Spring Cloud в микросервисной архитектуре или История одного внедрения

Олег Клименко

Домклик

JPoint

Домклик

Олег Клименко

klimenkoob

Bio

- В программировании с 2005
- 12 лет в продуктовой разработке
- 5 лет в Домклик

Инфраструктура Домклик

- 24 команды Java стека
- 200+ Java Сервисов
- 6000к RPS
- ~80 Java разработчиков

О чем расскажем?

- Расскажем о том как мы внедряли Spring Cloud
- С какими проблемами столкнулись
- Заглянем немного под капот
- О том какие сюрпризы нас ожидали
- HowTo к возникающим вопросам

Начало…

- На дворе шел 2018 год
- Стартап. Динамично развитие. Запуск продукта за продуктом
- Активной найм и рост

Что имеем?

- Приложения становятся слишком большие и тяжелые - нужно что поболее распределенное и легкое

Что имеем?

- Приложения становятся слишком большие и тяжелые - нужно что поболее распределенное и легкое
- Нужно сделать так что бы при этом у нас не увеличилась стоимость поддержки

Что имеем?

- Приложения становятся слишком большие и тяжелые - нужно что поболее распределенное и легкое
- Нужно сделать так что бы при этом у нас не увеличилась стоимость поддержки
- Чтобы не рос комплексити и сохранялась понятность

Что имеем?

- Приложения становятся слишком большие и тяжелые - нужно что поболее распределенное и легкое
- Нужно сделать так что бы при этом у нас не увеличилась стоимость поддержки
- Чтобы не рос комплексити и сохранялась понятность
- Множество различных подходов. Хочется общего решения. Стандартизация

Что делать?

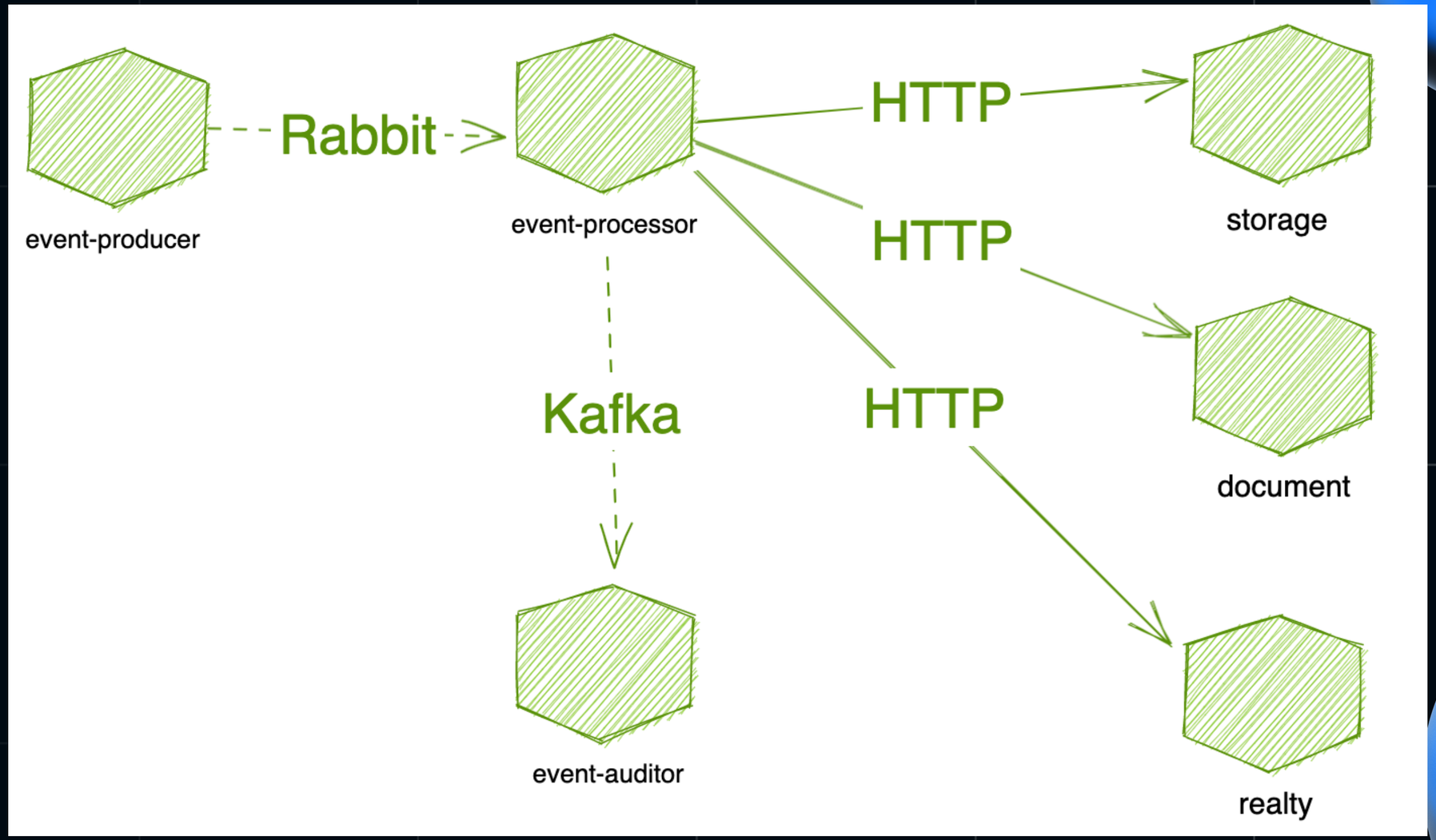
Группа энтузиастов из 4 человек решила попробовать написать новый проект на Spring Cloud используя его преимущества

С чего начать?

Взаимодействие:

- HTTP (REST)
- MQ (Rabbit/Kafka)

С чего начать?



Взаимодействие сервисов через REST

Что предлагает Spring Cloud?

Cloud Open Feign

Импорт:

implementation(“org.springframework.cloud:spring-cloud-starter-openfeign")

Что представляет собой Cloud Open Feign?

- Сам Open Feign
- Spring Cloud обертка для интеграции в Spring

Немного теории

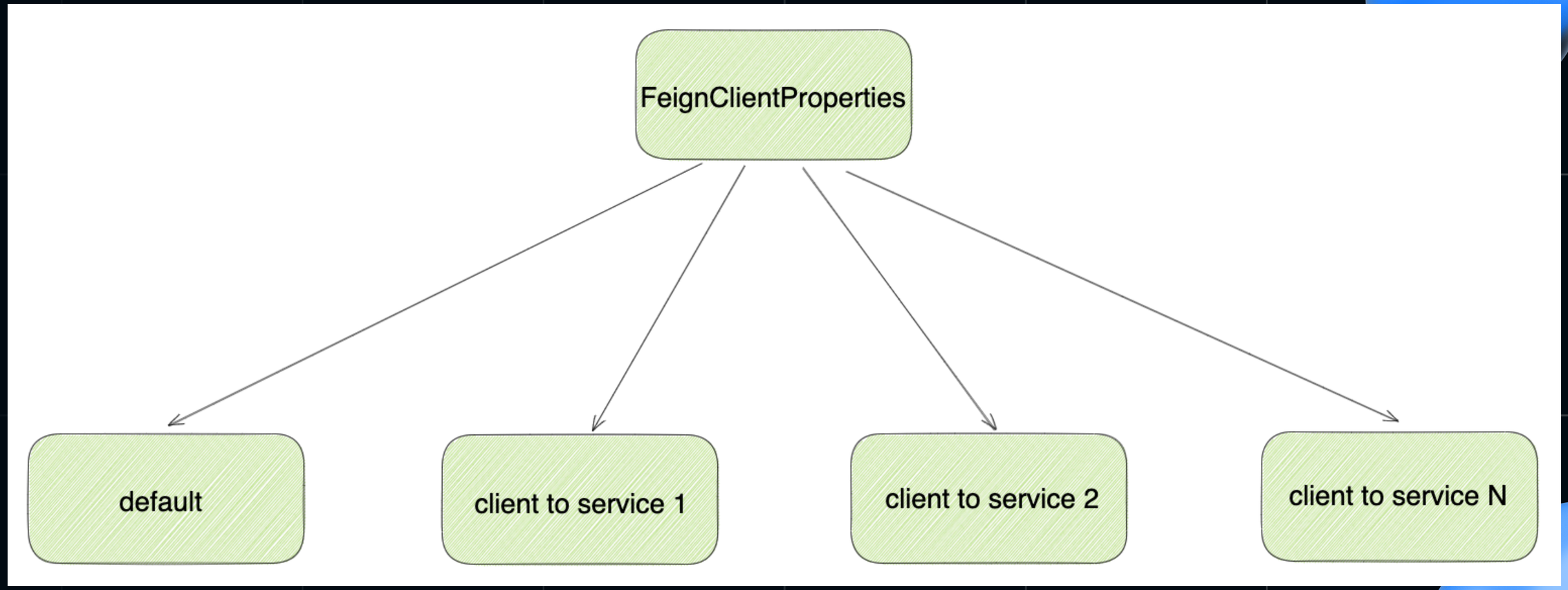
Что есть из коробки?

- @EnableFeignClients
- @FeignClient
- FeignClientProperties с субконфигурациями клиентов
- Собственный независимый контекст FeignContext/FeignClientFactory

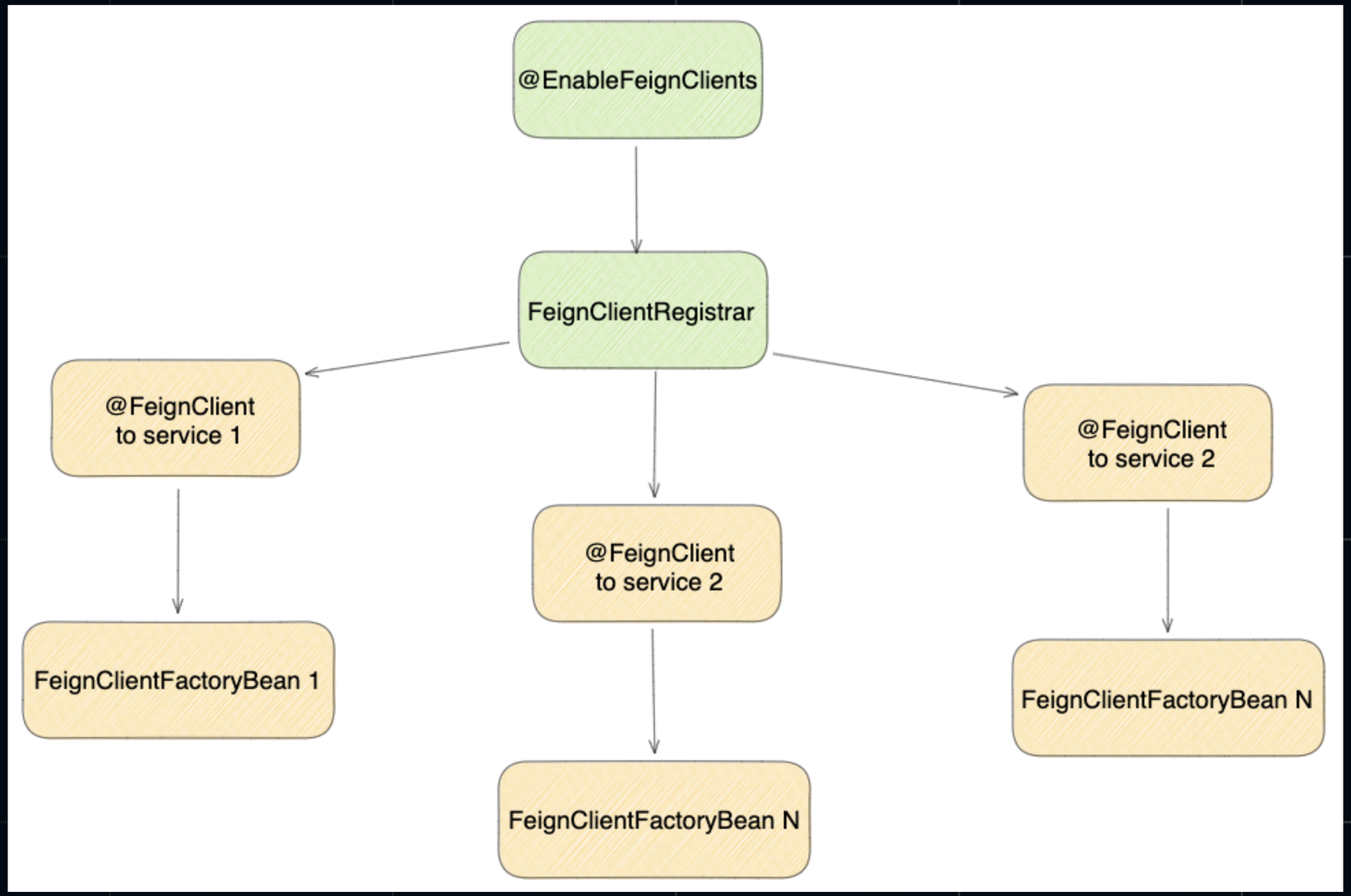
FeignClientProperties и его суб конфигурации:

```
@ConfigurationProperties("feign.client")
public class FeignClientProperties {

    5 usages
    private boolean defaultToProperties = true;

    5 usages
    private String defaultConfig = "default";

    5 usages
    private Map<String, FeignClientConfiguration> config = new HashMap<>();
```

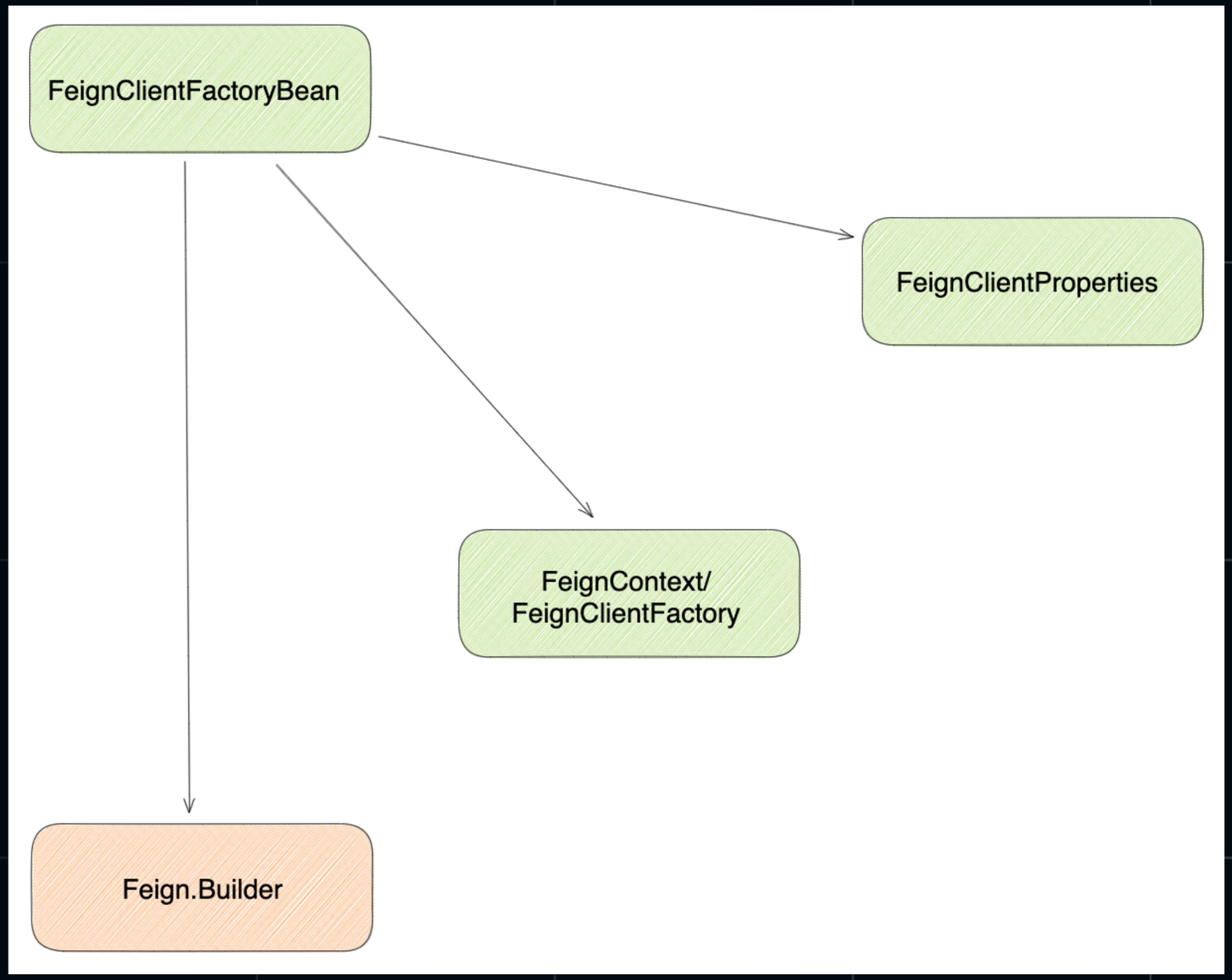
FeignClientProperties и его суб конфигурации:



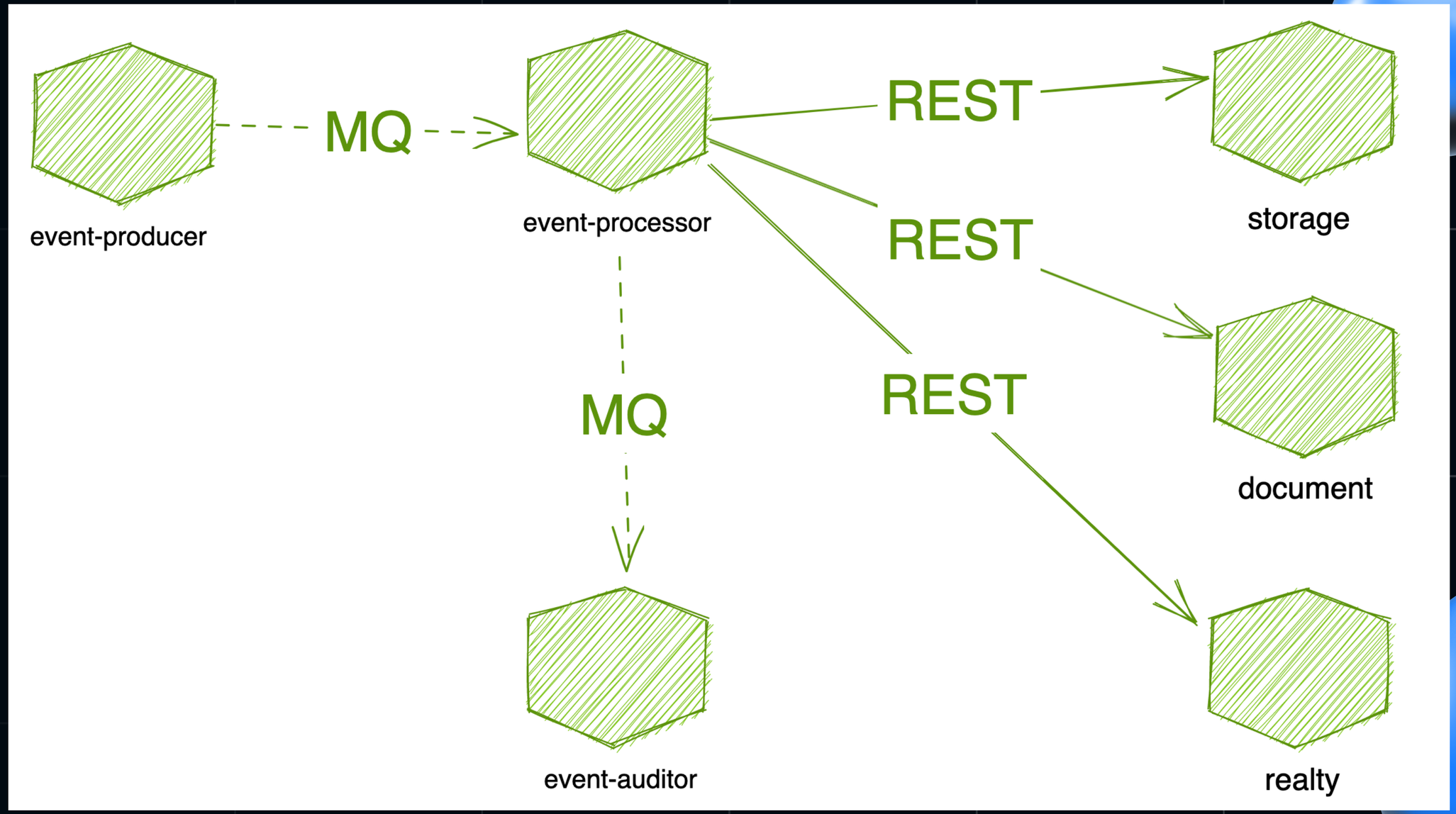
Сканирование самих клиентов:



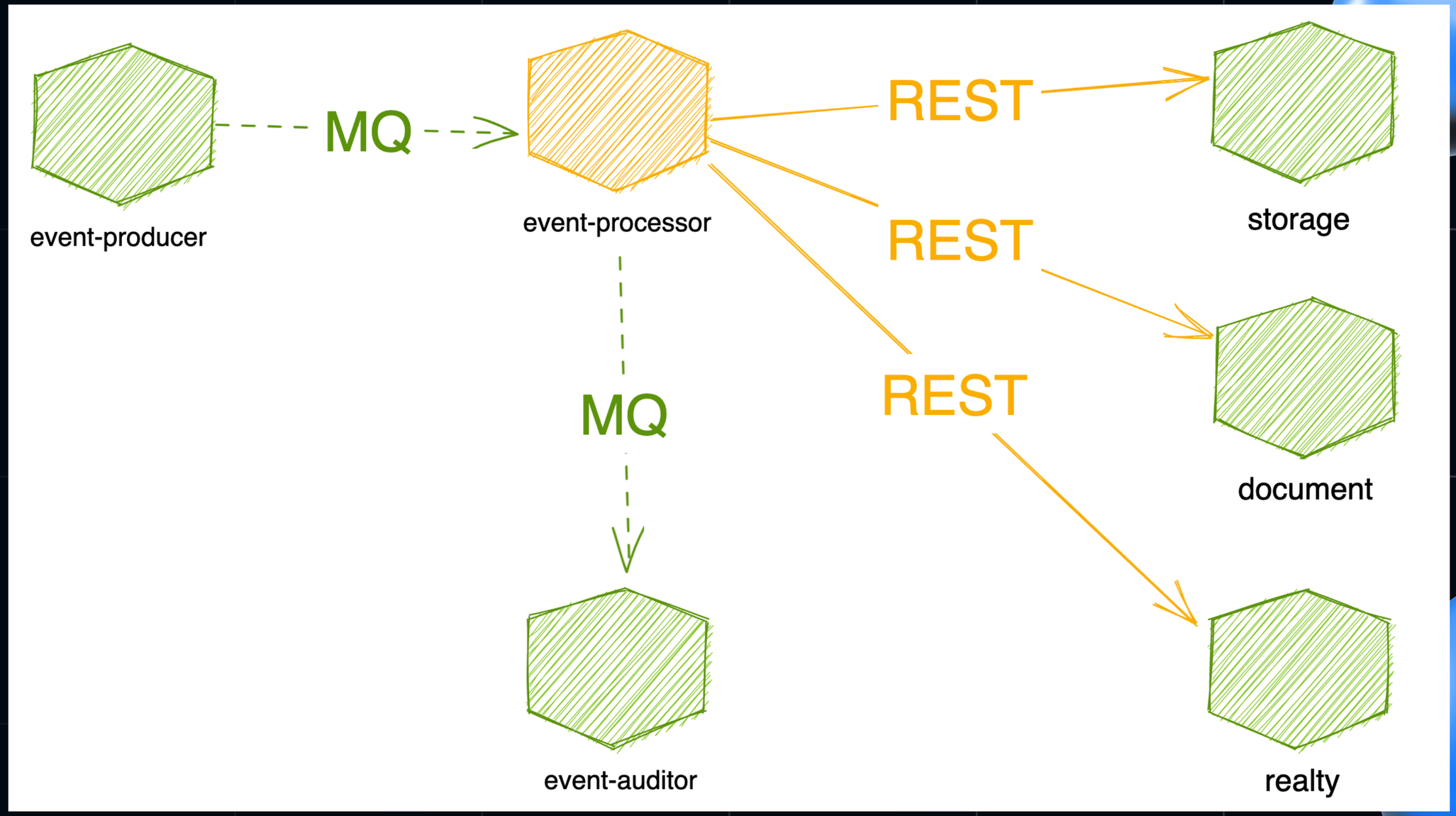
Связывание properties и factory bean и context



Вернемся к практике



Вернемся к практике



Конфигурация клиента

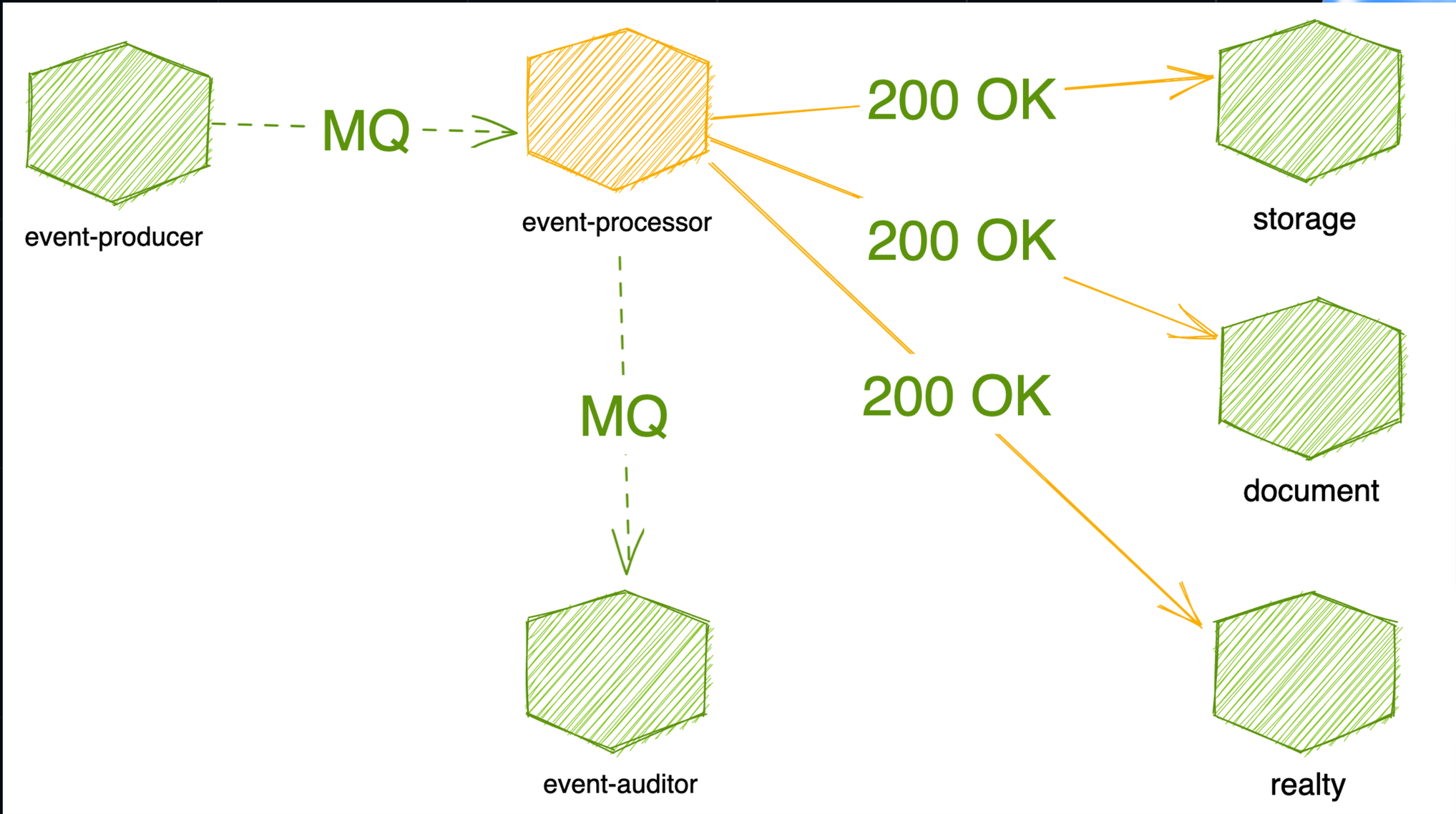
application.yml

```yaml
# ================================
# = Feign
# ================================
feign:
  client:
    config:
     default:
        logger-level: FULL
        connectTimeout: 5000
        readTimeout: 5000
     storage-client:
        connectTimeout: 100
        readTimeout: 200
     document-client:
        connectTimeout: 200
        readTimeout: 300
     realty-client:
        connectTimeout: 500
        readTimeout: 500
```

декларирование Клиента

```
@FeignClient(
    name = "storage-client",
    url = "\${feign.client.config.storage-client.base-url}",
    path = “\${feign.client.config.storage-client.sub-path}»,
    configuration = [
        StorageApiConfiguration::class
    ]
)
interface StorageClientApi {
    @GetMapping(path = [“/v1/storage/deals/{dealId}”])
    fun getDeal(
        @PathVariable dealId: Long
    ): DealDto
}
```

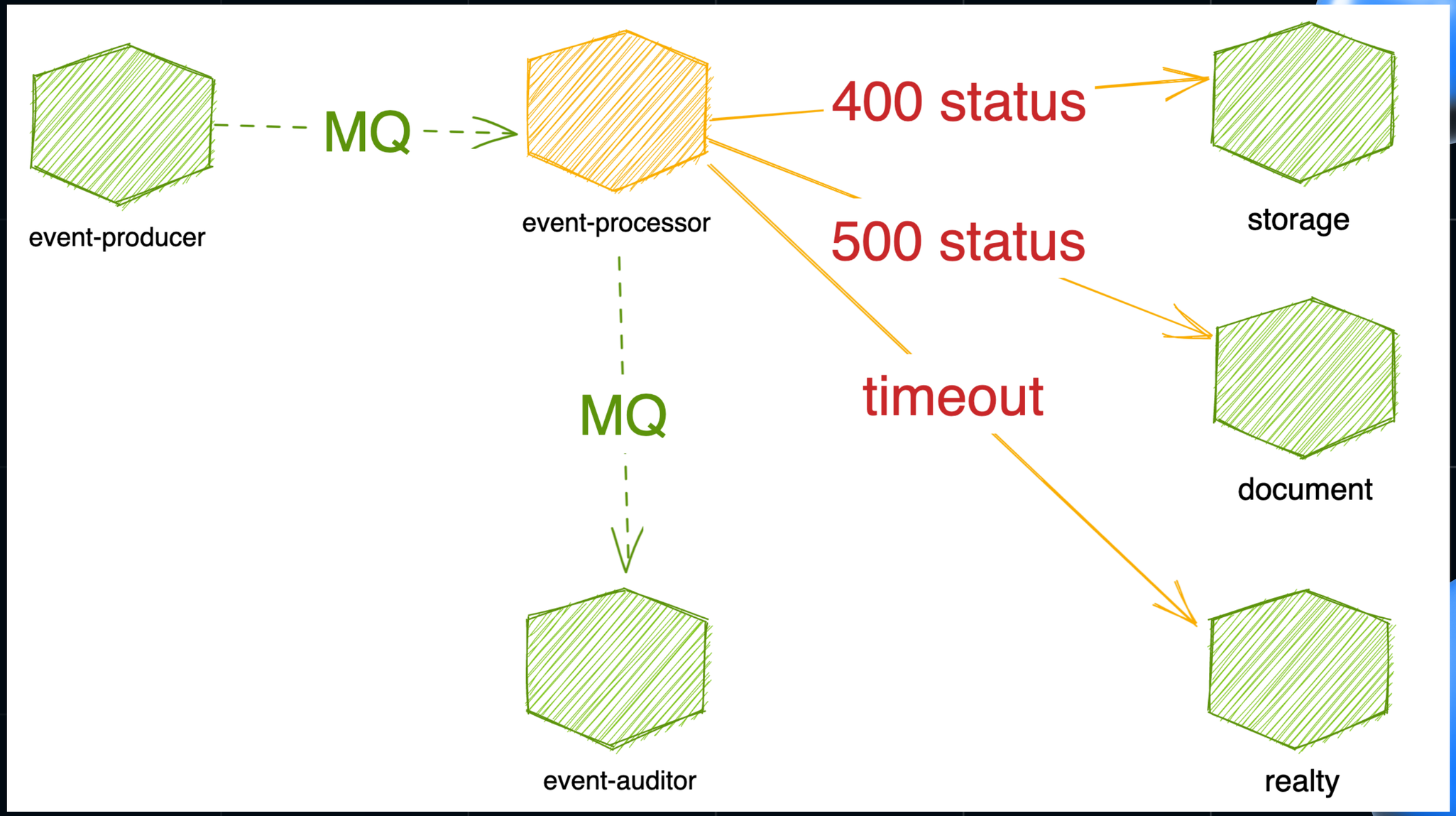
Пробуем взлететь...



Внезапно упал метеорит…

Или просто кто то зарелизился…

Пробуем взлететь...



Как сделать процесс обработки таких ошибок более корректным?

Для обработки ошибок нам помогут:

- Декодеры

И политики:

- Retry
- Circuit breaker

Помним, что у нас каждого клиента свой Feign Application Context, то есть:

- Каждому клиенту можно определить свой декодер

Помним, что у нас каждого клиента свой Feign Application Context, то есть:

- Каждому клиенту можно определить свой декодер
- Свои собственный политики

Помним, что у нас каждого клиента свой Feign Application Context, то есть:

- Каждому клиенту можно определить свой декодер
- Свои собственный политики
- Свои настройки timeouts

Сконфигурируем декодер

Декларирование Клиента

```
@FeignClient(
    name = "storage-client",
    url = "\${feign.client.config.storage-client.base-url}",
    path = "\${feign.client.config.storage-client.sub-path}",
    configuration = [
        StorageApiConfiguration::class
    ]
)
interface StorageClientApi {
    @GetMapping(path = ["/v1/storage/deals/{dealId}"])
    fun getDeal(
        @PathVariable dealId: Long
    ): DealDto
}
```

Конфигурация

```
class StorageClientApiConfiguration(
        private val storageClientApiProperties: StorageClientApiProperties
) {

    @Bean
    fun feignBuilder(): Feign.Builder {
        return Feign
                .builder()
                .exceptionPropagationPolicy(ExceptionPropagationPolicy.UNWRAP)
    }

    @Bean
    fun requestInterceptor() = AuthenticationSrvRequestInterceptor(dealClientApiProperties)

    @Bean
    fun errorDecoder(objectMapper: ObjectMapper) = StorageErrorDecoder(objectMapper)

    @Bean
    fun client(): Client = ApacheHttpClient(
            HttpClientBuilder.create()
                    .setMaxConnTotal(storageClientApiProperties.maxConnectionsTotal)
                    .setMaxConnPerRoute(storageClientApiProperties.maxConnectionsPerRoute)
                    .disableAutomaticRetries()
                    .build()
    )
}
```

Готовые реализации политики

Retry/CircuitBreaker?

Готовые реализации политик Retry/CircuitBreaker?

2 способа - в самом клиенте и через аннотации **Spring retry**

Готовые реализации политик Retry/CircuitBreaker?

2 способа - в самом клиенте и через аннотации **Spring retry**

org.springframework.retry:spring-retry

В нем как раз есть 2 аннотации **@Retryable** и **@CircuitBreaker**

Решили пойти для удобства путем написания alias к ним

```
@Retryable(
        include = [
            HttpHostConnectException::class,
            SocketException::class,
            SocketTimeoutException::class
        ],
        maxAttemptsExpression = "\${feign.client.config.storage-client.retry-count}",
        backoff = Backoff(delayExpression = «\${feign.client.config.storage-client.retry-backoff-time}")
)
@Target(AnnotationTarget.FUNCTION, AnnotationTarget.TYPE)
@Retention
annotation class StorageRetryable
```

```
@CircuitBreaker(
        include = [
            HttpHostConnectException::class,
            SocketException::class,
            SocketTimeoutException::class
        ],
        openTimeoutExpression = "#{ new Long(\${feign.client.config.storage-client.circuit-breaker-open-timeout}) }",
        resetTimeoutExpression = "#{ new Long(\${feign.client.config.storage-client.circuit-breaker-reset-timeout}) }"
)
@Target(AnnotationTarget.FUNCTION, AnnotationTarget.TYPE)
@Retention
annotation class StorageCircuitBreaker
```

А сами настройки *timeout* и *backoff* добавим в профиль приложения к остальным настройкам клиента:

```yaml
# ================================
# = Feign
# ================================
feign:
  client:
    config:
     default:
        logger-level: FULL
        connectTimeout: 5000
        readTimeout: 5000
     storage-client:
        connectTimeout: 100
        readTimeout: 200
        retry-count: 3
        retry-backoff-time: 2000
     document-client:
        connectTimeout: 400
        readTimeout: 600
        retry-count: 3
        retry-backoff-time: 2000
     realty-client:
        connectTimeout: 1000
        readTimeout: 2000
        circuit-breaker-open-timeout: 5000
        circuit-breaker-reset-timeout: 20000
```

Интегрируем в код

```
@Service
class StorageConnectorService(
        private val storageClientApi: StorageClientApi,
        private val documentClientApi: StorageClientApi
) {
    @StorageRetryable
    fun getStorageDeal(dealId: Long, expands: List<DealExpands>): DealDto = storageClientApi.getStorageDeal(dealId, expands)

    @DocumentCircuitBreaker
    fun getDocument(dealId: Long, documentId: Long): DocumentDto = storageClientApi.getStorageDeal(dealId, expands)
}
```

Интегрируем в код

```
@Service
class StorageConnectorService(
        private val storageClientApi: StorageClientApi,
        private val documentClientApi: StorageClientApi
) {
    @StorageRetryable
    fun getStorageDeal(dealId: Long, expands: List<DealExpands>): DealDto = storageClientApi.getStorageDeal(dealId, expands)

    @DocumentCircuitBreaker
    fun getDocument(dealId: Long, documentId: Long): DocumentDto = storageClientApi.getStorageDeal(dealId, expands)
}
```

Что делать если retry? не прошел?

Интегрируем в код

```
@Service
class StorageConnectorService(
        private val storageClientApi: StorageClientApi,
        private val documentClientApi: StorageClientApi
) {
    @StorageRetryable
    fun getStorageDeal(dealId: Long, expands: List<DealExpands>): DealDto = storageClientApi.getStorageDeal(dealId, expands)

    @DocumentCircuitBreaker
    fun getDocument(dealId: Long, documentId: Long): DocumentDto = storageClientApi.getStorageDeal(dealId, expands)
}
```

Что делать если retry? не прошел?

```
@Service
class StorageConnectorService(
        private val storageClientApi: StorageClientApi
) {
    @StorageRetryable
    fun getStorageDeal(dealId: Long, expands: List<DealExpands>): DealDto = storageClientApi.getStorageDeal(dealId, expands)

    @DocumentCircuitBreaker
    fun getDocument(dealId: Long, documentId: Long): DocumentDto = storageClientApi.getStorageDeal(dealId, expands)

    @Recover
    fun recoverGetStorageDeal(e: Throwable, dealId: Long): DealDto { return handleRetryException(e) }

    @Recover
    fun recoverGetDocument(e: Throwable, dealId: Long, documentId: Long): DocumentDto { return handleRetryException(e) }
}
```

А что если такой клиент нужен нескольким сервисам?

При желании можно завернуть в стартер

```
implementation("ru.myservices.group:storage-connector-starter")
implementation("ru.myservices.group:document-connector-starter")
implementation("ru.myservices.group:retry-connector-starter")
```

Зависимость Cloud Open Feign можно будет сделать транзитивной и прописать автоконфигурации в spring.factories и нужными @Conditional в нем

Все это будет так же автосканироваться через:

```
@EnableFeignClients(“my.clients.package”)
```

Какие итоги внедрения feign?

Какие итоги внедрения feign?

- Обертка как контракт. Единообразие подходов. Стандартизация

Какие итоги внедрения feign?

- Единообразие подходов. Стандартизация
- Возможность каждой команде использовать под оберткой привычную реализацию клиента (OKHttp/Apache и т.д)

Какие итоги внедрения feign?

- Единообразие подходов. Стандартизация
- Возможность каждой команде использовать под оберткой привычную реализацию клиента (OKHttp/Apache и т.д)
- Контрибьютит множество команд. Совместное ревью

Какие итоги внедрения feign?

- Единообразие подходов. Стандартизация
- Возможность каждой команде использовать под оберткой привычную реализацию клиента (OKHttp/Apache и т.д)
- Контрибьютит множество команд. Совместное ревью
- Быстрый импорт через универсальную зависимость (даже starter)

Какие итоги внедрения feign?

- Единообразие подходов. Стандартизация
- Возможность каждой команде использовать под оберткой привычную реализацию клиента (OKHttp/Apache и т.д)
- Контрибьютит множество команд. Совместное ревью
- Быстрый импорт через универсальную зависимость (даже starter)
- Из коробки интеграция с APM Elastic и Micrometer

Какие итоги внедрения feign?

- Единообразие подходов. Стандартизация
- Возможность каждой команде использовать под оберткой привычную реализацию клиента (OKHttp/Apache и т.д)
- Контрибьютит множество команд. Совместное ревью
- Быстрый импорт через универсальную зависимость (даже starter)
- Из коробки интеграция с APM Elastic и Micrometer
- Низкий порог вхождения для развития текущих клиентов

Что с асинхронным взаимодействием?

Взаимодействие через MQ

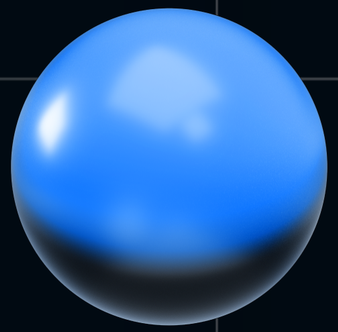
Взаимодействие через MQ

У нас:

- Rabbit
- Kafka

Что предлагает Spring Cloud?

Взаимодействие через MQ



У нас:

- Rabbit
- Kafka

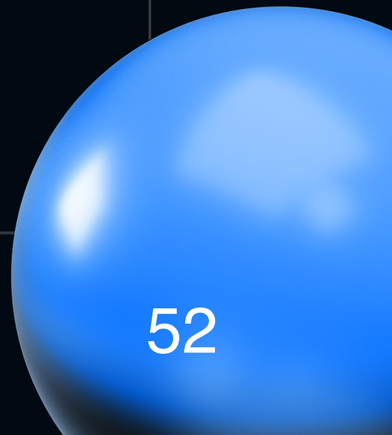
Что предлагает Spring Cloud?

Cloud Stream

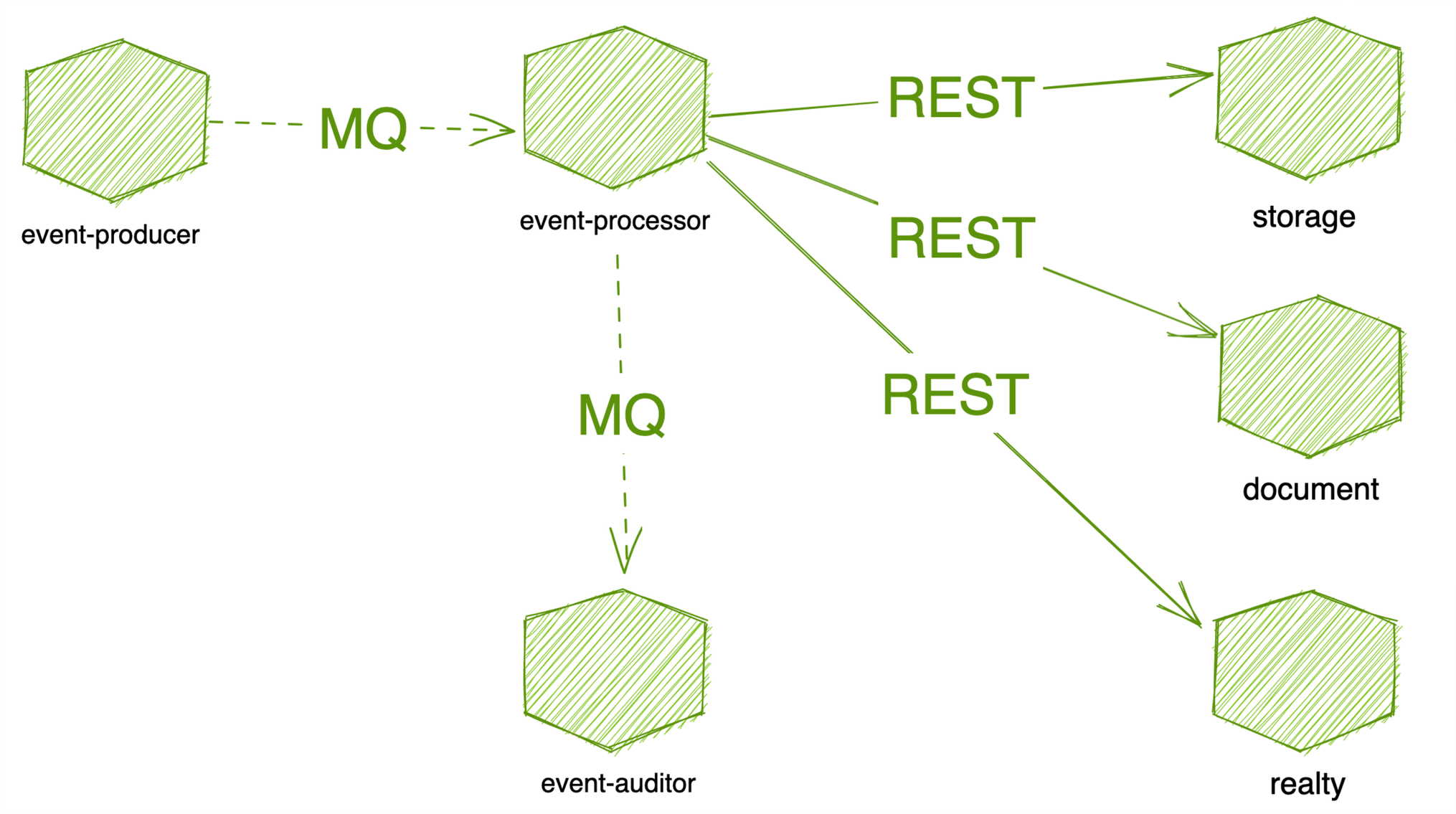
Импорт:

implementation("org.springframework.cloud:spring-cloud-starter-stream-kafka")

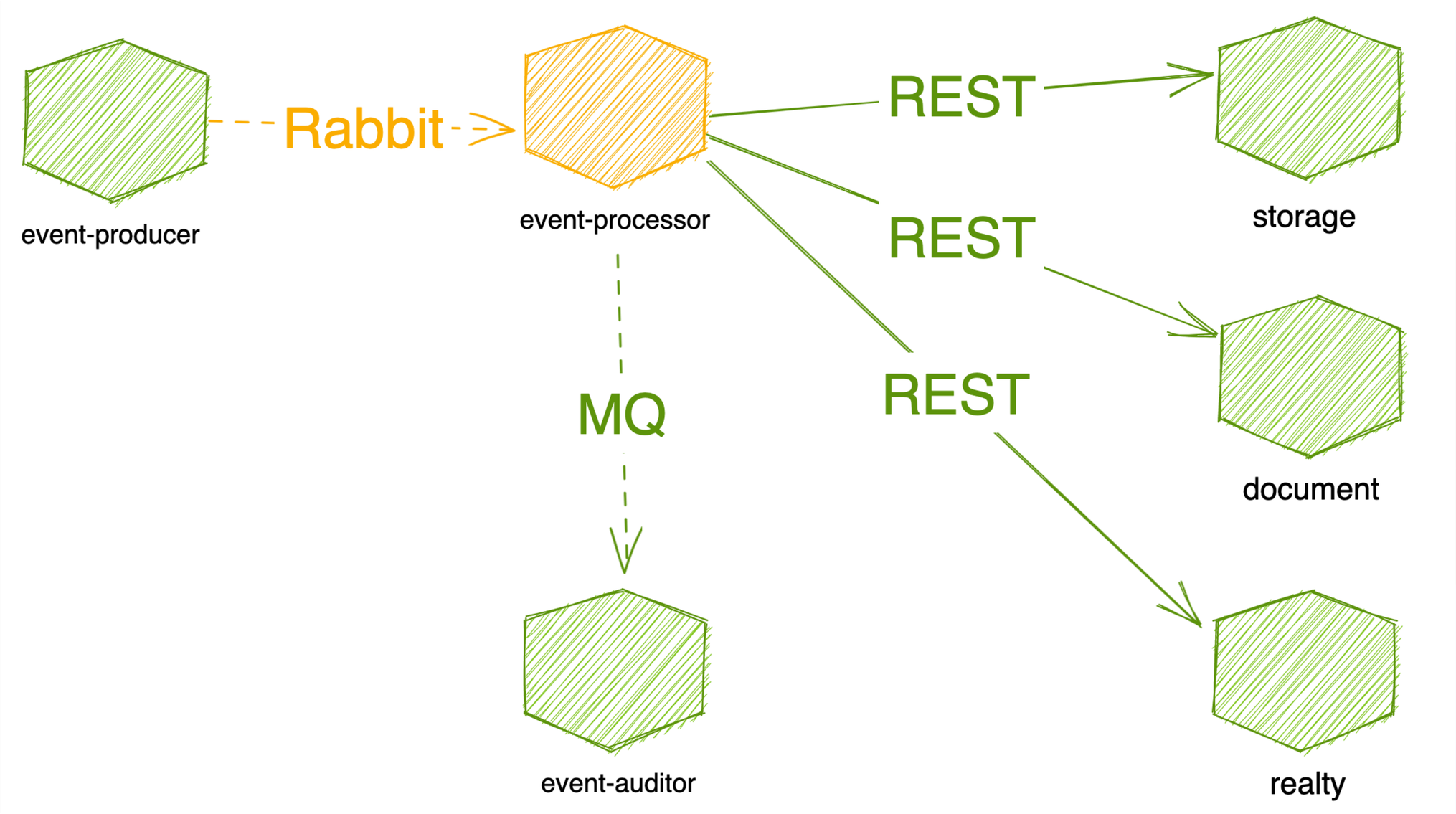
implementation("org.springframework.cloud:spring-cloud-starter-stream-rabbit")



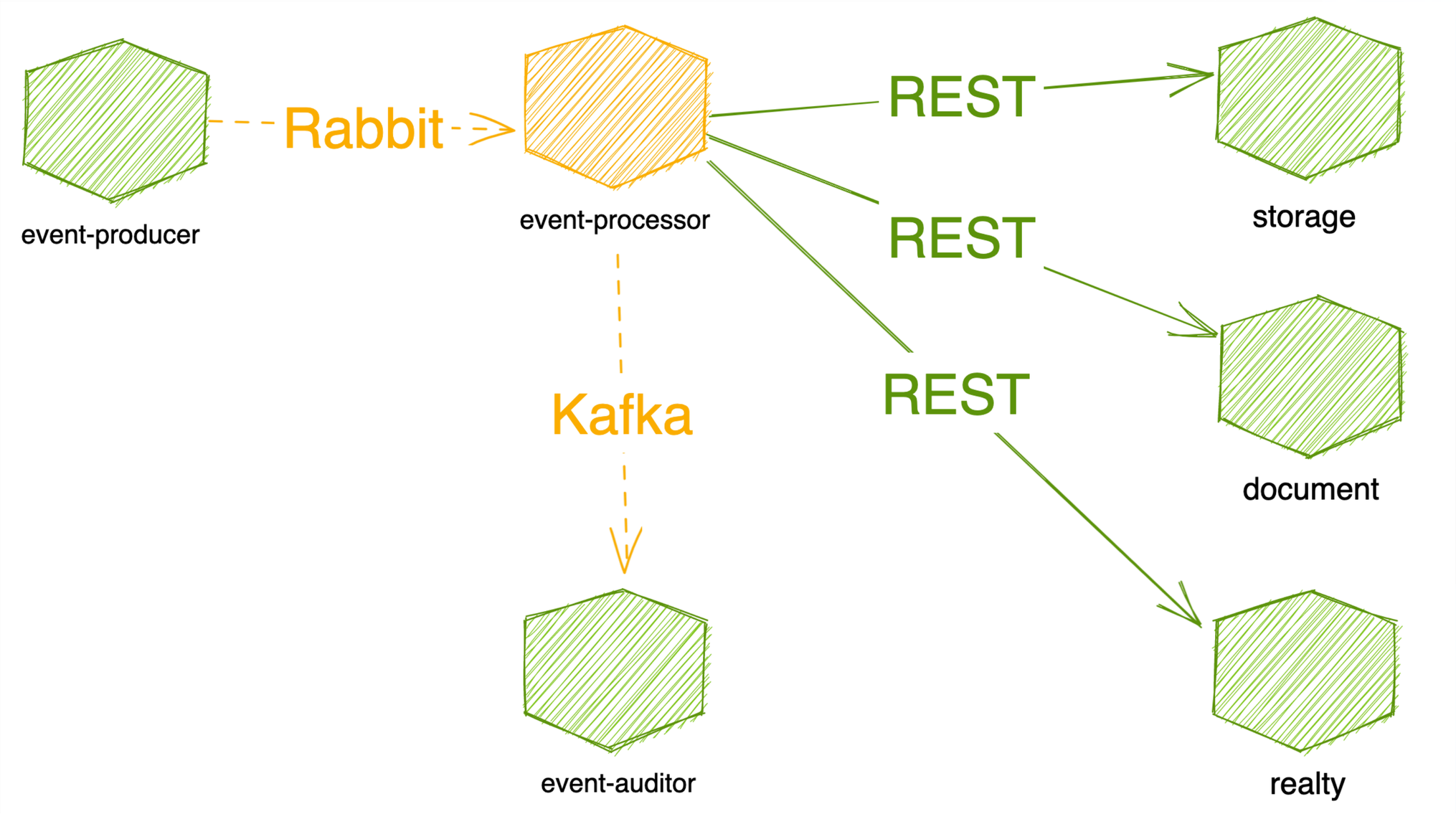
Наша задача


event-producer
MQ
event-processor
REST
REST
REST
storage
document
realty
MQ
event-auditor

Наша задача


event-producer
Rabbit
event-processor
REST
REST
REST
storage
document
realty
MQ
event-auditor

Наша задача


event-producer
Rabbit
event-processor
REST
REST
REST
storage
document
realty
Kafka
event-auditor

Как работали раньше?

- Через template (RabbitTemplate/KafkaTemplate)
- Писали отдельные @Configuration для конфигурирования Queue/Exchange/Binding
- Или просто передавали название Exchange в template

Как работали раньше?

- Через template (RabbitTemplate/KafkaTemplate)
- Писали отдельные @Configuration для конфигурирования Queue/Exchange/Binding
- Или просто передавали название Exchange в template

А как в CloudStream?

Конфигурация

Конфигурация

code-config отсутствует! А точнее он не требуется с нашей стороны…

Он под капотом

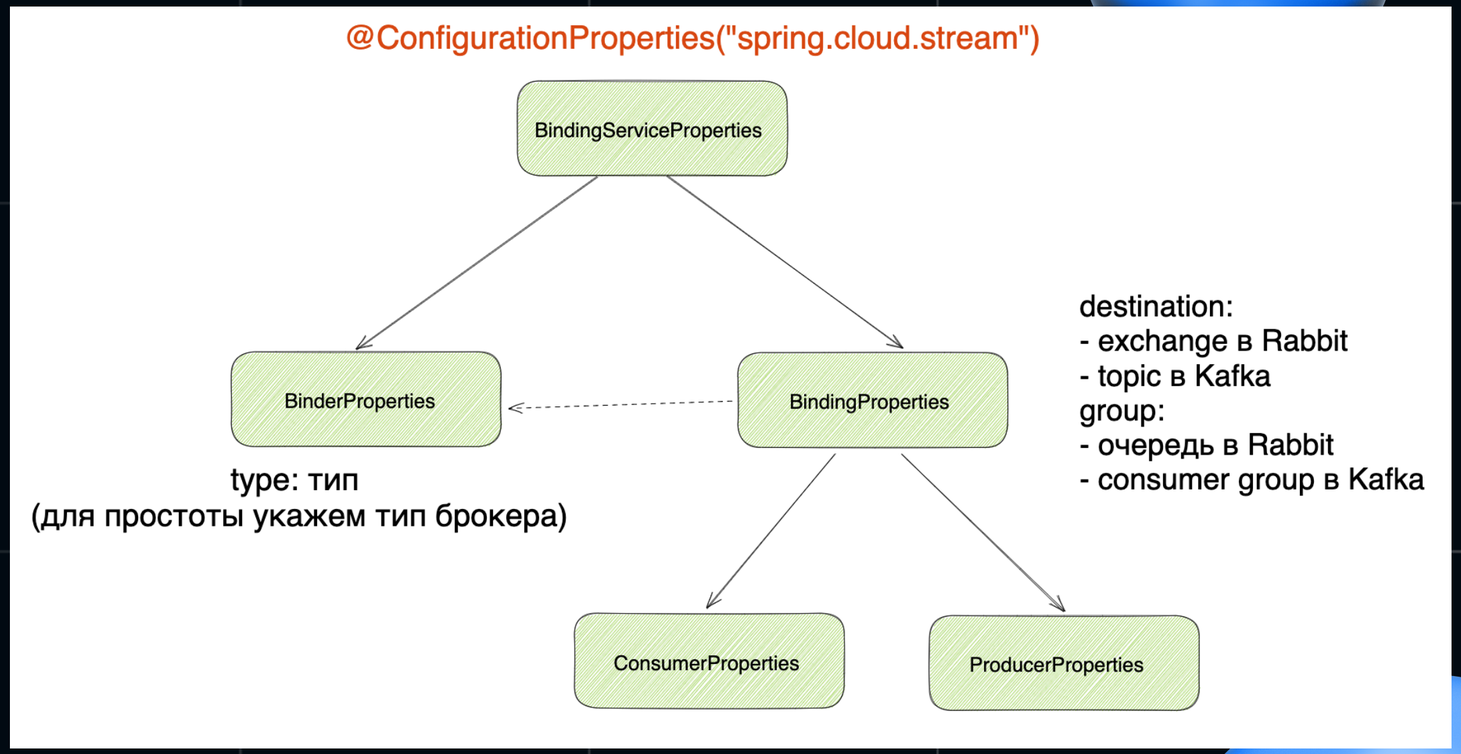
Все конфигурируется в application.yaml!

Попробуем разобраться…

Что под капотом? Чуть теории =)

Есть 2 основных понятия:

- Binders
- Bindings



Что под капотом? Чуть теории =)

Есть 2 основных понятия:

- Binders
- Bindings

В bindings:

destination:

- exchange в Rabbit
- topic в Kafka

group:

- очередь в Rabbit
- consumer group в Kafka

```yaml
spring:
  cloud:
    stream:
      function:
        definition: event-processor;
      binders:
        event-rabbit-binder:
          type: rabbit
        event-kafka-binder:
          type: kafka
      bindings:
        event-processor-in-0:
          binder: event-rabbit-binder
          destination: something-state-change.exchange
        event-processor-out-0:
          binder: event-kafka-binder
          destination: events_processing_topic
```

Пробуем стартануть...

Пробуем стартануть…

“Анонимные очереди” в RabbitMQ.

(non-durable, exclusive, auto-delete queue)

Почему?

Пробуем стартануть...

“Анонимные очереди” в RabbitMQ.

(non-durable, exclusive, auto-delete queue)

Почему?

Не указано название очереди явно!

```yaml
spring:
  cloud:
    stream:
      function:
        definition: event-processor;
      binders:
        event-rabbit-binder:
          type: rabbit
        event-kafka-binder:
          type: kafka
      bindings:
        event-processor-in-0:
        binder: event-rabbit-binder
          destination: something-state-change.exchange
          group: storage. something-state-queue
        storage-event-processor-out-0:
          binder: storage-kafka-binder
          destination: storage_events_topic
```

Чем это опасно?

- При рестарте/отказе брокера все сообщения так же теряются

```yaml
spring:
  cloud:
    stream:
      function:
        definition: event-processor;
      binders:
        event-rabbit-binder:
          type: rabbit
        event-kafka-binder:
          type: kafka
      bindings:
        event-processor-in-0:
        binder: event-rabbit-binder
          destination: something-state-change.exchange
          group: storage. something-state-queue
        storage-event-processor-out-0:
          binder: storage-kafka-binder
          destination: storage_events_topic
```

Чем это опасно?

- При рестарте/отказе брокера все сообщения так же теряются
- Сообщения из очереди теряются при передеплое

```yaml
spring:
  cloud:
    stream:
      function:
        definition: event-processor;
      binders:
        event-rabbit-binder:
          type: rabbit
        event-kafka-binder:
          type: kafka
      bindings:
        event-processor-in-0:
        binder: event-rabbit-binder
          destination: something-state-change.exchange
          group: storage. something-state-queue
        storage-event-processor-out-0:
          binder: storage-kafka-binder
          destination: storage_events_topic
```

Чем это опасно?

- При рестарте/отказе брокера все сообщения так же теряются
- Сообщения из очереди теряются при передеплое
- Приложение стартует без ошибок и очередь вроде бы разгребается

```yaml
spring:
  cloud:
    stream:
      function:
        definition: event-processor;
      binders:
        event—rabbit-binder:
          type: rabbit
        event-kafka-binder:
          type: kafka
      bindings:
        event-processor-in-0:
        binder: event-rabbit-binder
          destination: something-state-change.exchange
          group: storage. something-state-queue
        event-processor-out-0:
          binder: storage-kafka-binder
          destination: storage_events_topic
```

Немного про специфичные настройки для Rabbit/Kafka

Rabbit specific:

- Политики Dead-Letter-Queue (DLQ) с time-to-live
- Тип exchange (direct/topic/fanout)

Немного про специфичные настройки для Rabbit/Kafka

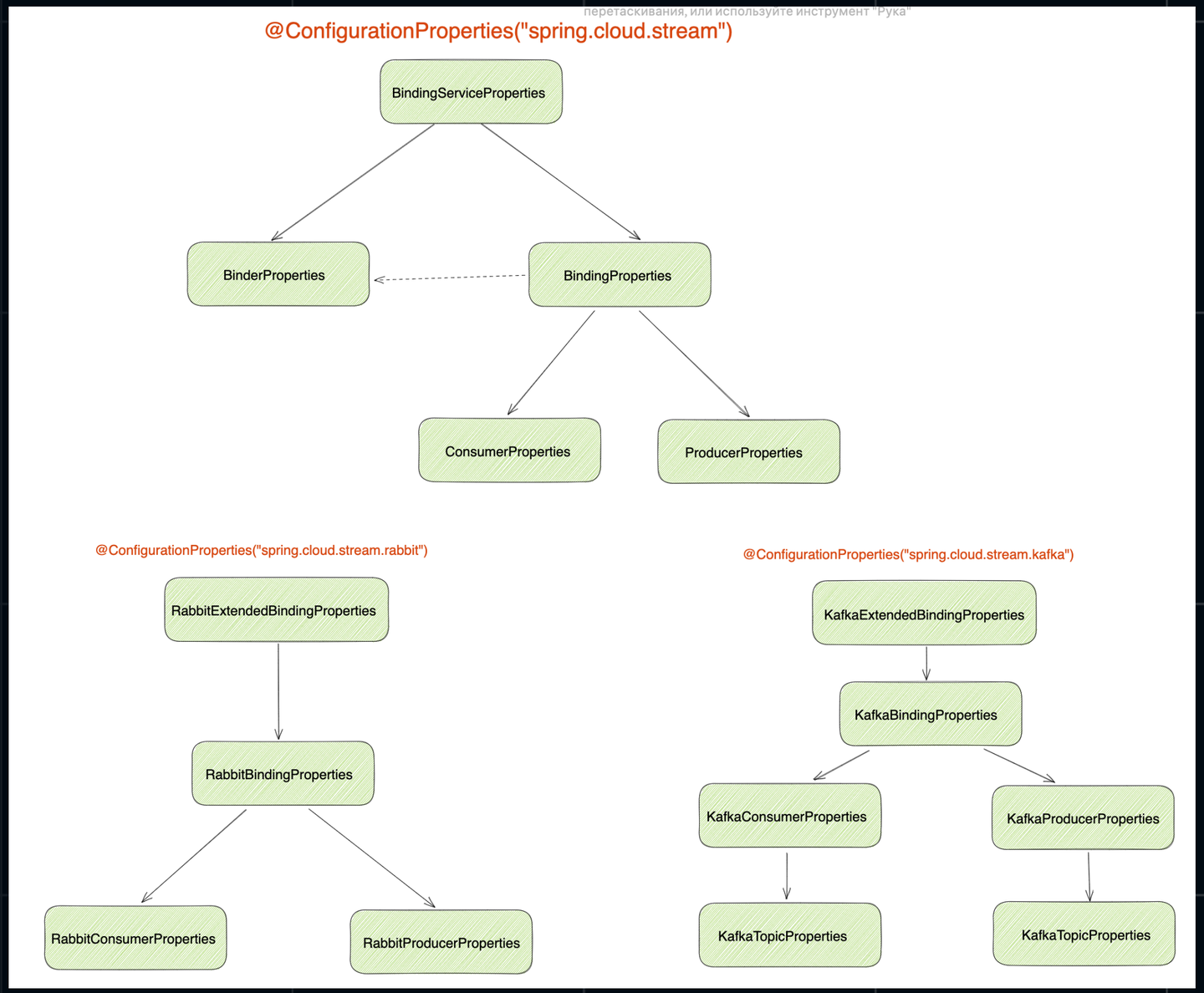
Rabbit specific:

- Политики Dead-Letter-Queue (DLQ) с time-to-live
- Тип exchange (direct/topic/fanout)

Kafka specific:

- Retention policy
- Compression type
- Количество партиций

Структура properties для Rabbit и Kafka своя. И она несимметрична



И отразится это в application.yml так:

```yaml
spring:
  cloud:
    stream:
      function:
        definition: event-processor;
      binders:
        event-rabbit-binder:
          type: rabbit
        event-kafka-binder:
          type: kafka
      bindings:
        event-processor-in-0:
        binder: event-rabbit-binder
          destination: something-state-change.exchange
          group: storage. something-state-queue
        storage-event-processor-out-0:
          binder: storage-kafka-binder
          destination: storage_events_topic
            producer:
            partition-count: 3
      rabbit:
        bindings:
          storage-event-processor-in-0:
            consumer:
              exchange-type: fanout
              declare-exchange: false
              bind-queue: true
              declare-dlx: true
              auto-bind-dlq: true
              dlq-ttl: 600000
      kafka:
        bindings:
          storage-event-processor-out-0:
            producer:
              topic:
                properties:
                  compression.type: lz4
                  retention.ms: 86400000
```

А как выглядит в коде?

```
@Component("event-processor")
class EventProcessor(
        private val eventService: EventService
) : Function<EventInDto, EventOutDto> {

    override fun apply(event: EventInDto) : EventOutDto {
        return eventService.doSomeWorkWithEvent(event.toModel()).toDto()
    }
}
```

Выводы по пропертям…

- Вся логика мигрировала в YML

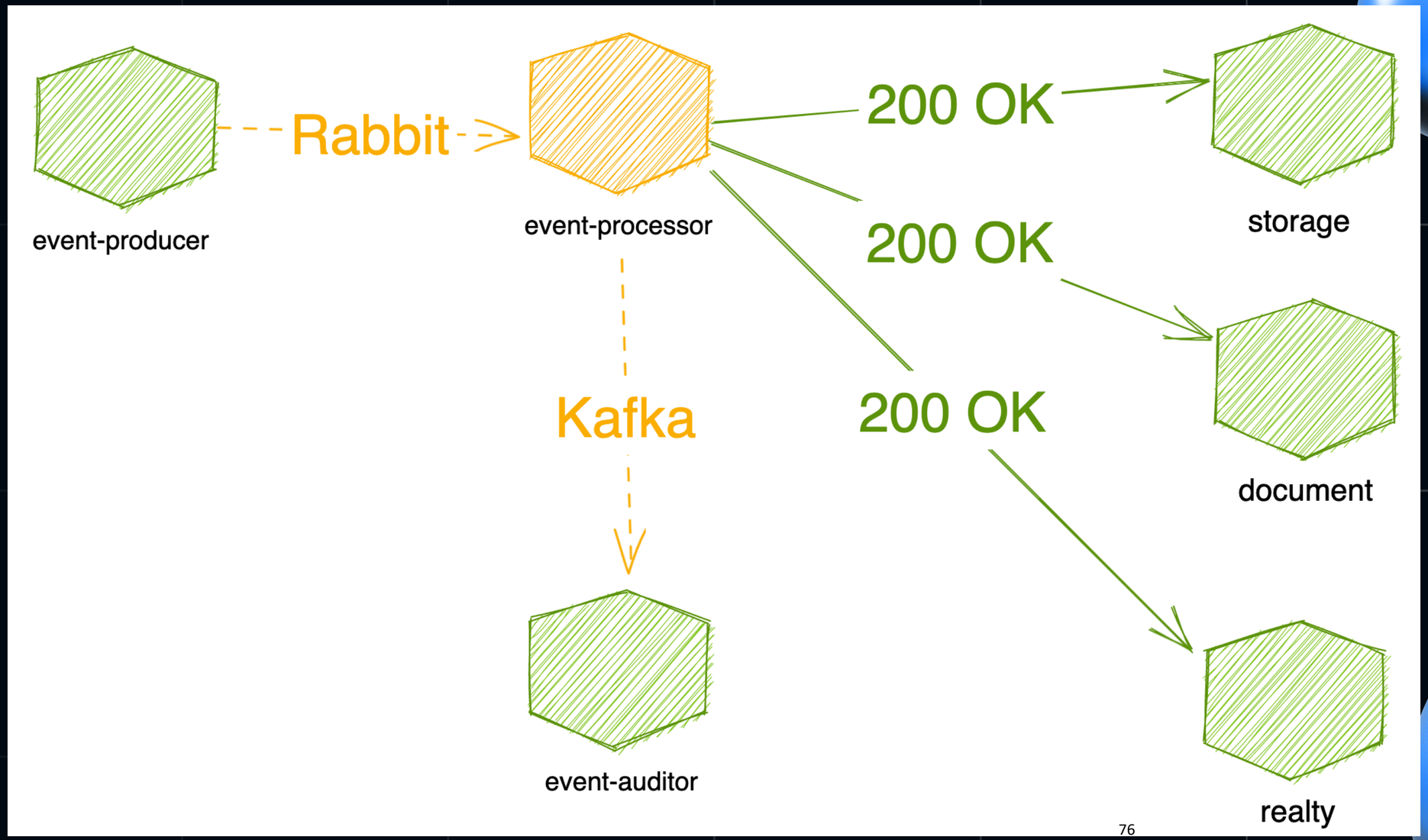
Выводы по пропертям…

- Вся логика мигрировала в YML
- Не всегда очевидно сразу в какой из @ConfigurationalProperties (общий, Rabbit, Kafka и тд) нужно добавлять ту или иную настройку

Выводы по пропертям…

- Вся логика мигрировала в YML
- Не всегда очевидно сразу в какой из @ConfigurationalProperties (общий, Rabbit, Kafka и тд) нужно добавлять ту или иную настройку
- Требует погружения на старте

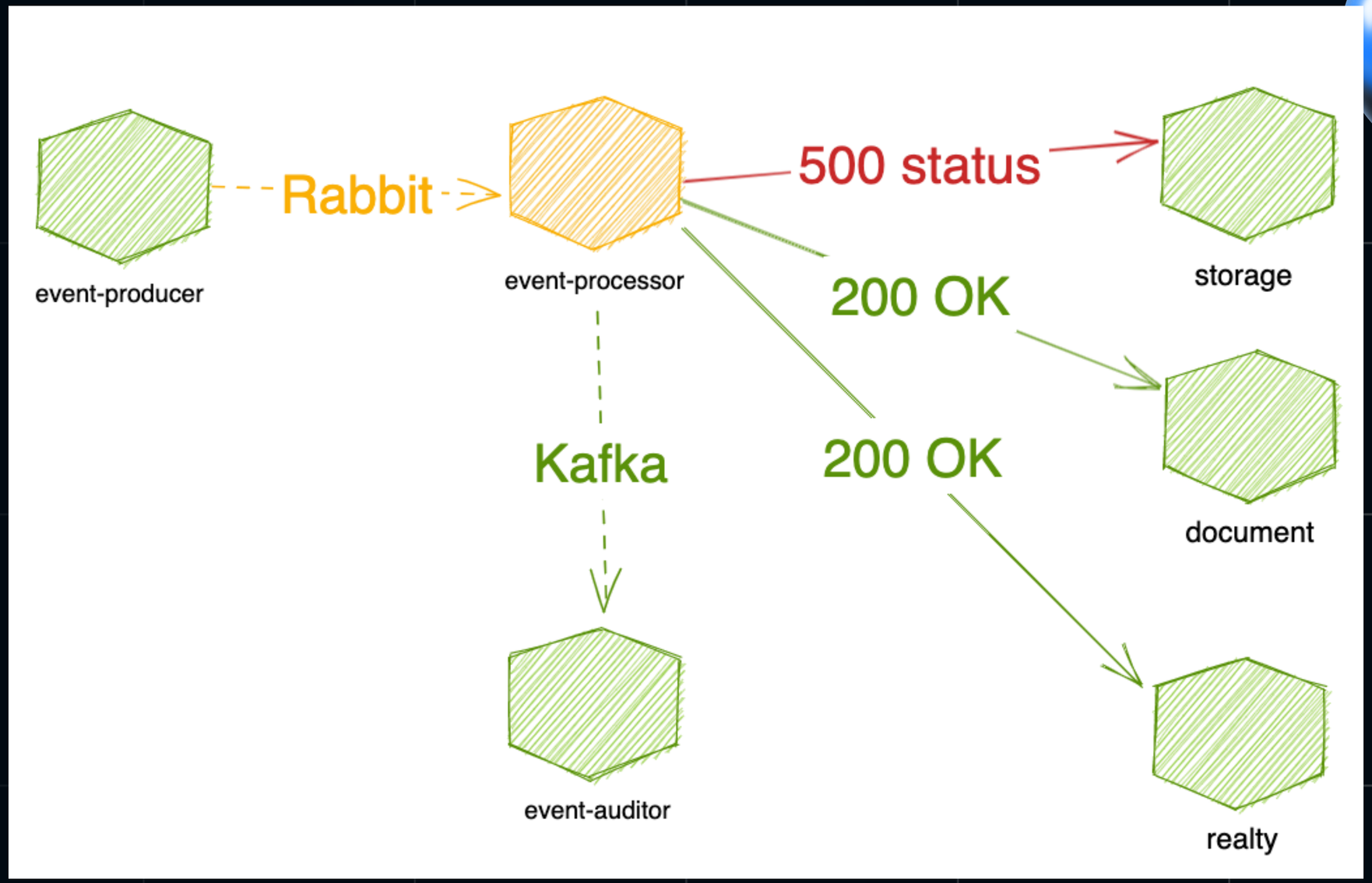
Движемся дальше…



Снова упал метеорит...

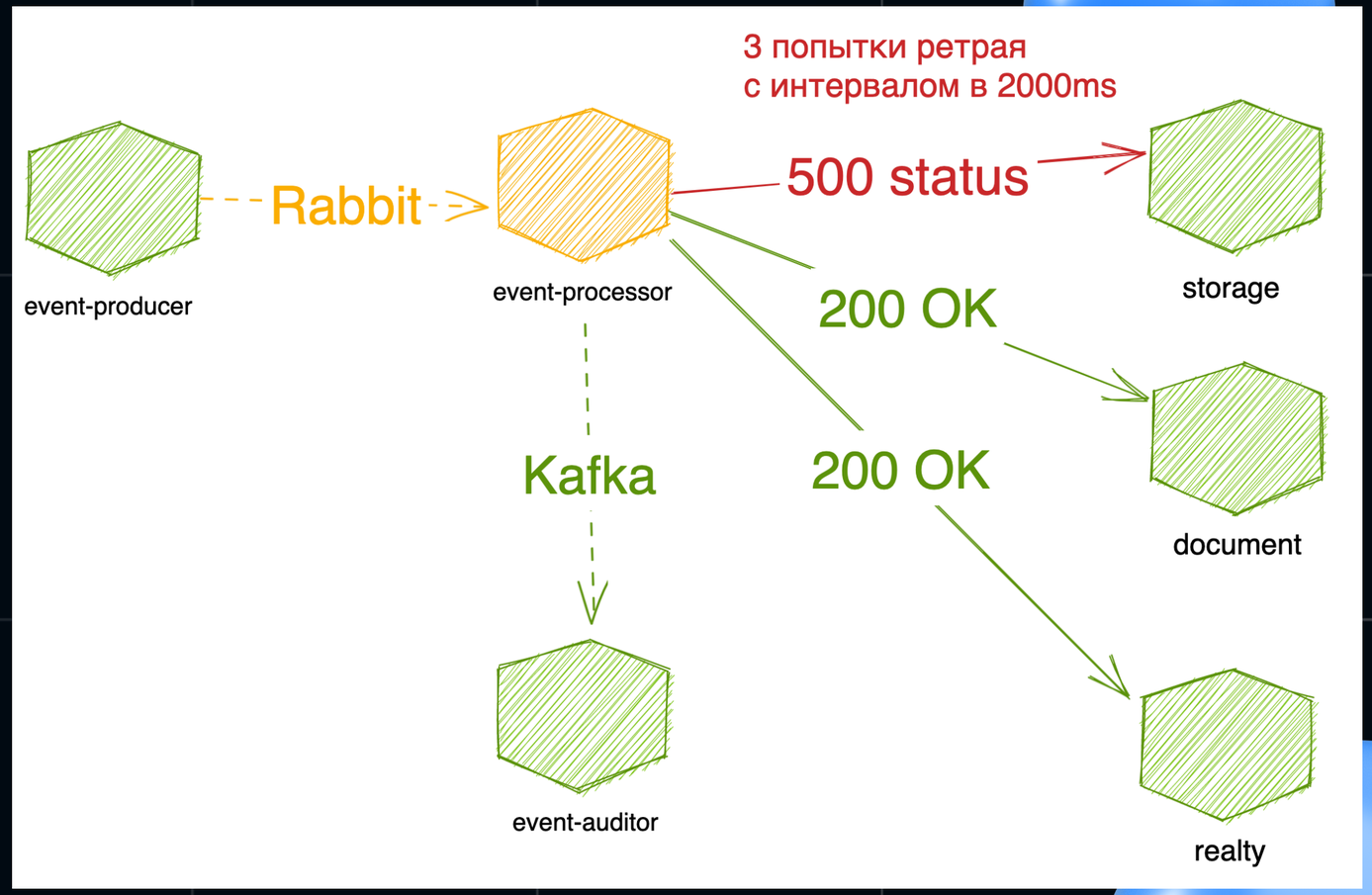
На самом деле релиз нет - у нас снова релиз

Случился сбой...



Случился сбой...

```yaml
# ==============================
# = Feign
# ==============================
feign:
  client:
    config:
     default:
        logger-level: FULL
        connectTimeout: 5000
        readTimeout: 5000
     storage-client:
        connectTimeout: 100
        readTimeout: 200
        retry-count: 3
        retry-backoff-time: 2000
     document-client:
        connectTimeout: 400
        readTimeout: 600
        retry-count: 3
        retry-backoff-time: 2000
     realty-client:
        connectTimeout: 1000
        readTimeout: 2000
        circuit-breaker-open-timeout: 5000
        circuit-breaker-reset-timeout: 20000
```



Смотрим мониторинг и логи

Что видно:

- Сервис storage видит повышенный фон обращений от event-processor

Смотрим мониторинг и логи

Что видно:

- Сервис storage видит повышенный фон обращений от event-processor
- Количество поступающих сообщений в очередь для event-processor в рамках нормы

Смотрим мониторинг и логи

Что видно:

- Сервис storage видит повышенный фон обращений от event-processor
- Количество поступающих сообщений в очередь для event-processor в рамках нормы
- Retry по 9 раз для каждого сообщения из очереди

Почему 9 ???

Откуда столько retry?

Откуда столько retry?

Путем дебага, находим что:

- В недрах Cloud Stream есть еще свои ретраи
- Используется тот же spring-retry
- Вызывается он 3 раза по дефолту

Откуда столько retry?

Путем дебага, находим что:

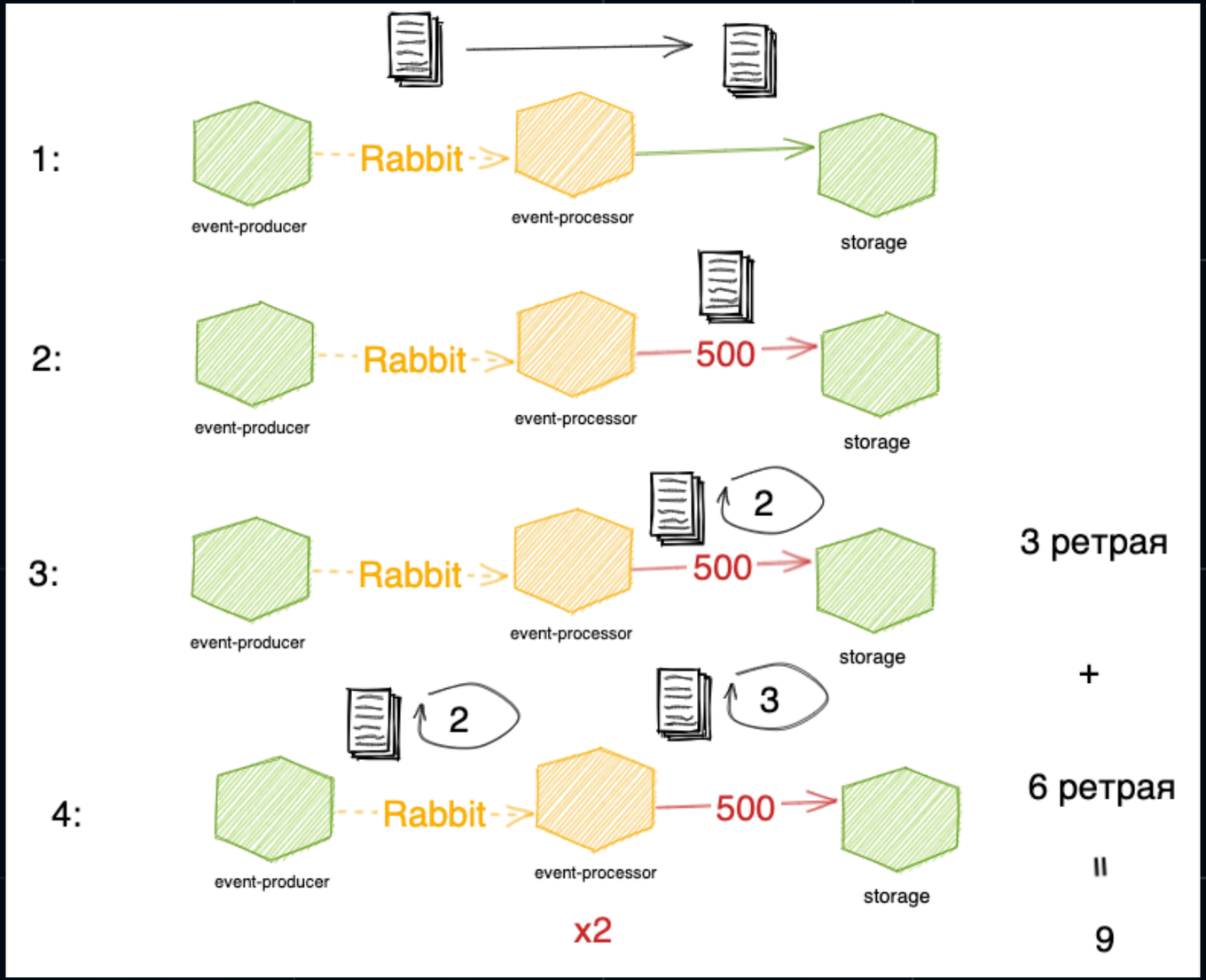
- В недрах Cloud Stream есть еще свои ретраи
- Используется тот же spring-retry
- Вызывается он 3 раза по дефолту

Оказывается, в ConsumerProperties есть такая штука:

```
The number of attempts to process the message (including the first) in the event of processing
failures. This is a RetryTemplate configuration which is provided by the framework. Default: 3. Set to 1
to disable retry. You can also provide custom RetryTemplate in the event you want to take complete
control of the RetryTemplate. Simply configure it as @Bean inside your application configuration.

2 usages
private int maxAttempts = 3;
```

Что ж 3x3 = 9…

Откуда столько retry?



Минутка субъективного и холиварного

По нашему опыту ценности в таких ретраях немного:

Минутка субъективного и холиварного

По нашему опыту ценности в таких ретраях немного:

- Это не вызов по http конкретного сервиса

Минутка субъективного и холиварного

По нашему опыту ценности в таких ретраях немного:

- Это не вызов по http конкретного сервиса
- Внешних интеграций в рамках обработки таких сообщение может быть много

Минутка субъективного и холиварного

По нашему опыту ценности в таких ретраях немного:

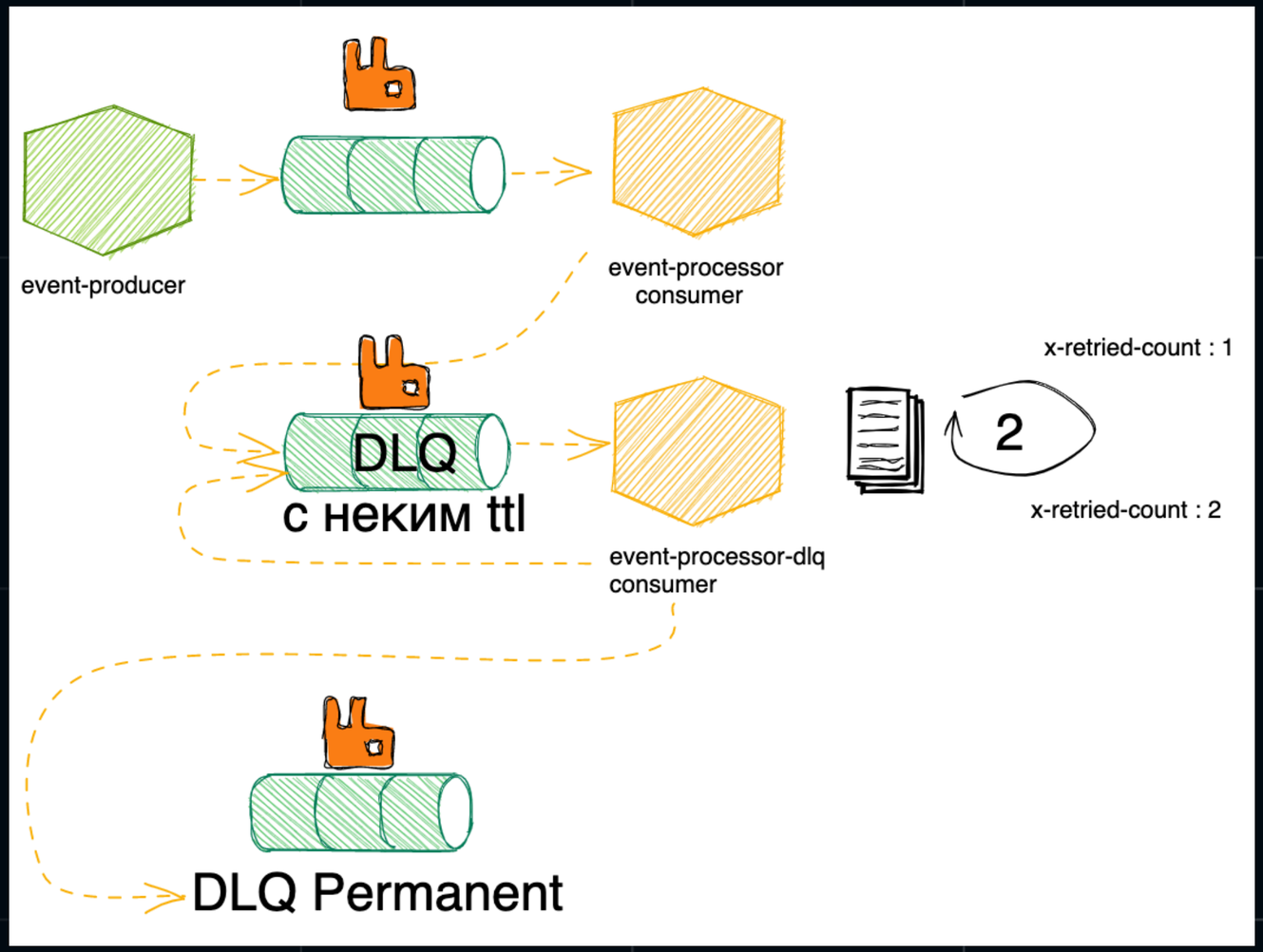
- Это не вызов по http конкретного сервиса
- Внешних интеграций в рамках обработки таких сообщение может быть много
- Ретрай происходит уже в рамках принимающего сервиса

Минутка субъективного и холиварного

По нашему опыту ценности в таких ретраях немного:

- Это не вызов по http конкретного сервиса
- Внешних интеграций в рамках обработки таких сообщение может быть много
- Ретрай происходит уже в рамках принимающего сервиса (ретрай по сути самого себя)
- Получаем мультипликатор нагрузки на сервис которому итак плохо

Альтернатива



Отложенные retry

В коде появится еще один consumer для dlq:

```
@Component(«event-processor-dlq»)
class EventProcessor(
        private val eventService: EventService
) : Function<Message<EventInDto>, Message<EventOutDto>?> {

    companion object : KLogging()

    override fun apply(event: EventInDto) : EventOutDto {
         If (message.headers[X_RETRIES_HEADER] == 2) return
         try {
          eventService.doSomeWorkWithEvent(event.toModel()).toDto()
         } catch (e: Exception) {
           ……
         }
         return MessageBuilder
                        .withPayload(message.payload)
                        .setHeader(X_RETRIES_HEADER, ++retries)
                  .build()
    }
}
```

Ну а в нашем application.yml еще блок с конфигурацией

Что еще интересного?

Что еще интересного?

Работа с routing-key для direct exchange

В application.yml:

```yaml
event-processor-out-0:
  producer:
    exchange-type: direct
    declare-exchange: false
    routing-key-expression: '''key_to_route'''
```

Что еще интересного?

Работа с routing-key для direct exchange

В application.yml:

```yaml
event-processor-out-0:
  producer:
    exchange-type: direct
    declare-exchange: false
    routing-key-expression: '''key_to_route'''
```

Сам же код процессора остается без изменений.

Роутинг на уровне конфигов

Итоги по Stream

Итоги по Stream

- Вся настройка в YML. В коде минимум работы с брокерами

Итоги по Stream

- Вся настройка в YML. В коде минимум работы с брокерами
- YML фалы профилей могут разрастаться до исполинских величин

Итоги по Stream

- Вся настройка в YML. В коде минимум работы с брокерами
- YML фалы профилей могут разрастаться до исполинских величин
- С 0 сложно разобраться с иерархией пропертей и кастомных настроек под каждый MQ

Итоги по Stream

- Вся настройка в YML. В коде минимум работы с брокерами
- YML фалы профилей могут разрастаться до исполинских величин
- С 0 сложно разобраться с иерархией пропертей и кастомных настроек под каждый MQ
- Могут быть сюрпризы в настройках по умолчанию и если некоторый настройки упустить

  Надо знать что под капотом и как эта штука работает

Теперь сервисы взаимодействуют друг с другом

Теперь сервисы взаимодействуют друг с другом

но!

Теперь сервисы взаимодействуют друг с другом

но!

- Как понять какой сервис сделал вызов?

Теперь сервисы взаимодействуют друг с другом

но!

- Как понять какой сервис сделал вызов?
- Какой сервис отправил эвент?

Теперь сервисы взаимодействуют друг с другом

но!

- Как понять какой сервис сделал вызов?
- Какой сервис отправил эвент?
- Логи в разных сервисах сложно совместить

Теперь сервисы взаимодействуют друг с другом

но!

- Как понять какой сервис сделал вызов?
- Какой сервис отправил эвент?
- Логи в разных сервисах сложно совместить
- Какой из нескольких внешних вызов вызвал интересующий нам вызов?

Пара слов про трейсинг!

Трейсинг

Такое тоже есть в Cloud

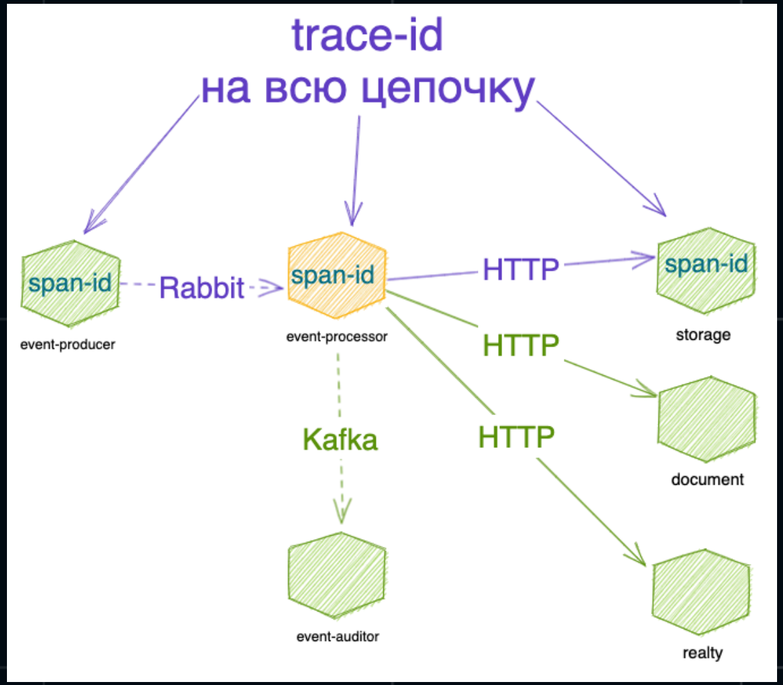
*implementation(“org.springframework.cloud:spring-cloud-starter-sleuth") ****

В Boot3 это уже работа с io.micrometer:micrometer-tracing

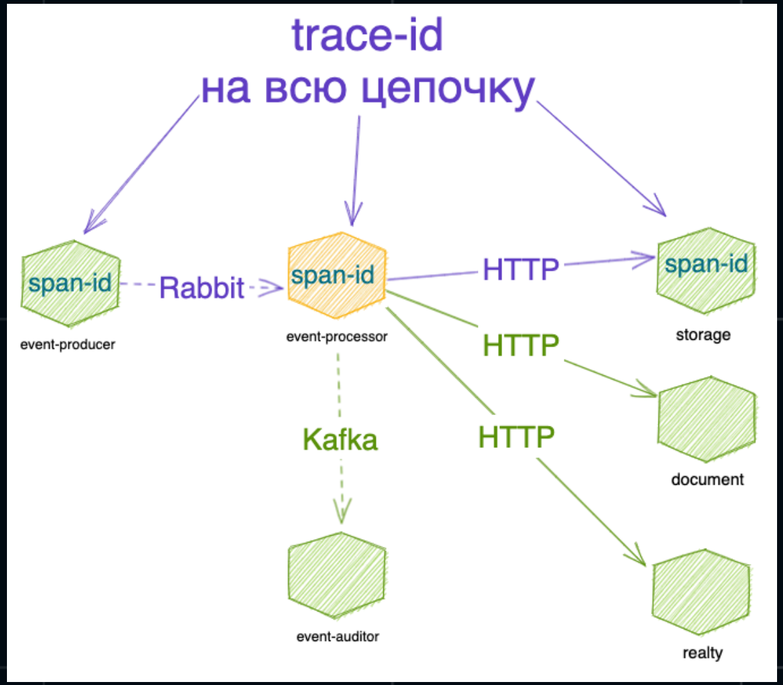
Тут без сюрпризов =) Работает с Cloud OpenFeign и Cloud Stream из коробки

А как?

Рассмотрим на примере



Рассмотрим на примере



>	Apr 17, 2023 @ 13:12:41.721	INFO	Start getBuilder()	50a8bec6fb2e73c2	861891cc7c770840
>	Apr 17, 2023 @ 13:12:41.720	INFO	Operation remove DocumentLinkEntity(id=32464)	50a8bec6fb2e73c2	ba594f5c946d1ccd
>	Apr 17, 2023 @ 13:12:41.718	INFO	Set header x-srv-user with value: 359942	50a8bec6fb2e73c2	50a8bec6fb2e73c2

Преимущества трейсинга

- У нас единый индекс/топик для групп микросервисов

Преимущества трейсинга

- У нас единый индекс/топик для групп микросервисов
- Возможность выбора - фильтровать по отдельному или нескольким сервисам

Преимущества трейсинга

- У нас единый индекс/топик для групп микросервисов
- Возможность выбора - фильтровать по отдельному или нескольким сервисам
- При инцидентах не тратится время на поиске нужного индекса в Kibana

Преимущества трейсинга

- У нас единый индекс/топик для групп микросервисов
- Возможность выбора - фильтровать по отдельному или нескольким сервисам
- При инцидентах не тратится время на поиске нужного индекса в Kibana
- Все ошибки группы в одном индексе

Преимущества трейсинга

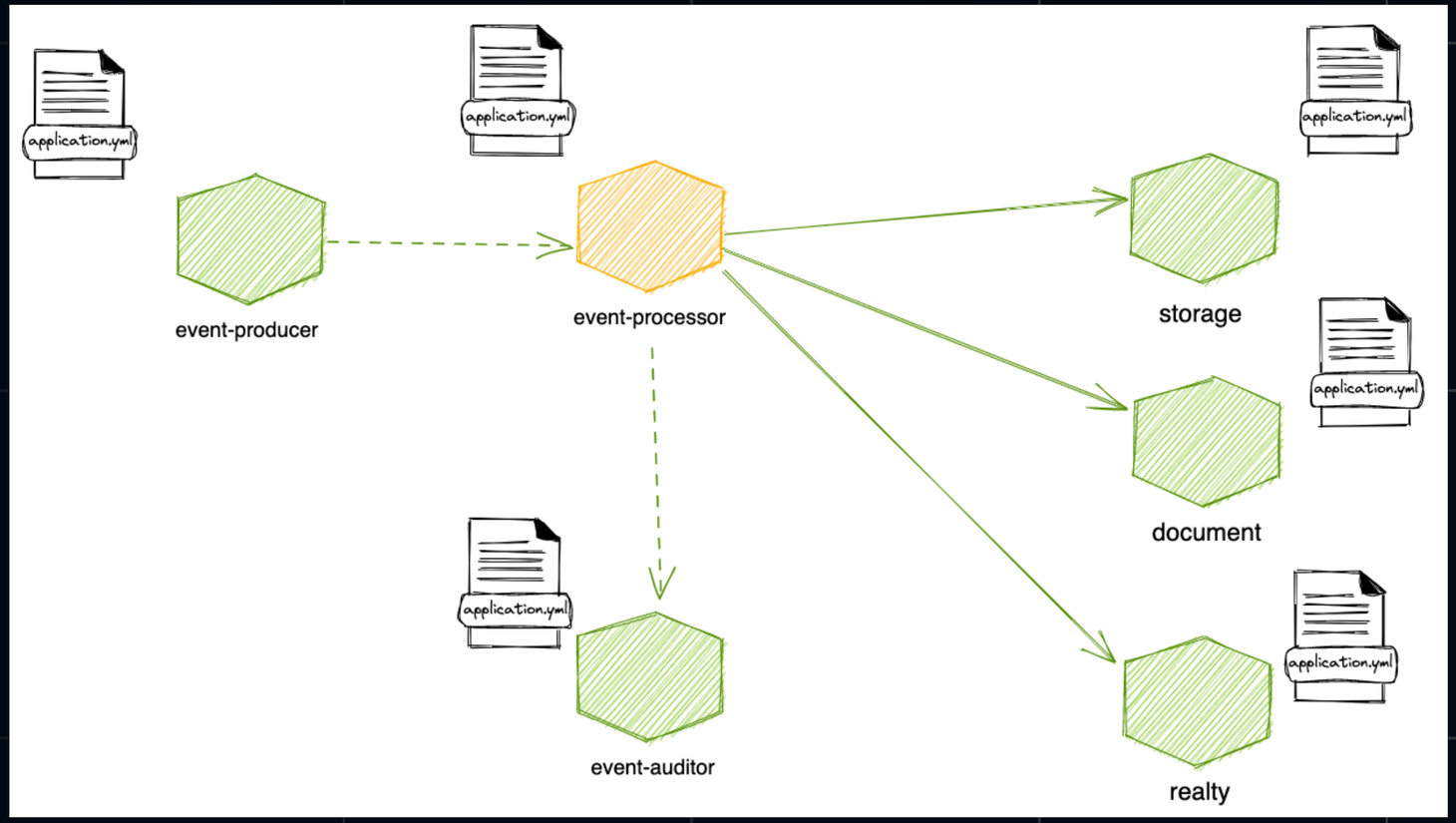
- У нас единый индекс/топик для групп микросервисов
- Возможность выбора - фильтровать по отдельному или нескольким сервисам
- При инцидентах не тратится время на поиске нужного индекса в Kibana
- Все ошибки группы в одном индексе
- При желании можно строить метрики и алертинг по индексу

Что дальше?

- Мы описали работу с Feign и Stream

Что дальше?

- Мы описали работу с Feign и Stream
- Настройки хранятся в application.yml



Внешние конфигурации

Задача

У нас много микросервисов. Нам надо поменять настройки в профиле для нескольких приложений:

- быстро

Задача

У нас много микросервисов. Нам надо поменять настройки в профиле для нескольких приложений:

- быстро
- без пересборки

Задача

У нас много микросервисов. Нам надо поменять настройки в профиле для нескольких приложений:

- быстро
- без пересборки
- без передеплоя

Задача

У нас много микросервисов. Нам надо поменять настройки в профиле для нескольких приложений:

- быстро
- без пересборки
- без передеплоя
- у нас Kubernetes

Что есть в Spring Cloud для этого?

Что есть в Spring Cloud для этого?

Spring Cloud Kubernetes

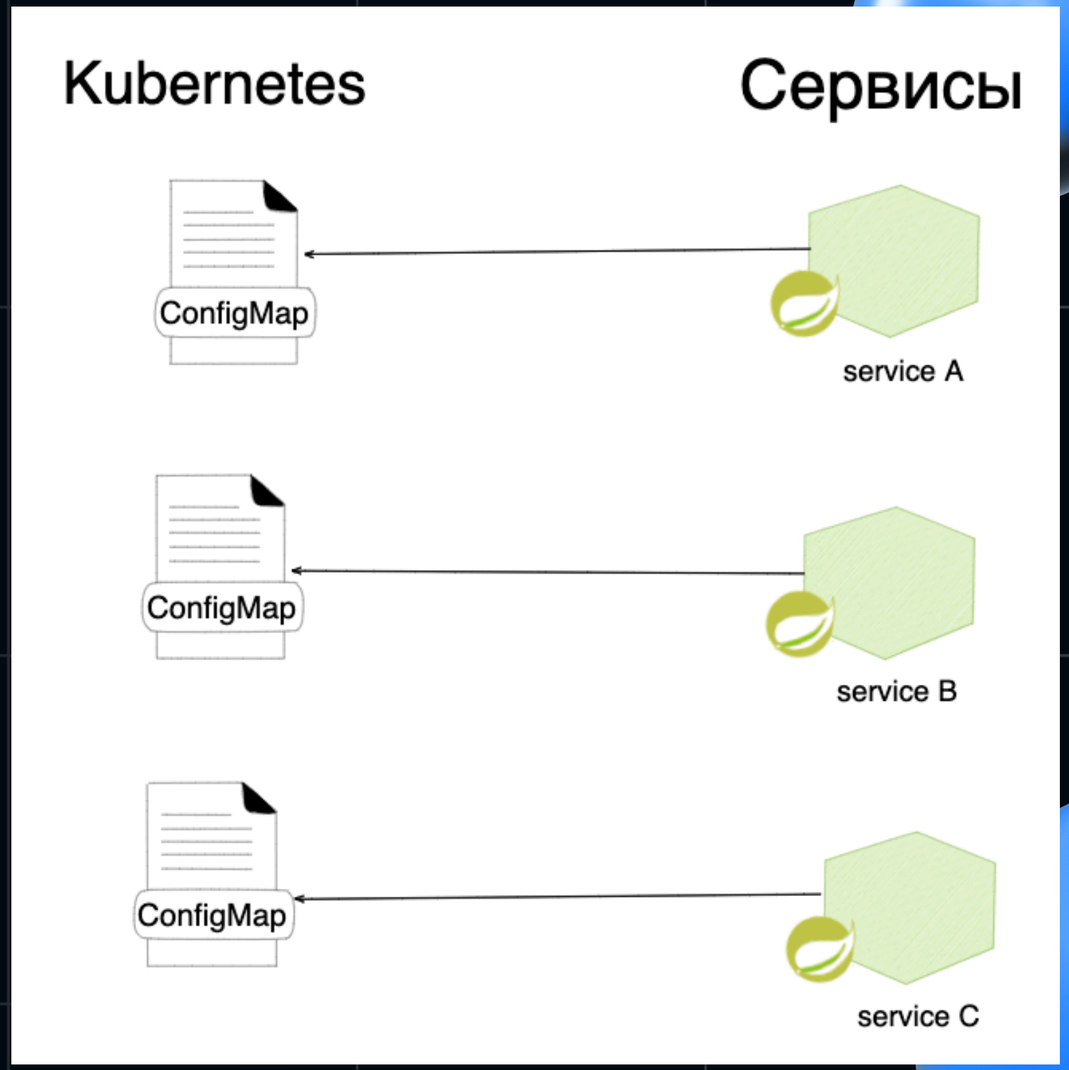
Импортируем:

implementation(“org.springframework.cloud:spring-cloud-starter-kubernetes-fabric8-config:”)

Еще немного теории как это работает

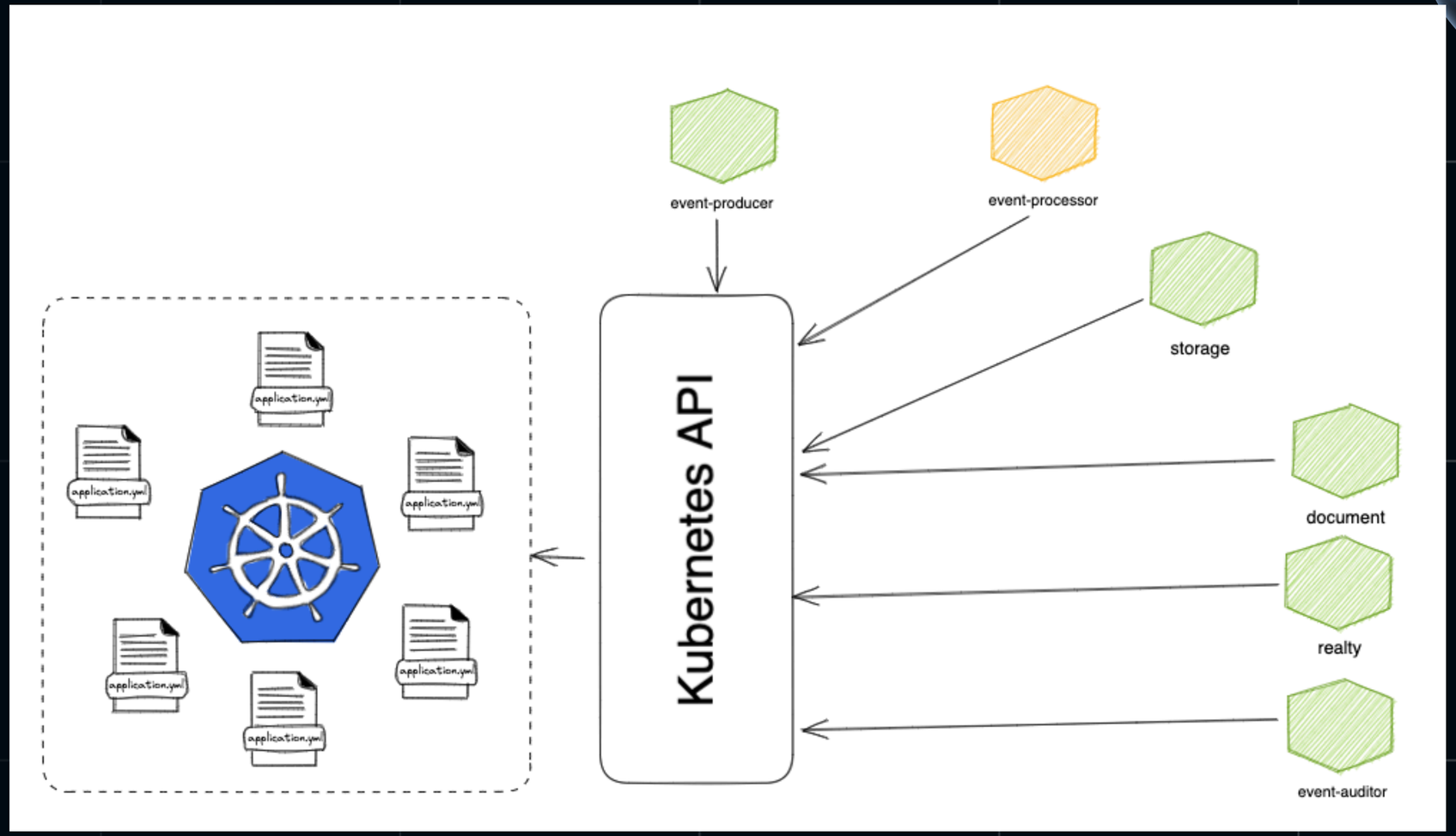
ConfigMap

Spring Cloud Kubernetes предлагаем нам хранить профили проложения application-prod.yml и подобное в самих ConfigMap



Минутка теории =)

Работает это все через Kubernetes API



Настраиваем окружение

Создаем и загружаем в ConfigMap профиль application-qa

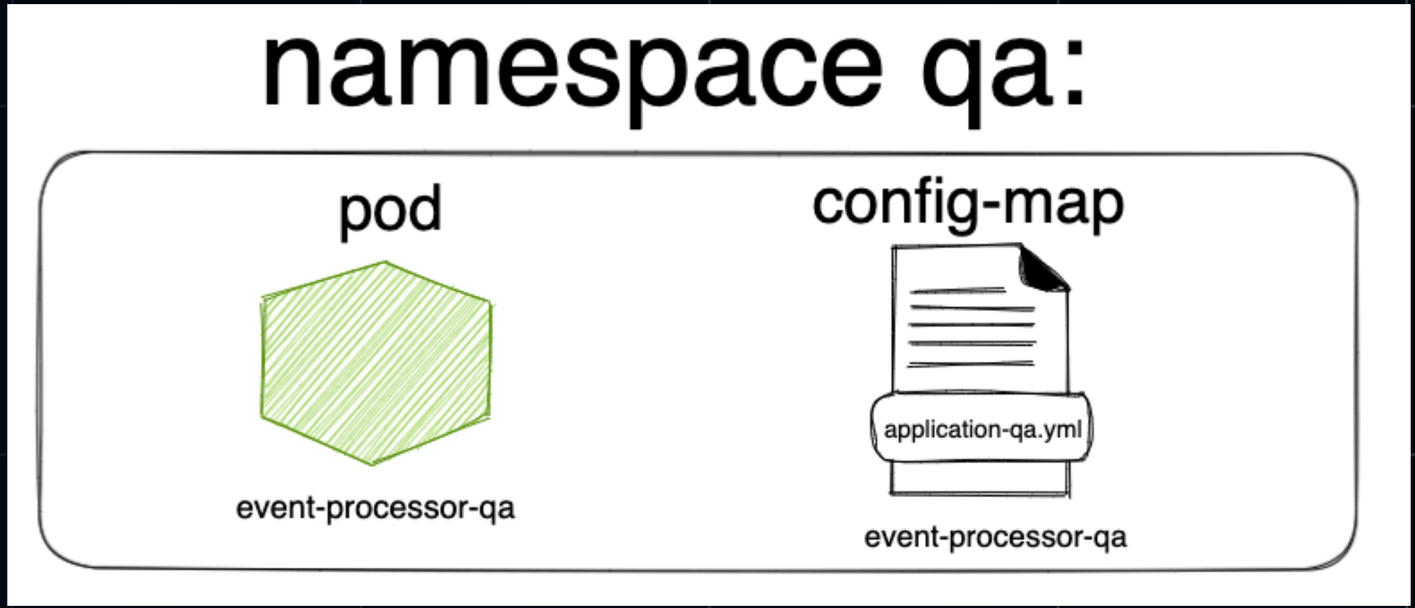
```
> kubectl config use-context qa
> kubectl create configmap event-processor—qa --from-file event-processor/profiles/application-qa.yml
```

```yml
# ==============================
# = General Settings
# ==============================
spring:
  config:
    activate:
      on-profile: qa
  datasource:
    url: jdbc:postgresql://some.host/event_processor_qa
    username: ep-user
    password: ${DB_PASSWORD}
  kafka:
    bootstrap-servers: kafka.dev.host:9092
  rabbitmq:
    host: rmqdev.host.ru
    port: 5673
    username: ep_dev
    password: ${RMQ_PASSWORD}"
```

***kubectl пол собой тоже использует

Kubernetes API

Настраиваем окружение



Configmap:

Jobs

Pods

Replica Sets

Replication Controllers

Stateful Sets

Service N

Ingresses

Services

Config and Storage

Config Maps N

Persistent Volume Claims N

Secrets N

Storage Classes

Data

```
1  {
2    "application-qa.yml": "# ================================
3      # = General Settings
4      # ================================
5      spring:
6        config:
7          activate:
8            on-profile: qa
9        datasource:
10         url: jdbc:postgresql://some.host/storage_qa
11         username: storage-user
12         password: ${DB_PASSWORD}
13       kafka:
14         bootstrap-servers: kafka.dev.host:9092
15       rabbitmq:
16         host: rmqdev.host.ru
17         port: 5673
18         username: storage_dev
19         password: ${RMQ_PASSWORD}\""
20 }
```

Запуск

Все профили загрузили. Пробуем стартануть…

Запуск

Все профили загрузили. Пробуем стартануть…

Нет доступа к чтению!

```
2023.03.12 17:41:59.482  WARN [storage,,] [main] org.springframework.cloud.kubernetes.fabric8.config.Fabric8ConfigMapPropertySource->getData: Can't read configMap with
name: [application] in namespace: [qa]. Ignoring.

io.fabric8.kubernetes.client.KubernetesClientException: Operation: [get]  for kind: [ConfigMap]  with name: [application]  in namespace: [qa]  failed.
    at io.fabric8.kubernetes.client.KubernetesClientException.launderThrowable(KubernetesClientException.java:64)
    at io.fabric8.kubernetes.client.KubernetesClientException.launderThrowable(KubernetesClientException.java:72)
    at io.fabric8.kubernetes.client.dsl.base.BaseOperation.getMandatory(BaseOperation.java:225)
    at io.fabric8.kubernetes.client.dsl.base.BaseOperation.get(BaseOperation.java:186)
    at io.fabric8.kubernetes.client.dsl.base.BaseOperation.get(BaseOperation.java:84)
    at org.springframework.cloud.kubernetes.fabric8.config.Fabric8ConfigUtils.getConfigMapData(Fabric8ConfigUtils.java:111)
    at org.springframework.cloud.kubernetes.fabric8.config.Fabric8ConfigMapPropertySource.getData(Fabric8ConfigMapPropertySource.java:70)
    at org.springframework.cloud.kubernetes.fabric8.config.Fabric8ConfigMapPropertySource.<init>(Fabric8ConfigMapPropertySource.java:64)
    at org.springframework.cloud.kubernetes.fabric8.config.Fabric8ConfigMapPropertySourceLocator.getMapPropertySource(Fabric8ConfigMapPropertySourceLocator.java:72)
    at
```

Разбираемся…Доки молчат…

Разбираемся…Доки молчат…

Путем дебага находим, что при старте приложения используется

```
io.fabric8.kubernetes.client.Config
```

Разбираемся…Доки молчат…

Путем дебага находим, что при старте приложения используется

`io.fabric8.kubernetes.client.Config`

Метод:

```
private static Config autoConfigure(Config config, String context) {
  if (!tryKubeConfig(config, context)) {
    tryServiceAccount(config);
    tryNamespaceFromPath(config);
  }
  configFromSysPropsOrEnvVars(config);

  config.masterUrl = ensureHttps(config.masterUrl, config);
  config.masterUrl = ensureEndsWithSlash(config.masterUrl);

  return config;
}
```

Разбираемся…Доки молчат…

Путем дебага находим, что при старте приложения используется

`io.fabric8.kubernetes.client.Config`

Метод:

```
private static Config autoConfigure(Config config, String context) {
  if (!tryKubeConfig(config, context)) {
    tryServiceAccount(config);
    tryNamespaceFromPath(config);
  }
  configFromSysPropsOrEnvVars(config);

  config.masterUrl = ensureHttps(config.masterUrl, config);
  config.masterUrl = ensureEndsWithSlash(config.masterUrl);

  return config;
}
```

Получается, что

- конфигурирование клиента может быть из 3 мест

А затем дообогащается :

- System properties
- ENV variables

Разбираемся…Доки молчат…

Путем дебага находим, что при старте приложения используется

`io.fabric8.kubernetes.client.Config`

Метод:

```
private static Config autoConfigure(Config config, String context) {
  if (!tryKubeConfig(config, context)) {
    tryServiceAccount(config);
    tryNamespaceFromPath(config);
  }
  configFromSysPropsOrEnvVars(config);

  config.masterUrl = ensureHttps(config.masterUrl, config);
  config.masterUrl = ensureEndsWithSlash(config.masterUrl);

  return config;
}
```

Получается, что

- конфигурирование клиента может быть из 3 мест

А затем дообогащается :

- System properties
- ENV variables

Ну а наш token не имеет прав для чтения данных из поды куба

Смотрим что у нас в configFromSysPropsOrEnvVars:

```
config.setTrustStorePassphrase(Utils.getSystemPropertyOrEnvVar(KUBERNETES_TRUSTSTORE_PASSPHRASE_PROPERTY, config.getTrustStoreP
config.setTrustStoreFile(Utils.getSystemPropertyOrEnvVar(KUBERNETES_TRUSTSTORE_FILE_PROPERTY, config.getTrustStoreFile()));
config.setKeyStorePassphrase(Utils.getSystemPropertyOrEnvVar(KUBERNETES_KEYSTORE_PASSPHRASE_PROPERTY, config.getKeyStorePassph
config.setKeyStoreFile(Utils.getSystemPropertyOrEnvVar(KUBERNETES_KEYSTORE_FILE_PROPERTY, config.getKeyStoreFile()));

config.setOauthToken(Utils.getSystemPropertyOrEnvVar(KUBERNETES_OAUTH_TOKEN_SYSTEM_PROPERTY, config.getOauthToken()));
config.setUsername(Utils.getSystemPropertyOrEnvVar(KUBERNETES_AUTH_BASIC_USERNAME_SYSTEM_PROPERTY, config.getUsername()));
config.setPassword(Utils.getSystemPropertyOrEnvVar(KUBERNETES_AUTH_BASIC_PASSWORD_SYSTEM_PROPERTY, config.getPassword()));

config.setImpersonateUsername(Utils.getSystemPropertyOrEnvVar(KUBERNETES_IMPERSONATE_USERNAME, config.getImpersonateUsername()
```

Интересная строка =) А что в константе?

Смотрим что у нас в configFromSysPropsOrEnvVars:

```
config.setTrustStorePassphrase(Utils.getSystemPropertyOrEnvVar(KUBERNETES_TRUSTSTORE_PASSPHRASE_PROPERTY, config.getTrustStoreP
config.setTrustStoreFile(Utils.getSystemPropertyOrEnvVar(KUBERNETES_TRUSTSTORE_FILE_PROPERTY, config.getTrustStoreFile()));
config.setKeyStorePassphrase(Utils.getSystemPropertyOrEnvVar(KUBERNETES_KEYSTORE_PASSPHRASE_PROPERTY, config.getKeyStorePassph
config.setKeyStoreFile(Utils.getSystemPropertyOrEnvVar(KUBERNETES_KEYSTORE_FILE_PROPERTY, config.getKeyStoreFile()));

config.setOauthToken(Utils.getSystemPropertyOrEnvVar(KUBERNETES_OAUTH_TOKEN_SYSTEM_PROPERTY, config.getOauthToken()));
config.setUsername(Utils.getSystemPropertyOrEnvVar(KUBERNETES_AUTH_BASIC_USERNAME_SYSTEM_PROPERTY, config.getUsername()));
config.setPassword(Utils.getSystemPropertyOrEnvVar(KUBERNETES_AUTH_BASIC_PASSWORD_SYSTEM_PROPERTY, config.getPassword()));

config.setImpersonateUsername(Utils.getSystemPropertyOrEnvVar(KUBERNETES_IMPERSONATE_USERNAME, config.getImpersonateUsername()
```

Интересная строка =) А что в константе?

```
3 usages
public static final String KUBERNETES_AUTH_BASIC_USERNAME_SYSTEM_PROPERTY = "kubernetes.auth.basic.username";
3 usages
public static final String KUBERNETES_AUTH_BASIC_PASSWORD_SYSTEM_PROPERTY = "kubernetes.auth.basic.password";
1 usage
public static final String KUBERNETES_AUTH_TRYKUBECONFIG_SYSTEM_PROPERTY = "kubernetes.auth.tryKubeConfig";
1 usage
public static final String KUBERNETES_AUTH_TRYSERVICEACCOUNT_SYSTEM_PROPERTY = "kubernetes.auth.tryServiceAccount";
3 usages
public static final String KUBERNETES_OAUTH_TOKEN_SYSTEM_PROPERTY = "kubernetes.auth.token";
```

Смотрим что у нас в configFromSysPropsOrEnvVars:

```
config.setTrustStorePassphrase(Utils.getSystemPropertyOrEnvVar(KUBERNETES_TRUSTSTORE_PASSPHRASE_PROPERTY, config.getTrustStoreP
config.setTrustStoreFile(Utils.getSystemPropertyOrEnvVar(KUBERNETES_TRUSTSTORE_FILE_PROPERTY, config.getTrustStoreFile()));
config.setKeyStorePassphrase(Utils.getSystemPropertyOrEnvVar(KUBERNETES_KEYSTORE_PASSPHRASE_PROPERTY, config.getKeyStorePassph
config.setKeyStoreFile(Utils.getSystemPropertyOrEnvVar(KUBERNETES_KEYSTORE_FILE_PROPERTY, config.getKeyStoreFile()));

config.setOauthToken(Utils.getSystemPropertyOrEnvVar(KUBERNETES_OAUTH_TOKEN_SYSTEM_PROPERTY, config.getOauthToken()));
config.setUsername(Utils.getSystemPropertyOrEnvVar(KUBERNETES_AUTH_BASIC_USERNAME_SYSTEM_PROPERTY, config.getUsername()));
config.setPassword(Utils.getSystemPropertyOrEnvVar(KUBERNETES_AUTH_BASIC_PASSWORD_SYSTEM_PROPERTY, config.getPassword()));

config.setImpersonateUsername(Utils.getSystemPropertyOrEnvVar(KUBERNETES_IMPERSONATE_USERNAME, config.getImpersonateUsername()
```

Интересная строка =) А что в константе?

```
3 usages
public static final String KUBERNETES_AUTH_BASIC_USERNAME_SYSTEM_PROPERTY = "kubernetes.auth.basic.username";
3 usages
public static final String KUBERNETES_AUTH_BASIC_PASSWORD_SYSTEM_PROPERTY = "kubernetes.auth.basic.password";
1 usage
public static final String KUBERNETES_AUTH_TRYKUBECONFIG_SYSTEM_PROPERTY = "kubernetes.auth.tryKubeConfig";
1 usage
public static final String KUBERNETES_AUTH_TRYSERVICEACCOUNT_SYSTEM_PROPERTY = "kubernetes.auth.tryServiceAccount";
3 usages
public static final String KUBERNETES_OAUTH_TOKEN_SYSTEM_PROPERTY = "kubernetes.auth.token";
```

Ну и в самом методе:

```
41 usages
public static String getSystemPropertyOrEnvVar(String systemPropertyName, String defaultValue) {
  return getSystemPropertyOrEnvVar(systemPropertyName, convertSystemPropertyNameToEnvVar(systemPropertyName), defaultValue);
}
```

Промежуточный итог

Промежуточный итог

Надо:

- Сгенерировать новый token для доступа в Kube API

Промежуточный итог

Надо:

- Сгенерировать новый token для доступа в Kube API
- Указать token как Environment variable:

  KUBERNETES_AUTH_TOKEN=<token>

Промежуточный итог

Надо:

- Сгенерировать новый token для доступа в Kube API
- Указать token как Environment variable:

  KUBERNETES_AUTH_TOKEN=<token>
- Быть внимательным с настройками доступа

  Fabric8 client умеет не только читать но и модифицировать

  (можно делать все что угодно, даже разворотить куб через API)

Промежуточный итог

Надо:

- Сгенерировать новый token для доступа в Kube API
- Указать token как Environment variable:

  KUBERNETES_AUTH_TOKEN=<token>
- Быть внимательным с настройками доступа

  Fabric8 client умеет не только читать но и модифицировать

  (можно делать все что угодно, даже разворотить куб через API)
- Могут быть сюрпризы в новых версиях Cloud с доступами или

  еще с чем

Сгенерировали. Запускаем снова

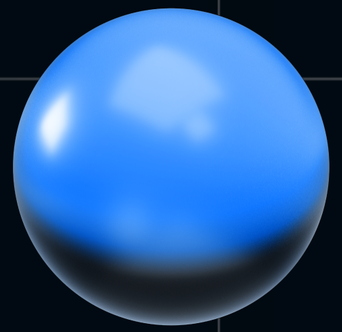
Сгенерировали. Запускаем снова

Тут уже ловим стандартную типовую ошибку

```
2023.03.12 17:41:59.482  WARN [storage,,] [main] org.springframework.cloud.kubernetes.fabric8.config.Fabric8ConfigMapPropertySource->getData: Can't read configMap
with name: [application] in namespace: [qa]. Ignoring.

io.fabric8.kubernetes.client.KubernetesClientException: Operation: [get]  for kind: [ConfigMap]  with name: [application]  in namespace: [qa]  failed.
    at io.fabric8.kubernetes.client.KubernetesClientException.launderThrowable(KubernetesClientException.java:64)
    at io.fabric8.kubernetes.client.KubernetesClientException.launderThrowable(KubernetesClientException.java:72)
    at io.fabric8.kubernetes.client.dsl.base.BaseOperation.getMandatory(BaseOperation.java:225)
    at io.fabric8.kubernetes.client.dsl.base.BaseOperation.get(BaseOperation.java:186)
    at io.fabric8.kubernetes.client.dsl.base.BaseOperation.get(BaseOperation.java:84)
    at org.springframework.cloud.kubernetes.fabric8.config.Fabric8ConfigUtils.getConfigMapData(Fabric8ConfigUtils.java:111)
    at org.springframework.cloud.kubernetes.fabric8.config.Fabric8ConfigMapPropertySource.getData(Fabric8ConfigMapPropertySource.java:70)
    at org.springframework.cloud.kubernetes.fabric8.config.Fabric8ConfigMapPropertySource.<init>(Fabric8ConfigMapPropertySource.java:64)
    at
```

Загрузили мы: **application-qa**.yml

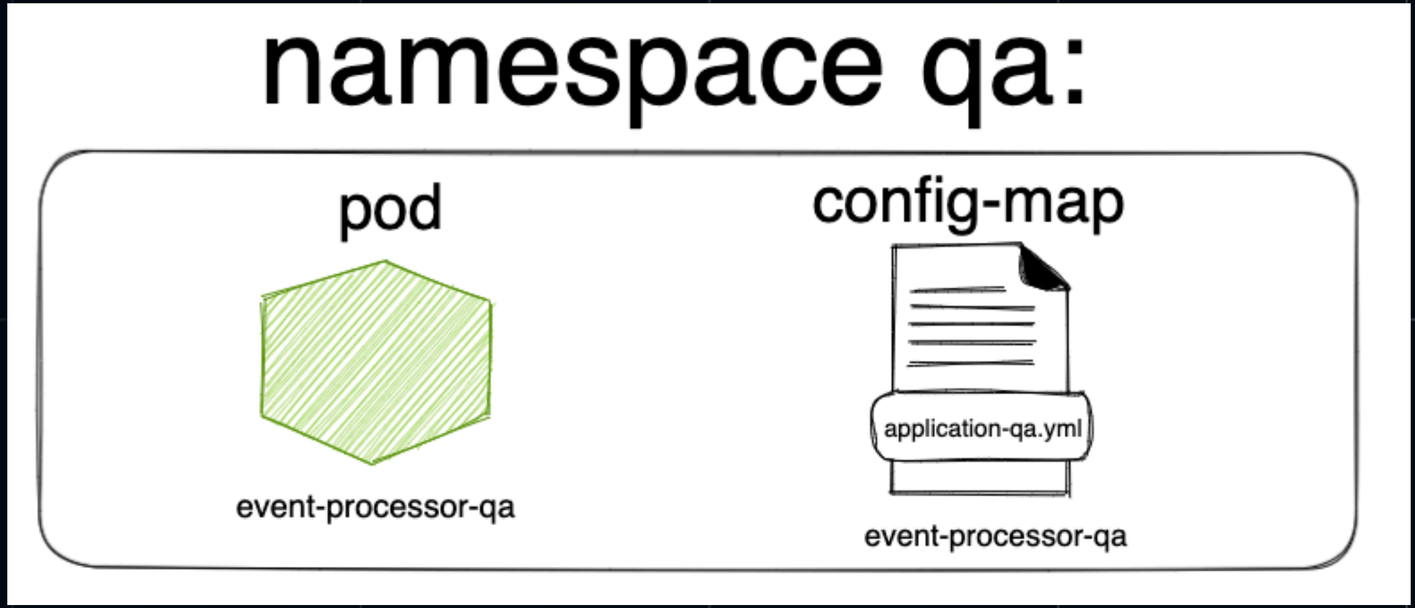
А конфигмап назвали: **event-processor-qa**



Загрузили мы: **application-qa**.yml

А конфигмап назвали: **event-processor-qa**

```
> kubectl config use-context qa
> kubectl create configmap event-processor—qa --from-file event-processor/profiles/application-qa.yml
```



Почему Spring Cloud Kubernetes ищет ConfigMap именно с названием application?

Почему Spring Cloud Kubernetes ищет ConfigMap именно с названием application?

Посмотрим в исходники, туда откуда вылетела ошибка

Почему Spring Cloud Kubernetes ищет ConfigMap именно с названием application?

Посмотрим в исходники, туда откуда вылетела ошибка

Fabric8ConfigMapPropertySource

```
public Fabric8ConfigMapPropertySource(KubernetesClient client, String applicationName, String namespace,
        Environment environment, String prefix) {
    super(getName(applicationName, getApplicationNamespace(client, namespace)),
            getData(client, applicationName, getApplicationNamespace(client, namespace), environment, prefix));
}

2 usages
private static Map<String, Object> getData(KubernetesClient client, String applicationName, String namespace,
        Environment environment, String prefix) {
    try {
        Map<String, String> data = getConfigMapData(client, namespace, applicationName);
        Map<String, Object> result = new HashMap<>(processAllEntries(data, environment));

        if (environment != null) {
            for (String activeProfile : environment.getActiveProfiles()) {
                String mapNameWithProfile = applicationName + "-" + activeProfile;
                Map<String, String> dataWithProfile = getConfigMapData(client, namespace, mapNameWithProfile);
                result.putAll(processAllEntries(dataWithProfile, environment));
            }
        }

        if (!"".equals(prefix)) {
            Map<String, Object> withPrefix = CollectionUtils.newHashMap(result.size());
            result.forEach((key, value) -> withPrefix.put(prefix + "." + key, value));
            return withPrefix;
        }

        return result;

    }
    catch (Exception e) {
        LOG.warn( message: "Can't read configMap with name: [" + applicationName + "] in namespace: [" + namespace
                + "]. Ignoring.", e);
    }

    return Collections.emptyMap();
}
```

Резюмируем

- Сначала определяется namespace в котором запускаемся

Резюмируем

- Сначала определяется namespace в котором запускаемся
- Потом определяется applicationName по значению

  spring.application.name

  и если такого нет то берется «**application**» по умолчанию

Резюмируем

- Сначала определяется namespace в котором запускаемся
- Потом определяется applicationName по значению

  spring.application.name

  и если такого нет то берется «**application**» по умолчанию
- Определяется постфикс к ***applicationName***

  по профилю приложения (-qal-stage и тд) и дефолтовым профилем с названием kubernetes

  то есть можно подтянуть N профилей kubernetes+qa I stage I prod

Почему название приложения и профиль важны?

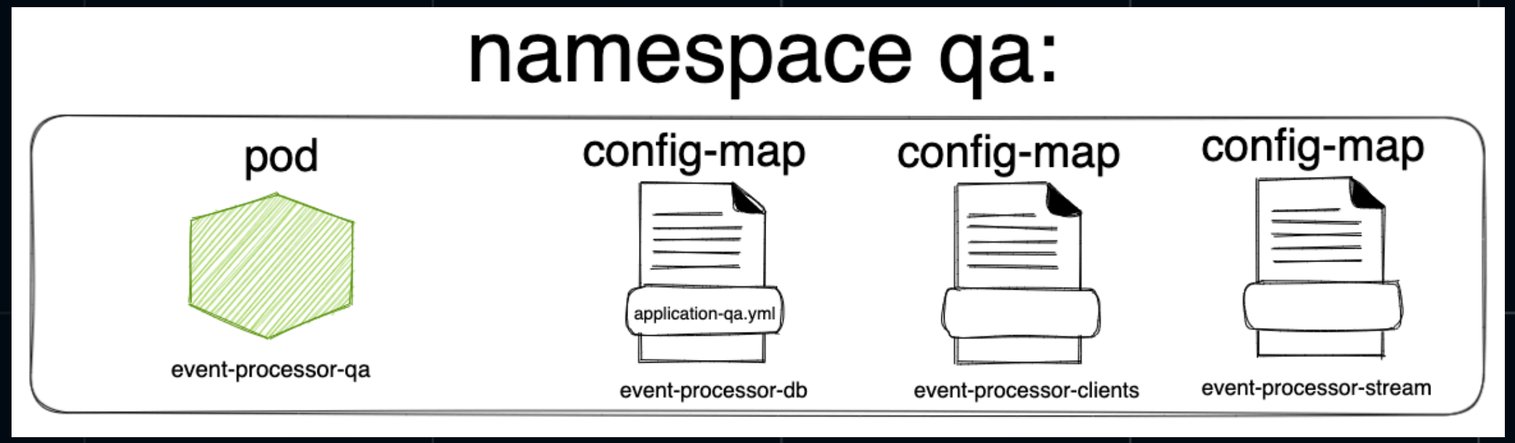
- У нас в одном namespace развернуто несколько микросервисов

Почему название приложения и профиль важны?

- У нас в одном namespace развернуто несколько микросервисов
- Нет риска подтянуть чужой профиль через неверное название configmap

Почему название приложения и профиль важны?

- У нас в одном namespace развернуто несколько микросервисов
- Нет риска подтянуть чужой профиль через неверное название configmap
- Возможность распилить один огромный профиль приложения на части db I feign I stream и обновлять их частями



Выкатились

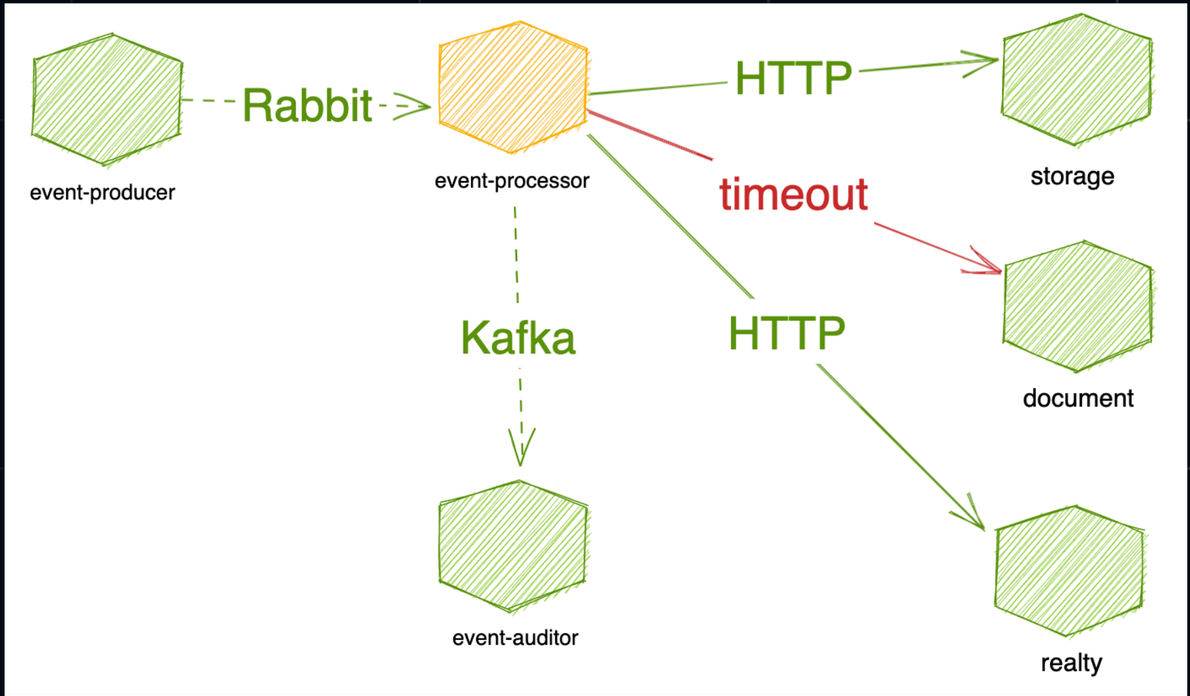
Живем нормально. Профили тянутся

Опять ошибки!

Выкатились

Живем нормально. Профили тянутся

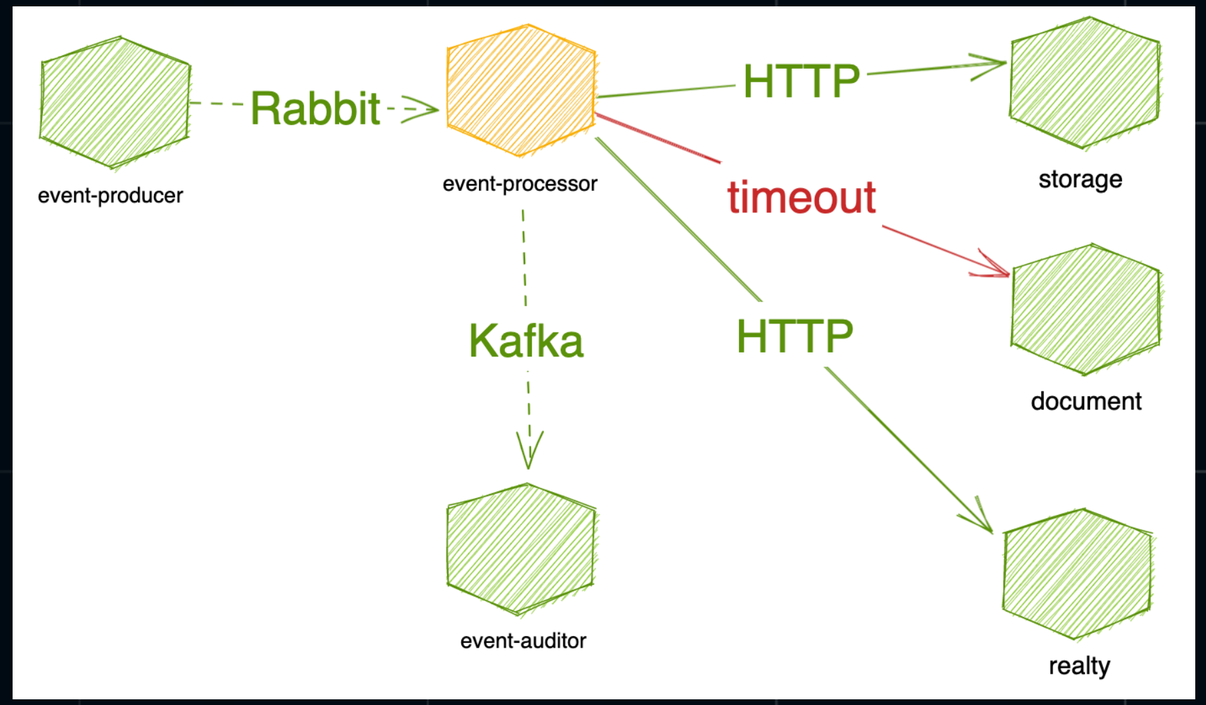
Снова кто то выкатился



Выкатились

Живем нормально. Профили тянутся

Снова кто то выкатился



На этот раз отличился сервис document. Таймаутит…

Быстро увеличим таймаут

Ок, а что будет если обновить профиль?

```yaml
# ==============================
# = Feign
# ==============================
feign:
  client:
    config:
     default:
        logger-level: FULL
        connectTimeout: 5000
        readTimeout: 5000
     storage-client:
        connectTimeout: 100
        readTimeout: 200
         retry-count: 3
         retry-backoff-time: 2000
     document-client:
        connectTimeout: 800
        readTimeout: 1500
        retry-count: 3
        retry-backoff-time: 3000
     realty-client:
        connectTimeout: 1000
        readTimeout: 2000
        circuit-breaker-open-timeout: 5000
        circuit-breaker-reset-timeout: 20000
```

Ок, а что будет если обновить профиль?

```yaml
# ==============================
# = Feign
# ==============================
feign:
  client:
    config:
     default:
        logger-level: FULL
        connectTimeout: 5000
        readTimeout: 5000
     storage-client:
        connectTimeout: 100
        readTimeout: 200
        retry-count: 3
        retry-backoff-time: 2000
     document-client:
        connectTimeout: 800
        readTimeout: 1500
        retry-count: 3
        retry-backoff-time: 3000
     realty-client:
        connectTimeout: 1000
        readTimeout: 2000
        circuit-breaker-open-timeout: 5000
        circuit-breaker-reset-timeout: 20000
```

Ничего! (Sarcasm)

Ок, а что будет если обновить профиль?

```yaml
# ==============================
# = Feign
# ==============================
feign:
  client:
    config:
     default:
        logger-level: FULL
        connectTimeout: 5000
        readTimeout: 5000
     storage-client:
        connectTimeout: 100
        readTimeout: 200
         retry-count: 3
         retry-backoff-time: 2000
     document-client:
        connectTimeout: 800
        readTimeout: 1500
        retry-count: 3
        retry-backoff-time: 3000
     realty-client:
        connectTimeout: 1000
        readTimeout: 2000
        circuit-breaker-open-timeout: 5000
        circuit-breaker-reset-timeout: 20000
```

Ничего! (Sarcasm)

Что сделать что б увидело? Варианты:

- Рестарт
- Автоапдейт (PropertySource Reload)*

Автоапдейт

Почему не стали включать автоапдейт профилей:

- Опасно из-за инфраструктуры и ее side effect
- Слабо контролируется

Что нам дали внешние конфигурации?

- Возможность быстро править профиль на месте
- Более быстрое устранение инцидентов
- Автоапдейт с риском
- Историю изменения профилей надо где то хранить (репозиторий/сервис профилей)
- Легкость загрузки новых профилей. Можно батчем

Что еще осталось и не уместилось?

- Gateway
- Dataflow
- Spring Cloud Task
- Spring Cloud Contract
- И много чего еще интересного…

Заключение

Чего будет стоить?

- Смены привычного подхода
- Знания того как все работает под капотом
- Умение дебажить и искать ответы самому
- Для внедрения можно начать с одной команды

Кому это может пригодиться?

- У кого монорепы
- Развитая микросервисная инфраструктура
- Есть разные broker
- Общие профили и желание стандартизировать настройки и гибко управлять ими

Общие итоги:

- Сократилось время разбора инцидентов
- Общая база знаний и контрибуция. Комьюнити внутри компании
- Единый подход для множества команд
- Не падает под нагрузкой
- Легко масштабируется

Спасибо!