DOI 10.37882/2223–2966.2022.03.15

АНАЛИЗ МИКРОСЕРВИСНЫХ АРХИТЕКТУР И МОДЕЛЕЙ

ANALYSIS OF MICROSERVICE ARCHITECTURES AND MODELS

I. Irbitskiy
A. Romanenkov
K. Stulnikov
N. Udalov

Summary. Microservice architecture has been mentioned since the beginning of the 21st century, but it has reached the peak of popularity only in our days. In modern product management, even stable, long-established projects are transferred to this architecture, and in new projects, they necessarily consider it as the basis for building, discarding options for a monolithic system, under the influence of new trends in the IT world. So why have microservices become widespread only now? Technological progress has made available key technologies, the absence of which greatly hindered the development and support of this architecture. In particular, CI/CD technologies and the evolution of the positions of developers and system administrators in the labor market in DevOps-engineers who replace a number of specialists. Also, a significant role in the popularity of microservices was played by the strong growth of Internet users and the degeneration of window applications. Now and in the future, microservices will remain in demand due to the already large number of projects based on this architecture.

Keywords: microservice, architecture, development, server, high-load software, queues.

Ирбитский Илья Сергеевич
Аспирант, Московский авиационный институт (национальный исследовательский университет)
scarletsurge.u@gmail.com
Романенков Александр Михайлович
К.т.н., доцент, Московский авиационный институт (национальный исследовательский университет); С.н.с., ФИЦ ИУ РАН
romanaleks@gmail.com
Стульников Кирилл Тимурович
Московский авиационный институт (национальный исследовательский университет)
frostik0409@gmail.com
Удалов Никита Николаевич
Московский авиационный институт (национальный исследовательский университет)
nnudalov@gmail.com

Аннотация. Микросервисная архитектура упоминается с начала 21 века, однако достигла пика популярности только в наши дни. В современном продакт менеджменте даже стабильные, давно внедрённые проекты переводят на данную архитектуру, а в новых проектах, обязательно, рассматривают её как основу построения, отбрасывая варианты монолитной системы, под веянием новых тенденций в мире IT. Так почему микросервисы получили распространение только сейчас? Технологический прогресс сделал доступными ключевые технологии, отсутствие которых сильно затрудняло разработку и поддержку данной архитектуры. В частности, технологии CI/CD и эволюция позиций разработчиков и системных администраторов на рынке труда в DevOps-инженеров, которые заменяют целый ряд специалистов. Также немалую роль в популярности микросервисов сыграл сильный рост пользователей сети Интернет и вырождение оконных приложений. Сейчас и в будущем микросервисы будут оставаться востребованными ввиду уже большого количества проектов, основанных на данной архитектуре.

Ключевые слова: микросервис, архитектура, разработка, сервер, высоконагруженное программное обеспечение, очереди.

Введение

С популяризацией Интернета многократно выросла нагрузка на всевозможные веб приложения. Многие компании, чьи продукты не были готовы к возрастающему трафику в долгосрочной перспективе, производят различные манипуляции для горизонтального расширения функциональности, зачастую весьма неудачные, но достаточные для решения проблемы. Такие расширения по большей части являются дорогостоящими как в плане оборудования, так и в плане рабочей силы. Однако это не означает, что решения монолитных систем являются плохими и не масштабируемыми. Из-за огромного количества программного обеспечения, которое адаптировалось под возрастающий трафик, у IT-сообщества накопилось много опыта по упрощению и удешевлению этого процесса. Монолитным системам тяжелее придерживаться принципа отказоустойчивости ввиду того, что критическая ошибка в любом месте монолита может привести, в лучшем случае, к его перезапуску, а в худшем — к полному отключению всей системы.

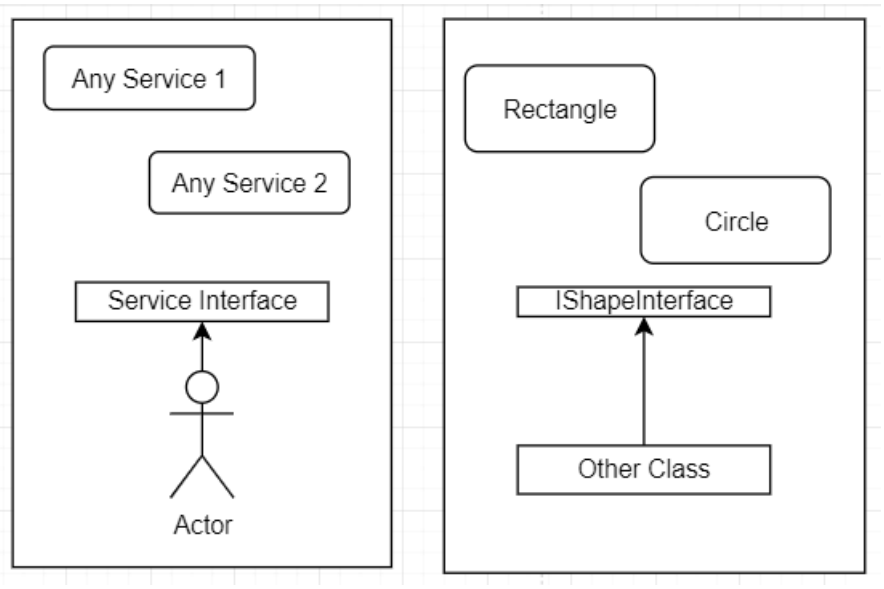

Рис. 1. Аналогия абстракции ООП и микросервисов

Альтернативным решением являются микросервисы — обособленные, маленькие, необязательно самостоятельные, но отказоустойчивые сервисы, рассчитанные под выполнение одной конкретной логической задачи. Микросервисы обладают одним весомым недостатком, из-за которого необходимо очень внимательно подходить к построению систем, основанных на микросервисах. О том, как правильно организовать такие системы и их масштабирование, мы рассмотрим дальше в этой статье.

Обзор микросервисной архитектуры

Микросервисная архитектура [1] прекрасно вписалась в современное программирование за счет предоставления определенного уровня абстракции, который упрощает разработку таких систем. В данном случае можно провести аналогию с объектно-ориентированным программированием (ООП) [2]. Как и в ООП, чем лучше определены зоны действия классов, тем сильнее повышается читаемость и дальнейшее вертикальное расширение функционала системы, ее гибкость к изменениям, так и в микросервисах, необходимо определить зоны ответственности каждого звена системы. Классы ООП в идеальной системе — это совершенно маленькие звенья, выполняющие элементарную функциональность, которую сейчас принято прятать за абстракцией в виде интерфейсов. Такие программы намного проще тестировать, что сокращает расходы на тестировщиков, само тестирование становится автоматизированным и применяется ко всем реализациям одной и той же абстракции. Микросервисы, в свою очередь, также предоставляют из себя интерфейсы, скрывающие реализацию, только уже для одной конкретной логической задачи. На рисунке 1 можно увидеть аналогию работы абстракции, что позволяет сделать любую реализацию заменяемой. Очевидным является факт легкой автоматизации тестирования микросервисов.

Если углубиться в тестирование, то фактически абстракции ООП достаточно для того, чтобы автоматизировать тестирование любой монолитной системы. Но естественным замечанием является то, что идеальных систем не существует и со временем логическое понимание задач классов может расплываться от разработчика к разработчику, а так же с появлением новых технологий, эта тенденция сильнее проявляется при текучке кадров и повышении количества самих классов, это влечет за собой необходимость тестирования, вплоть до всего функционала системы при даже небольшом нововведении, что негативно сказывается на времени выполнения тестов и их логического усложнения. В контексте микросервисов, данная архитектура позволяет нам строго ограничить логическую задачу и обеспечить атомарность тестирования.

Одним из наиболее важных преимуществ микросервисной архитектуры является возможность использовать концепцию независимых, постоянных изменений. Это позволяет бизнесу использовать последнюю версию реализации, что позволяет улучшить пользовательский опыт, так как быстрые откаты и “горячие” изменения (hotfix) автоматически разворачиваются и запускаются в максимально короткие сроки. Эта модульность позволяет одним частям системы функционировать при отсутствии неопределенного количества

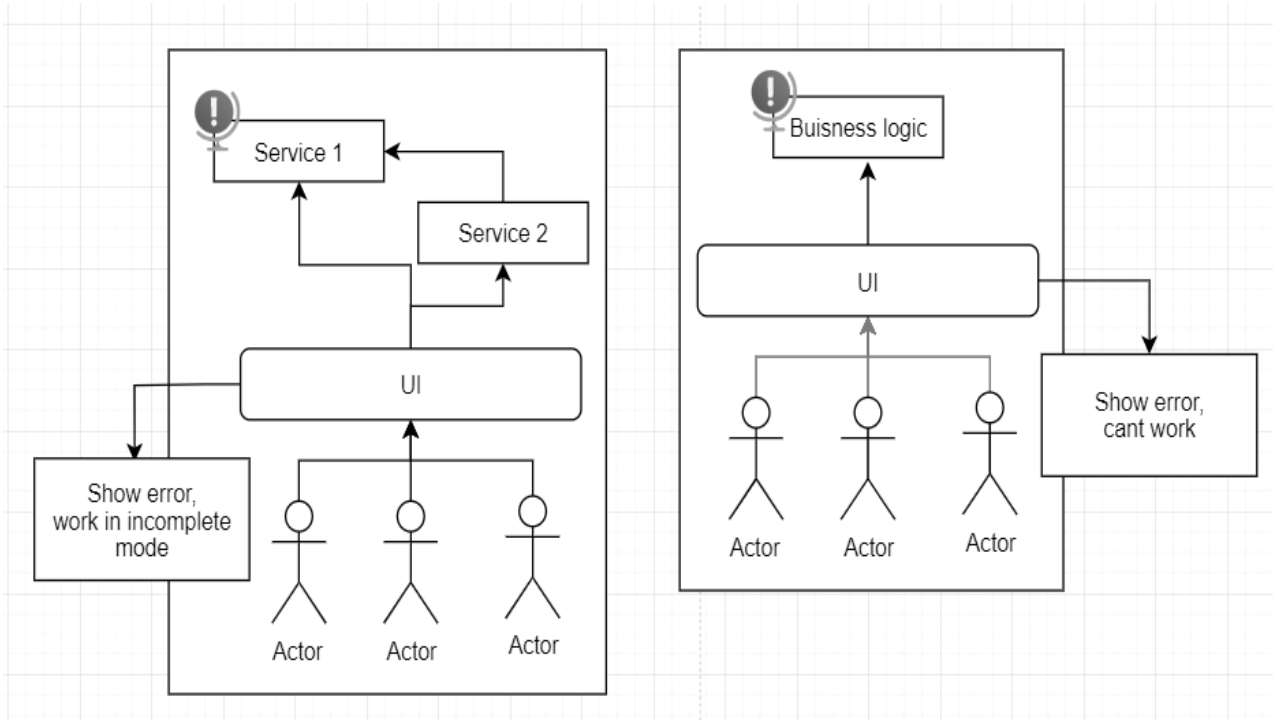

Рис. 2. Разница обработки ошибок в микросервисной архитектуре и монолитной системе

других частей системы, возможно, в режиме ограниченных возможностей, однако выполняя основные функции. Монолитные же системы, при возникновении критической ошибки, нуждаются в полном перезапуске, что влечет немалые временные потери, но что более важно, могут потеряться все данные пользователя или могут не выполниться важные операции [3]. Конечно, все эти проблемы решаются и в рамках монолитных систем, однако необходимое количество для этого ресурсов несоразмерно с таким же функционалам в микросервисной архитектуре.

Каналы связи микросервисов

Обычно внутри монолитной системы каналом связи [4], по факту, выступает одно адресное пространство оперативной памяти и постоянное хранилище. Это однозначно самые быстрые и стабильные каналы связи. Микросервисы в свою очередь независимы и не имеют такой привилегии, что и является их основным недостатком. Любой канал связи микросервисов не сравнится по скорости с монолитной системой, поэтому очень важно не допускать перегрузки эти каналов. Основной способ разгрузки сразу заложен в концепт изолированности микросервисов, поэтому правильным решением является выделение личной базы данных для каждого микросервиса, в которой он хранит только необходимые для своей работы данные. Неправильная загрузка каналов связи может привести к очень долгим ответам и, следовательно, полному нивелированию остальных преимуществ архитектуры, так как скорость работы системы является ключевым фактором удержания пользователей.

REST API

Наиболее гибким и популярным каналом связи является REST API [5]. Посредством протокола HTTPS [6] можно обеспечить передачу любых данных, как через Интернет, так и в локальной среде. REST API является легко документированным, а с помощью проксирования запросов можно заменять конечные микросервисы, для обработки запросов разными экземплярами микросервиса, что также используется для горизонтального расширения, распределения нагрузки или выбора оптимального по репозиции сервера (Рисунок 4), для сокращения задержки ответа. Главными минусами является сложность обеспечения гарантии выполнения запроса, настройка безопасного подключения и обеспечение транзакционности запросов.

Брокер сообщений

Брокеры сообщений [7] предоставляют более гибкий распределительный инструментарий для оперирования сообщениями в цепи системы. Представляет широкий спектр средств горизонтального масштабирования, так как микросервисы выступают конкурент-

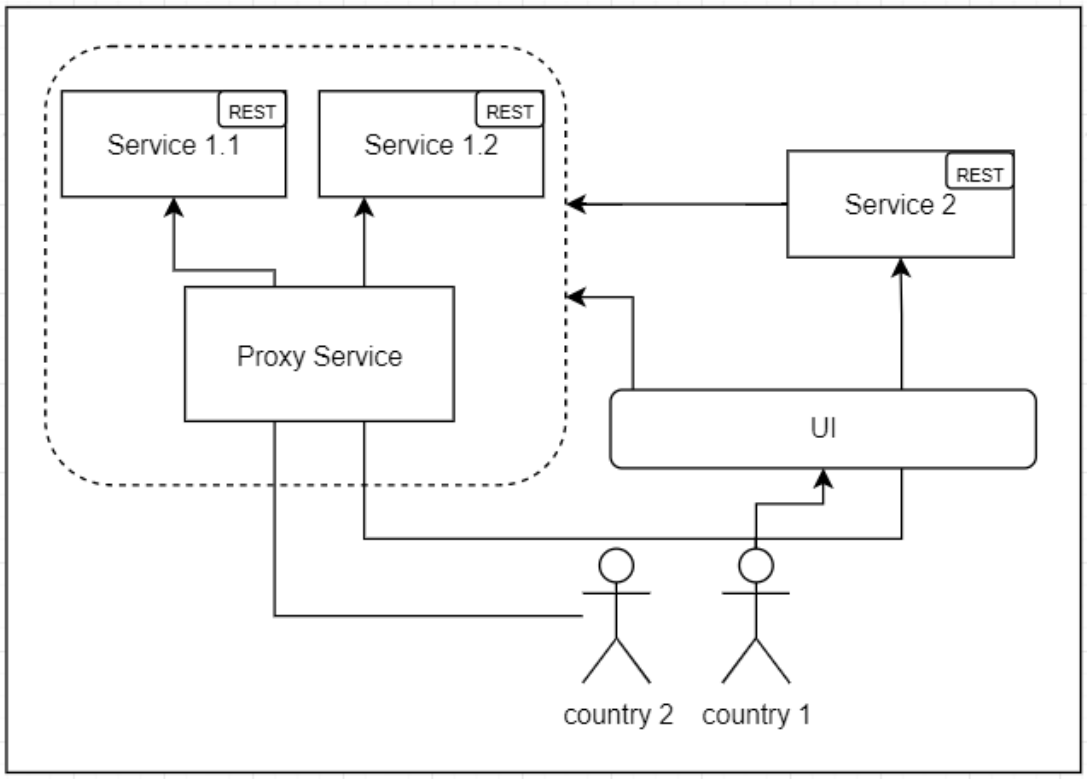

Рис. 3. Организация связи посредством REST API

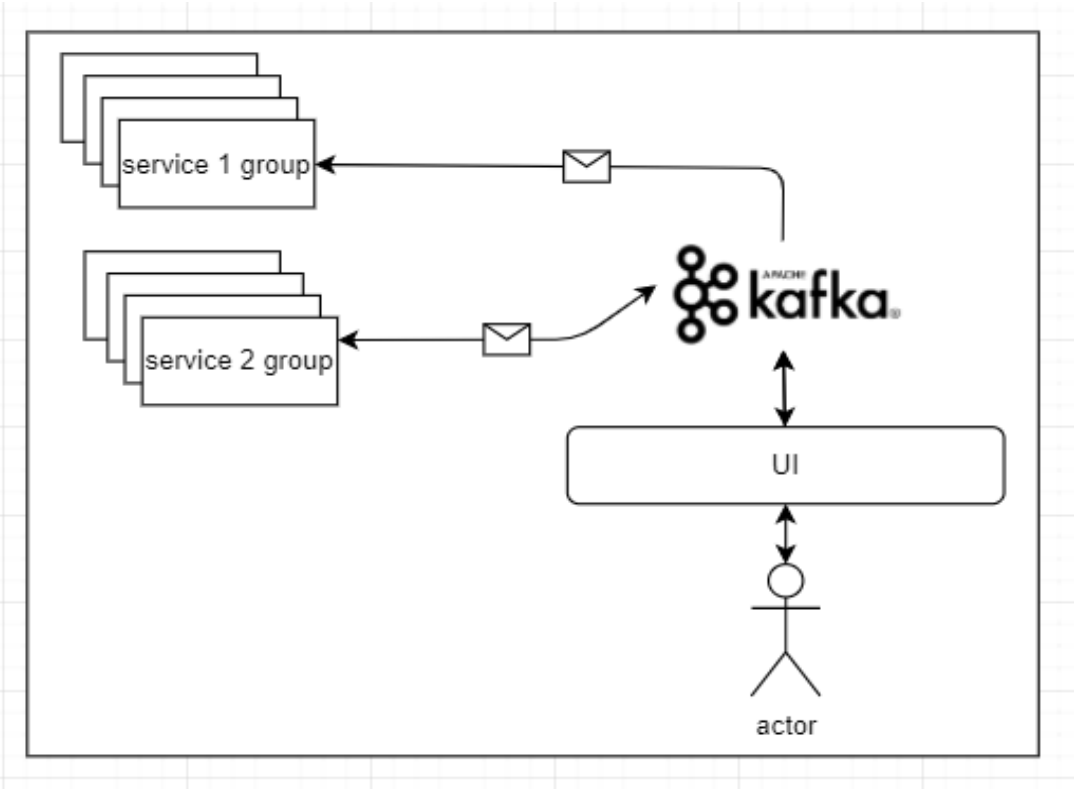

Рис. 4. Обеспечение связи с помощью брокера сообщений

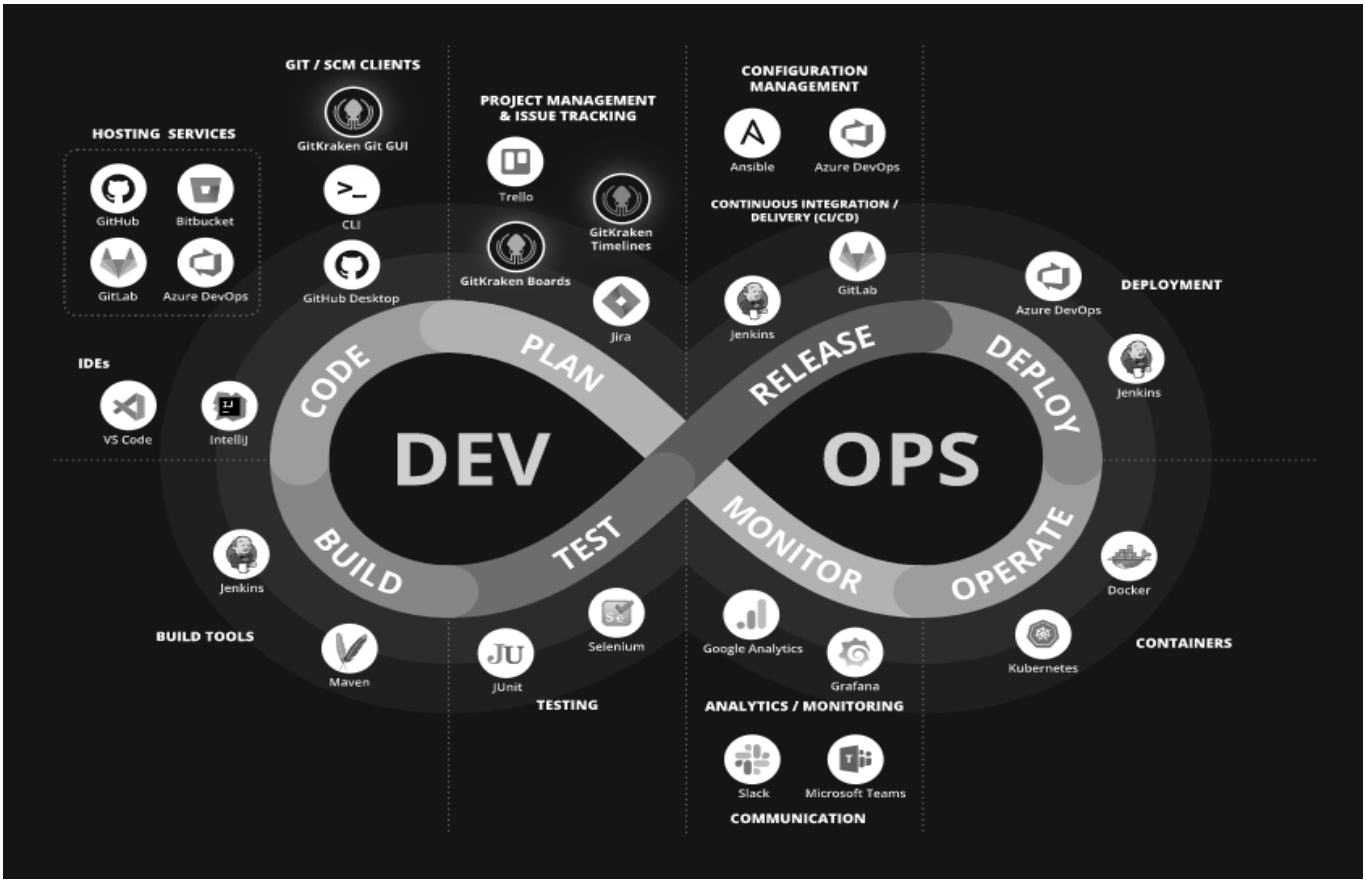

Рис. 5. Continuous Integration технологии

ными слушателями сообщений. Для относительно простых систем применяется RabbitMQ, а для самых сложных бизнес-решений применяется, довольно ресурсоемкая Apache Kafka.

В отличии от REST API, для развертывания очередей необходимы дополнительные мощности, в частности сам брокер-сервер. Однако довольно просто гарантировать выполнение операции или обеспечение транзакционности, все сообщения хранятся в брокер сервере и не потеряются в ходе отключения узлов. Однако увеличивается задержка между сообщениями из-за дополнительного узла, который проходит сообщение прежде, чем достигнет конечной точки. Еще одним недостатком очередей является невозможность подключения к микросервису без дополнительного функционала подключения к очереди. Также могут возникнуть трудности с возвращением результата пользователю, одно из возможных решений — создание обратной очереди.

Web-сокет

Web-сокет [8] позволяет организовать двустороннюю связь с микросервисом. Это может быть полезно, если ведётся работа с агрегирующим микросервисом, где данные подвержены частому изменению. С помощью этой технологии можно добиться обновлений в режиме реального времени. Данный канал связи больше подходит для обеспечения интерфейсной части системы самыми свежими данными. Конкурентом данной технологии является подход long polling (периодический "пинг" сервиса одним и тем же запросом).

Развёртывание микросервисных систем.

Долгое время микросервисные системы не пользовались популярностью ввиду сложности развертывания всей системы целиком и ее тестирования; часто для разработки приходилось разворачивать полную локальную копию всех микросервисов; разработку в таких условиях можно считать тяжелой и непроизводительной. С появлением и развитием технологий Continuous Integration (CI) [9], микросервисная архитектура получила свою популярность, так как её основные недостатки были устранены.

Существование микросервисной архитектуры подразумевает внедрение технологий CI в проект, а также

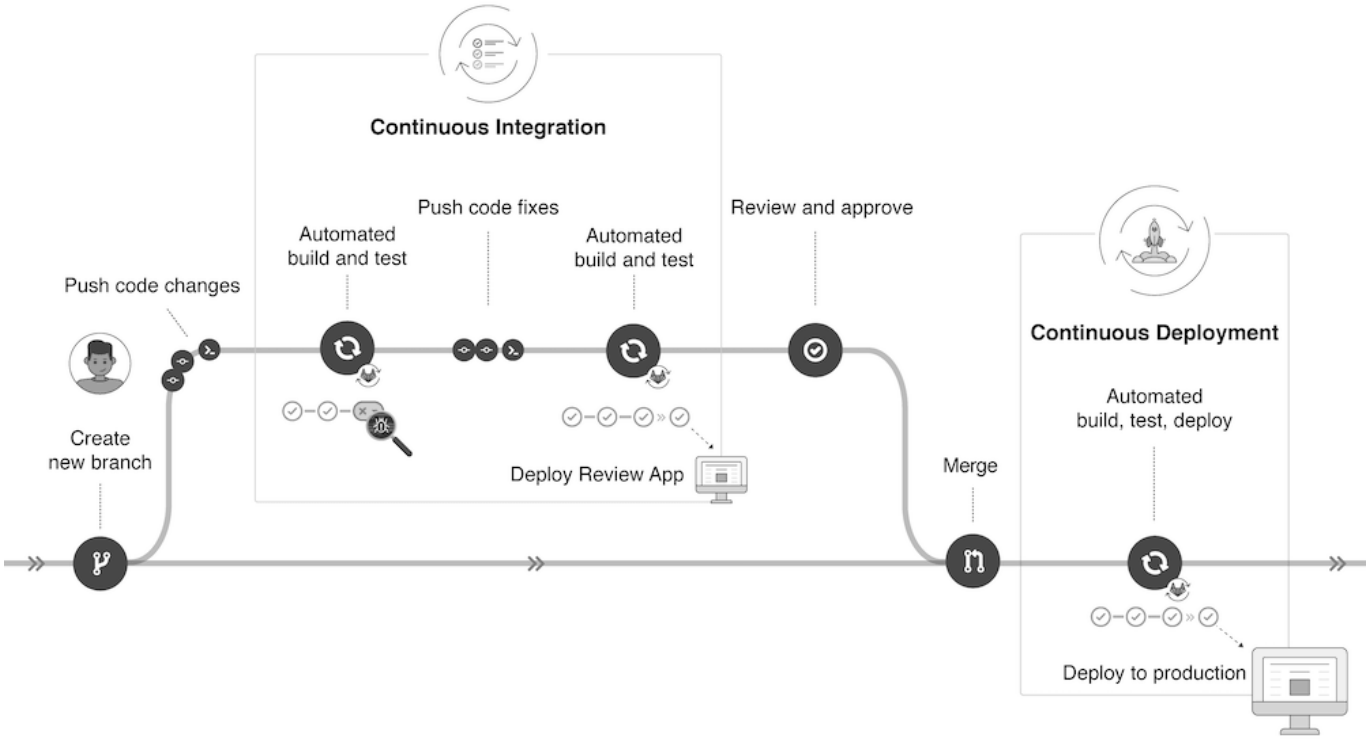

Рис. 6. Gitlab CI/CD

позволяет сократить штат разработчиков и сделать бизнес-процесс проекта максимально гибким.

Стоит обратить внимание на определенные аспекты operations.

Release and Deploy

Как упоминалось выше, серьезный недостаток микросервисов — это тестирование всей системы целиком и развертывание, и обновление. Обычно это выглядит следующим образом. После того, как один из узлов уже подготовлен к выпуску, он передается ответственным за развёртывание на сервере специалистам. Данные специалисты получают исходный код сервиса, собирают его, подготавливают сервер к развёртыванию данного узла. Это занимает много времени и в основном требует отдельного специалиста. Технологии CI/CD решили эту проблему, предоставив функциональные скрипты, которые автоматизируют весь процесс. Фактически единственное, что для необходимо сделать, это запустить сервер или runner и подготовить сам скрипт. Внедрив CI/CD в проект, после того как программист сливает свои изменения в основную ветку, специальные слушатели оповещают runner и он выполняет автоматическую, сборку, тестирование и развёртывает микросервис на определенном сервере или делегирует процесс развёртывания определенному программному обеспечению. Это позволяет отказаться от обычных моделей управления проектами, например от запланированных дат релизов. Изменения могут поступать постоянно без вмешательства человека.

Operate

Развёртывание микросервисов и оперирование ими тоже можно переложить на определенные технологии. Сервер тоже необходимо подготавливать к развёртыванию программного обеспечения. Также обычно на одной серверной машине, обычно, работают несколько различных программ и зачастую может выйти так что, для разных программ могут требоваться разные версии одного и того же вспомогательного программного обеспечения, что будет приводить к конфликтам и применять какие-либо решения. Микросервисы должны быть изолированы, как логически, так и программно. Изолировать модуль можно с помощью инструментов контерейнизации. Это позволяет избавиться от необходимости подготавливать сервер под каждую программу, избежать проблем с неподходящими зависимости, упростить сетевую организацию, а также настроить легкое горизонтальное расширение с помощью балансеров нагрузки управления контеренейзированными приложениями. Это делает возможным использование нескольких стендов систем для упрощения разработки, устранение необходимости локального развёртывания

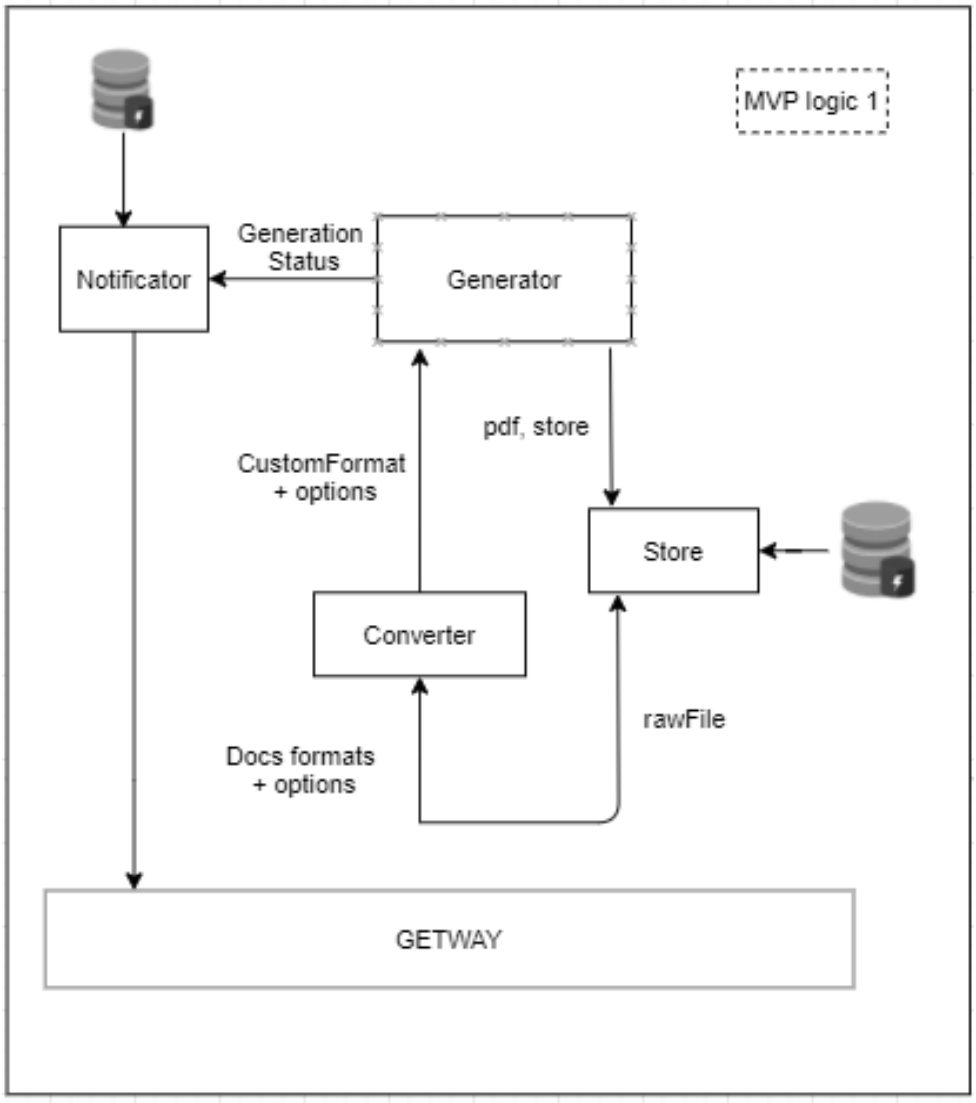

Рис. 7. Пример архитектуры

систем вследствие сокращения времени сборки и увеличения производительности труда.

Практическое применение

Рассмотрим пример микросервисной архитектуры web-системы для конвертации документов в рукописные экземпляры. Для минимальной реализации данной системы необходимы следующие компоненты:

1. Web-интерфейс для взаимодействия с пользователем;
2. Сервис для конвертации документов различных форматов в унифицированную форму для простоты обработки;
3. Сервис генерации рукописных образцов из унифицированных данных;
4. Сервис хранения документов;
5. Сервис нотификации пользователей на стороне web-клиента;
6. Reverse proxy [10] или Gateway для группировки микросервисов в единую систему для конечных пользователей.

Ключевым и самым логически нагруженным сервисом является сервис генерации, именно он реализует ядро всей системы и формирует value для пользователей. Так как сервис является логически нагруженным, причем логикой математической, наверное, самым очевидным языком разработки для этого сервиса будет являться Python, за счет его богатого обилия математическими библиотеками и возможностями. Если же рассмотреть данную систему в контексте монолита, то python фактически позволяет реализовать все перечисленные компоненты системы, однако python явля-

ется динамически типизированными языком и в будущем данную систему будет очень тяжело поддерживать и развивать. Также в Python не очень удобная работа с потоками, что является критически важным фактором при разработке высоконагруженных систем. Микросервисы дают намного больше свободы в выборе технологии и поэтому остальные компоненты можно развивать независимо от ключевого функционала. Однако это скрывает в себе следующую проблему: в монолитной системе один разработчик способен работать над любой его частью, в то время как в микросервисной необходимо наличие специалистов в различных областях для разработки различных микросервисов. В свою очередь большинство специалистов обладают возможность поддерживать большое количество технологии и языков, что нивелирует острую необходимость поиска узконаправленных специалистов.

Рассмотрим архитектуру на рисунке 7. Входной точкой системы является reverse proxy — он отфильтровывает все незарегистрированные маршруты и видоизменяет остальные запросы по определённо прописанным правилам. Это необходимо для абстракции всей внутренней реализации за определенным фасадом, что убирает необходимость прописывать логику общения с различными конечными пользователями.

Следующий этап — это конвертация, в данном случае основным протоколом является REST API. После конвертации файл, сохраняется в микросервисе хранилища также при помощи REST API. После сохранения и конвертации, файл кладется в очередь на генерацию, выбор очереди обусловлен тем, что генерация — математически сложная операция и требует определенных мероприятии по обеспечению горизонтального расширения с самого начала приложения, простые очереди на основе протокола AMQP идеально подходят для этих целей. Возможность развернуть неограниченное количество потребителей генераторов на различных машинах. Из-за долгой операции генерации возникает еще одна проблема — получение результата. Для решения этой проблемы можно использовать web socket, подключения для которого будут жить в отдельном сервисе нотификации, это позволит убрать нежелательные элементы UI, такие как лоадеры и показывать более приятные статусы работы программы.

Заключение

Микросервисы предоставляют множество возможностей для разработки и не подвержены основным проблемам монолитных систем, однако они требуют более строго менеджмента и более опытных команд разработчиков. Данные системы легко масштабируются как горизонтально, так и вертикально, а также за счет абстракции и сокрытия функционала под интерфейсами, появляется возможность легко подменять реализации тех или иных узлов. Микросервисы можно интегрировать в монолитные системы, что также позволит добиться более гибких и надежных решений.

ЛИТЕРАТУРА

1. Микросервисная архитектура: характерные особенности, достоинства и недостатки https://www.tadviser.ru/index.php/Статья: Микросервисная_архитектура:_характерные_особенности,_достоинства_и_недостатки (Дата обращения: 5.11.2021)
2. Объектно-ориентированное программирование: на пальцах https://thecode.media/objective/ (Дата обращения: 6.11.2021)
3. Монолитная VS микросервисная архитектура https://proglib.io/p/monolitnaya-vs-mikroservisnaya-arhitektura-2019–09–16 (Дата обращения: 6.11.2021)
4. Микросервисы (Microservices) https://habr.com/ru/post/249183/ (Дата обращения: 7.11.2021)
5. What is a REST API? https://www.redhat.com/en/topics/api/what-is-a-rest-api (Дата обращения: 2.11.2021)
6. What is HTTPS? https://www.cloudflare.com/learning/ssl/what-is-https/ (Дата обращения: 4.11.2021)
7. Немного о брокерах сообщений — Kafka и RabbitMQ http://nlpx.net/archives/566 (Дата обращения: 2.11.2021)
8. RFC6455 — The WebSocket Protocol https://datatracker.ietf.org/doc/html/rfc6455 (Дата обращения: 8.11.2021)
9. Что такое непрерывная интеграция? https://aws.amazon.com/ru/devops/continuous-integration (Дата обращения: 9.11.2021)
10. Reverse proxy https://www.imperva.com/learn/performance/reverse-proxy/ (Дата обращения: 9.11.2021)