DOI 10.37882/2223-2966.2020.10.23

РЕАЛИЗАЦИЯ СЕРВИСНО ОРИЕНТИРОВАННОЙ АРХИТЕКТУРЫ С ПРИМЕНЕНИЕМ ПРИЛОЖЕНИЯ АГРЕГАТОРА

IMPLEMENTATION OF A SERVICE-ORIENTED ARCHITECTURE USING THE AGGREGATOR APPLICATION

V. Putnin

Summary: The article created the implementation of a service-oriented architecture without taking into account the requirements for the subject area, using the design principle «Inversion of control», the pattern «Aggregator», the Java programming language and the Spring framework. The system description includes: 3 platform independent applications, 3 REST program interfaces, internal services for communicating program interfaces, utility classes for creating http connection objects. To build applications and connect dependent components, Maven was installed, which was originally designed as a tool for managing the vital cycle of applications and dependent libraries and frameworks at Apache, which allows you to make changes at every stage of application operation. The description includes a set of operations that allow the user to use the interface. URLs display requests. The implemented service system was a collection in a single organism for ease of management, call functions and further expansion of the system. This implementation performs 4 CRUD operations: creating records, updating records, deleting records, reading records.

Keywords: rest, service-oriented architecture, cross-platform, inversion of control, Java, spring, mining.

Путнин Валерий Игоревич
Рязанский государственный радиотехнический университет
putnin.v@gmail.com

Аннотация: В статье создана реализация сервисно ориентированной архитектуры без привязки к предметной области, с использованием принципа проектирования «Инверсия контроля», паттерна «Агрегатор», языка программирования Java и фреймворка Spring. В описании системы создано: 3 платформенно независимых приложения, 3 REST программных интерфейса, внутренние сервисы для общения программных интерфейсов, утилитарные классы для создания объектов http соединений. Для сборки приложений и подключения зависимостей установлен Maven, спроектированный изначально, как инструмент для управления жизненным циклом приложения и зависимыми библиотеками и фреймворками в компании Apache, который позволяет внедрять изменения на каждый фазе работы приложения. В описание включены наборы операций, позволяющих пользователю интерфейса осуществлять конкретные действия в программе и осуществлять изменения. Отображены url запросы с параметрами объектов для взаимодействия между различными состояниями, которые могут изменяться при выполнении запросов. Реализованная система сервисов была собрана в единый организм для удобства управления, вызова функций и дальнейшего расширения системы. Данная реализация выполняет 4 CRUD операции: создание записи, обновление записи, удаление записи, чтение записи.

Ключевые слова: сервисно ориентированная архитектура, кроссплатформенность, инверсия контроля, горная промышленность.

Большинство высоконагруженных приложений, реализованных с применением любых языков программирования, в основном в качестве архитектурного решения используют «монолит», в таких приложениях вся логика, нагрузка, бизнес требования, взаимодействие с базами данных расположено в одном приложении развернутое на одном сервере. Так при появлении новых разработчиков на проекте, им приходиться вникать в детали всего большого приложения, а при внесении изменений в исходный код требуется перезагрузка всего приложения, иногда это приводит к потери связи между пользователем и приложением. Так же обновление приложения становиться долгосрочной и дорогостоящей проблемой, ориентированной на экспертных программистов узкой области и почти не затрагивают программистов со знаниями в различных языках программирования и фреймворках. Разработка и реализация универсальной сервисно ориентированной архитектуры, пригодной для всех программистов различных уровней и специализаций, без привязки к языку программирования, позволяющего внедрять, расширять и обновлять существующее приложение новым функционалом, позволит решить несколько проблем одним решением. [1].

Для реализации прототипа приложения с применением сервисно ориентированной архитектуры был выбран фреймворк, реализующий принцип инверсии контроля Spring, ввиду простоты его использования, эффективности управления, возможности написания кода на языке Java и большого объема документации по фреймворку и языку программирования, доступной на многочисленных форумах и книгах [2].

Сначала определимся с количеством сервисов – их будет 3. Создадим агрегирующее приложение, которое будет иметь доступ к остальным приложениям. Агрегирующее приложение будет инициализирована с помощью модуля SpringBoot. Подобным образом мы создадим и другие приложения системы.

Далее следует создать класс, который будет за-

пускать. Для этого создается класс с аннотацией @ SpringBootApplication, у данного класса переопределяется метод main, где прописывается команда для запуска приложения. Сами приложения будут взаимодействовать между собой по http. Таким образом создаются сервисы (рис. 1) [3].

В качестве сборщика приложения будет применяться Maven – технология от Apache, позволяющая управлять жизненным циклом приложения, а также зависимостями через центральный публичный репозиторий. В одной из зависимостей будет определен spring-boot- web для работы с http запросами. Управление зависимостями таким способом происходит без внесения изменений в программный код, а с помощью редактирования xml документа, содержащего инструкции для управления жизненным циклом приложения.

Теперь необходимо реализовать сервис, доступ к которому будет осуществлен через сервис. Для этого определим предметную область сервиса, например «горнодобывающая промышленность». Создадим интерфейс с помощью которого сможем выполнять операции с информацией о количестве добытой руды.

Среда разработки приложения будет IntellijIDEA, т.к. это самый мощный инструмент из существующих для языка java, имеет бесплатные лицензии для студентов и научных сотрудников, и пробную лицензию для остальных программистов.

RockController.class – класс написанный на языке Java, описывающий методы программного интерфейса, связанные с получением информацией о количестве руды. В приложении может быть бесконечное количество контроллеров, но мы ограничимся только одним для демонстрации реализации архитектуры. В приложении не должно быть переменных, меняющих свое значение, т.к. все сервисы будут многопоточными и не должны хранить промежуточные состояния.

Опишем методы контроллера, которые будут делегировать логику выполнения в сторонние классы приложения:

public void getRockInfo(long rockId);
public long createRockInfo();
public void updateRockInfo(long rockId);
public void deleteRockInfo(long rockId);

Модификаторы доступа к переменным обязательно должны быть public для того, что бы можно было получить к ним доступ через http запрос. Так же для того, что бы к этим методом можно было обратиться по url, нужно пометить их аннотациями из фреймворка Spring, такой, как @RequestMapping(). [4].

Для выполнения логики будет создан отдельный класс, в котором будут реализованы все необходимые скрипты по получению данных о количестве руды с независимых источников (база данных, бурильная машина, CRM система) любая такая система, может быть написана под любую операционную систему.

Следующим шагом является описание точек доступа методов. К каждой точке доступа можно будет получить доступ через url запрос следующего вида:

https://host:port/rock/info

Выбор метода будет определяться адресом контроллера «rock», адресом метода «info», и http

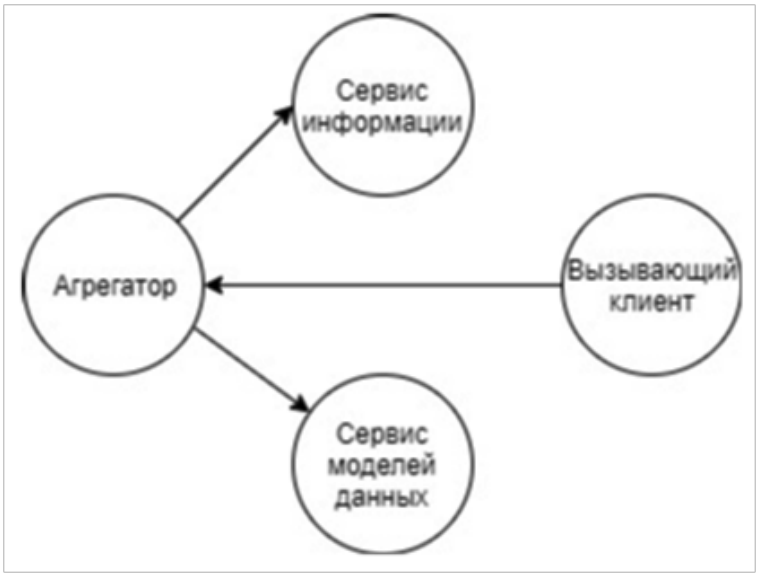

Рис. 1. Схема сервисно ориентированной архитектуры

методом(GET,POST,PUT,DELETE). В случае успешного выполнения запроса, будет возвращаться ответ в кодом 200.

Далее создадим приложение, которое будет агрегировать другие приложения, так же, как и rockApp. Принцип создания приложения такой же, как и для rockApp, но в нем будут другие контроллеры и точки доступа. Опишем методы контроллера, которые будут делегировать логику выполнения в сторонние классы приложения:

public void getCommonRockInfo(long rockId);
public long createCommonRockInfo();
public void updateCommonRockInfo(long rockId);
public void deleteCommonRockInfo(long rockId);

Данные методы будут описаны в классе контроллере CommonRockController.class. Далее через эти методы будет взаимодействие с объектом класса RockService.class в методах, которого будет реализована логика по созданию http запросов [5].

Внедрение объекта типа RockService в контроллер будет по принципу инверсии контроля, этот механизм реализованный во фреймворке Spring, внедрение происходит через аннотацию @Autowired.

Для запуска каждого приложения на одном компьютере, требуется переопределить порты на внутренних серверах поставляемых вместе с фреймворком SpringBoot. Для определения портов необходимо добавить файл *application.properties* и добавить в него переменную *server.port* и присвоить ей значение уникального порта. При запуске приложения порт отобразиться в консоли.

Далее реализуем утилитарный класс, в котором реализуем механизм создания соединения и http запросов. Класс назовем как *RequestBuilderUtil,* методы в этом классе будут статическими:

public static void sendGET(String url, String params);
public static void sendPOST(String url, String params);
public static void sendPUT(String url, String params);
public static void sendDELETE(String url, String params);

В данном фрагменте кода *public* является определителем режима доступа (public – глобальная переменная, private – локальная переменная). Каждый метод в этом классе генерирует запрос запрос с *http* методами *(get,post,put,delete)*. Скрипты, реализованные в теле методов, зависят от выбора разработчика, может быть любой и не влияет на построение архитектуры. В качестве параметров передается url-адрес сервиса, к которому нужно обратиться и параметры, которые нужно передать [6].

На детальной схеме взаимодействия между агрегатором и зависимым сервисом предст авлена зависимость с одним сервисом по средством rest контроллеров и http запросов. Не требуется инициализация объектов т.к. фреймворк Spring позволяет создавать динамическое связывание, данный пример кода взят из приложения агрегатор, который подключатся к приложению rockApp:

@Service
public class RockService {}
@Autowired
private RockService rockService;

В данных строках кода представлен пример, как пометить класс готового для внедрения.

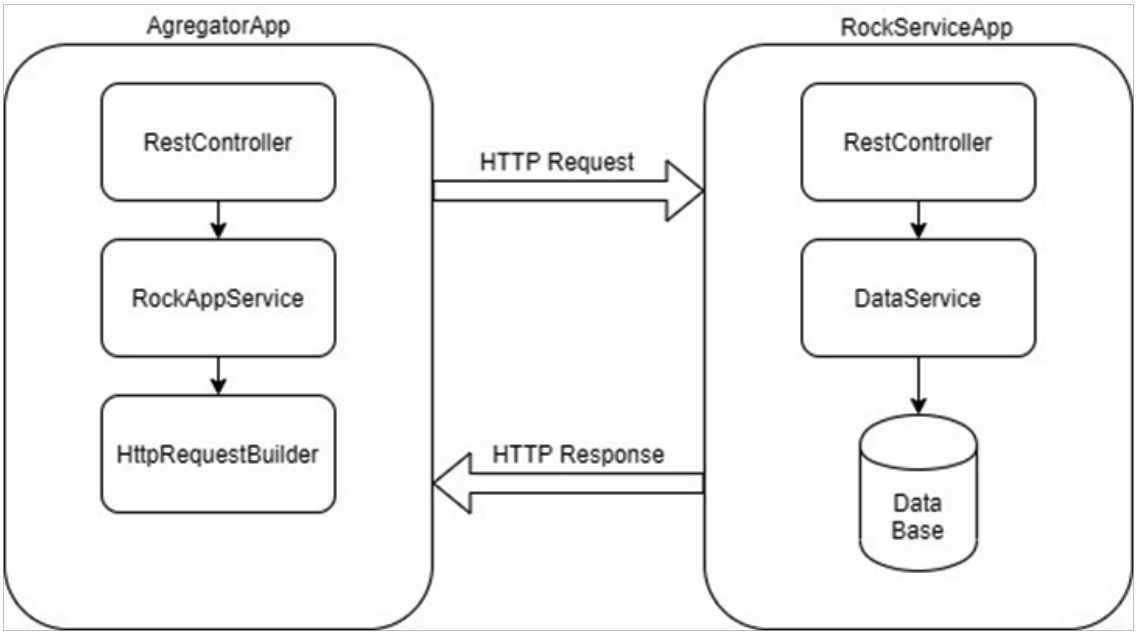

Рис. 2. Схема взаимодействия между приложениями

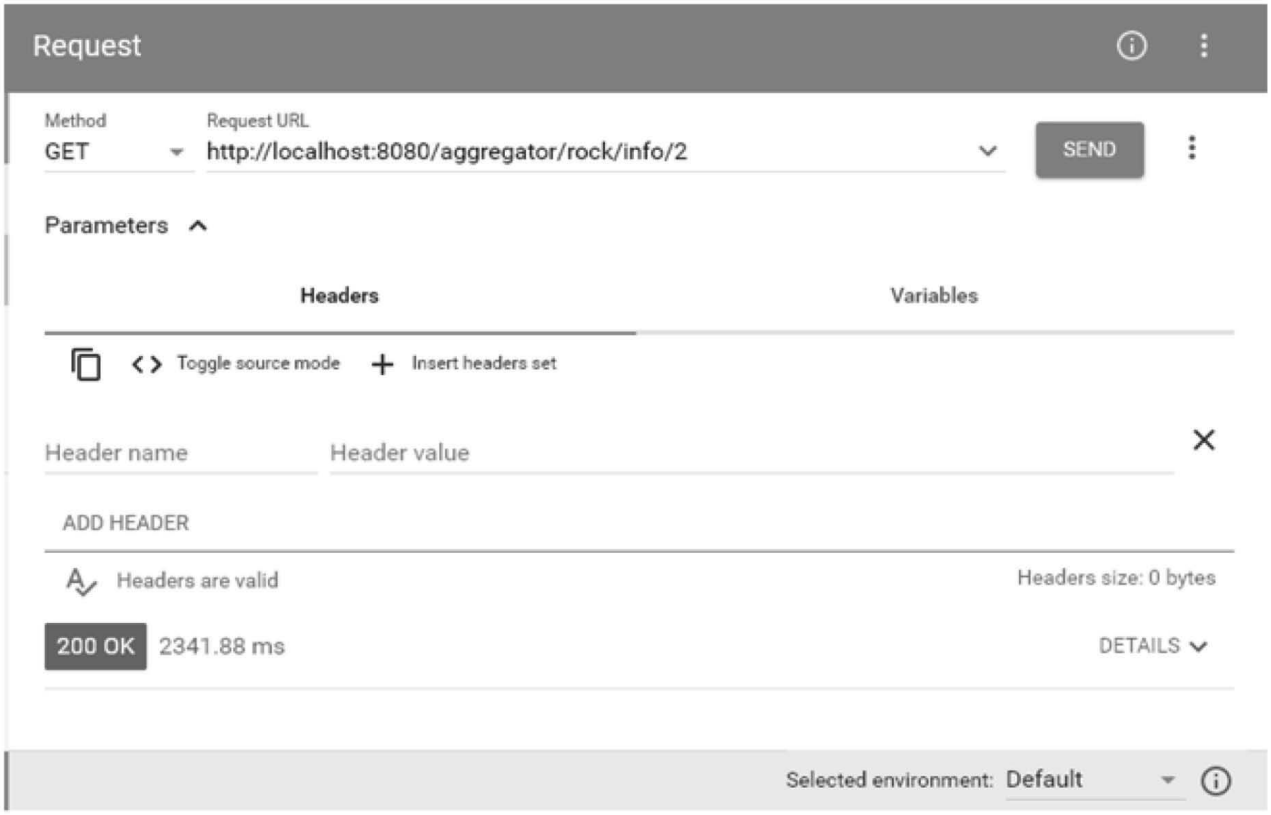

Рис. 3. Клиент приложения, отправляющий запрос на приложение

Так же необходимо создать точки доступа в агрегаторе:

```
@RestController
@RequestMapping("/home/rock")
public class AgregatorRockController{}
@RequestMapping(value="info", method="RequestMethod.GET")
public void getCommonRockInf(long rockId)
```

С помощью аннотаций создан маппинг на контроллер и методы. В качестве параметра method ставиться значение http метода, а в value ставиться значение маппинга. Далее необходимо протестировать взаимодействие между приложениями. При тестировании приложения будем использовать клиент RESTAdvancedClient который позволяет выполнять http запросы [7]. Запрос к приложению агрегатору будет следующего вида:

https://host:port/aggregator/rock/info

Для успешного кейса тестирования взаимодействия приложений, необходимо отобразить событие обращения к методам в консоль приложения, а ожидаемый код ответа должен быть 200.

По завершению реализации, следуют отметить преимущества данного архитектурного подхода по сравнению с монолитным приложением. Она позволяет создавать независимые приложения, объединять их в едином месте и взаимодействовать между этими приложениями. Данный подход подойдет для небольших компаний с количеством пользователей от 10 человек, так же для крупных корпораций с миллионами пользователей и со сложной инфраструктурой. Где каждое приложение создается под конкретную предметную область, количество пользователей может достигать миллионов пользователей. Это позволит быстро создавать новые информационные продукты например для банков, страховых компаний, транспортных и других, и встраивать новые продукты в уже существующую инфраструктуру[8].

Таким образом решаются следующие проблемы:

Поиск программиста со специфическим набором технологий.

Адаптация программиста на давно существующем проекте

Ознакомление с предметной областью не влияющей на новую разработку

Нагрузка на проект

Обновление независимых частей приложения

Данная реализация подходит для внедрения в любую информационную структуру, на любом языке программирования и на любой операционной системе.

ЛИТЕРАТУРА

1. Абельсон Х., Сассман Д.Д. Структура и интерпретация компьютерных программ: КДУ, Добросвет, 2018.
2. Козмина Ю., Харроп Р., Шефер К., Хо К. Spring 5 для профессионалов: Диалектика-Вильямс, 2019.
3. Бастани К., Лонг Д. Java в облаке. Spring Boot, Spring Cloud: Питер, 2017.
4. Мехди М., Мехди М. Непрерывное развитие API. Правильные решения в изменчивом технологическом ландшафте: Прогресс книга, 2020.
5. Smisek J., Jancosek M., Pajdla T. 3D with Kinect // Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on, 2011. P. 1154–1160.
6. Flinn K. Leadership Development: A Complexity Approach. London : Routledge, 2018. 40 p..
7. Abadi M., Barham P., Chen J., Chen Z., Davis A., et al. TensorFlow: a System for Large-Scale Machine Learning // OSDI. 2016. Vol. 16. P. 265-283.
8. Xia L., Chen C., Agarwal J. Human Detection Using Depth Information by Kinect // Computer Vision and Pattern Recognition Workshops (CVPRW), 2015 IEEE Computer Society Conference on, 2015. P. 15–22.
9. Майерс Г., Баджетт Т. Искусство тестирования программ: Диалектика, 2019.

Журнал «Современная наука: актуальные проблемы теории и практики»

Рязанский государственный радиотехнический университет