

# Микросервисная архитектура



# Содержание

1. Сервисориентированная архитектура
2. Микросервисная архитектура
3. Kubernetes
4. Cloud Native Foundation



# Базовые принципы SOA

- **Интегрируемость приложений**, ориентированных на пользователей
- **Многократное использование сервисов**
- **Независимость** от набора технологий
- **Автономность** (независимые эволюция, масштабируемость и развёртываемость)



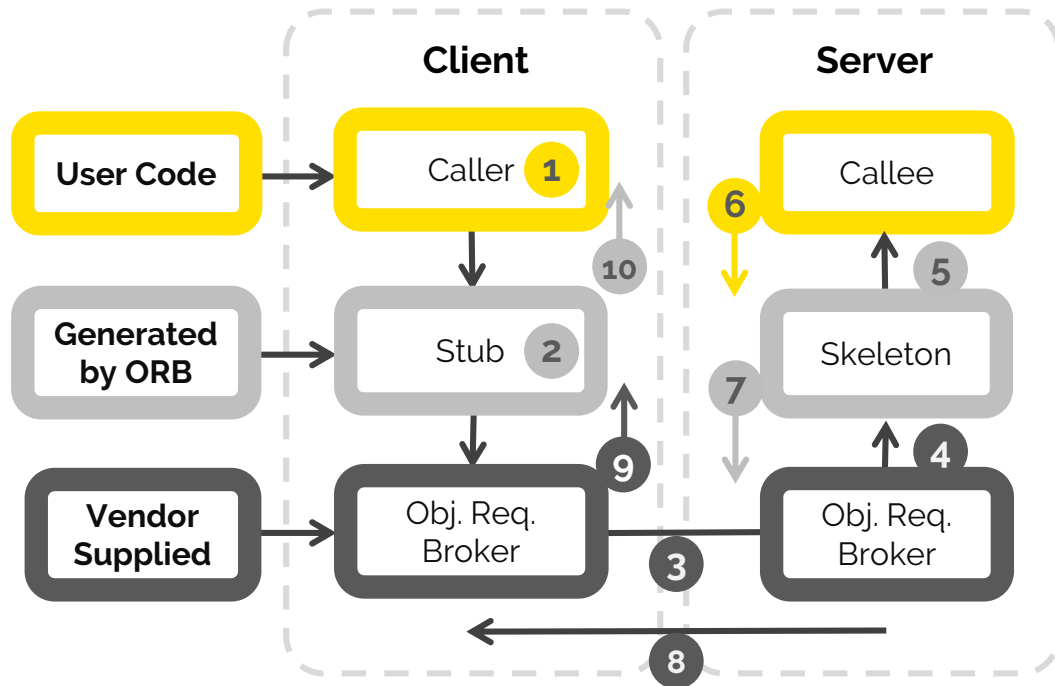
# CORBA Common Object Request

Стандарт создавался в **1980-х** объединением различных вендоров для распределенных клиент серверных вычислений, максимальный расцвет технологии получил к 1991 году.

## Основные принципы:

- Не зависящие от платформы вызовы удалённых процедур (Remote Procedure Call)
- Транзакции (в том числе удалённые!)
- Безопасность
- События
- Независимость от выбора языка программирования
- Независимость от выбора ОС
- Независимость от выбора оборудования
- Независимость от особенностей передачи данных/связи
- Набор данных через язык описания интерфейсов (Interface Definition Language, IDL)

# Структурная схема взаимодействия компонентов



1. Заглушка проверяет вызов, создаёт сообщение-запрос и передаёт его в ORB.
2. Клиентский ORB шлёт сообщение по сети на сервер и блокирует текущий поток выполнения.
3. Серверный ORB получает сообщение-запрос и создаёт экземпляр скелета.
4. Скелет исполняет процедуру в вызываемом объекте.
5. Вызываемый объект проводит вычисления и возвращает результат.
6. Скелет пакует выходные аргументы в сообщение-ответ и передаёт его в ORB.
7. ORB шлёт сообщение по сети клиенту.
8. Клиентский ORB получает сообщение, распаковывает и передаёт информацию заглушке.
9. Заглушка передаёт выходные аргументы вызываемому методу, разблокирует поток выполнения, и вызывающая программа продолжает свою работу.



# Достоинства

- Независимость от выбранных технологий (не считая реализации ORB).
- Независимость от особенностей передачи данных/связи.



# Недостатки

- **Независимость от местоположения:** клиентский код не имеет понятия, является ли вызов локальным или удалённым. Звучит неплохо, но длительность задержки и виды сбоев могут сильно варьироваться. Если мы не знаем, какой у нас вызов, то приложение не может выбрать подходящую стратегию обработки вызовов методов, а значит, и генерировать удалённые вызовы внутри цикла. В результате вся система работает медленнее.
- **Сложная, раздутая и неоднозначная спецификация:** её собрали из нескольких версий спецификаций разных вендоров, поэтому (на тот момент) она была раздутой, неоднозначной и трудной в реализации.
- **Заблокированные каналы связи (communication pipes):** используются специфические протоколы поверх TCP/IP, а также специфические порты (или даже случайные порты). Но правила корпоративной безопасности и фаерволы зачастую допускают HTTP-соединения только через 80-й порт, блокируя обмены данными CORBA.

# Web сервисы как замена Corba

- Простой протокол для обмена сообщениями: HTTP через порт 80.
- Платформенно-независимый язык XML или JSON.
- Упрощённая спецификация взаимодействия:
  1. Первый черновик SOAP появился в 1998-м, стал рекомендацией W3C в 2003-м, после чего превратился в стандарт. SOAP вобрал в себя некоторые идеи CORBA, вроде слоя для обработки обмена сообщениями и «документа», определяющего интерфейс с помощью языка описания веб-сервисов (Web Services Description Language, WSDL).
  2. Рой Филдинг в 2000-м описал REST в своей диссертации «Architectural Styles and the Design of Network-based Software Architectures». Его спецификация оказалась гораздо проще SOAP, поэтому вскоре REST обогнал SOAP по популярности.
  3. Facebook разработал GraphQL в 2012-м, а публичный релиз выпустил в 2015-м. Это язык запросов для API, позволяющий клиенту строго определять, какие данные сервер должен ему отправить, не больше и не меньше.



# Достоинства и недостатки SOAP

## Достоинства

1. Независимость набора технологий, развёртывания и масштабируемости сервисов.
2. Стандартный, простой и надёжный канал связи (передача текста по HTTP через порт 80).
3. Оптимизированный обмен сообщениями.
4. Стабильная спецификация обмена сообщениями.
5. Изолированность контекстов доменов (Domain contexts).

# Достоинства и недостатки SOAP

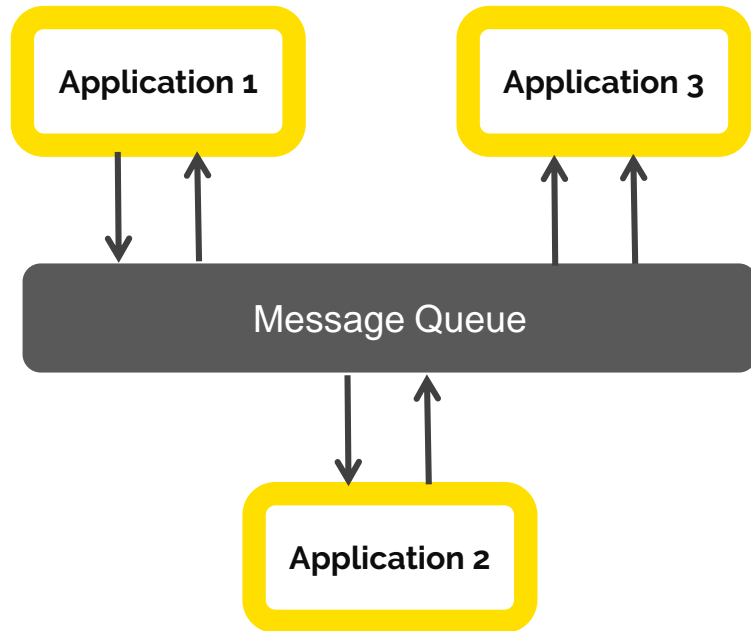
## Недостатки

1. Разные веб-сервисы тяжело интегрировать из-за различий в языках передачи сообщений.  
Например, два веб-сервиса, использующих разные JSON-представления одной и той же концепции.
2. Синхронный обмен сообщениями может перегрузить системы.

# Очередь сообщений

- **Запрос/Ответ**

Клиент шлёт в очередь сообщение, включая ссылку на «разговор» («conversation» reference). Сообщение приходит на специальный узел, который отвечает отправителю другим сообщением, где содержится ссылка на тот же разговор, так что получатель знает, на какой разговор ссылается сообщение, и может продолжать действовать. Это очень полезно для бизнес-процессов средней и большой продолжительности (цепочек событий, sagas).





# Очередь сообщений

- **Публикация/Подписка**  
**По спискам**

Очередь поддерживает списки опубликованных тем подписок (topics) и их подписчиков. Когда очередь получает сообщение для какой-то темы, то помещает его в соответствующий список. Сообщение сопоставляется с темой по типу сообщения или по заранее определённым набору критериев, включая и содержимое сообщения.

## **На основе вещания**


Когда очередь получает сообщение, она транслирует его всем узлам, прослушивающим очередь. Узлы должны сами фильтровать данные и обрабатывать только интересующие сообщения.

- **В pull-сценарии** клиент опрашивает очередь с определённой частотой. Клиент управляет своей нагрузкой, но при этом может возникнуть задержка: сообщение уже лежит в очереди, а клиент его ещё не обрабатывает, потому что не пришло время следующего опроса очереди.
- **В push-сценарии** очередь сразу же отдаёт клиентам сообщения по мере поступления. Задержки нет, но клиенты не управляют своей нагрузкой.

# Достоинства и недостатки очередей сообщений

## Достоинства

- Независимость набора технологий, развёртывания и масштабируемости сервисов.
- Стандартный, простой и надёжный канал связи (передача текста по HTTP через порт 80).
- Оптимизированный обмен сообщениями.
- Стабильная спецификация обмена сообщениями.
- Изолированность контекстов домена (Domain contexts).
- Простота подключения и отключения сервисов.
- Асинхронность обмена сообщениями помогает управлять нагрузкой на систему.

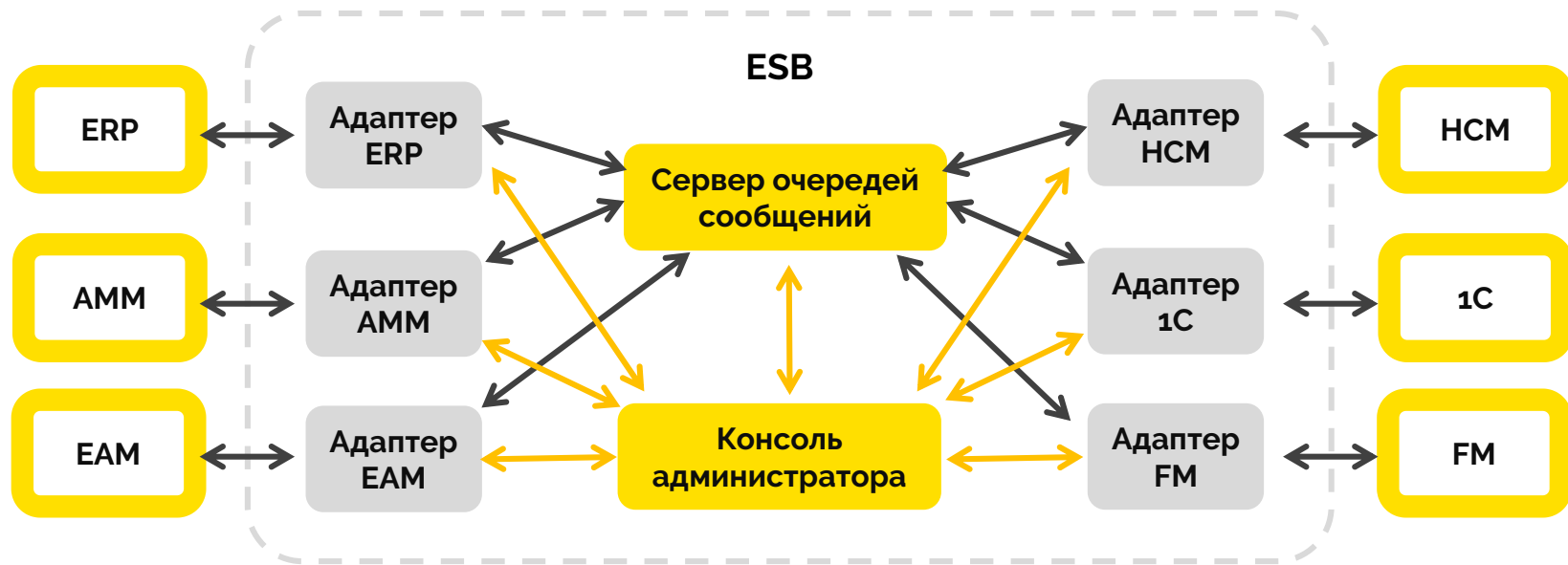



# Достоинства и недостатки очередей сообщений

## Недостатки

Разные веб-сервисы тяжело интегрировать из-за различий в языках передачи сообщений. Например, два веб-сервиса, использующих разные JSON-представления одной и той же концепции.

## Сервисная шина ESB (развитие очередей и SOA)





# Сервисная шина ESB (развитие очередей и SOA)

## Главные обязанности ESB:

- Отслеживать и маршрутизировать обмен сообщениями между сервисами
- Преобразовывать сообщения между общающимися сервисными компонентами
- Управлять развёртыванием и версионированием сервисов
- Управлять использованием избыточных сервисов
- Предоставлять стандартные сервисы обработки событий, преобразования и сопоставления данных, сервисы очередей сообщений и событий, сервисы обеспечения безопасности или обработки исключений, сервисы преобразования протоколов и обеспечения необходимого качества связи



# Достоинства и недостатки ESB

## Достоинства

- **Независимость набора технологий**, развёртывания и масштабируемости сервисов.
- **Стандартный, простой и надёжный канал связи** (передача текста по HTTP через порт 80).
- **Оптимизированный** обмен сообщениями
- **Стабильная** спецификация обмена сообщениями
- **Изолированность** контекстов домена (Domain contexts)
- **Простота** подключения и отключения сервисов
- **Асинхронность обмена** сообщениями помогает управлять нагрузкой на систему
- **Единая точка для управления** версионированием и преобразованием.

# Достоинства и недостатки ESB

## Недостатки

- Ниже скорость связи, особенно между уже совместимыми сервисами
- **Централизованная логика:**  
Единая точка отказа, способная обрушить системы связи всей компании
- Большая сложность конфигурирования и поддержки
- Со временем можно прийти к хранению в ESB бизнес-правил
- Шина так сложна, что для её управления вам потребуется целая команда
- Высокая зависимость сервисов от ESB.



# Микросервисы


В основе микросервисной архитектуры лежат **концепции SOA**.

**ESB** была создана в контексте интеграции отдельных приложений, чтобы получилось единое корпоративное распределённое приложение.

Микросервисная архитектура создавалась в контексте быстро и постоянно меняющихся бизнесов, которые (в основном) с нуля создают собственные облачные приложения.



# Определение



**Микросервисы** — это маленькие автономные сервисы, работающие вместе и спроектированные вокруг бизнес-домена.

**Sam Newman 2015, Principles Of Microservices**

# Принципы микро сервисной архитектуры по Ньюману

**Сэм Ньюман**, автор Building Microservices, выделяет восемь принципов микросервисной архитектуры

## 1. Проектирование сервисов вокруг бизнес-доменов

Это может дать нам стабильные интерфейсы, высокосвязные и мало зависящие друг от друга модули кода, а также чётко определённые разграниченные контексты.

## 2. Культура автоматизации

Это даст нам гораздо больше свободы, мы сможем развернуть больше модулей.

## 3. Скрытие подробностей реализации

Это позволяет сервисам развиваться независимо друг от друга.

## 4. Полная децентрализация

Децентрализуйте принятие решений и архитектурные концепции, предоставьте командам автономность, чтобы компания сама превратилась в сложную адаптивную систему, способную быстро приспосабливаться к переменам.

# Принципы микросервисной архитектуры по Ньюману

## 5. Независимое развёртывание

Можно развёртывать новую версию сервиса, не меняя ничего другого.

## 6. Сначала потребитель

Сервис должен быть простым в использовании, в том числе другими сервисами

## 7. Изолирование сбоев

Если один сервис падает, другие продолжают работать, это делает всю систему устойчивой к сбоям.

## 8. Удобство мониторинга

В системе много компонентов, поэтому трудно уследить за всем, что в ней происходит. Нам нужны сложные инструменты мониторинга, позволяющие заглянуть в каждый уголок системы и отследить любую цепочку событий.

# Запомним эти принципы!

1. Проектирование сервисов вокруг бизнес-доменов
2. Культура автоматизации
3. Скрытие подробностей реализации
4. Полная децентрализация
5. Независимое развёртывание
6. Сначала потребитель
7. Изолирование сбоев
8. Удобство мониторинга



# Монолитная архитектура

- Общая кодовая база для всего проекта
- Все бизнес-процессы в одном месте
- Удобное переиспользование кода
- Довольно просто отлаживать
- Довольно удобно разрабатывать
- Один язык, один стек технологий, один набор библиотек
- Как правило, одна БД на весь проект
- Целостный подход к безопасности

**Монолитные приложения могут быть и распределенными**





# Проблемы монолита

- Связность между компонентами постоянно растет и это затрудняет внесение изменений
- Для уменьшения связности приходится проводить рефакторинги
- В большом проекте (>30 разработчиков, >1M LoC) постоянно что-то не работает и чтобы выпустить релиз приходится принимать специальные меры: code freeze или gitflow
- Очень протяженный периметр безопасности
- Проект ограничен одним стеком




# Микросервисы

Проект разделён на набор микросервисов с изолированной кодовой базой

## У каждого микросервиса

- собственная команда разработки и собственный цикл релизов
  - простой и обозримый внешний интерфейс
  - Небольшой круг обязанностей
- Небольшой периметр безопасности
  - Собственная БД (как правило)



# Мотивация для использования микросервисов

1. Сохранять компоненты системы простыми
2. Автономная работа нескольких команд
3. Быстрый цикл разработки
4. Использование нескольких технологических стеков одновременно
5. Разделение сервисов по характеристикам (предметной области, надежности, масштабируемости, безопасности)

# Как разделить продукт на микросервисы

1. Какого размера должны быть микросервисы?
2. Как определить в какой микросервис должен пойти функционал?
3. Как микросервисы будут взаимодействовать между собой?
4. Что делать если необходимы транзакционные изменения между несколькими микросервисами?



# Размер микросервиса

От команды размером в “2 пиццы” до одна фича, **один микросервис** 😊

1. Управляемость команды (обычно 7-10 человек)
2. Взаимозаменяемость разработчиков (т. е. как минимум трое должны быть способны поддерживать микросервис)
3. Оптимальный размер сервиса 150 - 500 LoC

# Обязанности микросервиса

**High cohesion inside, loose coupling outside** (код микросервиса переиспользует одни и те же сервисы, модели и структуры данных, а наружу выставляет минимальный интерфейс)

**Функционал нужно добавлять в тот микросервис, где:**

1. Не потребуется изменять API этого микросервиса, либо потребуется минимальное изменение
2. Удастся переиспользовать существующие алгоритмы и структуры данных
3. Нефункциональные требования совпадают (безопасность, масштабирование)



# Межсервисное взаимодействие

- HTTP (REST)
- Очереди (AMQP, Kafka)
- gRPC

# REST (REpresentational State Transfer)

**Абстракция поверх протокола HTTP** позволяющая переиспользовать весь HTTP стек

Чтобы разработать **REST API**, нужно разложить требуемые бизнес-операции на набор манипуляций и ресурсов

**REST строится** вокруг ресурса определяемого своим URI и фиксированного набора операций над этим ресурсом определяемых известных как **HTTP методы** (GET, POST, PUT, DELETE, PATCH)

Формат данных обычно **JSON**, но это не обязательно

**REST in Practice:** Hypermedia and Systems Architecture (2010)



# Семантика HTTP

**GET** /order/123 – получить заказ №123

**GET** /orders – получить список всех заказов

**POST** /order/ – создать новый заказ

**PUT** /order/123 – обновить заказ №123

**DELETE** /order/123 – удалить заказ №123

**PATCH** /order/123 – частично обновить заказ №123

# Идемпотентность

**Идемпотентность** — свойство объекта или операции при повторном применении операции к объекту давать тот же результат, что и при первом.

**POST** /order/ – создать новый заказ

Неидемпотентная операция, при повторном вызове будет создан новый заказ

**PUT** /order/123 – обновить заказ №123.

Идемпотентная операция, при повторном вызове будут применены те же изменения к тому же заказу и состояние системы не изменится

# Вызовы процедур через REST

Допустим, нам нужно вызвать какую-то процедуру через REST API

POST /generateAndSendReport – **плохо**

Можно создавать “виртуальные ресурсы”:

POST /reports/ – **хорошо**

```
{ "reportId": 123, "status": "generating" }
```

```
GET /reports/123
```

```
{ "reportId": 123, "status": "complete" }
```

# Что нужно помнить про REST

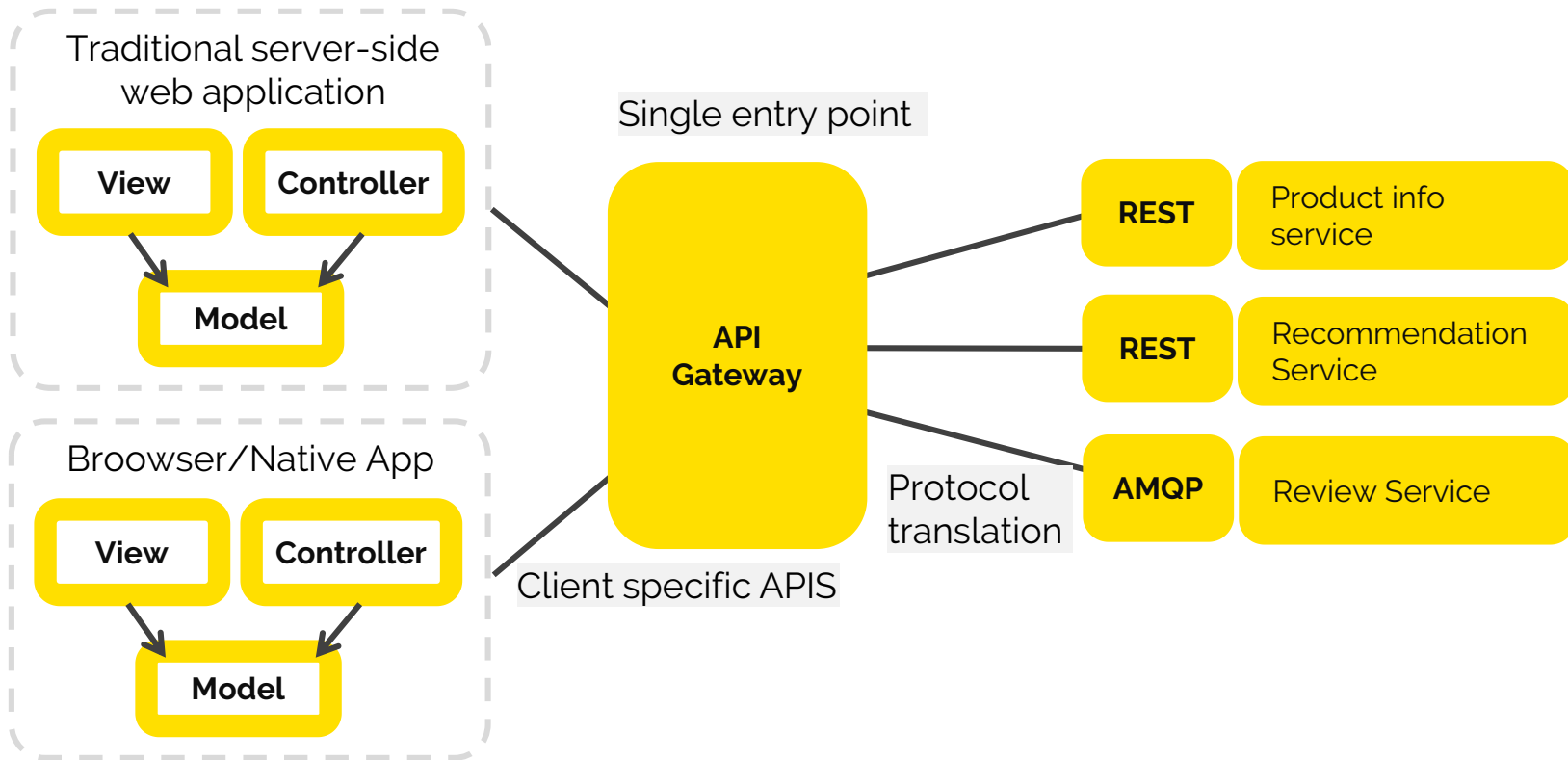
- **Вызовы REST синхронны.**  
Вызывающий код будет ждать ответа.
- Формат ресурса **не должен** зависеть от метода
- Если формат все-же зависит, то скорее всего, это **другой ресурс**
- REST удобно документировать через **OpenAPI (former Swagger)**  
<https://www.openapis.org/>
- **API нужно версионировать**  
GET /api/v1/orders

# API Gateway

**Мотивация** – запросы от внешних клиентов должны попасть конкретным микросервисам.

**Решение** — сделать еще один сервис, который будет принимать внешние запросы и перенаправлять их микросервисам. Заодно он же может заниматься аутентификацией пользователей и трансляцией **REST – AMQP**.

# API Gateway

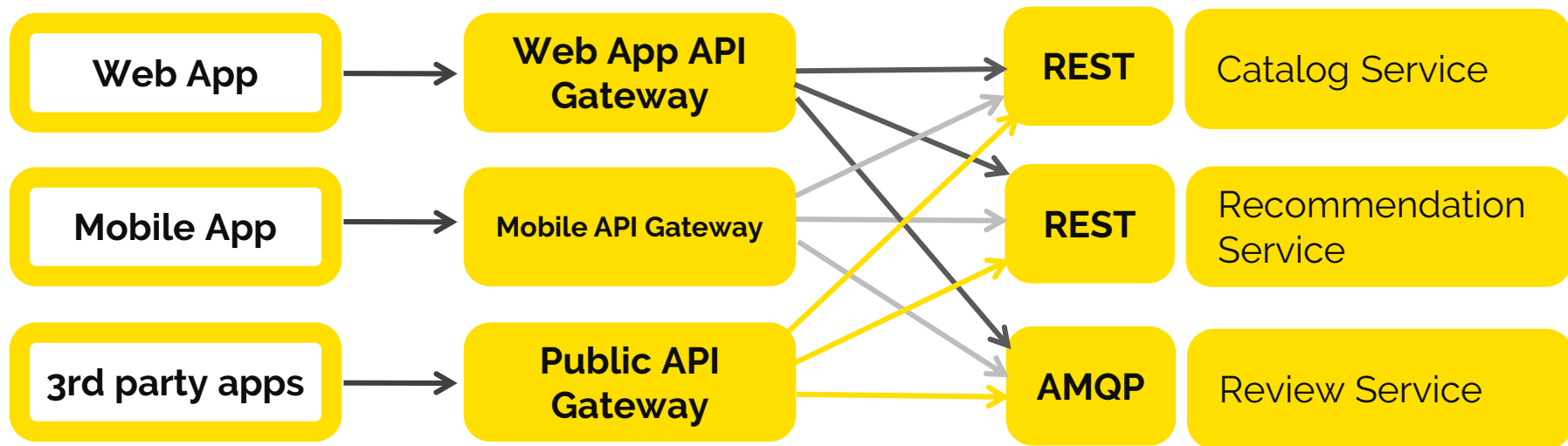


# Backend for frontend

**Мотивация** – для выполнения запроса необходимы данные из нескольких микросервисов при этом хочется обойтись минимумом запросов от клиента

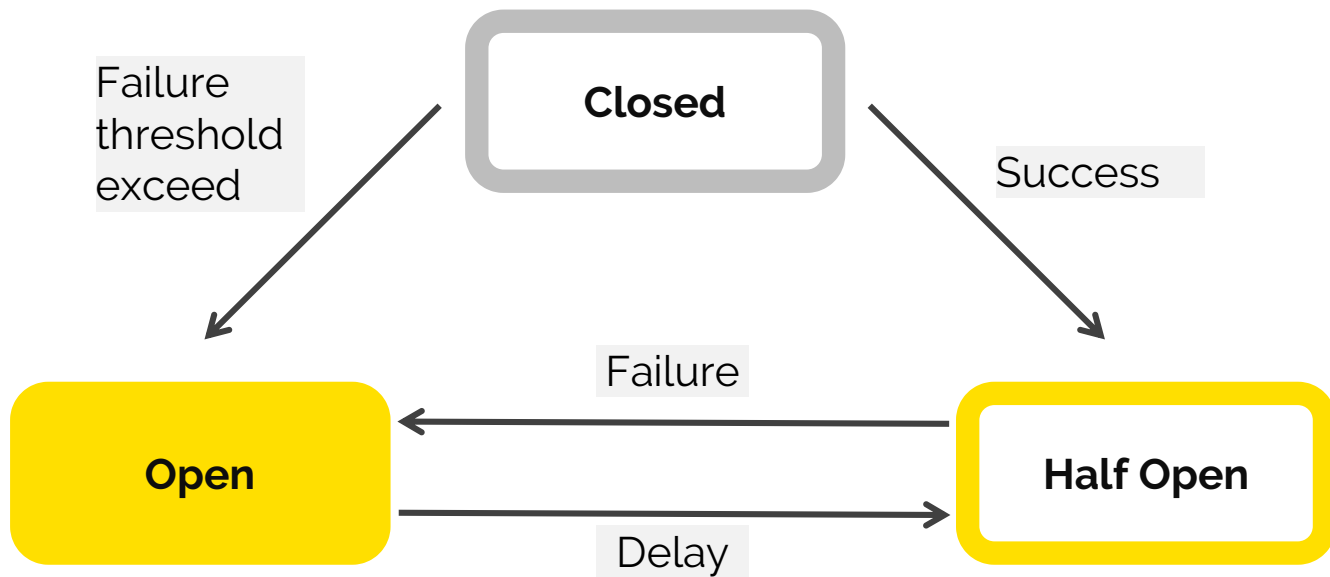
**Решение** — сделать для каждого фронтенда агрегирующий сервис, который будет принимать внешние запросы, отправлять запросы к нескольким микросервисам и объединять полученные данные.

## Variation: Backend for frontend





# Circuit breaker





# gRPC

- Бинарный протокол для удаленного вызова процедур
- Разработка Google
- Есть библиотеки для разных языков
- Поддержка как одиночных вызовов, так и потоковой передачи
- Протокол с учетом обратной совместимости

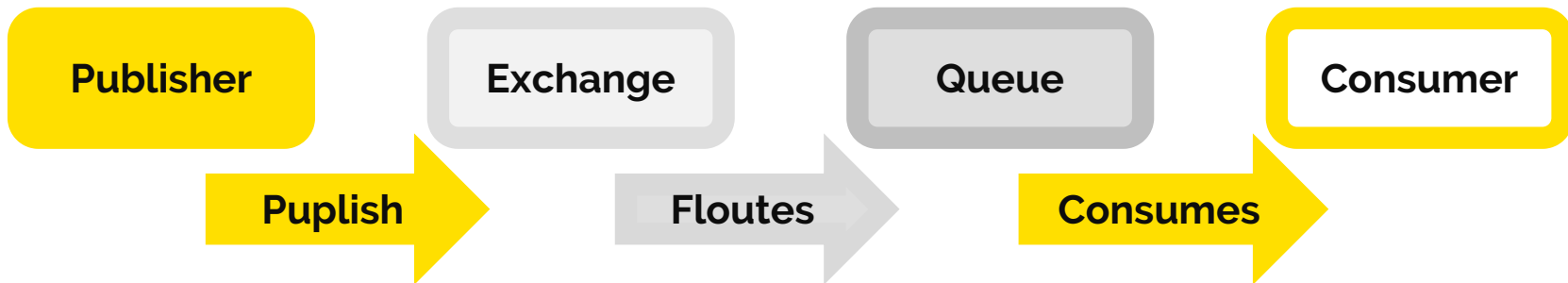
# AMQP

Открытый протокол прикладного уровня для передачи сообщений между компонентами системы. Основная идея состоит в том, что отдельные подсистемы (или независимые приложения) могут обмениваться произвольным образом сообщениями через AMQP-брокер, который осуществляет маршрутизацию, возможно гарантирует доставку, распределение потоков данных, подписку на нужные типы сообщений.

<https://ru.wikipedia.org/wiki/AMQP>

# RabbitMQ

"Hello, world"  
example routing

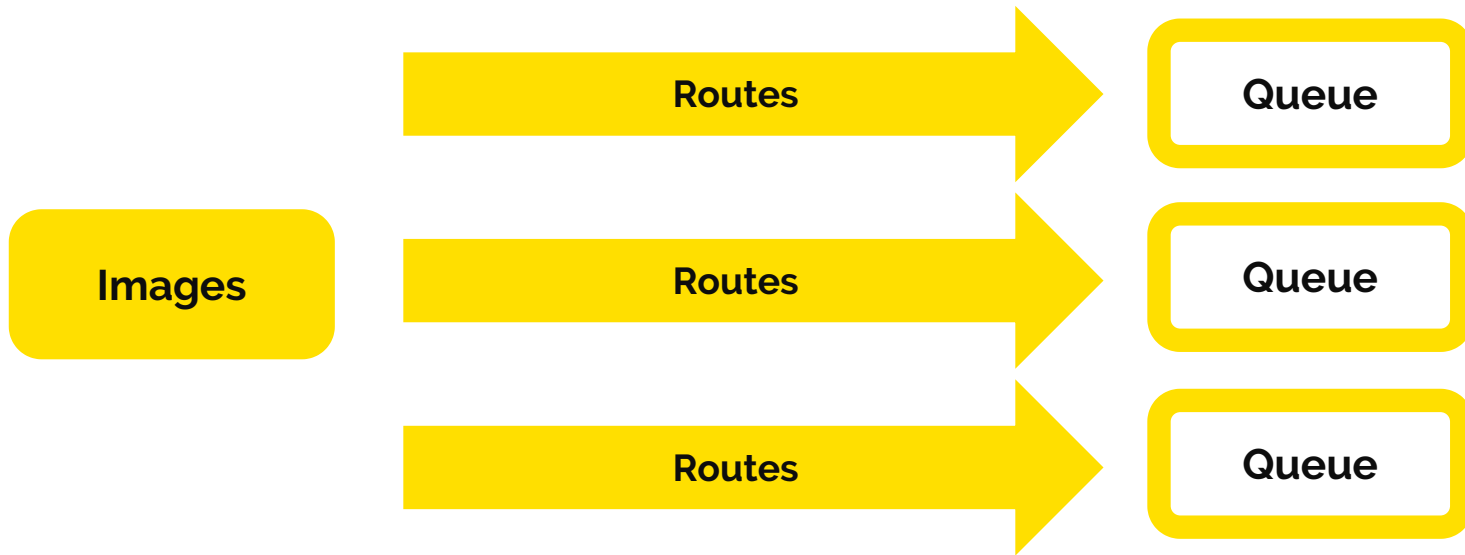


<https://www.rabbitmq.com/tutorials/amqp-concepts>

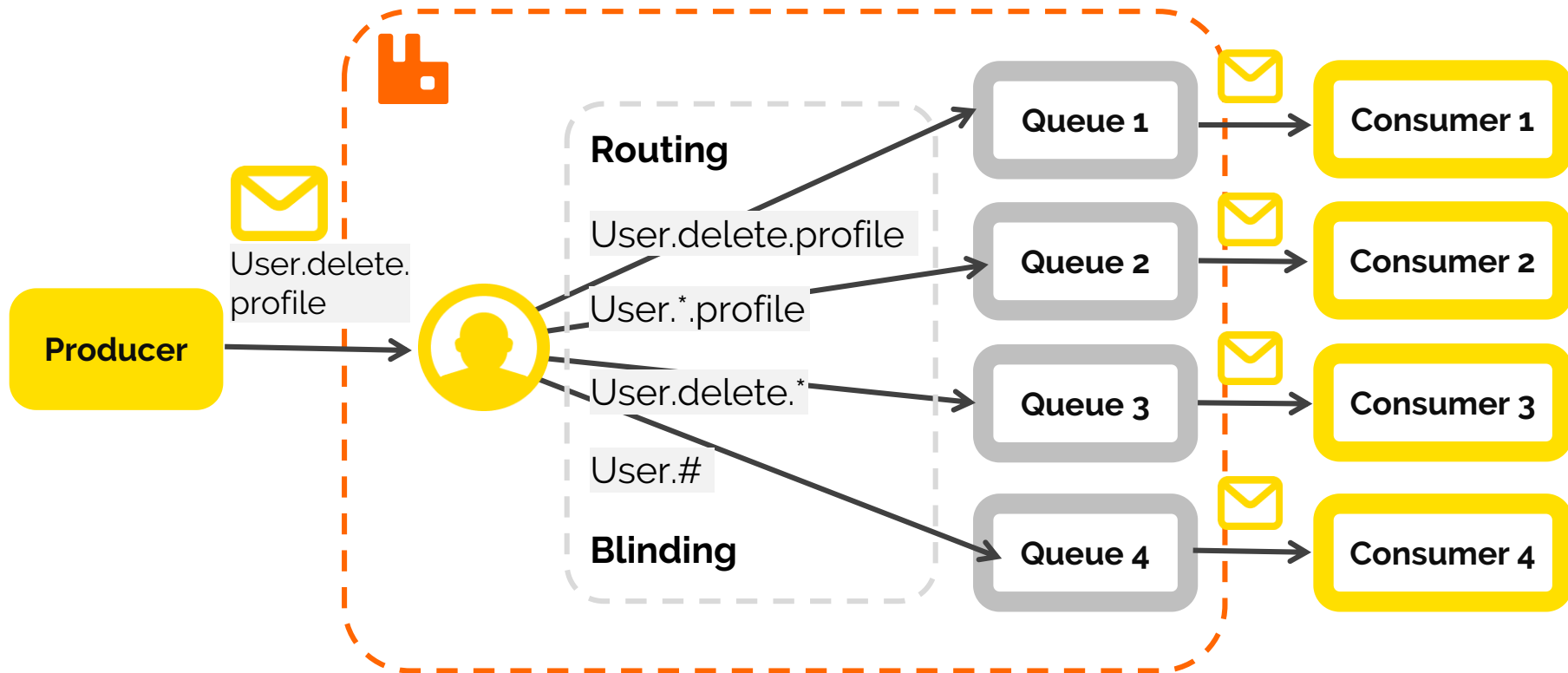
# Direct exchange routing



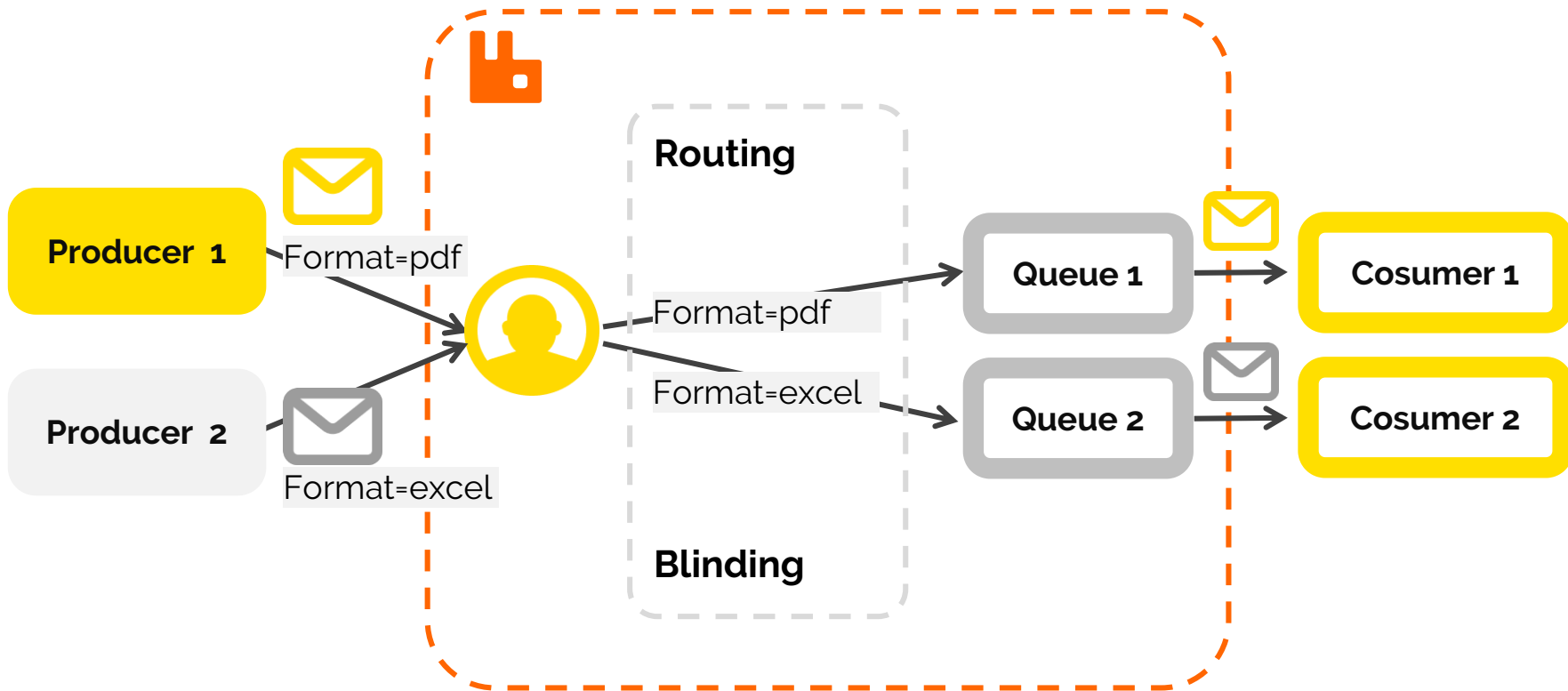
# Fanout exchange routing



# Topic exchange



# Headers exchange







# RabbitMQ

- **Распределенный сервис**, со всеми вытекающими последствиями
- Парадигма умный брокер, простой потребитель
- Заметная часть сложности приложения перемещается на **уровень брокера**
- Двухфазное чтение – **RECEIVE - ACK** – можно использовать для транзакций
- Можно возвращать сообщение обратно в очередь
- В определенных условиях может терять сообщения



# Гарантии доставки

## 1. **At-most-once delivery**

Гарантирует что сообщение может будет доставлено один раз но не больше. При этом сообщения могут теряться

## 2. **At-least-once delivery**

Гарантирует что сообщения не будут теряться, но при этом одно сообщение может быть доставлено несколько раз

## 3. **Exactly-once delivery**

Сообщение будет доставлено один и только один раз. Возможно только в ограниченных условиях.

## Гарантии сохранения порядка

# Apache Kafka

1.

## **Брокер сообщений**

оптимизированный для  
высокой пропускной  
способности

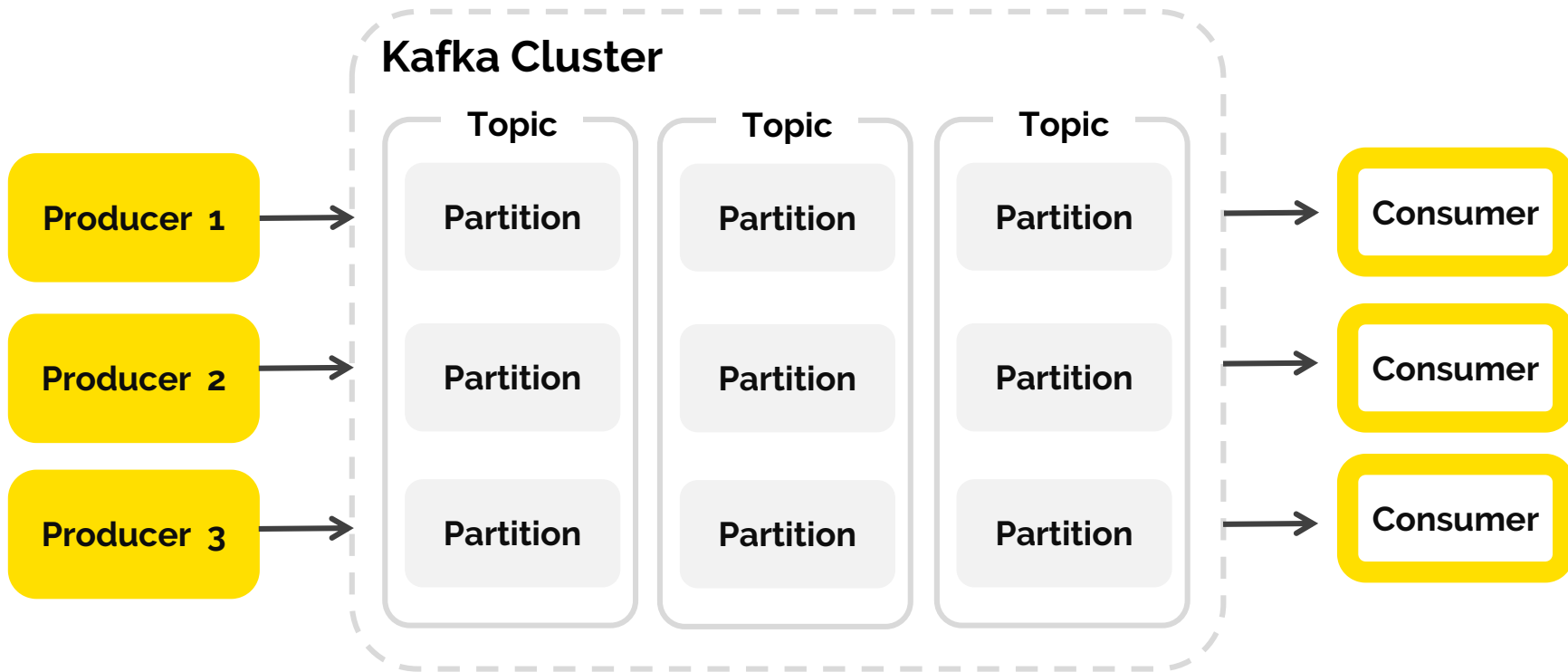
2.

**Парадигма** – глупый брокер, умный  
потребитель

3.

Сообщения сохраняются на диске  
и могут быть переобработаны

# Концепции Kafka



# Очереди: dead letter

**Мотивация** – консьюмер не смог обработать сообщение из-за ошибки в коде. Мы бы не хотели терять сообщение, но мы не можем просто вернуть его в очередь потому что оно опять будет прочитано тем же консьюмером

**Решение** — складывать такие сообщения в отдельную очередь dead letters и после исправления бага перемещать их в основную очередь вручную

# Sidecar pattern

**Мотивация** – нужно передать из сервиса А в сервис В сообщение вместе с достаточно большим объемом данных которым мы не хотим грузить сервис очередей

**Решение** — складывать данные в общее хранилище и передавать на них ссылку

**Сложности**— подчистить хранилище после того как данные перестанут быть нужны

# CQRS – Command and Query Responsibility Segregation

**Мотивация** – слишком большое количество запросов на изменения мешают друг другу и замедляют запросы на чтение.

**Решение** — выполнять запросы на изменения асинхронно

**Сложности**— обеспечить консистентность



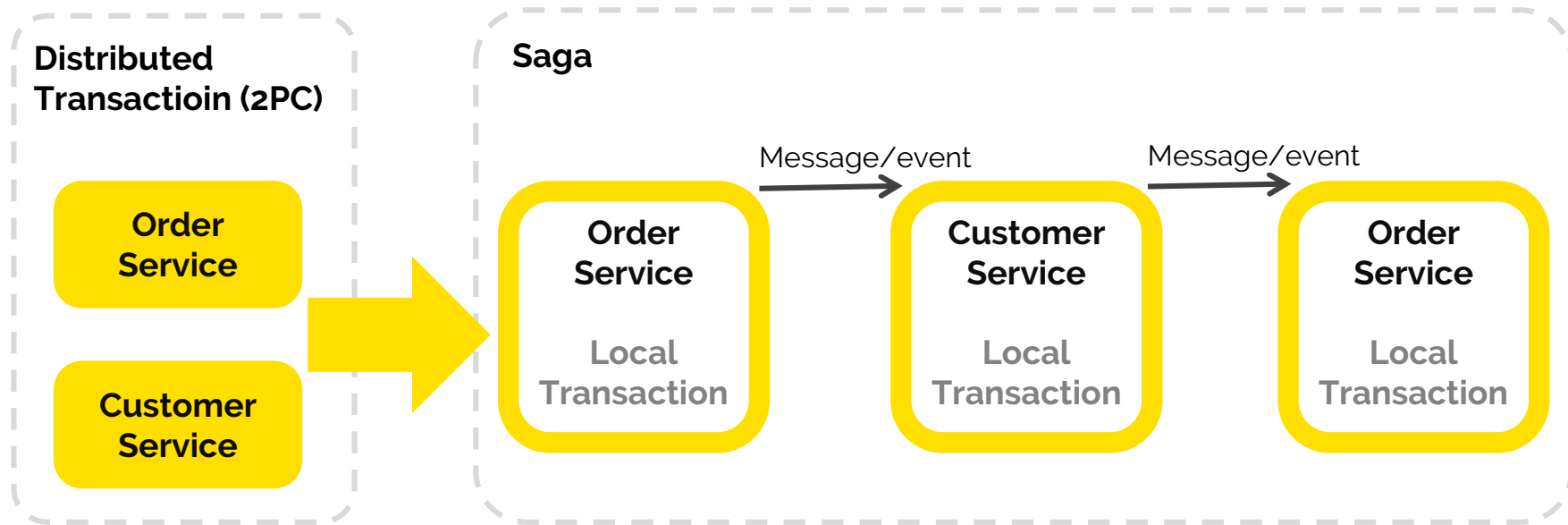
# Распределенные транзакции.

## 2 Phase Commit

1. **Координатор транзакции рассылает** всем участникам транзакции все данные необходимые для транзакции
2. **Участники транзакции сообщают** координатору о своей готовности или не готовности к коммиту
3. **Если все участники транзакции готовы**, координатор рассылает им подтверждение коммита
4. **Участники транзакции выполняют коммит** на своей стороне и отправляют координатору
5. **После того как координатор получил подтверждения** от всех участников, он завершает транзакцию



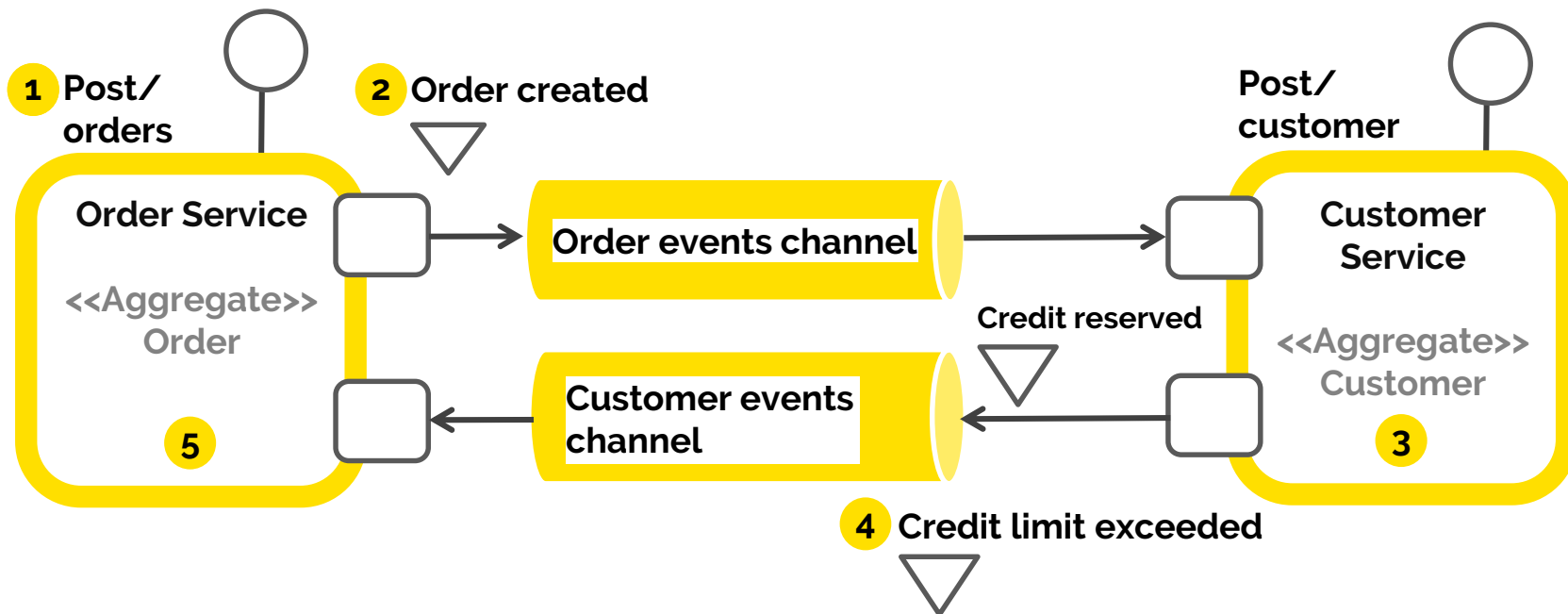
# Паттерн Saga



<https://microservices.io/patterns/data/saga.html>

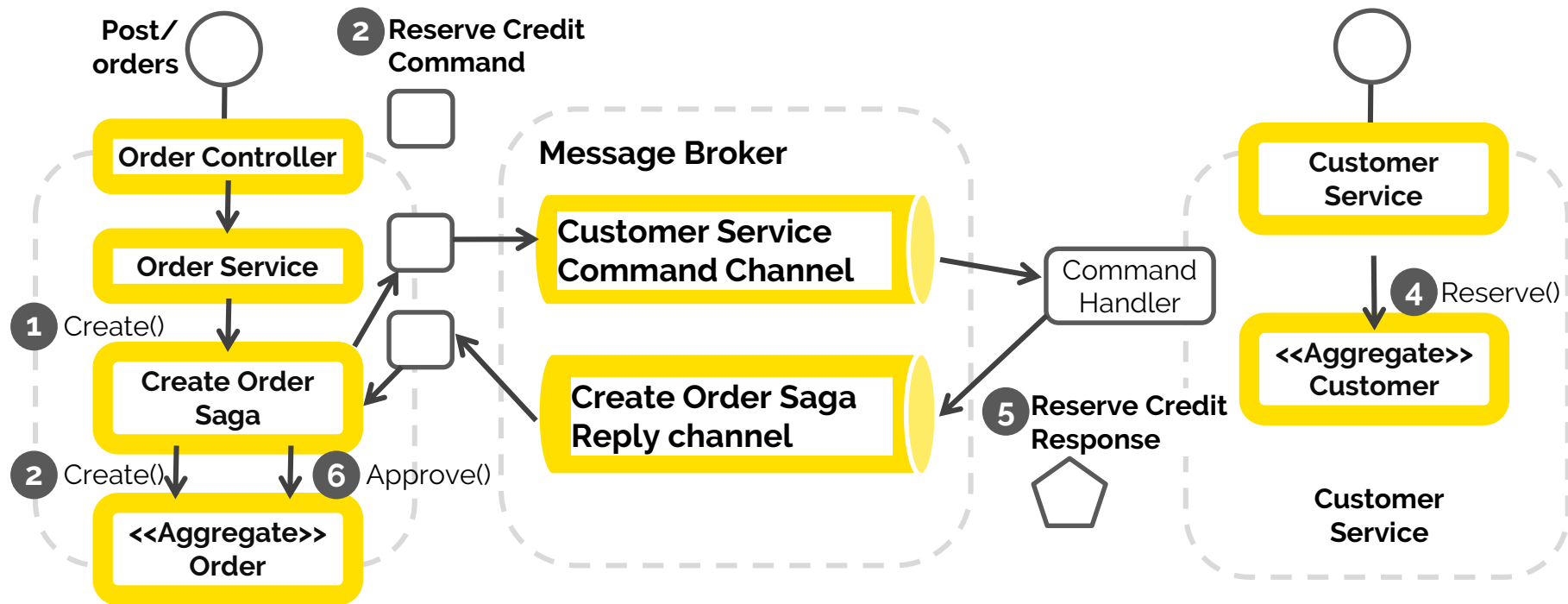
# Choreography-based saga


Каждая локальная транзакция формирует исходящее сообщение, которое, в свою очередь инициирует локальные транзакции в других сервисах



# Orchestration-based saga

Оркестратор сообщает всем участникам какие какие локальные транзакции  
они должны инициировать





# Влияние микросервисов на управление проектом

- Накладные расходы пропорциональны количеству сервисов
- Чем меньше микросервисы тем больше сложности переходит к оркестрации микросервисов и тем сложнее администрирование
- Чем меньше микросервисы, тем проще их разрабатывать в том числе и организационно
- Для микросервисной архитектуры требуется зрелый DevOps
- Микросервисы раздувают штат разработки
- Микросервисы поощряют узкую специализацию

# Компромиссы микросервисов (Фаулер)

## Микросервисы предоставляют:

1. Строгие границы между компонентами приложения
2. Независимое развертывание компонентов приложения
3. Технологическое разнообразие

## Но цена за это:

- **Распределенная система.** Работает медленнее, сложно проектировать, всегда может отказать
- **Eventual consistency** – добиться строгой консистентности практически невозможно и все вынуждены мириться с неконсистентными данными
- **Сложное администрирование**

# Достоинства микросервисной архитектуры

- **Независимость набора технологий**, развёртывания и масштабируемости сервисов.
- Стандартный, **простой и надёжный канал связи** (передача текста по HTTP через порт 80).
- **Оптимизированный** обмен сообщениями.
- **Стабильная спецификация** обмена сообщениями.
- **Изолированность контекстов** домена (Domain contexts).
- **Простота** подключения и отключения сервисов.
- **Асинхронность** обмена сообщениями помогает управлять **нагрузкой на систему**.
- **Синхронность** обмена сообщениями помогает управлять **производительностью системы**.
- Полностью **независимые и автономные** сервисы.
- Бизнес-логика **хранится** только в **сервисах**.
- Позволяют компании превратиться в **сложную адаптивную систему**, состоящую из нескольких маленьких автономных частей/команд, способную быстро адаптироваться к переменам.



# Недостатки микросервисной архитектуры

- **Высокая сложность эксплуатации:**  
Нужно много вложить в сильную DevOps-культуру.
- Использование **многочисленных технологий и библиотек** может выйти из-под контроля.
- **Нужно аккуратно управлять** изменениями входных/выходных API, потому что эти интерфейсы будут использовать многие приложения.
- Использование «**согласованности в конечном счёте**» (eventual consistency) может привести к серьёзным последствиям, которые нужно учитывать при разработке приложения, от бэкенда до UX.
- **Тестирование усложняется**, потому что изменения в интерфейсе могут непредсказуемо влиять на другие сервисы.

# Kubernetes самый крупный open source современности

11 лет назад, **6 июня 2014** года был сделан первый коммит K8S



Сегодня согласно отчету CNFC  
**K8S используют 71% компаний**  
из списка Fortune 100



# Как все начиналось

**В 1979 году** в unix появилась утилита **chroot** для изоляции процессов в рамках файловой системы

**В начале 2000-х** Google создает **Borg** – система запуска контейнерных рабочих нагрузок первого поколения

**В 2013 году** в Google появляется **Omega**, система запуска контейнерных нагрузок третьего поколения

**В 2013 году** выходит первая версия **Docker**

# Как все начиналось

Разработчиков публичного облака **Крейга Маклаки, Джо Беду и Брендана Бёрнса** так впечатлили возможности Docker, а точнее его ограничения, что **Docker был взят за основу.**

**Docker всё изменил.** Он популяризовал идею легковесной среды исполнения для контейнеров и дал простой способ упаковки, дистрибуции и развёртывания приложений. А тот инструментарий и пользовательский опыт, которые предоставлял Docker, открыли совершенно новый подход к упаковке и обслуживанию приложений в облаке. Если бы не Docker, изменивший взгляд разработчиков на облачные технологии, Kubernetes бы просто не появился.

**Брендан Бёрнс**, один из создателей Kubernetes

# Первые шаги K8S

В том же **2013 году Крейг Маклаки, Джо Беда и Брендан Бёрнс** предложили идею системы управления контейнерами с открытым исходным кодом. Они хотели перенести опыт внутренней инфраструктуры Google в сферу облачных вычислений — это позволило бы Google составить конкуренцию Amazon Web Services (AWS), на тот момент лидеру среди облачных провайдеров.

**Оцените масштаб идеи:** "Мы не можем конкурировать с AWS средствами классического маркетинга, потому что пользователи просто привыкли к их стилю работы, поэтому мы создадим новый Open Source-инструмент, который станет рыночным стандартом и навсегда изменит подход инженеров к разработке приложений ..."

# Первые шаги K8S

**10 июня 2014 года Kubernetes** был официально анонсирован на DockerCon, и с этого момента проект начал привлекать внимание сообщества. В то время он позиционировался как версия Borg с открытым исходным кодом.

# Первые шаги K8S

На ранних стадиях разработки Kubernetes включал в себя следующие базовые функции:

- Репликация для развёртывания нескольких экземпляров приложения;
- Балансировка нагрузки и обнаружение сервисов;
- Базовые проверки состояния и восстановление;
- Планирование для объединения многих машин в группы и распределения работы между ними.

**В 2015 году** на конференции O'Reilly Open Source Convention (OSCON) уже была представлена расширенная и доработанная версия оркестратора — Kubernetes 1.0. А в 2016 году Google передала Kubernetes в CNCF, подразделение Linux Foundation.



# Первые несколько лет

**К 2017 году** самые известные разработчики конкурирующих решений уже **объединились вокруг Kubernetes** и решили сделать ставку именно на него. Первыми объявили о поддержке K8s команды Pivotal Cloud Foundry, Marathon и Mesos (VMware и Mesosphere Inc.). А через пару месяцев в проект влились Docker, Inc., Microsoft Azure и AWS.



# Революционеров поглотила революция или судьба Docker

Была создана Open Container Initiative призванная стандартизировать мир контейнеров:

- Выделен интерфейс для запуска контейнеров  
**(Container Runtime Interface)**
- Выделен интерфейс управления контейнерами  
**(Open Container Interface)**

# Революционеров поглотила революция или судьба Docker

Теперь:

- В стандартной поставке K8S нет Docker, вместо этого **containerd**
- Контейнеры стали **интероперабельны** и их могут писать все, например Open Shift (crio)
- Среды для выполнения контейнеров разрабатывает каждый производитель **самостоятельно**



# В качестве заключения

У нас есть удобная единица разработки – микросервис и единица упаковки для развертывания – **мы можем сделать революции и назвать ее DevOps**

Об этом в следующих сериях 😊.....

**N\***

**Спасибо  
за внимание**

ФИТ Лекция №6. 25'