# Меня хорошо слышно && видно?

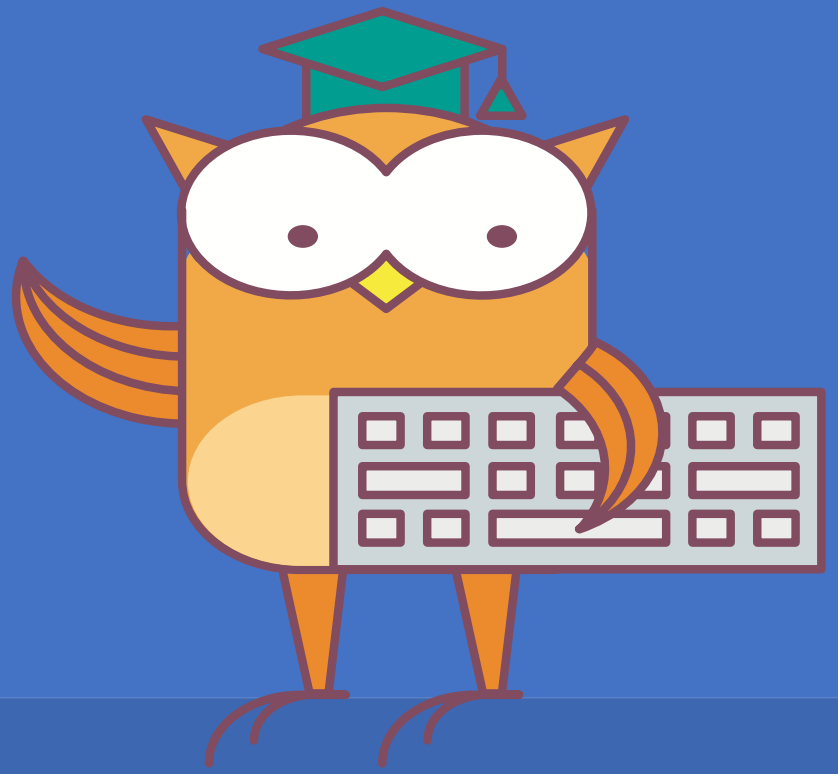
Напишите в чат, если есть проблемы!

Ставьте + если все хорошо

OTUS

# Асинхронный и синхронный API

Архитектор ПО



Щетинников Стас

# Преподаватель

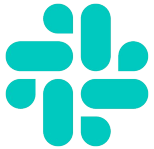
## Непомнящий Евгений

- 15 лет программировал контроллеры на C++ и руководил отделом разработки
- 3 года пишу на Java
- Последнее время пишу микросервисы на Java в Мвидео

# Правила вебинара

Активно участвуем

Задаем вопросы в чат



Off-topic обсуждаем в Slack #канал группы или #general

Вопросы вижу в чате, могу ответить не сразу

## Карта курса

5 Архитектор

1 Инфраструктурные паттерны

3 Распределенные системы

**2 Коммуникационные паттерны**

4 Децентрализованные системы

Опрос по программе - каждый месяц в ЛК

# Цели занятия

- рассмотреть основные типы межсервисного взаимодействия;
- обсудить версионирование API;
- рассмотреть описание API.

# Карта вебинара

- Асинхронный и синхронный API
- Message Bus
- CQRS
- Оркестрация и хореография
- Версионирование API
- IDL, API design first
- Rich vs Anemic

# 01

# Синхронное и асинхронное API

- Асинхронный и синхронный API
- Message Bus
- CQRS
- Оркестрация и хореография
- Версионирование API
- IDL, API design first
- Rich vs Anemic

# API

API (программный интерфейс приложения) — описание способов (набор классов, процедур, функций, структур или констант), которыми одна компьютерная программа может взаимодействовать с другой программой.

API реализуется с помощью каких-то протоколов обмена

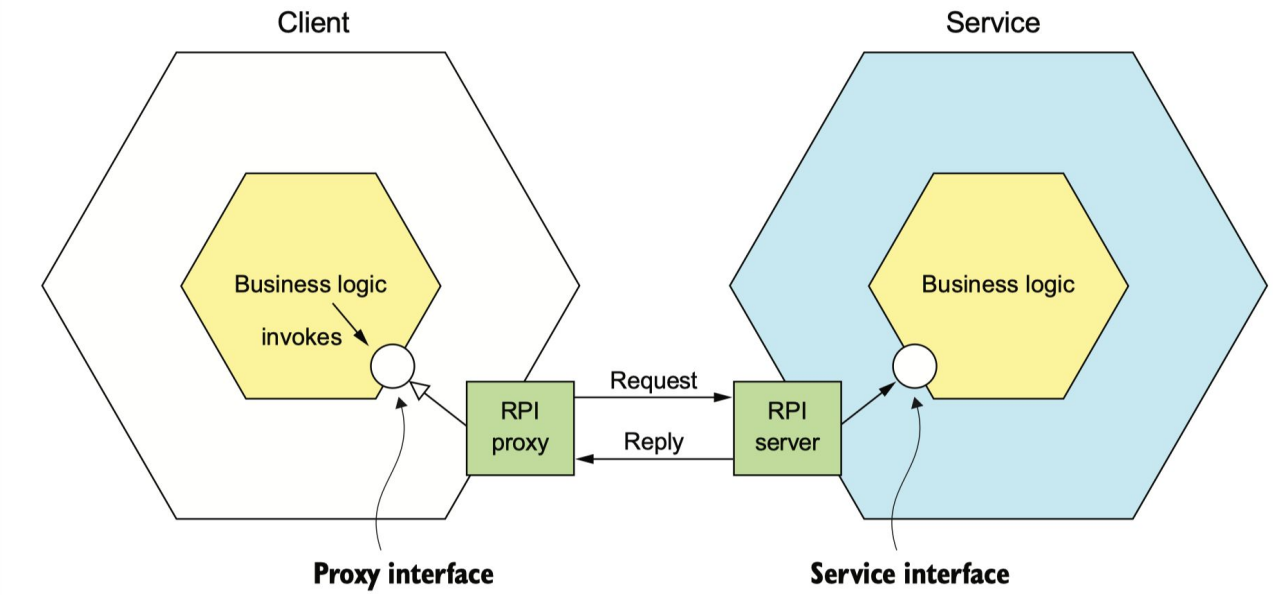
**Какие протоколы вы знаете?**

# Request-reply протоколы взаимодействия

**Request-reply** - протоколы, в которых клиент посылает запрос, сервис обрабатывает и отправляет ответ

**Какие примеры вы знаете?**

**Примеры:**
HTTP: REST, SOAP, graphQL
gRPC

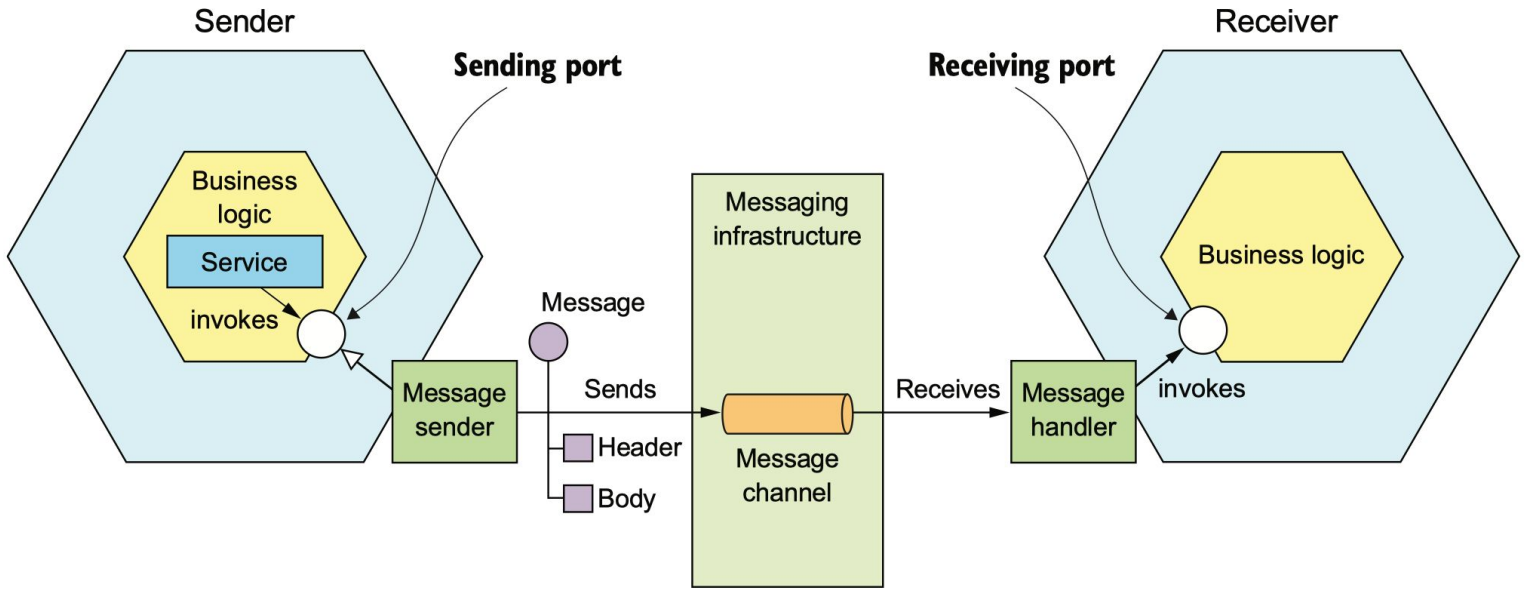


# Message-based протоколы взаимодействия

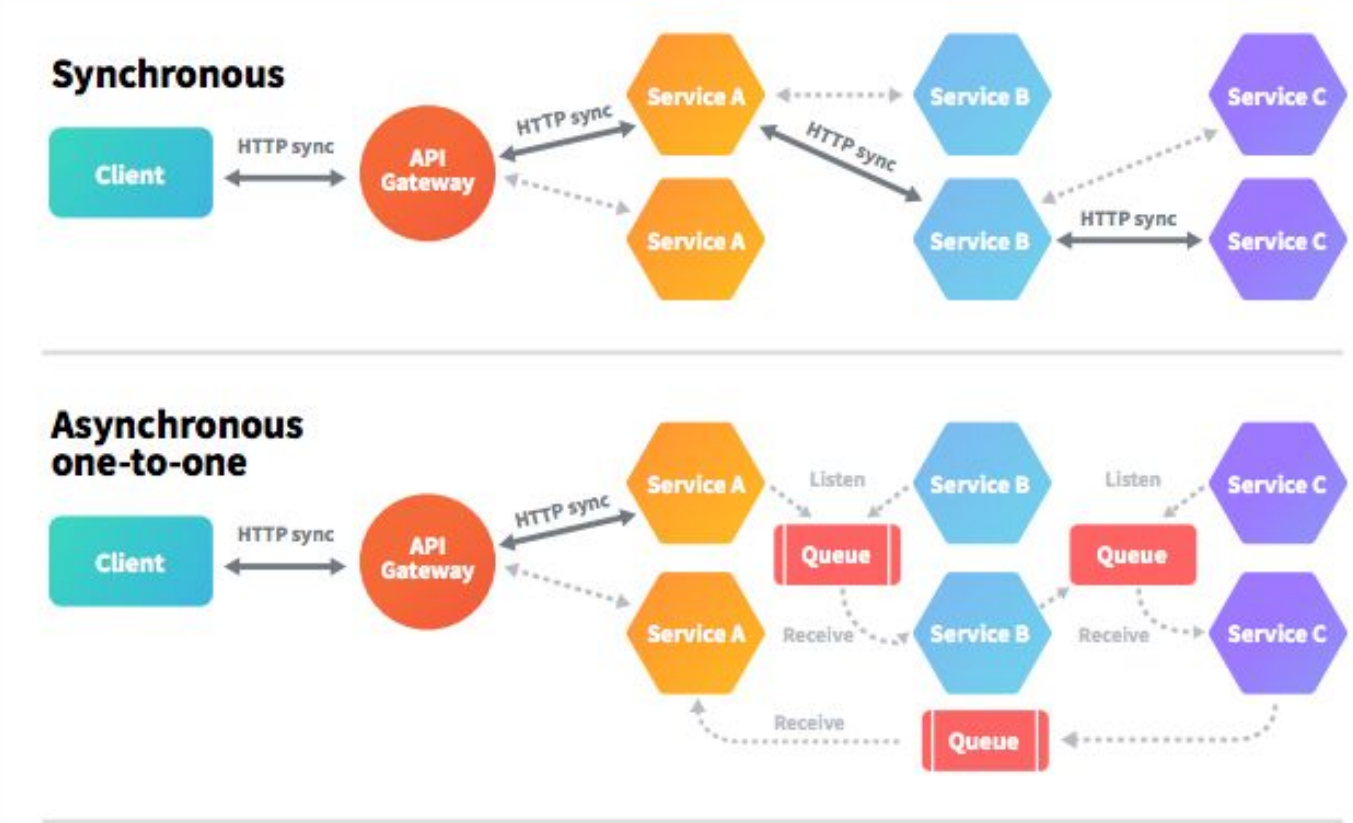
**Message-based** - протоколы, в которых клиент отправляет сообщение через брокер сообщений, а сервис сообщение читает из брокера сообщений

**Какие примеры вы знаете?**

**Примеры:**
Amqp, Kafka, Mqtt, ZeroMQ и т.д.



# Синхронное и асинхронное взаимодействие




Плюсы и минусы синхронное vs асинхронное

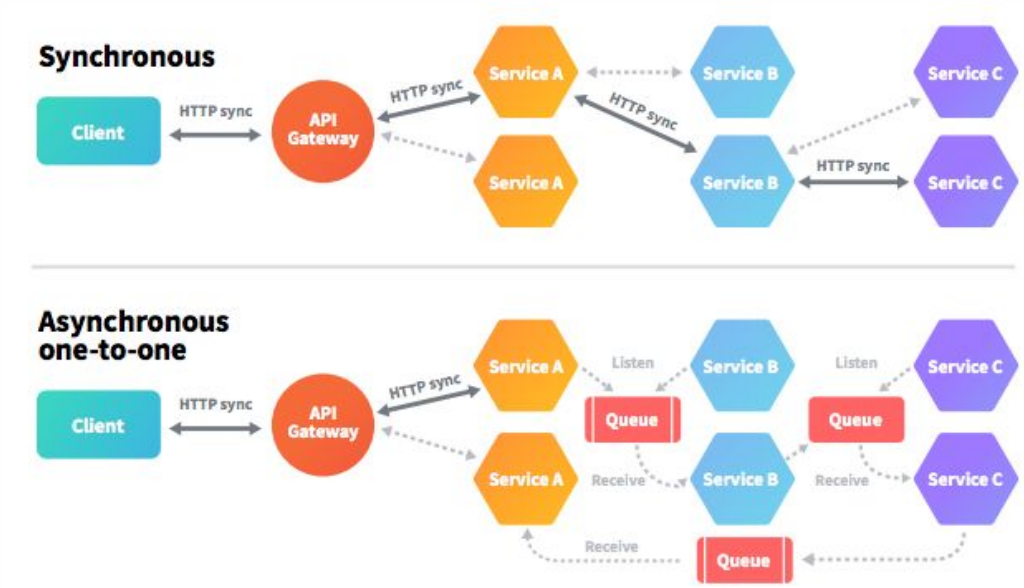
# Request-response vs message based

**Request-response**:

- Проще дебаг
- Latency ниже

**Message based**

- Дебаг сложнее
- Latency за счет посредников выше
- Разделение сервисов
- Роутинг
- Массовая отправка (kafka)



# Семантика взаимодействия

Следует различать семантику взаимодействия и транспортный протокол.

Request-Reply, Async, Pub/Sub – это «транспортный» уровень взаимодействия.

**Можете привести пример асинхронной семантики поверх синхронного транспортного уровня?**

# Синхронное взаимодействие

**Синхронное взаимодействие** – это такое взаимодействие при котором семантика сервиса А такова, что он не может продолжить выполнение запроса, если не получил ответа от Б.

**Например**

- Сервис «заказ» не может продолжить свою работу, пока не убедится, что пользователь обладает необходимыми правами для удаления заказа

- Сервис «оформления заказа» не может закончить свою работу, пока не придет подтверждение от биллинга об оплате.

Еще это называется функциональная зависимость. Когда результаты одной функции необходимы для работы другой.

# Асинхронное взаимодействие

**Асинхронное взаимодействие** – это такое взаимодействие при котором сервис А в соответствии с семантикой предметной области может продолжить выполнение,  не дожидаясь ответа от Б.

Например

- Сервису «регистраций» достаточно сообщить о факте создания нового пользователя, ему не нужен ответ от сервиса «нотификаций».
- Формирование отчетов брокера или налоговой

# Синхронное и асинхронное взаимодействие

**Синхронное взаимодействие** возможно и с помощью message-based протоколов (amqp RPC)

**Асинхронное взаимодействие** возможно с помощью request-reply протоколов (long polling, webhooks, async API)

# Синхронное и асинхронное взаимодействие

Правило большого пальца

- request-reply протокол для синхронных взаимодействий
- message-based протокол для асинхронного взаимодействия.

# Синхронное и асинхронное взаимодействие

Синхронное взаимодействие нельзя поменять на асинхронное просто поменяв транспортный уровень. Например, заставив всех общаться через общую очередь или по REST-у.

Такое изменение возможно только с изменением семантики работы сервисов, которое приводит к изменению и других характеристик системы – транзакционность, консистентность, время ответа, надежность, простота разработки.

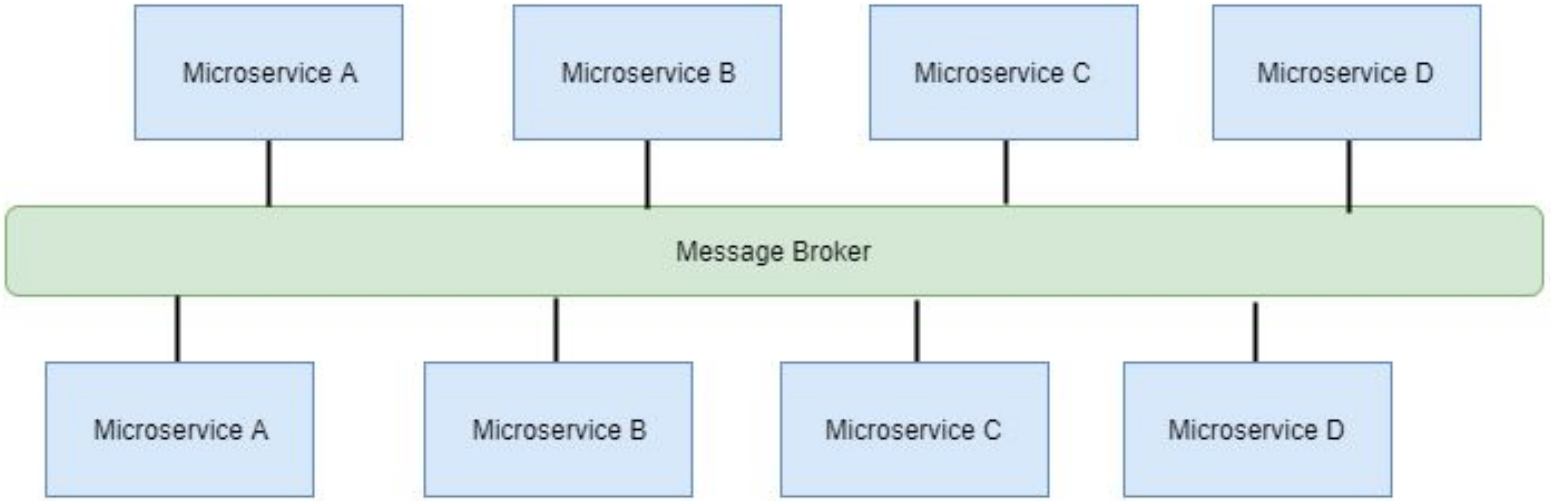
# 2

# Message bus паттерн

- Асинхронный и синхронный API
- Message Bus
- CQRS
- Оркестрация и хореография
- Версионирование API
- IDL, API design first
- Rich vs Anemic

# Message BUS

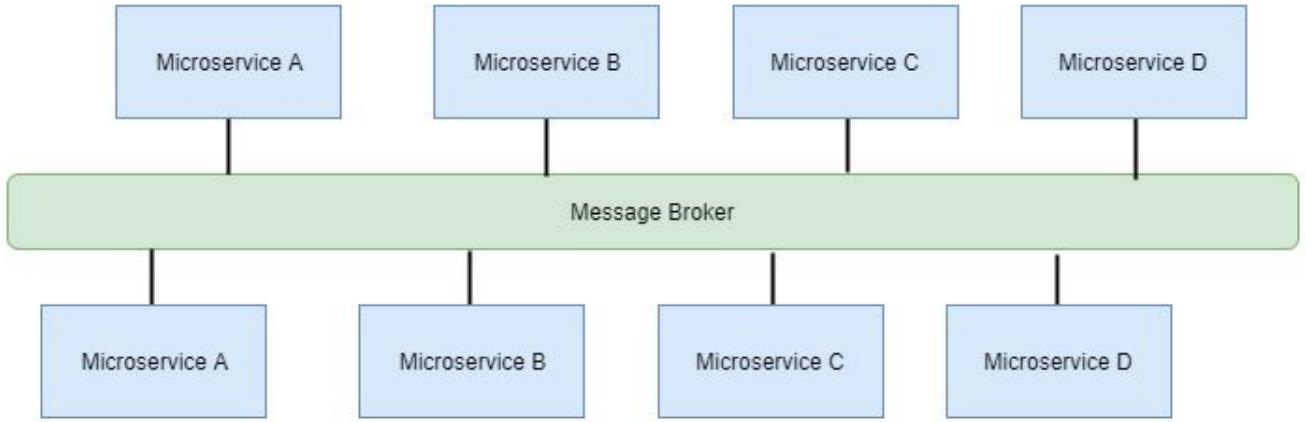
А почему бы всем сервисам не начать взаимодействовать асинхронно (через очередь)?



# Плюсы message bus

Особенности такого решения:

- Service discovery - сервисам не нужно знать где и кто находится, достаточно просто бросить сообщение в очередь
- Выше доступность – если сервис упал, то он просто не берет запросы из очереди
- Ниже вероятность каскадных падений
- Горизонтальное масштабирование до некоторых пределов
- Увеличивает эффективное использование ресурсов сервисов, написанных на языках не поддерживающих асинхронное программирование




https://www.reactivemanifesto.org/ru

# Минусы message bus

Минусы message bus

- Latency увеличивается
- Debug становится сложнее
- Single Point Of Failure в виде брокера сообщений

https://www.reactivemanifesto.org/ru

# Особенности message bus

В микросервисной архитектуре довольно большая часть взаимодействий все-равно будет синхронной – например, когда нам нужно получить какие-то данные от какого-то сервиса.

- Для синхронного взаимодействия крайне неудобно использование асинхронного протокола (пробрасывание correlation-id и т.д и т.п)
- Оркестратора(типа Kubernetes-а) закрывает потребность в service discovery, горизонтальном масштабировании, доступности: retry-ями и circuit-breaker-ами

Поэтому в чистом виде message bus в современном мире встречается редко.

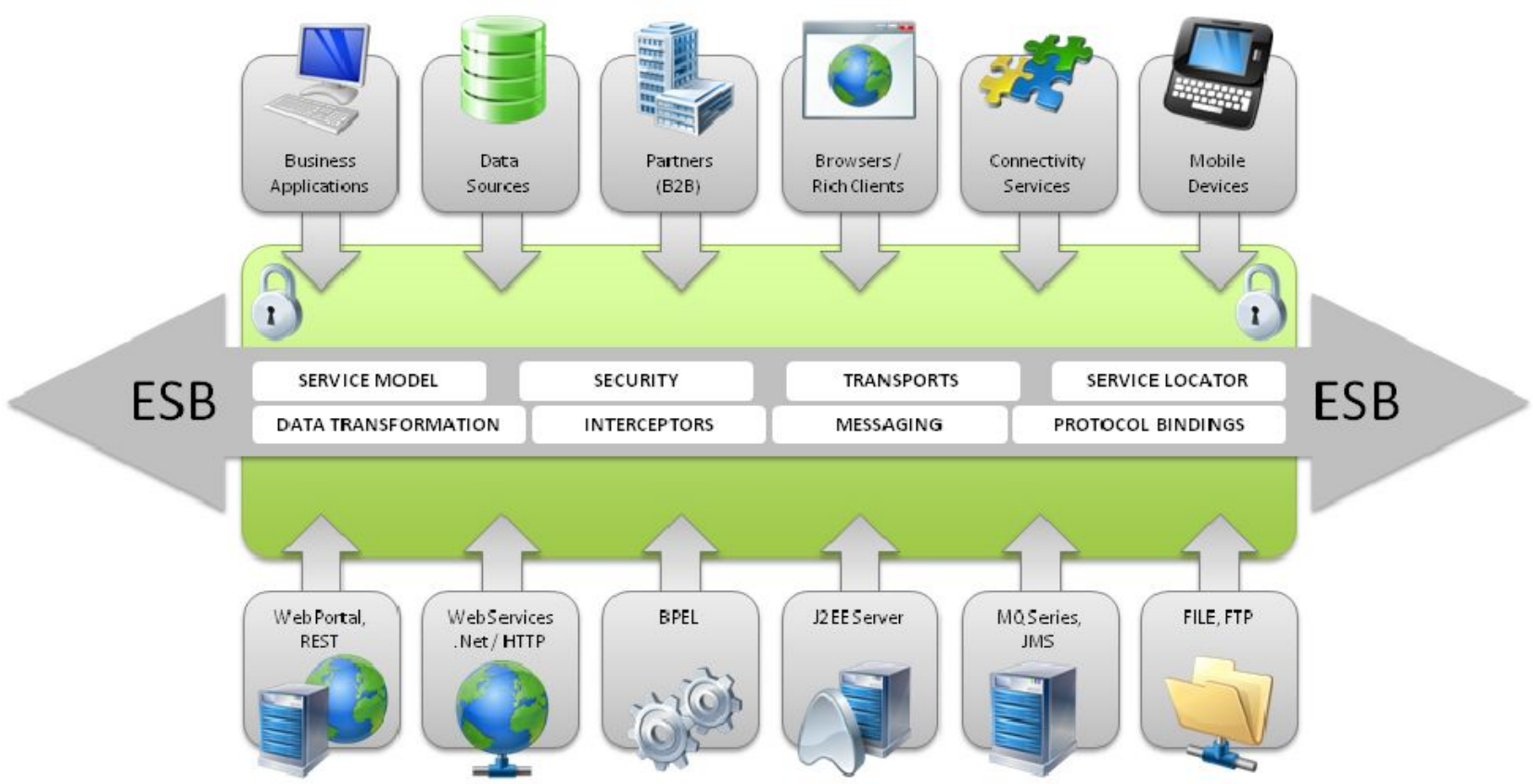
# 3

# Enterprise service bus паттерн

- Асинхронный и синхронный API
- Message Bus
- CQRS
- Оркестрация и хореография
- Версионирование API
- IDL, API design first
- Rich vs Anemic

# Enterprise Service Bus

**Сервисная шина предприятия** – это интеграционная платформа для связи сервисов или приложений друг с другом на основе брокера сообщений



# Основные обязанности ESB

Основные задачи, которые решает ESB:

- Трансляция протоколов
- Обработка, валидация и обогащение информацией сообщений
- Роутинг сообщений
- Безопасность: подписывание сообщений, права доступа сервисов друг к другу и т.д.
- Мониторинг, логирование
- Хореография и оркестрирование сервисов

# 4

# Command/Query паттерны

- Асинхронный и синхронный API
- Message Bus
- CQRS
- Оркестрация и хореография
- Версионирование API
- IDL, API design first
- Rich vs Anemic

# CQS

**Command Query Segregation –** принцип, в соответствии с которым все запросы имеет смысл разделить на

- запросы чтения (query), которые возвращают результат и не имеют побочных эффектов
- на изменения (command), которые ничего не возвращают

Иными словами не должно быть изменений, которые бы возвращали какие-то значения.

Если организовать взаимодействие в виде команд и запросов, то тогда

1) Использовать для запросов – синхронное взаимодействие
2) Использовать для команда – асинхронное взаимодействие

# Async update + sync read

Обычно такой паттерн превращается в async update + sync read.

К сожалению, не всегда все изменения можно преобразовать в команды.

# Async update + sync read

Например, если мы создаем заказ, и дальше планируем с этим заказом что-то делать, то очевидно нам нужно будет получить id этого заказа. Чаще всего такой id возвращается в ответе на создание.

Такой запрос не является командой.

В качестве решения проблемы предлагается использоваться id объектов, сгенерированных на клиенте, т.е. различные виды uuid (guid)

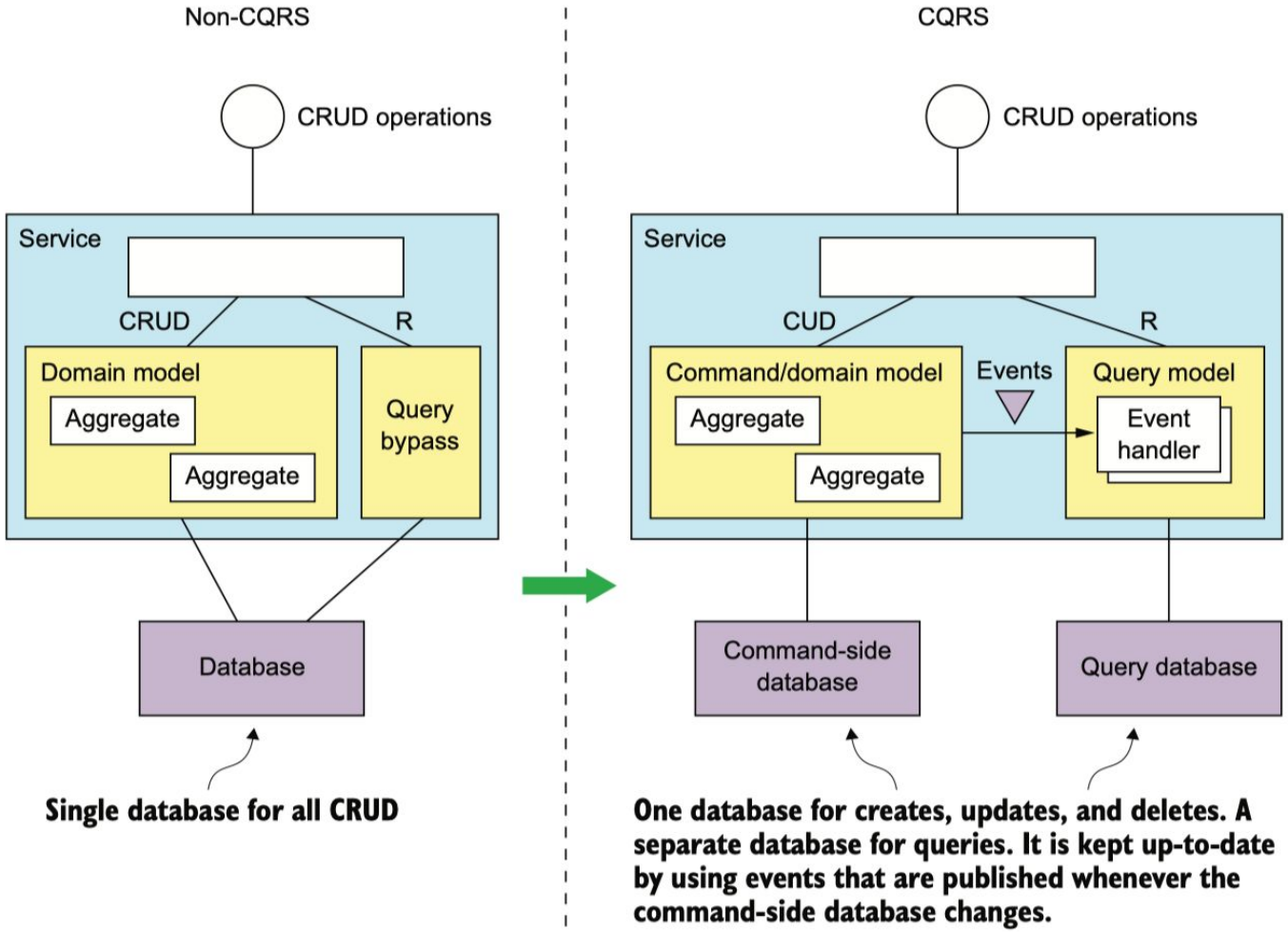
# Async update + sync read

Например, если нам необходимо подтверждение оплаты за заказ, то оплата не является (с формальной точки зрения) командой. Потому что процесс заблокирован до тех пор, пока это подтверждение не придет.

# CQRS

**CQRS** – Command/Query Responsibility Segregation

- Давайте отделим чтение (Read Model/Query) от записи (Write Model/Command)



# Зачем отделять чтения от записи?

Операции чтения (queries) и записи (commands) имеют несколько принципиальных различий в требованиях

**Консистентность**

- **Query**: Большинство систем «ок» с eventual consistency для запросов
- **Command**: Неконсистентные данные для операций записи часто очень плохо

**Хранение данных**

- **Command**: требуется транзакционность хранилища и зачастую удобнее иметь как минимум 3ью нормальную форму
- **Query**: для чтения лучше подходит денормализованная форма

**Масштабируемость**

- **Command**:  в большинстве систем команды производят не очень большую нагрузку (в целом).
- **Query**: в большинстве систем чтение производит намного больше нагрузки

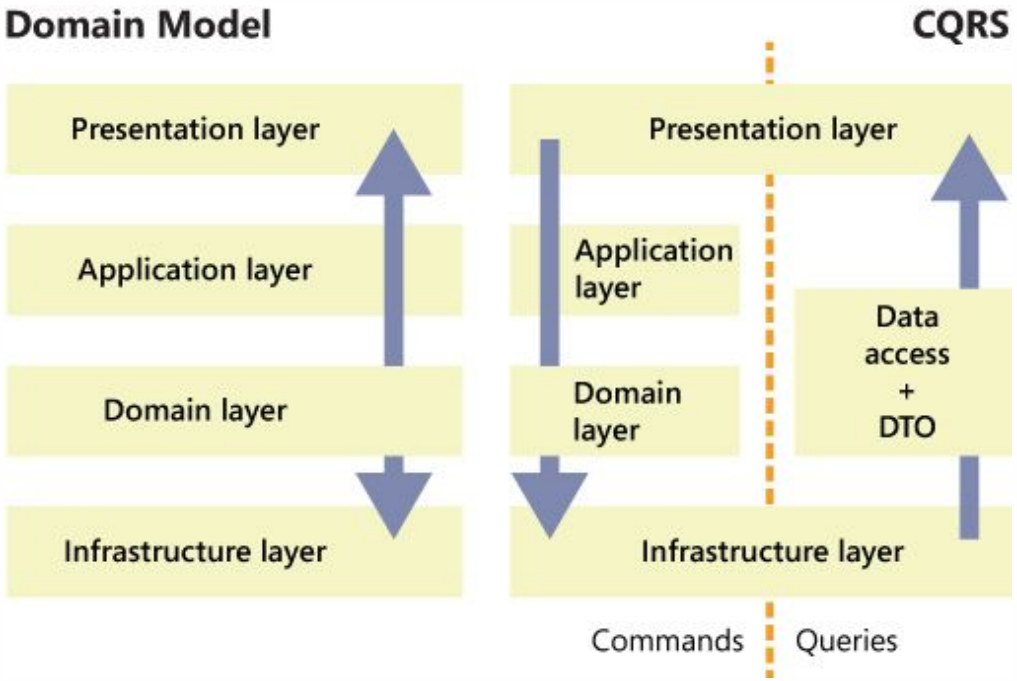
**Крайне тяжело сделать так, чтобы единая модель и единое хранилище удовлетворяло противоречащим требованиям Query и Command**

# CQRS

- **CQRS** является органичным способом дополнением Event Sourcing.

**CQRS** может быть без ES, ES без CQRS тоже может, но в редких случаях.

- CQRS не является архитектурным паттерном для всей системы
- CQRS не обязывает иметь 2 разных БД или NoSQL хранилище.
- CQRS не обязательно приводит к eventual consistency – Read Model может обновляться синхронно.
- Иногда Write модель может читать из Read Model-и
- Write модель может возвращать значения

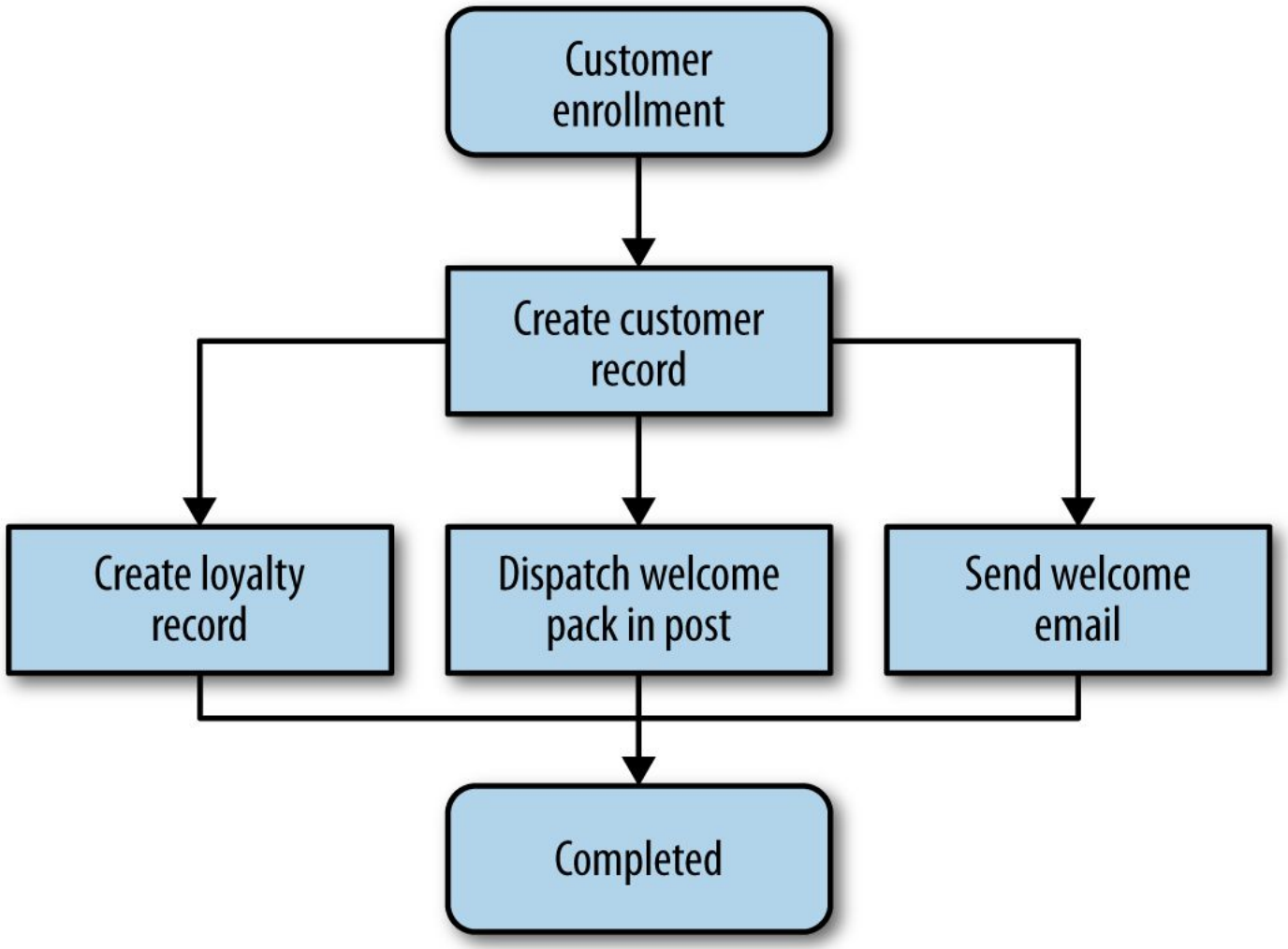


# 05

# Оркестрация и хореография

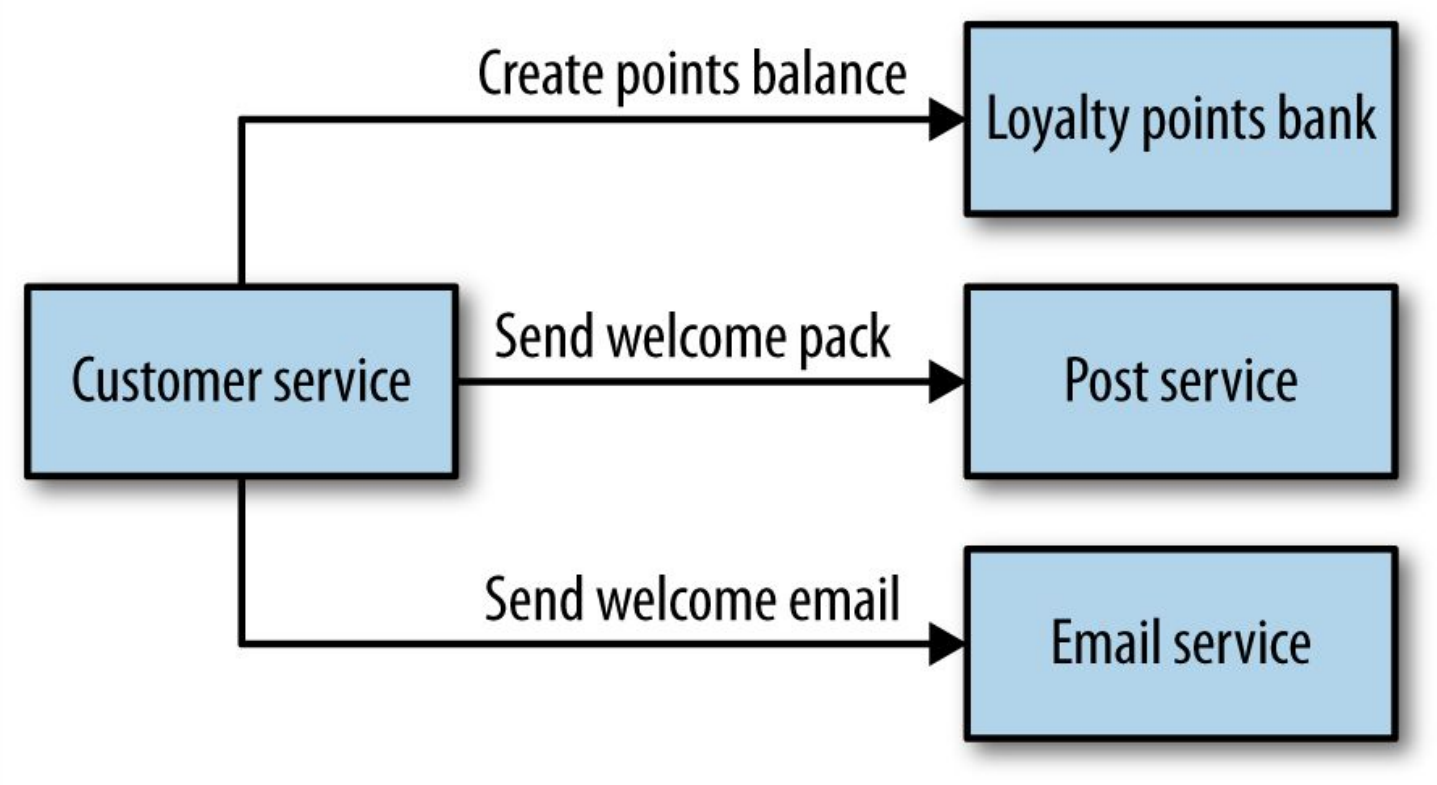
- Асинхронный и синхронный API
- Message Bus
- CQRS
- Оркестрация и хореография
- Версионирование API
- IDL, API design first
- Rich vs Anemic

# Оркестрация и хореография



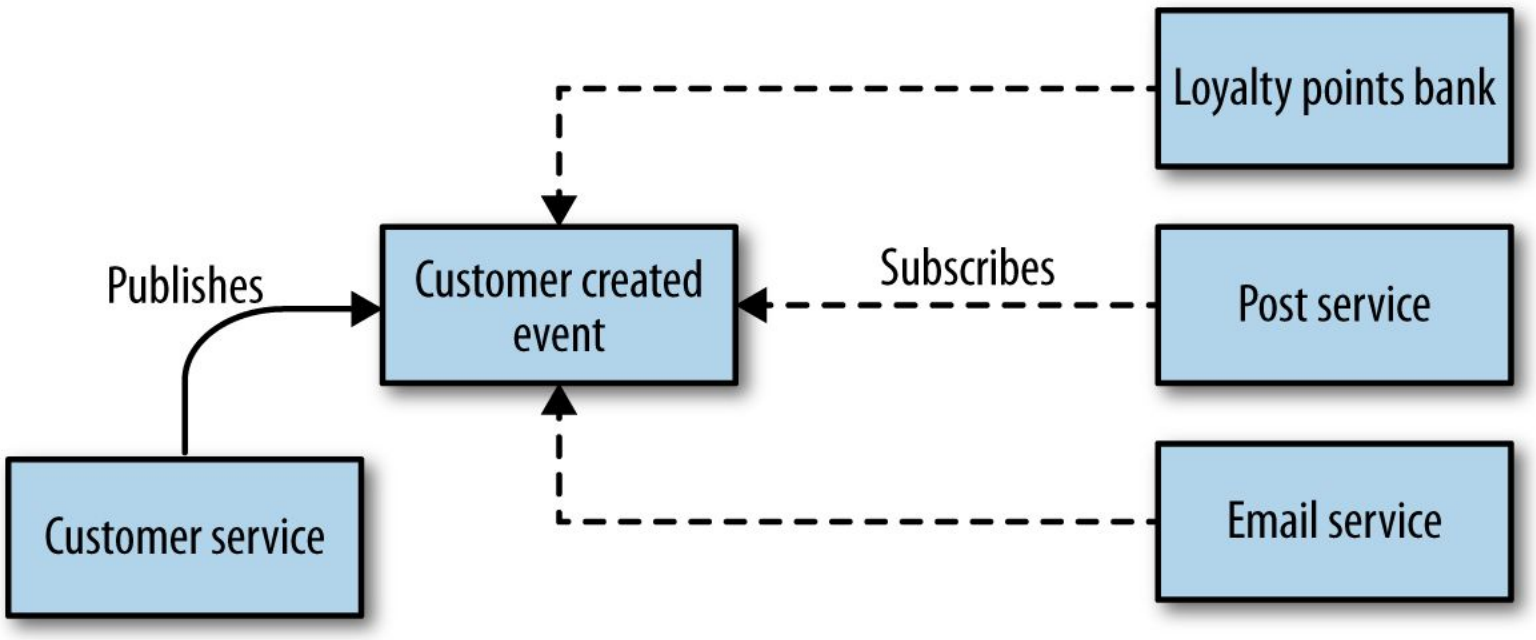
# Оркестрация

**Оркестрация** – один сервис координирует весь бизнес-процесс



# Хореография

**Хореография** – сервисы координируются между собой за счет асинхронного взаимодействия и событийной модели



# Оркестрация vs хореография

- **Оркестрация** дает четкое понимание, что происходит и как выглядит бизнес процесс
- **Оркестрация** может излишне «связать» сервисы друг с другом
- **Хореография** не всегда дает четкое понимание, что и когда происходит.
- **Хореография** «развязывает» сервисы и им нет необходимости знать друг про друга

## Оркестракторы

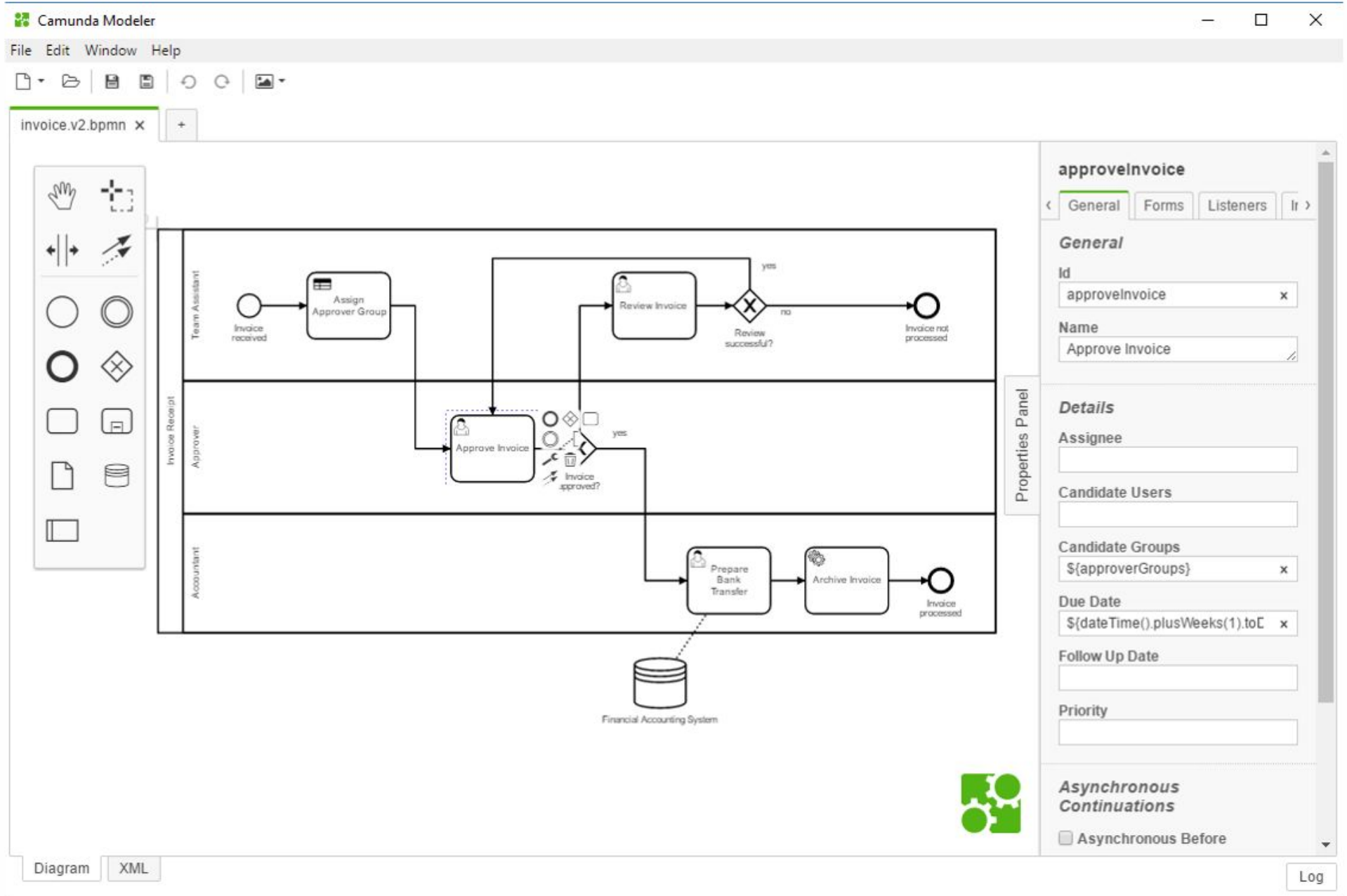
В принципе можно самому написать весь флоу. Но зачастую там требуются вещи, которые характерны для любого бизнес процесса: возможность делать ретраи, аудит-лог действий, асинхронное выполнение задач: например, через 2 дня и т.д.

Поэтому иногда имеет смысл использовать специальные фреймворки или сервисы, созданные как раз для оркестрации

## Camunda

**Camunda** – это BPMN движок, который позволяет визуализировать, создавать модели процессов на BPMN, запускать задачи. Состоит из большого количества различных приложений.

# Camunda


Camunda Modeler
File Edit Window Help
invoice.v2.bpmn
Invoice Receipt
Team Assistant
Approver
Accountant
Invoice received
Assign Approver Group
Review Invoice
Review successful?
yes
no
Invoice not processed
Approve Invoice
Invoice approved?
yes
Prepare Bank Transfer
Archive Invoice
Invoice processed
Financial Accounting System
Properties Panel
approveInvoice
General
Forms
Listeners
Id
approveInvoice
Name
Approve Invoice
Details
Assignee
Candidate Users
Candidate Groups
${approverGroups}
Due Date
${dateTime().plusWeeks(1).toD
Follow Up Date
Priority
Asynchronous Continuations
Asynchronous Before
Diagram
XML
Log

# Cadence

**Candence** – это оркестратор, который позволяет выстраивать и запускать процессы


https://cadenceworkflow.io/

# Cadence

```
public interface SubscriptionWorkflow {
    @WorkflowMethod
    void execute(String customerId);
}

public class SubscriptionWorkflowImpl implements SubscriptionWorkflow {

    private final SubscriptionActivities activities =
        Workflow.newActivityStub(SubscriptionActivities.class);

    @Override
    public void execute(String customerId) {
        activities.sendWelcomeEmail(customerId);
        try {
            boolean trialPeriod = true;
            while (true) {
                Workflow.sleep(Duration.ofDays(30));
                activities.chargeMonthlyFee(customerId);
                if (trialPeriod) {
                    activities.sendEndOfTrialEmail(customerId);
                    trialPeriod = false;
                } else {
                    activities.sendMonthlyChargeEmail(customerId);
                }
            }
        } catch (CancellationException e) {
            activities.processSubscriptionCancellation(customerId);
            activities.sendSorryToSeeYouGoEmail(customerId);
        }
    }
}
```

# 06

# Версионирование API

- Асинхронный и синхронный API
- Message Bus
- CQRS
- Оркестрация и хореография
- Версионирование API
- IDL, API design first
- Rich vs Anemic

# Требования к API

Менять API со сломом обратной совместимости – очень больно.

- API должен быть расширяем
- API достаточно стабилен
- Для rest часто major-версию указывают в пути
- Обычно старые версии поддерживают некоторое время, пока все клиенты не будут обновлены

# Семантическое версионирование API

http://semver.org

**MAJOR.MINOR.PATCH**

- **MAJOR –** изменения без обратной совместимости
- **MINOR** – улучшения API с обратной совместимостью
- **PATCH** – багфикс с обратной совместимостью

# Compatible версионирование API

https://gitlab.com/staltz/comver

**MAJOR.MINOR**

- **MAJOR –** изменения без обратной совместимости
- **MINOR** – улучшения API с обратной совместимостью

# Версионирование API

- Для версионирования публичных пакетов и библиотек semver must have
- Для версионирования публичного API лучше подходит comver или major-version

https://stackoverflow.com/questions/27901549/semantic-versioning-of-rest-apis

# 07

# IDL и API first design

- Асинхронный и синхронный API
- Message Bus
- CQRS
- Оркестрация и хореография
- Версионирование API
- IDL, API design first
- Rich vs Anemic

# IDL

Поскольку в микросервисной архитектуре сервисов стало много, стало много API методов в разных сервисах, то возникла необходимость как-то описывать эти методы и транслировать другим разработчикам внутри компании.

Когда приложение было модульное интерфейс описывался на языке программирования.

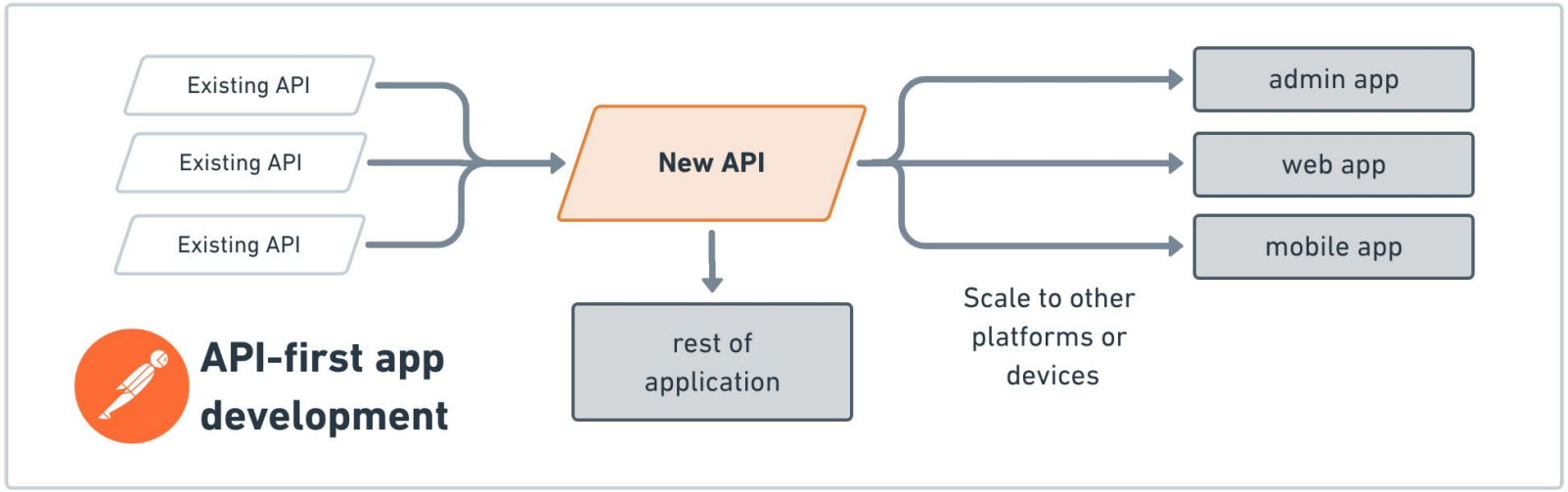
Для описания интерфейсов взаимодействия сервисов друг с другом чаще всего используются специализированные языки **IDL - Interface Definition Language.**

Например, OpenAPI для RESTful, AsyncAPI для описания событий, GrapqlScheme – для GraphQL и т.д.

# API First design

**API First design** – проектирование сервисов (или приложений) начинается с проектирования API.

- Совместно проектируется API и пишется на IDL спецификация
- Пишутся тесты и/или моки
- Начинается разработка сервиса.



# API First design

**Как неправильно?**

- Бекендеры сначала написали сервис, придумали API и отдали его фронтендерам, чтобы они запилили интерфейс

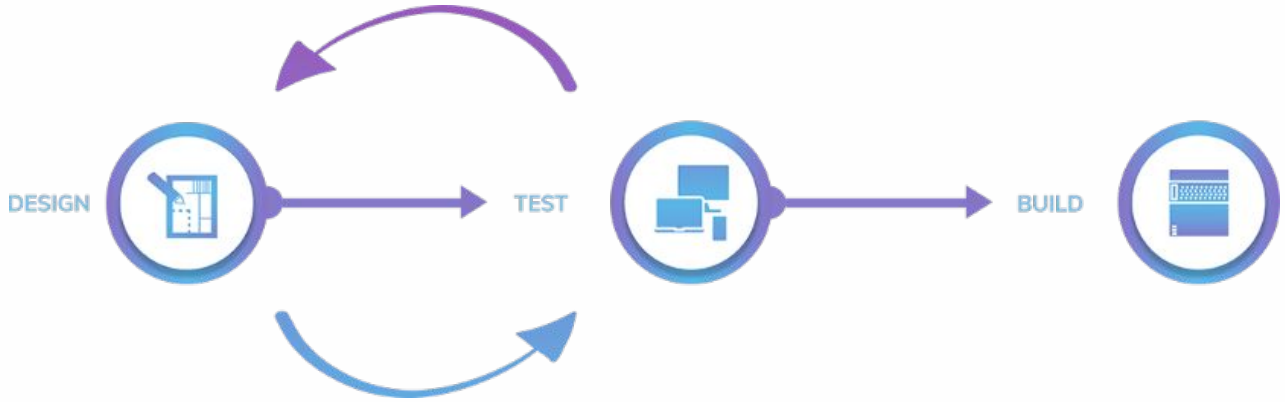
**Как правильно?**

- Бекендеры вместе с фронтендерами придумали API и описали его спецификацию.
- Написали моки для API, по которым фронтендеры могут начать разработку

# API First design

## Плюсы API First design

- Параллельная разработка
- Значительно меньше недоговоренностей
- Меньше багов
- Лучшая архитектура проекта

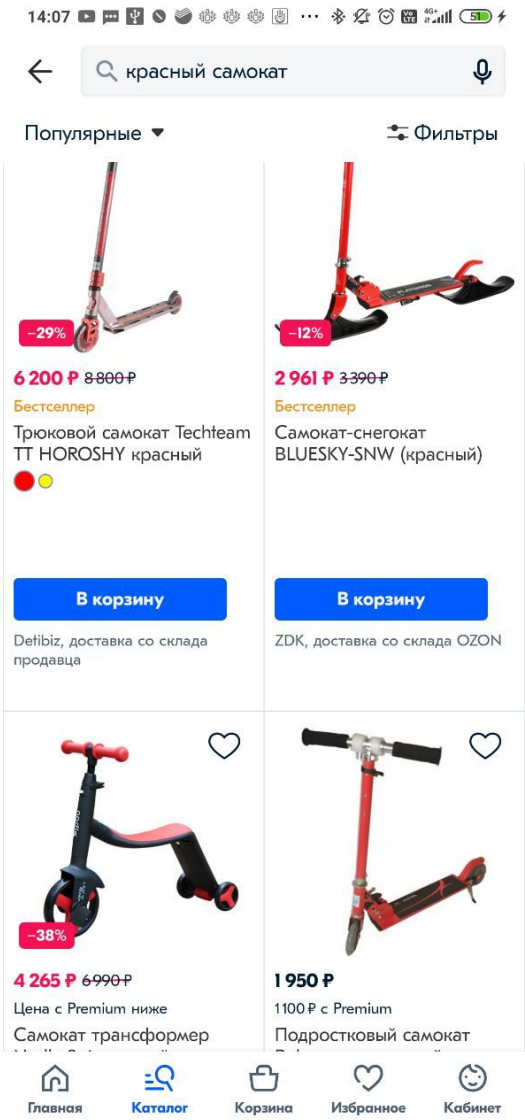


# С чего начинать проектирование API?

- Стоит отталкиваться от предметной области.
- Начинать с основного пользовательского сценария.
- Каждый метод API желательно связывать с "функцией" в предметной области
- На первых итерациях лучше не думать про реализацию на бекенде.

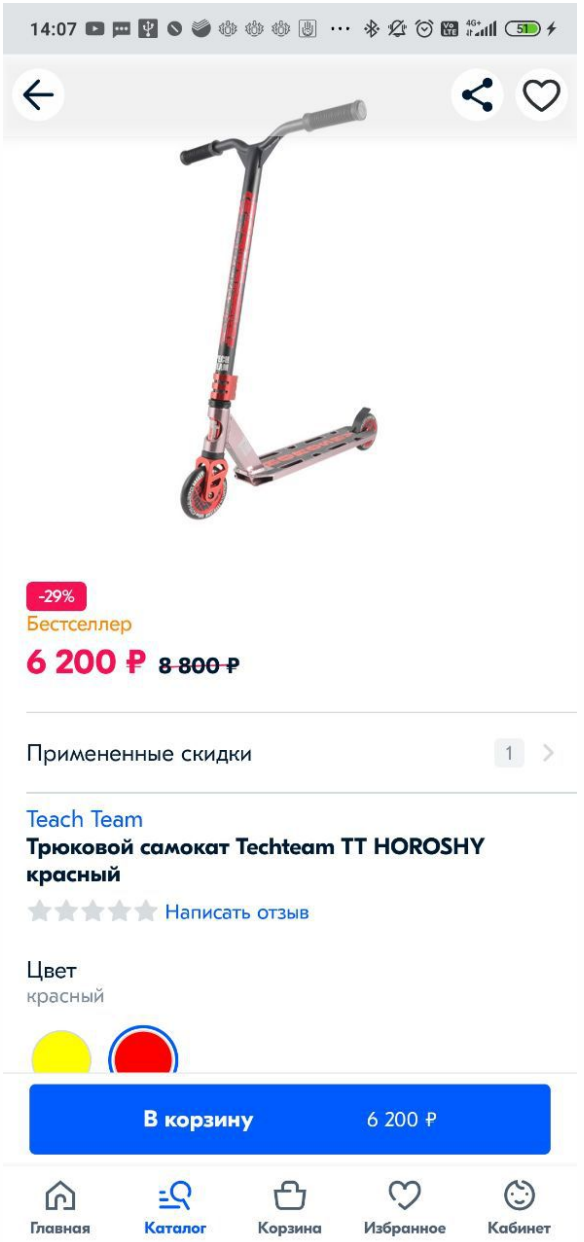
# Пользовательский сценарий

**Есть** авторизованный клиент
**И есть** интернет-магазин "Гозон"
**И** клиент вводит в строке поиска "Красный самокат"
**Тогда** появляется список товаров, подходящих под запрос клиента
**И** в карточке товара пользователь видит
1) название
2) цену
3) изображение
4) скидку
5) доступность в разных магазинах



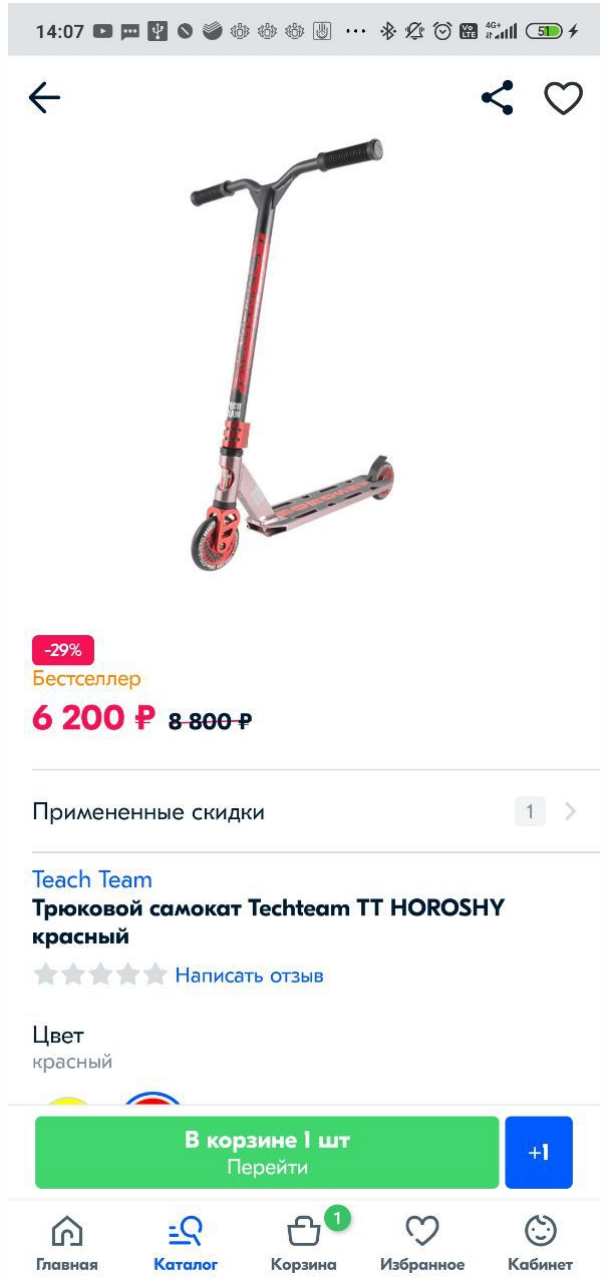
# Пользовательский сценарий

**Когда** клиент нажимает карточку на товар
**Тогда** клиент переходит на страничку с описанием товара



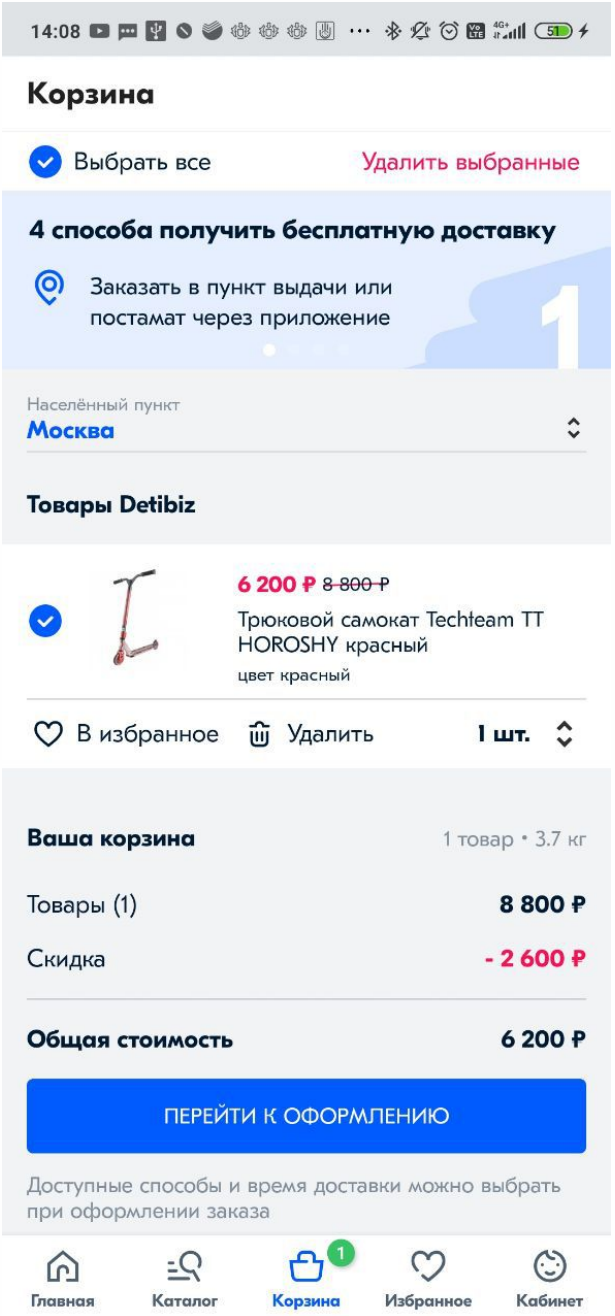
# Пользовательский сценарий

**Когда** клиент нажимает на кнопку "Добавить товар в корзину"
**Тогда** товар добавляется в корзину



# Пользовательский сценарий

**Когда** клиент переходит в корзину
**Тогда** он видит список всех добавленных в корзину товаров



# API first design

**И** клиент вводит в строке поиска "Красный самокат"
**Тогда** появляется список товаров, подходящих под запрос клиента

**GET** /api/v1/search/product_search/?q=Красный самокат&limit=20

**И** в карточке товара пользователь видит
1) название
2) цену
3) изображение
4) скидку
5) доступность в разных магазинах


```
[
  {
    "name": "Трюковый самокат",
    "price": "6200",
    "image_url": "http://gozon.ru/images/a123x-234ax-...jpg",
    "old_price": "8800",
    "availability": [
      { "shop_id": 15, "name": "Кузьминки", "count": "MANY" },
      ...
```

# API first design

**Когда** клиент нажимает карточку на товар
**Тогда** клиент переходит на страничку с описанием товара

**GET** /api/v1/products/{id}/

```
{
   "id": 42,
   "name": "Трюковый самокат",
   "price": "6200",
   "image_url": "http://gozon.ru/images/a123x-234ax-...jpg",
   "old_price": "8800",
   "availability": [
         { "shop_id": 15, "name": "Кузьминки", "count": "MANY" },
         ...
    ],
    "colours": ["RED", "YELLOW"],
    "rating": "0",
    "reviews": [ ... ]
    ...
}
```

# API first design

**Когда** клиент нажимает на кнопку "Добавить товар в корзину"
**Тогда** товар добавляется в корзину

**POST** /api/v1/cart/products/

```
{
   "product_id": 42,
   "quantity": 1
}
```

# API first design

**Когда** клиент переходит в корзину
**Тогда** он видит список всех добавленных в корзину товаров

**GET** /api/v1/cart/products/

```
[
   {
      "product_id": 42,
      "quantity": 1
   },
   {
      "product_id": 27,
      "quantity" 4
   }
   ...
]
```

**GET** /api/v1/products?ids=42,27
[ ... ]

# 08

# Rich vs Datacentric API

- Асинхронный и синхронный API
- Message Bus
- CQRS
- Оркестрация и хореография
- Версионирование API
- IDL, API design first
- Rich vs Anemic

# Datacentric API vs rich API

Для разработки не так важен конкретный протокол, сколько семантика API вызовов. Самый частый антипаттерн – это использование анемичного или дата-центричного API в сервисах со сложной предметной областью.

https://martinfowler.com/bliki/AnemicDomainModel.html

# Сервис JARA

Проектируется сервис JARA – таск-трекер, похожий, на JIRA. Тикет может быть разных типов. Каждому типу соответствует разный набор полей и немного разная валидация. Также у тикета есть статусная модель — конечный автомат по переходу из одного статуса в другой.

Пусть API будет выглядеть так:

методы POST/PATCH/GET, по урлу /api/v1/tickets/{ticket_id}.json которые обновляют создают и обновляют данные.

# Сервис JARA

Получить данные о тикете:
GET /api/v1/tickets/42.json

```
{
  "id": 42,
  "status": "IN_PROGRESS",
   "description": "Lorem ipsum",
   "type": "feature"
}
```

Обновить данные о тикете:
PATCH /api/v1/tickets/42.json

```
{
   "status": "closed",
   "description": "Ipsum Lorem"
}
```

# Сервис JARA

Но что произойдет, если попытаться обновить у тикета два связанных между собой поля?

PATCH /api/v1/tickets/42.json

```
{
    "type": "bug",
    "status": "closed",
    "description": "на самом деле фича"
}
```

т.е одновременно меняется тип тикета и статус. Но тип тикета и статуса связаны между собой. Например,  для типа Bug следующий статус по workflow “QA Approval”, без прохождения которого закрыть тикет нельзя?

# Datacentric API vs rich API

Если изменение какого-то поля в рамках CRUD-методов API — это не просто изменение данных, а операция, завязанная на согласованное изменение состояния сущности, то эту операцию нужно выносить в отдельный метод и не давать напрямую менять данные

# Решение 1

Вместо универсального
ресурса /api/v1/tickets.json добавить еще ресурсы-
глаголы:

POST /api/v1/tickets/{ticket_id}/migrate.json —
смигрировать из одного типа в другой

POST /api/v1/tickets/{ticket_id}/set-status.json — если есть
статусная модель

# Решение 2

Вместо универсального
ресурса /api/v1/tickets.json добавить еще ресурсы
сущности:

POST /api/v1/tickets/migrations.json

POST /api/v1/subscriptions/trial.json  - создать триальную
подписку

POST /api/v1/money_transfers.json – перевести деньги

И т.д.

# Datacentric API vs rich API

Проблема касается не только REST-like, но и RPC итерфейсы

editTicket() или editCampaign() ничем не лучше

**API и интерфейсы должны отражать предметную область и опираться на пользовательские сценарии, а не быть просто HTTP прослойкой к БД.**

# Цели занятия - рефлексия

- Асинхронный и синхронный API
- Message Bus
- CQRS
- Оркестрация и хореография
- Версионирование API
- IDL, API design first
- Rich vs Anemic

# Спасибо
# за внимание!

# Опрос

https://otus.ru/polls/40822/

# Домашнее задание - через 3 занятия, но можно начинать

Реализовать сервис заказа. Сервис биллинга. Сервис нотификаций.

- При создании пользователя, необходимо создавать аккаунт в сервисе биллинга. В сервисе биллинга должна быть возможность положить деньги на аккаунт и снять деньги.
- Сервис нотификаций позволяет отправить сообщение на email. И позволяет получить список сообщений по методу API.

- Пользователь может создать заказ. У заказа есть  параметр - цена заказа.
- Заказ происходит в 2 этапа:

    1) сначала снимаем деньги с пользователя с помощью сервиса биллинга

    2) отсылаем пользователю сообщение на почту с результатами оформления заказа. Если биллинг подтвердил платеж, должно отослаться письмо счастья. Если нет, то письмо горя.

**Подробнее - см на сайте**

# Домашнее задание - через 3 занятия, но можно начинать

Спроектировать взаимодействие сервисов при создании заказов. Предоставить варианты взаимодействий в следующих стилях в виде sequence диаграммы с описанием API на IDL:

- только HTTP взаимодействие
- событийное взаимодействие с использование брокера сообщений для нотификаций (уведомлений)
- Event Collaboration стиль взаимодействия с использованием брокера сообщений
- вариант, который вам кажется наиболее адекватным для решения данной задачи. Если он совпадает одним из вариантов выше - просто отметить это.

Выбрать один из вариантов и реализовать его.

На выходе должны быть

0) описание архитектурного решения и схема взаимодействия сервисов (в виде картинки)

1) команда установки приложения (из helm-а или из манифестов). Обязательно указать в каком namespace нужно устанавливать.

2) тесты постмана, которые прогоняют сценарий: