



**ARC-015**

# **Микросервисы**

Взаимодействие сервисов

Владислав Родин

**luxoft**  
A DXC Technology Company

# Значимость

- **Независимо от того, как вы нарезаете свое приложение, общий принцип успеха микросервисов заключается в том, чтобы конечные приложения были отделены друг от друга**

# Взаимодействие клиент-сервер

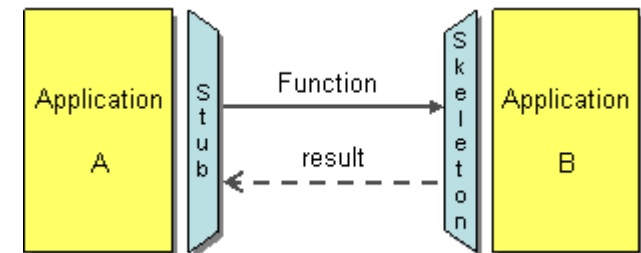


# Взаимодействие клиент-сервер: Варианты

		Вызов	Сообщение
Синхронное		Синхронный вызов	Ожидание на клиенте и задержка выполнения на сервере Сложно реализовать. Антипаттерн
Асинхронное	One-Way	Асинхронный вызов	Нотификация
	Duplex	Отложенный запрос (Deferred response) Обратный вызов (Callback)	Запрос-ответ (Request/Reply)

# ВЫЗОВ

- **Преимущества:**
  - Просто в реализации (синхронный вызов)
  - Предсказуемое время обработки
- **Недостатки:**
  - Сервер не может принимать запросы будучи занятым



# Сообщение

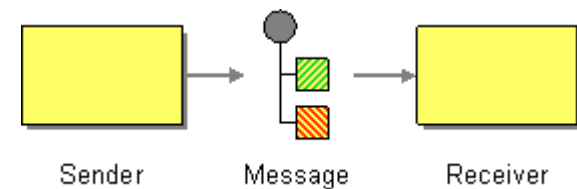
## ■ Преимущества:

- Сервер может принимать запросы будучи занятым
- Сглаживаются всплески активностей
- Выше пропускная способность

(загрузка новой задачи на обработку не по сигналу из вне, а по готовности сервера)

## ■ Недостатки:

- Требуется временное хранение сообщений (очередь)



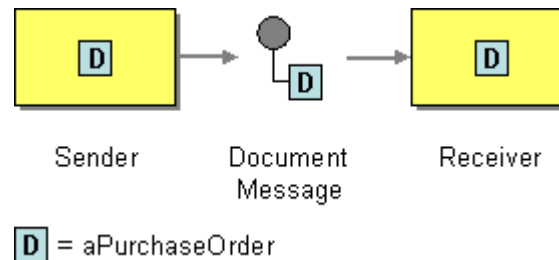
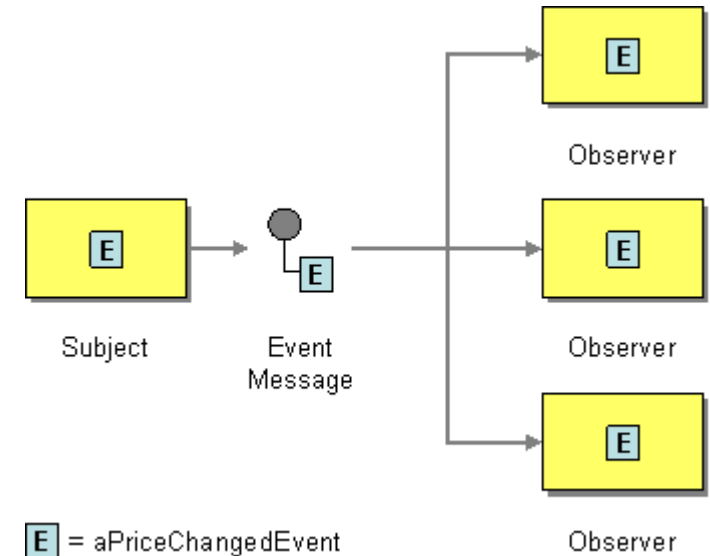
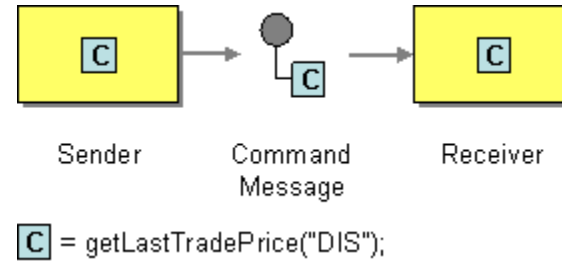
# Типы сообщений (messages)

- Запросы (Request)

- Запросы (Query)
- Команды (Commands)

- Ответы (Response)

- События (Events)
- Данные (Data)



# Очередь сообщений (Message Bus)

- Для организации доступа клиента и сервера к сообщениям используется очередь

- Очередь может располагаться

- **на клиенте**

- сложно организовать доступ сервера к клиенту, сложно решить задачу для множества клиентов

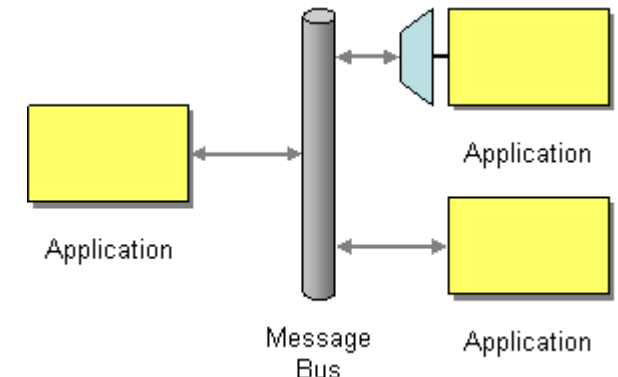
- **на сервере**

- производительность, большая надежность, отслеживаемость

- **на отдельном узле (брокер сообщения)**

- может принимать запросы будучи отключенным, клиент не знает сервер

Прямое сообщение

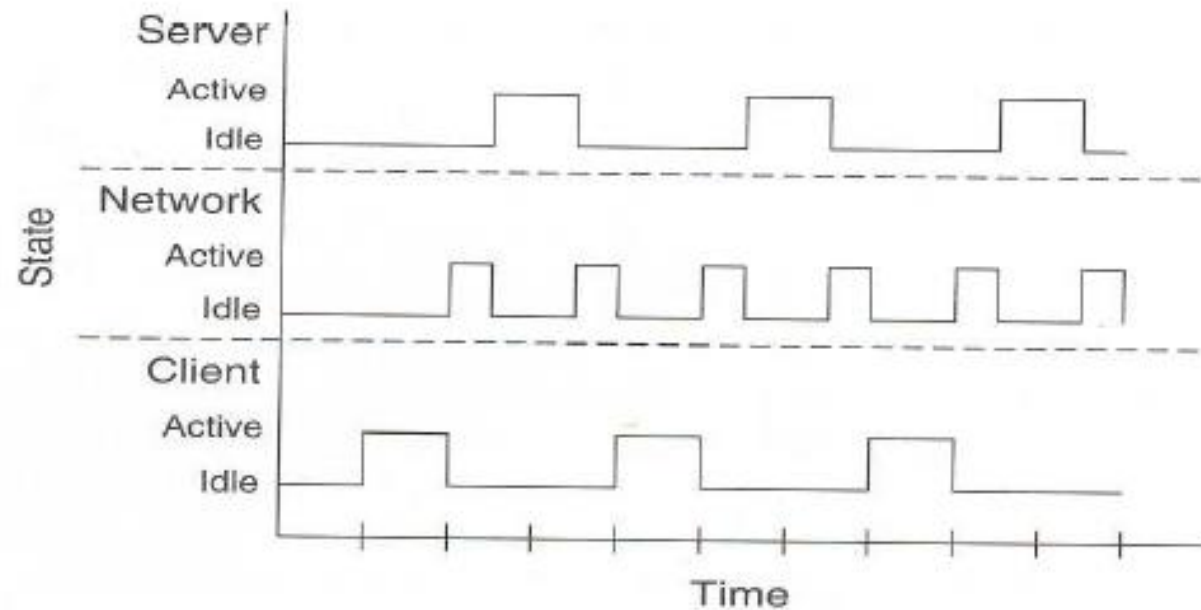
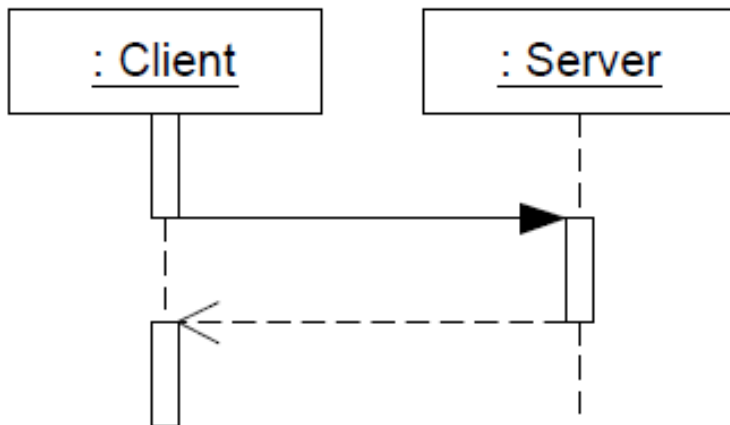




# Синхронное взаимодействие

# Синхронное взаимодействие

Стиль общения, при котором вызывающий абонент ожидает ответ вызываемого



# Синхронное взаимодействие: Результат

- **Преимущества:**

- **Просто в реализации** (синхронный вызов)
  - что делает его подходящим в большинстве ситуаций
- **Широкая распространенность**

- **Недостатки:**

- **В процессе ожидания простаивают занятые ресурсы (потoki)**
  - Можно решать за счет легковесных потоков (fibers)
- **Всплеск на одном компоненте может затопить все остальные**
- **Сложно организовать синхронное сообщение**

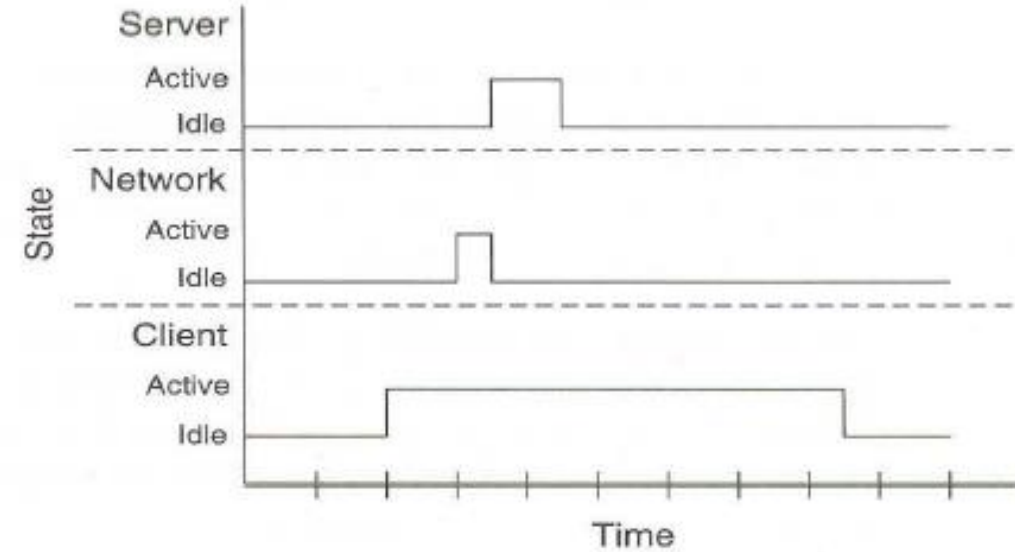
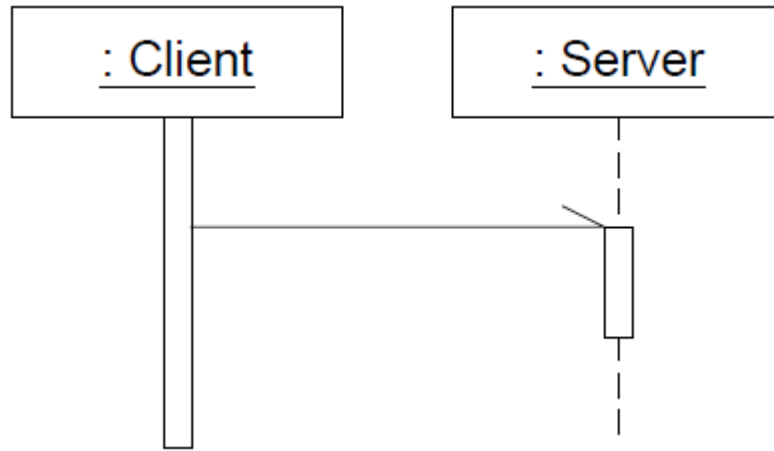
# Асинхронное взаимодействие

# Асинхронное взаимодействие

Взаимодействие без ожидания ответа.

- Варианты:
  - Асинхронные **вызовы**
    - One-way
    - Обратный вызов (callback)
  - Асинхронные **сообщения**
    - Сообщение-нотификация
    - Запрос-ответ (Request-Replay)

# Односторонний (One-way) вызов



# Асинхронные взаимодействия: Проблема

- При асинхронном взаимодействии затруднена организация ответа
- Варианты решения:
  - Обратный вызов (Callback)
  - Запрос-ответ (Request-Reply)

# Асинхронный возврат

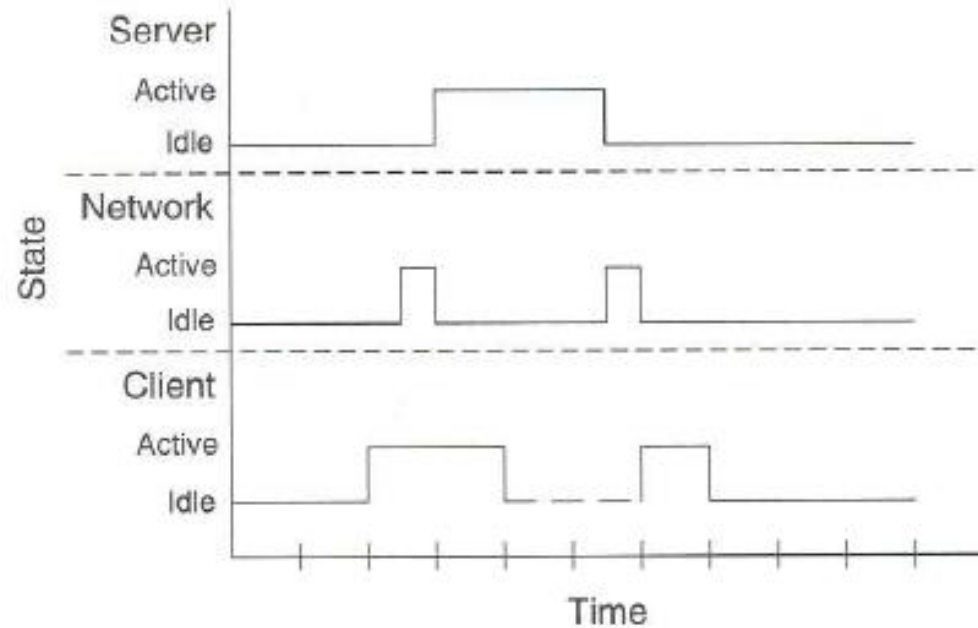
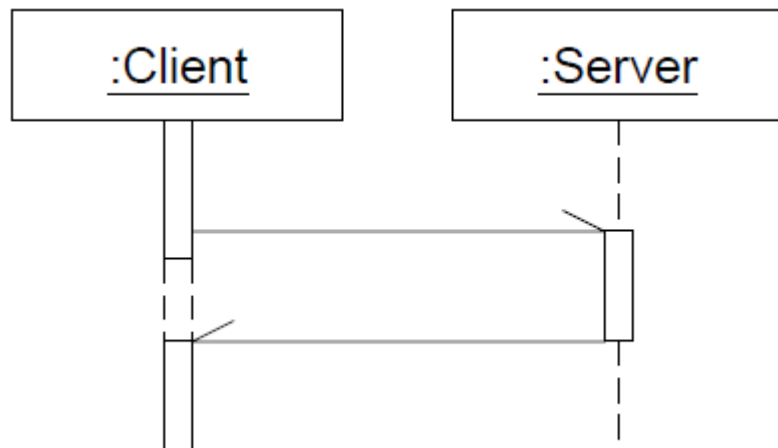
- При асинхронном взаимодействии затруднена организация ответа

## Возможные решения:

- Вызовы (возврат обработан сразу же по получению)  
требуется доступность клиента
  - Отложенный запрос (Deferred response)
    - Отложенный запрос должен решить проблему таймаута ожидания, не эффективен по времени отклика
  - Обратный вызов (Callback)
    - Обратный вызов ведет к обоюдной видимости клиента и сервера (Callback Hell)
- Сообщения (возврат сохранен в очереди и может быть обработан позже)  
не требуется доступность клиента
  - Возвращаемое сообщение (Request/Reply)
  - Возвращаемое сообщение – событие (Event)



# Обратный вызов (Callback)



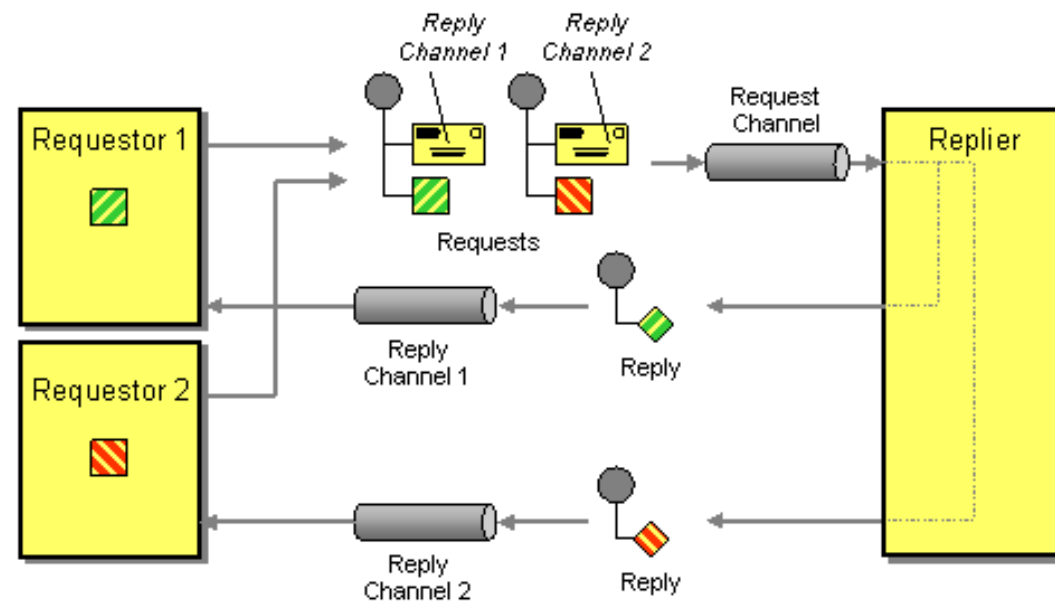
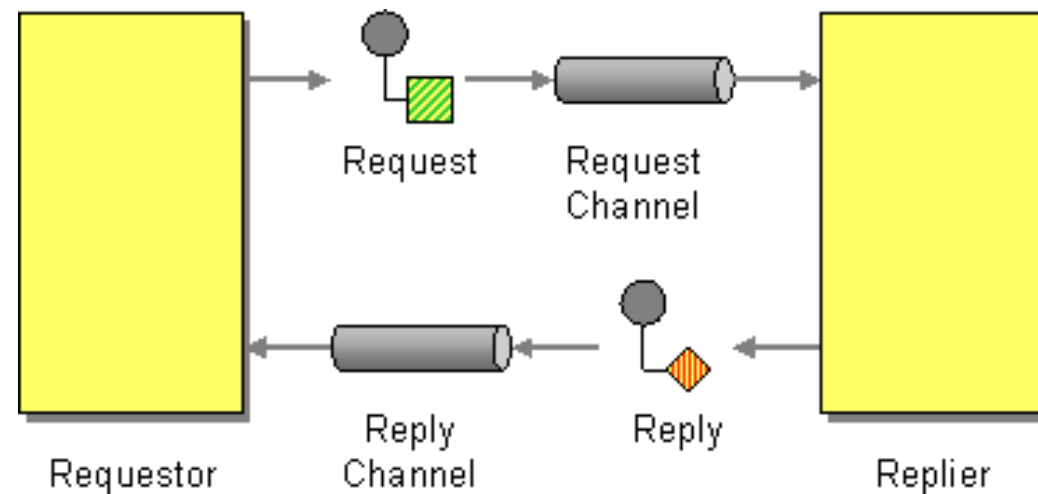
## Сильная связанность при асинхронном возврате

- При организации прямого асинхронного возврата возникает проблема двунаправленной зависимости
- Высокая связанность сервисов приводит к потере модифицируемости
- Высокая связанность затрудняет понимание процесса взаимодействия (callback hell)



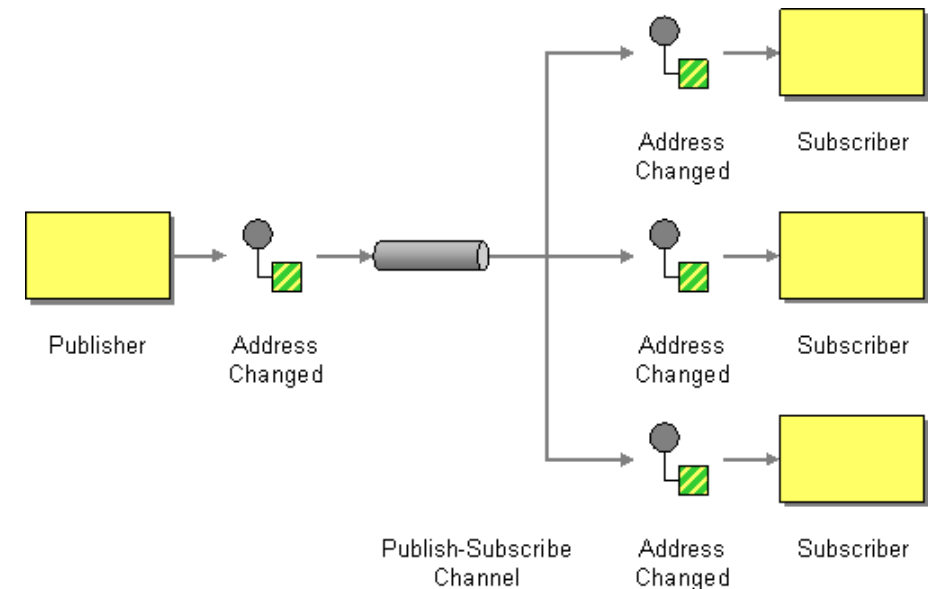
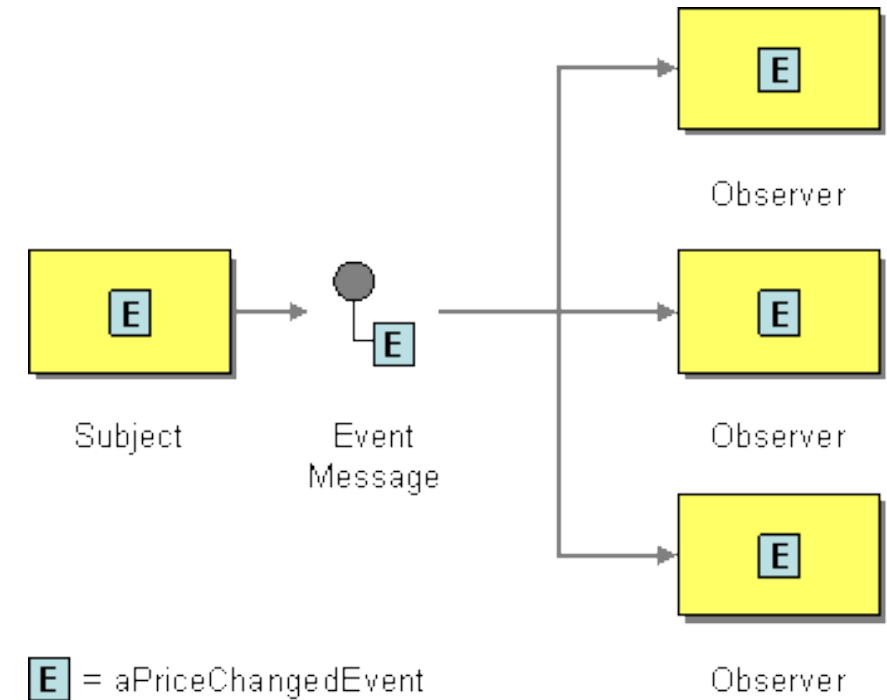
# Запрос-ответ (Request-Reply)

- Проблема:
  - Определение адреса получателя в случае нескольких клиентов
- Решение
  - Адрес возврата (Return address)



# Событие (Event)

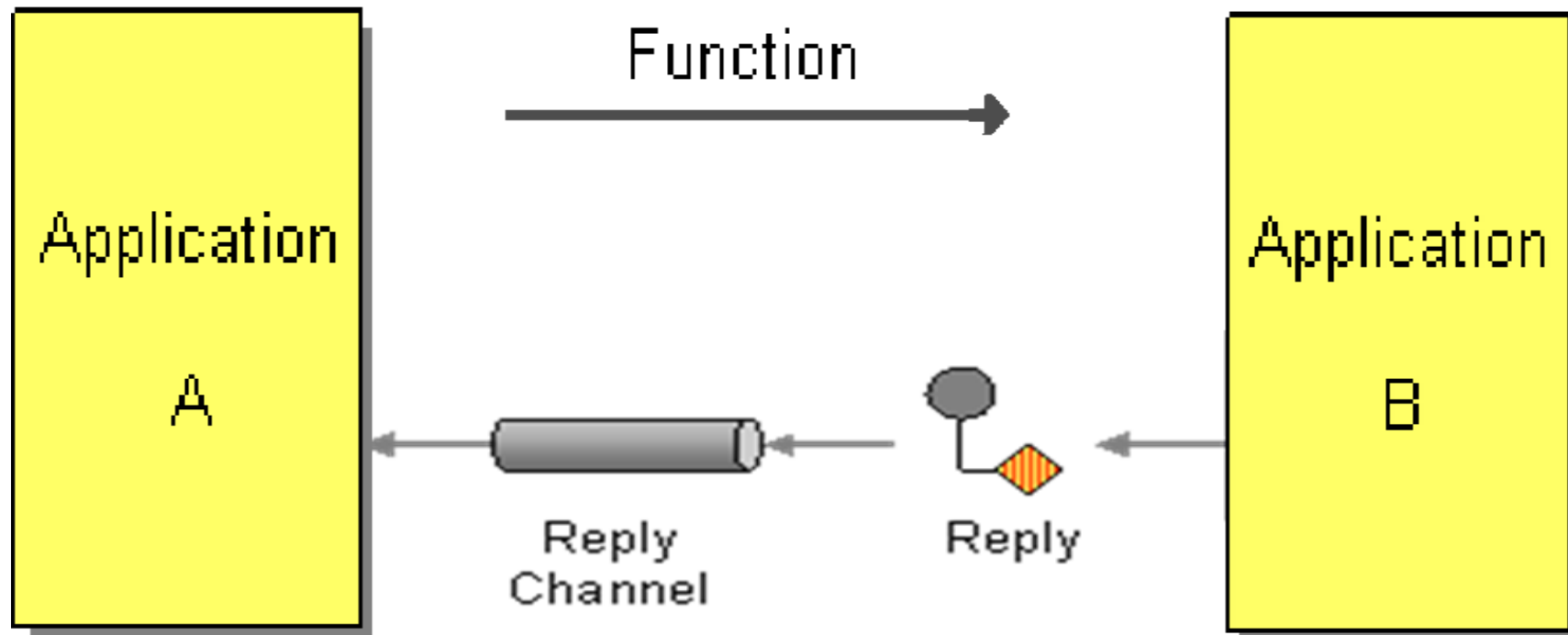
- Результат операции
- Может быть использовано при взаимодействии в распределенной системе как одно из сообщений
- Не должно иметь адресата
  - Для устранения проблемы двунаправленных связей
- Проблема:
  - Доставка результата клиенту
- Решение:
  - Подписка (Publisher-Subscriber)



# Гибридное взаимодействие

## Sync – Async

- Синхронный вызов и асинхронное событие на возврат

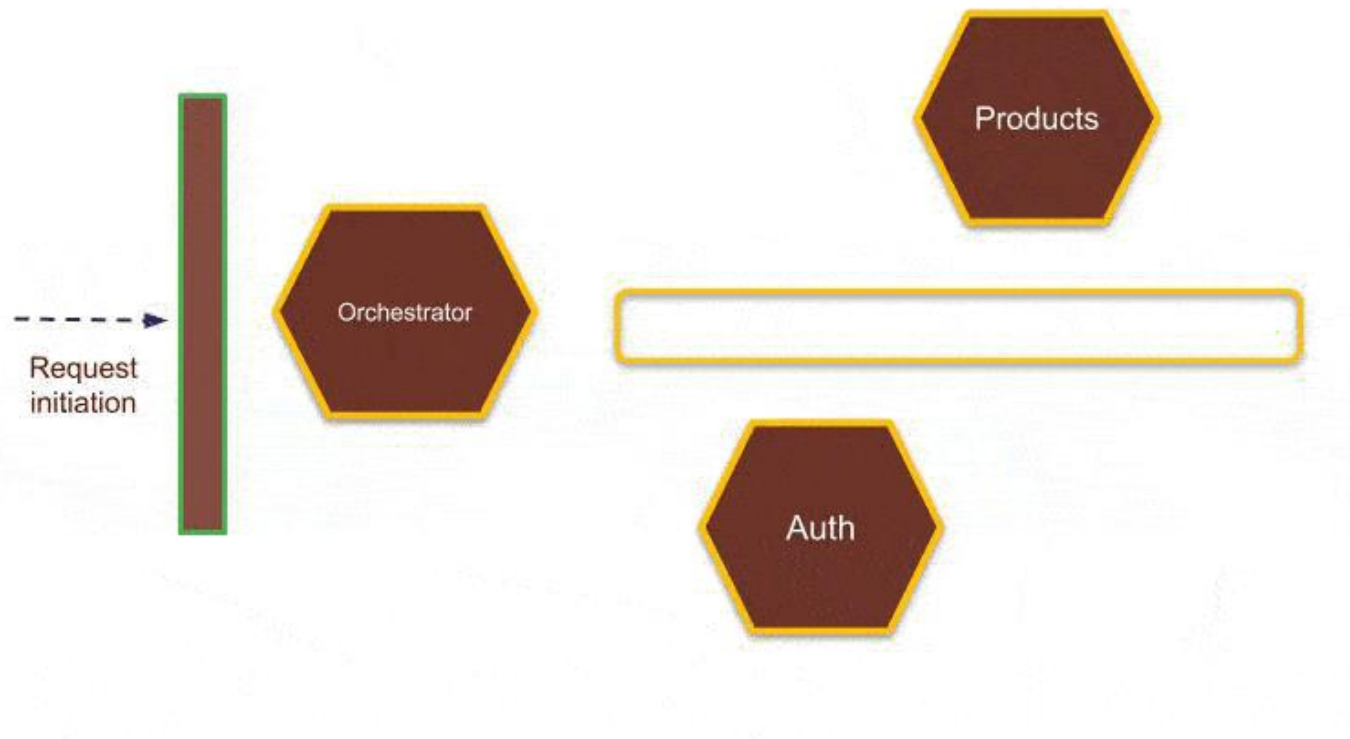


# Синхронный клиент и асинхронный сервис

## Sync – Async – Sync

- **Причины:**
  - Не возможно переписать код клиента
  - Клиент представляет UI
- **Решения:**
  - Синхронная обертка (Sync Wrapper)
  - Двойная поддержка (Dual Support)
  - Разделение обработки команд и запросов (CQRS)

# Синхронная обертка (Sync Wrapper)



Не самое эффективное, но простое решение

Должно быть рассмотрено в первую очередь

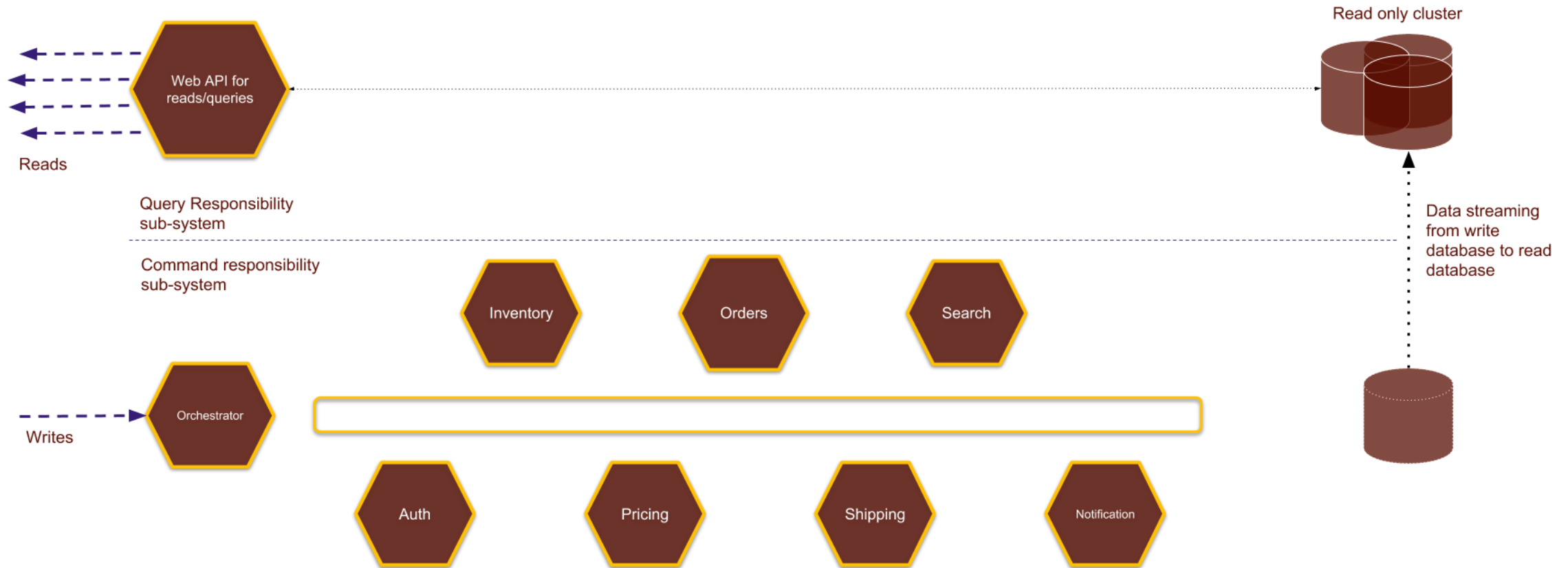
## Двойная поддержка (Dual Support)

**Каждый сервис может поддерживать синхронные запросы и асинхронные записи**

- **Решение хорошо подходит для систем со средним уровнем масштабирования**
- **Теряется возможность независимой оптимизации операций записи и чтения**



# Разделение обработки команд и запросов (CQRS)



# Асинхронные взаимодействия: Результат

- **Достоинства**
  - Не занимает ресурсы клиента на время ожидания ответа
- **Компромиссы**
  - Высокая сложность системы
  - Сложно поддерживать согласованность данных

# Разделение обработки команд и запросов (CQRS)

# Разделение обработки команд и запросов Command Query Responsibility Segregation (CQRS)

**Паттерн CQRS впервые представил  
Грег Янг (Greg Young)**



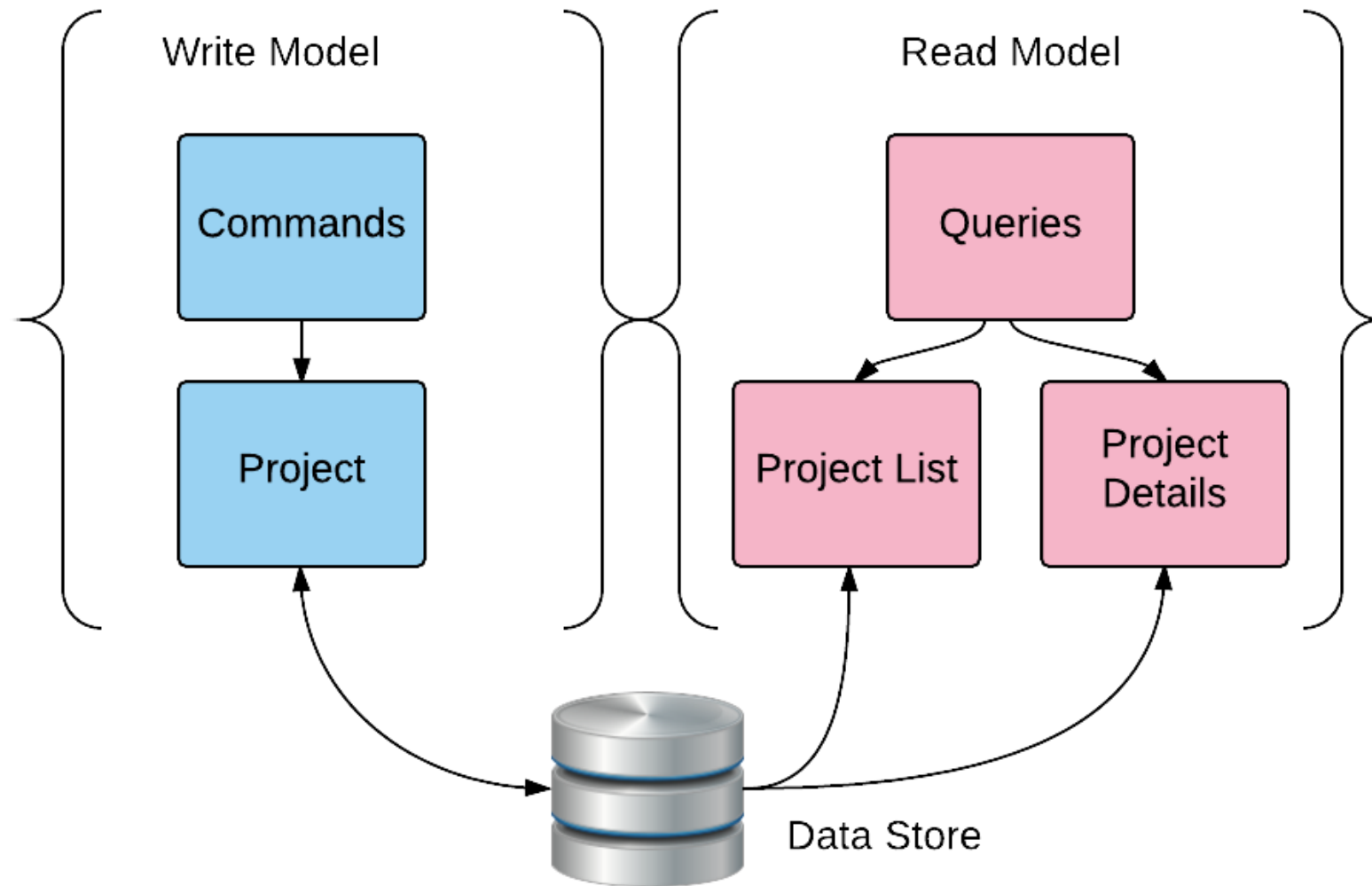
**Паттерн основан на известном принципе дизайна  
«разделение команды и запроса»  
Command-query separation (CQS)  
описанном Бертраном Мейером (Bertrand Meyer)**

# Разделение обработки команд и запросов

## Проблемы

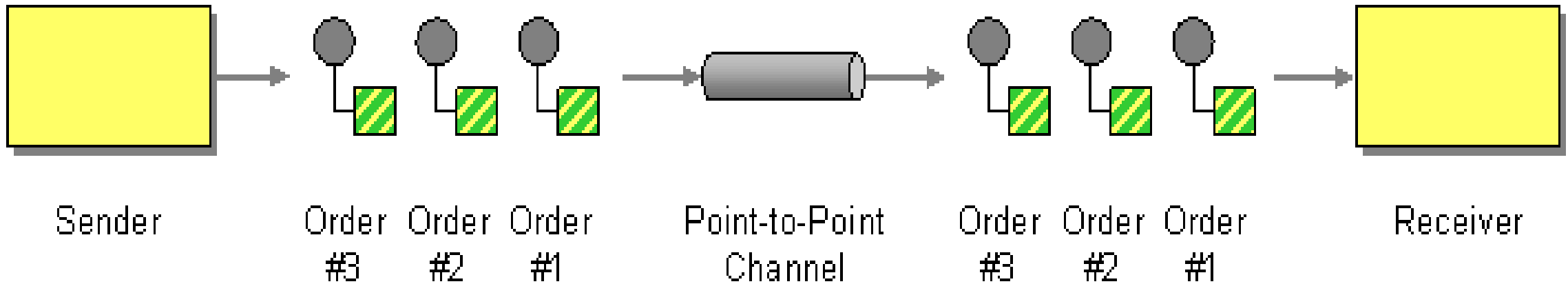
- **Операции чтения и модификации существенно отличаются друг от друга**
  - **Использовать для этих операций одни и те же программные механизмы не целесообразно**
- **Операции чтения и модификации могут ограничивать пропускную способность друг друга**

# Архитектурный уровень



# Брокер сообщений

## Прямое соединение (Point-to-point channel)



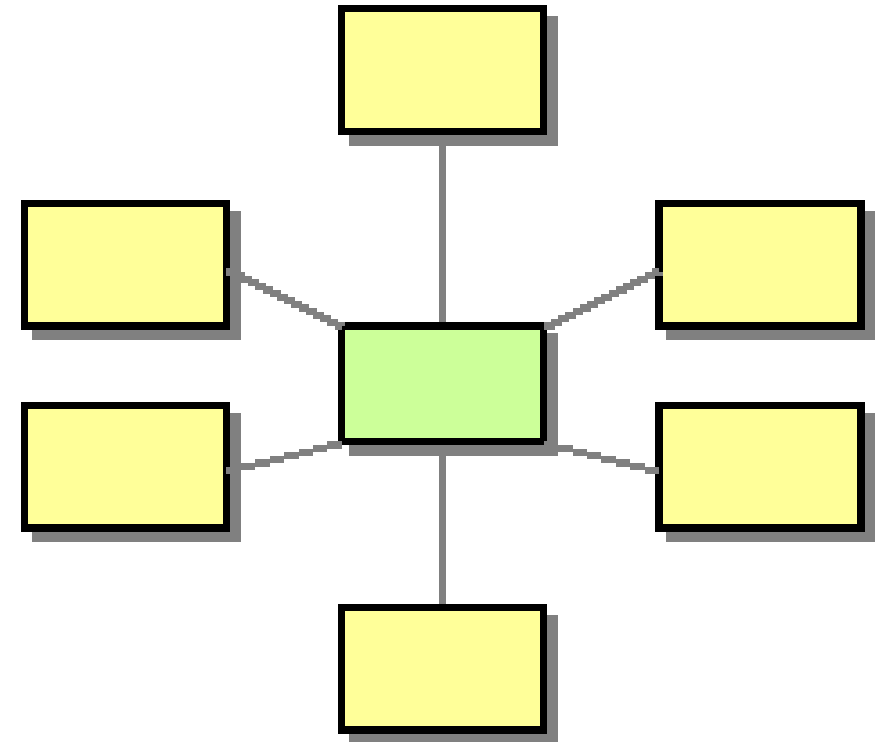


## Прямые сообщения: недостатки

- **Требуется механизм обнаружения сервисов**
  - **Сильная (прямая) связанность сервисов ограничивает модифицируемость**
- **Требуется механизм балансировки**
- **Временная недоступность сервера не позволяет клиенту передать сообщение**

# Брокер сообщений (Message Broker)

- Посредник, принимающий сообщения из различных источников, определяющий пункт назначения и обеспечивающий доставку сообщения в нужный канал



# Брокер сообщений: Результат

## ■ Достоинства

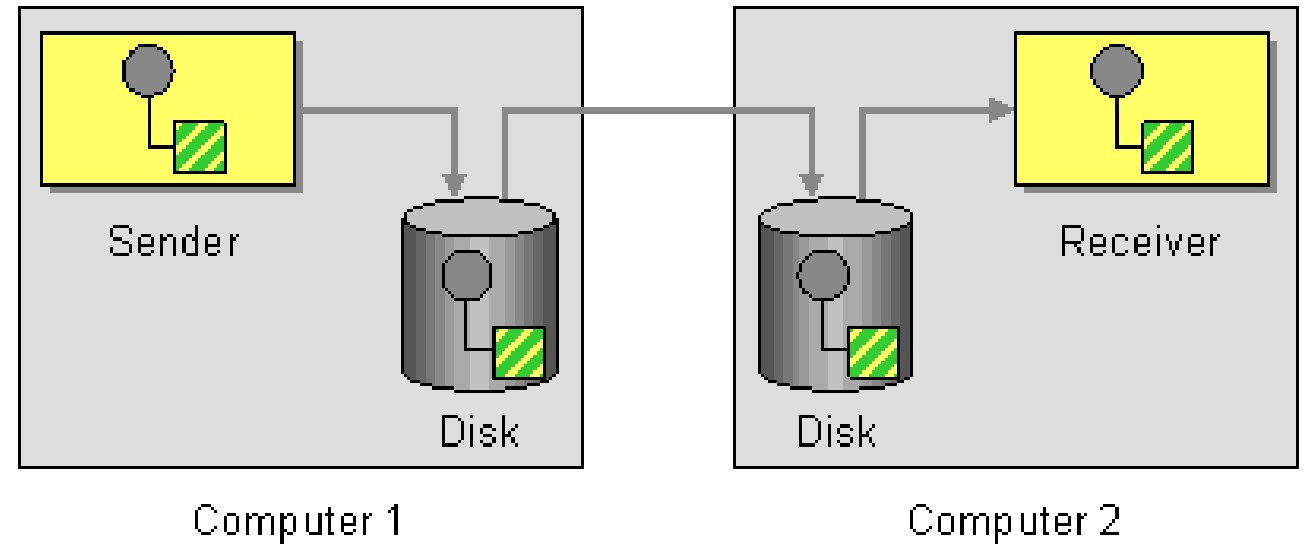
- Согласованная семантика связи и доставки сообщений
- Обнаружение сервиса необходимо только для самой очереди
- Не требуются балансировщики нагрузки и упрощается масштабирование
- Может быть обеспечена **гарантированная доставка**
- Этот подход более распространен

## ■ Недостатки

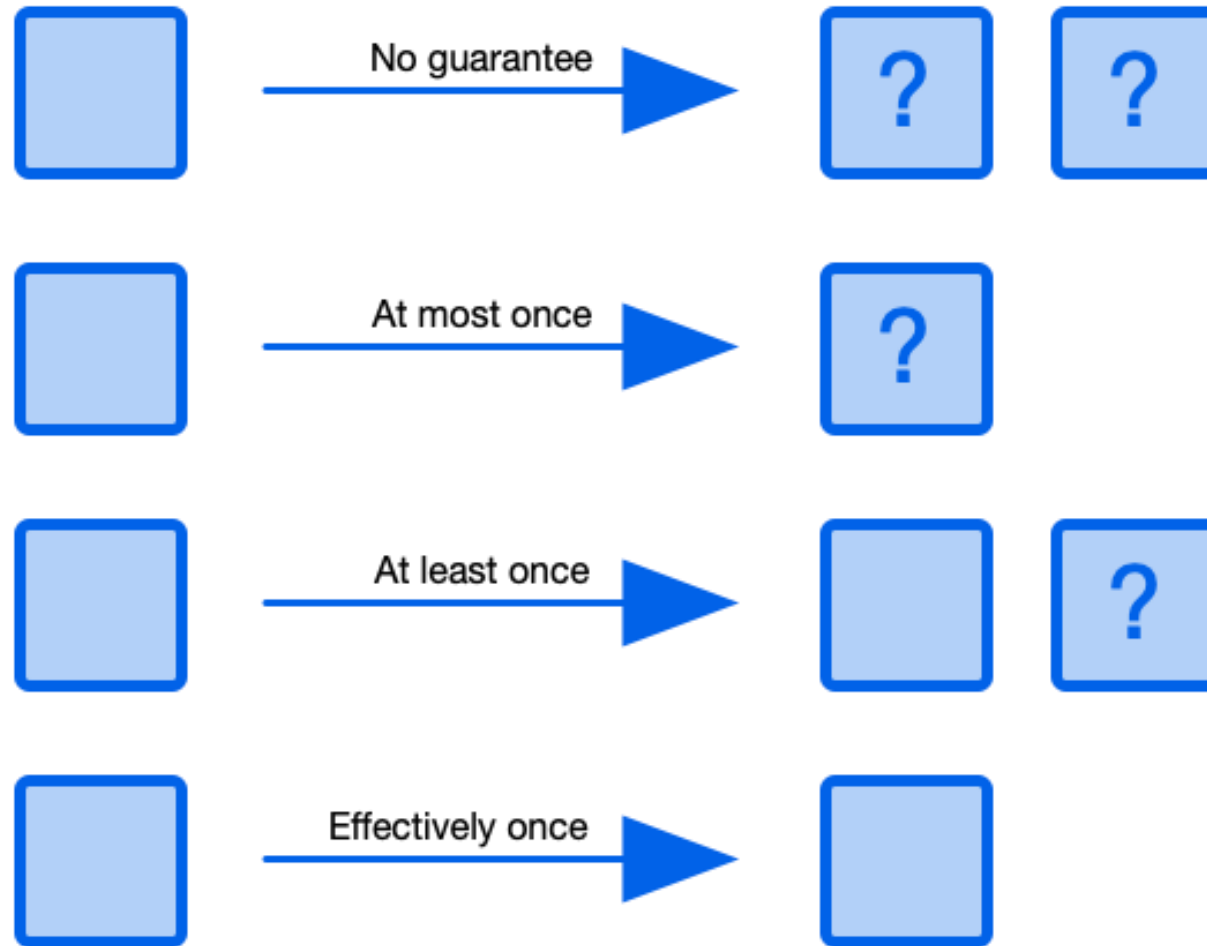
- Центральная шина – единая точка отказа  
Необходимо масштабирование шины, что приводит к дополнительным проблемам
- Запросы на чтение требуют посредничества

# Гарантированная доставка (Guaranteed Delivery)

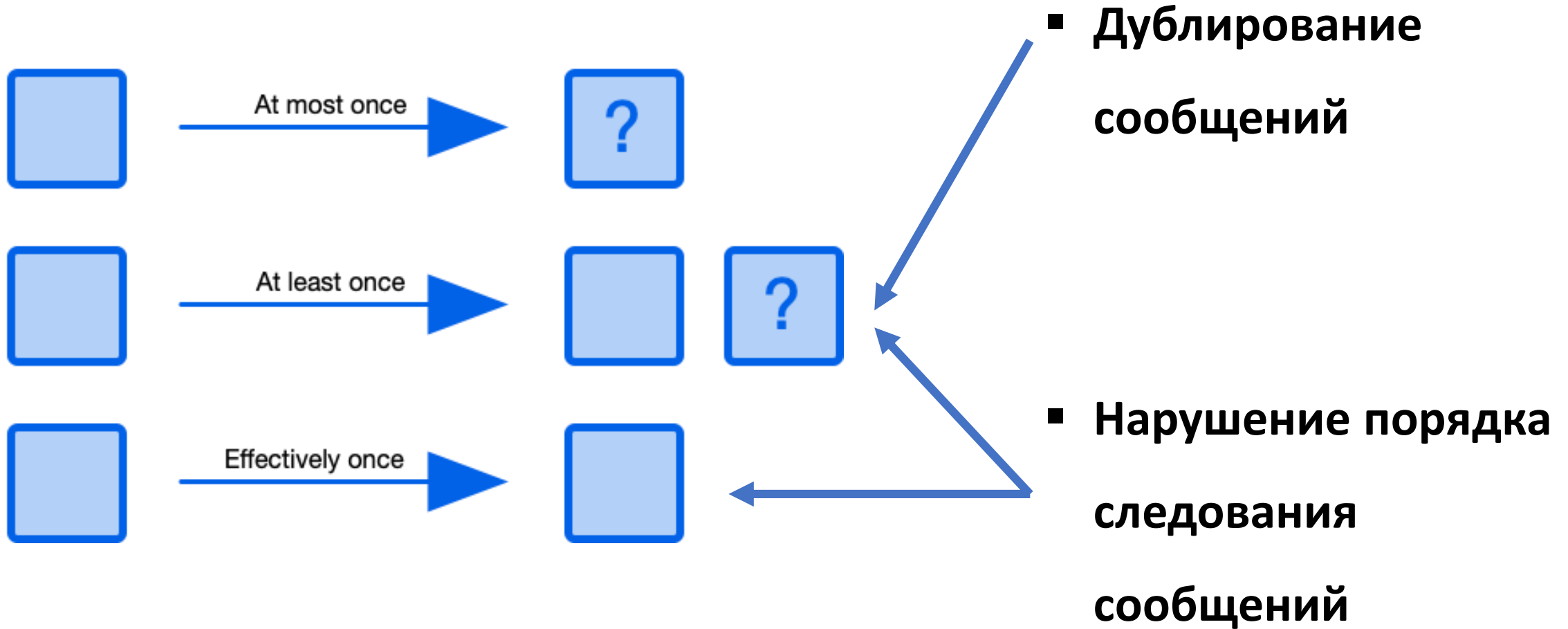
- **Сохранение сообщений**  
предотвращает их потерю даже в случае сбоя канала связи
- **Недостатки**
  - Усложнение системы
  - Снижение производительности



# Гарантированная доставка: Варианты



# Проблемы распределенных очередей



# Нарушение порядка следования сообщений

- **Решения:**

- **Временная метка (анти-паттерн, если нет атомных часов)**

- Если вы получаете событие с определенной временной отметкой, вы не знаете, нужно ли вам еще ждать какое-то предыдущее событие с более низкой временной отметкой или все предыдущие события прибыли

- Не стоит полагаться на синхронизацию часов

- **Версионирование (транзакционный раздатчик версий)**

- **Выгрузка вместо проталкивания (считывание батчем)**

# Дублирование сообщений

- **Решение:**
  - **Идемпотентный получатель (Idempotent Receiver)**
    - Идемпотентное сообщение
    - Отслеживание сообщений (Tracking messages)



# Паттерн «Идемпотентный получатель» (Idempotent Receiver)

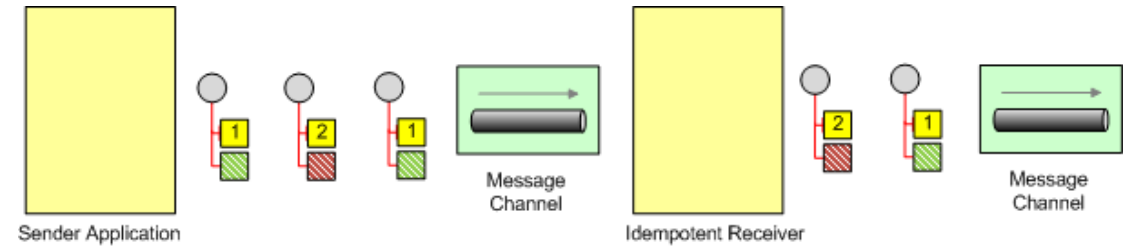
- Идемпотентность

Одно и тоже действие может выполняться сколь угодно много раз  
с одним результатом

$$F(x) = F(F(x))$$

- Решения:

- Идемпотентное сообщение
- Отслеживание и удаление дубликатов
  - сохранять все обработанные сообщения и игнорировать те, которые уже были обработаны

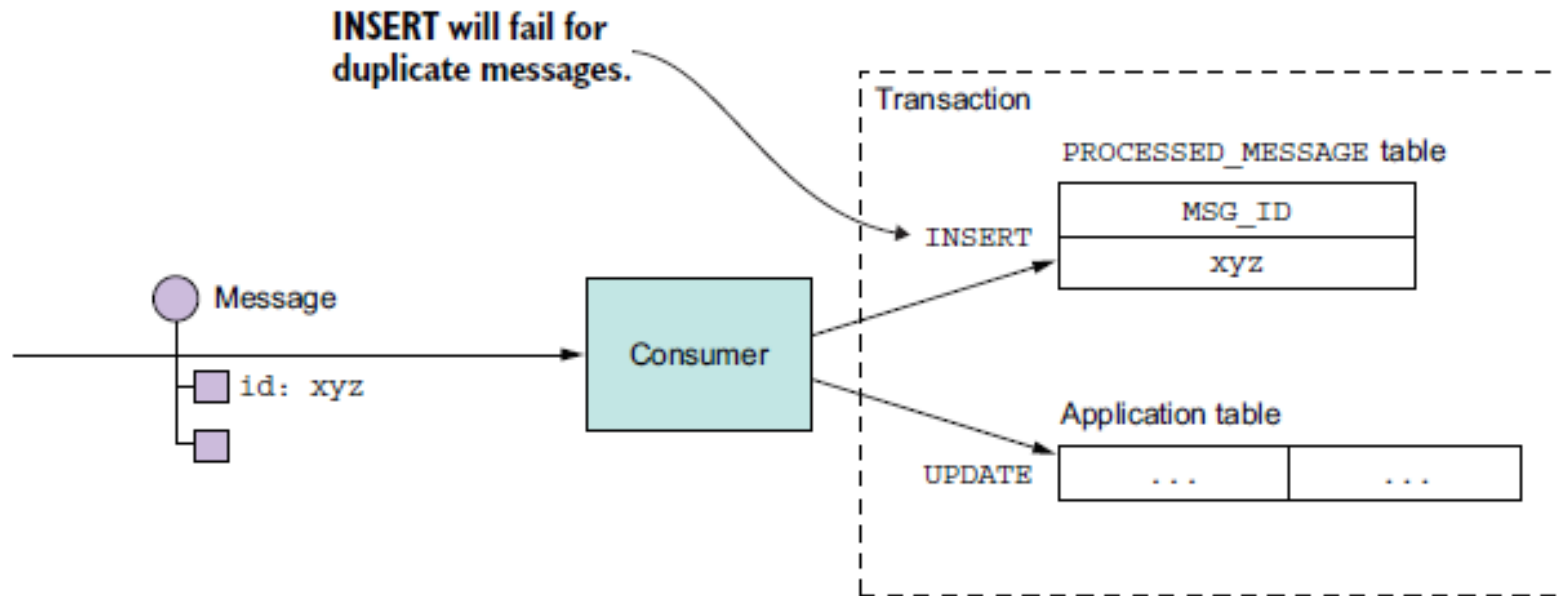


# Идемпотентное сообщение

- Определить семантику сообщений так, чтобы применение сообщения несколько раз вело к одному и тому же результату вне зависимости от того, сколько раз оно было применено
- Пример
  - вместо  
*«положить на счет 10\$»*  
использовать  
*«установить остаток на счёте в 150\$»*

# Отслеживание сообщений и отклонение дубликатов

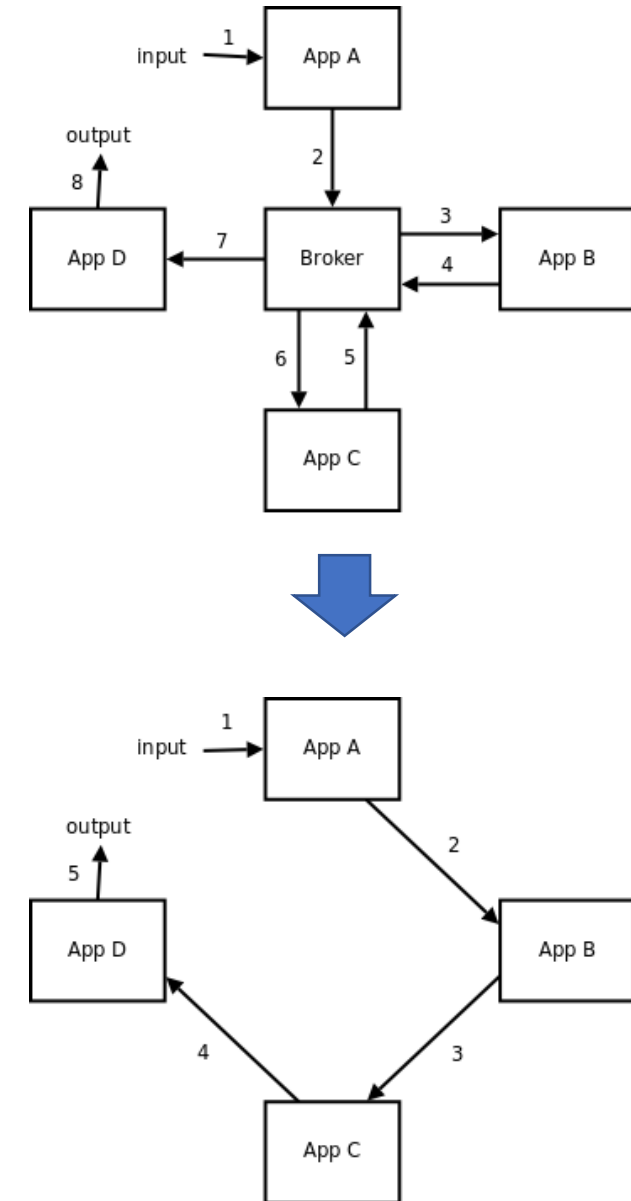
- Так как не всегда удастся сформировать семантику идемпотентных сообщений, иногда приходится журналировать сообщения и явно удалять дубликаты



# Асинхронные сообщения без брокера

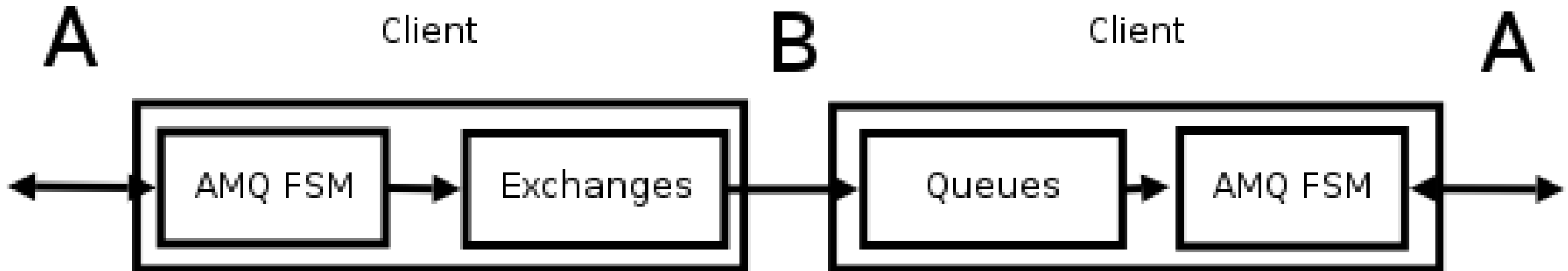
# Асинхронное взаимодействие без посредника (без брокера, Brokerless)

- Решение проблем (распределенного) брокера
  - Единая точка отказа
  - Дублирование
  - Потеря последовательности сообщений
- Улучшение производительности
- Бонусы асинхронной обработки
  - Сглаживание всплесков
- Компромиссы
  - Требуются механизмы балансировки и надежной доставки
  - Требуется служба обнаружения сервисов



# Без брокера: Реализация

- ZeroMQ

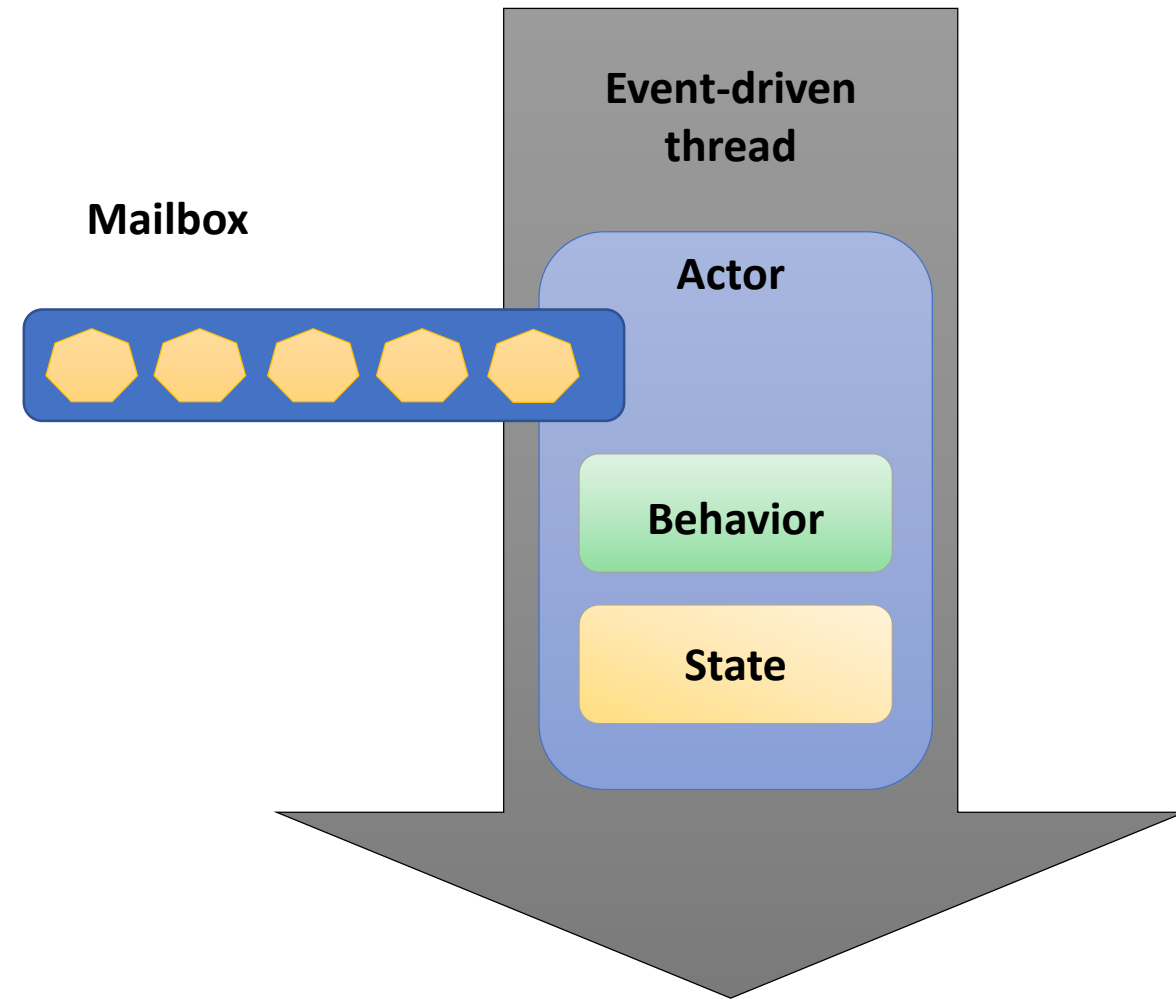


# Модель Акторов

- **Модель конкурентного доступа к ресурсам, в которой отсутствуют разделенные (share) состояния**
  - Ресурс закреплен за определенной компонентой (актором)
  - В каждый конкретный момент актор обрабатывает только одну задачу
  - Взаимодействие между акторами асинхронное и осуществляется посредством передачи сообщений

# Элементы модели Акторов

- Акторы (actor) общаются только посредством сообщений, которые выстраиваются в очередь (mailbox)
- В каждый конкретный момент времени к состоянию актора (state) имеет доступ только одна задача







Архитектурны стили,  
построенные на сообщениях

## Последовательность сообщений

**Request → Response → Request →**

**Команды/Запросы → События/Данные → Команды/Запросы →**

- **Обработчик команды может порождать события**
- **Обработчик событий может порождать команды**

## Два типа систем

- Ориентированные на сообщения (команды и запросы, message-driven)

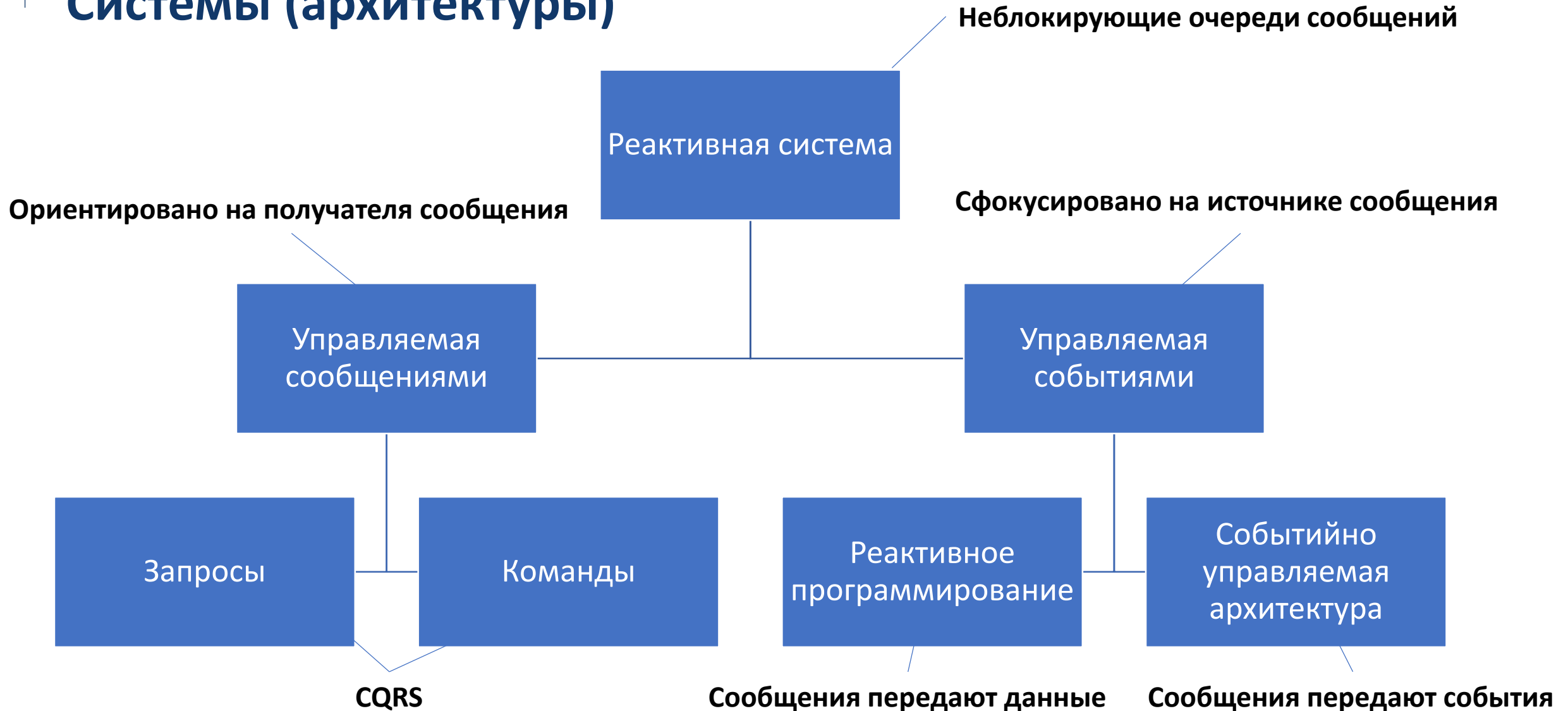


- требуется знание исполнителя команд и их синтаксиса (кто и что)
- Ориентированные на события (события и данные, event-driven)



- требуется знание типа и формата нужного события

# Системы (архитектуры)



# Реактивное программирование

- Компоненты всё время активны и всегда готовы получать сообщения
- Компоненты реагируют на события, т.е.

- реагируют на повышение нагрузки

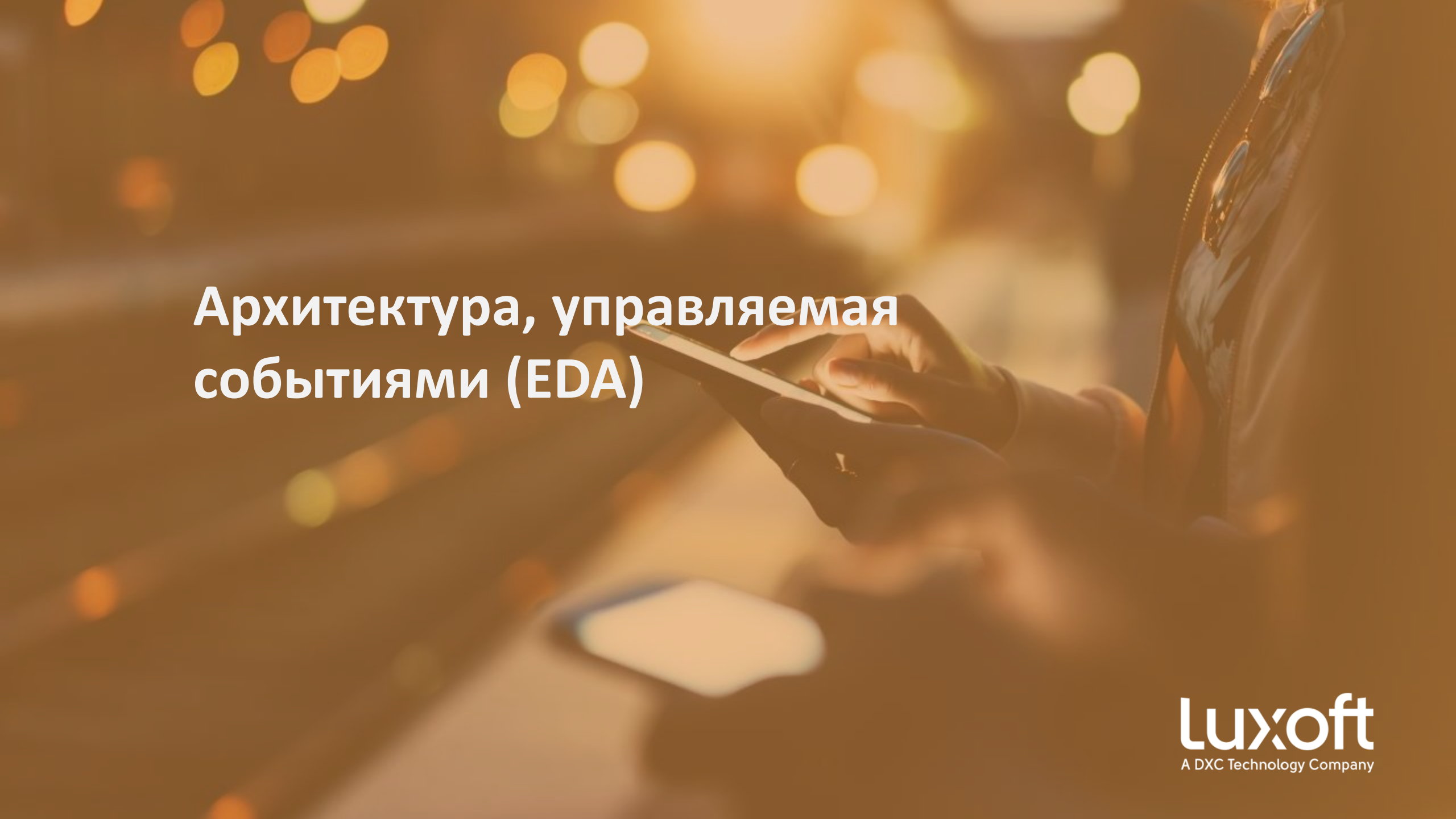
Фокус на масштабируемость, конкурентный доступ к общедоступным ресурсам сводится к минимуму

- реагируют на сбои

Строятся отказоустойчивые системы с возможностью восстанавливаться на всех уровнях

- реагируют на пользователей

Гарантированное время отклика, не зависящее от нагрузки



# Архитектура, управляемая событиями (EDA)

# Архитектурные стили при асинхронных взаимодействиях

## Message-Driven

- **Объекты передачи – сообщения (Message)**
  - Запросы (на получение данных)
  - Команды (на изменение данных)
  - Данные
- **Известен адресат передачи**

## Event-Driven

- **Объекты передачи – события (Event)**
- **Адресат передачи (наблюдатель) неизвестен ( $0 \div N$ )**

## **События (Domain Events)**

**Фиксируют (в памяти) любые значимые события, связанные с доменной моделью**

**Чаще всего возникают как реакция на изменение состояния (trigger)**

**События являются поводом для запуска некоторых процессов обработки**

**Могут быть сохранены (журналированы)**

**Чаще всего- любое изменение агрегата**



# Характеристики событий

- События всегда говорят о чем-то уже произошедшем  
“the speaker was booked,” “the seat was reserved,” “the cash was dispensed”
- Событие представляет собою однонаправленное сообщение (без ответа), которое может быть передано одним сервисом и считано несколькими получателями  
Pattern: one-way messages
- Обычно сообщение содержит параметры  
“Seat E23 was booked by Alice”
- Сообщение связано с задачами бизнеса

**Правильно:**

“Seat E23 was booked by Alice”

**Неправильно:**

“In the bookings table, the row with key E23 had the name field updated with the value Alice”

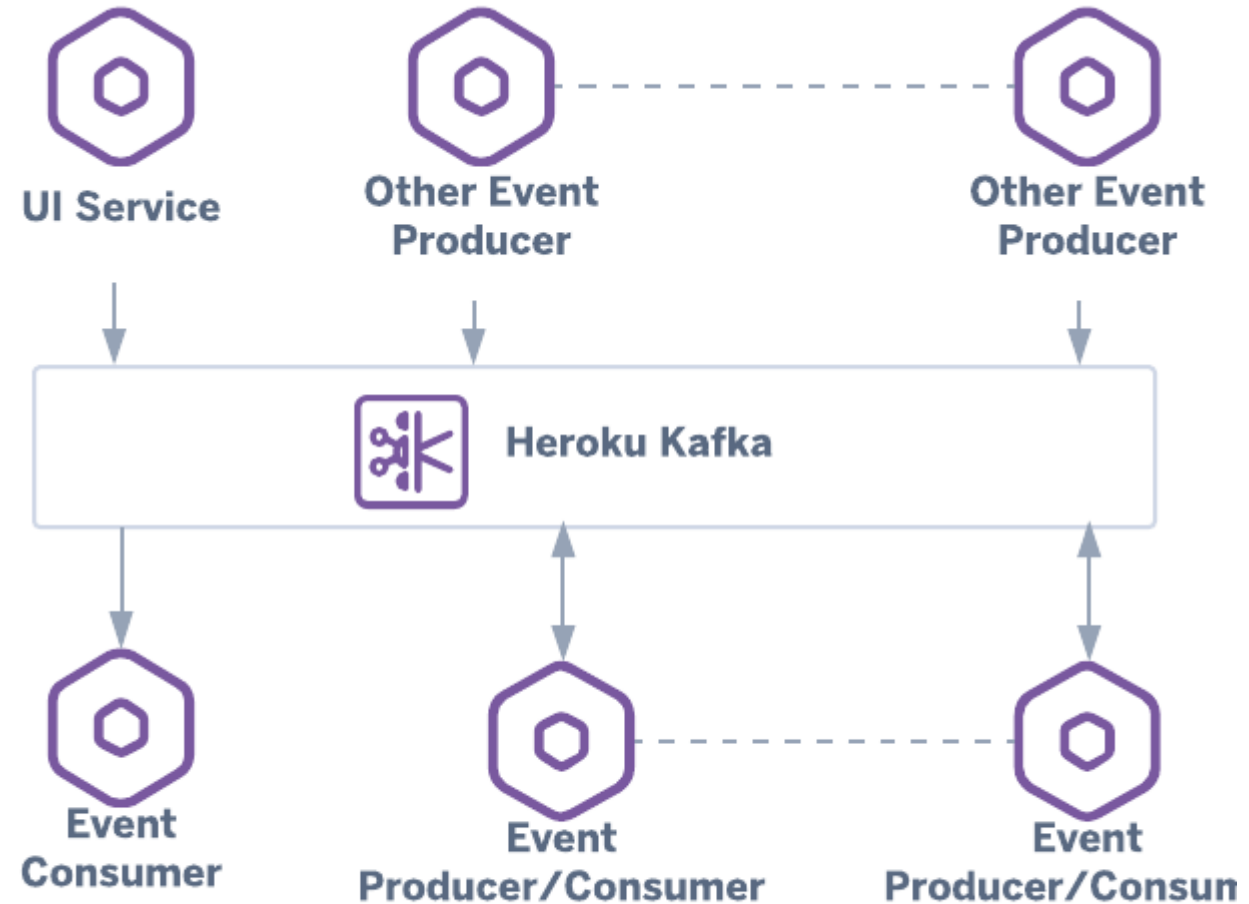
# Событийно-управляемая архитектура (EDA)

- Сервисы могут выступать в ролях поставщик (издатель) и потребитель (подписчик)
  - Поставщик – сервис, выполняющий некоторую работу и генерирующий событие, передаваемое в общий канал связи
  - Потребитель – сервис, прослушивающий события в общем канале связи и реагирующий на некоторые из них

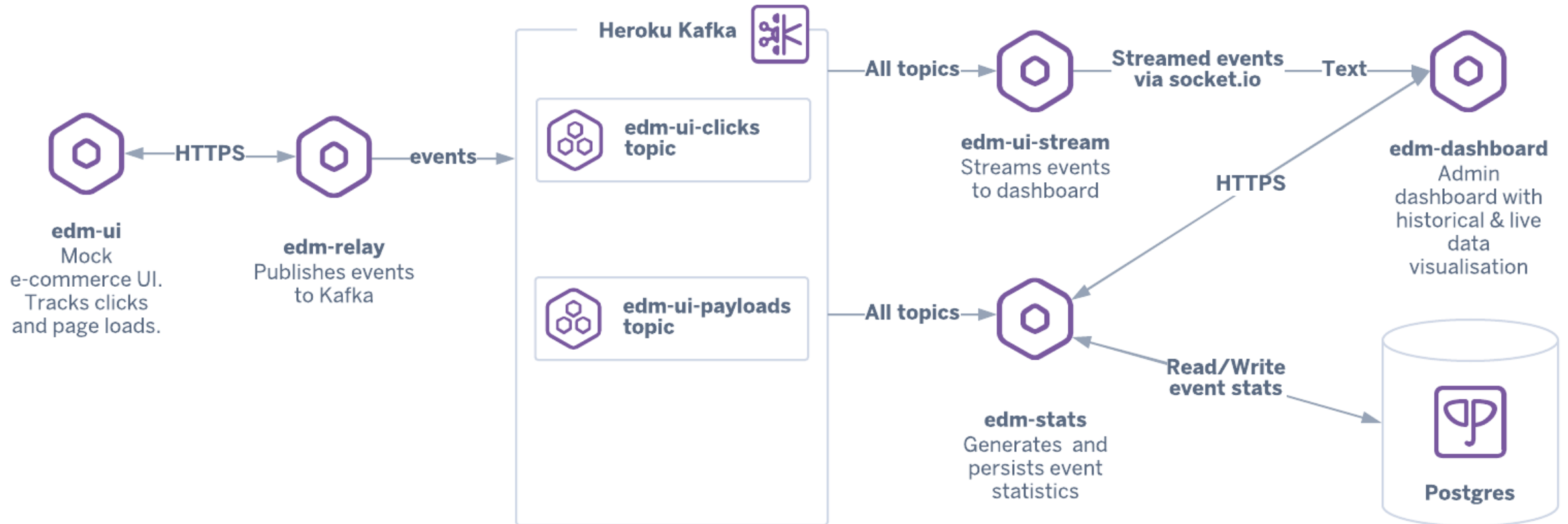
# Эталонная архитектура EDA

## ■ Сценарий

- У вас есть большое количество микросервисов, которые должны взаимодействовать асинхронно
- Вы хотите, чтобы ваши микросервисы были развязаны, взаимозаменяемы и поддерживались независимо
- У вас есть один или несколько сервисов, которые генерируют события, которые должны обрабатываться многими сервисами
- Вы хотите использовать шаблон связи микросервисов, который более слаб, чем типичный подход HTTPS



# Пример реализации EDA



# Преимущества и недостатки EDA

## ■ Преимущества

- Сервисы разделены, взаимозаменяемы и независимы.
- Сообщения буферизуются и используются, когда ресурсы доступны.
- Сервисы легко масштабируются для обработки больших объемов событий.
- Потребители/производители могут быть реализованы на разных языках.
- При необходимости восстановления после сбоя можно просто «переиграть» события.

## ■ Особенности

- Отказоустойчивость системы связана с отказоустойчивостью шины.
- Особое внимание к проблемам согласованности.

## ■ Недостатки

- Эта архитектура вносит некоторые эксплуатационные сложности.
- Обработка частичных сбоев может быть сложной.

# Проблемы EDA и подходы к их решению

- **Трудно реализовать транзакционные операции над распределенными реляционными данными**
  - **Сохраняем информацию в формате журнала событий (Event Sourcing)**
- **Трудно восстановить состояние объекта по набору событий**
  - **Отделяем представление состояний от оперативных данных (CQRS)**
- **Получение информации о доступных в системе событиях и их форматах**
- **Реакция на изменение событий**

# Анти-паттерны EDA (1 из 2)

- Слишком хорошо тоже не хорошо (Too Much of a Good Thing)
  - Не каждый метод должен порождать события  
Хорошей практикой является разделение внутренних событий системы и событий, переданных во внешнюю шину
- Обобщенные события (Generic Events)
  - Событие всегда должно иметь конкретную цель и конкретное имя
- Сложные графы зависимостей (Complex Dependency Graphs)
  - Следите за зависимостями между сервисами, даже если они не явные
- Сложные механизмы доставки (Depending on Guaranteed Order, Delivery, or Side Effects)
  - Включение в механизм обработки предположений о строгом порядке следования событий, отсутствии дубликатов или допущении сторонних эффектов сильно усложнит механизм доставки

## Анти-паттерны EDA (2 из 2)

- **Преждевременная оптимизация (Premature Optimization)**
  - Не решайте вопросы масштабирования и увеличения пропускной способности до тех пор пока не убедитесь в их необходимости
- **Управление событиями – серебряная пуля (Expecting Event-Driven to Fix Everything)**
  - Переход на EDA решает многие проблемы, но далеко не все, особенно, если они связаны с организацией процесса и зрелостью команды



# Доменные и интеграционные события

- **Доменные события**  
возникают в рамках одной транзакции в памяти  
одного процесса, то есть в пределах одного микросервиса
  - Посылаются на обработку в момент возникновения
- **Двухфазные доменные события (а-ля Джимми Богард)**
  - В рамках одного процесса накапливаются и затем одновременно передаются на обработку
- **Интеграционные события**
  - Передаются во внешний процесс после фиксации в базе данных

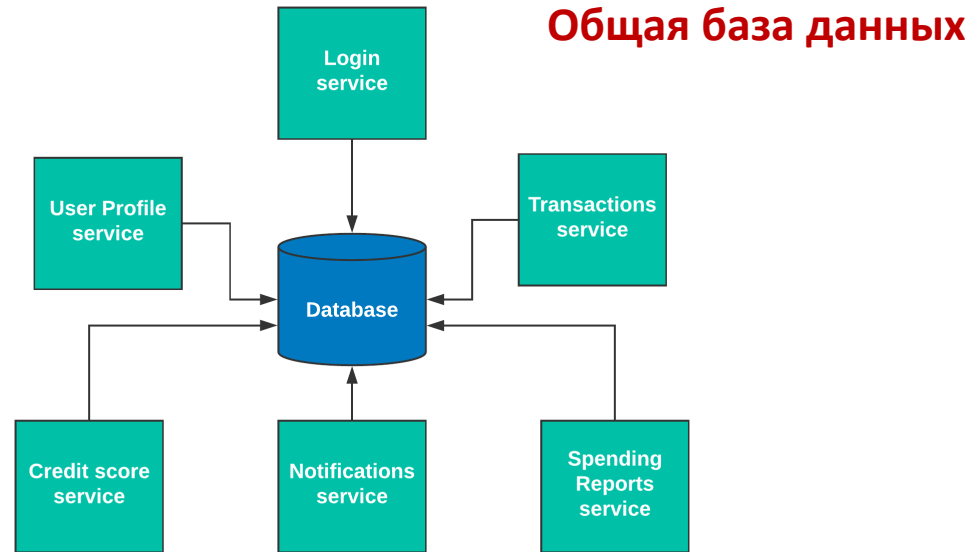


# Паттерны организации взаимодействия

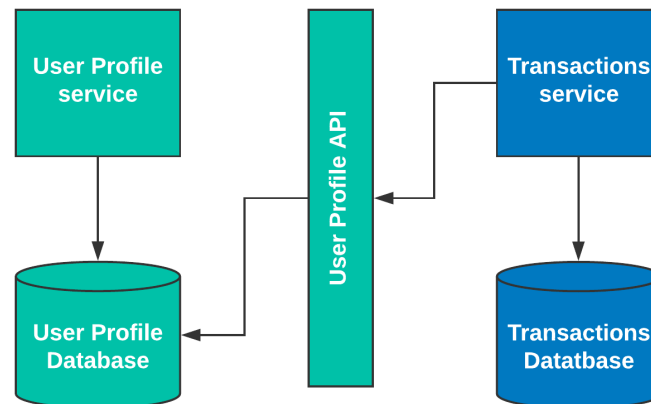
# Желаемые качества

- **Модифицируемость**
  - Изменение одного из сервисов должно по минимуму сказываться на другие и не влиять на работоспособность системы в целом
- **Возможность многократного повторного использования**
  - **Необходима унификация механизмов взаимодействия**
- **Производительность**
  - Переход к сетевым способам взаимодействия требует повышенного внимания к вопросам производительности
- **Надежность и отказоустойчивость**
  - Сбой одного из сервисов не должен препятствовать доставке сообщения
- **Безопасность**
  - Исключаем перехват и подмену сообщения

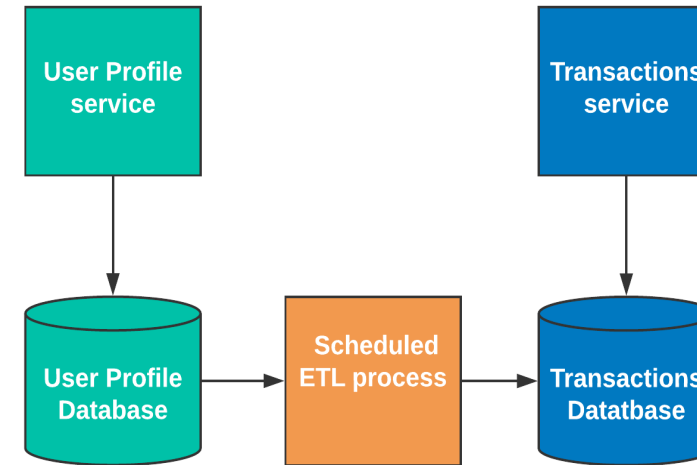
# Коннекторы



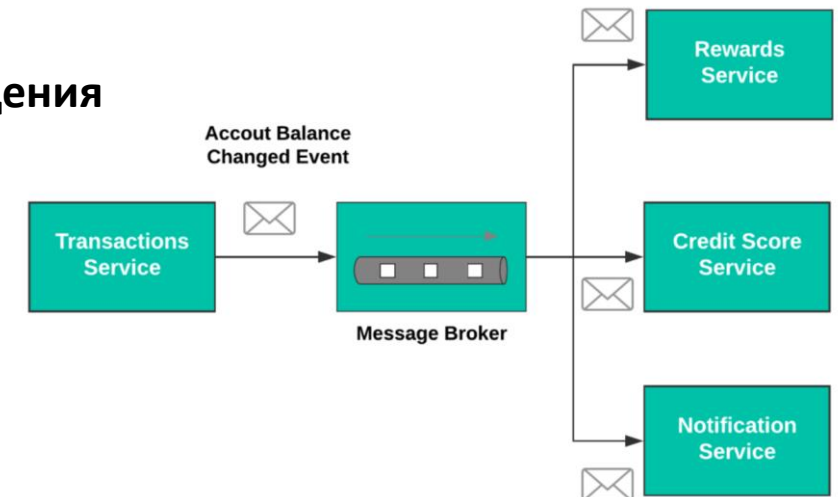
## Синхронный вызов



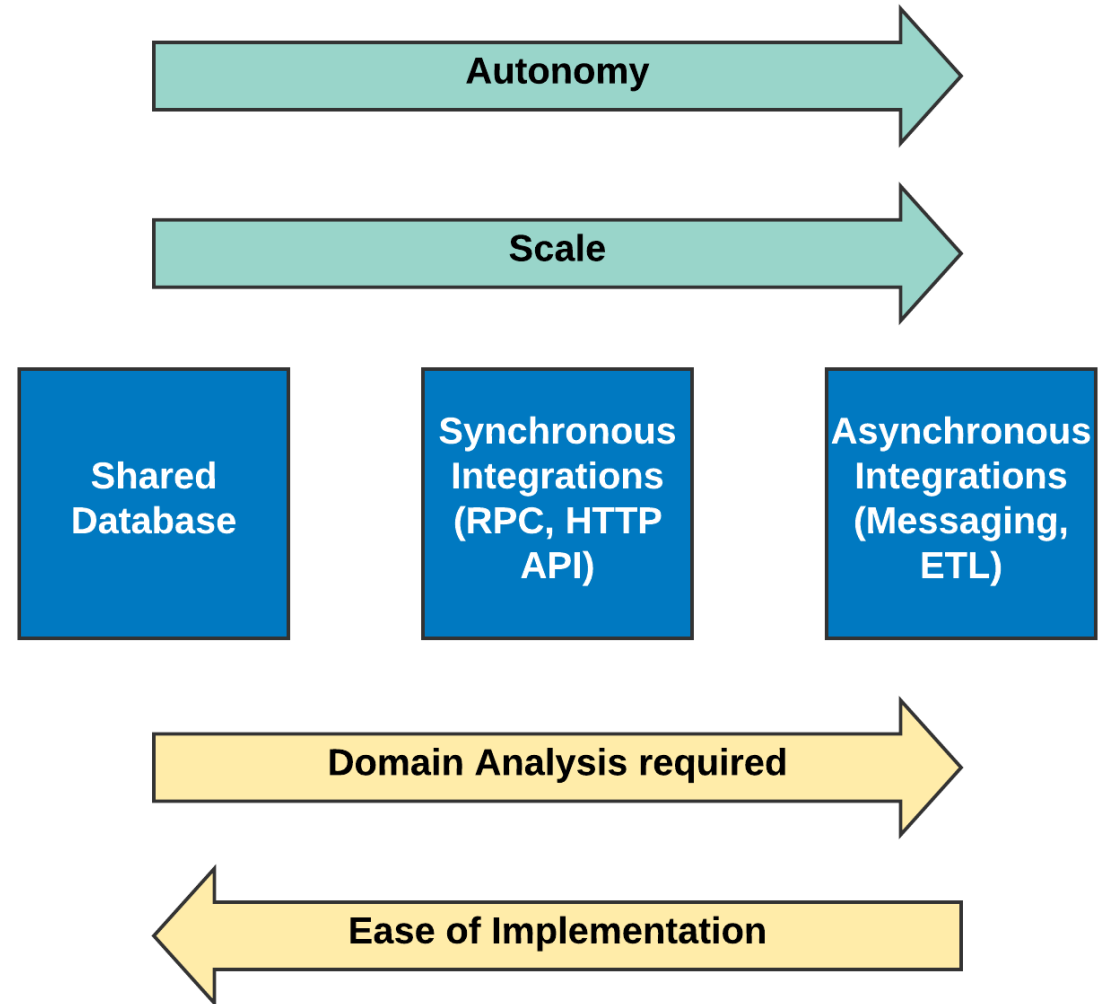
## Преобразование



## Сообщения



# Сравнение коннекторов



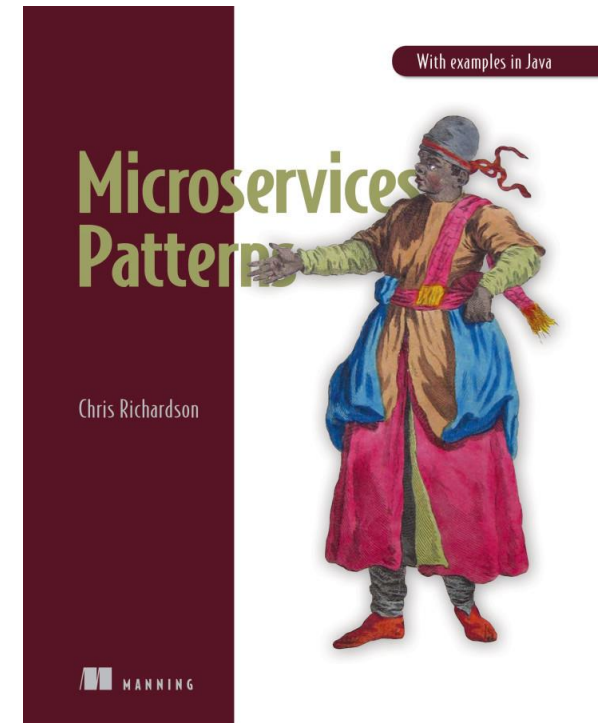
## Хорошая практика для MSA

- Для построения системы выбираем минимальное количество протоколов взаимодействия
  - по одному на каждый поддерживаемый тип коннектора

# Паттерны организации взаимодействия

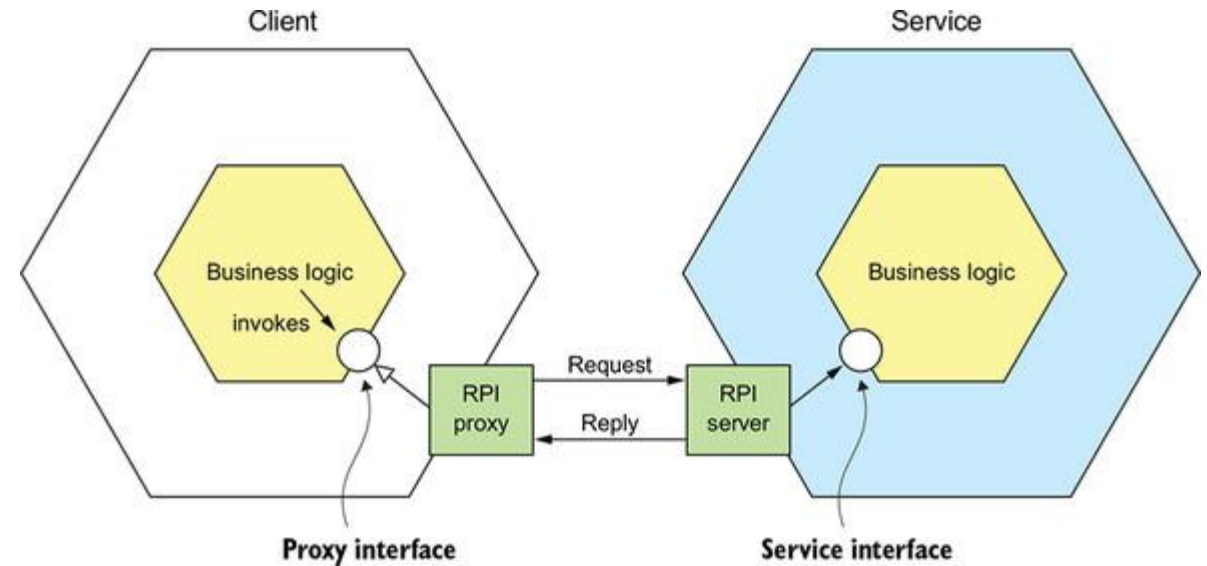
## Крис Ричардсон

- Удаленный вызов процедуры  
(Remote Procedure Invocation, RPI)
- Обмен сообщениями (Messaging)
- Специфичный для предметной области протокол  
(Domain-specific protocol)



# Удаленный вызов процедуры (Remote Procedure Invocation, RPI)

- Протокол в стиле «запрос-ответ»
- Распространенные решения
  - REST, gRPC, Thrift



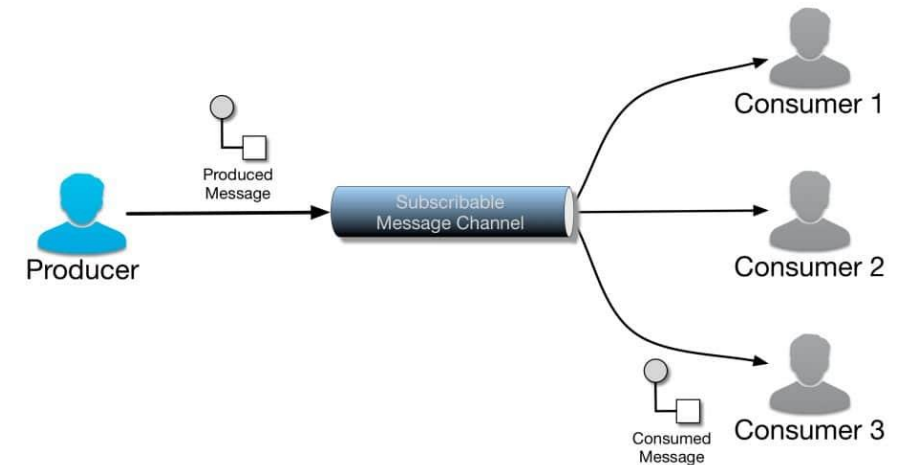
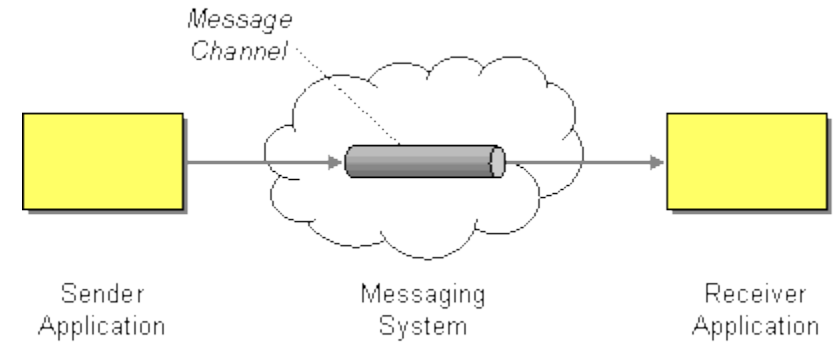


# Удаленный вызов процедуры: Результат

- **Преимущества**
  - Прост (знаком, простая схема, нет посредников)
- **Недостатки**
  - Хорошо поддерживает только протоколы «запрос-ответ» (плохо все асинхронные)
  - Клиент и сервис должны быть доступны на момент взаимодействия
- **Проблемы**
  - Клиент должен знать расположение сервиса

# Обмен сообщениями (Messaging)

- Обмен сообщениями через канал
- Варианты
  - Нотификация (One-way notifications)
  - Запрос – ответ (Request/Reply)
  - Издатель-подписчик (Publish/Subscribe)
- Распространенные решения
  - MQTT, AMQP (RabbitMQ)
  - Event Log (Apache Kafka)



# Обмен сообщениями: Результат

## ■ Преимущества

- Слабая связь во время выполнения, поскольку она разъединяет отправителя сообщения от потребителя
- Улучшенная доступность, поскольку брокер сообщений буферизует сообщения до тех пор, пока потребитель не сможет их обработать
- Поддерживает различные шаблоны связи

## ■ Недостатки

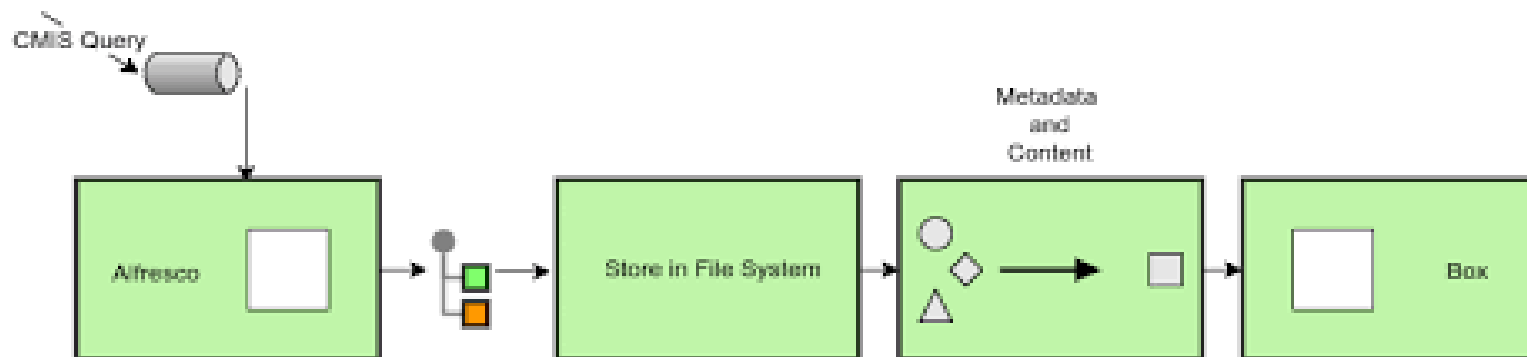
- Брокер сообщения становится возможной точкой отказа

## ■ Проблемы

- Реализация ответов (в частности при ошибках) – сложная задача

## Специфичный для предметной области протокол (Domain-specific protocol)

- Используем специфичный протокол
- Существует множество доменных протоколов, в том числе:
  - Протоколы электронной почты, такие как SMTP и IMAP
  - Протоколы потоковой передачи мультимедиа, такие как RTMP, HLS и HDS



# Протоколы удаленного вызова

# Основные протоколы удаленного вызова

- **REST**
- **RPC**
  - **gRPC**
  - **Apache Thrift**
- **Языки запросов**
  - **GraphQL**
  - **OData (поверх REST)**
  - **ORDS (поверх REST)**

# REST

**это архитектурный стиль, поддерживающий следующие ограничения:**

- Система должна быть разделена на клиенты и на сервера (Client-Server)
- Сервер не должен хранить какой-либо информации о клиентах. В запросе должна храниться вся необходимая информация для обработки запроса и, если необходимо, идентификации клиента (Stateless)
- Каждый ответ должен быть отмечен является ли он кэшируемым или нет (Cache)
- Поддерживается универсальный интерфейс между компонентами системы (Uniform Interface)
- Допускается разделить систему на иерархию слоев, но с условием, что каждый компонент может видеть компоненты только непосредственно следующего слоя (Layered System)
- Позволяется загрузка и выполнение кода или программы на стороне клиента (Code-On-Demand) опционально

# Универсальный интерфейс (Uniform Interface)

Для получения универсального интерфейса вводятся следующие ограничения:

- В REST ресурсом является все то, чему можно дать имя.  
Каждый ресурс в REST должен быть идентифицирован посредством стабильного идентификатора, который не меняется при изменении состояния ресурса.  
В случае HTTP идентификатором является URI (Identification of resources).
- Представление в REST используется для выполнения действий над ресурсами.  
Представление ресурса представляет собой текущее или желаемое состояние ресурса (Manipulation of resources through representations).
- Запрос и ответ должны хранить в себе всю необходимую информацию для их обработки.  
Не должны быть дополнительные сообщения или кэши для обработки одного запроса (Self-descriptive messages).
- Для навигации по API должен быть использован гипертекст (hypermedia as the engine of application state, HATEOAS).

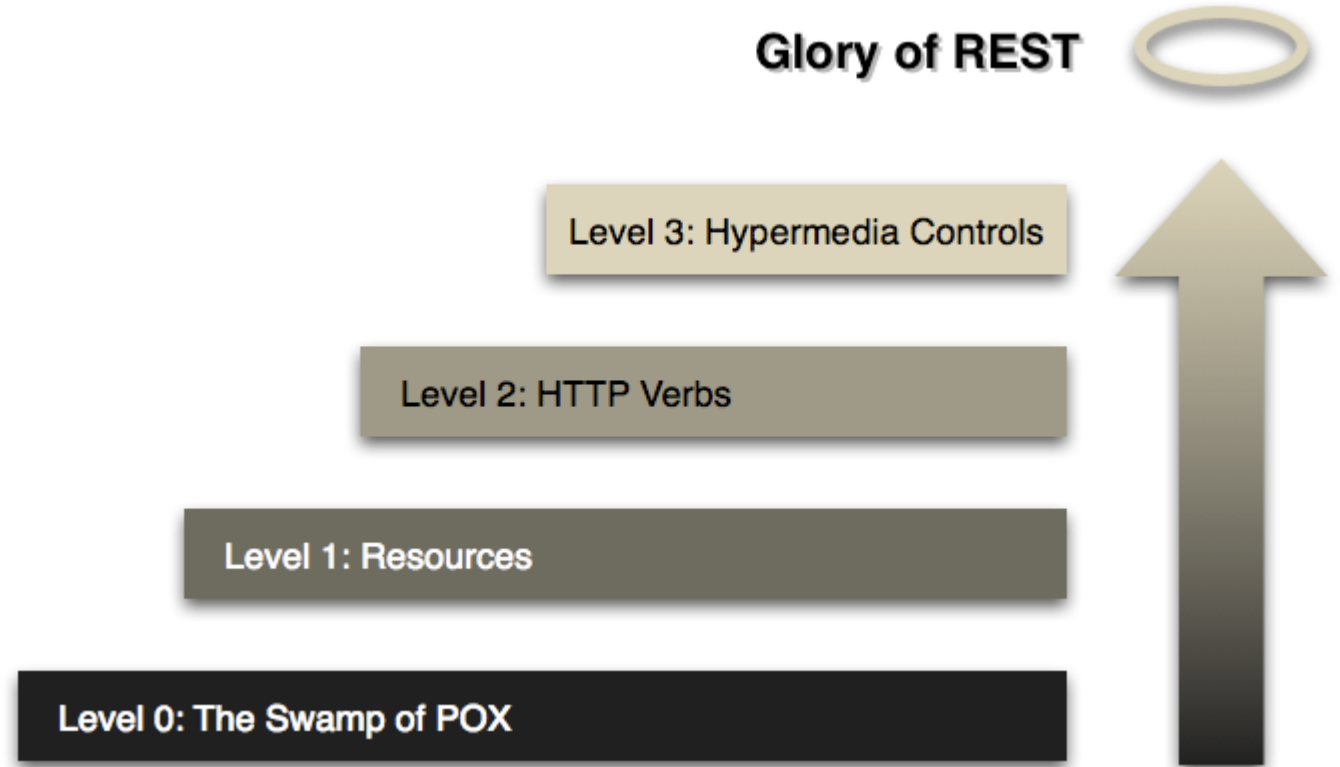


# Преимущества REST

- **Архитектура REST позволяет построенным на базе ее протоколам (RESTful протоколам) не зависеть от сигнатуры вызываемых методов**
  - **Зависимость между клиентом и сервисов возникает только по передаваемым данным**
- **Простота**
  - **Семантическая простота**
  - **Отсутствие оберток для данных (данные передаются как есть)**
  - **Простота реализации**
- **Поддержка сообществом**
  - **Большое количество реализаций**

# Модель зрелости REST (Richardson Maturity Model)

- Свобода реализации протоколов по REST архитектуре привела к большому разнообразию решений
- Ричардсон упорядочил реализации введением модели зрелости (степень RESTFul)



Leonard Richardson

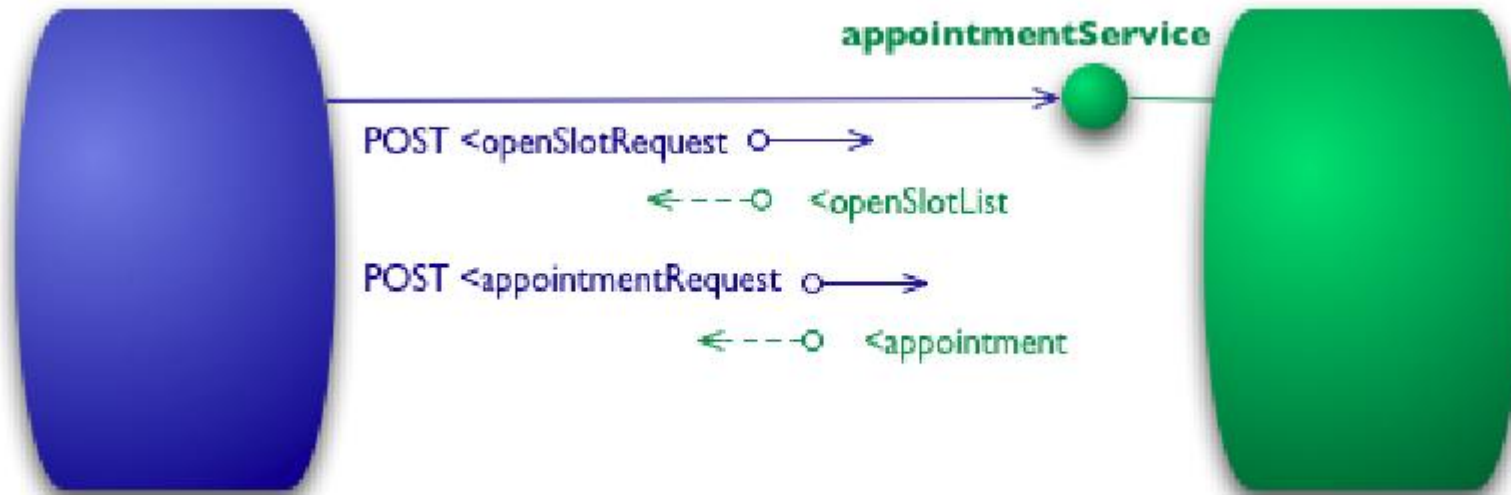


## Модель зрелости REST (Level 0)

- Один URI, один HTTP метод
- HTTP используется для взаимодействия компонентов распределенной системы

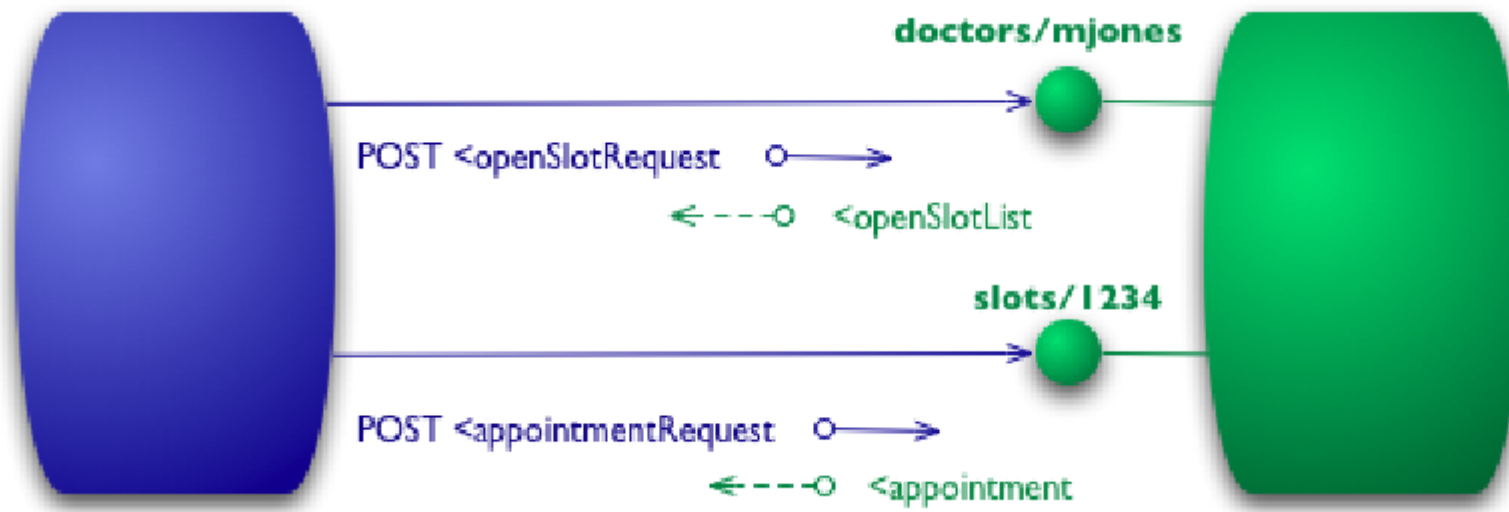
Из методов используется только один, например POST

Примеры: XML-RPC, SOAP



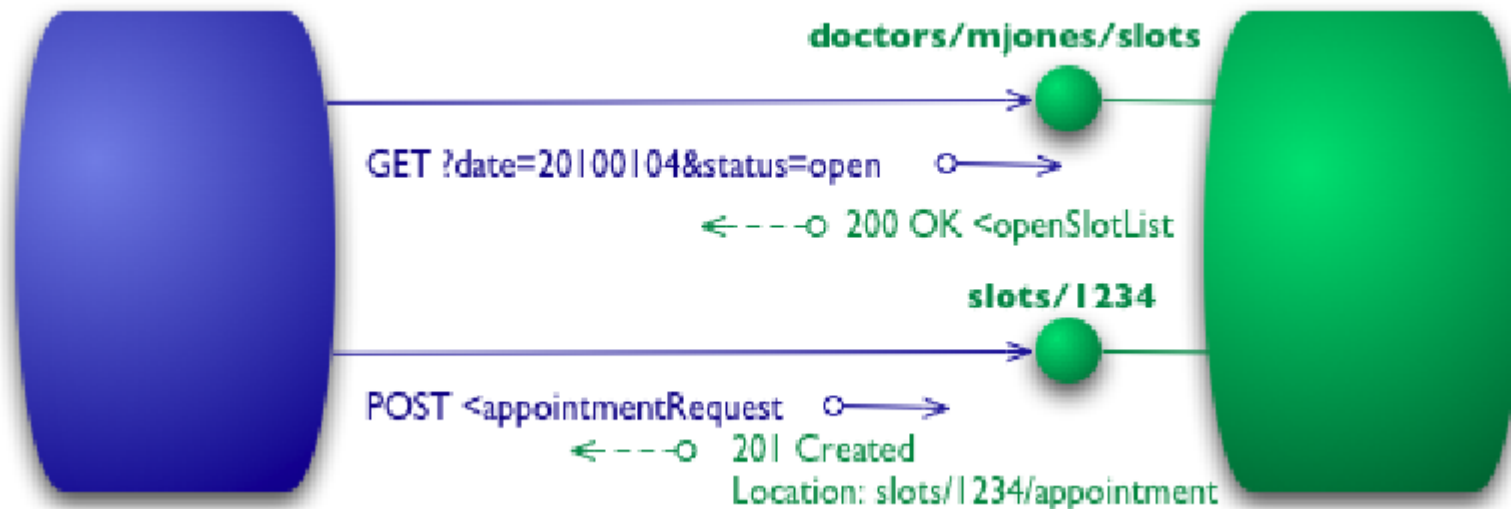
# Модель зрелости REST (Level 1)

- Несколько URI, один HTTP метод
- В службе вводится понятие ресурсов и для действия с конкретным ресурсом используется URL этого ресурса



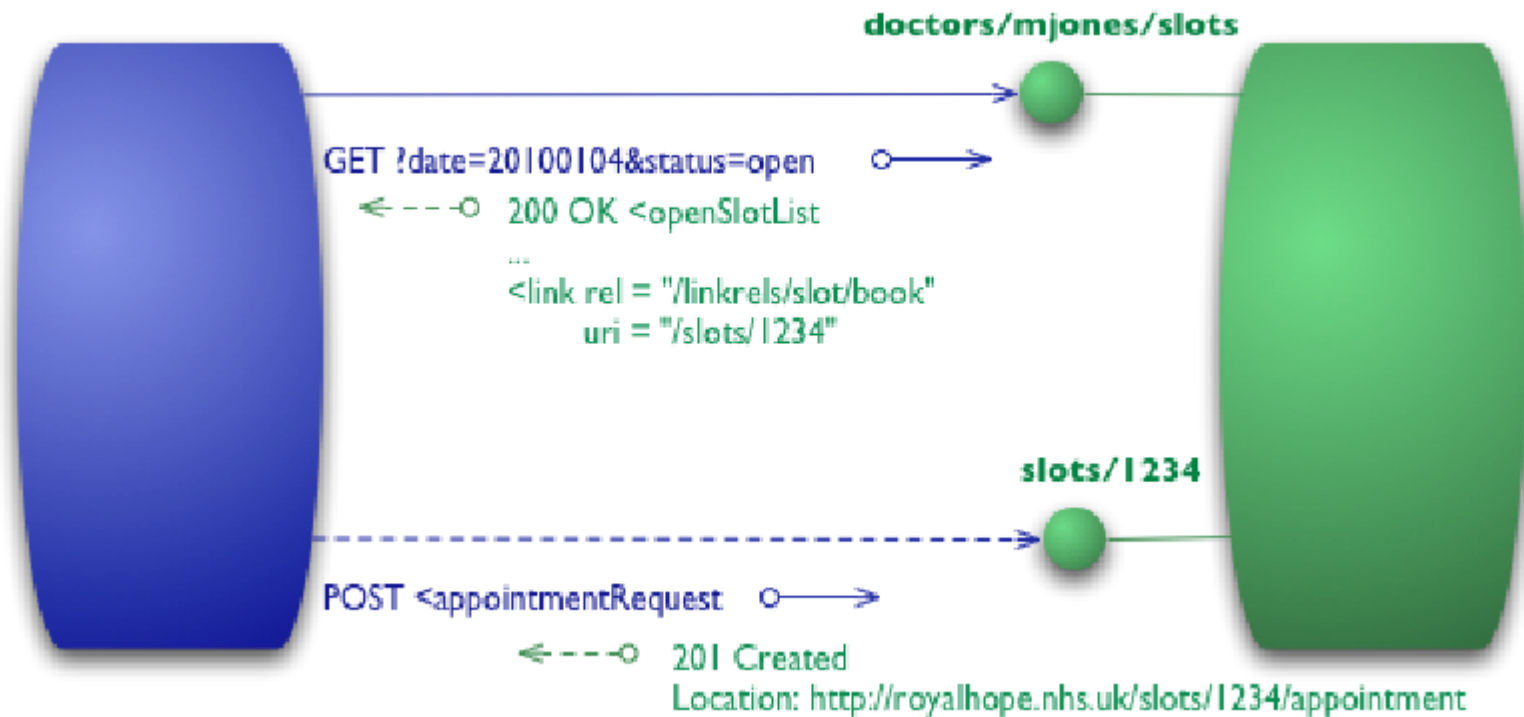
## Модель зрелости REST (Level 2)

- Несколько URI, каждый поддерживает разные HTTP методы
- Используются разные методы взаимодействия с ресурсом и разные статусы для возврата сообщений



## Модель зрелости REST (Level 3)

- HATEOAS. Ресурсы сами описывают свои возможности и взаимосвязи
- Сервис возвращает список допустимых действий
- Появляется возможность менять URI независимо от клиентов



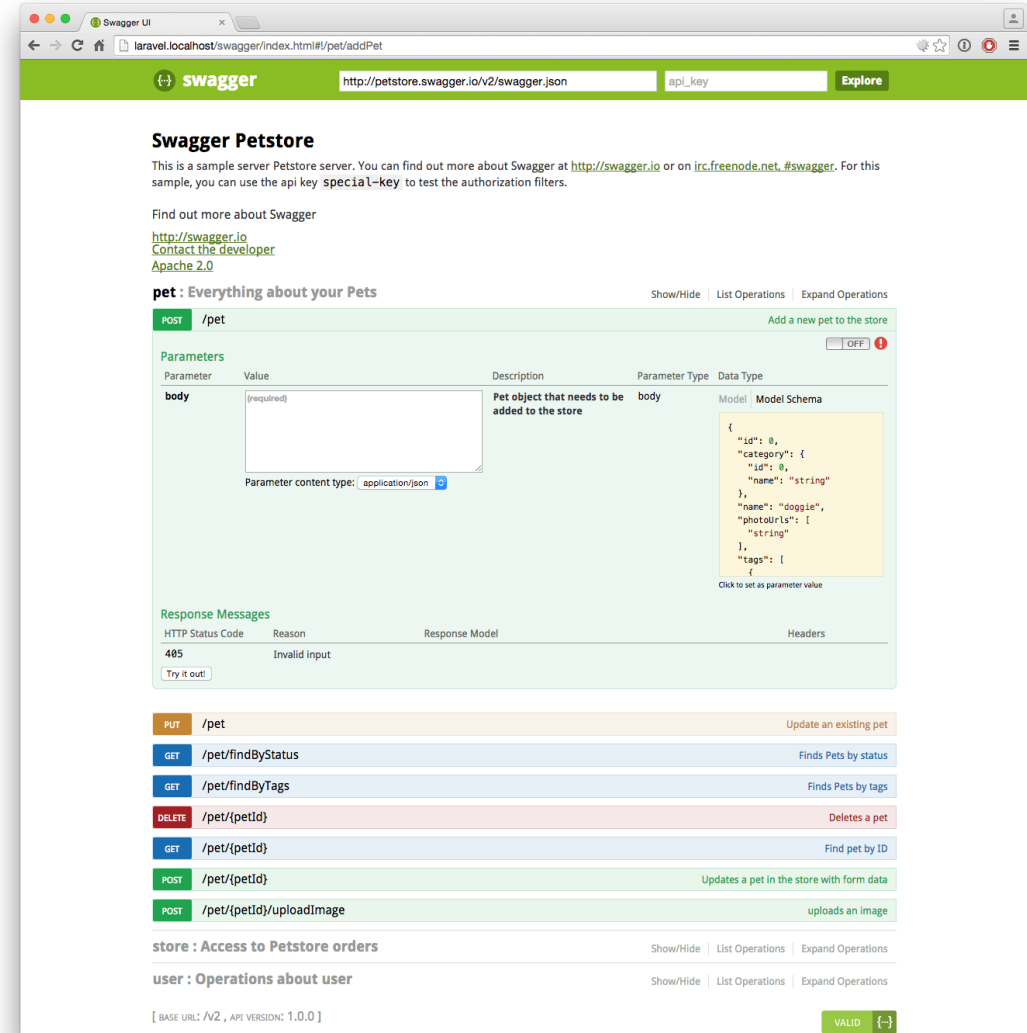
# Проектирование RESTful API

## ■ Подходы

- Вначале API (API First)
- Вначале код (Code First)
- Вначале контракт (Contract First)

## ■ Инструменты

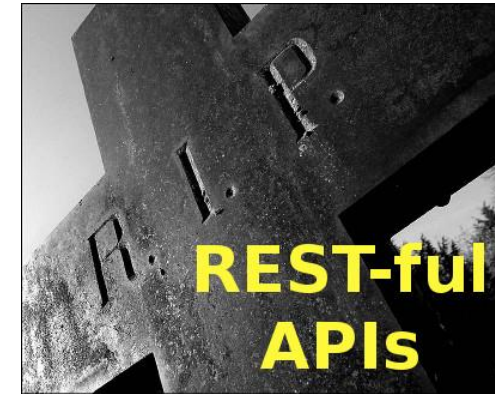
- Swagger
- RAML



# Недостатки REST

## RESTful API довольно ужасно

- До сих пор нет общего согласования того что такое RESTful API
- Словарь REST поддерживается не полностью
  - Get не поддерживает Body
  - Решение – глагол в URL
- Словарь REST недостаточно насыщен
  - Три разнородных словаря (Методы – имена: GET, POST..., Статусы – числа: 200, 404..., данные – JSON)
- Трудно отлаживать
  - Нам приходится просматривать сразу 7 мест: Метод HTTP запроса, Адрес запроса, *Метод, который мы на самом деле подразумеваем (в теле запроса)*, Собственно, тело запроса, Код ответа, *Код, который мы подразумевали (в теле ответа)*, Собственно, тело ответа



Michael S. Mikowski



# Сложности проектирования REST API

- Работая с единичными ресурсами трудно осуществлять групповые операции
- Работая с единичными ресурсами трудно осуществлять транзакционные операции
- Трудности зачастую преодолеваются грамотным проектированием
  - Например, выделение агрегата (DDD) позволяет осуществлять транзакции
- Требуется серьезной культуры проектирования



- Независящая от языка высокопроизводительная платформа удаленного вызова процедур (RPC), работающая поверх HTTP/2
- Разработчик – Google
- Преимущества
  - Имеет встроенную поддержку для балансировки нагрузки, трассировки, аутентификации и проверки жизнеспособности сервисов
  - Есть возможность создавать клиентские библиотеки для работы с бэкендом на 10 языках
  - Высокая производительность достигается за счет использования протокола HTTP/2 и Protocol Buffers

# Типы gRPC

- **Унарный (Unary RPC)**

Синхронный запрос клиента, который блокируется пока не будет получен ответ от сервера

- **Серверный стрим (Server streaming RPC)**

При подключении клиента сервер открывает стрим и начинает отправлять сообщения

- **Клиентский стрим (Client streaming RPC)**

То же самое, что и серверный, только клиент начинает стримить сообщения на сервер

- **Двунаправленный стрим (Bidirectional streaming)**

Клиент инициализирует соединение, создаются два стрима

Сервер может отправить изначальные данные при подключении или отвечать на каждый запрос клиента по типу “пинг-понга”

# Проектирование gRPC

- Сначала проектирование (Design first) + кодогенерация

```
syntax = "proto3";

package todos;

message Todo {
    string content = 1;
    bool finished = 2;
}

message GetTodoRequest {
    int32 id = 1;
}

service TodoService {
    rpc GetTodoById (GetTodoRequest) returns (Todo);
}
```

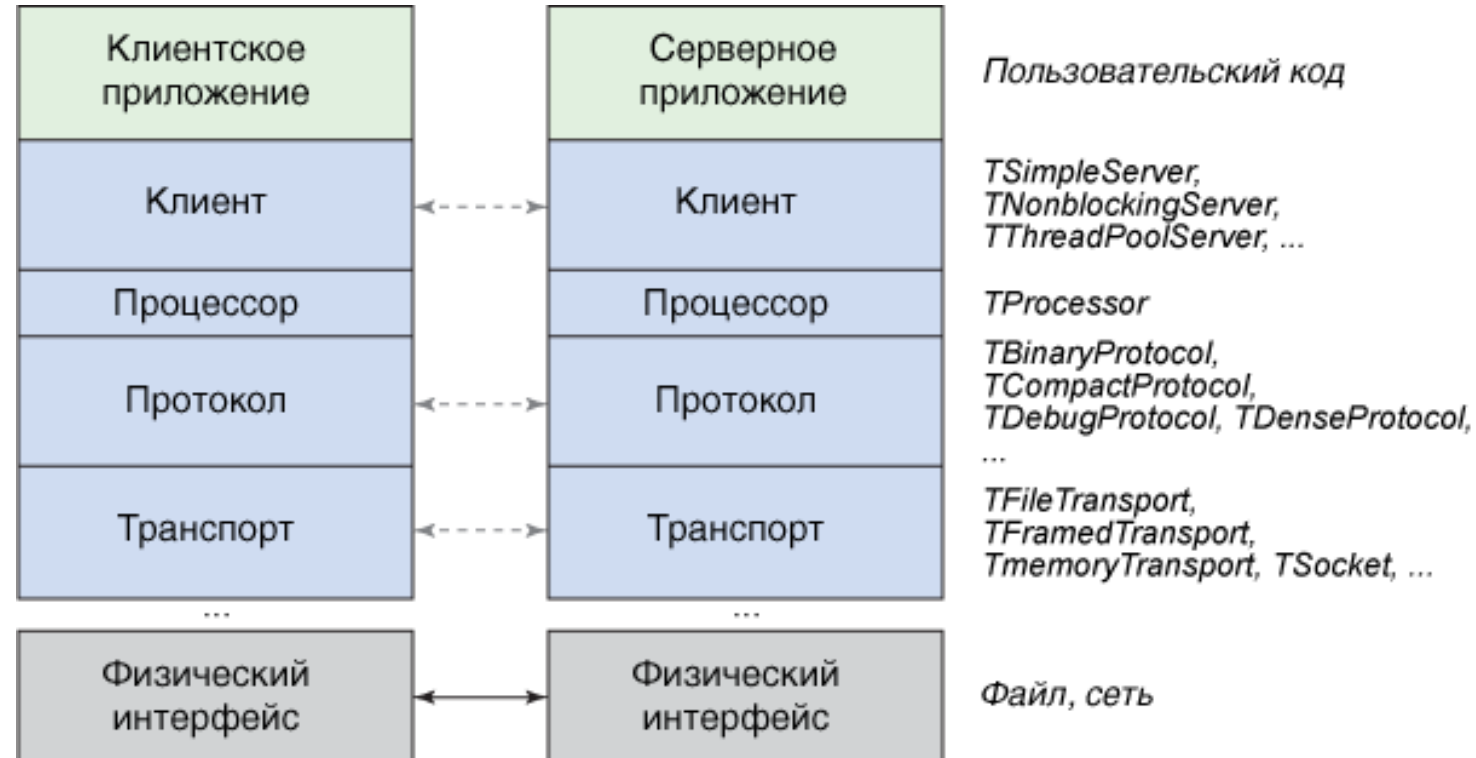
# Apache Thrift



**Thrift — это инфраструктура для масштабируемой кросс-языковой разработки сервисов, которая не только поддерживает генерацию кода на множестве языков, но и предоставляет стек программного обеспечения, упрощающий разработку сетевых сервисов**

# Архитектура Thrift

- Apache Thrift реализует стек программного обеспечения, который упрощает разработку взаимодействующих многоязыковых приложений



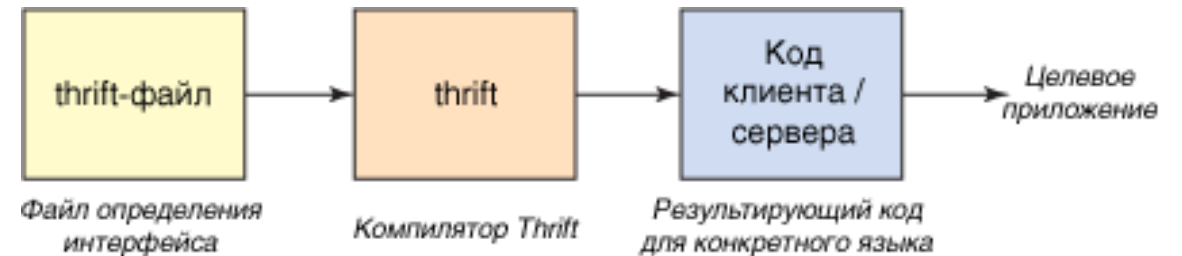
# Проектирование Thrift

- Первый шаг состоит в формировании файла с определением интерфейса
- Данный файл позволяет определять типы и сервисы, которые вы собираетесь представлять
- Этот файл не зависит от языка и использует типы и определения Thrift
- С помощью этого файла Thrift-компилятор генерирует исходные файлы (как для клиента, так и для сервера) на выбранном вами языке

```

1  # proj.thrift
2
3  namespace py demosever
4  namespace rb demosever
5
6  /* Каждый операнд является 32-разрядным целым числом и имеет имя Value */
7  typedef i32 Value
8  typedef i32 Result
9
10 /* Сервис MyMath представляет определенную математическую функцию */
11 service MyMath
12 {
13     Result add( 1: Value op1, 2: Value op2 ),
14     Result mul( 1: Value op1, 2: Value op2 ),
15     Result min( 1: Value op1, 2: Value op2 ),
16     Result max( 1: Value op1, 2: Value op2 )
17 }

```



## Отличие gRPC от Thrift

- Встроенная поддержка потокового RPC (в частности чанкинг)
- API перехватчика

Предоставляет мощный способ добавления общих функций к нескольким конечным точкам обслуживания

- Встроенное управление потоком и поддержка TLS
- Отличная поддержка сообщества



# Сравнение REST и RPC

## ■ REST

- Широко распространенный стандарт, который можно использовать без дополнительной поддержки
- Если у вас есть разумно определенный API, команды могут написать свои собственные клиентские библиотеки для связи с ним
- REST хорошо совместим, поскольку определение API не зависит от языка и реализации

## ■ RPC (gRPC/Thrift)

- Обычно предлагает лучшую производительность, поскольку обычно обрабатывает сериализацию с использованием двоичных форматов
- С помощью RPC вы можете избежать анализа ответов JSON или извлечения параметров пути запроса
- RPC также не зависит от транспорта, и вы можете использовать альтернативный транспортный механизм, такой как очередь сообщений

# GraphQL (Убийца REST API)

**язык запросов, который описывает как получить данные с сервера, и используется в API для загрузки данных с сервера на клиент**

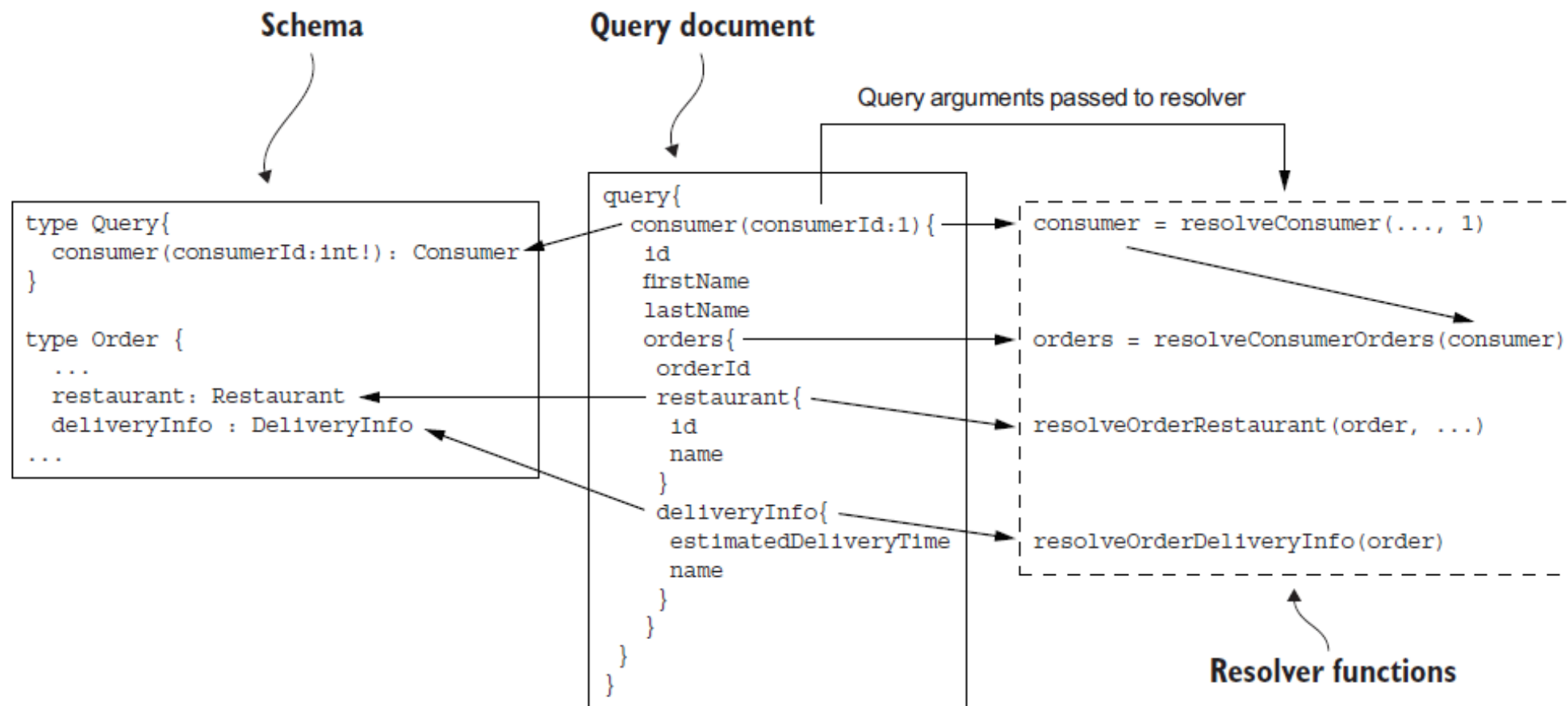
- **Слой GraphQL живет между клиентом и одним или несколькими источниками данных, принимает клиентские запросы и извлекает необходимые данные в соответствии с требованиями клиента**
- **Преимущество**
  - **Получать именно ту информацию, которая востребована клиентом**

# GraphQL: Строительные блоки

- **схемы (schema)**
  - описание объектов доступных через API
- **запросы (queries)**
  - запросы к серверу
- **резолверы (resolvers)**
  - описывает, как и где получить данные для конкретного поля объекта
  - может описывать как изменить значение поля (мутациям, mutation)

```
query {  
  posts { # массив  
    id  
    title  
    text  
    author { # вложенная сущность  
      id  
      name  
    }  
  }  
}
```

# GraphQL: Структура



# Сравнение GraphQL и REST

- С помощью GraphQL легче формировать запросы к данным, но
- REST может делать тоже самое, что и GraphQL
  - Типизация – JSON-схемы
  - Гибкий язык запросов – OData, ORDS
- GraphQL делает некоторые задания более сложными
  - Например, требуется парсить код ответа
- Легче использовать веб-кэш в сочетании с REST чем с GraphQL.
  - REST реализует кэш на уровне HTTP
- У вас могут быть проблемы с производительностью с GraphQL-запросами
  - Вы позволяете клиентам выполнить любой запрос, который они хотят
- Способ работы GraphQL схем может быть проблемой
  - Схемы поддерживают статическую типизацию и не могут быть изменены в процессе выполнения

# OData

- Язык запросов поверх REST протокола
- Разработан Microsoft в 2007 году
- OData является стандартным REST API OASIS и используется такими компаниями, как Microsoft, SAP, CA, IBM и Salesforce
- OData предоставляет богатый набор возможностей и быстро завоевывает популярность благодаря открытому исходному коду, а также исключительной масштабируемости

OData - orderBy

- HTTP GET serviceRoot/OPPORTUNITIES?\$orderBy=name

```
{
  "@odata.context": "serviceRoot/$metadata#OPPORTUNITIES",
  "value": [
    {
      "NAME": "Batman",
      "AMOUNT": 10000,
      "ACCOUNTID": 123
    },
    {
      "NAME": "Superman",
      "AMOUNT": 8500,
      "ACCOUNTID": 123
    }
  ]
}
```

# ORDS (Oracle REST Data Services)

- Служба Oracle REST
- Позволяет разработчикам, обладающим навыками SQL и другими навыками работы с базами данных, создавать API доступа к данным корпоративного класса к базе данных Oracle
- Шестьдесят групп в Oracle используют ORDS, включая Oracle Database, Times Ten и NoSQL

## Использование синхронных протоколов (Рекомендации)

- **REST – используем в общем случае**
- **RPC (gRPC) – используем при проблемах с производительностью**
  - При этом стоимость использования Protobufs очень высока
    - Сообщения нечитаемы и сложны в отладке
    - Строгая типизация негибка (только примитивы и перечисления, структура, а не объект)
    - Обратная совместимость требует помечать новые поля как необязательные
- **OData – используем для организации работы клиента, которому требуется гибкий механизм доступа к данным**
  - И в этом случае предпочитаем вынести API со слоя микросервисов на слой API Gateway (BFF)



# Проблема совместимости

- Для поддержания совместимости клиента и сервиса используем следующие подходы
  - семантическое **версионирование** (Semantic Versioning)  
MAJOR.MINOR.PATCH (2.0.0)
    - При смене MAJOR версии предполагается возможность изменений, ломающих код клиента
    - Смена MINOR добавляет новые возможности не меняя интерфейсы
    - PATCH сигнализирует об исправлении дефектов
  - Обратная совместимость (backward-compatible changes)
    - Сервис поддерживает несколько предыдущих версий системы

# Версионирование REST интерфейсов

- Версионируется **формат передаваемых данных**
- Добавление версии в URL – **анти-паттерн**
  - <http://example.com/v1/users>
- Хорошей практикой является добавление версии в заголовок запроса

```
// как параметр
Асcept: application/json; version=v1
// как тип содержимого, определенный поставщиком API
Асcept: application/vnd.company.myapp-v1+json
```

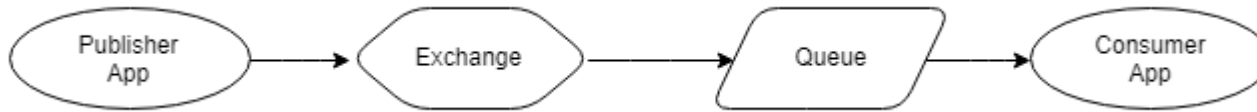
# Протоколы обмена сообщениями

# Основные протоколы обмена сообщениями

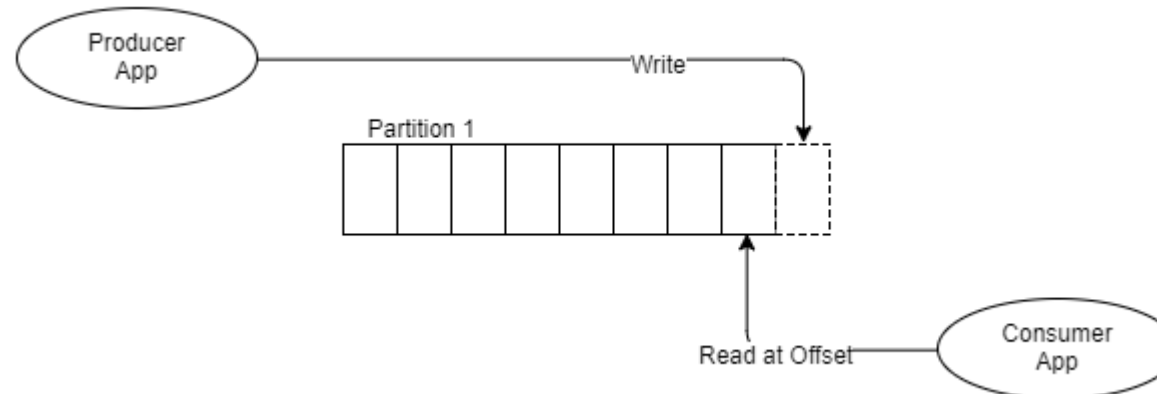
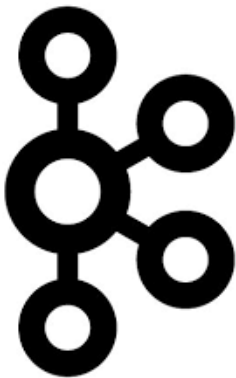
- **Очередь сообщений (Message Queue)**
  - MQTT
  - AMQP
  - STOMP
- **Журнал событий (Events Log)**
  - Apache Kafka
  - Apache Pulsar

# Стратегии доставки (Pull vs Push)

Сообщения могут проталкиваться (Push)



либо выгружаться (Pull) по запросу



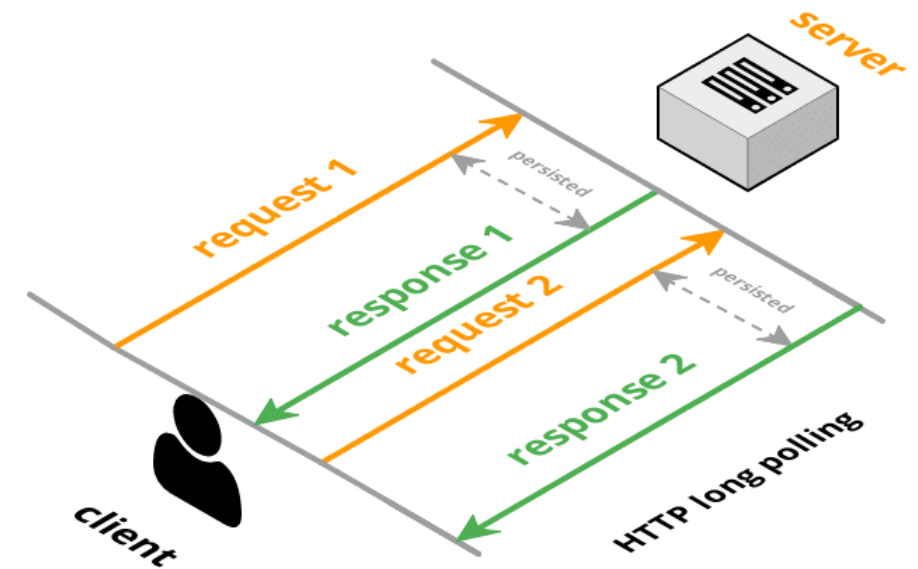
# Сравнение стратегий доставки (Pull vs Push)

- **Преимущества проталкивания (Push)**
  - **Меньше время задержки**
  - **Равномерное распределение задач (по одной)**
- **Преимущество выгрузки (Pull)**
  - **Нет перегрузок на клиенте**
  - **Гарантия соблюдения порядка обработки сообщений**
  - **Удобное пакетирование**

# MQTT (MQ Telemetry Transport)

протокол обмена сообщениями по шаблону издатель-подписчик (pub/sub)

- способ поддержания связи между машинами в сетях с ограниченной пропускной способностью или непредсказуемой связью
- протокол сделан маленьким и лёгким
- Пример брокера
  - Eclipse Mosquitto



# AMQP (Advanced Message Queueing Protocol)

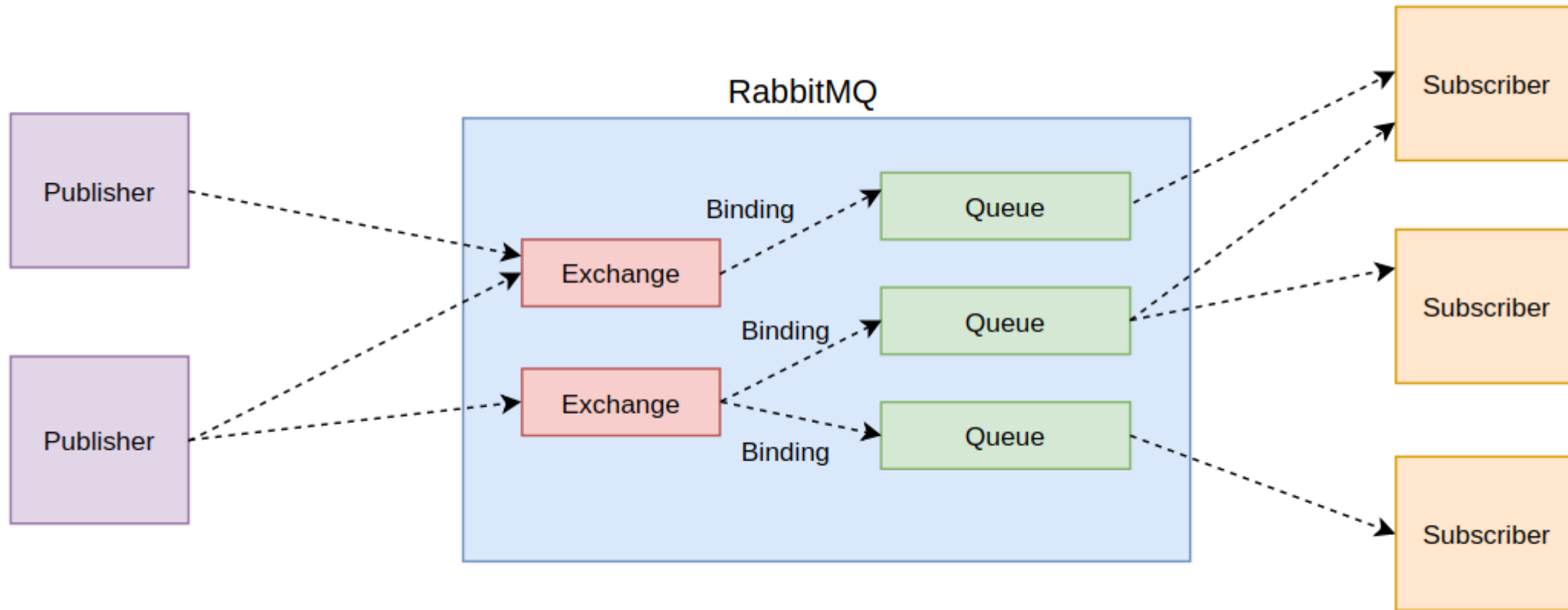
**открытый протокол для передачи сообщений между компонентами системы с низкой задержкой и на высокой скорости**

- **Семантика обмена сообщениями настраивается под нужды конкретного проекта**
  - **подразумевает как способ организации сетей, так и работу брокеров сообщений**
- **Первый стандарт, для которого существует большое количество свободных реализаций**
  - **Примеры брокеров**
    - **RabbitMQ**
    - **Apache ActiveMQ**



# RabbitMQ

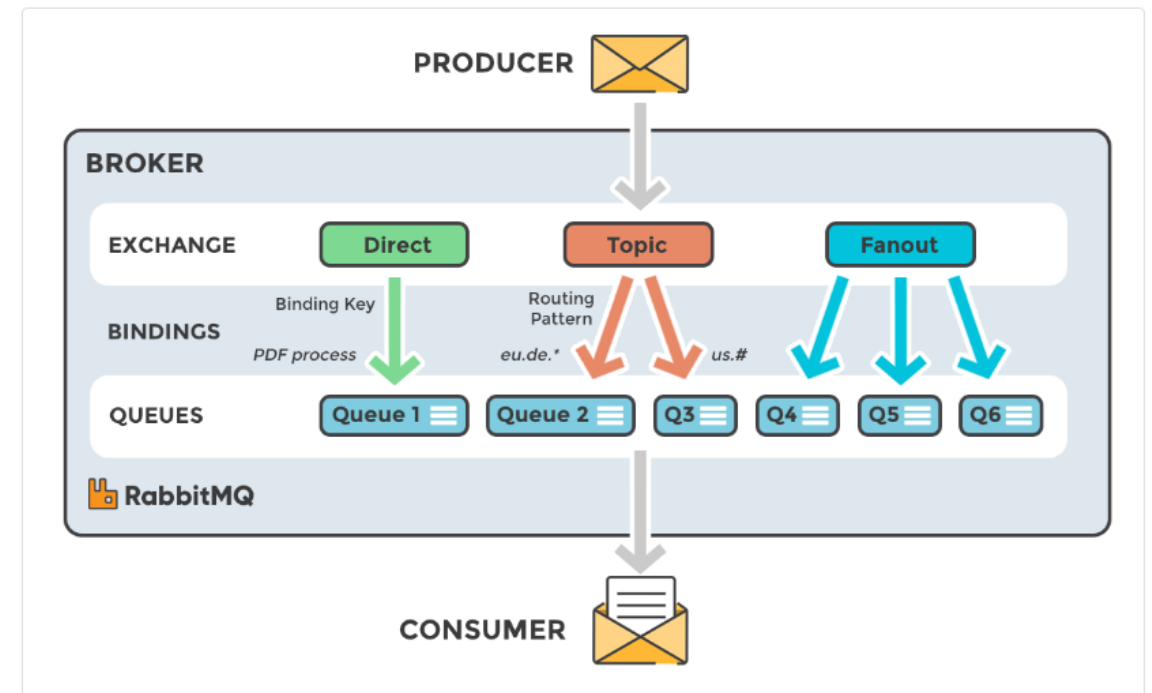
программный брокер сообщений на основе стандарта AMQP



# RabbitMQ

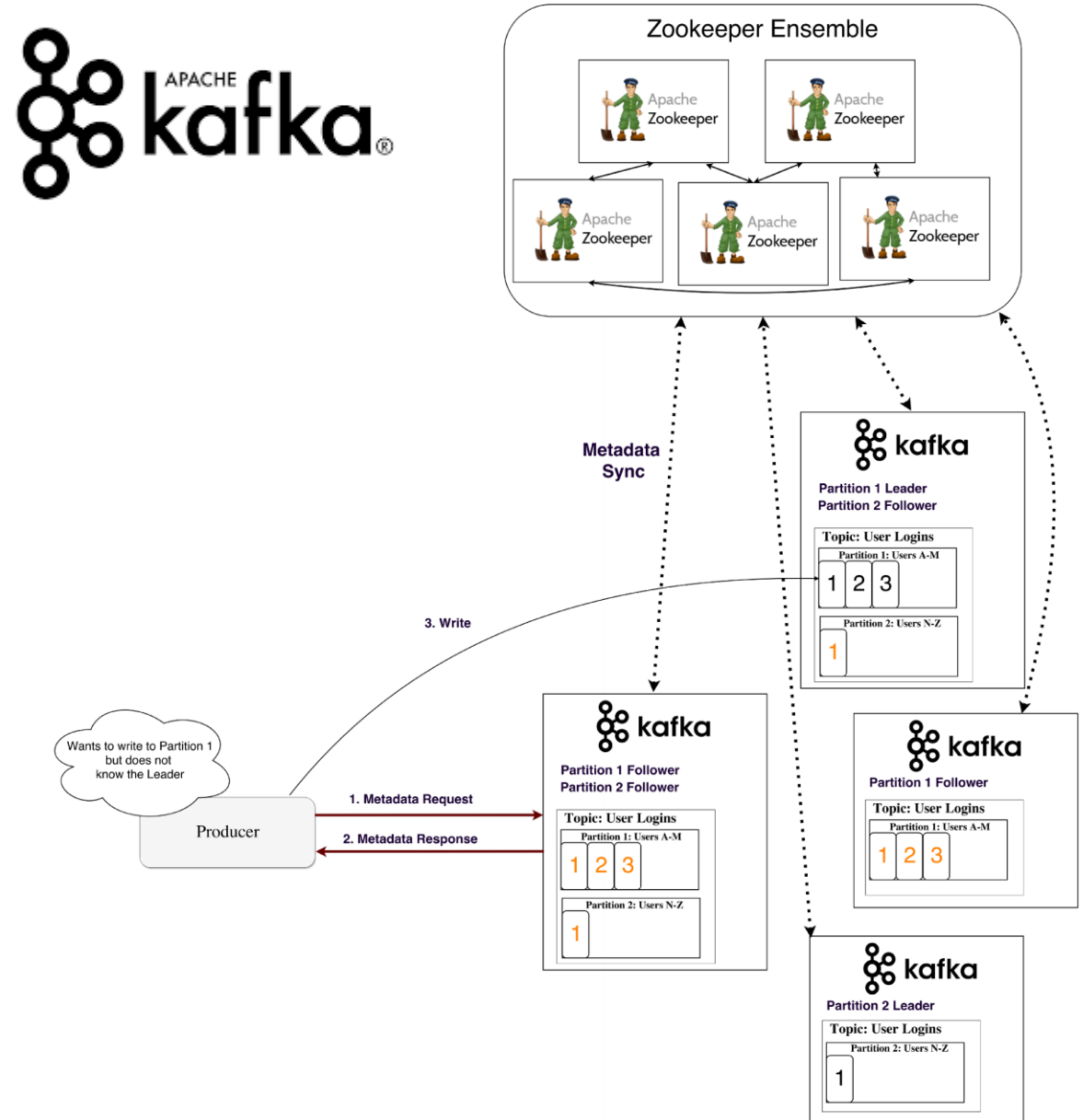
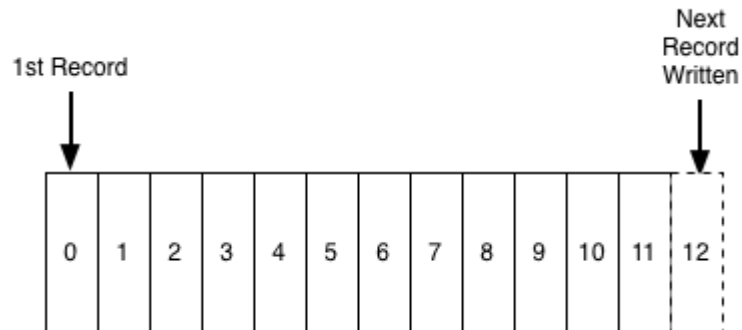
## Binders

- **Direct:** It directly maps an exchange type to a specific queue based on the routing key.
- **Fanout:** It routes messages to all the queues from the bound exchange.
- **Topic:** It routes messages to queues (0, 1, or more) based on either full or a portion of routing key matches.
- **Headers:** It is similar to topic exchange type, but it routes based on header values instead of routing keys.



# Apache Kafka

распределенный горизонтально  
масштабируемый отказоустойчивый  
журнал коммитов

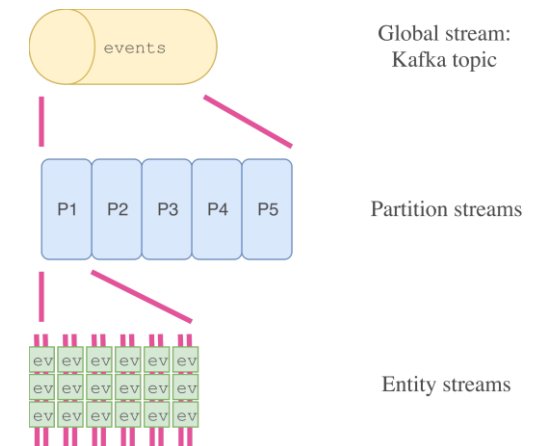


# Событийные архитектуры и Kafka

- **Разделение событий по топикам Kafka**
  - События одного агрегата должны оказаться в одном топике, в противном случае может быть нарушена последовательность событий
- **Решения**
  - Все события в одном топике (single topic)
  - На каждый **тип** агрегата приходится свой топик (topic-per-entity-type)
  - На каждый **экземпляр** агрегата приходится свой топик (topic-per-entity)
    - Плохо реализуемый на Kafka вариант

Создание нового экземпляра потребует создание нового топика, что сложно технически

На одного брокера должно приходиться порядка сотен топиков, в противном случае страдает производительность



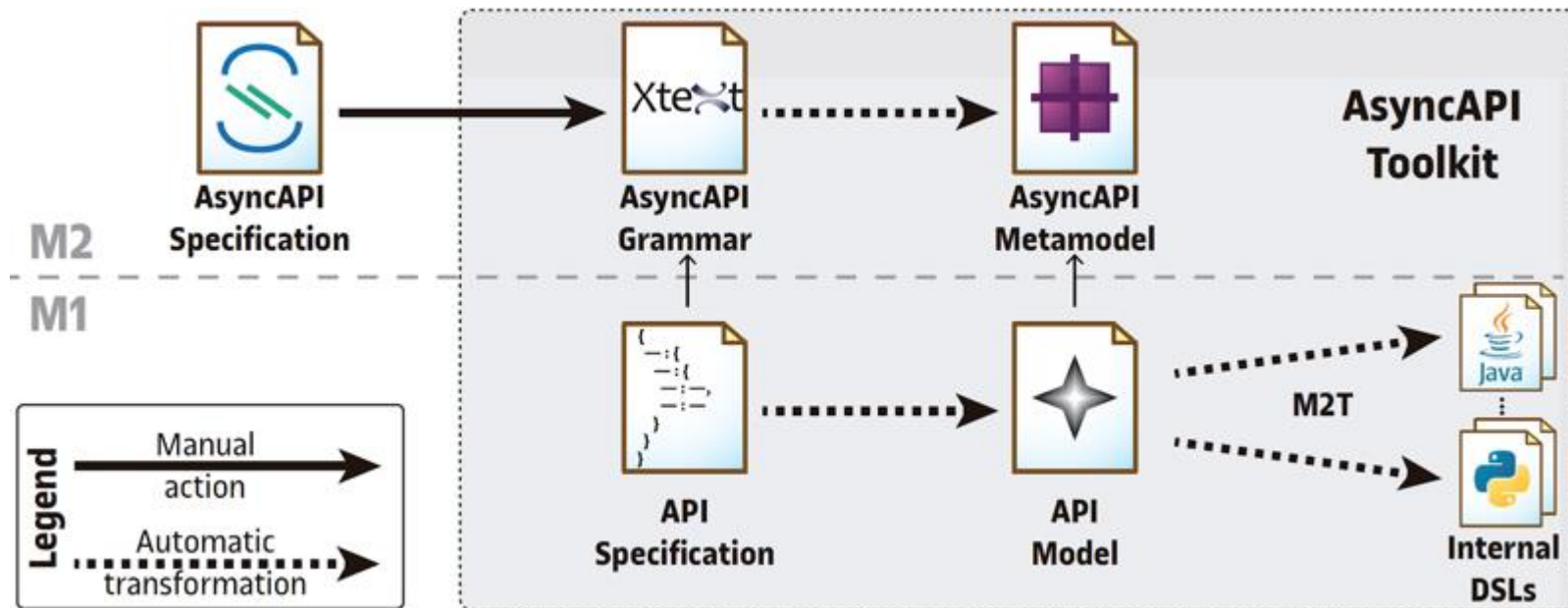
# Apache Avro



- **Инфраструктура, которая позволяет сериализовать данные в формате со встроенной схемой**
- **Сериализованные данные представлены в компактном двоичном формате, который не требует генерации прокси-объектов или программного кода**
- **Вместо использования библиотек сгенерированных прокси-объектов и строгой типизации формат Avro интенсивно задействует схемы, которые отсылаются вместе с сериализованными данными**
- **Передача схем вместе с Avro-сообщениями позволяет любому приложению десериализовать данные**

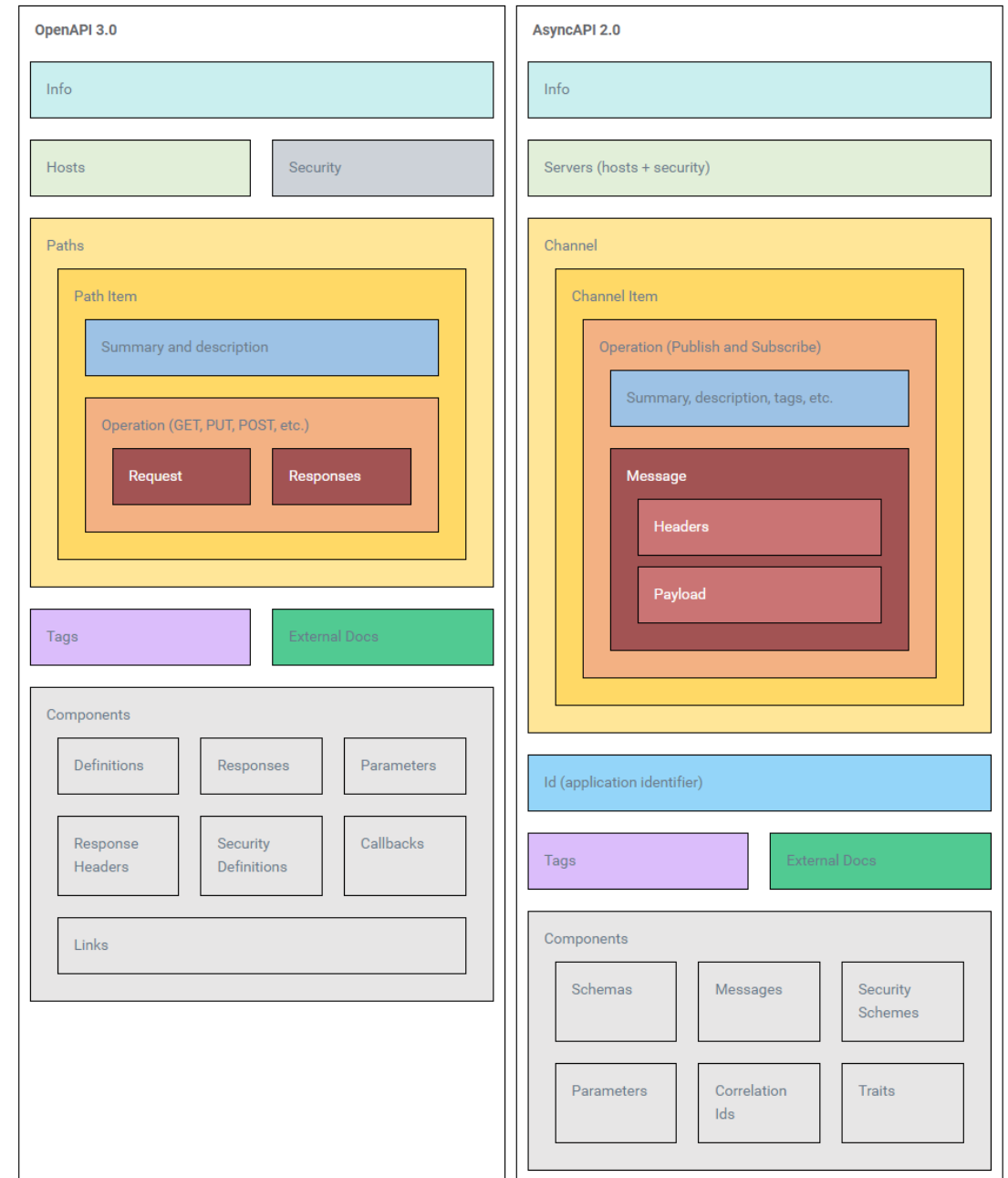
# Спецификация AsyncAPI

- Спецификация, которая позволяет вам определять API-интерфейс, управляемый сообщениями, в машиночитаемом формате
- Вы можете использовать его для API, которые работают над MQTT, AMQP, WebSockets, STOMP, и т. д.
- Спецификация очень похожа на OpenAPI (Swagger)



# Сравнение OpenAPI и AsyncAPI

*“REST APIs have OpenAPI.  
Messaging has AsyncAPI.”*



# Распределение общих ответственностей



## Нераспределенные ответственности

- При разбиении системы на сервисы остаются общие ответственности (responsibility) системы, которые нельзя отнести к конкретному сервису
- К подобным ответственностям относятся
  - Предоставление внешнего интерфейса системы
  - Управление процессом

# Управление процессами

# Управление рабочим процессом (Workflow Management)

- **Управление потоком от пользователя (Human Interaction)**

- Пользователь помнит состояние текущего процесса и через интерфейс вызывает следующую функцию

- **Управление потоком от шины**

- Умная шина, ESB в SOA

Анти-паттерны в MSA

- **Управление потоком от клиента (Drive flow from Client)**

- **Оркестровка (Orchestrated)**

- Выделенный сервис (оркестратор) управляет потоком действий

- **Хореография (Choreographed)**

- Нет единого сервиса управления процессом

Каждый сервис сам знает кому передавать управление дальше

# Управление потоком от пользователя (Human Interaction)

- Проблемы
  - Связь функций в процессе неясна
  - Контекст операции может быть потерян
  - Сложности при обработке ошибок
  - Гонки
  - Обработка тайм-аутов



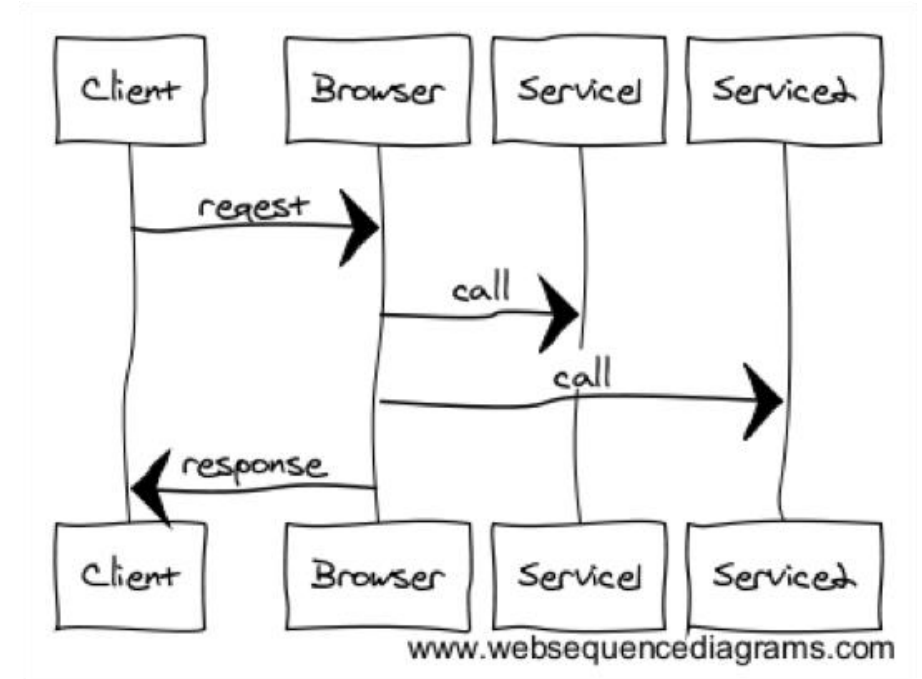
# Управление потоком от шины

## ■ Проблемы ESB

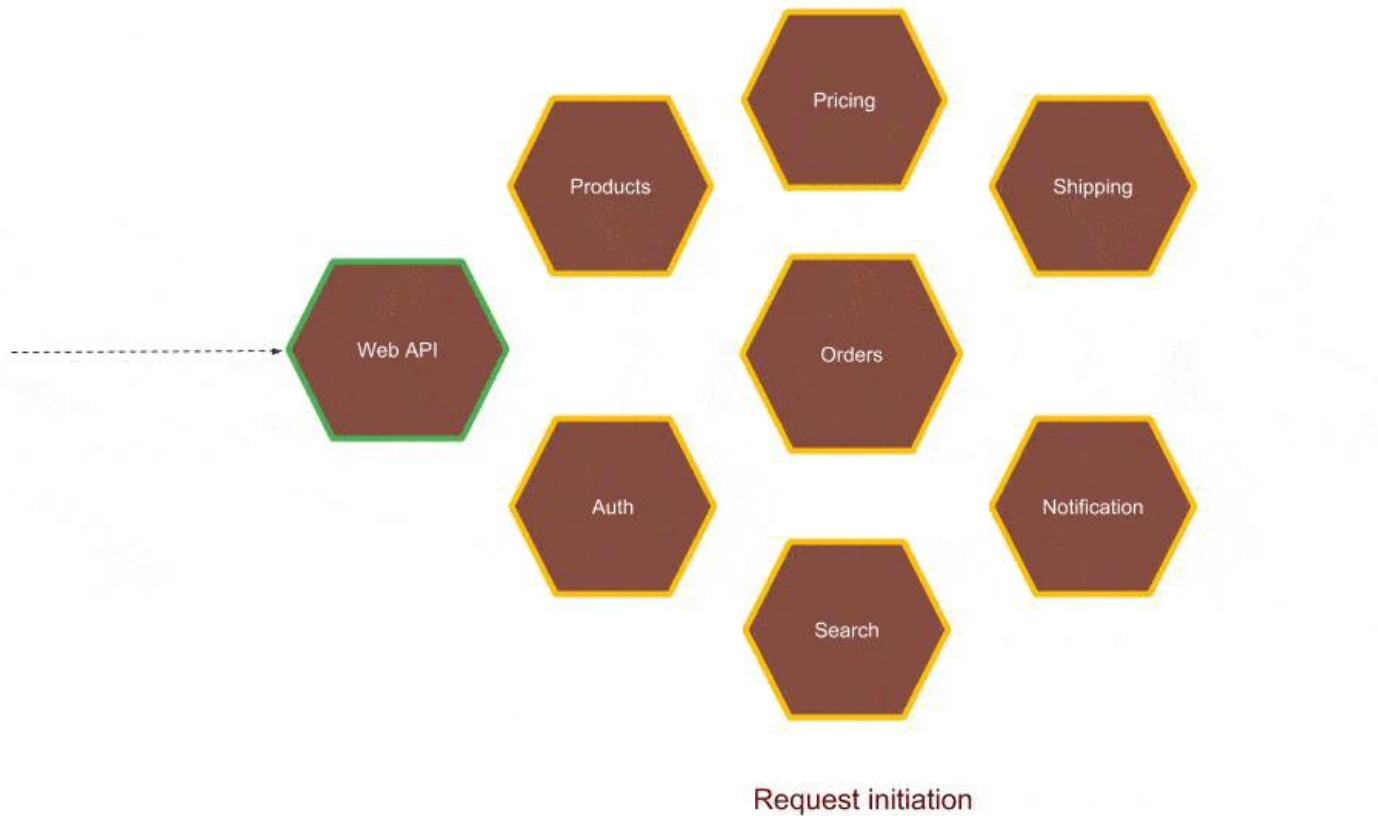
- ESB дает слишком высокую связанность
  - Сервисы зависят от ESB, в которой содержится много «магии»
- Управление SOAP отнимает слишком много времени
  - Любое изменение ломает все клиенты
- ESB эксперты – узкое место
  - Специалисты по ESB обычно очень занятые люди
- Оркестровка через ESB – дорого
  - Требуется как модификация сервиса, так и модификация шины
- ESB завязана на вендоров

# Управление потоком от клиента

- Оркестратором выступает клиент, который знает в какой последовательности какие функции должны быть вызваны
- Проблемы
  - Большое количество вызовов через интернет
  - Проблемы безопасности
  - Может ли быть запущен сервис на стороне клиента
  - Может ли клиент держать состояние, доверяем ли ему



# Синхронная хореография

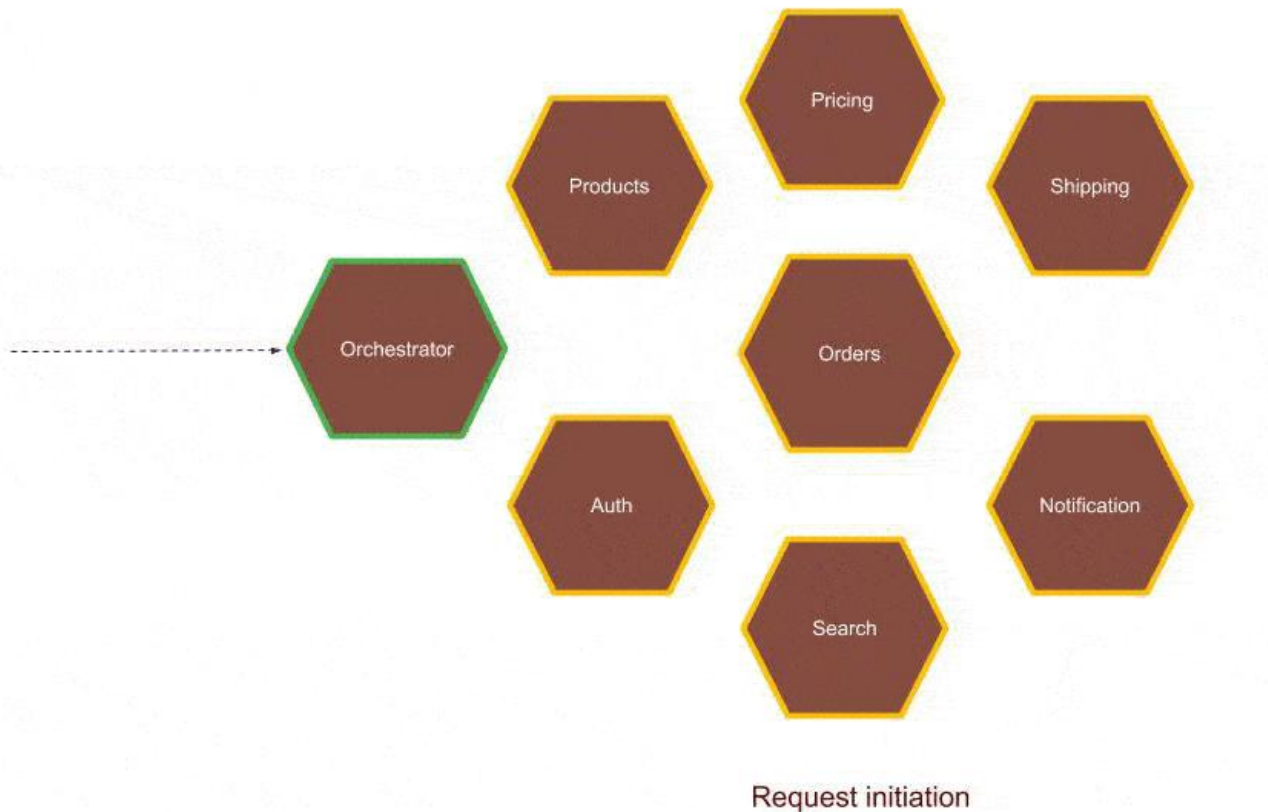


**Сложно изменять рабочий процесс**  
**Управление процессом**  
**распределено**

**Простота при получении ответа**  
**Синхронный процесс**

**Низкая пропускная способность**  
**Каждый вызов блокирует работу**

# Синхронная последовательная оркестровка



**Просто изменять рабочий процесс**  
**Управление процессом**  
**сосредоточено в оркестраторе**

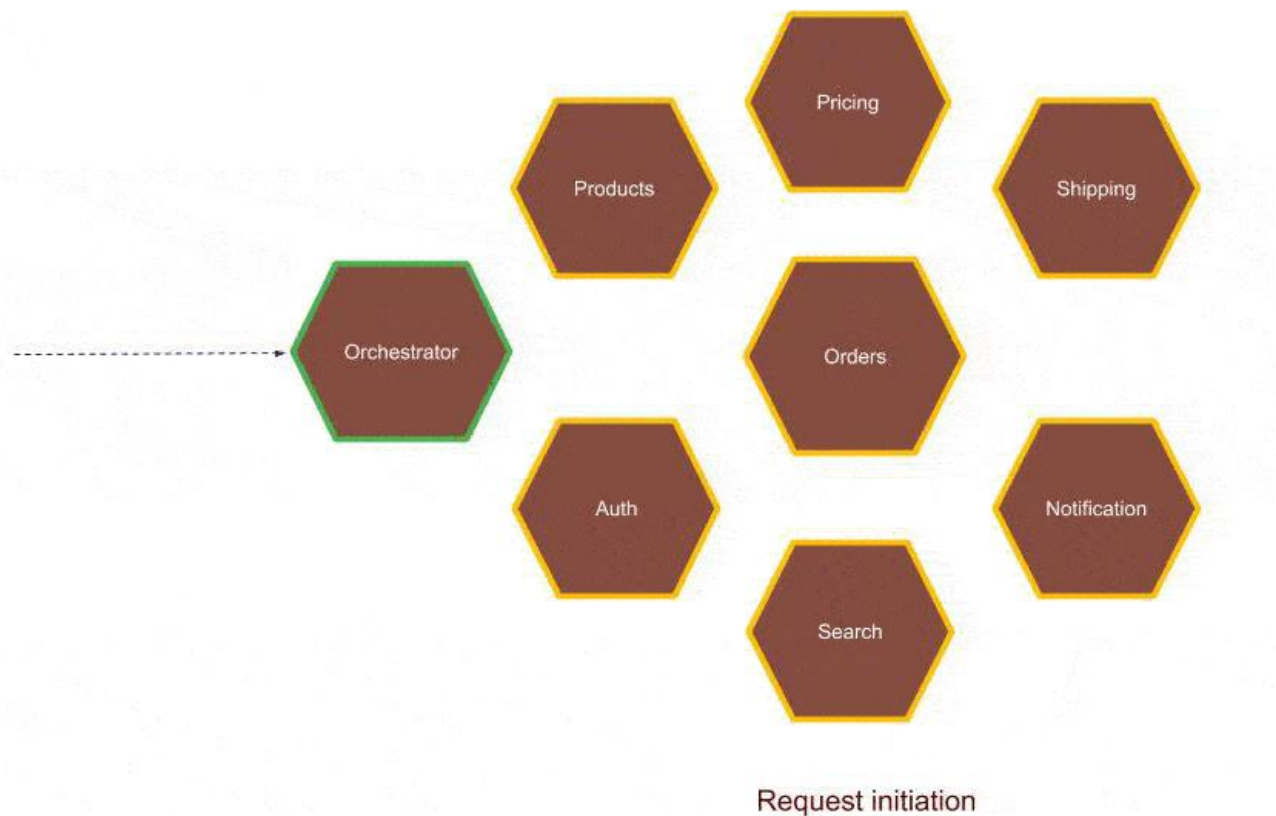
**Простота при получении ответа**  
**Синхронный процесс**

**Низкая надежность**  
**Оркестратор единая точка отказа**

**Приемлемая пропускная**  
**способность по чтению**  
**Оркестратор является бутылочным**  
**горлышком**



# Синхронная параллельная оркестровка



**Просто изменять рабочий процесс**  
**Управление процессом**  
**сосредоточено в оркестраторе**

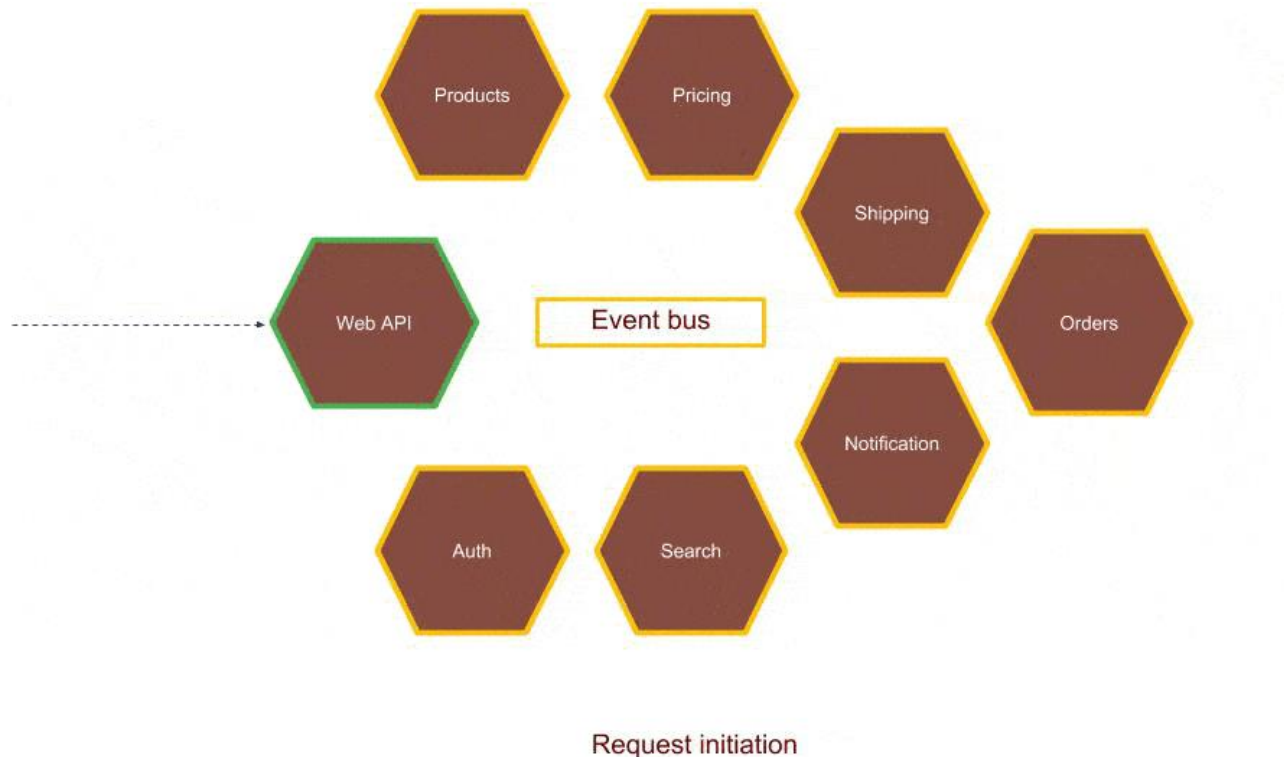
**Простота при получении ответа**  
**Синхронный процесс**

**Сложнее в реализации за счет**  
**распараллеливания**

**Низкая надежность**  
**Оркестратор единая точка отказа**

**Повышенная пропускная**  
**способность**  
**За счет распараллеливания потоков**  
**Оркестратор является бутылочным**  
**горлышком**

# Асинхронная хореография



**Сложно изменять рабочий процесс**  
**Управление процессом**  
**распределено по системе**

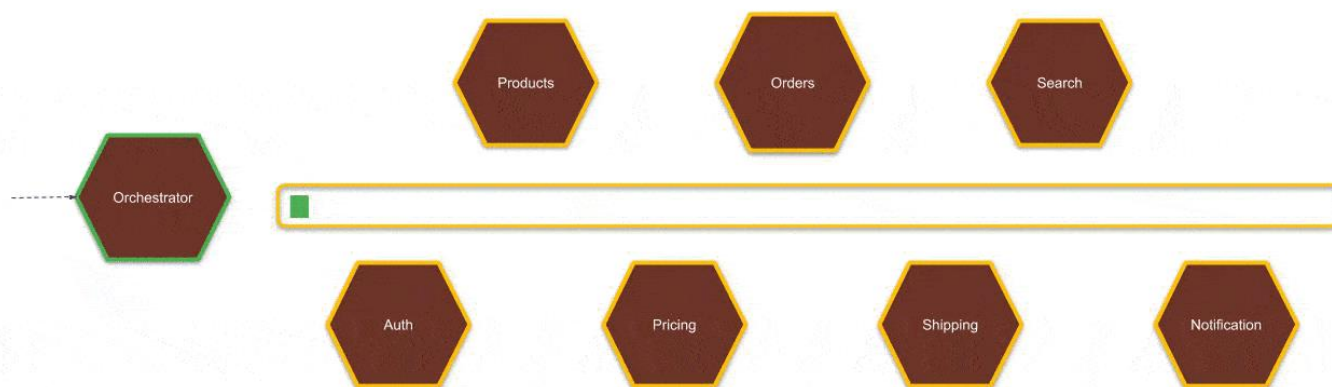
**При получении ответа требуется**  
**посредник**

**Сложно поддерживать**  
**согласованность между**  
**потребителями и поставщиками**  
**событий (выбор типа сообщения)**

**Хорошая пропускная способность**

**Подходит для интенсивных**  
**операций записи и для реализации**  
**сайд эффектов (например,**  
**нотификаций)**

# Асинхронная последовательная оркестровка



**Просто изменять рабочий процесс**  
**Управление процессом**  
**сосредоточено в оркестраторе**

**При получении ответа требуется**  
**посредник**

**Низкая надежность**  
**Оркестратор единая точка отказа**

**Хорошая пропускная способность**

**Подходит для интенсивных**  
**операций записи**

## Гибрид с оркестровкой и событийной хореографией

- Другим успешным вариантом являются гибридные системы с оркестровкой и событийной хореографией
  - Оркестровка для явного выполнения потока (бизнес-процесс)
  - Хореография обрабатывает неявное выполнение (сайд эффект)
- Это объединение двух подходов обеспечивает лучшее из обоих миров. Хотя существует необходимость в мерах предосторожности, чтобы гарантировать, что они не пересекаются с обязанностями и четкие границы диктуют их функционирование

# Организация внешних коммуникаций

Взаимодействие север-юг

## Внешние клиенты

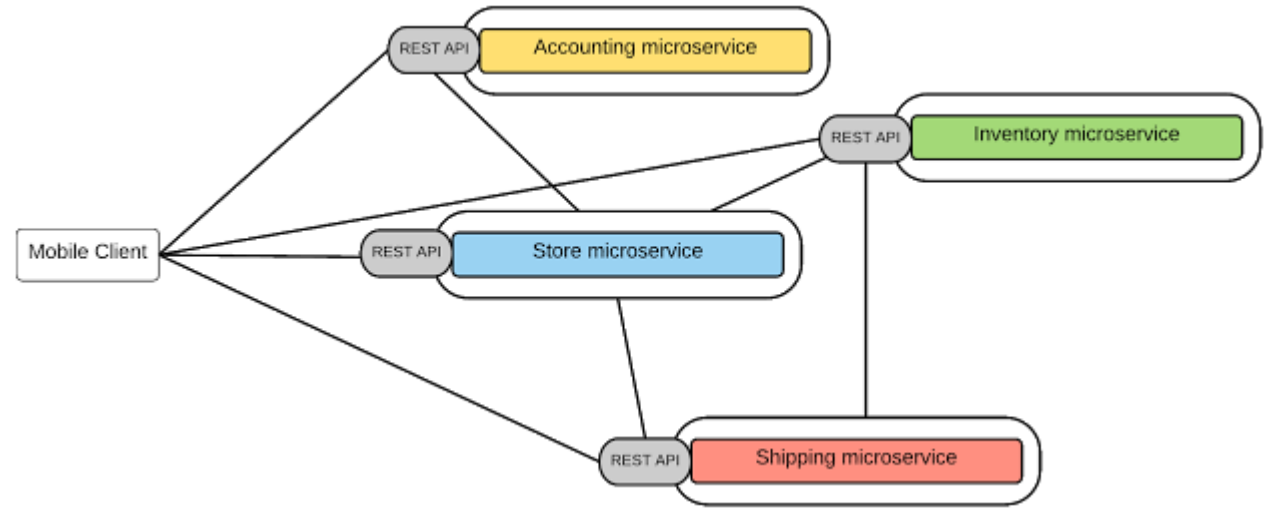
- Веб-приложения, реализующие браузерные пользовательские интерфейсы для заказчиков, и администраторов
- JavaScript-приложение, работающее в браузере
- Мобильные приложения — одно для заказчиков, второе для курьеров
- Приложения, написанные сторонними разработчиками

# Архитектурные стили

- **Прямой вызов (Point-to-point Style)**
  - Клиент знает сервис
- **Шлюз (API-Gateway Style)**
- **Брокер сообщений (Message Broker style)**

## Прямой вызов (Point-to-point Style)

- Логика маршрутизации находится на сервисе (endpoint)
- Сервисы общаются напрямую, обращаясь к API друг друга
- Для больших систем считается анти-паттерном



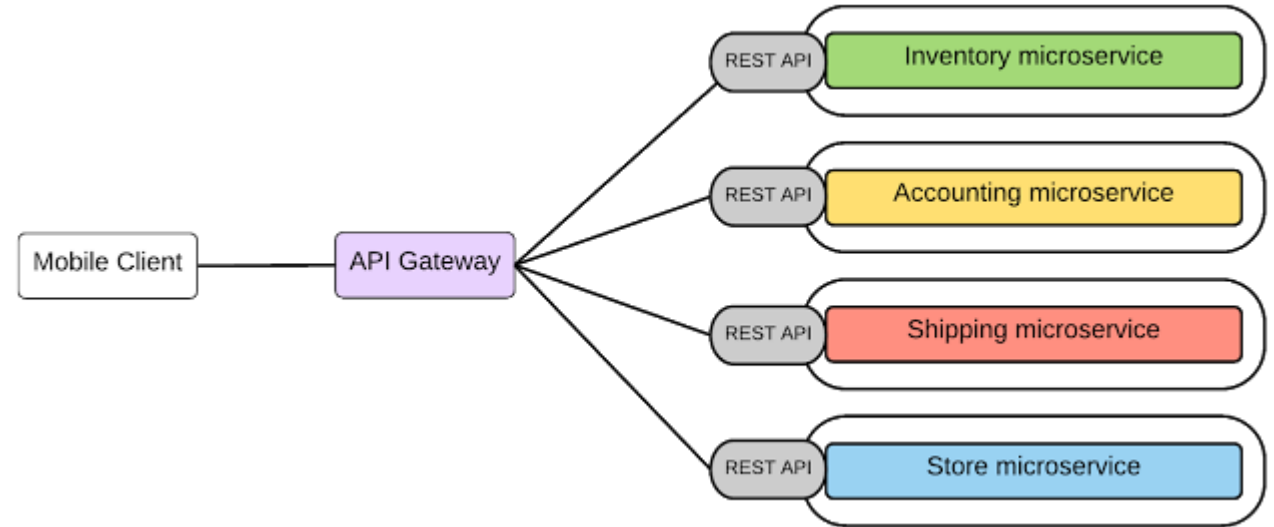


# Проблемы прямого вызова

- **Сложность реализации не клиенте.**
  - Для извлечения нужных данных с помощью мелко раздробленных API клиентам приходится выполнять несколько запросов.
- **Недостаточная инкапсуляция, связанная с тем, что клиенты знают о каждом сервисе и его API, затрудняет внесение изменений в архитектуру и API.**
- **Сервисы могут задействовать механизмы IPC, которые нецелесообразно или неудобно использовать на клиентской стороне, особенно клиентам за пределами брандмауэра.**
- **Дублирование вспомогательных функций (авторизация, throttling, мониторинг и т. д.).**
- **Ненадежный контроль коммуникаций.**

# Шлюз (API-Gateway Style)

- Выносим вспомогательные функции на шлюз
- Управляемый API (GraphQL) поверх REST
- Комбинация стилей Microservices и API-Management
- Возможности
  - Каждому клиенту свой API (BFF)
  - Единое пространство реализации вспомогательных функций
  - Облегченные механизмы маршрутизации и преобразования данных
  - Облегченные микросервисы



# Функции API шлюза

- **Маршрутизация запросов**
  - Эта функция идентична возможностям обратного прокси, которые предоставляют такие серверы как NGINX
- **Объединение API**
  - Один запрос может быть обработан несколькими сервисами
- **Преобразование протоколов**
  - Например, преобразуя GraphQL запросы в REST и gRPC запросы внутри системы
- **Граничные функции (Edge functions)**

## Граничные функции (Edge functions)

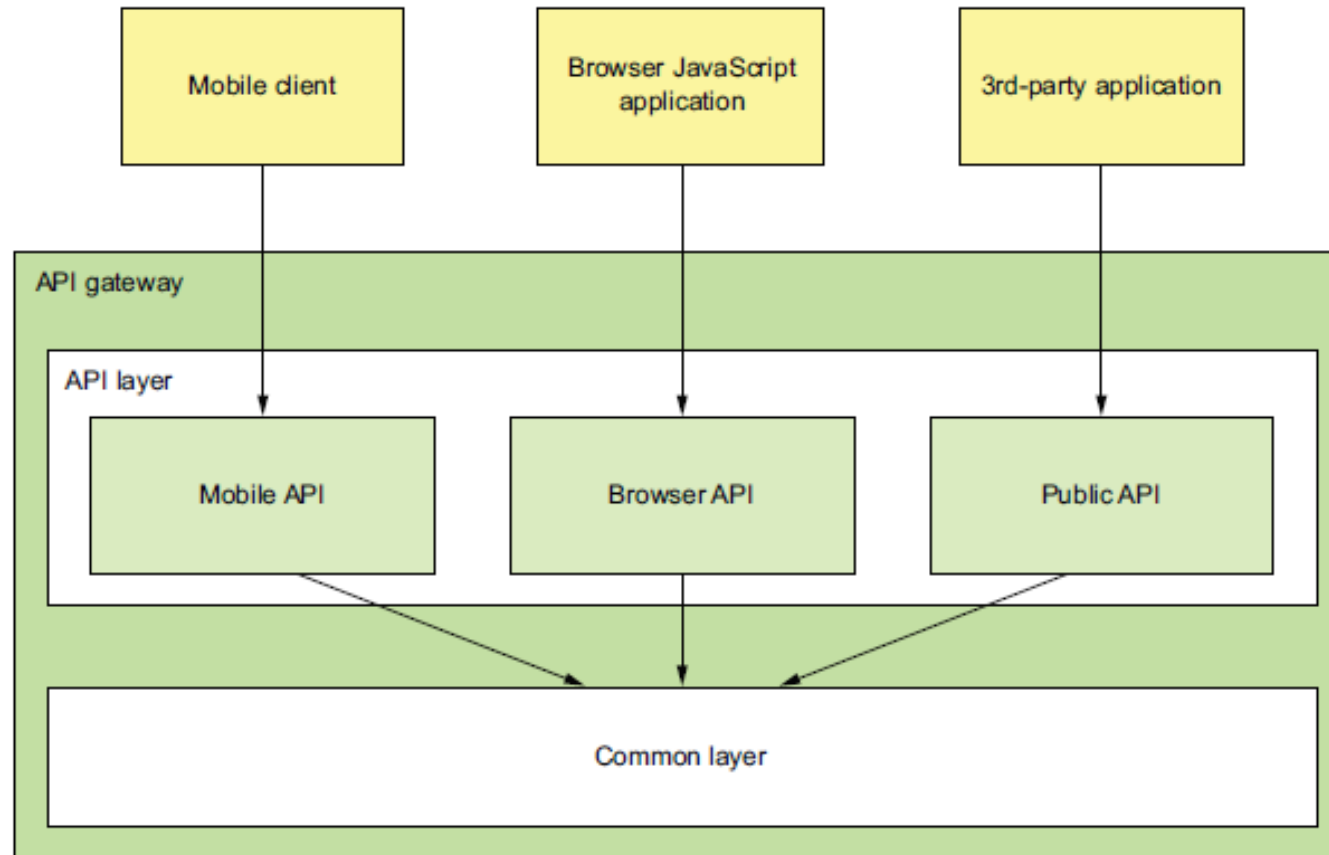
- **Аутентификация** — проверка подлинности клиента, который делает запрос
- **Авторизация** — проверка того, что клиенту позволено выполнять определенную операцию
- **Ограничение частоты запросов** — контроль над тем, сколько запросов в секунду могут выполнять определенный клиент и/или все клиенты вместе
- **Кэширование** — кэширование ответов для снижения количества запросов к сервисам
- **Сбор показателей** — сбор показателей использования API для анализа, связанного с биллингом
- **Ведение журнала запросов** — запись запросов в журнал

## Раздельный API для клиентов

- **API шлюз может предоставлять единое API для всех клиентов**
  - **этот подход не является эффективным**
- **API шлюз может предоставлять каждому клиенту отдельное API**
- **Каждый клиент может получить свой собственный шлюз**
  - **Backend for Frontend (BFF)**

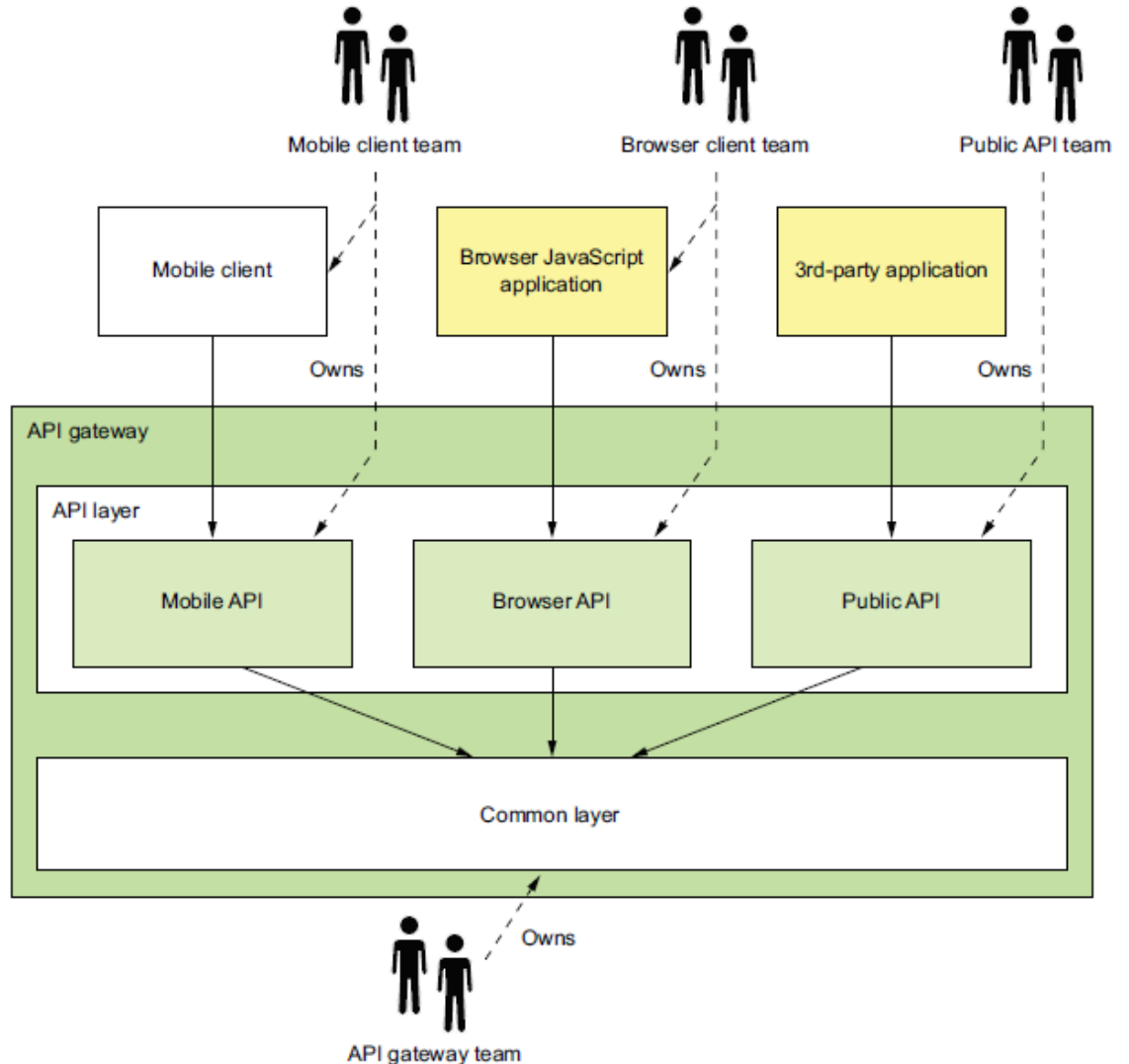
# Архитектура API-шлюза

- API-шлюз имеет двухуровневую архитектуру



# Модель владения

- Netflix: отдать API-модуль на откуп клиентским разработчикам, занимающимся мобильными устройствами, веб-приложениями и публичным API
- Если ваши клиенты пишут на JavaScript, то логично и шлюз реализовывать на этом языке (платформа NodeJS)



# Серверы для клиентов (Backend for Frontend, BFF)

- Проблема

- Различные части шлюза могут достаться различным командам

- Решение

- разделение шлюза по клиентам (Backend for Frontend (BFF))

Фил Кальсадо (Phil Calcado)

- Результат

- Каждый шлюз может быть разработан с помощью отдельной технологии

- Проблема

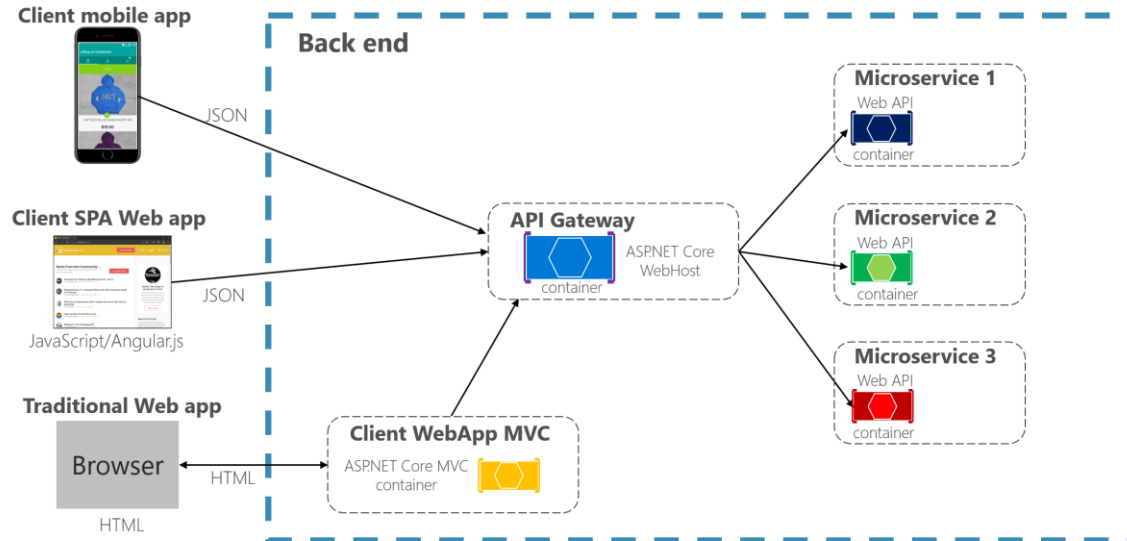
- что делать с общей частью шлюза ?

Ее нельзя выделить в отдельный подключаемый модуль (дублирование)

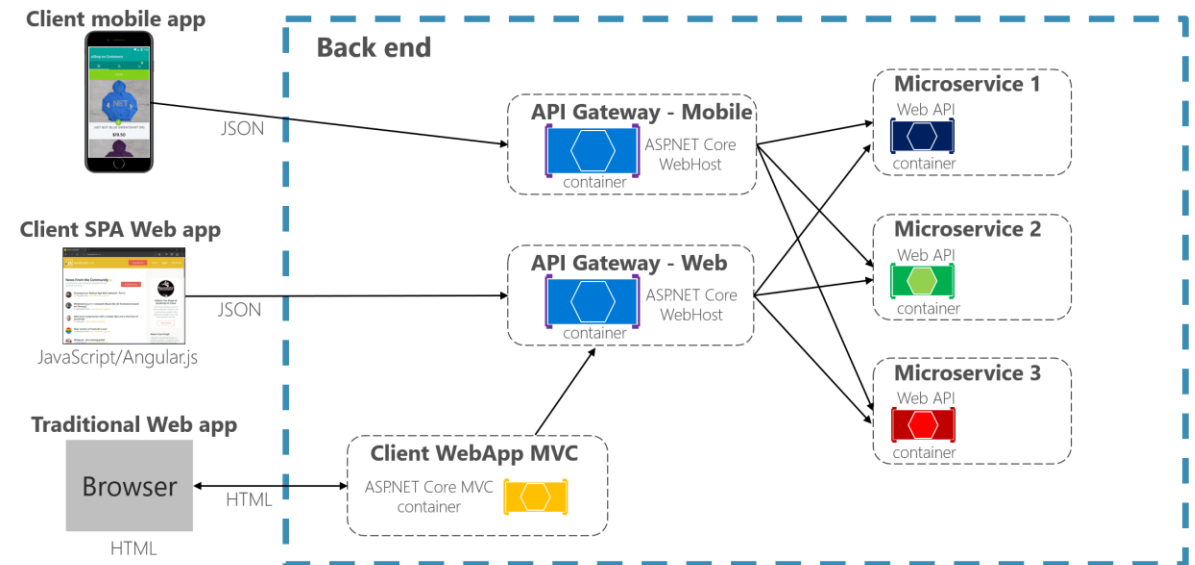


# Backend for Frontend (BFF)

## Using a single custom **API Gateway service**



## Using multiple **API Gateways / BFF**



# API Шлюз (Результаты)

- **Преимущества:**
  - Инкапсулирует внутреннюю структуру приложения.
  - Каждый клиент получает отдельный API, что снижает количество запросов между ним и приложением.
  - Упрощает клиентский код.
  - Упрощает код сервисов (выносятся граничные функции).
- **Недостатки:**
  - Это еще один высокодоступный компонент, который нужно разрабатывать, развертывать и администрировать.
  - Существует риск того, что API-шлюз начнет тормозить разработку.
    - Его следует обновлять каждый раз при «выставлении наружу» API очередного сервиса.

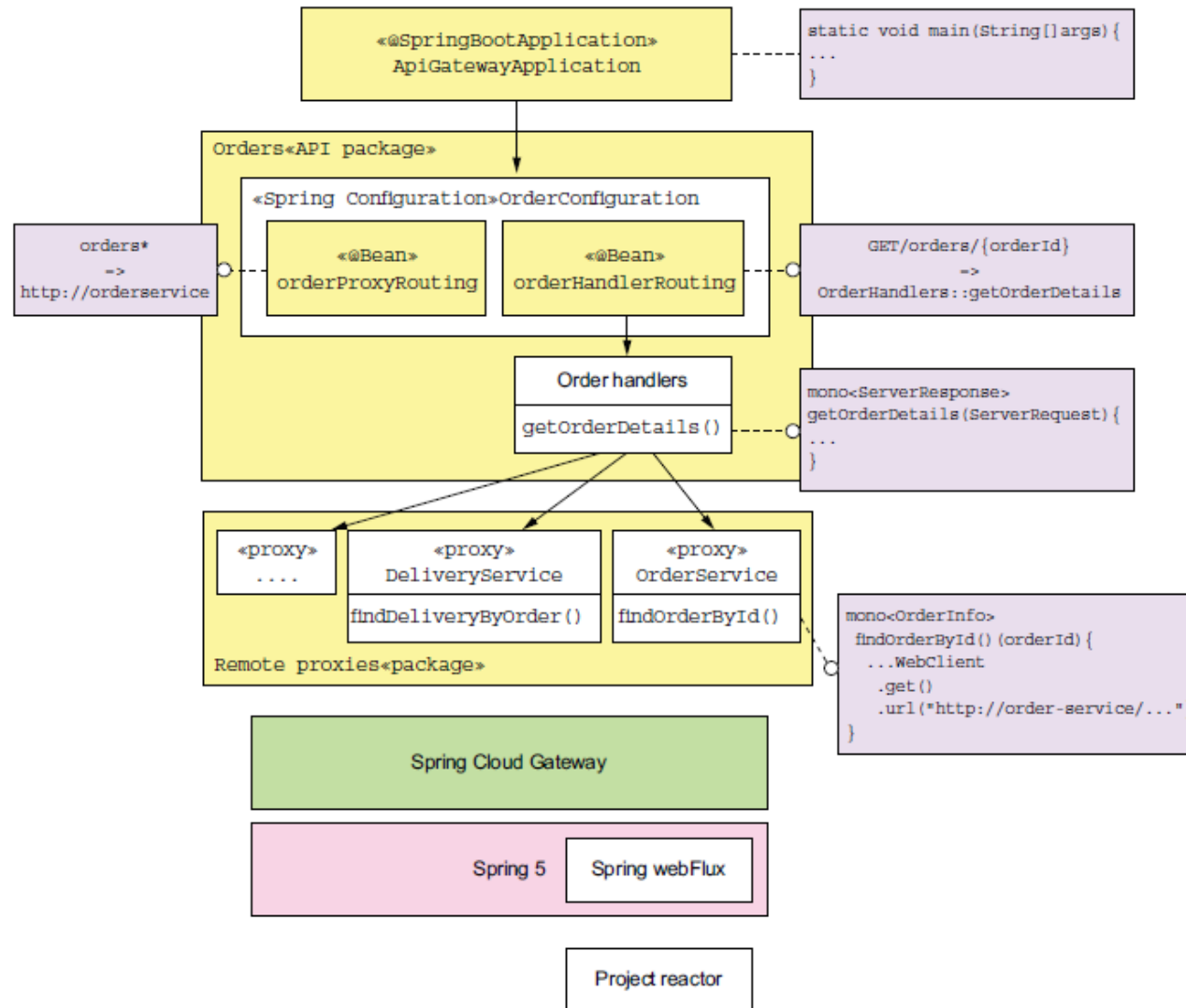
# Реализация API Шлюза

- **Производительность и масштабируемость.**
  - Синхронные и несинхронные варианты работы
  - Проблема C10K
- **Написание поддерживаемого кода с помощью абстракций реактивного программирования.**
  - Распараллеливание обращений
- **Обработка частичных отказов.**
  - Избыточное количество шлюзов за балансировщиком
  - Прекращение работы с недоступным сервисом (прерыватель)
- **Реализация шаблонов, общих для архитектуры приложения**
  - Службы обнаружения сервисов, службы мониторинга и т. п.

# Использование готовых решений

- **Готовые решения**
  - Kong (на базе NGINX)
  - Traefik (Golang)
  - Tyk.io (Golang)
  - WSO2 API Manager
  - Gravitee
- **Каркасы (фреймворки)**
  - Netflix Zuul (NodeJS Express)
  - Spring Cloud Gateway (JVM)
- **Технологии структурированных API**
  - Netflix Falcor (моделирует данные в виде виртуального объектного графа JSON)
  - GraphQL (моделирует данные в виде графа с полями и ссылками на другие объекты)
    - GraphQL Apollo + DataLoader

# Архитектура API шлюза, созданного с помощью Spring Cloud Gateway



# Kong (1 из 2)



**Шлюз API с открытым исходным кодом, построенный на основе NGINX**

- **Лицензия**

- Основной функционал бесплатен
- Предоставляет лицензии на обслуживание и поддержку для крупных предприятий (KongHQ)
- Некоторые функции, такие как интерфейс администратора, безопасность и портал разработчика, доступны только с корпоративной лицензией

- **Развертывание**

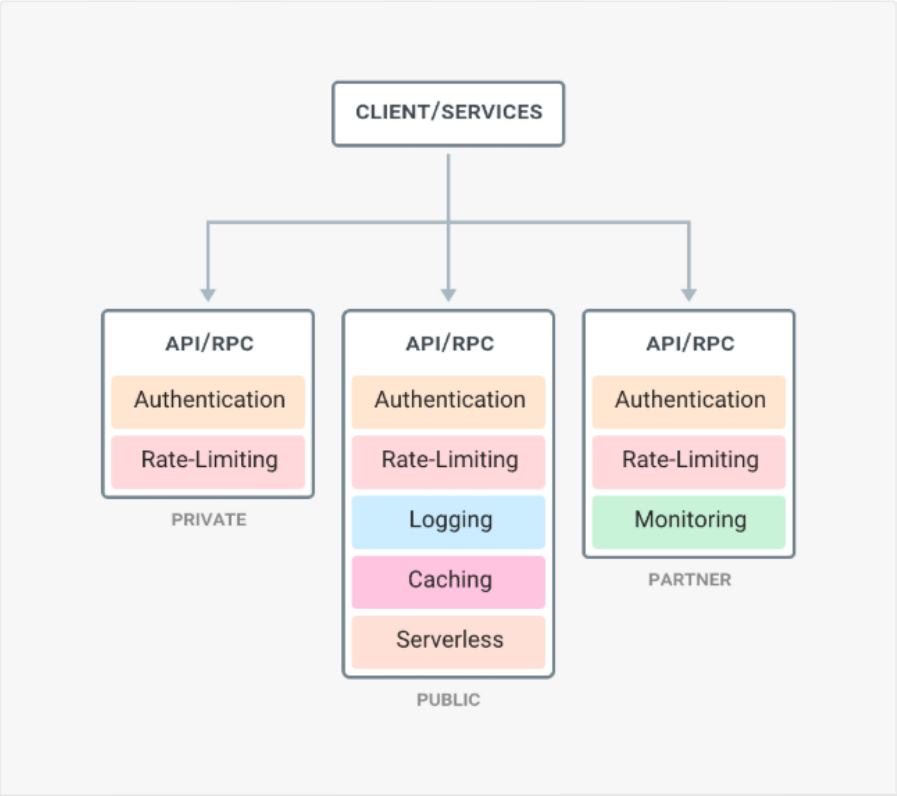
- Широкий выбор вариантов установки с готовыми контейнерами, такими как Docker и Vagrant
- При развертывании требует запуск БД (PostgreSQL, Cassandra)
- Некоторые плагины (ограничение скорости) требуют Redis

# Kong (2 из 2)

## ■ Функциональность

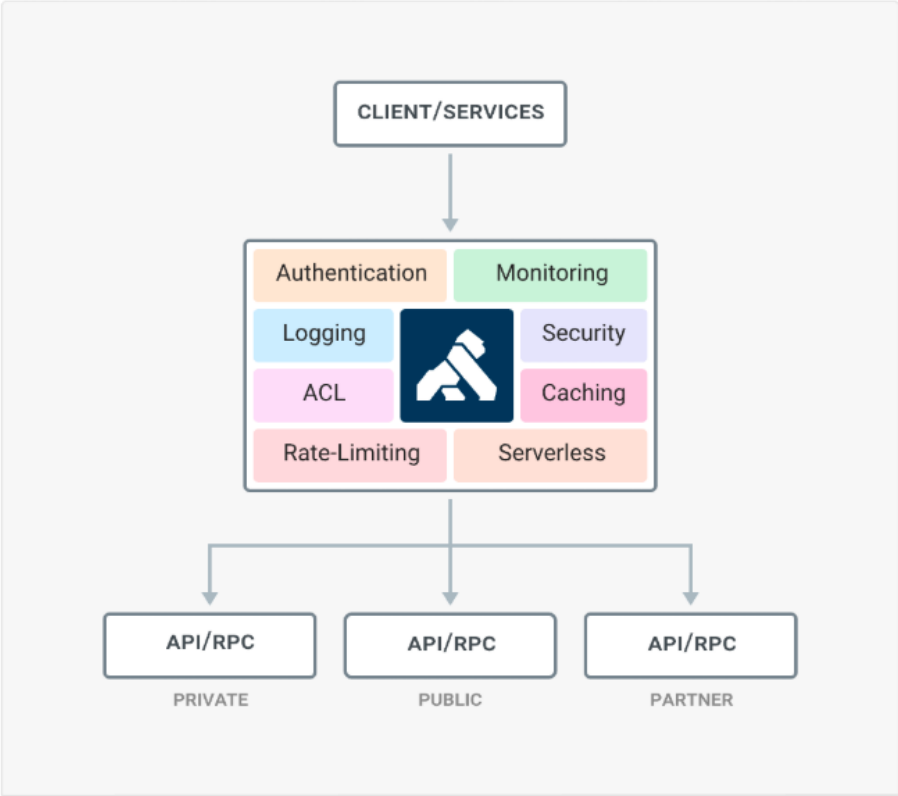
- Поддержка основных функций шлюза: создание ключей API, маршрутизация на несколько микросервисов и т. п.
- Нет биллинга
- REST API администратора
- Десятки плагинов от сообщества
  - API-аналитика (Moesif)
  - Кэширования
  - Проверка JWT (JSON Web Tokens)
- Нет слоя преобразования данных (SOAP, XML)
- Не поддерживает «Объединение API»

### The Redundant Old Way



- ✗ Common functionality is duplicated across multiple services
- ✗ Systems tend to be monolithic and hard to maintain
- ✗ Difficult to expand without impacting other services
- ✗ Productivity is inefficient because of system constraints

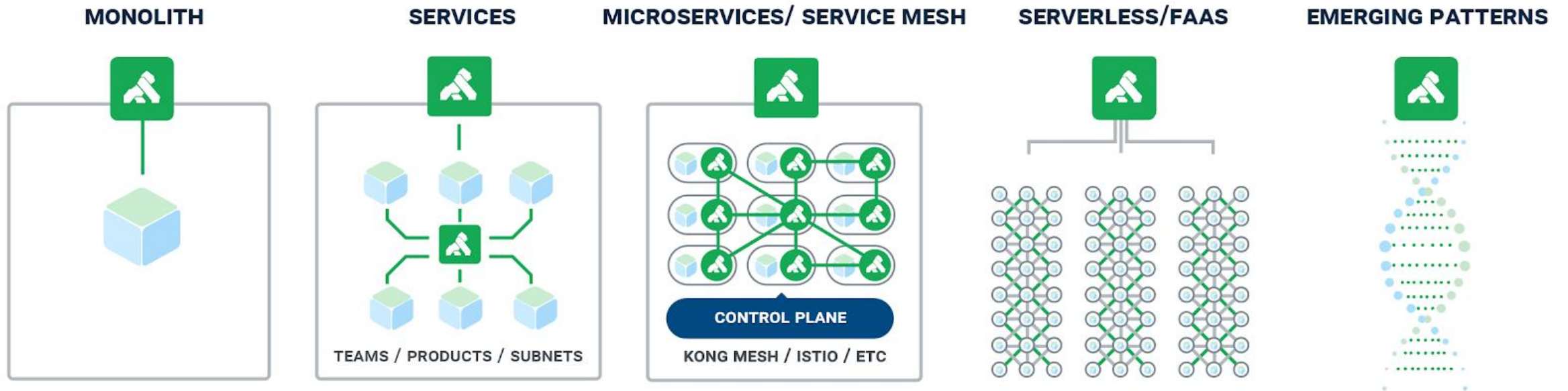
### The Kong Way



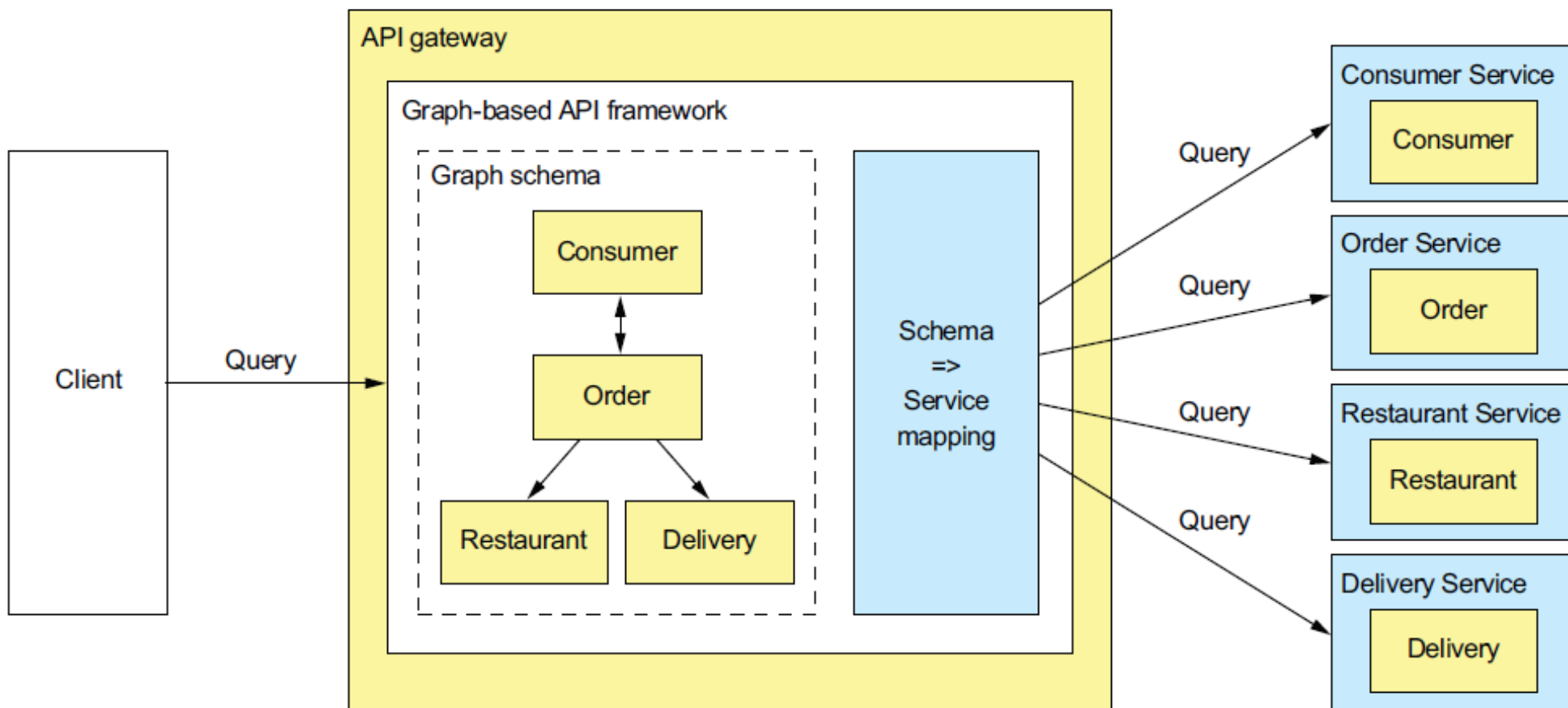
- ✓ Kong orchestrates common functionality
- ✓ Build efficient distributed architectures ready to scale
- ✓ Expand functionality from one place with a simple command
- ✓ Focus on your product and let Kong do the REST



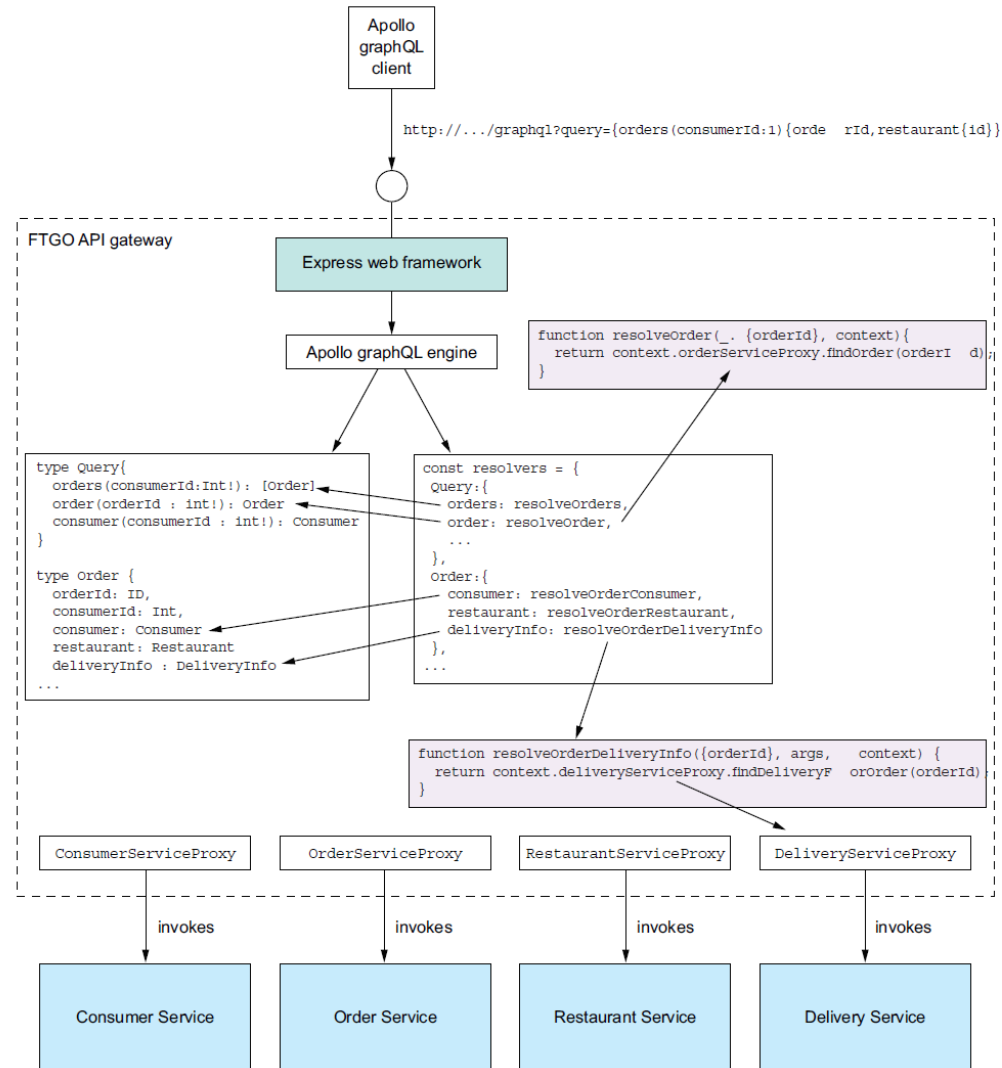
# Kong (варианты развертывания)



# Структурированные API

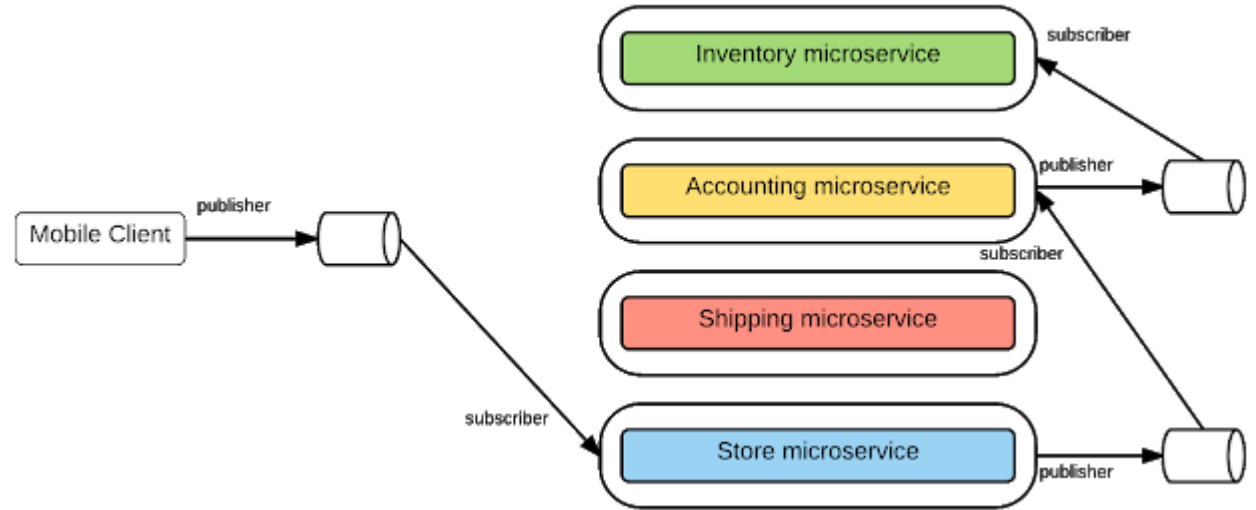


# Структурированные API (Пример реализации)



# Брокер сообщений (Message Broker style)

- Поддерживает сценарии
  - one-way requests
  - publish-subscribe
- Потребитель сообщения не знает о поставщике сообщения
- Сообщение хранится до тех пор, пока потребитель не может его обработать



**Буду рад ответить на ваши вопросы**

