

Способы версионирования и одновременного запуска несколько различных версий сервисов

Вихляев Дмитрий Романович

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В данной работе описываются способы версионирования контрактов и программных компонентов сервисов. Освещаются процессы управления изменениями в интерфейсах взаимодействия, набор формальных правил, описывающих API, структуру данных и логику взаимодействия между микросервисами. В качестве результата приводятся решения одновременного запуска нескольких сервисов различных версий и способы скрытия внутренней сложности системы от внешних потребителей.

Ключевые слова: микросервисная архитектура, версионирование, сервисный контракт.

Methods for versioning and running multiple different versions of services simultaneously

Vikhlyayev Dmitry Romanovich

Sholom-Aleichem Priamursky State University

Student

Abstract

This paper describes the methods of versioning contracts and software components of services. It highlights the processes of managing changes in interaction interfaces, a set of formal rules describing the API, data structure, and interaction logic between microservices. As a result, it provides solutions for simultaneously running multiple services of different versions and ways to hide the internal complexity of the system from external consumers.

Keywords: microservice architecture, versioning, and service contracts.

1 Введение

1.1 Актуальность

Важнейшими аспектами версионирования сервисов и их контрактов является поддержка обратной совместимости контрактов таким образом, чтобы потребители не изменялись при изменении контракта сервиса. Необходимо поддерживать актуальную документацию контрактов, таким образом, чтобы разработчики при использовании могли легко и просто понять, каким образом взаимодействовать с сервисом. Помимо этого необходимо формулировать стратегии версионирования сервисов и контрактов, чтобы

специалисты могли понимать, какие правила установлены в системе по версионированию и каким образом их стоит придерживаться. При этом неважно, какой протокол выбран для взаимодействия, всегда стоит придерживаться одних и тех же принципов версионирования.

1.2 Обзор исследований

Статья А.М.Шитько посвящена изучению технологии построения микросервисной архитектуры, которая позволяет сохранять модульность и независимость развертывания компонент Enterprise приложения [1]. В работе Д.С.Кугушева дано описание принципа работы микросервисной архитектуры, приведены ее отличительные особенности, позволяющие существенно упростить процесс разработки больших продуктов, а также упомянуты основные технологические особенности микросервисного подхода, которые необходимо учитывать при выборе этой технологии разработки. [2]. В исследовании М.Э.Джалалова особое внимание уделяется шаблонам API Gateway, Backend For Frontend и Circuit Breaker, их реализации и оптимизации в контексте микросервисных архитектур. [3]. В качестве результата исследования А.Абдирашитулы, стала предложенная концепция LLM-ориентированной микросервисной архитектуры, обеспечивающая изоляцию ресурсов и независимое масштабирование компонентов. [4]. В статье М.Г.Городничев и Р.В.Полонский сделан обзор типов современных веб-приложений и архитектуры, используемой для их разработки, дано сравнение микросервисной и монолитной архитектуры [5]. М.Н.Карпович исследовал особенности проектирования микросервисно-событийных архитектур для высоконагруженных распределенных систем обработки информации [6]. С.В.Маличенко рассмотрел проблемы перехода от монолитной к микросервисной архитектуре [7]. Г.Р.Берайлик описал способы спроектирования информационных систем с помощью микросервисной архитектуры [8].

1.3 Цель исследования

Цель исследования – описать способы версионирования и одновременного запуска несколько различных версий сервисов и их контрактов.

2 Материалы и методы

В работе приводятся методы, включающие временную поддержку предыдущих и параллельную поддержку версий. Представлены архитектурные шаблоны управления контрактами сервисов через API Gateway и Backend for Frontend.

3 Результаты и обсуждения

Так как сервисы разрабатываются с разной скоростью, разными командами в разное время. И командам, чьи сервисы взаимодействуют между собой, важно понимать, что изменилось. Для обеспечения коммуникации об

изменениях в сервисах и библиотеках применяют подход SemVer, который позволяет сообщить потребителю, каким образом и насколько изменился контракт или сервисы или библиотеки. SemVer состоит из трёх фасетов: patch, minor и major.

Изменение Фасета patch сообщают о том, что были внесены небольшие изменения. Это может быть изменение внутренней реализации или исправление какого-то бага, не влияющего на внешний контракт сервиса.

Изменение фасета minor говорит о том, что внешний контракт сервиса поменялся так, что не сломана обратная совместимость. То есть существующие клиенты обновлять не нужно, их можно использовать даже с новой версией сервиса.

Изменение фасета major говорит о том, что произошло обратное несовместимое изменение внешнего контракта, и потребители обязаны обновить своё поведение.

Таким образом, всего в трёх цифрах. Можно сообщить о наличии или отсутствии обратно несовместимых изменений для потребителей, которые уже могут, ориентируясь на эти цифры, решать, нужно ли им обновлять свои сервисы или нет.

При поддержке различных версий конечных точек (Endpoint) у одного и того же сервиса, важно понимать, что усложняется кодовая база сервиса. То есть одновременно нужно иметь код, который обслуживает как первую версию, так и вторую. Это усложняет сервис и делает работу программиста более сложной. Изначально клиенты работают с сервисом по версии 1, после этого публикуется версия сервиса, которая поддерживает обе версии Endpoint: V1 и V2. Постепенно клиенты этого сервиса переключаются на вторую версию Endpoint. После этого релизуется новая версия сервиса, для которой отключается Endpoint версии 1, остаётся только Endpoint версии 2. Таким образом, происходит постепенный переход от версии 1 к версии 2 с сохранением функциональности всей системы (рис.1).

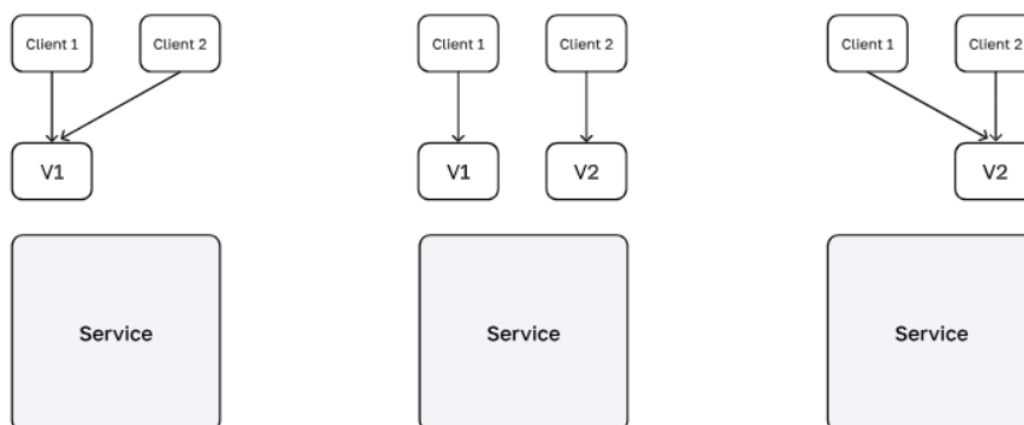


Рис. 1. Постепенный переход к новым версиям с временной поддержкой предыдущей версии

Другой вариант предусматривает одновременный запуск сервисов различных версий. Например, версия 1 сервиса реализует контракт с Endpoint версии 1, а версия 2 сервиса контракт с Endpoint версии 2. При этом обе эти версии работают с одной и той же базой данных. Таким образом, получается, что на разных хостах живут. Различные версии сервисов, обеспечивающие реализацию различных версий контракта (рис.2).

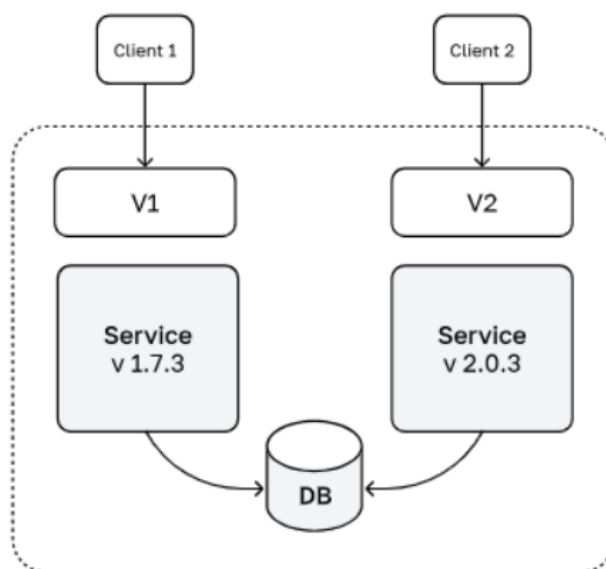


Рис. 2. Параллельная поддержка версий

Клиенты точно так же постепенно могут переключиться с одного хоста на другой, и версия один будет отключена. Особенностью данной схемы является то, что она работает с одной базой данных. И необходимо обращать особое внимание на то, каким образом как версия 1, так и версия 2 работает с базой данных. Потому что могут случиться ситуации, когда изменения в базе данных для сервиса версии 2 несовместимы с реализацией в сервисе 1. Таким образом, изменения в базе могут сломать работу первой версии. Это нужно учитывать и разрабатывать сервис соответственно.

В микросервисной архитектуре это влечёт за собой ряд сложностей. Контракты сервисов очень гранулярны, то есть каждый сервис реализует какой-то небольшой кусочек каждого конкретного API. Это сложно для потребителей, потому что им приходится знать, какому конкретному сервису какой запрос послать. Также у разных потребителей могут быть разные требования к контрактам. Например, веб-приложение может работать только по HTTP, а мобильное приложение удобнее работать по GRPC или любому другому бинарному протоколу. Кроме того, Количество и адреса сервисов могут динамически изменяться в зависимости от конфигурации системы. Разделение системы на сервисы может со временем меняться. Что усложнит использование данной системы потребителями, потому что при изменении или перемещении функционала из одного сервиса в другой, или выделении функционала в отдельный сервис придётся изменять потребителей. Кроме

того, разные сервисы могут использовать разные протоколы взаимодействия из-за особенностей своей реализации.

Для решения этих проблем большинство систем, построенных на микросервисной архитектуре, применяют подход сдельной системой, которая предоставляет единую точку доступа к внешнему контракту всей системы. API шлюз (Gateway) собирает контракт всех сервисов и позволяет внешним потребителям обращаться в единую точку для получения функциональности системы (рис.3).

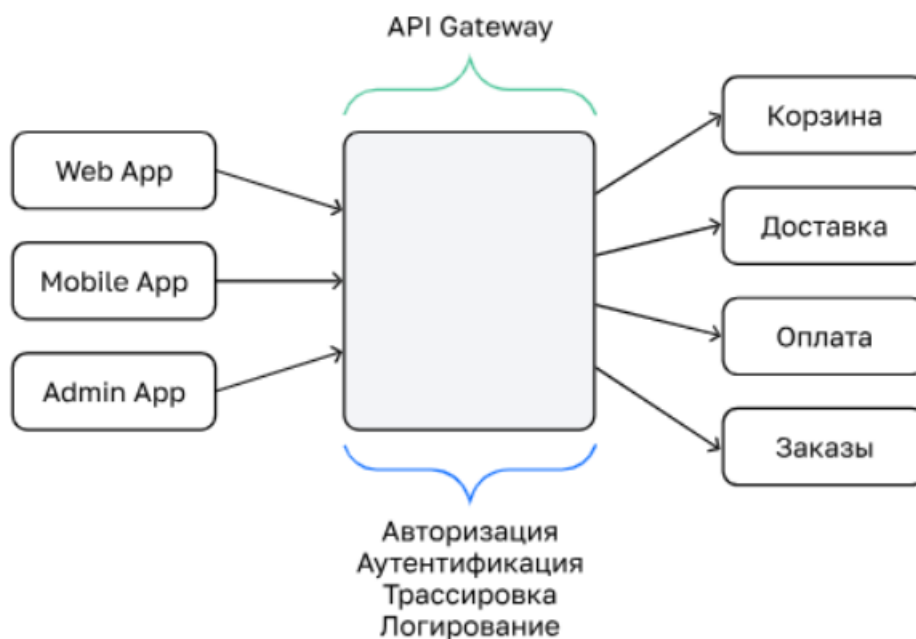


Рис. 3. Организация сервисных контрактов через API Gateway

Такой подход позволяет скрыть от потребителей разделение на большое количество небольших сервисов с разным API. Обеспечить балансировку и маршрутизацию запросов, в том числе с учётом динамического изменения адресов и количества сервисов, которые реализуют функциональность системы. Скрыть от потребителей изменения в составе и структуре сервисов, которые обеспечивают реализацию функциональности системы.

У такого подхода есть и минусы. Во-первых, это внесение дополнительного компонента в систему, который добавляет сложности. А также увеличение времени сетевого запроса, так как появляется ещё один запрос от API Gateway к конкретным сервисам.

Иногда ограничения системы накладывают дополнительные требования к внешним API для различных потребителей. Например, для обеспечения безопасности в административной панели может понадобиться дополнительная система проверки прав доступа. А для обеспечения работы мобильного приложения в сетях с надёжным интернет соединением. Дополнительные подходы в организации такого взаимодействия. В этом случае применяют подход Backend for Frontend. Когда для каждого типа

фронта, например, для мобильного приложения, для веб-приложения, для административной панели, для систем, а для устройств в мире интернет-вещей создаётся отдельный внешний API, который реализует протокол, удобный именно для этого типа потребителя (рис.4).

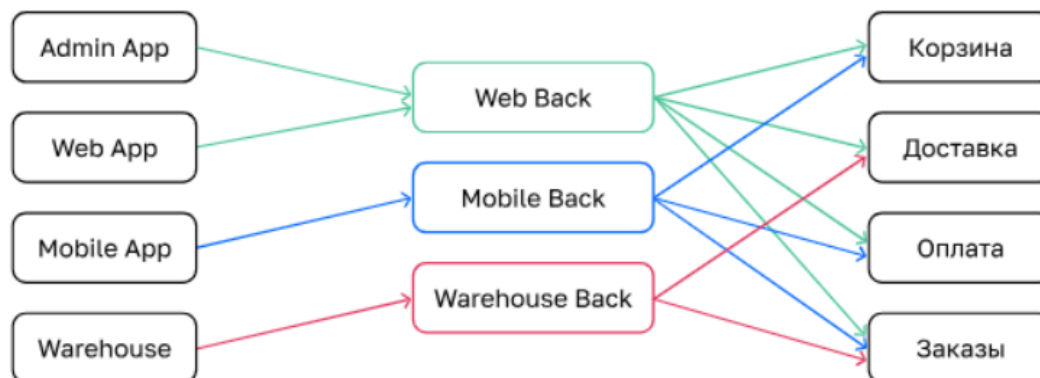


Рис. 4. Подход Backend for Frontend

А при их наличии правильно коммуницировать их с помощью протокола SMV. Также стоит применять подходы API Gateway и Backend for Frontend для того, чтобы скрыть внутреннюю сложность системы от внешних потребителей.

4 Выводы

Таким образом, важно избегать обратно несовместимых изменений во внешних контрактах, как отдельных сервисов, так и всей системы. Несмотря на имеющиеся минусы, большинство систем, реализующиеся по принципам микросервисной архитектуры, используют подход API Gateway, предоставляя внешний API для своих потребителей. Поскольку поддержка и управление единой системой предотвращает появление многих потенциальных ошибок, критичных для больших проектов.

Библиографический список

1. Шитько А.М. Проектирование микросервисной архитектуры программного обеспечения //Труды БГТУ. Серия 3: Физико-математические науки и информатика. 2017. №. 9 (200). С. 122-125.
2. Кугушева Д.С. Проектирование сложного программного обеспечения с использованием микросервисной архитектуры //Инновации и инвестиции. 2020. №. 5. С. 188-190.
3. Джалалов М.Э. Применение шаблонов проектирования для управления API в микросервисной архитектуре //Экономика и качество систем связи. 2024. №. 1 (31). С. 128-136.
4. Абдирашитұлы А. Микросервисная архитектура в NLP платформах:

- проектирование и масштабируемость //Вестник науки. 2026. Т. 1. №. 5 (98). С. 609-619.
5. Городничев М.Г., Полонский Р.В. Оценка возможности использования микросервисной архитектуры при разработке пользовательских интерфейсов клиент-серверного программного обеспечения //Экономика и качество систем связи. 2020. №. 3 (17). С. 33-43.
 6. Карпович М.Н. Особенности проектирования микросервисно-событийных архитектур для высоконагруженных распределенных систем обработки информации //Труды БГТУ. Серия 3: Физико-математические науки и информатика. 2023. №. 1 (266). С. 89-95.
 7. Маличенко С.В. Проблемы перехода от монолитной к микросервисной архитектуре //Евразийский научный журнал. 2022. №. 5. С. 8-19.
 8. Берайлик Г.Р. Проектирование информационных систем с помощью микросервисной архитектуры //Инновационные подходы к решению технико-экономических проблем. 2020. С. 187-191.