

DevFest

Event Sourcing

Как перестать беспокоиться и
начать хранить события



Evgeny Shigaev

Codderz

@eashigaev



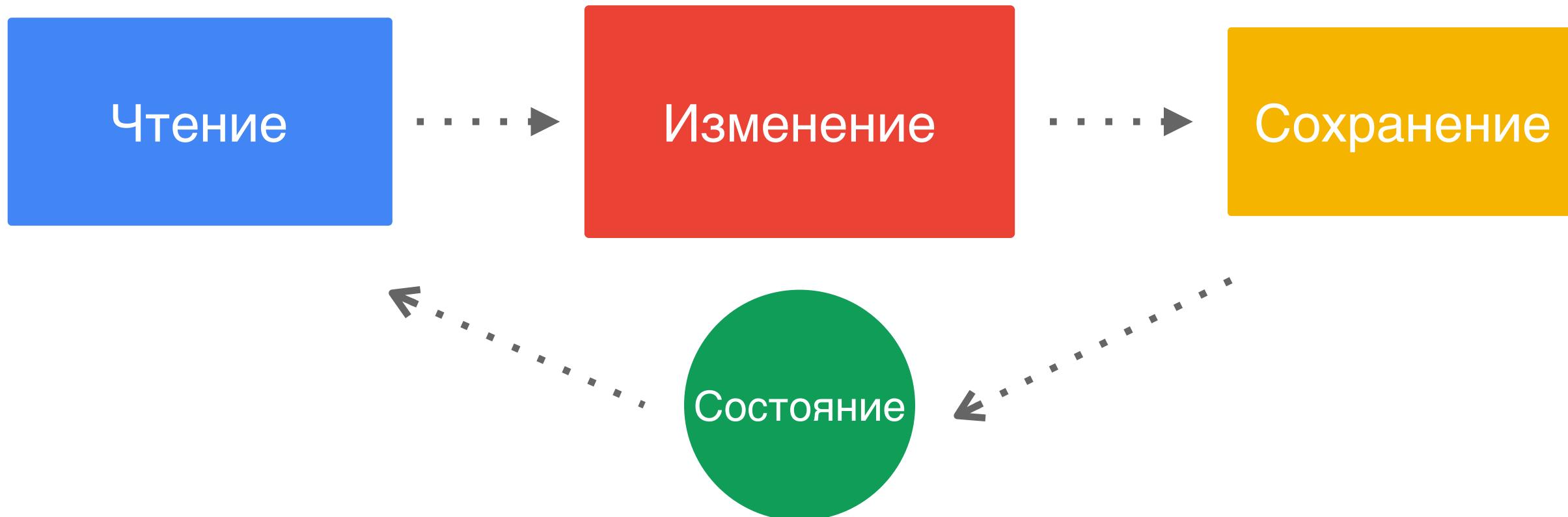
Event Sourcing

- Что это?
- Зачем это?
- Я правда перестану беспокоиться?

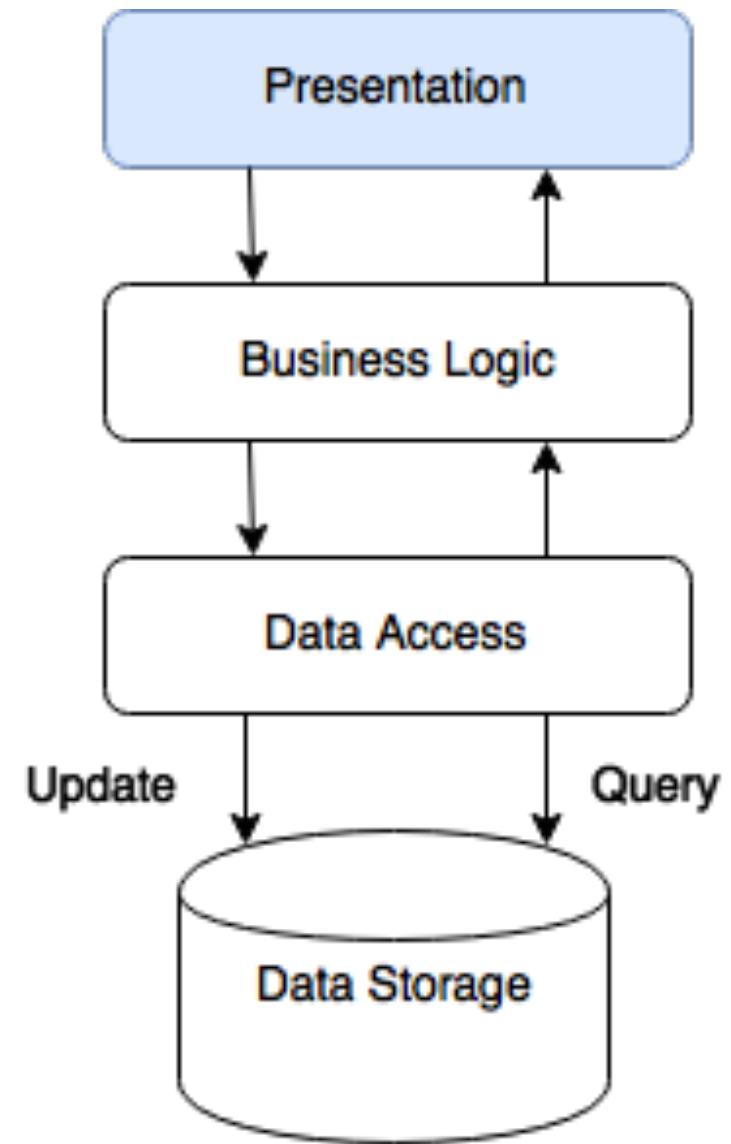


Состояние

Типичный подход, используемый в приложении, — поддержка текущего состояния данных за счет обновления по мере работы с ними.



Просто, зато в цветах Гугла



CRUD

Event Sourcing

- Не сохраняем текущее **состояние** объектов
- Сохраняем **события**, которые привели к этому состоянию

EVENT

Что-то, произошедшее в прошлом

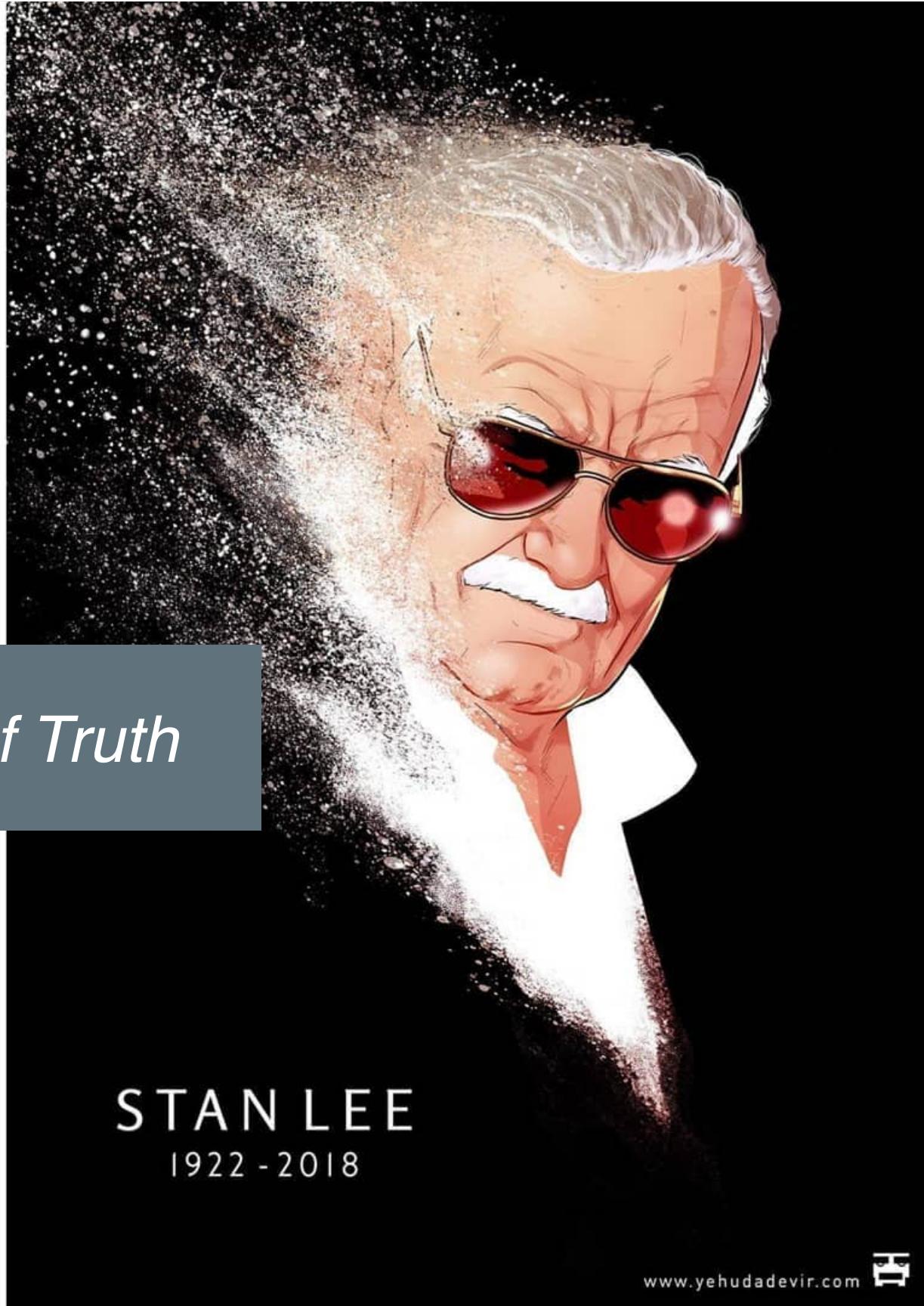
Свершившийся факт. Используется для принятия решений в других частях системы.

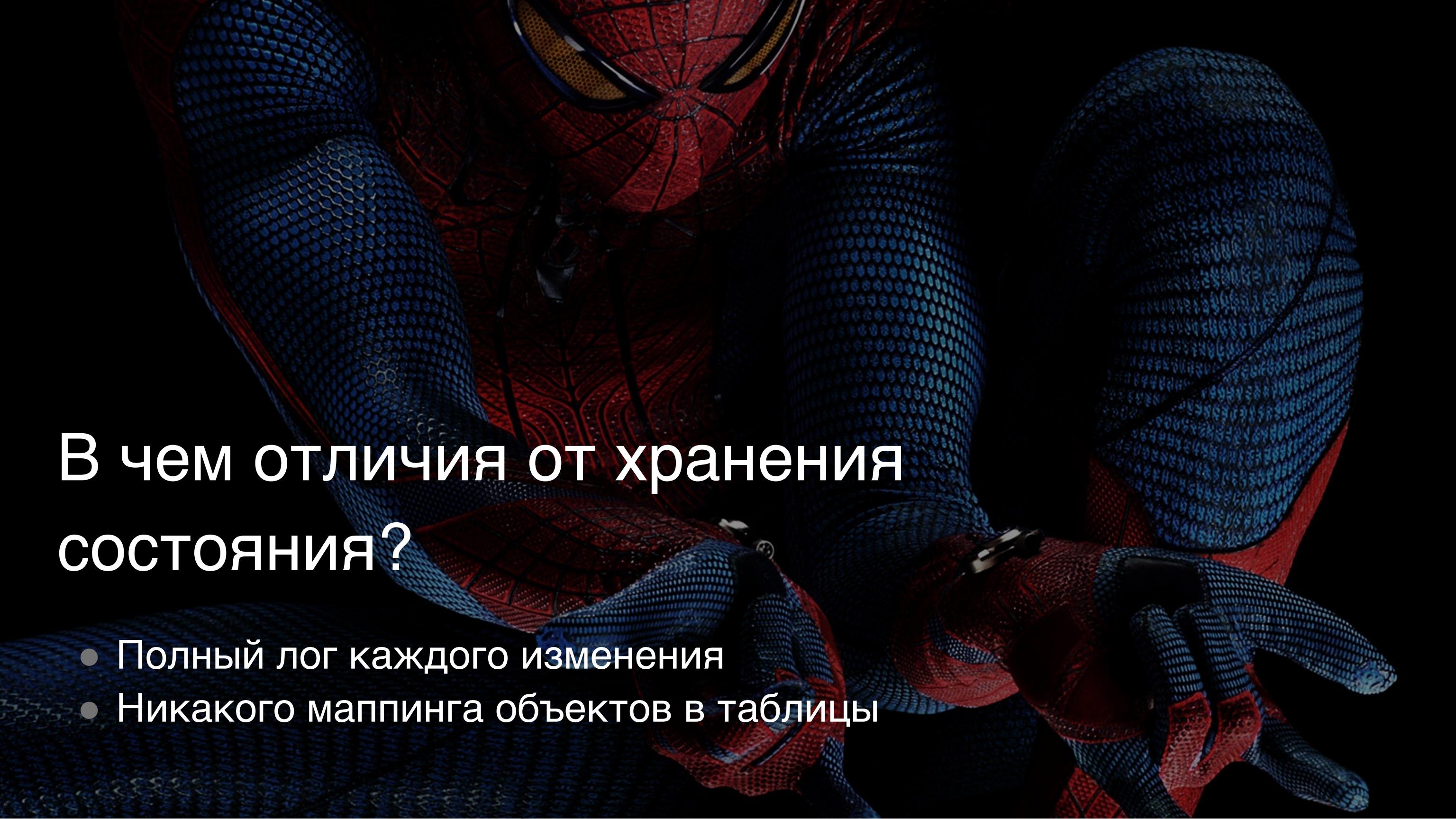
События неизменны.

Сохраняются в хранилище (**Event Store**)

История неудачных проектных решений

Source of Truth



A close-up, low-angle shot of Spider-Man's suit, focusing on the red and blue patterned fabric and the black webbing across the chest and shoulders. The mask is partially visible, showing the iconic spider emblem. The lighting highlights the texture of the suit material.

В чем отличия от хранения состояния?

- Полный лог каждого изменения
- Никакого маппинга объектов в таблицы

MAPPING

Преобразование данных

Маппинг объектов в базу данных и
наоборот. Много времени и усилий.
«Неожиданно» непросто.

The Addison-Wesley Signature Series

PATTERNS OF ENTERPRISE APPLICATION ARCHITECTURE

MARTIN FOWLER

WITH CONTRIBUTIONS BY
DAVID RICE,
MATTHEW FOEMMEL,
EDWARD HIEATT,
ROBERT MEE, AND
RANDY STAFFORD



143 страницы про
mapping

A MARTIN FOWLER SIGNATURE BOOK
Kim



Uncle Bob Martin

@unclebobmartin

 Follow

The biggest problem with ORM's is that they don't really map O to R. Tables are not objects. They never were; and never will be.

 Reply

 Retweet

 Favorite

 More

126

RETWEETS

22

FAVORITES



7:12 AM - 30 Sep 13



Как это работает?

Сохранение объектов

BankAccountCreated
id: 123
owner: John Doe

DepositPerformed
accountId: 123
amount: 20€

OwnerChanged
accountId: 123
newOwner: Jane Doe

WithdrawalPerformed
accountId: 123
amount: 10€

- Создать **Event** для каждого изменения состояния объекта
- Сохранить этот поток событий (**Event Stream**), соблюдая очередь

Восстановление объектов

BankAccountCreated
id: 123
owner: John Doe

DepositPerformed
accountId: 123
amount: 20€

OwnerChanged
accountId: 123
newOwner: Jane Doe

WithdrawalPerformed
accountId: 123
amount: 10€

- Поочередно применить события из **Event Stream** к "пустому" экземпляру объекта

BankAccountCreated

id: 123

owner: John Doe

applyTo

BankAccount

empty

produces

BankAccount

id: 123

owner: John Doe

balance: 0 €

DepositPerformed
accountId: 123
amount: **20€**

applyTo →

BankAccount
id: 123
owner: John Doe
balance: **0 €**

produces

↓
BankAccount
id: 123
owner: John Doe
balance: **20 €**

OwnerChanged
accountId: 123
newOwner: **Jane Doe**

applyTo →

BankAccount
id: 123
owner: John Doe
balance: 20 €

produces

BankAccount
id: 123
owner: **Jane Doe**
balance: 20 €

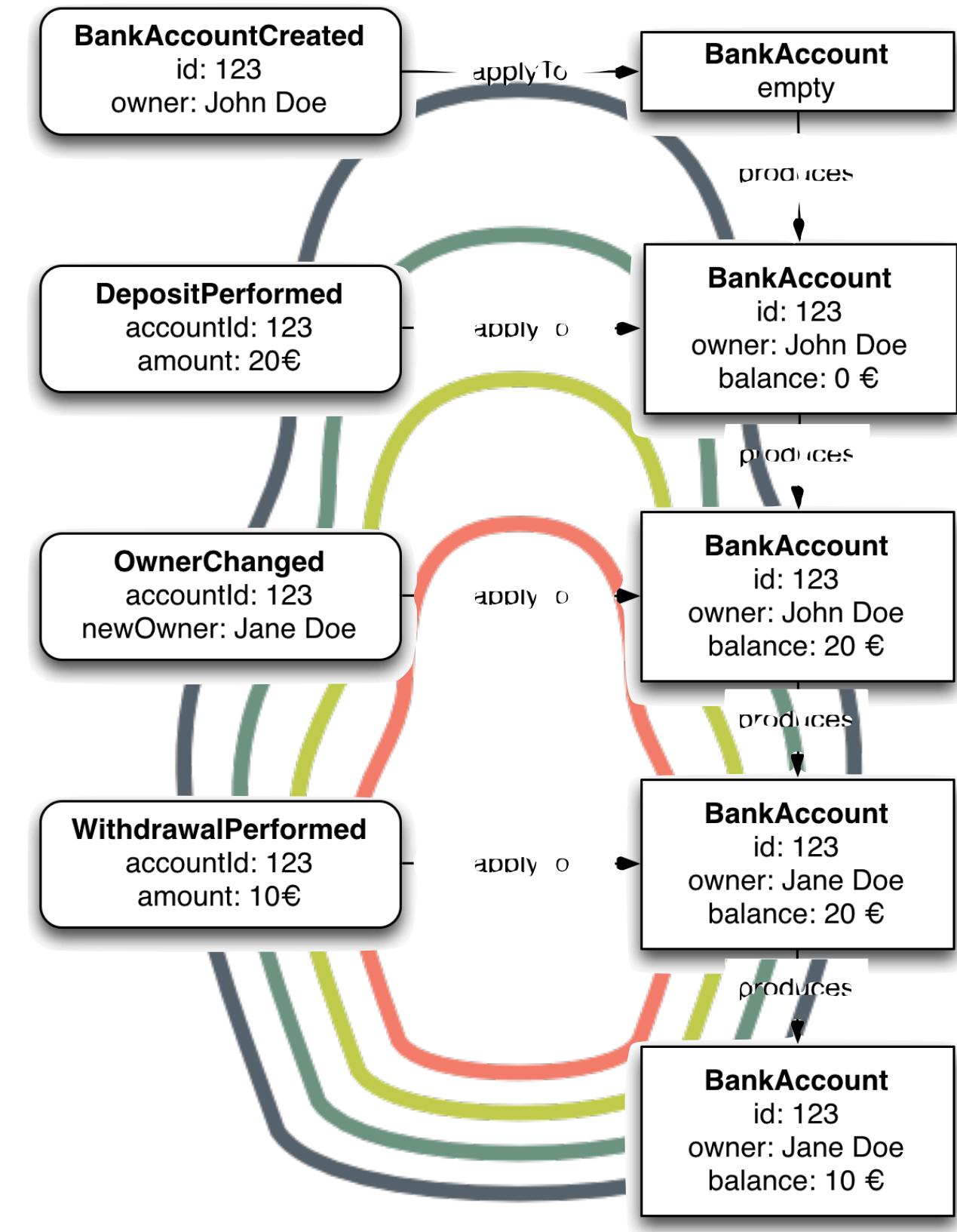
WithdrawalPerformed
accountId: 123
amount: **10€**

applyTo →

BankAccount
id: 123
owner: Jane Doe
balance: 20 €

produces

BankAccount
id: 123
owner: Jane Doe
balance: **10 €**



Обновление объектов

BankAccountCreated
id: 123
owner: John Doe

DepositPerformed
accountId: 123
amount: 20€

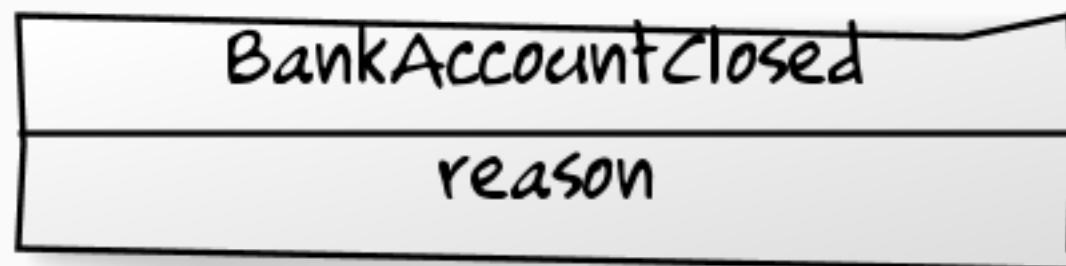
OwnerChanged
accountId: 123
newOwner: Jane Doe

WithdrawalPerformed
accountId: 123
amount: 10€

WithdrawalPerformed
accountId: 123
amount: 10€

```
const account = accountRepository.load(123)  
const modifiedAccount = account.withdraw(new Euro(10))  
accountRepository.save(modifiedAccount)
```

Удаление объектов

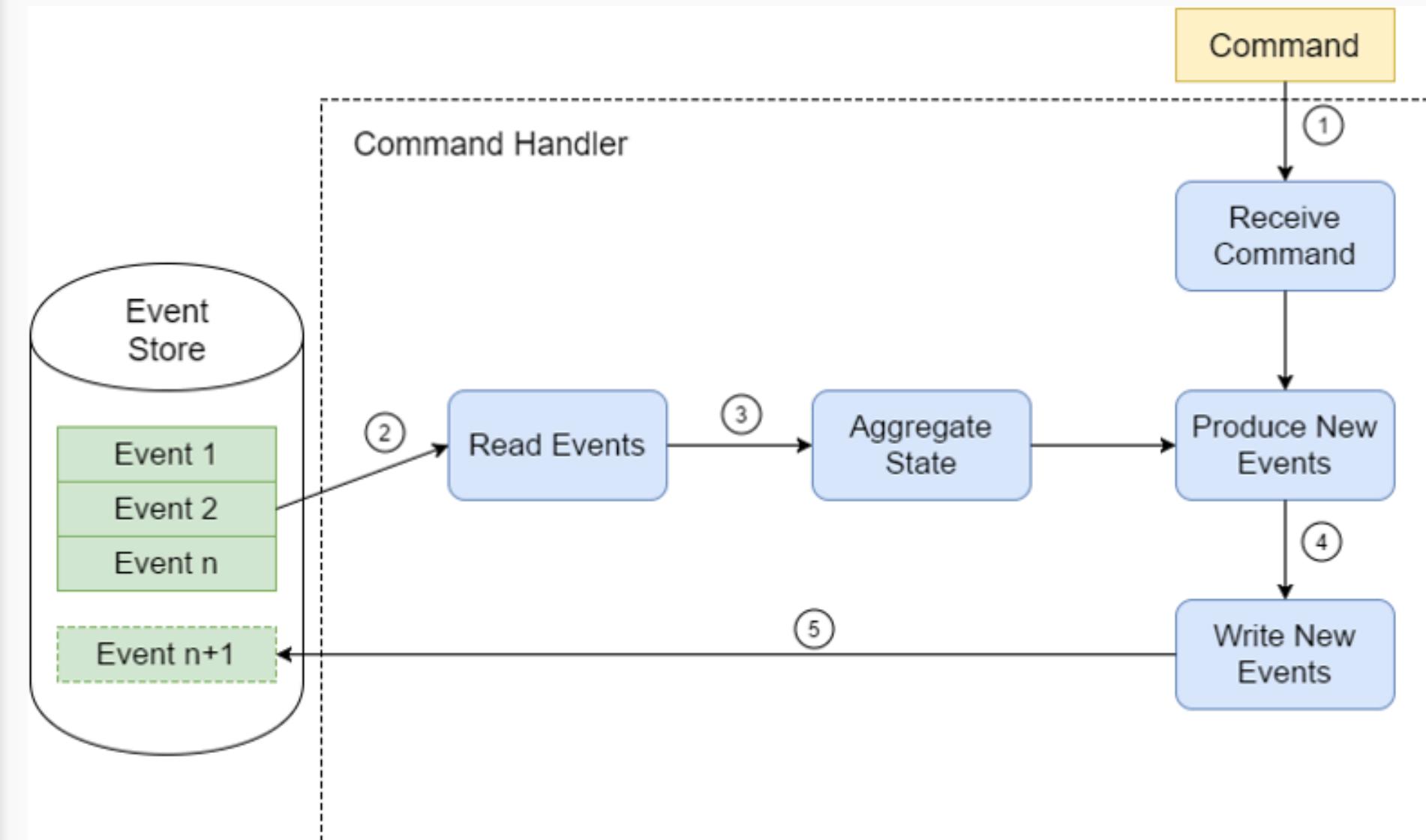


Retroactive Event

Событие, отменяющее что-то, произошедшее в прошлом

Схема приложения

1. Получаем команду из интерфейса
2. Запрашиваем необходимые события
3. Восстанавливаем состояние
4. Генерируем новые события
5. Сохраняем эти события



Практика

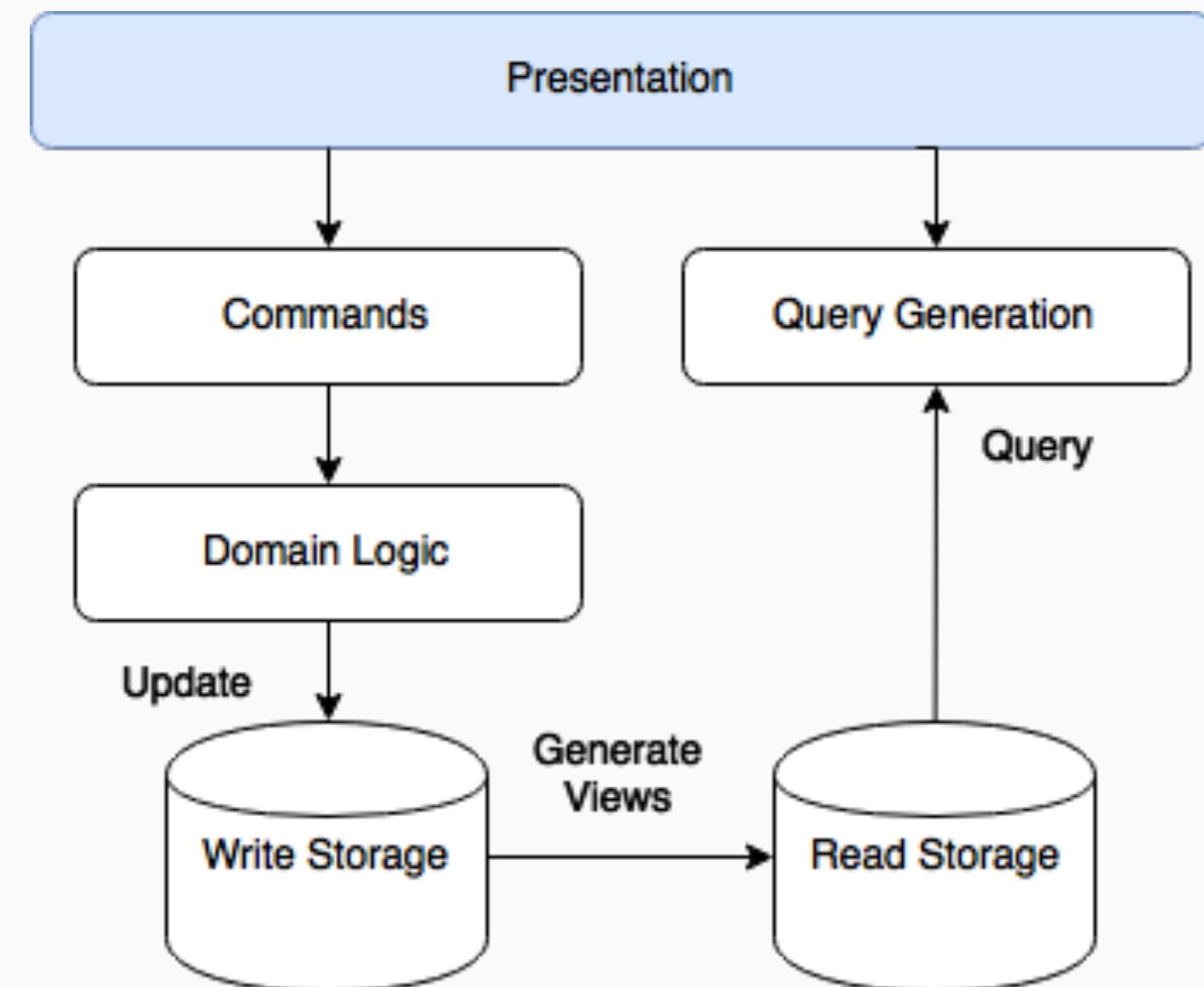


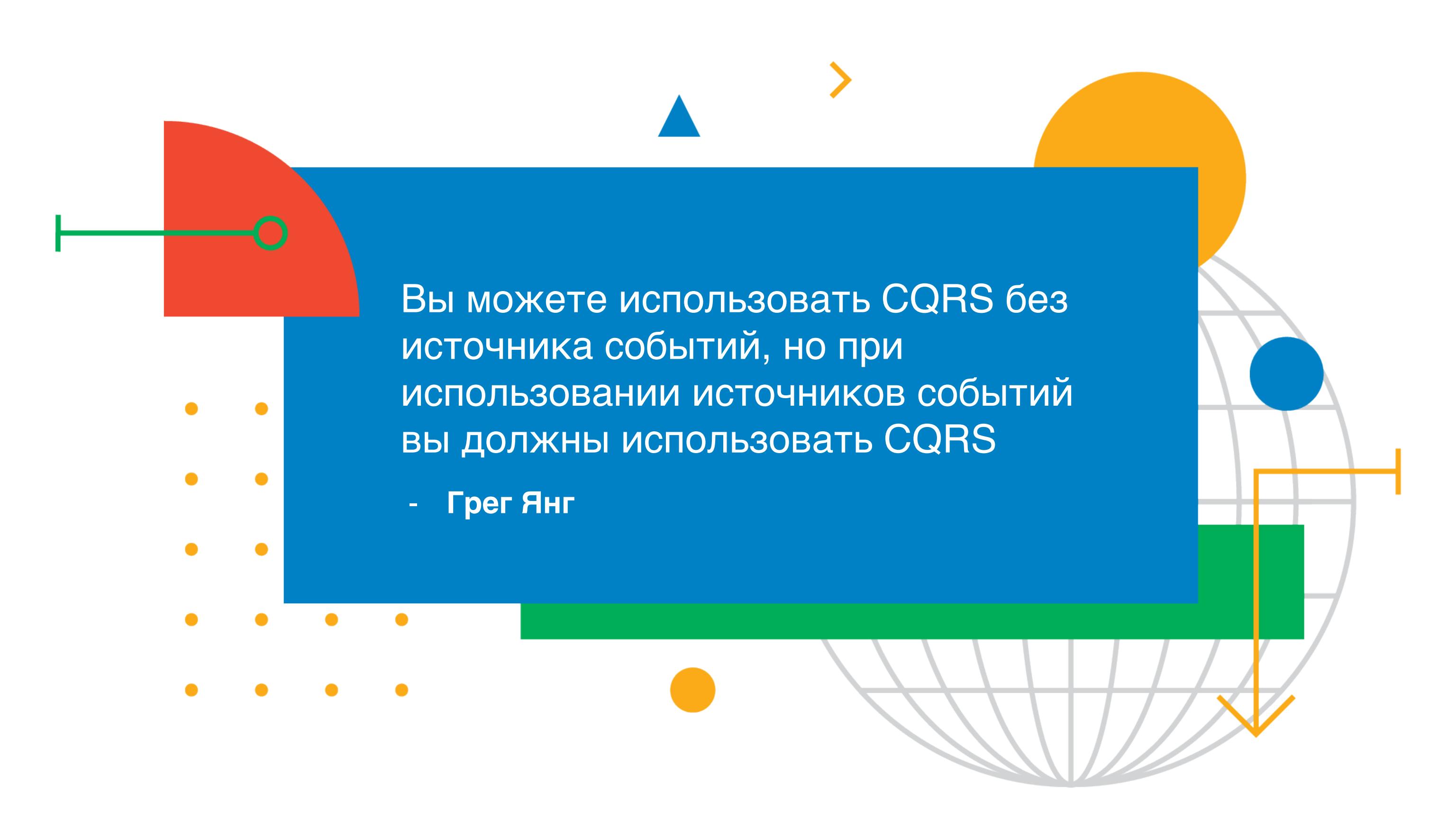
Запросы и отчеты

Как выполнить запрос «все счета, с балансом не менее N»? Неужели придется восстанавливать каждый счет? А если постранично?

CQRS

Command-query separation (CQS) или command-query responsibility segregation (CQRS) – принцип разделения команд и запросов. Метод должен быть либо командой (**Command**), выполняющей какое-то действие, либо запросом (**Query**), возвращающим данные, но не одновременно. Возвращать значение можно только чистым, не имеющим побочных эффектов методам.





Вы можете использовать CQRS без источника событий, но при использовании источников событий вы должны использовать CQRS

- Грег Янг

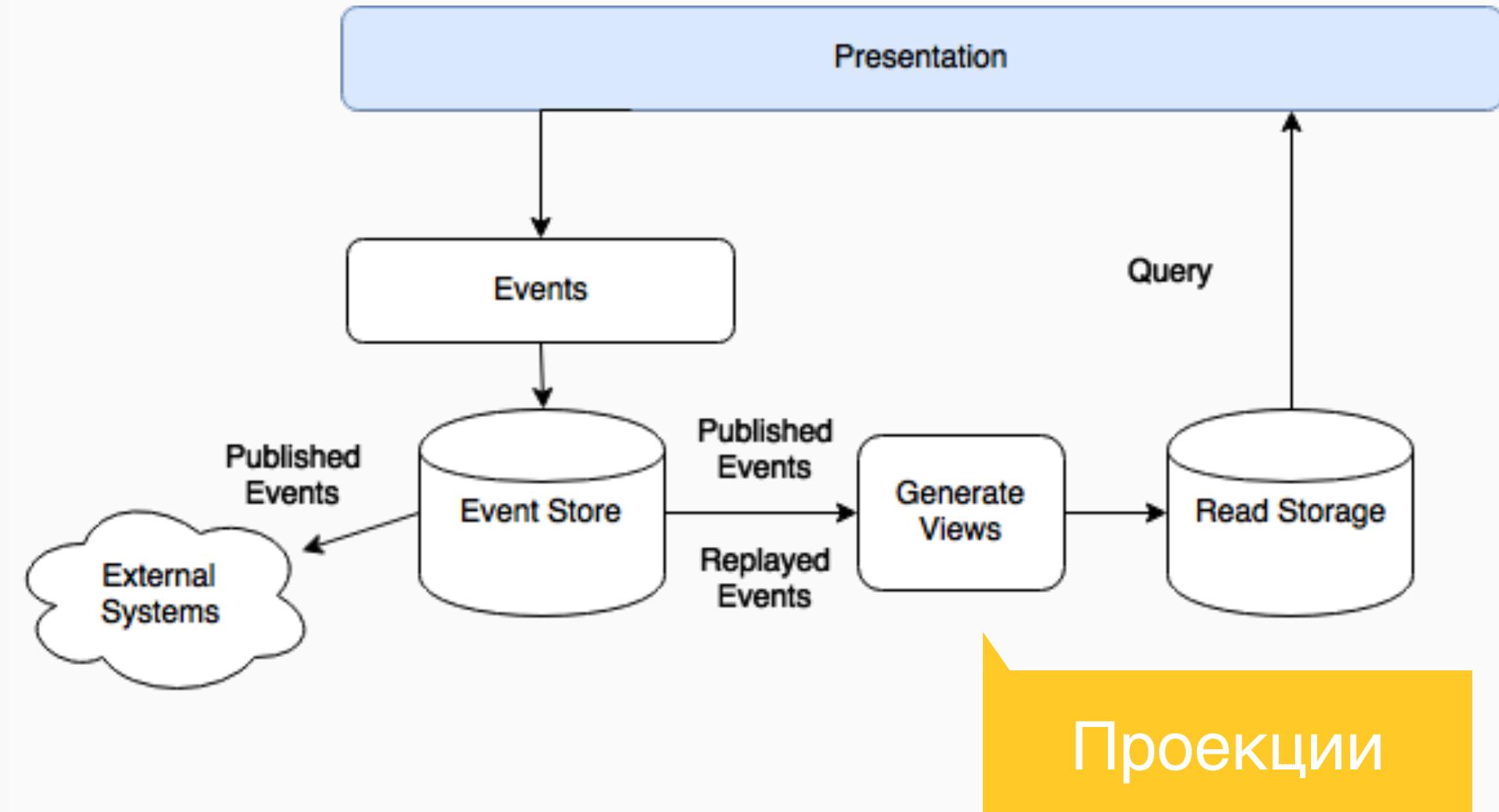
CQRS/ES

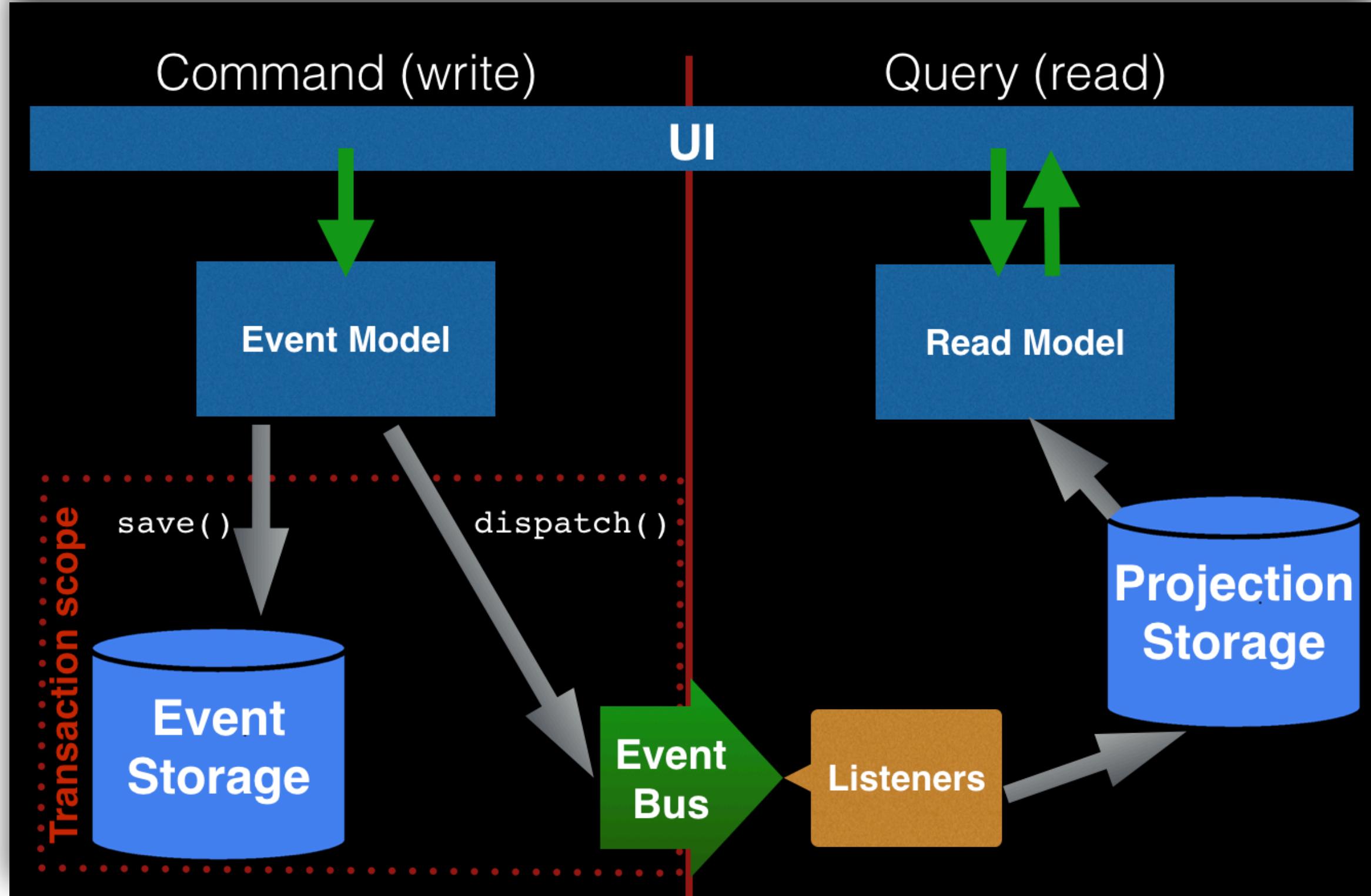
Страна чтения всегда «отстает» от записи. Приложение становится «в конечном итоге согласованным». Дешевле и легче масштабировать **Read Side**, но **Write Side** более сложнее, чем монолитное приложение CRUD.

Eventual Consistency

(Согласованность в конечном счете)

Децентрализация немедленной согласованности (все имеет одинаковый вид данных все время) в системе, в обмен на более высокую доступность и большую автономность компонентов.





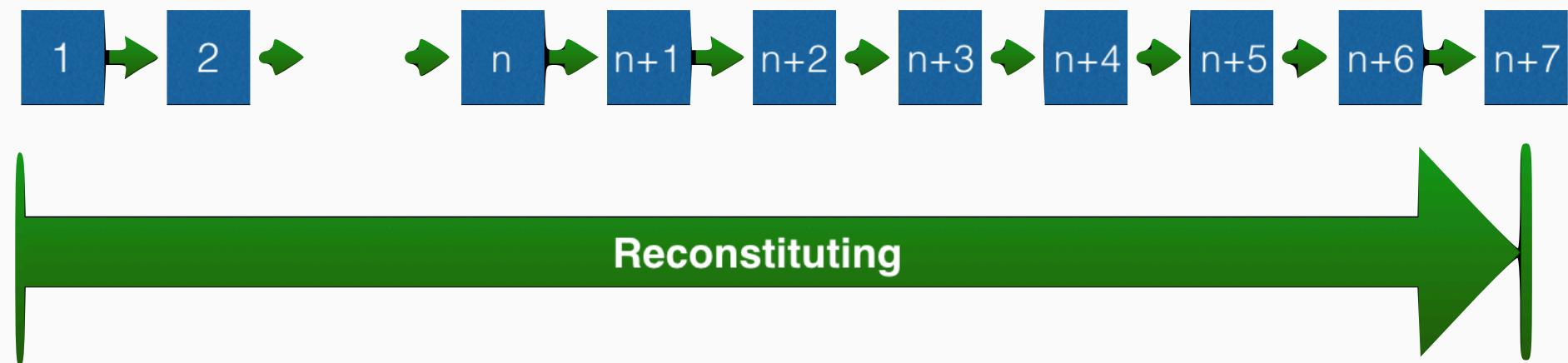
Моментальные снимки

А если ооооочень много событий?
Все же будет тормозить при восстановлении!

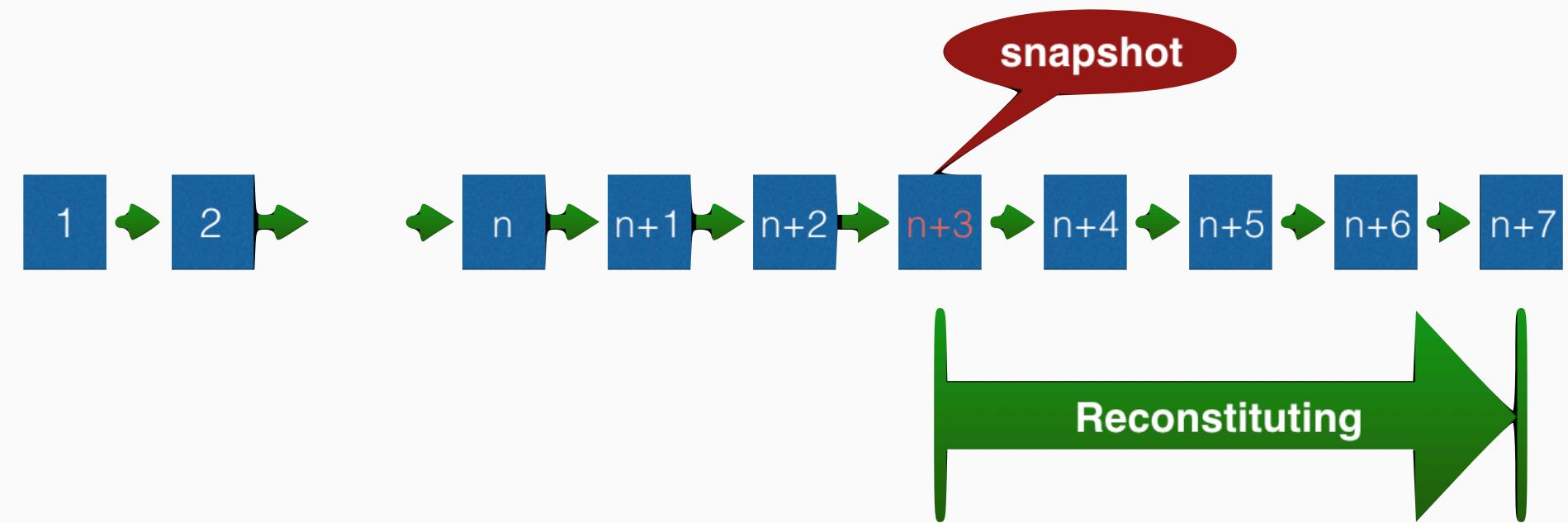
Snapshotting

Механизм, который позволяет восстанавливать агрегат из ненулевого состояния.

Можно избежать обработки большого количества событий с ним.



Reconstituting



snapshot

Reconstituting

Хочу еще...

Что не успели, но стоит посмотреть?

- Проблема: побочные эффекты (**Side-effects**).

Решение 1: Отделить эффект и изменение состояния.

Решение 2: Эффекты в подписках на события.

- Проблема: рефакторинг событий.

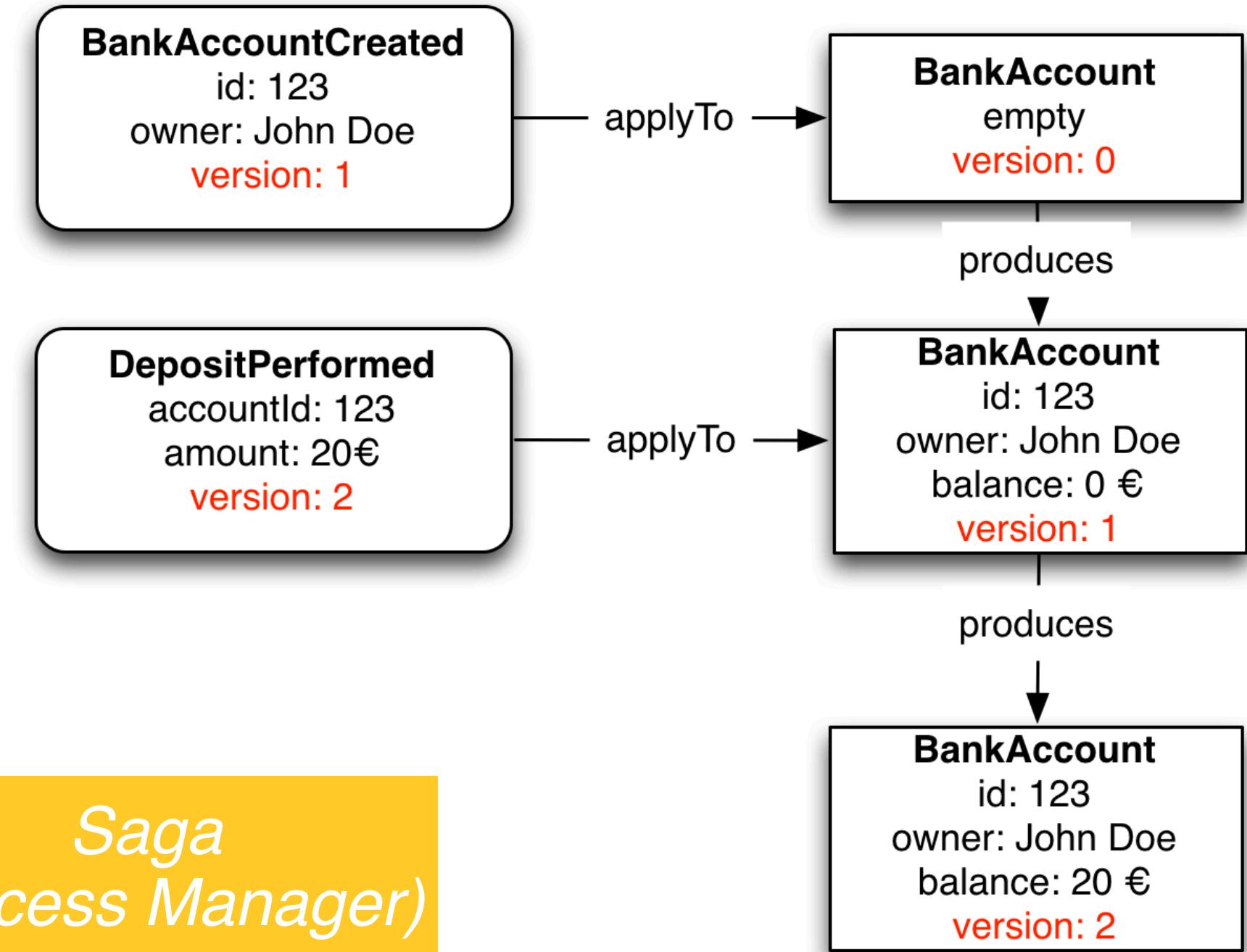
Решение 1: Перезапись событий в хранилище.

Решение 2: Обновление событий в рантайме.

Решение 3: Snapshotting.

- Проблема: одновременная запись.

Решение: оптимистичная блокировка (**Optimistic Locking**)



*Saga
(Process Manager)*

HAWAII



События могут обрабатываться в фоновом режиме

Отсутствие конкуренции во время обработки транзакций может улучшить производительность и масштабируемость приложений.

События представляют собой простые объекты

Они просто записываются для обработки в соответствующее время. Это позволяет упростить управление и реализацию.

События представляют ценность для эксперта

Они могут быть прочитаны и проанализированы не только разработчиком, но и специалистом предметной области.

События отделены от задач

Обеспечиваются гибкость и расширяемость. Каждое событие может обрабатывать несколько задач. Простая интеграция с другими службами и системами.



Преимущества

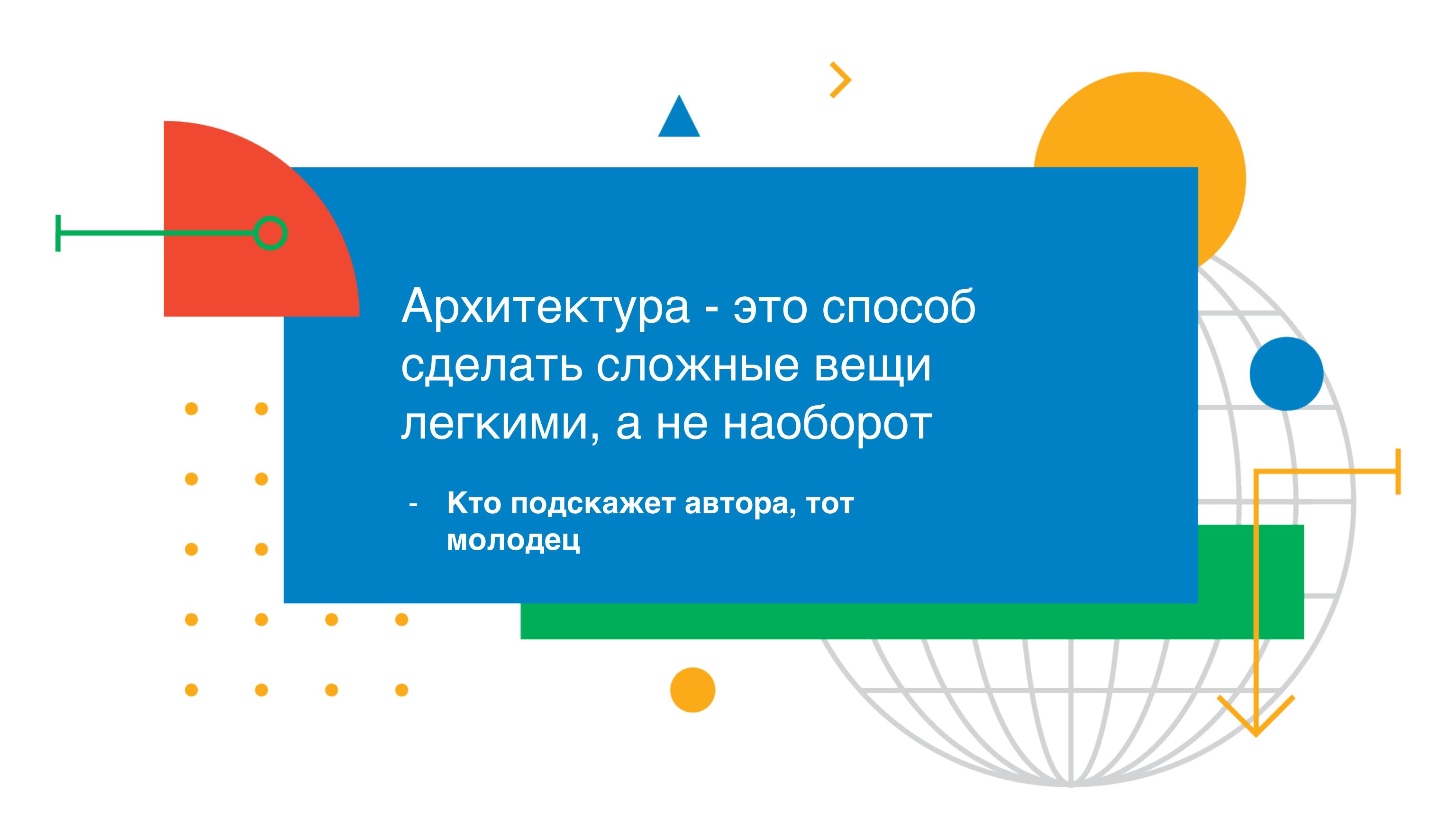
- Полный аудит событий
- Возможность восстановления
- Анализ экспертами
- Хорошая производительность
- Нет маппинга в БД и обратно
- Нет транзакций
- Нет сложных запросов
- Проще тестировать и отлаживать
- Конечная согласованность



Недостатки

- Непростой рефакторинг
- Непростая реализация
- Так себе инструментарий
- Требования к дисковому пространству
- Расхождение с интерфейсом
- Конечная согласованность)





Архитектура - это способ
сделать сложные вещи
легкими, а не наоборот

- Кто подскажет автора, тот
молодец

“That's all Folks!”

A dark, grainy image showing the character Deadpool from the chest up. He is wearing his signature red and black mask and a red, white, and grey horizontally striped cardigan over a white t-shirt. He is looking slightly to the right of the camera with a neutral expression. The background is a plain, light-colored wall.

Сцена после титров

Типы событий

```
// Common

const eventTypes = {
    ACCOUNT_OPENED: 'ACCOUNT_OPENED',
    ACCOUNT_CREDITED: 'ACCOUNT_CREDITED',
    ACCOUNT_DEBITED: 'ACCOUNT_DEBITED'
};
```

Генерация

```
// WRITE MODEL :: Domain layer

class Account {
    // . . .

    open(id, ownerId, balance = 0) {
        if (this.balance < 0) throw new Error('Can't open');

        this.recordThat([
            {type: eventTypes.ACCOUNT_OPENED, payload: {id, ownerId, balance}}
        ]);
    }

    credit(amount) {
        this.recordThat([
            {type: eventTypes.ACCOUNT_CREDITED, payload: {id: this.id, amount}}
        ]);
    }

    debit(amount) {
        if (this.balance < amount) throw new Error('Can't debit');

        this.recordThat([
            {type: eventTypes.ACCOUNT_DEBITED, payload: {id: this.id, amount}}
        ]);
    }

    // . . .
}
```

Открытие счета

```
open(id, ownerId, balance = 0) {
    if (this.balance < 0) throw new Error('Cant open');

    this.recordThat([
        {type: eventTypes.ACCOUNT_OPENED, payload: {id, ownerId, balance}}
    ]);
}
```

// WRITE MODEL :: Domain layer

```
class Account {  
    // . . .  
  
    constructor() {  
        this.recordedEvents = [ ];  
    }  
  
    getRecordedEvents() {  
        return [...this.events];  
    }  
  
    recordThat(events) {  
        this.recordedEvents.concat(events);  
        this.replay(events);  
    }  
  
    replay(events) {  
        events.map((event) => this.apply(event));  
    }  
  
    // . . .  
}
```

Запись и
применение

Запись и применение

```
// WRITE MODEL :: Domain layer

class Account {
    // . .

    apply({type, payload}) {
        switch (type) {
            case eventTypes.ACCOUNT_OPENED: return this.applyOpened(payload);
            case eventTypes.ACCOUNT_CREDITED: return this.applyCredited(payload);
            case eventTypes.ACCOUNT_DEBITED: return this.applyDebited(payload);
        }
    }

    applyOpened({id, ownerId, balance}) {
        this.id = id;
        this.ownerId = ownerId;
        this.balance = balance;
    }

    applyCredited({amount}) {
        this.balance += amount;
    }

    applyDebited({amount}) {
        this.balance -= amount;
    }

    // . .
}
```

Применение открытия счета

```
apply({type, payload}) {
    switch (type) {
        case eventTypes.ACCOUNT_OPENED: return this.applyOpened(payload);
        case eventTypes.ACCOUNT_CREDITED: return this.applyCredited(payload);
        case eventTypes.ACCOUNT_DEBITED: return this.applyDebited(payload);
    }
}

applyOpened({id, ownerId, balance}) {
    this.id = id;
    this.ownerId = ownerId;
    this.balance = balance;
}
```

// WRITE MODEL :: Domain layer

```
class Account {  
    // . . .  
  
    static fromHistory(events) {  
        const self = new Account();  
        self.replay(events);  
        return self;  
    }  
  
    // . . .  
}
```

Восстановление
из истории

// Application layer

```
class AnyCommandHandler {  
    constructor(eventStore) {  
        this.eventStore = eventStore;  
    }  
  
    async handle(command) {  
  
        const account = Account::fromHistory(  
            await this.eventStore.load(  
                command.accountId  
            )  
        );  
  
        account.open(command.ownerId, 500);  
        account.credit(1000);  
        account.debit(200);  
  
        return await this.eventStore.saveAndDispatch(  
            account.getRecordedEvents()  
        );  
    }  
}
```

Какой-то обработчик
какой-то команды

// Presentation layer

```
const req = {  
    body: {  
        accountId: 'aAccountId',  
        ownerId: 'aUserId'  
    }  
};
```

```
router.post('/accounts/handle', async (req, res) => {  
  
    try {  
        await (new AnyCommandHandler(eventStore)).handle(  
            req.body  
        );  
        res.sendStatus(200);  
    } catch (e) {  
        res.sendStatus(500);  
    }  
});
```

Какой-то обработчик
какой-то команды

// // READ MODEL

// Projection

```
eventStore.on(eventTypes.ACCOUNT_OPENED, ({id, ownerId})  
    //INSERT INTO `accounts` VALUES (id, ownerId, balance)  
});
```

```
eventStore.on(eventTypes.ACCOUNT_CREDITED, ({id, amount}) => {  
    //UPDATE `accounts` SET `balance` = `balance` + amount WHERE `id` = id  
});
```

```
eventStore.on(eventTypes.ACCOUNT_DEBITED, ({id, amount}) => {  
    //UPDATE `accounts` SET `balance` = `balance` - amount WHERE `id` = id  
});
```

// Listener

```
eventStore.on(eventTypes.ACCOUNT_OPENED, ({id, ownerId}) => {  
    mailService.sendWelcome(ownerId);  
});
```

Что бы почитать?

YES

NO

MAYBE

DON'T
KNOW