1. Добрый день.

Меня зовут Виталий. И сегодня мы поговорим о таком подходе к построению архитектуры как микросервисы

2. Эта презентация будет состоять из двух частей.

В первой мы поговорим о том:

* Что такое микросервисы,
* как они позволяют ориентироваться на бизнес
* и как подготовить для них инфраструктуру

Во второй мы рассмотрим практический подход к дизайну микросервисов

Презентация называется микросервисы на практике неспроста – со всеми нюансами, всеми подходами описанными здесь наша команда столкнулась на практике при создании нашего приложения

3. Итак, что же такое микросервисы?

Традиционно вся реализуемая система, вся логика упаковывается в одно большое приложение. Такое приложение называется монолитным, потому что разрабатывается, выпускается и деплоится оно единым целым.

При микросервисном подходе, напротив, вся система представляет собой набор сервисов, которые достаточно слабо связаны друг с другом и потому могут разрабатываться и деплоиться отдельно друг от друга.

Этот подход к архитектуре формировался и применялся в том или ином виде на продолжении многих лет, но более-менее подробно был описан сравнительно недавно.

4. Для заказчика Copyright Clearance Center наша команда разработала приложение которое называется Rightslink for Open Access (ROA сокращенно). Оно позволяет оплачивать публикацию авторами статей в журналах под открытыми лицензиями. И для корректной работы оно должно представлять достаточно широкий спектр функциональности:

* Управление справочниками издателей
* Предоставление скидок
* Формирование отчетов
* Работа с кредитками

И если бы РОА создавалось по монолитной архитектуре, то вся эта функциональность включалась бы в приложение в качестве кода или библиотек

5. И если бы весь спектр услуг Copyright Clearance Center создавался по монолитной архитектуре, то наше приложение было бы еще одним кусочком еще большего монолита, супермонолита

6. Но вместо этого РОА как и другие приложения системы были реализованы по микросервисной архитектуре, когда многие части, которые не относятся к бизнесу приложения напрямую, выносятся в отдельные слабо связанные сервисы.

Так, управление кредитными карточками (добавление, удаление) или работа со скидками были выделены в отдельные сервисы, оставив нашему приложению то что входит в его компетенцию – работу с авторами, статьями и покупкой прав.

Ключевые особенности такой архитектуры:

* Сервисы деплоятся на разные сервера
* И взаимодействуют друг с другом через РЕСТ

7. Естественно, микросервисы не являются серебряной пулей, как и у любого подхода, у него есть плюсы и минусы.

К плюсам традиционно относятся

* явно прописанный протокол взаимодействия
* Простота отдельно взятого сервиса, поскольку он концентрируется лишь на собственной бизнес-логике
* За счет простоты повышается скорость разработки
* Команды становятся слабо зависимыми друг от друга
* Технологии используемые в одном приложении могут отличаться от используемых в другом

8. Вместе с тем есть и минусы:

* Система, за счет распределенности и необходимости работы по сети становится сложнее и более подверженной ошибкам
* Как результат, сложность разработки также повышается
* Сложность системы приводит и к более трудоемкому развертыванию, поэтому частичная или полная автоматизация является практически обязательной

9. В этой презентации я буду ссылаться на основные особенности микросервисной архитектуры, описанные в статье мартина фаулера (они будут подчеркнуты)

Поэтому если вы видите сложный термин или упаси боже термин на английском, то это оттуда

Первая особенность, первый подход – это ориентация вокруг потребностей бизнеса

При традиционном подходе команда часто разделяется на тех кто работает на уровне базы данных, уровне доступа к данным, на уровне фронтенда.

При микросервисном подходе, за счет сравнительной простоты сервиса разработчик просто вынужден работать на всех уровнях и понимать бизнес клиента.

При этом создаются разные сервисы под разные нужды заказчика. Эти сервисы создаются отдельными командами. Каждая команда отвечает за свой сервис, в том числе и при поддержке на продакшене. Различные проблемы возникающие на продакшене, стимулируют к созданию вспомогательных утилит для диагностики и исправления исключительных ситуаций. В нашем случае это привело к созданию отдельного приложения, которое изначально даже не планировалось, но оказалось существенным подспорьем

Это приложение позволяет:

* Просмотреть внутреннее состояние системы и ее компонентов
* Изменять статусы транзакций и работ автора
* И другие вспомогательные и диагностические работы

10. Инфраструктура для микросервисов, в силу их распределенности, также играет важную роль

Для микросервисной архитектуры важно уметь абстрагироваться от инфрастуктуры и предъявлять минимум требований к подготовке окружения

Практической необходимостью является наличие нескольких окружений – у нас есть отдельные окружения для разработки, тестирования, продакшен. При этом инфраструктура этих окружений должна быть одинаковой (ОС, сервер приложений, версия явы), это позволит исключить множество странных и плохо диагностируемых проблем.

Развертывание инфраструктуры должно быть по возможности максимально автоматизировано.

11. Для микросервисов автоматизация крайне важна из-за их количества и общей сложности развертывания системы.

Мы используем следующие практики:

* Управление циклом выпуска приложения (через CI). Приложение постоянно собирается на дженкинсе для контроля ??корректности??. Он же помогает собрать и задеплоить финальную версию приложения на окружение разработчика для проверки интеграции сервисов друг с другом.
* Развертывание производится нажатием одной кнопки
* Структуры БД создаются и изменяются эволюционно, согласно требованиям текущего состояния приложения. Каждый раз при деплое новая порция изменений накатывается на БД, гибко изменяя ее.
* Уже существующие данные необходимо адаптировать под изменившуюся структуру БД выполняя автоматизированную миграцию данных
* Скрипты создания/обновления БД автоматически запускаются при деплое (том самом, одной кнопкой)
* Само окружение (установка ява машины, установка и настройка сервера приложений) автоматизировано с помощью шеф-скриптов

12. Если ранее мы говорили о дизайне системы в виде набора микросервисов, то сейчас мы рассмотрим практические подходы к дизайну конкретной шестеренки этой системы, конкретного сервиса.

Первый подход, который мы использовали – децентрализованное управление

13. Этот подход несмотря на сложное название, достаточно прост – каждая команда работающая над сервисом принимает решение о том как достигнуть цели создания своего приложения самостоятельно.

В чем же заключается это решение?

Для разных сервисов и целей подходящими могут быть разные фреймворки, БД. Например в нашем сервисе используются постгрес и монго, в другом – хазелкаст, в третьем – амазон с3

Архитектурные решений (как управлять транзакциями, ???реализовавывть взаимодействи????) также могут варьироваться

Однако при это у нас есть общая платформа – просто чтобы не плодить копи-паст и иметь экспертизу по выбранным технологиям. Общая платформа вводит единые стандарты кода, систему сборки Gradle, библиотеки (активно используется Guava)

14. Мы рассмотрели подход к децентрализации управления.

А второй подход также рассматривает децентрализацию, но уже данных

15. Каждый микросервис может использовать свою базу данных, ту, которая лучше всего подходит под его нужды. Также приветствуется то что называется Polyglot persistence, когда отдельные части информации распределены по разным базам данных. В случае нашего приложения статьи авторов (которые представляют собой сложные документы) мы храним в монго, а заказы в реляционной БД.

Важнейший принцип, показанный на рисунке – клиент микросервиса никогда не получает доступ к БД напрямую, вместо этого он использует API предоставляемый микросервисом (чаще всего в виде РЕСТа)

16. Мы рассмотрели децентрализацию данных и теперь стоит задаться вопросом:

Как же взаимодействуют микросервисы друг с другом?

17. В принципе, протокол, по которому они взаимодействуют, может быть любым, но наиболее распространенными являются

* Выставление РЕСТ АПИ
* И обмен асинхронными сообщениями (в случае Java платформы чаще всего это JMS сообщения)

При проектировании РЕСТ АПИ важно соблюдать версионирование, чтобы разработчики приложений использующих ваш микросервис узнали, изменился ли он, остался ли обратно совместимым или необходимо пересмотреть правила его использования

Также важное правило при проектировании РЕСТ АПИ – состояние клиента (сессии) не хранятся на сервере. Каждый вызов РЕСТ сервиса самодостаточен и может быть перенаправлен на совсем другой сервер чем предыдущий вызов. Таким образом микросервисы позволяют горизонтально масштабироваться

18. Поскольку система из составленная из микросервисов по сравнению с монолитным приложением более хрупкая, отказы отдельных микросервисов (из-за проблем сети, серверов, баз данных) это неизбежное зло, которое нужно учитывать при проектировании

19. При проектировании под отказ необходимо для начала найти отказ. Для определения, все ли части приложения корректно работают, во всех модулях у нас используется подробное логгирование с использованием библиотеки logback, а события с уровнем warn или error отслеживаются спланком, который рассылает команде разработки и поддержки письма со стектрейсами

После того как проблема найдена, ее необходимо исправить и чем быстрее, тем лучше. Тут есть два подхода:

Автоматическое исправление и ручное.

Автоматическое исправление исправляет большую часть ошибок.

Так, автоматическая повторная попытка (когда мы просто еще раз повторяем действие) позволяет исправить ошибки связанные с временными проблемами сети или недоступностью БД. А High Availability позволяет спрятать падение сервера, перекинув запрос на другой сервер.

Ручное же исправление требует участия человека и отдельного внимания к каждой найденной ошибке, когда автоматическое исправление ничем помочь не может

20. Мы рассмотрели как предусматривать отказы в системе, теперь стоит взглянуть на то, какие типы взаимодействия бывают, потому что при проектировании микросервиса важно выбрать тип его взаимодействия с другими сервисами.

21. Взаимодействие может быть либо синхронным, когда клиент ожидает что запрашиваемые данные будут возвращены тут же, в цикле создание-чтение-обновление-удаление это обычно чтение. Например, просмотр заказа – клиент ожидает что все детали заказа будут возвращены тут же

Либо асинхронным, когда выполнение действия прямо сейчас клиенту не так и важно, главное что запрос принят на обработку. Например, когда клиент заказывает электронную отсканированную копию документа, он не ждет пока документ отсканируется, а продолжает работу. А отсканированная копия позже будет прислана по емейлу. Это могут быть (хоть и необязательно) создание-обновление-удаление

22. При синхронном взаимодействии сам запрос должен обрабатываться максимально быстро. Если он длится больше 2-3 секунд, имеет смысл сделать его асинхронным

Например, в нашей системе создание заказа состоит из 3 шагов:

* Сохранение заказа в БД микросервиса
* Снятие денег с карточки за заказ
* Создание лицензии в финансовой системе

Из этих действий первые 2 по архитектуре мы сделать асинхронными никак не можем, потому что они требуют внимания пользователя. Последнее (создание лицензии) вполне можно сделать асинхронным, так как оно довольно тяжеловесно, а в случае ошибки пользователь все равно никак это не исправит

Также при получении данных от микросервисов имеет смысл использовать кэширование. Если объем данных небольшой, можно использовать внутрипроцессный Guava кэш как одно из самых легких и быстрых решений

23. При асинхронном взаимодействии запрос клиента принимается микросервисом

24. сохраняется

25. клиенту возвращается идентификатор принятого запроса

26. Клиент сохраняет на своей стороне идентификатор запроса и занимается своими делами. Позже по идентификатору он может периодически проверять статус работы.

27. Микросервис же выполняет асинхронную обработку запроса

28. Когда фоновая обработка завершена, микросервис обновляет статус запроса в БД

29. и может уведомить клиента асинхронным сообщением о том, что работа выполнена.

30. Или клиент сам запросит статус если не поддерживает обмен сообщениями

Как правило, некоторые данные (статус обработки запроса, идентификатор и так далее) сохраняются как в БД клиента так и в БД сервиса, что приводит к некоторой избыточности, но она необходима для диагностики отказов

31. Возвращаясь к теме отказов на примере этих двух типов взаимодействия (синхронного и асинхронного):

Необходимо быть готовым к отказу и проектировать приложения, считая что если в этом месте что-то может отказать, то это случится. Асинхронная обработка в предыдущем примере может не быть закончена из-за падения сервера

Быть готовым к восстановлению данных или состояния системы. Например, если в предыдущем примере асинхронная обработка упала, мы должны предписать что должна делать служба поддержки

Также необходимо найти точки отказа. В предыдущем примере их более чем достаточно:

* БД микросервиса может быть недоступна
* БД клиента может быть недоступна
* Асинхронная обработка упала
* Отсылка асинхронного сообщения не удалась

Определить как восстанавливать:

Синхронное взаимодействие – на стороне клиента, показать пользователю ошибку и попросить выполнить действие еще раз

Асинхронное- на стороне сервиса обработка не удалась, автоматически попробовать выполнить ее еще раз

Ответственный в этом случае разработчик, который должен предусмотреть пути восстановления

32. Автоматическая повторная доставка на самом деле довольно мощный инструмент.

Лучше всего она подходит для асинхронных сообщений. В этом случае на промежуточное хранилище для сообщений (так называемый брокер) приходит запрос от клиента, который помещается в очередь1.

33. Консьюмер читает сообщение и выполняет обработку запроса

34. Если она неудачна, обработка повторяется еще раз.

35. Если в течении нескольких попыток обработать сообшение автоматически корректно не получилось, оно перемещается в специальную очередь DLQ (Dead Letter Queue).

36. Эту очередь периодически просматривает служба поддержки и пытается исправить проблему

37. либо изменяет само сообщение либо состояние системы и пересылает сообщение обратно в очередь1, откуда снова начинается цикл.

Итак, мы рассмотрели подходы которые используются в нашем приложении при создании микросервисов.

Но при этом возникает вопрос когда проектировать приложение по монолитной архитектуре, а когда использовать микросервисный подход.

38. Система достаточно понятна

* + Если вы можете сказать глядя на приложение, как реализован тот или иной функционал и не возникает вопроса «Что здесь происходит?»
* Система достаточно размыта
  + Когда сложно разбить систему на слабо связанные сервисы с ограниченными доменными областями

39. Система чересчур большая

* + просто не помещается на одном сервере
* Команда чересчур большая
  + И распределенная
* Система чересчур сложная
  + Сложная в понимании
  + Сложная в разработке
  + Сложная в развертывании

В этом случае разбиение на микросервисы не решит сложность системы, но позволит от нее абстрагироваться

* Система слишком новая
  + Изолировать легаси приложения
    - Именно так и зародилось наше приложение

40. Упрощают понимание – за счет необходимости разбиения системы на слабосвязанные сервисы

Упрощают масштабирование – за счет возможности создания множества серверов с развернутыми микросервисами

Подбирая нужный инструмент для каждой цели, мы используем максимально удобный способ решения бизнес-задач

41. На этом презентация завершена, и я хотел бы узнать, есть ли у вас вопросы