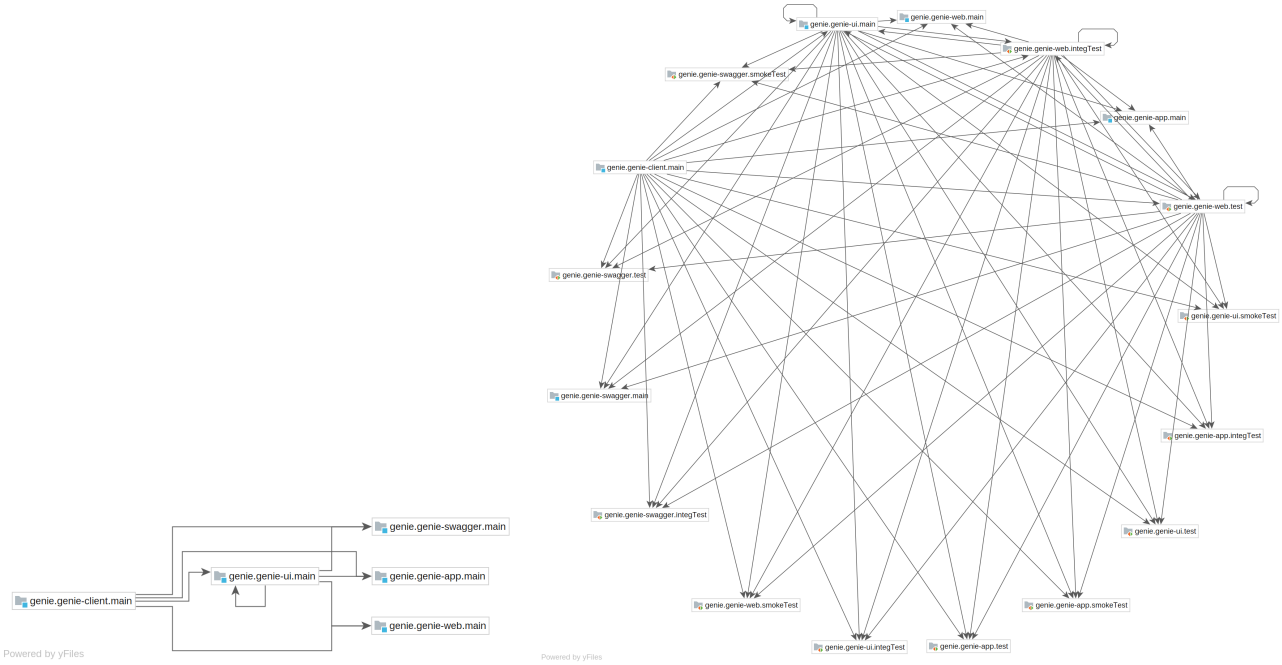
(a) Только продуктовые модули (b) С учетом модулей с тестами

Рис. 16: Визуализация проекта FGTO Application

- Структура продуктовых (production) модулей проекта довольно тривиальна, однако при включении в визуализацию модулей, содержащих тесты, число ребер и узлов кратно возрастает. В целом такую диаграмму можно применять для контроля, что на каждое возможное ребро продуктового запроса есть соответствующее ребро запроса в тестовой инфраструктуре.

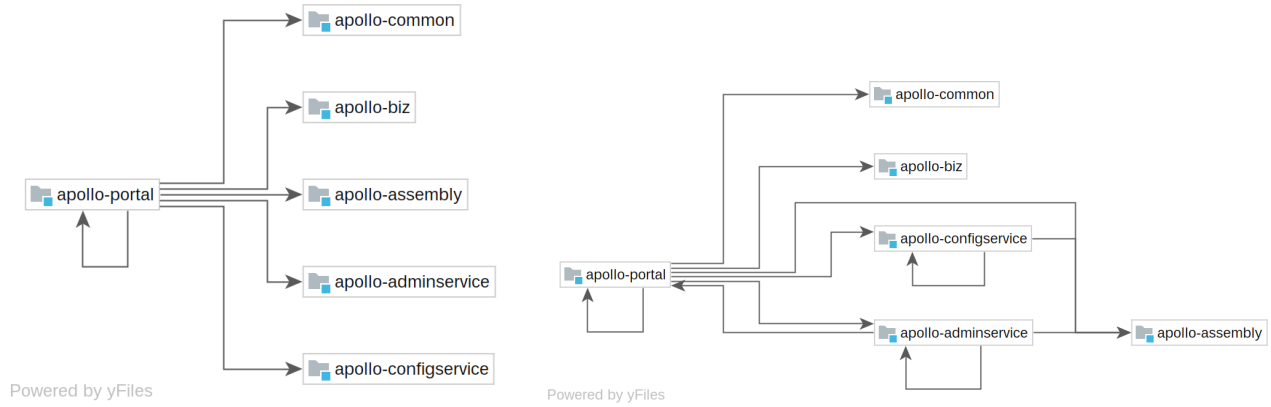
Визуализация проектов Genie и Apollo (рис. 17 и 18 соответственно) подтверждают последнее наблюдение, что включение модульных и, в особенности, интеграционных тестов кратно увеличивает отображаемый граф. При этом необходимо отметить, что продуктовая структура весьма тривиальна — суть пропуск от сервиса шлюза (gateway) к необходимому сервису в один шаг. Это вновь указывает на печально известный факт как такового отсутствия больших проектов с открытым исходным кодом и нетривиальными топологиями сервисов, что серьёзно осложняет тестирование и апробирование инструментов для микросервисной разработки.

Крайне важным проектом для апробации является Asc-Lab POC. Он представляет собой идиоматичную реализацию промышленного приложения с применением новейших версий фреймворка Micronaut, т.е. является хорошим ориентиром относительно настоящих индустриальных проектов. Репозиторий содержит собственную диаграмму взаимодействия составляющих микросервисов, с которой можно поэлементно сравнить визуализацию, сгенерированную нашим плагином и представленную на рис. 19. Оказывается, что были найдены все связи, кроме одной (от `agent-portal-gateway` до `payment-service`), ввиду нетривиальной ошибки внутри анализатора. Это также показывает, что механизм диаграмм может служить **полезным инструментом для**



(a) Только продуктовые модули (b) С учетом модулей с тестами

Рис. 17: Визуализация проекта Genie

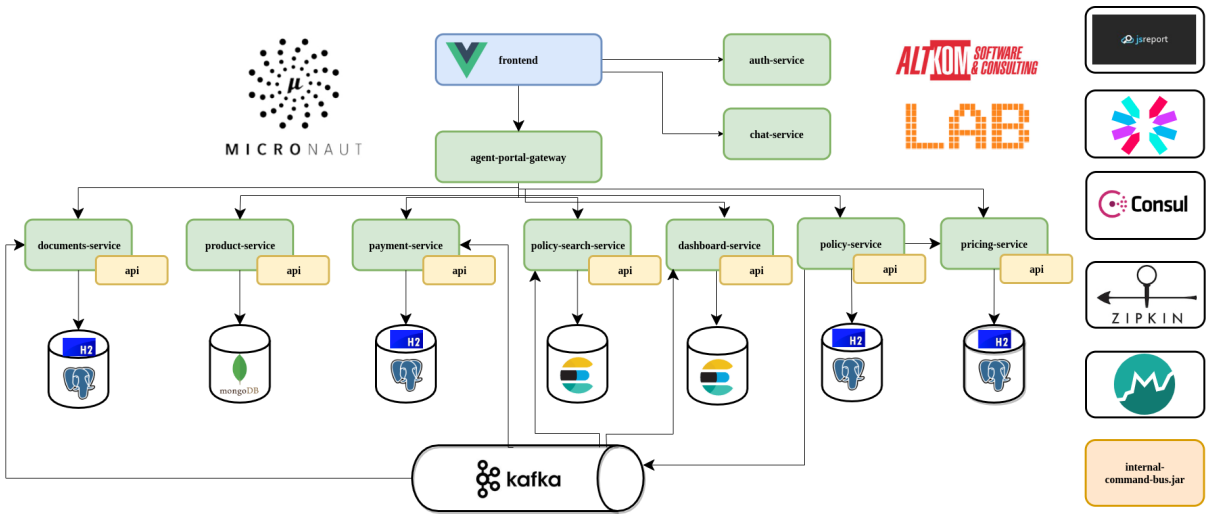


(a) Только продуктовые модули (b) С учетом модулей с тестами

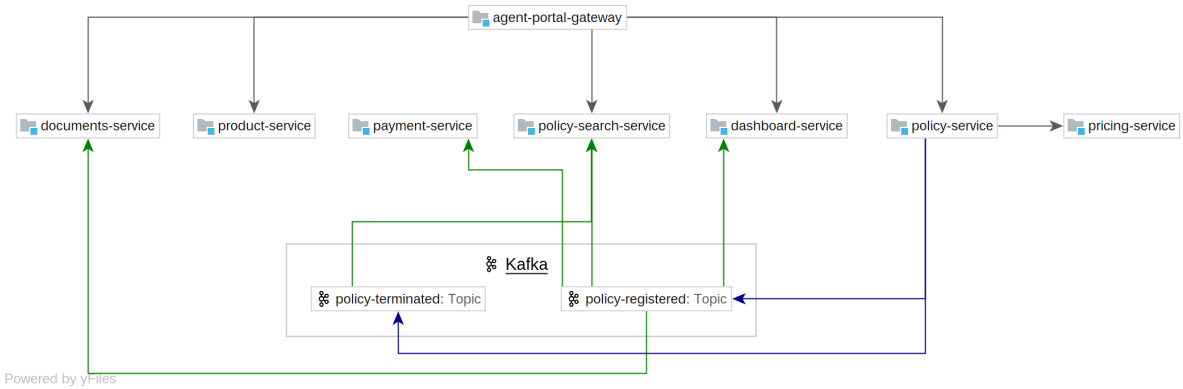
Рис. 18: Визуализация проекта Apollo

тестирования собственных разработок для микросервисных фреймворков. Также важно отметить, что наш инструмент визуализации **не отображает взаимодействий с базами данных** (ввиду текущего отсутствия анализатора конфигураций баз данных фреймворков), что является ощутимым упущением и достойно в качестве отдельной задачи.

Проекты Piomin Kafka и Jexp Meetup отображают взаимодействие микросервисов посредством очереди сообщений — соответствующие визуализации представлены на рис.20. Как видно, даже для таких простейших проектов как Piomin Kafka топология может быть весьма нетривиальной.

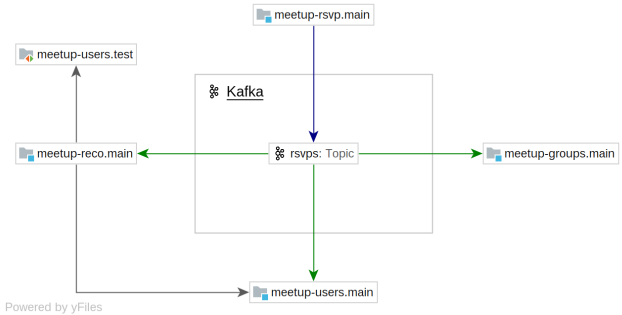


(a) Авторская диаграмма архитектуры

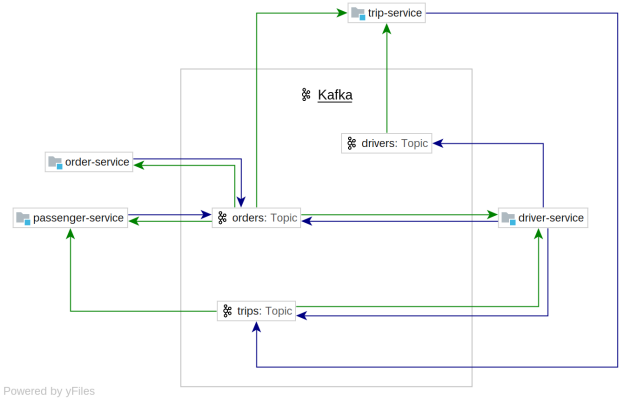


(b) Сгенерированная диаграмма взаимодействия

Рис. 19: Визуализация проекта Asc-Lab POC



(a) Jexp Meetups



(b) Piomin Kafka

Рис. 20: Визуализация проектов с очередями сообщений

В заключение, необходимо отметить, что существующая подсистема анализа микросервисных фреймворков показала себя блестяще: визуализации в значительной степени соответствуют актуальным структурам проектов, а время анализа, без каких бы

то ни было дополнительно произведённых оптимизаций, не превосходило 30 секунд[1] даже для проекта из 50 сервисов (при этом можно было продолжать свободно пользоваться всей функциональностью IDE).

В то же время, подсистема отображения диаграмм показала себя несостоятельной.

- Ввиду многочисленных технических ошибок и ограничений вручную эффективно расположить узлы было невозможно, а все поддерживаемые варианты автоматической раскладки (коих было всего 6) подчас оказывались неудачными (или вовсе нечитаемыми) для получаемых графов.
- Артефакты отрисовки затрудняли работу с диаграммой и влияли на пользовательское впечатление об инструменте в целом.
- Отсутствие склейки рёбер кратно увеличивало занимаемое место, что серьёзно осложняло ориентирование по диаграмме.

В ходе работы было исправлено значительное количество ошибок и недоработок, перечисленных в заключении.

6.2. Возможные пути развития

Устранение указанных в ходе работы не фундаментальных, технических ограничений естественным образом представляется соответствующими способами развития проекта, а именно следующими.

- Устранение ограничения 5.1, т.е. интегрирование механизма вставки ссылок в места использования константных переменных, содержащих якорь ссылки.
- Поддержка визуализации RPC-фреймворков, в частности gRPC. Это весьма широко используемый метод общения микросервисов, для которого в IntelliJ IDEA на данный момент нет соответствующих модулей анализа. Добавление таких взаимодействий на диаграмму потребует не более чем реализации одного метода интерфейса `RequestsSearcher` из 4.
- Поддержка поиска конфигураций баз данных на основании сведений фреймворка.
- Добавление к визуализации данных времени развёртывания, к примеру, из файлов Docker Compose, Kubernetes, OpenAPI и др.

[1] На машине с Intel Core I9-9980HK, 32 Гб ОЗУ

- Реализация плагинов поддержки микросервисных фреймворков для иных языков программирования. Важным преимуществом микросервисной архитектуры является почти полная независимость технологий, используемых для реализации каждого сервиса, в частности, языка программирования. На данный момент поддерживаются только Java, Kotlin и в незначительной степени JavaScript, однако не менее популярными для реализации сервисов являются также языки C# и Go (8 и 6 проектов в списке [12] соответственно). Отметим, что при реализации для данных языков модулей анализа на базе Microservices API IntelliJ Platform, генерация диаграмм начнет работать автоматически, не требуя дополнительных вмешательств.

- Интеграция с UML-диаграммами классов. Диаграмма взаимодействия микросервисов представляет собой высокоуровневую структуру проекта, UML-диаграмма классов — низкоуровневую. Уже несколько лет поднимается вопрос об объединении отображения подобных структур посредством масштабирования для получения мощного инструмента многоуровневой визуальной навигации по проекту.

- Также перспективной идеей для внедрения реализованной функциональности является интеграция с системой непрерывной серверной аналитики исходного кода Qodana[2]. Это позволит при подключении репозитория проекта автоматически генерировать для него указанную диаграмму (и, к примеру, отображать её в README) и даже обновлять её на каждый совершённый коммит.

[2] https://www.jetbrains.com/help/qodana/getting-started.html

Заключение

В ходе данной работы были получены следующие результаты.

- Сделан обзор предметной области. Рассмотрены и классифицированы существующие инструменты визуализации микросервисов, включая системы управления производительностью, диаграммы зависимостей модулей развёртывания и технологии автоматической декомпозиции монолитных проектов.
- Сформулированы требования и ограничения к целевой функциональности. Это позволяет ориентироваться в степени поддержки IDE многообразия существующих языков, фреймворков, протоколов, технологий и др.
- Спроектирована архитектура целевой функциональности. Компонент диаграммы способен автоматически поддерживать модули анализа для любых языков и фреймворков при реализации таковыми Microservices API, при этом интеграция отображений иных видов взаимодействий (с базами данных, посредством RPC-фреймворков и т.д.) тривиальна и состоит в реализации одного интерфейса из одного метода.
- Реализована функциональность генерации диаграмм микросервисов как часть Endpoints плагина. Поддерживается отображение взаимодействий с помощью HTTP/Websocket протоколов и/или очереди сообщений, а также перечисление посредством окна поиска всех мест в исходном коде, на основании которых взаимодействие было установлено.
- Выполнена апробация на ряде крупнейших микросервисных проектов с открытым исходным кодом. Способ представления окончаний URL-адресов в исходном коде оказывает наибольшее влияние на успешность анализа. Подсистема поддержки микросервисных фреймворков справляется со всеми поставленными задачами, в то время как подсистема диаграмм оказалась существенно несостоятельной. Сформулированы возможные варианты развития представленной функциональности.

Перечисленные результаты интегрированы во внутренний репозиторий компании JetBrains, предоставлен акт о внедрении.

Благодарности

Автор выражает отдельную признательность и благодарность коллегам, помогавшим ему при выполнении данной работы:

- Артамонову Юрию Сергеевичу — за реализацию высококачественной системы поиска эндпоинтов и всестороннюю консультацию по её наиболее эффективному применению;
- Васильеву Сергею Николаевичу — за внимательное ревью внесённых мной изменений, состоящих из нескольких тысяч строк кода по всей подсистеме диаграмм, находившейся в замороженном состоянии на протяжении четырёх лет.

Также автор благодарит компанию JetBrains за предоставленное оборудование и всестороннюю поддержку в непростых условиях дистанционного режима работы, своего научного руководителя, Луцива Дмитрия Вадимовича, за помощь, неоднократно проявленное терпение и многочисленные ценные замечания по тексту работы, а также лично Митропольского Николая Николаевича за выделенное время и затраченные усилия на изучение и рецензирование данной работы.

Список литературы

[1] Application Performance Management: State of the Art and Challenges for the Future / Christoph Heger, André van Hoorn, Mario Mann, Dušan Okanović. — 2017. — 04. — С. 429–432.

[2] Clauset Aaron, Newman M, Moore Cristopher. Finding community structure in very large networks // Physical review. E, Statistical, nonlinear, and soft matter physics. — 2005. — 01. — Т. 70. — С. 066111.

[3] Fowler Martin. Monolith First. — 2015. — Режим доступа: https://martinfowler.com/bliki/MonolithFirst.html (дата обращения: 10.05.2021).

[4] Garriga Martin. Towards a Taxonomy of Microservices Architectures. — 2018. — 02. — С. 203–218. — ISBN: 978-3-319-74780-4.

[5] Goldsmith Kevin. Microservices at Spotify. — 2015. — International software development conference «GOTO Berlin». Режим доступа: https://www.kevingoldsmith.com/talks/microservices-at-spotify.html.

[6] IntelliJ Idea Micronaut Plugin. — Режим доступа: https://www.jetbrains.com/help/idea/endpoints-tool-window.html (дата обращения: 10.05.2021).

[7] IntelliJ Idea Функциональная и эргономичная IDE для JVM. — Режим доступа: https://www.jetbrains.com/ru-ru/idea/ (дата обращения: 10.05.2021).

[8] Koschke Rainer. Software Visualization in Software Maintenance, Reverse Engineering, and Reengineering: A Research Survey // Journal on Software Maintenance and Evolution. — 2003. — 03. — Т. 15. — С. 87–109.

[9] Lewis James, Fowler Martin. Microservices. — 2014. — Режим доступа: https://martinfowler.com/articles/microservices.html (дата обращения: 10.05.2021).

[10] Lohnertz Jakob. Toward automatic decomposition of monolithic software into microservices : дис. ... канд. наук / Jakob Lohnertz ; University of Amsterdam. — 2020. — Авг. — Режим доступа: https://doi.org/10.5281/zenodo.4280725.

[11] Meshenberg Ruslan. Microservices at Netflix Scale - First Principles, Tradeoffs and Lessons Learned. — 2016. — International software development conference «GOTO Amsterdam». Режим доступа: https://gotocon.com/amsterdam-2016/presentation/Microservices%20at%20Netflix%20Scale%20-%20First%20Principles,%20Tradeoffs%20&%20Lessons%20Learned.

[12] Rahman Mohammad Imranur, Panichella Sebastiano, Taibi Davide. A curated Dataset of Microservices-Based Systems // CoRR. — 2019. — Vol. abs/1909.03249. — 1909.03249.

[13] Ranney Matt. What I Wish I Had Known Before Scaling Uber to 1000 Services. — 2016. — International software development conference «GOTO Chicago». Режим доступа: https://gotocon.com/chicago-2016/presentation/What%20I%20Wish%20I%20Had%20Known%20Before%20Scaling%20Uber%20to%201000%20Services.

[14] Reflections on the REST Architectural Style and "Principled Design of the Modern Web Architecture" (Impact Paper Award) / Roy T. Fielding, Richard N. Taylor, Justin R. Erenkrantz и др. // Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. — ESEC/FSE 2017. — New York, NY, USA : Association for Computing Machinery, 2017. — С. 4–14. — Режим доступа: https://doi.org/10.1145/3106237.3121282.

[15] Richardson Chris. Microservices Architecture. — 2014. — Режим доступа: https://microservices.io/articles/whoisusingmicroservices.html (дата обращения: 10.05.2021).

[16] Tilkov Stefan. Don't start with a monolith. — 2015. — Режим доступа: https://martinfowler.com/articles/dont-start-monolith.html (дата обращения: 10.05.2021).

[17] Visualization Tool for Designing Microservices with the Monolith-First Approach / R. Nakazawa, T. Ueda, M. Enoki, H. Horii // 2018 IEEE Working Conference on Software Visualization (VISSOFT). — 2018. — С. 32–42.

[18] Visualizing a Microservices Architecture with AHAS. — Режим доступа: https://www.alibabacloud.com/blog/visualizing-a-microservices-architecture-with-ahas_594621 (дата обращения: 10.05.2021).

[19] Кознов Д.В. Основы визуального моделирования. Основы информационных технологий. — Интернет-университет информационных технологий, 2008. — ISBN: 9785947748239. — Режим доступа: https://books.google.ru/books?id=vOheNwAACAAJ.