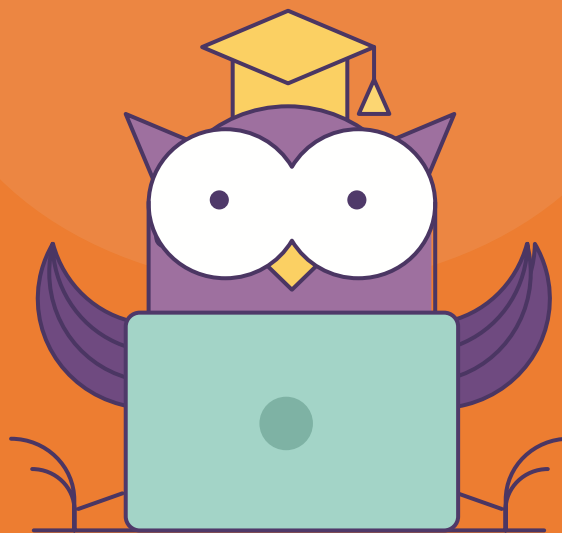


# Меня хорошо слышно && видно?



Напишите в чат, если есть проблемы!

Ставьте  если все хорошо

# Event Driven Architecture

Архитектор ПО



# 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
- Телеграм @EvgeniyN

# 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



Опрос по программе - каждый месяц в ЛК

# Цели занятия

- рассмотреть основы Event Driven Architecture;
- рассмотреть взаимодействие на основе событий;
- посмотреть на подходы к проектированию событий;
- рассмотреть подход Event Sourcing

# Карта вебинара

- События и сообщения
- Межсервисное взаимодействие
- Паттерны проектирования событий
- Event Sourcing

# 01

## Введение



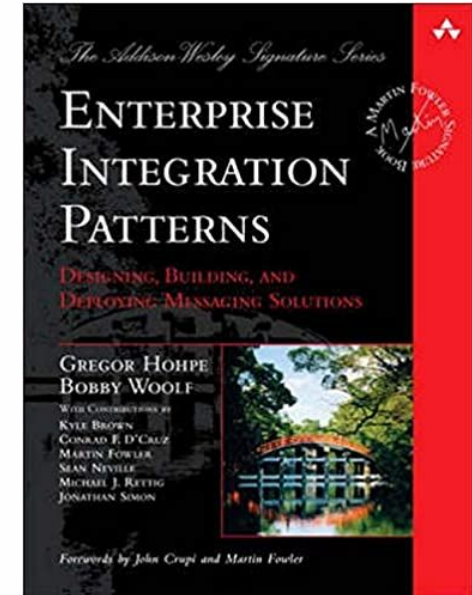
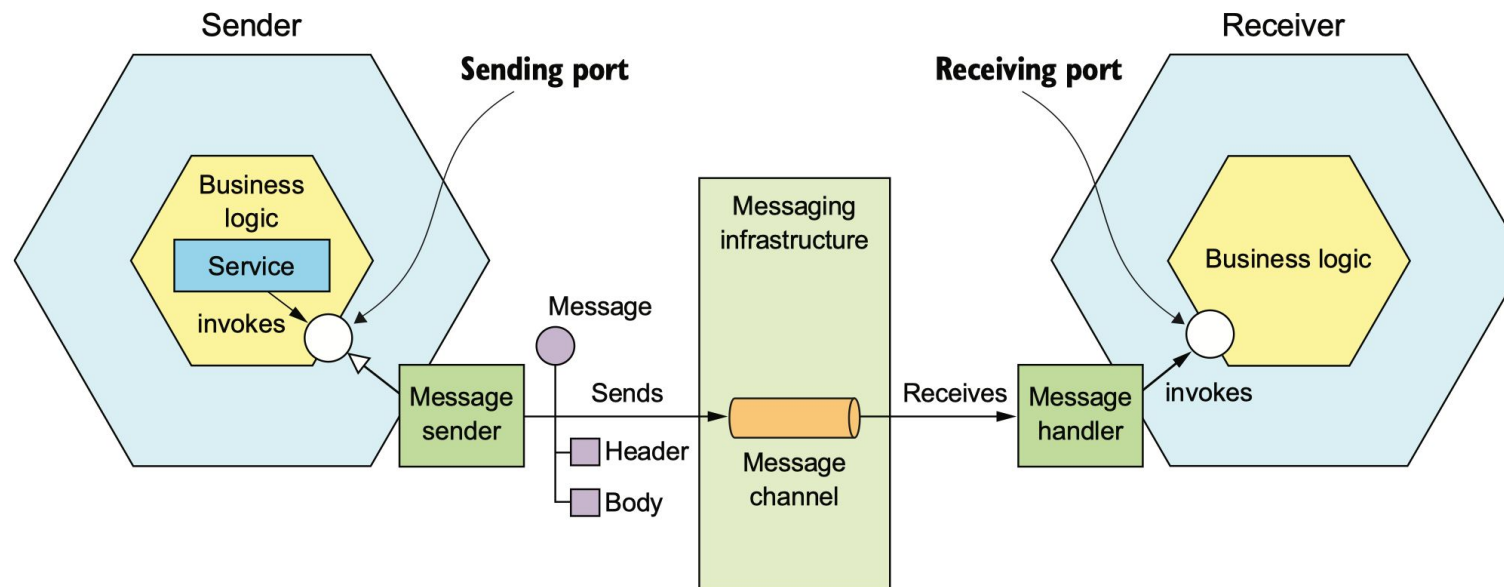
# Сообщения

<https://www.enterpriseintegrationpatterns.com/>

**Команда** - указание сделать что-либо

**Событие** - сообщение о том, что с отправителем произошло что-то, заслуживающее внимания. Часто представляет изменение состояния доменного объекта

**Документ** - сообщение, содержащее только данные



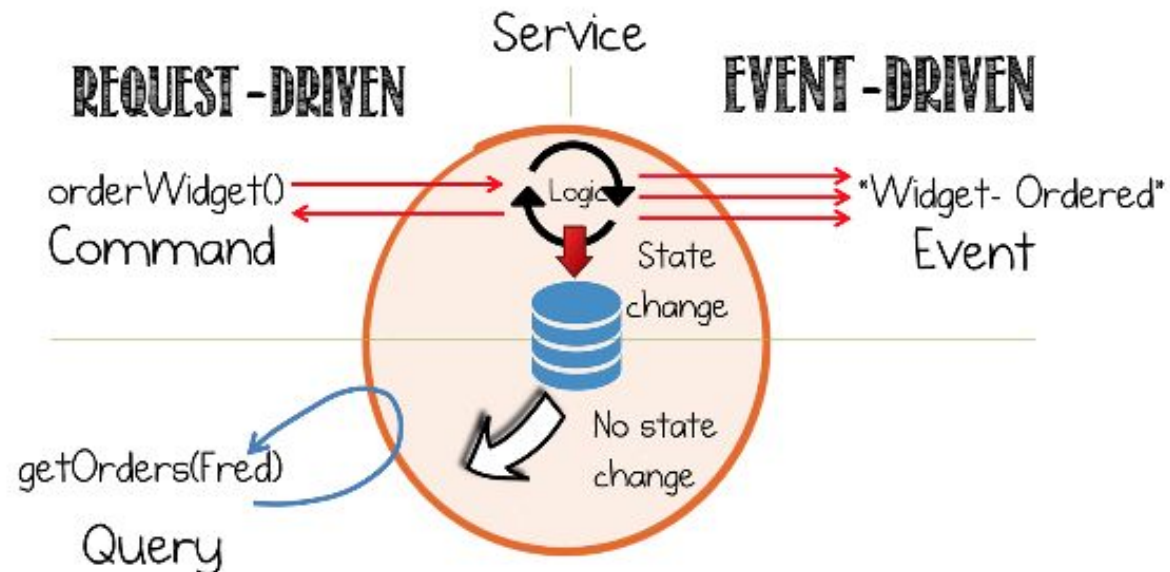
# Event vs Command

**Команда** – это «глагол» - требование что-то сделать от другого сервиса:

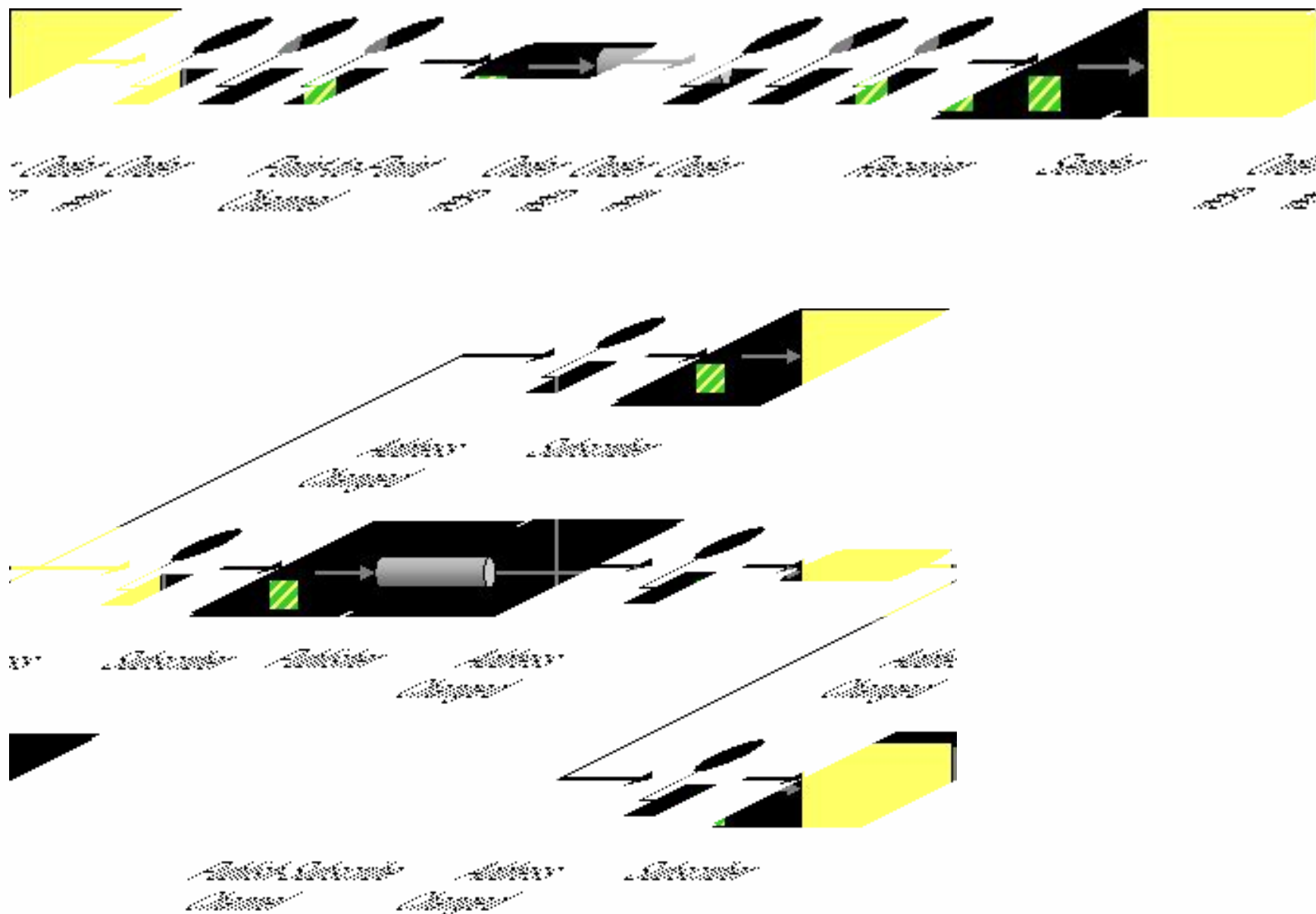
AddProduct, CreateOrder

**Событие** – это «факт» - сообщение о том, что что-то произошло, и требование от другого сервиса отсутствует:

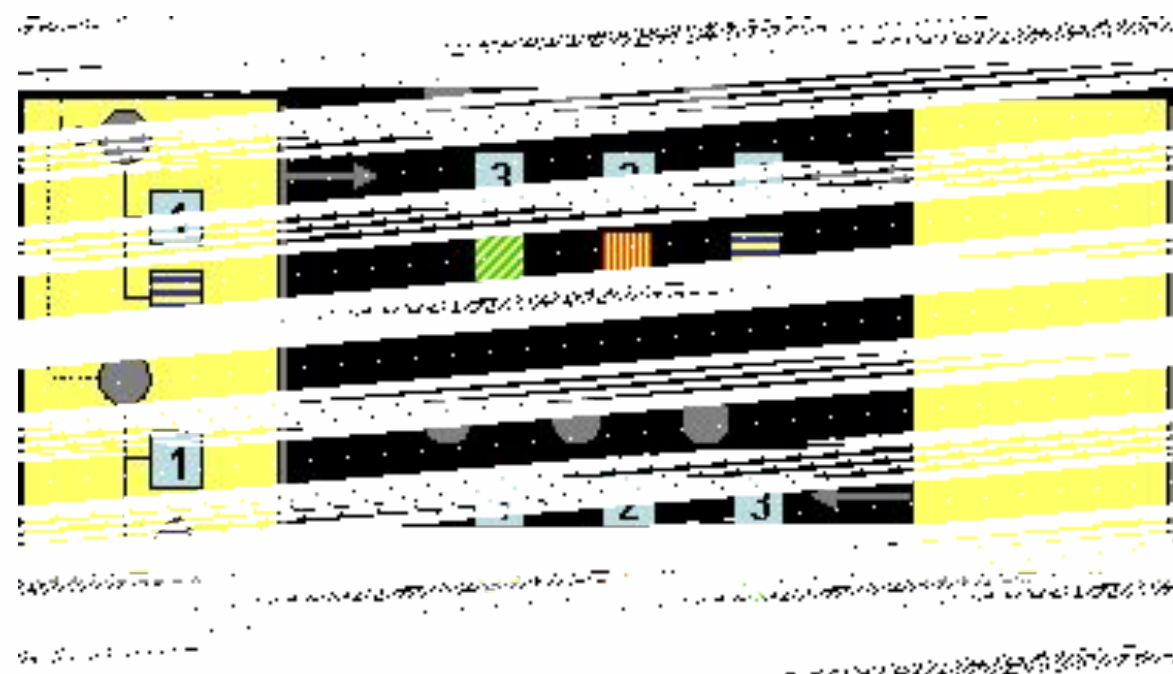
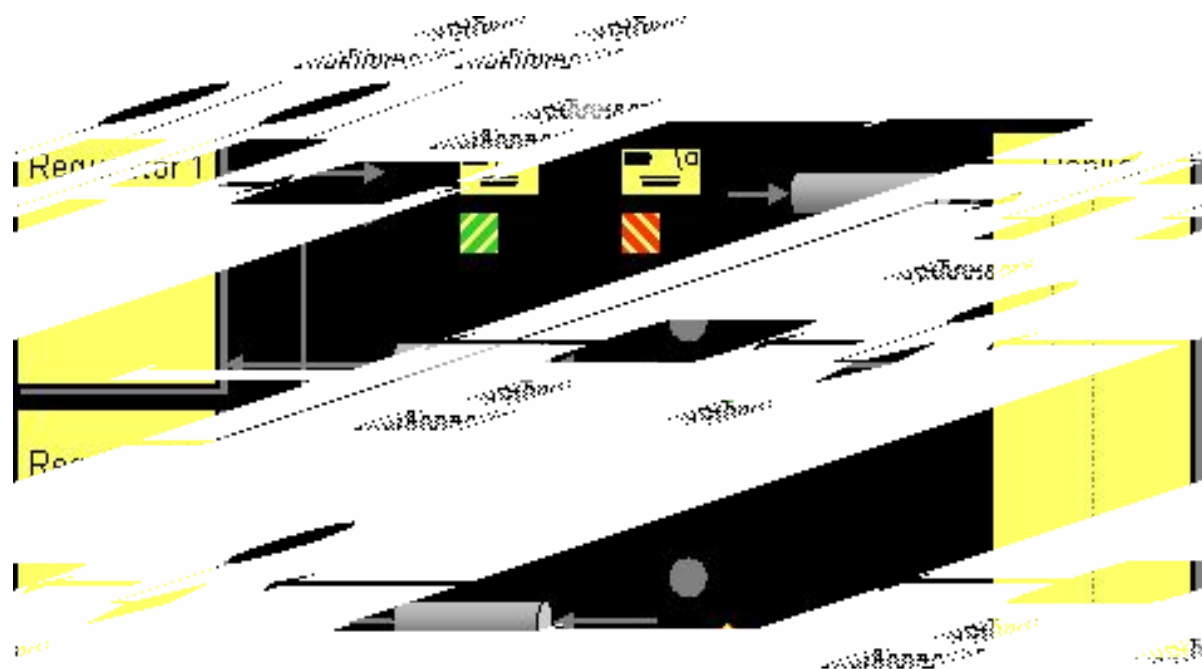
ProductAdded, OrderCreated



# Каналы



# RPC через сообщения



# Что в теле?

- JSON
- XML
- Protobuf
- Thrift
- Avro

```
message Person {  
  required string name = 1;  
  required int32 id = 2;  
  optional string email = 3;  
}
```

```
struct ListBonks {  
  1: list<Bonk> bonk  
}  
struct NestedListsBonk {  
  1: list<list<list<Bonk>>> bonk  
}  
  
struct BoolTest {  
  1: optional bool b = true;  
  2: optional string s = "true";  
}
```

```
{  
  "namespace": "example.avro",  
  "type": "record",  
  "name": "User",  
  "fields": [  
    {"name": "name", "type": "string"},  
    {"name": "favorite_number", "type": ["int", "null"]},  
    {"name": "favorite_color", "type": ["string", "null"]}  
  ]  
}
```

# Брокеры

- RabbitMQ
- Kafka
- ActiveMQ
- Mosquitto
- ...

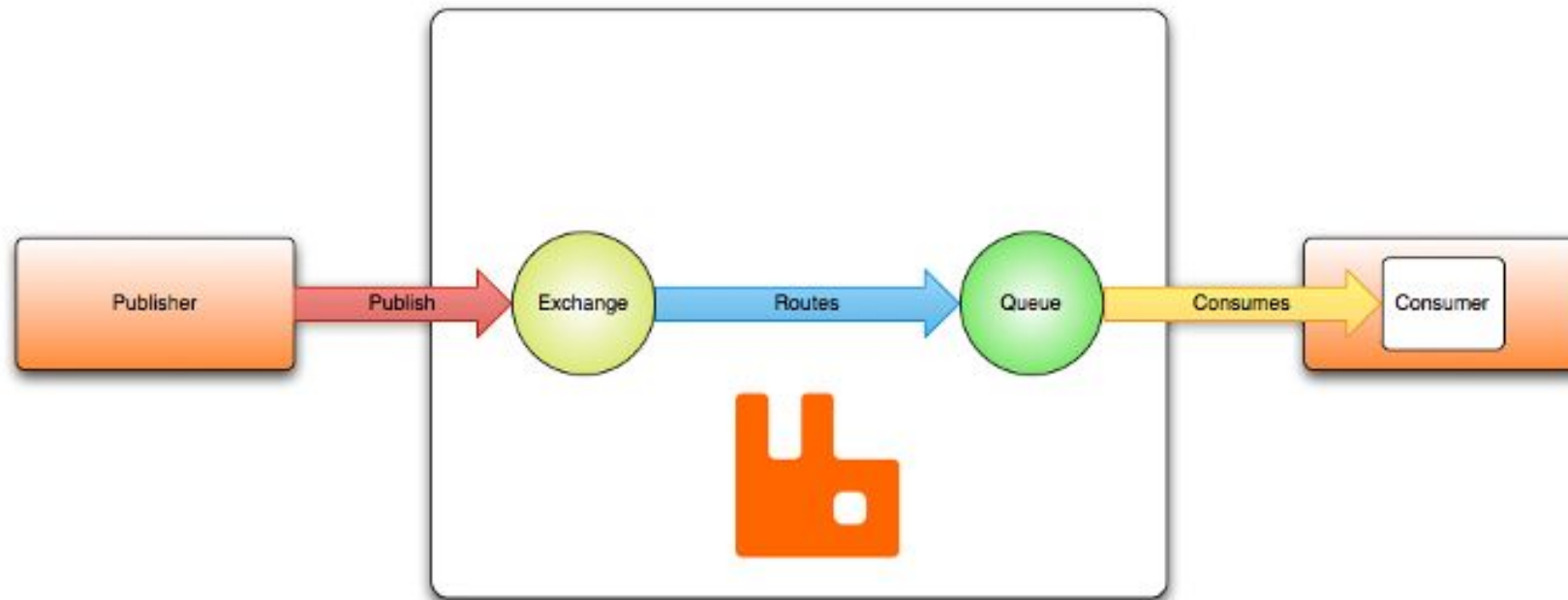
# 02

## RabbitMQ

# RabbitMQ

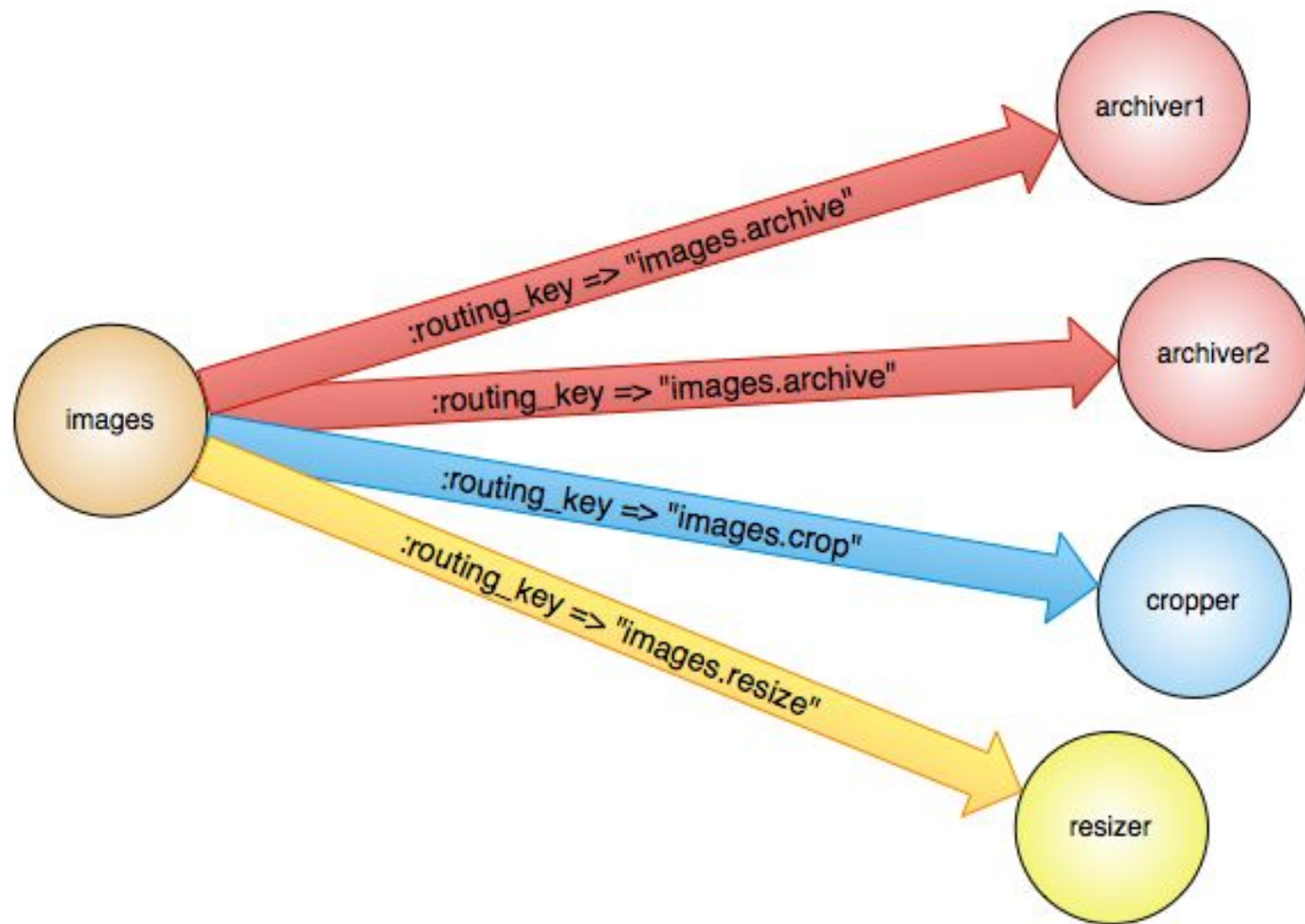
- Exchange - direct, fanout, topic, headers - сюда пишет продюсер (или несколько)
- Queue - отсюда читает консьюмер (или несколько). Может сохраняться на диск
- Bind - связывает Exchange и Queue

## "Hello, world" example routing

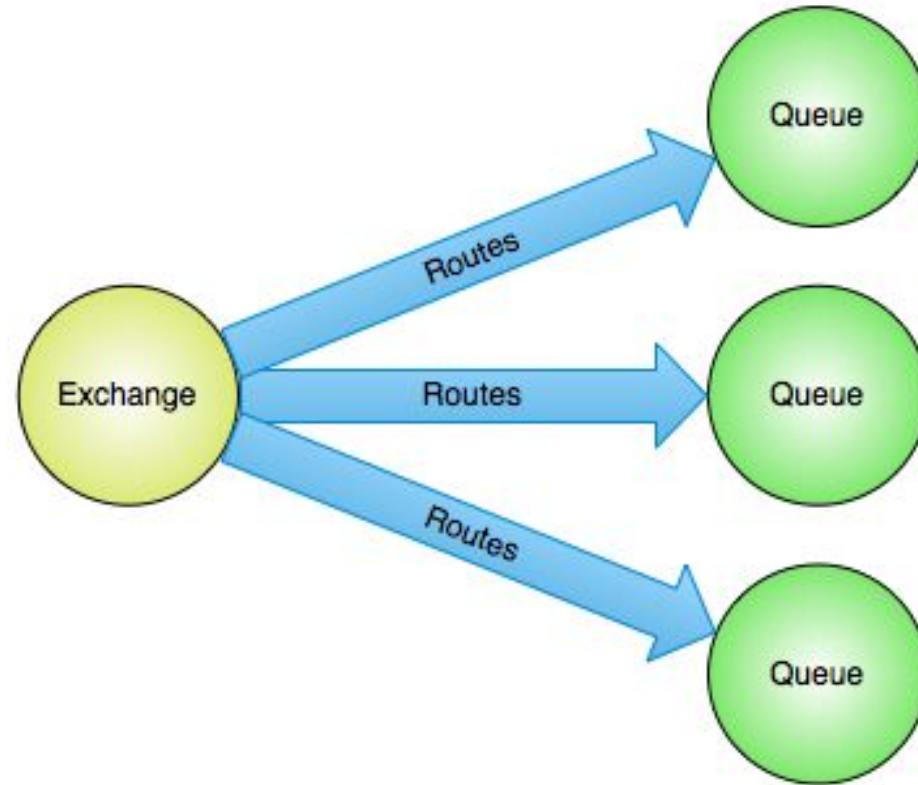




# Direct exchange routing



# Fanout exchange routing



# 03

События для межсервисного  
взаимодействия

# События и асинхронность

**События** – это сообщения о каком-то свершившемся факте.

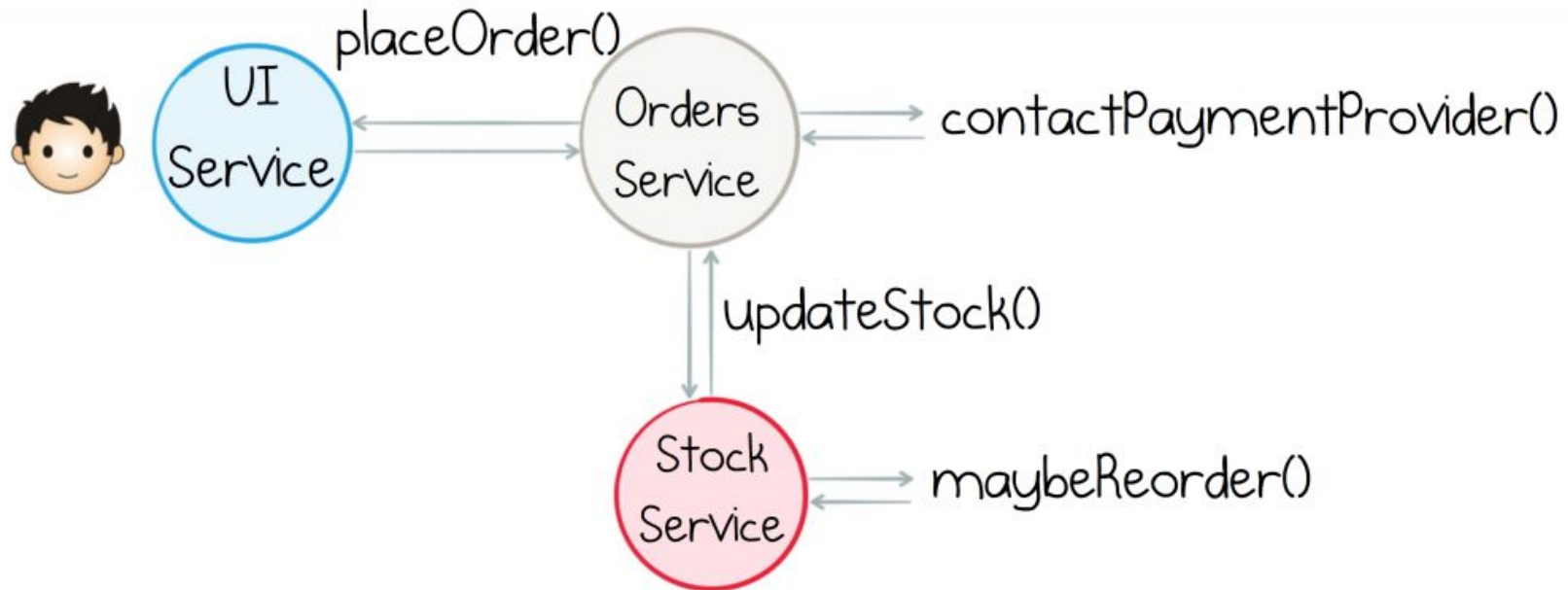
События являются семантически асинхронными: они сообщают о произошедшем факте и не требуют ответа.

Поэтому проектирование архитектуры на основе событийной модели позволяет по-настоящему использовать асинхронное взаимодействие и его плюсы.

- Надежность: падение сервисов не влияет друг на друга, и на общую доступность
- Масштабируемость: можно добавлять инстансы как потребителей, так и продюсеров.
- Независимость: продюсеры ничего не знают про потребителей и можно легко добавлять новых потребителей событий не меняя продюсеров.

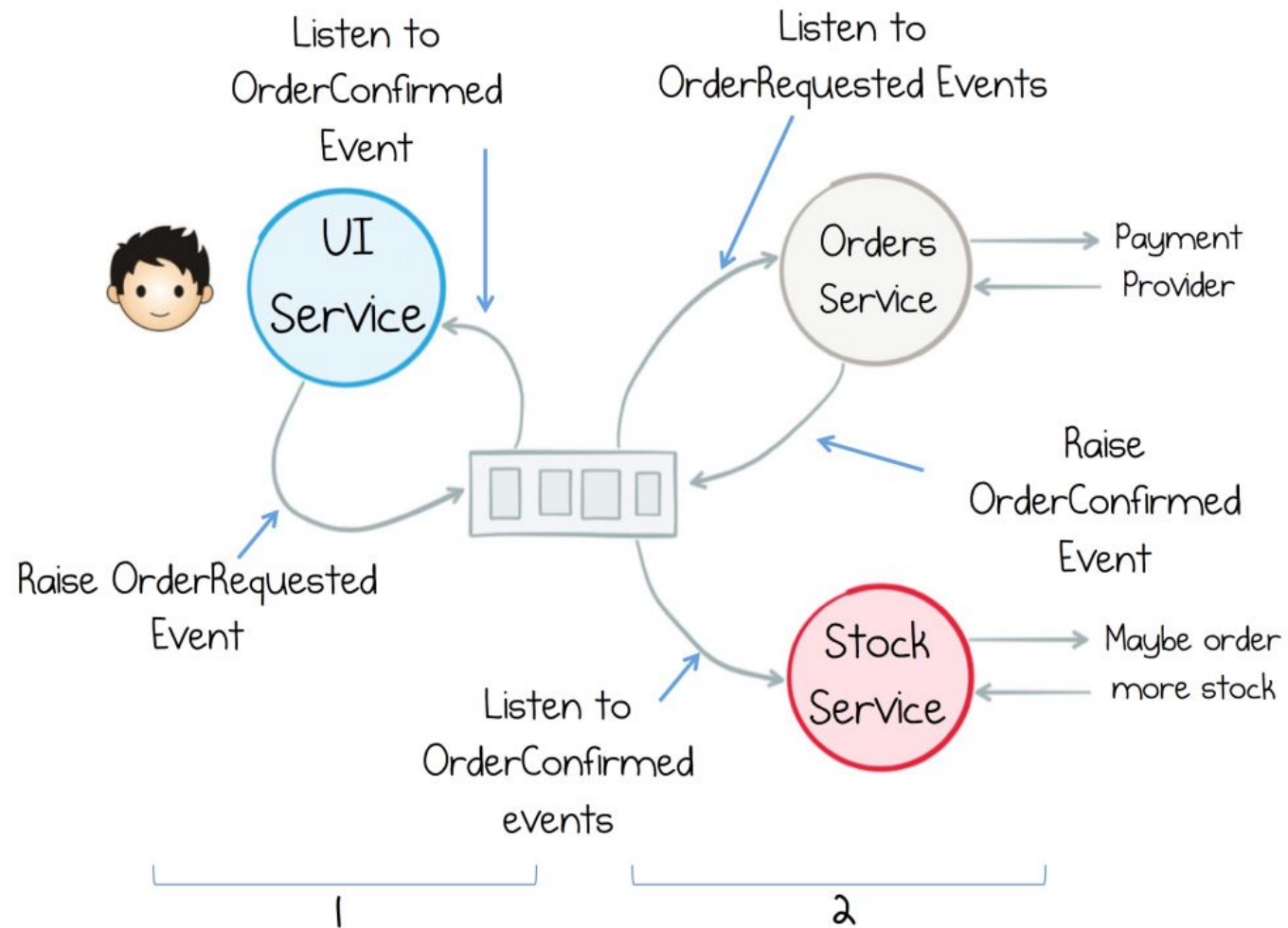
## Request-Reply архитектура

Например, пользователь нажал оформить заказ.  
Взаимодействие может происходить в синхронном режиме.



# События для нотификаций

События для нотификаций. В данном случае один сервис уведомляет другой о том, что событие произошло.



# События для нотификаций

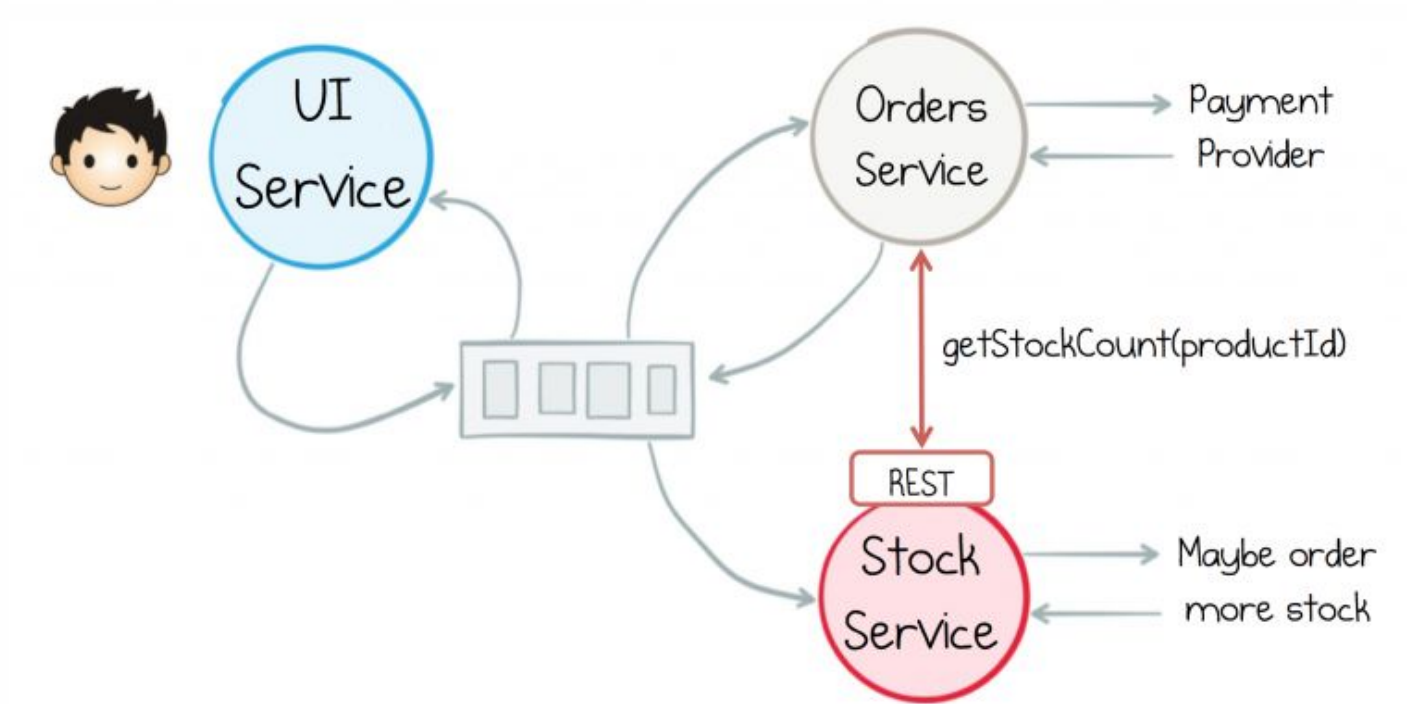
Использование событий для нотификаций других сервисов

- позволяет развязать логику (low coupling) между сервисами
- Может привести к тому, что флоу действий станет неявным. Чтобы понять, что происходит, будет очень сложно.

Антипаттерном является «пассивно-агрессивная» семантика: т.е. когда сервис, который отправляет событие ждет, что кто-то определенным образом его обработает.

## События для репликации данных

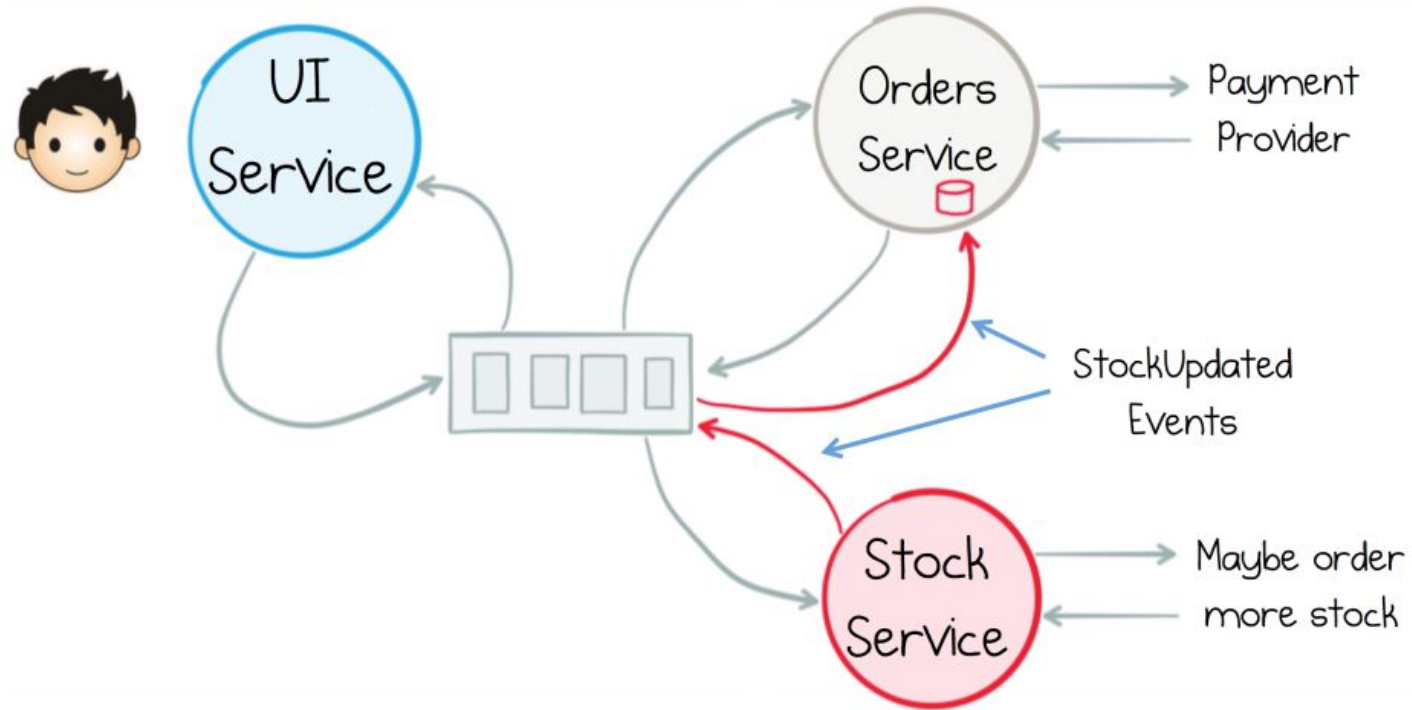
События могут использоваться не только для нотификаций, но и для переноса (миграции) данных, что снижает количество синхронных запросов.





## События для репликации данных

Сервис доставки присылает события, и эти события сервис заказа использует для того, чтобы сохранить локальный кэш данных, и не ходить в сервис доставки за этими данными.

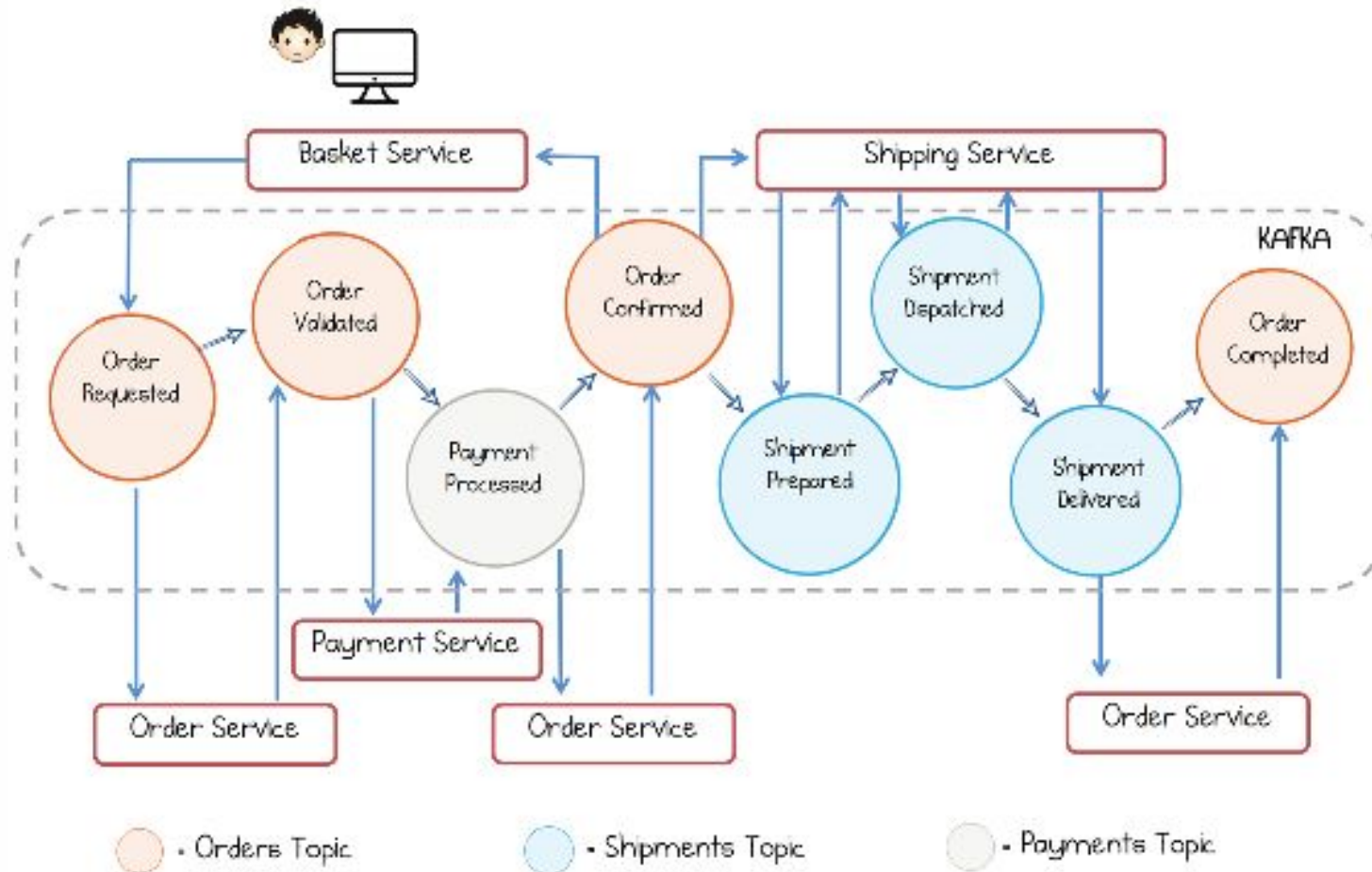


## Репликация данных с помощью событий

Репликация данных с помощью событий в чистом виде не всегда приводит к слабому связыванию между сервисами. Логика применения событий для изменения состояния в локальной базе может быть не очень простой и ее придется реализовывать во всех сервисах, где эти данные нужны. Это может быть значительно сложнее, чем просто сделать GET запрос.

# Event Collaboration

**Event Collaboration** – используем ТОЛЬКО события для межсервисного взаимодействия.



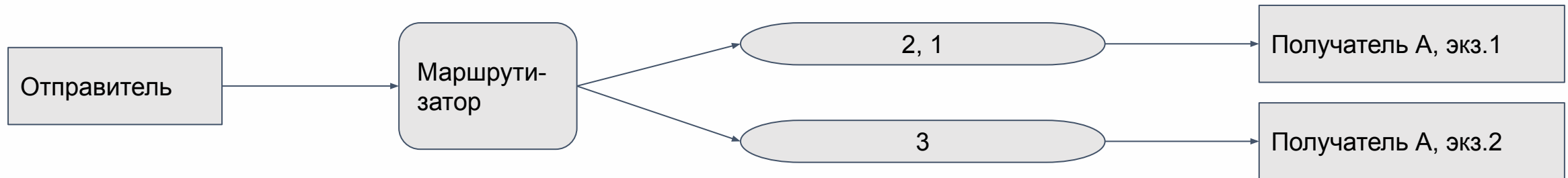
# Event Collaboration

Особенности:

- Для использования event collaboration крайне важна надежная доставка сообщений, сохранение порядка и дедупликация. Это довольно сложно реализовать.
- Использование паттернов хореографии для сложных бизнес процессов может приводить к их непрозрачности.

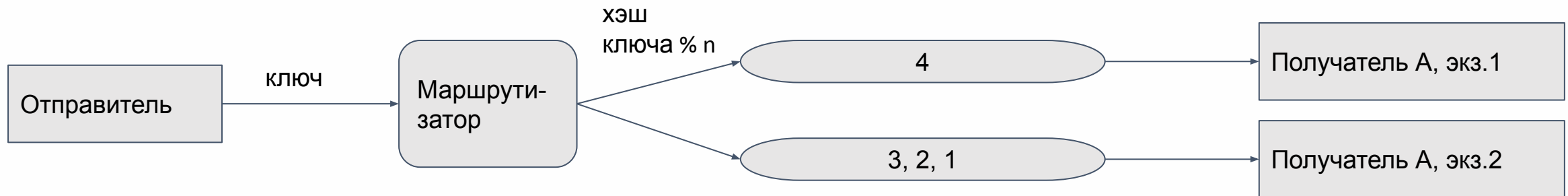
# Порядок следования событий

1. Заказ создан
2. Добавлен товар
3. Заказ отменен



## Порядок следования событий - сегментирование

1. Заказ 1 создан
2. Добавлен товар в заказ 1
3. Заказ 1 отменен
4. Заказ 2 создан

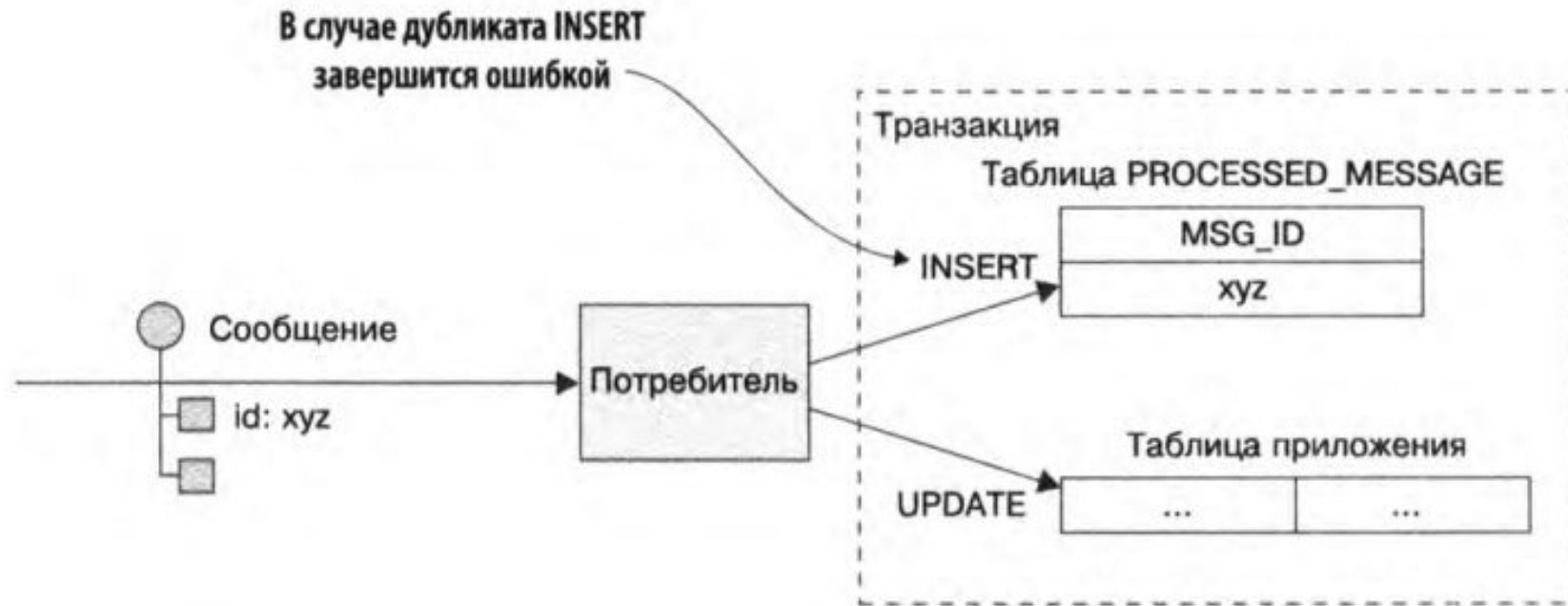


# Дублирование сообщений



1. Идемпотентные сообщения
2. Дедупликация

# Дублирование сообщений - дедупликация





# 04

## Паттерны проектирования событий

## Domain Events

Под событиями обычно подразумеваются **события предметной области** – т.е. факты о том, что случилось с сущностями предметной области.

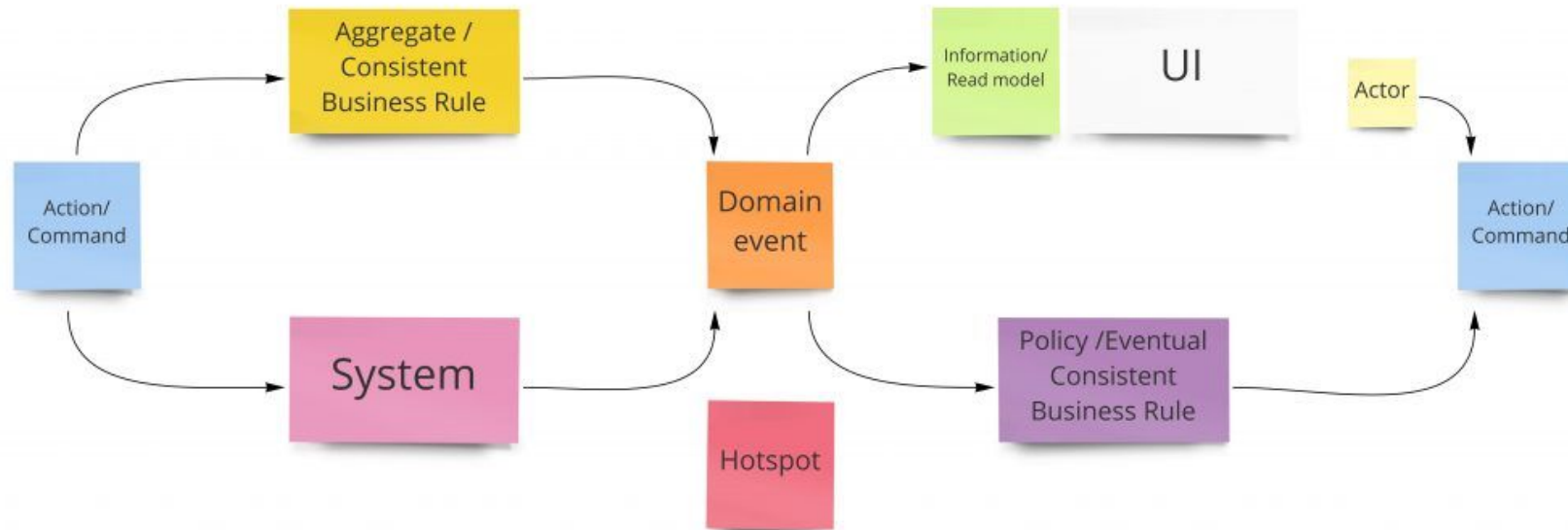
## С чего начинать проектирование событий?

- Стоит отталкиваться от предметной области.
- Начинать с основного пользовательского сценария.
- Каждое событие стоит связывать с предметной областью
- На первых итерациях лучше не думать про реализацию на бекенде.

# Event storming

Для выработки событий может использоваться Event Storming.

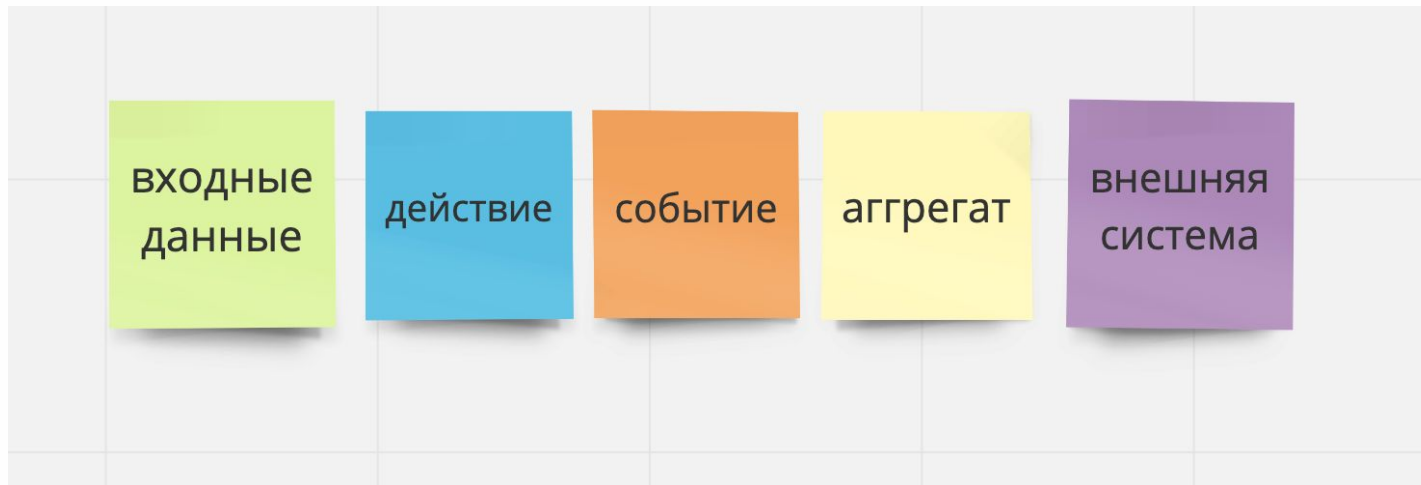
- Представляем бизнес процесс в виде набора действий, запросов, событий, акторов, агрегатов и внешних систем.



<https://www.eventstorming.com/book/>

# Event storming

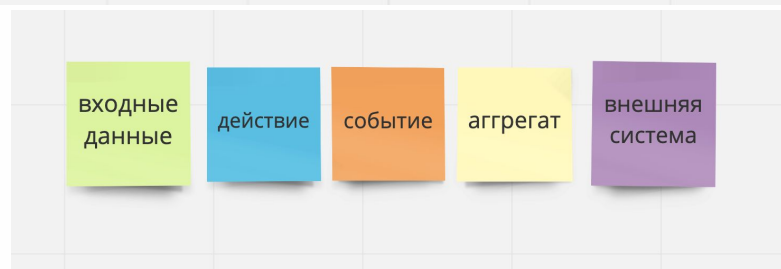
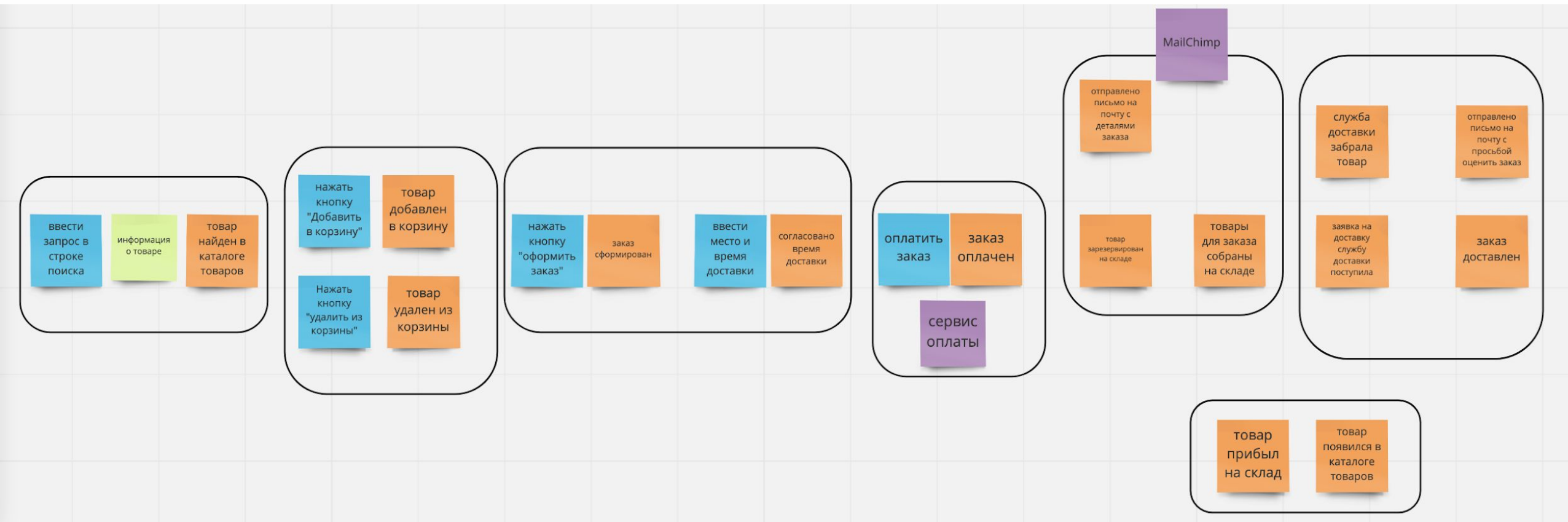
Отражаем на доске бизнес процесс в виде набора стикеров



<https://www.eventstorming.com/book/>

# Event storming

[https://miro.com/app/board/o9J\\_kvo2IAo=](https://miro.com/app/board/o9J_kvo2IAo=/)



## DataCentric Events

Точно также как и в обычном API есть желание использовать датацентричные события. Например, посылать наружу прямо изменения в БД

Т.е. события в таком паттерне выглядят как  
ObjectCreated/ObjectUpdated.

В этом случае часто теряется выразительная сила API.

## DataCentric Events. Пример.

При покупке продукта сервис отправляет такое событие. В базе конечная цена не хранится, а только цена + скидка. И это в таком же виде отправляется

ProductPurchased

```
{  
  "price": "10.5",  
  "discount": "0.5"  
}
```

Клиенты на своей стороне вычисляют настоящую цену, как  $\text{price} - \text{discount}$ .



## DataCentric Events. Пример.

Вдруг оказалось, что на некоторых рынках нужно еще считать и налоги, соответственно, добавилось поле taxes

ProductPurchased

```
{  
  "price": "10.5",  
  "discount": "0.5",  
  "taxes": "2.0"  
}
```

Т.е. на самом деле клиент должен заплатить price – discount + taxes.

И всем, кто читает это событие, придется переписать свою логику.

## DataCentric Events. Пример.

Конечно, же имело смысл сразу в событие писать поле, которые вычислялось в сервисе

ProductPurchased

```
{  
  "price": "10.5",  
  "discount": "0.5",  
  "totalPrice": "10.0"  
}
```

В этом случае, при изменении расчета, не придется править тех, кто читает эти события.

# Проектирование событий

Status поменялся с active на blocked у пользователя.  
Какое событие отправлять?

- UserUpdated?
- UserBlocked?
- UserStatusChanged?
- Или может быть все вместе?

Что отправлять в payload такого события?

- Только изменение?
- Полностью новое состояние?
- Только id?

## Проектирование событий

Блокирование пользователя приводит к блокированию всех его кампаний и баннеров. Сервис, который блокирует кампании и баннеры пользователя должен отослать одно событие? Или несколько по количеству объектов?

## Проектирование событий

На все эти вопросы нет однозначного ответа. Также как нет однозначного ответа на то, каким должен быть REST интерфейс.

Должен быть метод PATCH /api/v1/users/{userid}

Или POST /api/v1/users/{userId}/block?

Или POST /api/v1/users/update?

И т.д.

## Events Antipatterns

Отправка нескольких событий на одно изменение, со специализированным Payload-ом для каждого типа консьюмера. Или одного события, но с полями специально подобранным для консьюмера.

Такие вещи связывают сервисы, и заставляют их знать друг про друга. И является антипаттерном.

## Single writer pattern

Наличие нескольких сервисов, которые отправляют одинаковый тип событий тоже не очень хорошо. Т.к. при изменении формата события, придется править одновременно два сервиса.

**Single-writer pattern** – только один сервис имеет право создавать события об определенном процессе

# 05

## Event Sourcing



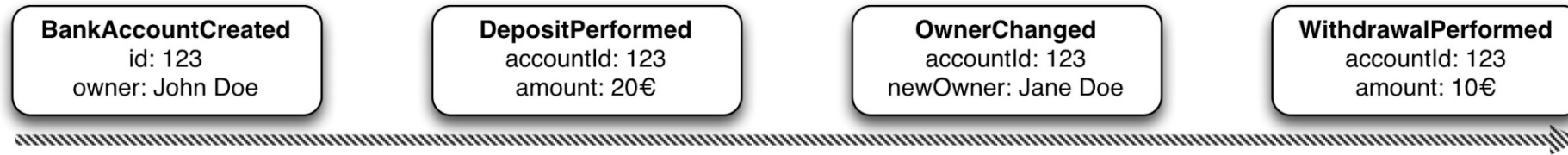
# Event Sourcing

А почему бы в базе хранить не состояние объекта, а просто набор событий?

- Банковский счет – это производная от набора транзакций  
 $\text{balance} = \text{sum}(\text{transaction.value})$ , а не наоборот
- Любое состояние – это производная от набора фактов
- Факты или события – не меняются, а то, как мы представляем данные – меняется. На самом деле мы храним факты, а не представление.

# Event Sourcing

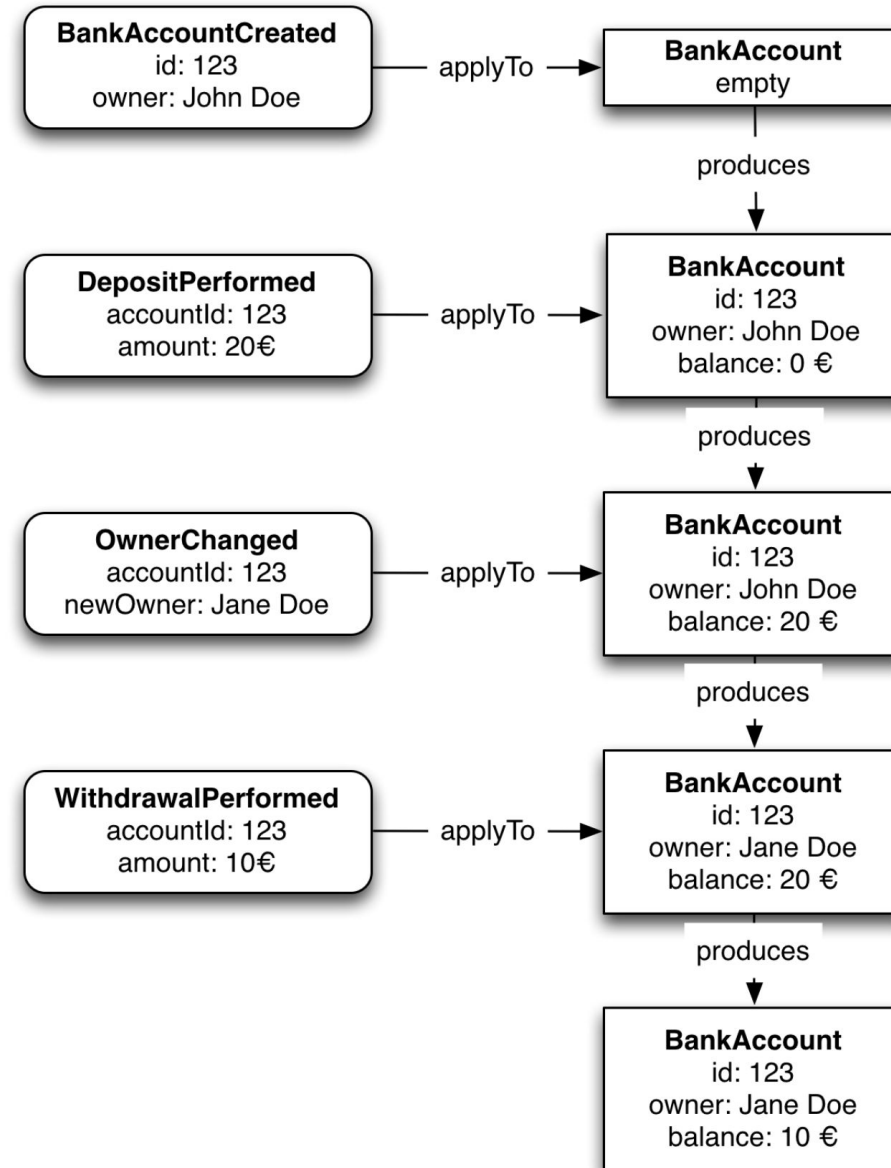
Пусть будет набор событий во времени:



# Event Sourcing

Если мы проиграем во времени все события, получится конечное состояние.

state -> func (init(), events)

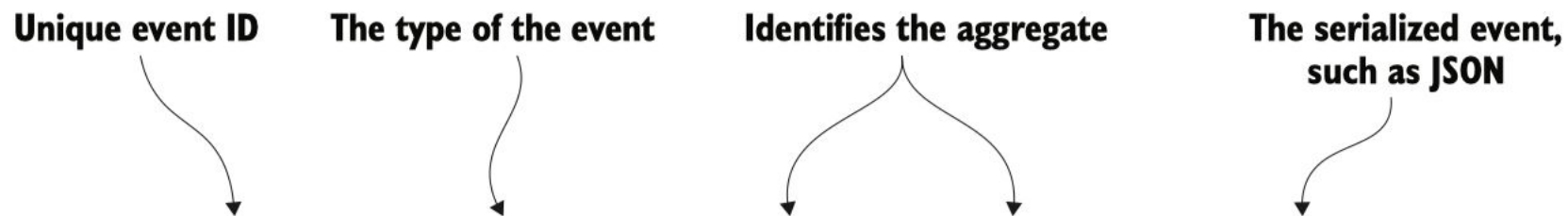


# Что может дать Event Sourcing?

- Аудит лог за бесплатно
- Крутые средства для дебага
- Не теряешь данных и можешь делать какие угодно проекции (“машина времени”)

# Event Sourcing

- Например, можно хранить вот в такой табличке:



event_id	event_type	entity_type	entity_id	event_data
102	Order Created	Order	101	{...}
103	Order Approved	Order	101	{...}
104	Order Shipped	Order	101	{...}
105	Order Delivered	Order	101	{...}
...	...	...	...	...

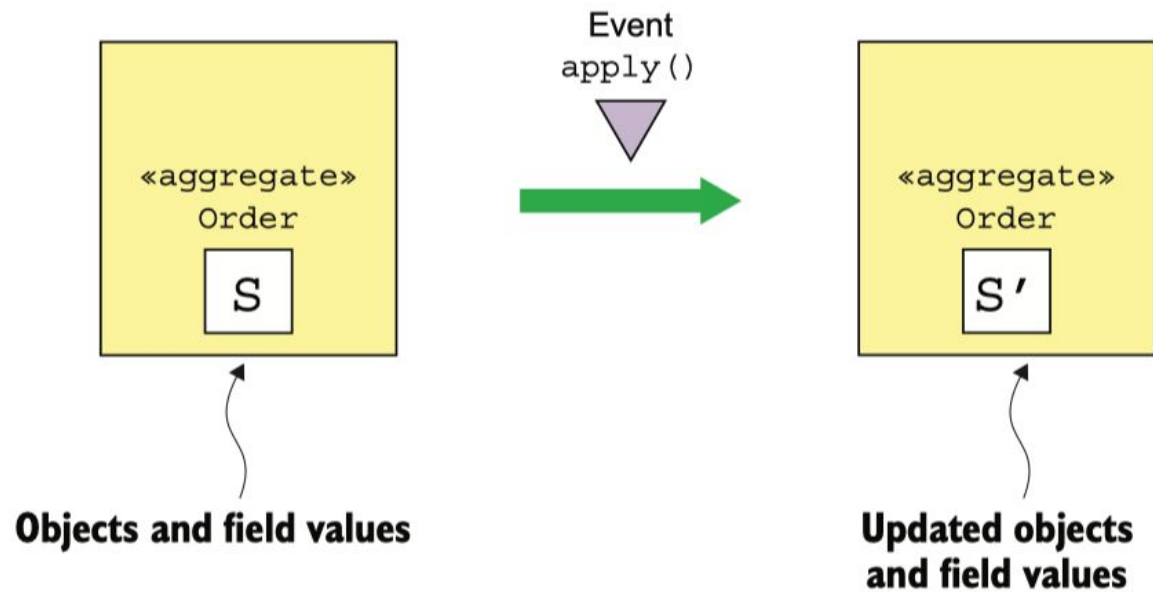
EVENTS table

# Event Sourcing

Как получить текущее состояние объекта (агрегата)?

- Последовательно применяем события к объекту и меняем его состояние

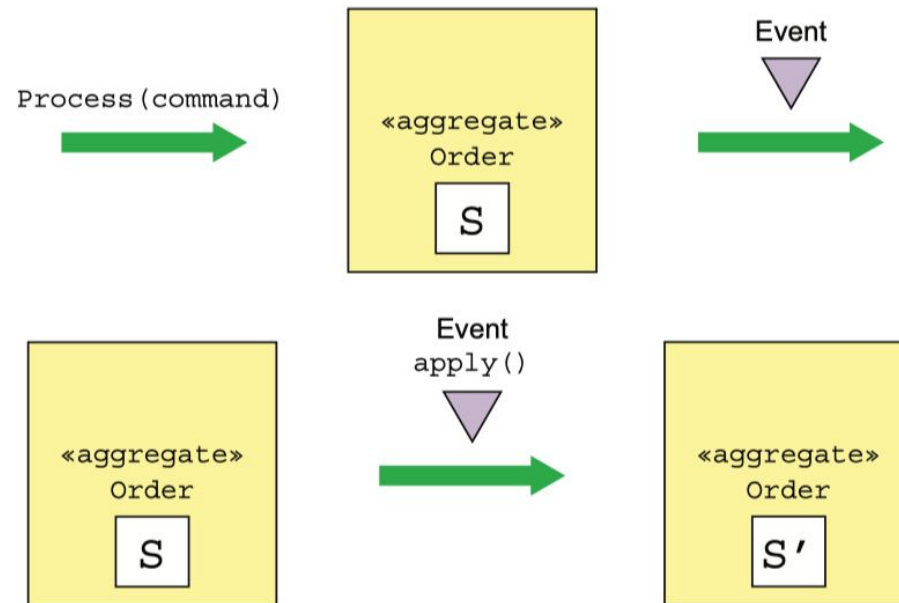
Поскольку события – единственный источник состояния в Event Sourcing, то Event должен иметь все(!) необходимые данные для того, чтобы можно было вычислить новое состояние (объекта).



# Команды, события и применение

В мире Event Sourcing есть:

- **События** – что-то, что случилось в прошлом
- **Команды** - не изменяют состояния, возвращают события  
`func command(state, events) -> event(s)'`
- **Применения** – изменяют состояние  
`func apply(state, events) -> state'`



## Event Sourcing – меняет способ написания кода

Крайне важно, чтобы изменение состояния не происходило вне событийной модели.

Функции или методы классов, которые меняют состояния

```
func f(state, arguments) -> state'
```

должны быть в явном виде разделены на команды и применения

```
func f_command(state, arguments) -> events
```

```
func f_apply(state, events) -> state'
```

При этом функции, которые не трогают состояние, можно не трогать (инфраструктурный слой или утилиты, например)



## Event Sourcing – меняет способ написания кода

- Любое изменение состояния происходит только, как результат применения события

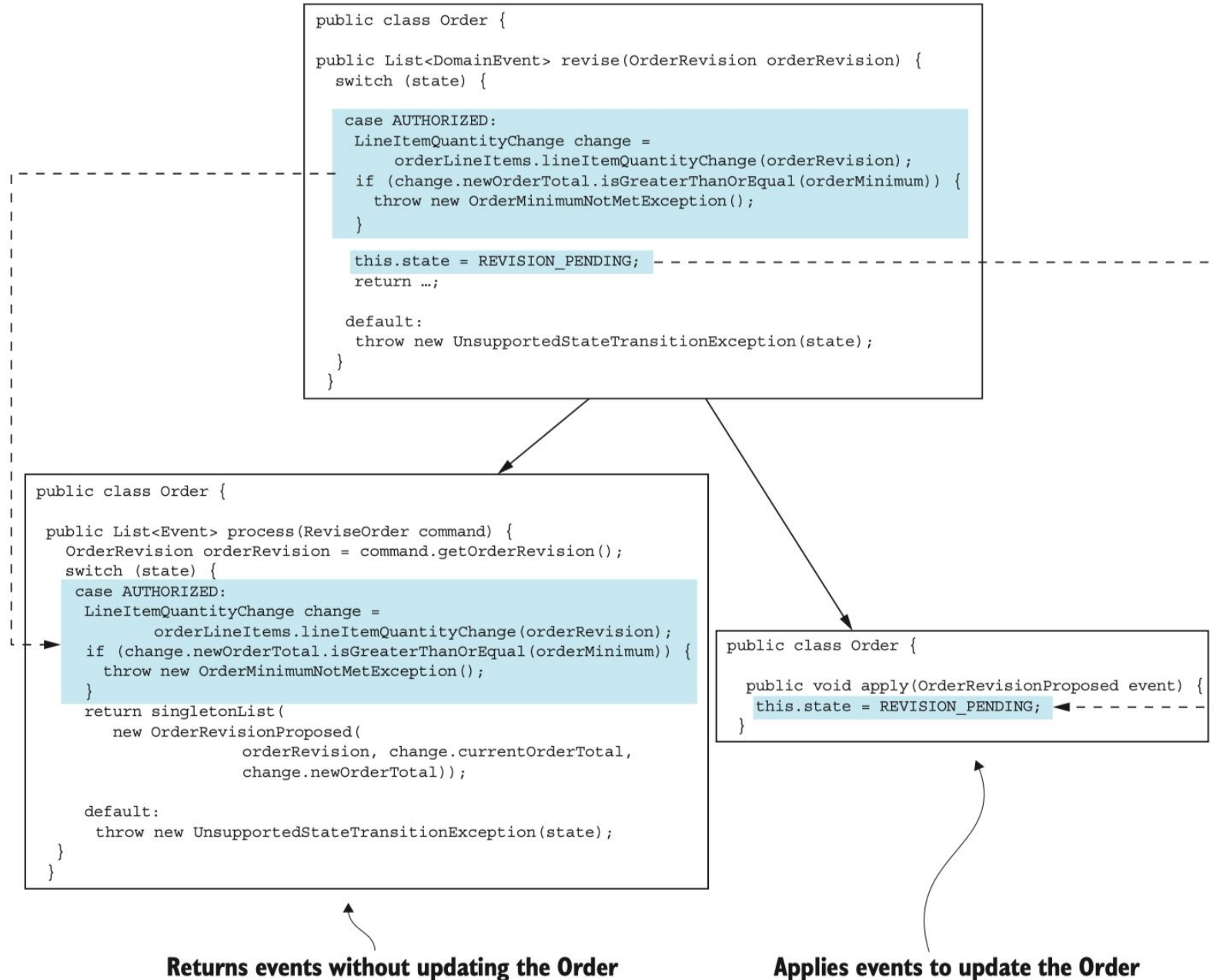
### Раньше:

```
class Account():  
    def change_name(self, new_name):  
        self.validate_name(new_name)  
        self.name = new_name
```

### Event Sourcing:

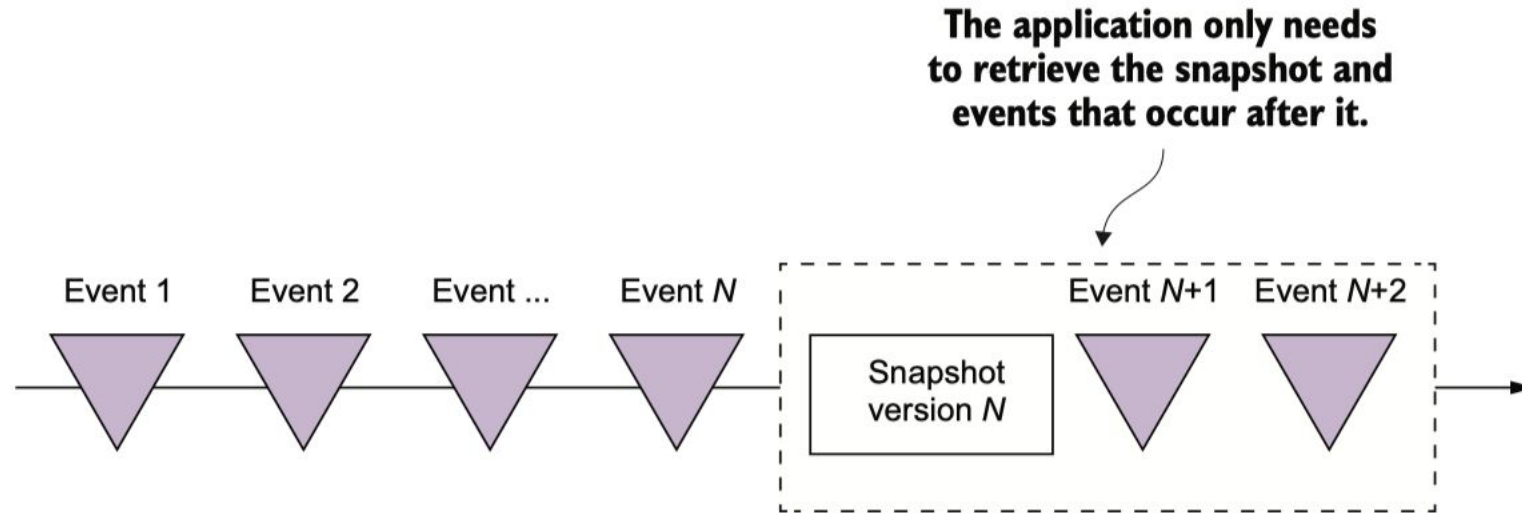
```
class Account():  
    def change_name(self, new_name):  
        self.validate_name(new_name)  
        self.raise(AccountNameChanged(new_name), apply=True)  
  
    @on(AccountNameChanged)  
    def apply(self, e):  
        self.name = e.new_name
```

# Event Sourcing – меняет способ написания кода



# Снапшоты

Время от времени для производительности, приходится делать снапшоты



EVENTS

entity_id	event_type	event_type	entity_id	event_data
...	...	...	...	...
103	...	Customer	101	{...}
104	Credit Reserved	Customer	101	{...}
105	Address Changed	Customer	101	{...}
106	Credit Reserved	Customer	101	{...}

SNAPSHOTS

entity_id	event_type	entity_id	snapshot_data
...	...	...	...
103	Customer	101	{name: "...", ...}
...	...	...	...
...	...	...	...

## Event Sourcing - миграции

- агрегаты
- перечень событий
- состав каждого события

Уровень	Изменение	Обратно совместимое
Структура	Определение нового типа агрегата	Да
Удаление агрегата	Удаление существующего агрегата	Нет
Переименование агрегата	Изменение названия типа агрегата	Нет
Агрегат	Добавление нового типа событий	Да
Удаление события	Удаление типа событий	Нет
Переименование события	Изменение названия типа событий	Нет
Событие	Добавление нового поля	Да
Удаление поля	Удаление поля	Нет
Переименование поля	Переименование поля	Нет
Изменение типа поля	Изменение типа поля	Нет

## Event Sourcing - удаление части данных

Сидоров Иван Петрович, [sidorov@mail.ru](mailto:sidorov@mail.ru)

- создан заказ
- изменен получатель
- оставлен комментарий
- товар оценен
- изменен email

Как забыть



- Хранить в событии зашифрованные данные
- Ключ хранить отдельно
- Удалить ключ

## Event Sourcing - паттерн уровня хранения данных

- Event Sourcing – это не паттерн уровня архитектуры всей системы, а паттерн уровня хранения.
- У нас архитектура основана на Event Sourcing» == «у нас архитектура основана на RDBMS»
- Один event store на всю компанию == Один postgres на всю компанию
- Совершенно нормально иметь в одном микросервисе ES/CQRS, а в другом RDBMS.

# Event Sourcing – плюсы и минусы



## Плюсы

- Бесплатный аудит-лог
- Производительность event-store-ов обычно хорошая
- Можно делать темпоральные запросы
- Можно строить какую угодно аналитику, т.к. храним все данные

## Минусы

- Миграции данных делать тяжело
- Рефакторинг и изменение событийной модели может быть сложно
- Крутая кривая обучения из-за того, что меняется стиль программирования
- Возможное дублирование кода
- Делать выборки (quering) сложно (без CQRS паттерна использовать крайне тяжело)

## А как делать выборки?

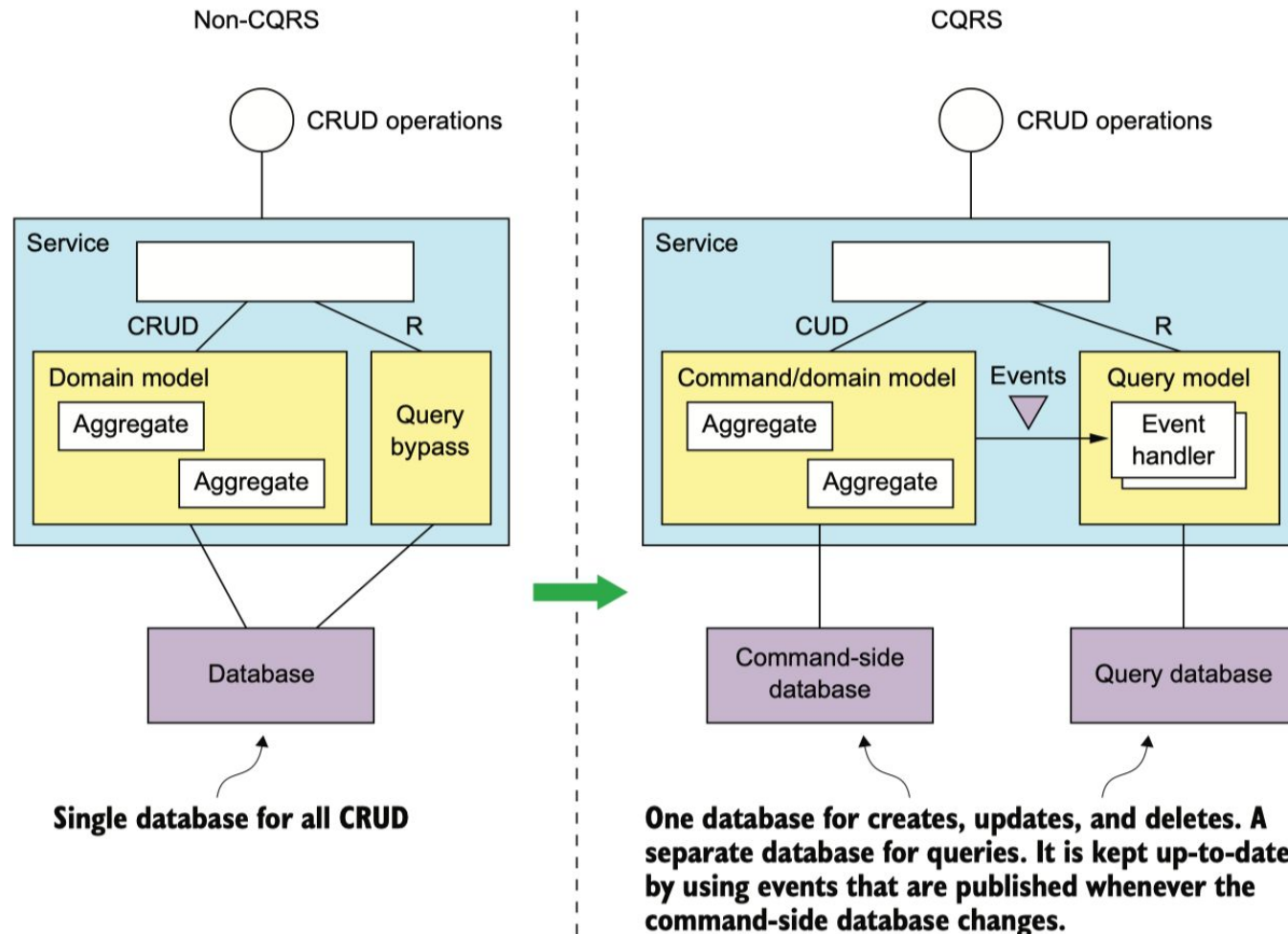
Сложные выборки из EventStore делать сложно. Поэтому на помощь приходит паттерн CQRS



# CQRS

## CQRS – Command/Query Responsibility Segregation

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



## Event Sourcing – пример pet project

- <https://github.com/gregoryyoung/m-r> – пример реализации Event Sourcing от его создателя Грегга Янга на C#

## Event Sourcing библиотеки и хранилища

- EventStore – хранилище от Янга
- Axon - <https://axoniq.io/>
- Eventuate <http://eventuate.io/>

# 06

## Рефлексия

# Цели занятия

- рассмотреть основы Event Driven Architecture;
- рассмотреть взаимодействие на основе событий;
- научиться проектировать события
- рассмотреть подход Event Sourcing

# Опрос

<https://otus.ru/polls/40823/>

## Домашнее задание - через 2 занятия, но можно начинать

Реализовать сервис заказа. Сервис биллинга. Сервис уведомлений.

- При создании пользователя, необходимо создавать аккаунт в сервисе биллинга. В сервисе биллинга должна быть возможность положить деньги на аккаунт и снять деньги.
- Сервис уведомлений 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 2 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 использованием 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...

**Спасибо  
за внимание!**

