Инфраструктура

Spring Based Microservices

Дайджест выполнен для личного употребления Krocodl (https://www.linkedin.com/in/victoralekseev/) в 2019-ом году на основании следующей литературы:

• John Carnell "Spring Microservices in Action"

• Josh Long, Kenny Bastani "Cloud Native Java"

• Raj Malhotra "Rapid Java Persistence and Microservices"

Отражает исключительно субъективный взгляд автора. Бесплатен для любого некоммерческого использования.

Все вопросы, замечания и предложения на krocodl@gmail.com

Contents

[Введение 1](#_Toc15569934)

[Современные тенденции 1](#_Toc15569935)

[Монолит 1](#_Toc15569936)

[Традиционный подход 1](#_Toc15569937)

[Проблемы 4](#_Toc15569938)

[Тем не менее 5](#_Toc15569939)

[Микросервисы 5](#_Toc15569940)

[Основные принципы: 5](#_Toc15569941)

[Проблемы микросервисов 6](#_Toc15569942)

[Развертывание 8](#_Toc15569943)

[Основные вопросы 10](#_Toc15569944)

[Обзор понятий и шаблонов 10](#_Toc15569945)

[Core 10](#_Toc15569946)

[Routing (маршрутизация запросов) 10](#_Toc15569947)

[Client resiliency (предотвращение распространения проблем) 11](#_Toc15569948)

[Security 11](#_Toc15569949)

[Logging & Tracing 11](#_Toc15569950)

[Build & Deployment 11](#_Toc15569951)

[Spring Boot & Spring Cloud как инфраструктура 12](#_Toc15569952)

[12 DevOps best practices при разработке и развертывании микросервисов 13](#_Toc15569953)

[Ключевые идеи 13](#_Toc15569954)

[Манифест Adam Wiggins 13](#_Toc15569955)

[Особенности микросервисов применительно к манифесту 15](#_Toc15569956)

[Декомпозиция 17](#_Toc15569957)

[Гранулярность 17](#_Toc15569958)

[Работа с хранилищем данных 18](#_Toc15569959)

[Конфигурирование 18](#_Toc15569960)

[Этапы работы с конфигурацией в Spring 18](#_Toc15569961)

[Типовой подход 19](#_Toc15569962)

[Принципы управления конфигурацией: 19](#_Toc15569963)

[Основные сервисы хранения конфигурационной информации 20](#_Toc15569964)

[Consul 21](#_Toc15569965)

[Spring Cloud Configuration Server 21](#_Toc15569966)

[Хранение конфигурации 21](#_Toc15569967)

[Обращение со стороны клиента 22](#_Toc15569968)

[Безопасность при работе с конфигурацией 23](#_Toc15569969)

[Service Discovery 24](#_Toc15569970)

[Spring Cloud + Netflix Eureka service discovery agent 25](#_Toc15569971)

[Серверная часть 25](#_Toc15569972)

[Клиентская часть - DiscoveryClient 26](#_Toc15569973)

[Consul 28](#_Toc15569974)

[Best practices 28](#_Toc15569975)

[Client resiliency 30](#_Toc15569976)

[Spring Retry 31](#_Toc15569977)

[“Fail Fast” на основе Hystrix 31](#_Toc15569978)

[Минимум 32](#_Toc15569979)

[Настройка 32](#_Toc15569980)

[Кастомизация 34](#_Toc15569981)

[Объединение команд 35](#_Toc15569982)

[Мониторинг 36](#_Toc15569983)

[Service routing на основе Zuul 38](#_Toc15569984)

[Основной функционал 39](#_Toc15569985)

[Архитектура 40](#_Toc15569986)

[Серверная конфигурация 41](#_Toc15569987)

[Роутинг входных запросов 41](#_Toc15569988)

[Фильтры 43](#_Toc15569989)

[Типы фильтров 43](#_Toc15569990)

[Общие моменты реализации 43](#_Toc15569991)

[Распространение по системе correlationId 44](#_Toc15569992)

[Фильтр динамической маршрутизации 45](#_Toc15569993)

[Фильтр ограничениия нагрузки 45](#_Toc15569994)

[Неизбежные проблемы 45](#_Toc15569995)

[Zuul-2 46](#_Toc15569996)

[Безопасность 47](#_Toc15569997)

[Введение (более подробно в приложении) 47](#_Toc15569998)

[Архитектура Spring Security 48](#_Toc15569999)

[Oauth2 + SAML 49](#_Toc15570000)

[Сервер авторизации 49](#_Toc15570001)

[Схема авторизации пользователя 50](#_Toc15570002)

[Защита сервисов (Resource Servers) 51](#_Toc15570003)

[OAuth2 + JWT 52](#_Toc15570004)

[Сервер авторизации 53](#_Toc15570005)

[Сервисы 53](#_Toc15570006)

[Общие рекомендации 54](#_Toc15570007)

[Интеграция и обмен сообщениями 55](#_Toc15570008)

[Spring integration 57](#_Toc15570009)

[Основные абстракции 57](#_Toc15570010)

[Реализация 59](#_Toc15570011)

[Spring Cloud Stream 59](#_Toc15570012)

[Отправка сообщений: 60](#_Toc15570013)

[Получение сообщений 60](#_Toc15570014)

[EDA на основе HTTP 61](#_Toc15570015)

[Spring Cloud Data Flow 62](#_Toc15570016)

[Выполнение заданий (пакетная обработка данных) 64](#_Toc15570017)

[Основные элементы архитектуры 64](#_Toc15570018)

[Локальное выполнение 64](#_Toc15570019)

[Распределенное выполнение 65](#_Toc15570020)

[Управление задачами 67](#_Toc15570021)

[Обеспечение целостности данных 68](#_Toc15570022)

[Проблемы: 69](#_Toc15570023)

[X / Open XA транзакции 70](#_Toc15570024)

[Дополнительные недостатки: 70](#_Toc15570025)

[Шаблоны оптимизации 71](#_Toc15570026)

[SAGA шаблон 72](#_Toc15570027)

[Принципы реализации 74](#_Toc15570028)

[Стандартные фреймворки 75](#_Toc15570029)

[Простейшая реализация 76](#_Toc15570030)

[CQRS шаблон 77](#_Toc15570031)

[Реализация на основе Axon 78](#_Toc15570032)

[TCC ( Try-Confirm / Cancel ) 79](#_Toc15570033)

[REST + ACID на коленке 79](#_Toc15570034)

[Промышленная эксплуатация 79](#_Toc15570035)

[Диагностика – отслеживаемость состояния системы 79](#_Toc15570036)

[Spring Actuator – основной источник диагностической информации 79](#_Toc15570037)

[Системные средства диагностики 79](#_Toc15570038)

[Логирование 79](#_Toc15570039)

[Формирование логов 79](#_Toc15570040)

[Аггрегация логов 79](#_Toc15570041)

[Визуализация логов 79](#_Toc15570042)

[Дополнительные элементы лога 79](#_Toc15570043)

[Администрирование 79](#_Toc15570044)

[Развертывание 79](#_Toc15570045)

[Развертывание в Amazon Cloud 79](#_Toc15570046)

[Конфигурирование Amazon инфрастуктуры 79](#_Toc15570047)

[Запуск системы ручками 79](#_Toc15570048)

[Автоматизация развертывания 79](#_Toc15570049)

[Cloud Foundry 79](#_Toc15570050)

[Типовые проблемы с пермещением приложений в облако 79](#_Toc15570051)

[Приложение 1 – тюнинг серверов 79](#_Toc15570052)

[Приложение 2 – Oauth 2.0 и вокруг 79](#_Toc15570053)

[Основные варианты взаимодействия участников (Types of grants) 79](#_Toc15570054)

[Authorization Code Grant Flow (three legged OAuth) 79](#_Toc15570055)

[Implicit Grant 79](#_Toc15570056)

[Resource Owner Password Credemtials Grant 79](#_Toc15570057)

[Client Credential Grant 79](#_Toc15570058)

[Refresh tokens 79](#_Toc15570059)

[Недостатки 79](#_Toc15570060)

[Веб токены 79](#_Toc15570061)

[SAML токен 79](#_Toc15570062)

[JWT токен 79](#_Toc15570063)

[Приложение 3 – Advanced Messagin Patterns 79](#_Toc15570064)

[Синхронный вызов через асинхронный обмен сообщениями 79](#_Toc15570065)

[Sticky Event “Session” 79](#_Toc15570066)

[Развертывание event driven GUI в console-like стиле 79](#_Toc15570067)

[Обратный адрес (Return Address) 79](#_Toc15570068)

[Цепочка сообщений (Message Sequence) 79](#_Toc15570069)

[Преобразователь порядка (Resequencer) 79](#_Toc15570070)

[Квитанция (Claim Check) – сокращение объема данных в пересылаемом сообщении 79](#_Toc15570071)

# Введение

## Современные тенденции

* Вместо долго процесса разработки, зачастую на основе водопада или с достаточно продолжительными итерациями – много компактных итераций с быстрой поставкой пользователям промежуточного результата, получением обратной связи и постепенным уточнением задачи.
  + Скорость итерации при этом становится намного ценнее текущей эффективности выполнения кода, так как в крайнем случаее можно всегда забросать железом или оптимизировать код.
  + Куда важнее иметь возможность быстрого обновления функциональности с целью получения конкурентных преимуществ в бизнесе
* Рост сложности – современные приложения обычно используют несколько источников данных, взаимодействуют со множеством удаленных сервисов и сами развертываются в нескольких независимых дата центрах
* ПО из обеспечивающего все чаще становится основным конкурентным ресурсом, позволяя бизнесу вводить новые источники дохода, а также оптимизировать использование традиционных
* Быстрая доставка – бизнес требует регулярного внесения необходимых изменений на недельной и даже дневной основе. Множественные заказчики, как правило, не хотят подстраиваться под единый цикл релизов, а оперативно поставляемые ими запросы в принципе не дают возможность его спланировать.
* Производительность и масштабируемость
  + уровень нагрузки в целом может расти очень быстро, вертикальной масштабируемости становится принципиально недостаточно
  + пики / спады нагрузки непредсказуемы - необходима возможность оперативно расширяться / сжиматься, так как ресурсы облаков стоят дорого
  + современные объемы данных, обрабатываемые приложениями, выходят за пределы возможного одного экземпляра БД. Требуется распределять данные по различным базам и хранить в максимально используемом виде
* Жесткие требования к доступности приложения
  + Вследствие непрерывной эволюции системы и непредсказуемой нагрузки проблемы в отдельных частях системы неизбежны и регулярны. Однако одни не должны каскадно распространяться по системе и оказывать влияние на доступность приложения в целом

## Монолит

### Традиционный подход

* Пользовательский интерфейс, бизнес логика и логика работы с данными разрабатываются, собираются и развертываются все вместе в качестве одного артефакта на одном / нескольких выделенных серверах приложений в собственном дата-центре компании.
* Как правило, такое приложение:
  + работает с одной БД и слабо интегрировано с другими системами
  + обновляется достаточно редко – несколько релизов в год, каждый месяц это уже достижение
  + нагрузка достаточно предсказуемая или вообще практически постоянная во времени, что делает вместимость системы достаточно предсказуемой и дает возможность заранее подготовиться к необходимости ее увеличения
  + имеется возможность останавливать приложение для проведения обновления / обслуживания либо планово, либо по необходимости, но в любом случае достаточно безболезненно для пользователей
* Этапы разработки и эксплуатации системы достаточно четко разделены. Очередной релиз приложения пропихивается командой разработки в форточку приемных испытаний, после чего вся ответственность за него как правило передается другой команде
* Будущее такого приложения достаточно прозрачно и предсказуемо, что позволяет закладывать фундаментальные архитектурные решения с долгосрочными последствиями, которые могут быть как положительными, так и отрицательными.

### Проблемы

* Общая проблема - техническая и организационная масштабируемость монолитно организованного приложения
* по мере роста размера и сложности монолита стоимость коммуникации и координации между командами, разрабатывающими отдельные его подсистемы, растут непропорционально быстро
* Дополнительная работа: каждый раз, когда надо внести изменения в одну из частей необходимо собрать, протестировать и развернуть весь монолит
  + Несоответствие ЖЦ: релизные циклы различных частей монолита становятся зависимыми друг от друга, что ведет за собой необходимость дополнительной координации всех участников процесса между собой и снижение темпов работы
  + Вообще, при создании и сопровождении ПО повышение централизации снижает риск сбоев и повышает затраты на согласование вносимых изменений
  + Изменения во внешних приложениях ведут к релизу всего монолита, а не только небольшого шлюза, ответственного за внешнее взаимодействие
  + Типовая жалоба: фукциональность компонента A разработана, но мы не можем выполнить интеграционное тестирование, потому что компоненты В и С не готовы / выдают ошибку / приложение в принципе не поднимается / forkflow не доходит до вызова нашей компоненты из-за каких-то еще ошибок. В результате все проблемы скапливаются в конце цикла релиза и активно припятствуют исправления друг друга
* Единая архитектура монолита, которой должны следовать все компоненты, даже те, для котороых она не оптимальна или излишняя
* Отдельные компоненты монолита тесно связаны между собой на основе конкретного языка / технологии вызова вместо того, чтобы использовать для взаимодействия открытые платформенно нейтральные протоколы
* «По умолчанию» высокая доступность отдельных компонент монолита к ресурсам друг друга и за счет этого высокое зацепление их реализаций. В результате, при небрежном проектировании (а в случае долго разрабатываемых систем практически уже всегда):
  + Внутренние детали реализации каждой компоненты имеют тенденцию «утекать». Особенно через структуру базы данных, в целях «оптимизации» порождающую неявные зависимости между компонентами
  + Все зависит от всего, вроде бы локальные изменения приводят к неочевидным последствиям по всему приложению – глобальное тестирование необходимо даже в случае небольших изменений, локальных для отдельного компонента
  + Непредсказуемость изменений – один баг исправляем и в результате получаем десять новых. Аналогично с введением новых возможностей. В результате с течением времени каждое новое изменение требует все большей и большей трудоемкости
* Runtime проблемы
  + масштабируется только целиком, хотя реально дополнительные ресурсы могут быть нужны только одной из компонен и только время от времени. Лишний код, масштабируемый попутно, только жрет ресурсы, хотя бы за счет увеличение расходов на переключение контектов потоков
  + Runtime проблемы в каждой из компонент мгновенно становятся проблемами приложения в целом – неустойчивость системы в целом
* В современных условиях требования к приложению могут меняться достаточно быстро, поэтому «оптимальность» архитектуры и дизайна можно оценить только в достаточно короткосрочной перспективе. Поэтому «фундаментальность» заложенной единой архитектуры может выйти боком

### Тем не менее

* Основное достоинство: все уже давно более или менее научились его проектировать, разрабатывать, отлаживать, мониторить и так далее
* Простота работы с локальными вызовами.
* Возможность использовать ACID транзакции
* Нет проблем с коггеретностью версий компонент: все несоответствия API выявляются в момент сборки, а не в runtime
* Вопреки распространенному заблуждению монолит вполне может быть масштабируемым и отказоустойчивым.
  + Проблема в том, что степень этого ограничена. Например, если мы множим экземпляры монолита на одной базе данных, то имеем ограничение пропускной способности базы, а также проблемы с оптимистическими транзакциями, когда все участники «топчутся на месте»
  + Кроме того, масштабирование всего вместе в значительной степени съедает ресурсы за счет работы тех компонент монолита, которые в масштабировании не нуждаются. Прежде всего потому, что «попутные» компоненты приносят свои потоки, на переключение которых тратятся ресурсы и память
* Примеры действительно больших и успешных монолитов:
  + Shopify – SaaS платформа для интернет магазинов, на середину 2018 года на ней работало порядка 600 тыс магазинов принося 17$ миллионов в месяц
  + Basecamp

## Микросервисы

* распределенные, слабо связанные, легко управляемые сервисы, каждый из которых выполняет небольшое количество четко и по возможности достаточно узко определенных задач
* Не существует индустриального стандарта на архитектуру / реализацию микросервисов. Это скорее набор принципов организации, нежели отдельные технологический стек вроде веб-сервисов.

### Основные принципы:

* Каждая команда разработчиков при этом полностью и единолично владеет исходным кодом и инфраструктурой.
  + Они имеют независимый репозиторий, отдельные установки необходимых серверов (app / database / ???), собирают / развертывают / тестируют сервис независимо друг от друга.
  + Разработчики как правило ответственны не только за собственно разработку, но и за всю последовательность операций включая финальное развертывание, а также за эксплуатацию системы. Принцип Амазона: вы создаете этот продукт, вам и обслуживать его работу
* Каждый микросервис может быть собран / развернут / протестирован независимо от других сервисов приложения. Так как размер микросервиса мал, то данные операции достаточно дешевы и происходят быстро
* Каждый микросервис имеет собственный четко определенный бизнес домен и область ответственности, за которые он исключительно и полностью отвечает.
  + В некотором смысле это следование UNIX модели – приложение есть не больше чем коллекция сервисов, каждый из которых делает только одну вещь, но зато делает хорошо
  + Как сами микросервисы, так и команды их разработчиков, организуются вокруг конкретных бизнес-процессов, за успешность которых они несут полную ответственность. Команда становится группой, нацеленой на конкретный результат с измеряемой бизнес-значимостью
* Каждый микросервис владеет собственным хранилищем данных, доступ на модификацию к которому есть только у него.
  + Даже если несколько микросервисов оперируют одними и теми же данными, то каждый хранит собственную реплику в оптимальной для себя форме.
  + Вообще, любые совместно используемые ресурсы, не только базы данных, но и, например, сервера приложений, затрудняют расширение функциональности, замедляют процесс от разработки до развертывания и осложняют контроль за работой при эксплатации
  + Когда Amazon в процессе маштабирования разработки столкнулся с необходимостью совместного доступа нескольких команд к одной БД, то был выработан принцип, что интеграция данных должна выполняться в терминах API, а не на основе общих структур данных
* Взаимодействие между микросервисами описывается посредством стандартного для системы API, которое является предметом соглашения между отдельными командами. Пока API (и набор входящих в него пре / пост условий) неизменно, разработчики сервиса вправе производить в нем любые изменения.
* Общение микросервисов между собой осуществляется на основании легковесных протоколов, не привязанных к определенной технологической платформе. Как правило это HTTP + JSON / XML, но могут применяться системы обмена сообщениями, бинарные форматы обмена и так далее. Основной принцип – тонкая транспортная среда + умный клиент. ESB, к примеру, как способ взаимодействия не рекомендуется
* Микросервисы должны быть повторно используемыми со стороны различных приложений. Если этого не происходит, то, скорее всего, его бизнес домен определен не верно и он должен быть частью другого микросервиса
* Эффект траектории:
  + если однажды в больом монолитном приложении было принято плохое архитектурное или дизайнерское решение, то скорее всего оно уже расползлось по всей системе, его выкорчевывание требует значительных затрат (а также переходного периода, когда стабильность решения будет под угрозой), а в свою очередь оно продолжает вызывать последовательность неудачных решений и компромиссов, разворачивающуюся в будущее
  + Микросервисы это в частности еще и способ пересмотреть ранее сделанные неудачные решения и избавиться от их негативного влияния сегодня и, что более ценно, завтра. Даже если мы где-то опять примем плохое решение, то scope его негативных последствий будет гораздо уже и исправить его будет гораздо дешевле

### Проблемы микросервисов

* Любое удаленное взаимодействие
  + чисто программно сложнее организовано, особенно когда мы хотим параллелить вызовы, выполнять асинхронное продолжение обработки результатов и так далее
  + на порядок медленнее, чем локальный вызов
  + все может пойти в любой момент не так: запросы / ответы могут потеряться и даже прийти в неверном порядке, сервис может вообще временно перестать отвечать или быть доступным. Отдельная головная боль, когда ответ вообще не получен и не понятно, операция была выполнена или нет (потерялся запрос или ответ).
* Гораздо труднее отлаживать, так как каждый вызов проходит через несколько экземпляров серверов. В результате нету полного контекста, а также невозможно продолжить отладку запроса при удаленом обращении. Аналогично с логированием и мониторингом.
* Любое тестирование так или иначе выливается в интеграционное тестирование со всеми его недостатками. Или же все по отдельности работает нормально, но вместе не взлетает к примеру из-за ошибок целостноти данных аргументов вызовов.
* Невозможность использования привычных ACID транзакций (распределенные не выход, они неустойчивы и убивают производительность). Данные приложения постоянно находятся в слегка несогласованном состоянии и процессы надо проектировать с расчетов на это
* Проблемы выделения common бизнес кода – он не стабилен и как только подвергается изменению, то волна потенциальных изменений прокатывается по всем микросервисам, даже не задействованных изменяемой функциональностью. Как результат – необходимость дублирования кода.
* многократно возрастают операционные расходы
  + как минимум по отдельному репозиторию на сервис с цепочкой CI задач для сборки / тестирования / приемного тестирования /развертывания.
  + Реализация каждого фрагмента функциональности как правило затрагивает несколько репозиториев – трудно для анализа
  + Число артефактов стремительно возрастает, как минимум <число сервисов> х 3: API сервиса, реализация сервиса, Docker контейнер с сервисом и так далее. Также для каждого сервиса надо хранить и поставлять в качестве артефакта конфигурацию для каждой из сред исполнения.
  + версионность: у каждого сервиса независимый ЖЦ, поэтому у всех разные версии, на которые надо вовремя переключаться.
  + специальная инфраструктура для развертывания в различных окружениях и мониторинга состояния
  + Необходимость специальных систем сбора и совместного анализа логов
  + В общем нужны выделенные DevOps специалисты для поддержки всего этого зоопарка, без них шансов на успех довольно мало
* Поэтому зачастую применяется антипатерн «распределенный монолит»
  + Единая команда, единый репозиторий из которого каждый раз собирается весь набор микросервисов одной версии
  + Много common бизнес кода, который разделяется всеми сервисами.
  + Единая база данных. Даже если таблицы не шарятся между сервисами, то они все равно используют внешние ключи друг на друга. В результате, даже если применяем распределенные транзакции, то имеем проблемы, так как на уровне базы разные ее части не видят изменений друг друга
  + Общие релизы (собственно это релиз монолита как такового)
  + Единая процедура одновременной инсталляции всех микросервисов при развертывании приложения. Зачастую все микросервисы запускаются на одном сервере или сервере приложений.
  + В общем от всех достоинств микросервисов имеем только возможность поднять дополнительные экземпляры отдельных сервисов (но для этого как правило нет соответствующей инфраструктуры). Но зато имеем практически все недостатки обоих подходов
* Микросервисы это не то, с чего надо начинать. Это довольно болезненный способ решения проблем до которых еще надо дорасти.
  + И большинство проектов до этих проблем реально никогда не дорастает
  + Надо понимать, что при использовании микросервисов обмениваем значительные дополнительные усилия и ресурсы на масштабирование – разработки и исполнения. Если масштабирование реально не нужно, усилия и ресурсы будут выброшены зря в угоду моде
  + Итого – стартуем с монолита и держим в уме, что в случае нашего очень большого успеха когда-нибудь нам придется его разделять. А потому не допускаем совсем уж откровенной грязи в процесе дизайна и разработки.
* Противопоказания
  + Не имеет смысла впрягаться в микросервисы, если компания не согласна солидно инвестировать в автоматизацию и выделенных DevOp специалистов
  + Средние приложения уровня департамента организации с малым числом пользователей и предсказуемой нагрузкой. Но зачастую даже крупные приложения с миллионом пользователей (Shopify, Basecamp) – смотря как писать.
  + Приложение, занятое в основном сложной агрегацией / преобразованием данных, получаемых из множественных источников
  + Жесткие требования к целостности и доступности данных

### Развертывание

* Специфика развертывания обуславливается тем, что мы можем иметь одновременно сотни экземпляров, динамически поднимаемых и опускаемых в произвольные моменты времени
* Требования к микросервисам
  + Должны быть представлены в виде единого артефакта, содержать внутри себя все необходимое для запуска и быть независимо друг от друга развертываемыми. Это подразумевает, что JEE контейнер должен содержаться внутри запускаемого артефакта микросервиса.
  + Должны быть конфигурируемыми и в процессе старта получать всю необходимую для работы информацию из центрального источника или переменных окружения. Ручное конфигурирование / переконфигурирование в процессе эксплуатации исключается
  + По сравнению с традиционным приложением экземпляр микросервиса живет гораздо короче (часто перезапускается), одновременно запущено гораздо большее количество экземпляров и в любой момент времени какие-то из них опускаются / поднимаются, получая случайные IP
  + В процессе старта должен регистрироваться в системном service discovery, при завершении, соответственно, удаляться оттуда
  + Их расположение должно быть прозрачно для клиента. Клиент никогда не должен знать точного физического адреса сервиса, к которому он обращается. Вместо этого он должен каждый раз получать его из service discovery
  + Микросервис должен отвечать на запросы service discovery о своей доступности / здоровье, чтобы предотвратить обращение клиентов к «больным» или вовсе недоступным экземплярам
  + Микросервис должен предоставлять стандартной системной инфраструктуре данные для мониторинга и собственный лог в стандартной форме
* Типовые способы развертывания микросервисов:
  + **Физические сервера** – наиболее привычный, но вертикальную вместимость невозможно наращивать бесконечно, а горизонтальная в этом случае значительно затруднена. Система не эластична (если конечно самому не заниматься запуском / остановкой по требованию). Трудно восстанавливать систему в случае падения самих серверов
  + **Виртуальные машины** – уже лучше, их легко запускать / останавливать / мигрировать в облаке, предоставляемым провайдером IaaS, одна виртуалка может нести целый пакет сервисов, абсолютный контроль над операцинным окружением сервиса. Однако, они большие, их дорого собирать и доставлять в облако, приходится заниматься задачей их конфигурирования. Наиболее гибкое, но тяжелое решение
  + **Виртуальные контейнеры** – наиболее гибко (и, кстати, вполне совместимо с предыдущим вариантом), посредством контейнеров мы нарезаем виртуальные машины на отдельные области, выделяемые сервисам. Контейнеры достаточно маленькие, быстро собираются прямо в ходе CI процесса, быстро доставляются, оперативно поднимаются / опускаются вручную или за счет предоставляемого IaaS провайдером API независимо от виртуальных машин
  + **Использование облаков –** само приложение релизует минимум функциональности, напрямую не связанной с бизнесом, оставляя остальное на ответственность поставщика инфрастуктуры.
    - Позволяет при правильном подходе обеспечить масшабируемость, производительность и доступность, недостижимую для локального inhouse развертывания, особенно, в ситуации, когда бизнес становится глобальным
    - Нет необходимости предсказывать на месяцы вперед, какие будут потребности в оборудовании, хранилищах и сетевой аппаратуре
    - Вопреки распространенному мнению не обязательно приводит к удорожанию эксплуатации, особенно если подсчитать все вложения вместе, и начальные и регулярные. Перевод бизнеса Netflix на использование AWS позволило снизить затраты на 87%
* Типы облаков – по мере перехода все большую доля функциональности приложения является ответственностью поставщика сервиса. Также в данном направлении усиливается наша зависимость от конкретного вендора и переносимость системы мужду различными провайдерами.
  + **Infrastructure as a Service (IaaS)** – предоставляется основная сетевая инфраструктура и виртуальные сервера, мы вольны выбирать любую технологию реализации приложения и конфигурировать виртуальные машины как нам нравится
  + **Platform as a Service (PaaS)** – провайдер берет на себя эксплуатацию виртуальных серверов и серверов приложений, а мы предоставляем код для развертывания в стандартных видах (jar / war архивы, например).
    - Платформы позволяют автоматизировать решение задач общего характера, не играющих существенную роль в реализации конкретных бизнес-требований, предъявляемых к приложению
    - Командам предоставляется интерфейс самообслуживания (обычно веб + REST), позволяющий сформировать инфраструктуру, необходимую для запуска приложения, за счет пула стандартных внешних зависимостей – база данных, служба кэширования данных, маршрутизатор запросов с поддержкой политик безопасности и так далее
    - Приложения помещаются в стандартный пакет, снабженный описанием необходимых ему сервисов – внешних зависимостей. В процессе запуска, платформа информирует приложение о конкретном расположении внешних зависимостей, как правило, посредством переменных окружений
    - Платформа берет на себя основные операции, связанные с обеспечением жизнеспособности приложения в целом. Например, перезапуск сбойных экземпляров или масштабирование числа запущенных экземпляров в зависимости от нагрузки
    - Примеры – Amazon Cloud Foundary или Heroku.
    - Тоже гибко и достаточно переносимо, но как правило дорого.
  + **Container as a Service (CaaS)** – разновидность PAAS, провайдер принимает некоторые контейнеры (обычно Docker) в которые мы упаковываем свое приложение. Предоставляет инфраструктуру управления контейнерами, берет на себя обслуживание виртуальных серверов
  + **Function as a Service (Faas)** – разновидность PAAS,примеры Amazon Lambda или Google Cloud. Мы имеем возможность развертывать небольшие serverless фрагменты кода и платить только за ресурсы, которые тратятся именно на их выполнение. Обслуживаение инфраструктуры серверов приложенией и коружения кода берет на себя провайдер. Быстрый старт, но тяжело разрабатывать большие решения и уже практически невозможно куда-нибудь мигрировать без коренной переработки системы.
  + **Software as a Service (SaaS)** – провайдер предоставляет прикладное приложение целиком

### Основные вопросы

* **Right size** - как разбить функциональность всего приложения на отдельные микросервисы оптимального размера так, чтобы с одной стороны иметь возможность быстро вносить в них изменения, а с другой – снизить риск возможной деградации приложения в целом
* **Location transparent -** как абстрагироваться от параметров размещения / доступа к конкретным экземплярам микросервисов с учетом динамического числа их экземпляров
* **Resilient -** как защитить клиентов микросервисов то того, что в любой момент они могут быть опущены / физически недоступны / безнадежно сломаны
* **Repeatable –** каким образом обеспечить идентичность всех экземпляров конфигурации одного типа микросервисов с учетом того, что одни могут жить уже достаточно долго, а другие поднимаются только что
* **Scalable** – каким образом (обычно на основе асинхронной обработки событий) минимизировать непосредственные зависимости между сервисами и обеспечить максимальную масштабируемость приложения

## Обзор понятий и шаблонов

### Core

* **Service granularity** - как обеспечить оптимальную декомпозицию приложения на отдельные сервисы и выбрать оптимальный уровень ответственности для каждого.
  + Слишком крупные сервисы тяжело поддерживать и изменять.
  + Слишком мелкие увеличивают сложность приложения , требуют при разработке каждой бизнес функциональности внесения изменения в большое число сервисов. Кроме того, значительно возрастают затраты на удаленное взаимодействие и количество потенциальных точек отказа.
* **Communication protocols** – обычно это HTTP + XML / JSON, но бывают, к примеру, Thrift или Grps. Принципиальный выбор идет между REST или RPC типом API
* **Interface design** – как структурировать URL точек доступа и поддерживать версионность, идемпотентности, степень гранулированности передаваемых данных, способ обработки ошибок и так далее
* **Configuration management**
  + каким образом поставлять / обновлять конфигурацию экземплярам микросервисов, чтобы обеспечить их работу в различных окружениях: для разработки / тестовом / боевом на различных площадках и в составе различных кластеров
  + Управление конфигурационными данными как централизованный сервис
    - данные конфигурации полностью независимы от собственно реализации микросервисов.
    - всем экземплярам одного микросервиса автоматически доставляется (pull / push принцип) одна и та же конфигурация
* **Event processing** – каким образом минимизировать непосредственные зависимости между сервисами и обеспечить максимальную масштабируемость приложения

### Routing (маршрутизация запросов)

* **Service discovery** – каким образом клиент по логическому наименованию получает доступ к одному из экземпляров требуемого типа микросервисов с учетом того, что экземпляры могут в любой момент быть подняты / опущены
  + Абстрагирует логическую ссылку на сервис от его физического нахождения.
  + Обычно каждый стартующий экземпляр сервиса самостоятельно регистрируется в базе и становится доступным для запрашивающих его по логическому имени клиентов
* **Service routing** – предоставление входной точки доступа к всем экземплярам одного типа микросервисов, обеспечивающей авторизацию, аутентификацию, контроль данных и прочие пограничные кросс – сервисы

### Client resiliency (предотвращение распространения проблем)

* **Client side balancing** – каким образом со стороны клиента разбрасывать запросы между различными экземплярами микросервисов с учетом их «здоровья»
* **Circuit breakers (fail fast)** – каким образом временно выводить «больные» экземпляры микросервисов из обращения в надежде на то, что они «выздоровеют». А также как не тратить время в попытках выполнить запрос при помощи заведомо больного сервиса.
* **Fallback** – каким образом предоставить альтернативу «больному» сервису, позволяющую хоть как-то поддержать работу приложения
* **Bulkhead** – каким образом делать так, чтобы тормозящие запросы к «больным» сервисам не влияли на работу клиента в целом

### Security

* **Authentication & Authorization**
* **Credential management & Propagation** – каким образом сделать так, чтобы клиенту не приходилось предоставляться логин / пароль при каждом обращении, а также защищенным способом передавать данную информацию дальше по цепочке вызовов

### Logging & Tracing

* Основной недостаток микросервисов заключается в том, что debug / trace запросов значительно усложняются
* **Log correlation** – как собрать воедино все фрагменты логов, относящиеся к одному бизнес-запросу к приложению
* **Log aggregation** – как на регулярной основе собирать все логи со всех экземпляров микросервисов в единое хранилище для дальнейшего анализа
* **Microservice tracing** – как анализировать собранные логи в целях выявления проблем с производительностью, контроля SLA показателей и так далее. Каким образом сопоставлять между собой бизнес и технические термины при анализе данных.

### Build & Deployment

* Все активные экземпляры одного микросервиса должны быть идентичны, в противном случае мы имеем “configuration drift”. Основная проблема при ручном управлении: делаем какое-то изменение на сервере тестирования / разработки, но забываем перенести его на основной сервер.
* **Build and deployment pipeline** – идеал это полная автоматизация от коммита до развертывания в боевом окружении
* **Infrastructure as code** – все, относящееся к развертыванию микросервисов, является исполняемыми артефактами, находящимися под контролем системы версий
* **Immutable servers** - так как экземпляры микросервисов достаточно компактны и обычно быстро стартуют / опускаются, то применяется концепция «immutable infrastructure» - после того, как сервис автоматически развернут, никакие манипуляции с ним (ручные или посредством API не важно) не разрешаются.
* **Phoenix server** – чем дольше сервер живет, тем больше вероятность “configuration drift”. Выход – регулярно обновлять
* Наряду с традиционными ролями архитектора и разработчика появляется новая – DevOps, занимающийся автоматизацией операций сборки, развертывания и управления сервисами во всех типах окружения

## Spring Boot & Spring Cloud как инфраструктура

* предоставляют средства миграции традиционных монолитных Spring based приложений в микросервисную архитектуру, пригодную для развертывания в облаке.
* Организованы за счет обертывания нескольких хорошо себя зарекомендовавших облачных библиотек, происходящих из инициативы Netflix. Они тоже в накладе не остались: замечательным качеством проекта с открытым исходным кодом, решающим задачу общего характера, является то, что он дает импульс собственному развитию и длительное время поддерживается циклом непрерывного самосовершенствования
* Важно: предоставляет не только собственно программные средства для реализации микросервисов, но и целостную инфраструктуру / набор шаблонов проектирования для эксплуатации всего приложения на их основе: сбор логов, аудит, мониторинг, администрирование и так далее.
  + Недостаток: не предоставляет никаких сервисов для развертывания
  + Достоинство: вовсе не обязательно пользоваться сразу всем богатством фич, особенно если у нас система из 5-6 сервисов. Можно добавлять по мере необходимости

|  |  |
| --- | --- |
| **Управление конфигурацией** | * **Spring Cloud Config** на основе Git репозитория, а также базы данных конфигурационных значений, предоставляемой Consul / Eureka серверами * Consul * Vault |
| **Регистрация и динамическое обнаружение сервисов** | * **Spring Cloud Service discovery** на основе серверов **Consul / Eureka** |
| **Динамический балансир и роутинг запросов** | * Обертка вокруг **Netflix Hystrix** обеспечивает circuit **breaker и bulkhead** защиту за счет выделения отдельного пула потоков для выполнения запросов к каждому из серверов * **Netflix Ribbon** облегчает интеграцию с service discovery, кэширование данных при временной недоступности discovery сервиса и клиентский балансир запросов, который позволяет отказаться от отдельных выделенных балансировщиков (являются дополнительными точками отказа и требуют специального администрирования) * **Hystrix Dashbord –** средство визуализации состояния клиентов и собираемой статистики * **Turbine** – способ аггрегации данных от различных экземпляров сервисов для передачи в средство визуализации * Интеграция с **Netflix Zuul** позволяет реализовать frontend, прозрачно проксирующий все запросы к микросервисам, расположенным за ним, и применяющий необходимые политики доступа /правила маршрутизации |
| **Защита удаленных вызовов** | * Netflix Hystrix * Resilience4j * Sentifel |
| **Интеграция** | * **Spring Cloud Stream**   + Легковесная интеграция микросервисов между собой на основе обмена сообщениями   + Может легко адаптировать такие широко распространенные брокеры как RabbitMQ или Kafka * Spring Integration * Spring Data Flow |
| **API Gateway** | * Прежде всего **Netflix Zuul** * Nginx * Облачные провайдеры |
| **Распределенная трассировка** | * **Spring Cloud Sleuth**   + Позволяет автоматически подмешивать идентификаторы корреляции запросов в HTTP трафик и сообщения, а также агрегировать данные логов с различных экземпляров микросервисов   + Собираемые результаты можно агрегировать / визуализировать при помощи Papertrail / Zipkin * Zipkin – средство визуализации и анализа логов. Аналоги - ELK стек, Dapper |
| **Мониторинг** | * Spring Actuator + Dropwizard * Spring Boot Admin * Datalog * New Relic * Традиционные Java Melody и Glowroot |
| **Логирование** | * Spring Sleuth * ELK * Splunk |
| **Безопасность** | * **Spring Cloud Security**   + Идентификация / аутентификация клиентских запросов на основе токенов, выдаваемых централизованным сервером   + Интеграция с OAuth2 и SAML / JWT * Apache Shiro * Dropwizard |
| **Документирование** | * Swagger * RAML * GraphQL |

# 12 DevOps best practices при разработке и развертывании микросервисов

## Ключевые идеи

* использование декларативных форматов для автоматизации установок и настроек, свидение за счет этого к минимуму времени и усилий, ускорение введения в проект новых членов
* формализация соглашений с окружением, обеспечивающее возможность переносимости между средами выполнения
* обеспечение возможности развертывания на современных облачных платформах, избавляющих от необходимости использовать сервера и вручную производить системное администрирование
* формализация и автоматизация процесса развертывания, позволяющие перейти к модели непрерывного развертывания, сводящей к минимум число необхходимых ручных операций
* Масштабирование приложений должно осуществляться без необходимости внесения изменений в архитектуру

## Манифест Adam Wiggins

* **Codebase** – все исходники (сервисов, конфигурации сервисов, конфигурации развертываемых серверов и так далее) должны быть под управлением системы контроля версий.
  + У каждого микросервиса должен быть собственный репозиторий с исходным кодом, содержащим конфигурируемые ссылки на все внешние зависимости приложения
  + Развертывания приложения во всех средах осуществяется из единой кодовой базы.
  + Перемещение приложения между средами не должно требовать перекомпиляции или пересборки
* **Dependencies** – всегда экстернализировать и явно конфигурировать версии всех зависимостей посредством maven / gradle
  + Зависимости должны быть только от релизных версий, а не от SNAPSHOT.
  + Средства сборки и любые используемые в процессе сборки / развертывания / конфигурирования артефакты тоже должны иметь версии.
  + Работающая система должна предоставлять возможность получения перечня версий всех компонент, которые были использованы при ее развертывании.
  + Все вместе это обеспечивает воспроизводимость состояния системы, что критично про воспроизведении / исправлении проблем. В особенности, когда система эксплуатируется клиентом самостоятельно и доступа к ней просто нет.
* **Config** – вся конфигурация и особенно конфигурация, изменяющая в зависимости от среды исполнения, должна хранится / версионироваться независимо от исполняемого кода приложения
  + Среда должна поставлять конфиграцию приложению, а не наоборот. Любая разница между средами должна рассматриваться в качестве элемента конфигурации и сохраняться в среде
  + Обычно под конфигурацию выделяется отдельный репозиторий
  + Такие элементы конфигурации, как строки подключения, учетные данные, имена хостов внешних ресурсов должны поставляться приложению через переменные окружения
  + Хороший критерий достижения: код прилоения в любой момент может быть раскрыть для публичного доступа без компрометации чувствительных данных
* **Backing services** – основные сервисы работы с данными (БД, система обмена сообщениями) должны поставляться приложению в виде конфигурируемых ресурсов и использоваться таким образом, чтобы допускать произвольное перемещение приложения между различными средами / площадками
* **release, build, test, run** – данные операции должны выполняться автоматически, отдельно друг от друга с передачей результатов друг другу через хранилище артефактов. Любые изменения, требуемые в конце цепочки, выполняются на начальных этапах и дальше проносятся только автоматическими штатными процедурами
* **processes** – микросервисы должны быть stateless и допускать возможность убийства / замены в любой момент без потери обрабатываемых данных. Тест на достижимость данного требования: возможность полного демонтажа и последующего возсоздания среды выполнения без потери данных приложения.
* **port binding –** микросервисы должны быть представлены полностью автономными исполняемыми артефактами
  + то есть содержать в себе все необходимое и не требовать развертывания на отдельно установленном (и кем-то конфигурируемом) сервере приложений.
  + Они стартуют из командной строки и немедленно должны быть доступными по HTTP или другим протоколам доступа
  + Должны допускать запуск на произвольном порту, поставляемом конфигурации среды, а также в свою очередь использовать зависимости, разверные на произвольных портах
* **Concurrency –** масштабируемость достигается за счет запуска множественных экземпляров одного микросервиса (горизонтальная), а не за счет возможности использования большего количества потоков внутри внутри него (вертикальная).
  + Соответственно процедура обработки данных должна быть совместима с множественными потребителями запросов, совместно использующими на конкурентной основе при их обработке единое хралище данных
  + Хотя, конечно, параметры используемых пулов потоков должны быть конфигурируемы
* **Disposability –** должны запускаться / останавливаться по требованию ( обычно командная строка / kill сигнал).
  + Время остановки должно быть минимальным, а обрабатываемые в этот момент данные должны быть корректно завершены. Долгое время остановки не позволяет оперативно перегружать сервера по мере идентификации проблем с их здоровьем. Увеличивает риск повреждения данных из-за частично обработанных или потерянных запросов
  + Быстрый запуск сокращает время, необходимое на осуществление масштабирования при резком повышении нагрузки. Если приложение запускается долго, то за это время всплеск трафика способен перегрузить все имеющиеся экземпляры, а затем, когда они умрут, убить и свеже поднятый
* **Dev / prod parity –** среды, в которых сервисы разрабатываются / тестируются / выполняются должны быть максимально инфраструктурно похожи друг на друга.
  + Сервисы по возможности должны максимально быстро передаваться из среды в среду без необходимости выполнения каких-либо ручных операций
    - Если передача микросервиса из среды в среду становится затруднителным, то команда начинает опасаться данного этапа и его откладывать. При этом увеличивается объем очередных вносимых изменений, а значит риск неудачи. Негативный процесс приобретает положительную обратную связь
    - Основной принцип: введение изменений мелкими порциями, частое и быстрое переразвертывание, малозатратный откат в случае проблем
  + Типы расхождений между средами
    - Расхождения во времени: разработчики должны рассчитывать на то, что вносимые ими изменения будут быстро введены в реальную эксплуатацию
    - Расхождения в персонале: разработка, внедрение и техническая эксплуатация приложения должна осуществляться одними и теми же людьми
    - Расхождение в инструментарии: в каждой среде должны применяться один и тот же инструментарий и режимы работы, чтобы исключить неожиданные эффектв из-за неучтенных несоответствий
* **Logs** – логи должны содержать потоки событий, которые в конечном итоге подхватываются средствами типа **Splunk** или **FLuentd**, которые их отправляют в центральное хроанилище.
  + Микросервисы не должны сами определять механизм доставки логов, полагаясь на подобные механизмы. Обычно микросервисы пишут лог в STDOUT и этим ограничиваются.
  + Параллельная запись в файлы это конечно хорошо, но чревато переполнением выделенного контейнеру дискового пространства (особенно, в случае непредвиденных проблем, когда объем логов возрастает многократно из-за стеков ошибок).
* **Admin processes** – операции обслуживания типа миграции / конвертации данных не должны выполняться вручную. Вместо этого используются скрипты, которые кладутся под контроль версий и тестируются последовательно на всех средах. Они должны быть идемпотентными (допускать повторное применение), применение их в различных окружениях не должно влечь за собой необходимость их ручного редактирования (конфигурируемость)

## Особенности микросервисов применительно к манифесту

* Развертывание
  + Микросервисы содержат JEE контейнер (обычно легковесный, типа Tomcat / Jetty) внутри себя как один из программных компонент
  + Это существенное отличие от монолита, который обычно развертывается на отдельно установленном и администрируемым сервере приложений, на котором вдобавок еще могут крутиться и другие системы. Администраторы такого сервера (серверов в нескольких средах) все время чего-то забывают подкрутить / синхронизировать, операции выполняются вручную, данные не хранятся под в CVS, в результате чего реальная конфигурация сервиса «плывет».
  + соответственно такой контейнер должен конфигурироваться самим сервисом при запуске. Ручное до-конфигурирование контейнер в процессе эксплуатации НЕ допускается, вместо этого сервис перезапускается с другой конфигурацией.
* Конфигурирование
  + Поставляемая конфигурационная информация также должна находится под контролем CVS и распространяться при помощи инфраструктурного сервиса. Такой сервис может потенциально быть единой точкой отказа. Также надо принимать во внимание, что микросервисы могут быть распределены по отдельным площадкам, каждая из которых потребует отдельного экземпляра такого сервиса для обеспечения надежности и скорости доступа
  + Если конфигурационная информация меняется, то все сервисы, которых она касается, либо автоматически перезапускаются (что проще и лучше, если конечно речь не идет о тысячах экземпляров по всему миру), либо получают сообщение о необходимость получить и применить свежую версию (реализация сложнее и менее надежна, но более гибкая).
  + Конфигурационная информация обычно представлена в виде набора пар «ключ / значение» редко меняется и часто читается, причем при чтении требуется малая задержка. Использование в качестве ее хранилища реляционной БД не слишком оправдано
* Взаимодействие между сервисами
  + При старте микросервис получает от инфраструктуры случайный IP и бежит регистрироваться в discovery service, поставляемый окружением.
  + При регистрации сообщает URL интерфейса, а также специализированные URL-ы, по которому discovery service может мониторить его здоровье, запросить обновление конфигурации или даже shutdown, после чего уберет этот экземпляр из перечня доступных клиенту
  + При обращении к микросервису клиент запрашивает у discovery service перечень доступных «здоровых» экземпляров, а затем при помощи собственного round robin балансира выполняет запрос
  + Необходимо учитывать, что микросервис может помереть как раз непосредственно перед выполнением запроса, так что его недоступность все равно может случиться в любое время
  + Данная возможность реализуется автоматически за счет добавления зависимости от **Spring Actuator**. У сервиса появляется **/health** интерфейс, который публикует статус активности, размер места на диске и так далее. Данный перечень можно расширить собственными провайдерами значений
* Не рекомендуется разрабатывать бизнес-компоненты, которые совместно используются многими микросервисами как библиотеки.
  + Такие библиотеки мало стабильны и вызывают проблемы со стабильностью остальных частей приложений. Все зависимости должны быть в сторону более стабильных компонент, в данном случае это правило нарушается.
  + Релизные циклы микросервисов начинают влиять друг на друга. Переключаясь на новую версию такого компонента сервис получает кроме необходимых также совершенно посторонние для себя изменения. Реализация бизнес требования, изменяя такую библиотеку, может вызвать волну изменений, затрагивающую сервисы, которые в теории никак ею на затронуты.
  + Можно выделять библиотеки для работы с **БПС / инфраструктурой**, то есть которые стабильны и меняются достаточно редко.
  + С оговоркой можно вынести в подобную библиотеку “Common Data Model”, но у такой глобальной зависимости есть другие противопоказания.
    - Например, для каждого в отдельности интерфейса, она может быть избыточной и требовать обязательной передачи данных, которые в данном конкретном случае не нужны
    - Наболее приемлимым считается следующий подход: каждый микросервис ответственен за некоторый фрагмент модели данных - **ProductModel**, которая эволюционируется вместе с его функциональность. Его клиенты индивидуально определяют у себя минимально необходимые подмножества это модели – **ProductView.** Даже если основная модель изменится, ее представления смогут сохранить актуальность.
  + Серьезные проблемы начинаются, когда в такие библиотеки начинают выносить бизнес-код, например, проверки данных или маппинг. Как правило, необходимость таких библиотек указывает на то, что несколько микросервисов разделяют один бизнес домен и должны быть объединены

# Декомпозиция

* Задача архитектора – разбить весь бизнес – процесс на отдельные фрагменты, каждый из которых представлен отдельным бизнес доменом, который инкапсулирует внутри себя бизнес-правила и логику работы с данными
* Если в приложении удается выделить несколько бизнес доменов, то это хороший показатель к микросервисной архитектуре
* Интересный способ выделения бизнес доменов – через кластеризацию таблиц в базе данных.
  + Если таблицы удается разбить на несколько тесно связанных внутри и слабо связанных между кластеров, то каждый из них скорее всего представляет собой бизнес домен и кандидат на хранилище данных отдельного сервиса
  + В общем, это не значит, что каждый микросервис должен иметь свой отдельный экземпляр базы данных. Но отдельную схему под него сделать надо, чтобы не было соблазна «оптимизировать» работу и в результате получить неявные зависимости между сервисами
  + Проблемы с точки зрения DBA: невозможность внешних ключей, пересекающих домены, а также единых транкций

## Гранулярность

* Основной вопрос декомпозиции
* Лучше начать с крупных сервисов и разбивать их по мере необходимости. Чем выше гранулярность, тем больше проблем с взаимодействием мы имеем.
  + Как минимум
    - тем больше производительности мы теряем за счет множественных удаленных вызовов.
    - Тем более ненадежным является приложение в целом, так как каждый удаленный вызов это потенциальная ошибка или даже несколько с риском оставить данные в не консистентном состоянии. Тем более специальных усилий мы должны прилагать, чтобы добиться высокой надежности
  + Однако маленький размер это
    - Скорость процесса - быстрая сборка, тестирование и развертывание, что облегчает как исправление багов, так и выкатывание новой функциональности.
    - Ограниченность scope – легче проходят операции анализа и модификации. Экстемистский подход – оптимальный размер, это когда проще переписать, чем разбираться с унаслдованным кодом
    - Быстрая обратная связь – ключевое условие гибкости. Если внесение изменения в пару строк требует нескольких дней от разработчика до эксплуатации, то никакого agile нет в принципе, не надо себя обманывать
* Прежде всего, обратить внимание на то, как сервисы взаимодействуют друг с другом, это позволит выделить грубые интерфейсы между основными бизнес доменами приложения
* Области ответственности отдельных сервисов неизбежно будут меняться по мере углубления нашего понимания бизнеса. Соответственно сервисы будут разбиваться (выделаться – основной микросервис при этом становится дирижёром набора подчиненных) – это нормально. Соответственно необходимо их организовывать таким образом, чтобы их было легко изменить или даже просто переписать некоторую часть с нуля
* Признаки слишком грубой гранулярности:
  + У сервиса слишком много ответственностей и сценариев использования. Соответственно слишком много юнит / интеграционных тестов, контролирующих его целостность
  + Слишком сложное хранилище данных (автор считает нормальным от 3 до 5 таблиц)
* Признаки слишком высокой гранулярности
  + Выполнение каждой бизнес операции требует вызова большого количества сервисов
  + Микросервисы сильно зависят друг от друга. Если при этом граф зависимостей кластеризуется, то это хороший способ объединения их в логически натуральные сервисы большего размера
  + Появляются микросервисы выполнения CRUD операций, а также множественные чисто технические микросервисы, методы которых не имеют бизнес ценности

## Работа с хранилищем данных

* Теоретически каждый микросервис должен иметь свое собственное изолированное хранилище данных, наилучшим образом подходящее к его требованиям. Не обязательно универсальную реляционную БД, для кого-то будет гораздо лучше NOSQL или вообще шаренный in-memory кэш данных
* Основные подходы к использованию реляционной БД. Каждому сервису:
  + набор таблиц
    - легчайший подход, для приложения мало что меняется
    - Типовой вариант для начала рефакторинга монолита в микросервисную архитектуру
    - целостность данных легко контролируется за счет ФК
  + схему в одном сервере БД
    - сложно шарить общие данные
    - Вместо **@\*To\*** аннотаций в JPA придется хранить FK как строки / числа и использовать как есть в сырых запросах
    - невозможно осуществить одновременный запрос данных в нескольких схемах
    - придется выполнять денормализацию общей схемы данных и вводить некоторую избыточность
    - Либо распределенные транзакции либо частичная несогласованность данных
    - проблемы с ФК в другую схему
  + собственный сервер БД
    - + все надостатки нескольких схем
    - Наиболее чистый и гибкий вариант. Например, если сервису требуется в основном read-only хранилище, то можно задействовать простенькую базу.
    - наибольшие накладные расходы

# Конфигурирование

* Крайне неудобно, что когда говорим о Spring мешаются две концепции: конфигурации бинов и конфигурация приложения

## Этапы работы с конфигурацией в Spring

* Изначально – **PropertyPlaceholderConfigurer**, препроцессор XML конфигурации бинов Spring, позволяющий использовать в качестве значений свойств ссылки на конфигурационные ключи, значения которых считываются из properties файла, определенного через URL, обычно с префиксом classpath: или file:
* Начиная с 3.0 / 3.1 версии
  + **@Inject Environment и @Value** на свойствах бина. Свойства грузим за счет того же **PropertyPlaceholderConfigurer** (он все равно нужен как препроцессор, поэтому регистрируется как статический бин) или **@PropertySource**
  + Механизм профилей **@Profile** позволяет определять какие наборы бинов будут активными в каком окружении.
    - Типовой пример: H2 источник данных для тестирования и нормальный для боевого развертывания.
    - Одновременно может быть активизировано несколько профилей или “default”, если не активизирован не один. Бин опционально конфигурируется набором профилей, для которых он активен.
    - Активизация бинов через конфигурацию, переменой JVM или окружения
* Затем Spring Boot
  + Добавлена целая иерархия источников конфигурационных свойств, которые могут перекрывать друг друга. От аргументов командной строки и JNDI до уже знакомых внутренних ресурсов приложения
  + В зависимости от активного профиля могут считываться properties файлы с различными постфиками
  + Конфигурация разделена на **bootstrap и application** конфигурацию, чтобы иметь возможность загрузить конфигурацию источников конфигурации
  + Вместо конфигурирования посредством **@Value** отдельных свойств, разброшеных по бинам, рекомендуется вводить специальные конфигурационные бины с **@ConfigurationProperties**
* Тем не менее остаются открытыми проблемы
  + После внесения изменений в конфигурацию требуется перезапуск приложения
  + Как отслеживать изменения самой конфигурации и при необходимости из откатывать
  + Конфигурация децентрализована и не всегда понятно куда смотреть, чтобы переконфигурировать тот или иной аспект
  + Конфигурационые данные не защищены

## Типовой подход

* Обычный способ экстернализации конфигурации – **json / xml / yaml** файл с необходимыми конфигурационными данными, который (которые, по одному для каждой среды) хранится под CVS и развертывается вместе с сервисом
  + Храниться может как отдельный артефакт, так и прямо в составе исходного кода сервиса. Последнее особенно ужасно – каждое изменение конфигурационной информации это отдельный релиз
  + Spring boot предоставляет легкое API, как собирать готовую конфигурацию из нескольких источников: от переменных среды, до конфигурационных файлов, причем дефолтных (включаемых внутрь сервиса) и оперативных (подкладываемых рядом при развертывании)
  + Как правило, содержат параметры подключения к другим системам, причем даже логины / пароли, что особенно неприятно. Есть решения, позволяющие шифровать, например - <http://www.jasypt.org/encrypting-configuration.html>
  + Решение допустимое для маленького числа экземпляров сервисов.
    - В противном случае умножаем несколько сот конфигурационных файлов минимум на три среды и на число нод, если должны указывать какие-то параметры индивидуально для каждой ноды. Получаем кошмар…
    - В облаке, когда один сервис может развертываться в сотнях экземплярах доставка такого файла в множество мест представляет головную боль.
    - Кроме того, в облаке обычно сервисы запускаются при помощи неизменяемых docker образов, соответственно чтобы подсунуть разные версии конфигурационного файла в разных средах приходится сильно изощряться. Это можно, но стоит сил и времени.

## Принципы управления конфигурацией:

* Runtime конфигурация не должна развертываться в составе или вместе с сервисом.
  + Вместо этого, параметры конфигурации должны доставляться сервису как переменные операционной среды (как правило, это немногочисленные bootstrap параметры типа наименование приложения, тип среды и URL сервиса публикации конфигурации) и затем считываться самим сервисом в централизованном репозитарии (многочисленные остальные параметры)
  + Антипаттерн – зашивать внутрь сервисов URL сервиса публикации конфигурации. Может быть непредсказуемые последствия, особенно при миграции между средами и замены сервиса публикации со старого на новый
  + В составе сервиса могут быть определены «дефолтные» значения конфигурационных параметров. Это нормально для параметров, относящихся к тюнингу поведения сервиса, определяемых в процессе разработки и редко изменяемых при развертывании
* Конфигурация должна поставляться сервису в абстрактном виде вне зависимости от механизма ее хранения.
  + Как правило, это пары ключ / значение, где ключи имеют иерархическое наименование.
  + Для Spring Boot нормально хранение в виде YAML файлов, которые сильно проще чем XML, но в отличии от properties наглядно показывают иерархию различных конфигурационных параметров.
  + Если же требуется автоматически процессить конфигурационные файлы в ходе развертывания (что крайне НЕ рекомендуется), то YAML крайне не удобен и лучше вернуться к классическим properties
* Так как система может состоять из сотен сервисов, то лучше минимизировать число репозитариев, хранящих конфигурационную информацию. Лучше как можно меньше вплоть до одного
* Сервис распространения конфигурационной информации в данном случае становится центральной точкой отказа, соответственно его развертывание требует особенного внимания к high availability & redundant.
* Сервис управления конфигурацией берет данные из какого-то хранилища, природа которого для сервисов не важна. Важно, чтобы
  + данные были под CSV и доставка изменений до сервисов во всех средах осуществлялась бы единообразно и автоматически, без необходимости применения ручных операций.
  + В противном случае неизбежны инциденты, когда что-то доехало до сервисов, что-то нет и в результате имеем зоопарк и несогласованность работы

## Основные сервисы хранения конфигурационной информации

* **Etcd** - open source проект, основанный на Go, предоставляет легковесный распределенный service discovery & key-value management и работу с распределенными блокировками. Использует Raft протокол для достижения консенсуса в ненадежной среде.
* **Eureka** – решение от Netflix, аналогично, в отличии от остальных предоставляет dynamic client refresh в дефолтной конфигурации
* **Consul** – решение от Hashicorp, аналогично, но на основе протокола SWIM. Предлагает нативный service discovery на основе DNS
* **Zookeeper** – Apache проект для работы с распределенными блокировками, тоже предоставляет хранилище key-value. Самое старое и наверное наиболее жестко протестированное решение. Но низкоуровневое и рекомендуется использовать только если его уже и так для чего-то ввели в проект
* **Spring Cloud Configuration Server** – может использовать в качестве источника для хранения информации файловую структуру, git, Eureka и Consul. В отличии от остальных НЕ-распределенное хранилище данных. При использовании в качестве хранилища Git автоматически достигается версионность конфигурации, для всех остальных сервисов это надо прикручивать вручную. Тесно интегрировано с остальными Spring инструментами и рекомендуется к использованию вместе с Spring Boot based сервисами.

## Consul

* Работа с данными – все через REST доступ, для тестирования можно применять curl
  + Создание - PUT -d 'From Consul Config' <http://localhost:8500/v1/kv/bookpub/my/config/value>
  + Получение
    - GET <http://localhost:8500/v1/kv/bookpub/my/config/value>
    - В результате имеем: [{"CreateIndex":20,"ModifyIndex":20,"LockIndex":0,"Key":"bookpub/my/config/value","Flags":0,"Value":"RnJvbSBDb25zdWwgQ29uZmln"}]
    - Параметр **ModifyIndex** будет использован в дальнейшем при редактировании в качестве значения для оптимистической блокировки с наименованием «cas»
  + Изменение - PUT –d 'newval' <http://localhost:8500/v1/kv/bookpub/my/config/value?cas=20>
* Вообще, архитектура развертывания следующая: кластер консул-серверов и на каждой ноде с приложением свой локальный агент-прокси для обращения к кластеру. Но для разработки можно натравить и прямо на единственный сервер
* Обычно приложение ничего о нем не знает, поэтому на ноде запускается приложение **envconsul**, которое
  + считывает конфигурацию из кластера
  + устанавливает его в качестве системных переменных окружения и запускает наше приложение
  + опционально мониторит изменение конфигурационных значений и при необходимости перезагружает приложение. По умолчанию шлет SIGTERM, что требует времени на правильное опускание приложения, но в принципе это настраивается за счет опации –kill-signal
  + параметрами запуска **envconsul** определяем как транслировать дерево занчений консула в конфигурационные переменные приложения

## Spring Cloud Configuration Server

* Зачастую PaaS, например, Cloud Foundry, предоставляет его как стандартный сервис. В таком случае его надо только запросить и сконфигурировать. Прежде всего ссылку на Git репозиторий с конфигурационными данными.
* В общем случае НЕ поставляется как отдельное приложение. Это просто библиотека, при помощи которой можно руками построить отдельно стоящий выделенный сервер или внедрить его в собственное приложение
* Зависимость **spring-cloud-config-server**, затем на главном файле **@EnableConfigServer**

### Хранение конфигурации

* Конфигурационные файлы для каждого приложения хранятся иерархически: папка **/applicationName/application-scope.yml** (для хранения, специфических для сред значений) или application.yml (для хранения дефолтных значений). Сервер:
  + Определяет механизм хранения информации через собственный **application.yml**
  + считывает эту информацию и выдает как REST ресурс с наименованием, соответствующему наименованию приложения.
  + По умолчанию это **8888/applicationName/${scope}.** Если scope = default, то выдает только дефолтные свойства. Если scope, к примеру, это dev, то выдает два набора свойств – дефолтные и переопределение для dev. Правильно сложить их между собой это задача клиента
* Использование файловой системы
  + Не рекомендуется, так как это как минимум означает, что место хранения файлов надо будет предоставлять как точку монтирования всем экземплярам серверов. В облаке такой шаринг фрагмента файловой системы может быть затруднен.
  + **spring.profiles.active = native**
  + **spring.cloud.config.server.native.searchLocation** – расположение на файловой системе отдельных директорий с конфигурационными данными для отдельных сервисов через запятую
* Использование в качестве хранилища Git или SVN
* **spring.profiles.active = git**
* дочерние свойства **spring.cloud.config.server.git : uri, username / password, searchPaths** (перечень под-директорий для каждого сервиса)
* Также имеется возможность хранения конфигурационных данных в базе
* Обычно ссылка на источник конфигурационой информации и сопутствующие свойства поставляются запускаемому серверу через переменные окружения

### Обращение со стороны клиента

* Зависимость **spring-cloud-config-client**
* В приложении есть два файла параметров:
  + **Bootstrap.yml** – параметры для инициализации, сюда кладем дефолтные (лучше фейковые, чтобы в случае ошибки сервис на prod не ломанулся в за dev конфигурацией) **spring.application.name, spring.profiles.active и spring.cloud.config.uri**
  + **Application.yml** – сюда кладем дефолтные значения параметров, которые используются, если они не переопределены каким-либо образом
* Доставка bootstrap параметров при развертывании сервисов в различных средах
  + Для каждой среды определяем переменные окружения – расположение сервиса конфигурации и наименование среды
  + Когда пакуем Docker контейнер значения переменных передаем как –D при запуске jar сервиса. Значение переменной «наименование приложения» фиксированное, для остальных считываем снаружи контейнера
* В любых компонентах применяем **@Value("${aaa.bbb:defaultValue}")** на **private** свойствах бинов и в аргументах метода-контруктора в конфигурации
  + особенно часто для **@Value("${local.server.port:0}")** – если порт не определен, то первый свободный
  + Рекомендуется определять все такие свойства в одном бине-конфигурации, чтобы остальные бины пользовались для получении конфигурации уже его методами. Чтобы протестировать конфигурацию на целостность можно использовать **@NotNull** и так далее, реализовать интерфейс **Validator** или просто выполнить проверку в **@PostConstruct** методе
  + На худой конец, всегда можно получить доступ к бинам типа **Environment** / **ApplicationArguments** считать свойство ручками
* Обновление конфигурации клиента
* По умолчанию конфигурация считывается только раз в процессе старта и затем не перечитывается
* **Spring Boot Actuator**добавляет **/refresh** URL, по которому за счет пустого POST запроса производится обновление конфигурации всех бинов, аннотированных **@RefreshScope**.
  + У таких бинов методы ЖЦ будут проигнорированы, а **@Value / @Autowired** вызываны повторно
  + Можно за счет **@EventListener** подписаться на **RefreshScopeRefreshedEvent** и сделать ручками все более аккуратно
  + Однако вызывать его надо ручками, что противоречит идеологии
  + Альтернатива **/refresh** URL это вызвать соответствующую команду через JMX
* Варианты что делать после изменения конфигурации
  + Программно обратиться к discovery service, получить все активные сервисы и на каждом вызвать **/refresh**
  + Просто по очереди опустить / поднять все сервисы
  + За счет **Spring Cloud Bus** разослать всем сервисам уведомление о событии изменения конфигурации
    - Зависимость **spring-cloud-starter-bus-amqp** + конфигурация **spring.rabbitmq.\*** подключения к сервису очередей RabbitMQ
    - Появляется новый URL в Actuator - **/bus/refresh,** который отправит сообщение в сервис очередей, а через него всем остальным экземплярам клиента

### Безопасность при работе с конфигурацией

* Хранить чувствительные данные типа логин / пароль в виде простого текста под CVS и распространять по сети достаточно больно. Поддерживается защита чувствительных данных как в части хранения (шифрование), так и в части передачи (идентификация / аутентификация пользователя)
* **Spring Cloud Config Server** поддерживает шифрование, как симметричное, так и ассиметричное. Чтобы им воспользоваться, необходимо скачать и установить в JRE набор Oracle JCE jars. Обычно это делается при помощи curl в процессе сборки Docker контейнера. В исходниках к книге есть пример соответствующих скриптов инсталляции
* Работа с ключом
  + Передать серверу, например, симетричный ключ как **ENCRYPT\_KEY** переменную.
  + НЕ надо зашивать ключ в Docker контейнер при сборке – это плохая практика и в случае необходимости быстро его сменить для какой-то среды будет иметь головную боль
  + Лучше, конечно, использовать ассиметричные: закрытый для сервера конфигурации и открытый для остальных сервисов. Это сделает систему более устойчивой ко взлому путем подмены конфигурационных переметров.
  + Spring Cloud Config Server предоставляет два служебных URL: **/ encrypt и /decrypt**, за счет этого можно работать ручками с конкретными значениями. Внимание, надо использовать POST
* Защищенное хранение
  + Теперь в конфигурационный файл записываем зашифрованные значения с префиксом **{cipher}** и не волнуемся, что их украдут из-под CVS.
  + Однако клиентам конфигурационная информация по-прежнему предоставляется в открытом виде.
* Защищенная передача
  + На сервере: **spring,cloud.config.server.encrypt.enabled=false**, теперь в результате REST запроса конфигурации будут зашифрованные значения с префиксом **{cipher}**
  + На клиенте
    - Зависимость **spring-security-rsa**
    - Симметричный ключ как переменная окружения **ENCRYPT\_KEY**
* Защита самого API
  + Зависимость **spring-boot-starter-security**
  + В простейшем случае HTTP BASIC
    - Свойства конфигурации security.user.name + password или поставляем собственную реализацию UserDetailsService
    - Клиент подключается как https://user:password@host.com

# Service Discovery

* При взаимодействии между микросервисами ни в коем случае нельзя полагаться на фиксированные IP адреса. Необходима косвенная адресация как средство динамического перенаправления вызовов и обеспечение гибкости развертывания
* Таким образом Service Discovery это критичная технология для микросервисов, обеспечивает:
  + Обнаружение сервисов-зависимостей, которым при старте в облаке присваиваются случайные IP. Это, во-первых, новые / начальные экземпляры, а во-вторых, перезапущенные по тем или иным причинам сервисы.
  + горизонтальное масштабирование с эластичностью мощности кластера. Без нее пришлось всегда держать число нод по максимому ожидаемой нагрузки
  + устойчивость к ошибкам, за счет возможности исключать из числа доступных сбойные или мертвые экземпляры сервисов
  + Возможность направление на сервис количества нагрузки, соответствующей его мощности
* Традиционные способы
  + Совсем плохо – каждому сервису подкладывается файл, в котором перечислены все URL его зависимостей, в итоге имеем дикий зоопарк версий файлов
  + DNS – достаточно традиционное, но не гибкое и медленное решение
  + Балансир запросов, (F5 или HAProxy)
  + UDDI – требует остальной тежелой программной инфраструктуры
* Проблемы с DNS
  + DNS информация кэшируется клиентом, поэтому придется для всех открываемых сетевых соединений задавать специальный TTL. Также придется настраивать его для репликации дынных между DNS серверами.
    - Это дополнительная нагрузка, при этом легко ошибиться или вообще не иметь возможности этого сделать, если код внутри какой-нибудь недоступной зависимости
    - Некоторые библиотеки кэшируют внутри себя получаемую от DNS информацию об IP адресах, что создает проблемы
  + В облаке это как правило занимает дополнительное время, так как запрос должен покинуть среду и вернуться обратно через общий со входными запросами маршрутизатор. Также это раскрывает внутренние сервисы для внешней адресации – проблемы с безопасностью Некоторые облака предлагают многосетевое DNS разрешение с открытыми и закрытыми адресами, что снимает проблему с производительность. Но при этом код должен принимать такую особенность во внимание и это сложно воспроизвести в локальной среде разработчика
  + DNS не предоставляет информации о состоянии сервиса. Таким образом, если клиент обратится к вышедшему из строю экземпляру, то он будет на некоторое время заблокирован. Соответственно, надо выставлять для всех URLConnection минимальный таймаут подключения и опять-таки проблема с третьим кодом
  + Проблемы с алгоритмами балансировки – только циклический перебор и round-robin. Все остальное, например, несимметричная балансировка, принимающая во внимание мощность экземпляра, отпадают
* Проблемы с балансирами
  + хорошо подходят
    - для традиционных систем, когда балансируемых экземпляров сервисов несколько, их число не изменяется, после перезагрузки они оказываются на том же самом месте (с тем же IP) а нагрузка на систему в целом предсказуема и меняется в узком диапазоне
    - для снятия SSL канала, а также для реализации политики сетевой безопасности (фильтрация портов). При этом это азачстую вход в систему в целом, который требует специального сертифицирования и соответствия PCI политики
  + единая точка отказа. В крайнем случае предоставляется возможность горячего резервирования (primary / secondary, hot swap model). Даже временный отказ балансира делает всю систему недоступной
  + проблема масштабирования. Особенно чувствительная, потому что они работают как прокси
  + конфигурация статическая и чаще всего задается руками - в принципе не предназначены для динамической регистрации / дерегистрации сервисов. Как правило автоматического распределения изменений конфигурации между серверами.
  + стоимость лицензирования
  + их обслуживание отделено от работы с сервисами и затрудняет обслуживание системы в целом
  + Если и способны привязать запросы пользователя к определенному узлу, то только на основании HTTP заголовков типа JSESSIONID. Зачастую хочется использовать для этого информацию из JWT / OAuth токена, например, при работе с потоковым видео.
* Требования к Service Discovery:
  + Серверная часть: высокая доступность без балансира запроса перед кластером, распространение изменений peer-to-peer, отсутствие центральной точки отказа, предоставление информации о доступных экземплярах по логическому имени (возможно с дополнительными метаданными, если экземпляры как-то разнятся между собой), периодическаая проверка жизнеспособности экземпляров
  + Клиентская часть: получение перечня точек доступ по логическому наименованию прикладного сервиса, автоматическое распределение нагрузки между экземплярами прикладных сервисов (client side load balancing), производительность и устойчивость к временной недоступности серверной части (кэширование конфигурации + переброс сбойного запроса + инвализация кэша), собственные регистрация / де-регистрация + предоставление информации о своем здоровье
  + Принципиальная особенность: сам Service Discovery найти экземпляры сервисов не в состоянии, она пассивна

## Spring Cloud + Netflix Eureka service discovery agent

* На основе Eureka Server и Ribbon клиент, который автоматически берет информацию о доступных сервисах с него
* Роли основных компонент:
  + **Eureka** - это реестр сервисов, каждый экземпляр при старте автоматически регистрируется в нем за счет библиотеки-клиента. Также можно зарегистрировать экземпляр произвольного сервис посредством REST запроса
  + **ServiceDiscovery** – возможность запросить у Eureka перечень URL экземпляров какого-то сервиса и запросить, например, по первому / случайному
  + **Ribbon** – реализация различных стратегий выбора конкретного экземпляра сервиса для запроса (балансировки)

### Серверная часть

* Как обычно: сервера как отдельного продукта нет. Собираем отдельное приложение или внедряем в себя
* Зависимость **spring-cloud-starter-eureka-server**
* Простейшая конфигурация без репликации между отдельными нодами
  + **eureka**
    - **instance.hostname** – наименование сервера, не всегда определяется корректно, например, в случае контейнера
    - **client**
      * **registrerWithEureka = false**
      * **fetchRegistry = false**
      * **serviceUrl.defaultZone = на себя самого**
    - **server**
      * **waitTimeInMsWhenSyncEmpty = 5** – сколько ждать при старте перед тем, как начинать отвечать. Стандартно ждет 5 минут, чтобы дать возможность всем сервисам зарегистрироваться
      * **enable-self-preservation** – если значительное число зарегистрированных сервисов не станет проявлять признаков жизни, то Eureka сочтет, что это проблемы с сетью и не станет удалять их из реестра. Иногда разумно установить true, но при этом в случае небольшого числа экземпляров (локальная разработки) Eureka перестанет отрабатывать перезапуск / остановку экземпляра сервиса
* На классе сервера: **@EnableEurekaServer**
* Важно: после того, как сервис зарегистрировался, Eureka три раза вызывает его **heartbeat** URL с паузой 10 секунд, прежде чем распространяет информацию о доступности данного экземпляра всем клиентам. Соответственно, сервис будет доступен клиенту только через 30 секунд – это надо учитывать в интеграционных тестах
* Предоставляет дефолтный веб-интерфейс на порту 8761 на «/» и JSON XML (по умолчанию, чтобы запросить JSON нужен соответствующий Accept заголовок) данные на **«/apps/<APPID>»,** где про каждый сервис
  + ИД - логическое наименование (его потом надо использовать в **@LoadBalanced RestTemplate)**
  + ИД экземпляра - используется в процессе регистрации для того, чтобы определить новые экземпляр сервиса был добавлен в систему или старый переподнялся
  + Где запущен, статус, URL доступа к странице с данными здоровья
* Чтобы запустить несколько серверов и заставить их обмениваться информацией
  + Устанавливаем для каждого сервера **eureka.client.serviceUrl.defaultZone** в перечень остальных через «,». Добавляем определение клиентского поведения.

### Клиентская часть - DiscoveryClient

* Зависимость **spring-cloud-starter-eureka** + **spring-cloud-starter-ribbon (**в принципе, ribbon клиент может работать и со статически заданным перечнем серверов без поддержки Eureka**)**
  + В принципе DiscoveryClient сособен работать не только с Eureka, но и с другими реестрами серверов: Apache Zookeeper, Consul, ETCD и другими
  + На основе **Apache HTTP Client** c дополнительными возможностями
* На классе приложения **@EnableDiscoveryClient** (в аннотации Eureka НЕ упоминается, так как может использоваться и другой сервис)
* Регистрация сервиса производится под именем, определенным как **spring.application.name в bootstrap.yaml**
* свойства приложения
  + **eureka.instance.hostname**
  + **eureka.client.serviceUrl.defaultZone: http://${ eureka.instance.hostname }:${server.port}/eureka/**
  + **registryWithEureka = true** – регистрируемся на сервере
* Дополнительные свойства:
  + **eureka.client.fetchRegistry=true** – если нам самим надо обнаруживать и вызывать другие сервисы. В результате приложение периодически выкачивает и кэширует данные с Eureka сервера, что требует некоторых усилий. Поэтому, если вызывают только нас – false
  + **eureka.instance.leaseRenewalIntervalInSeconds** – как часто оповещать сервер регистрации о том, что данный сервис еще жив, по умолчанию 30 секунд. В разработке лучше ставить меньше - часто перезапускаем, в процессе эксплуатации может быть больше.
  + **eureka.instance.preferIpAddress** – регистрируем с помощью IP адреса или логического наименования сервера. Первое требуется, например, когда сервис внутри Docker контейнер и отдельного имени не имеет. Ну и вообще надежнее.
  + **eureka.instance.metadataMap.instanceId –** уникальный ИД данного экземпляра сервиса. Стандартное значение **${spring.application.name}:${spring.application.instance\_id}:${server.port}.** Как правило НЕ требует переопределения. Можно добавить **${random.value}**
* Предоставление метаданных, на основании которых клиент сможет выбирать подходящий экземпляр сервиса. Например, сервис, обслуживающий определенную страну
  + **eureka.instance.metadataMap.someProperty** – произвольные статические метаданные, которые будут доступны клиенту
  + Предоставление динамических метаданных:
    - **@Autowired ApplicationInfoManager. getInfo().getMetadata()** и набиваем его необходимыми ключами со значениями.
      * Например, степень загруженности сервиса или отдельный регион, который он обслуживает.
      * Необходимо только помнить, что клиенты кэшируют данную информацию, так что быстро они обновления, увы, не получат. Поэтому лучше не «степень загруженности», а «доля всех запросов», соответственно мощности оборудования
    - Также данный класс удобен, чтобы управлять другими свойствами экземпляра сервиса с точки зрения Service Registry, например, статус.
* Четыре клиента: **EurekaClient**, **Spring Discovery / Spring Discovery + RestTemplate / Netflix Feign,** каждый последующий предлагает сервис более высокого уровня
* **EurekaClient –** полный нативный доступ к функциональности Eureka, в том числе возможность регистрировать / дерегистрировать экземпляры сервисов с произвольными метаданными. Внутри себя с Eureka работает через REST
* **Spring Discovery** – просто способ получить доступ к перечню экземпляров одного сервиса
  + **@Autowired DiscoveryClient.getInstance(serviceName)** возвращает список **ServiceInstance**, каждый из которых предоставляет информацию
    - **Host, port, uri, isSecure**
    - **Metadata в виде Map<String, String>**
  + Чтобы получить перечень всех сервисов **dicoveryClient.getServices(),** в том числе возвращается и собственное описание, доступное также через **dicoveryClient.getLocalServiceInstance()**
  + Соответственно, если мы потом пользуемся **RestTemplate** или любым другим клиентом (например, **Retrofit** с возможностью реактивных запросов), то
    - URL составляем сами, ровно, как и round robin балансир тоже реализуем сами
    - Вынуждены вручную создавать **RestTemplate**, что вообще-то не рекомендуется. Созданный вручную он ничего о Discovery Service знать не будет
  + Зато можно запрашивать только сервисы с определенными метаданными и реализовывать хитрые политики разбрасывания запросов
  + Дефолтная поддержка различных стратегий балансирования запрсов:
    - Создание стратегии балансирования как экземпляра интерфейса **IRule**, например, **RoundRobinRule**
    - Создание балансира **LoadBalancerBuilder** с определением стратегии балансировании и перечня серверов, полученного из **dicoveryClient**
    - Каждый раз получаем конкретный сервер за счет **loadBalancer.chooseServer()**
* **Spring Discovery + RestTemplate** – если мы определяем бин типа **RestTemplate** с квалификатором **@LoadBalanced (**можно его также куда-нибудь **@Inject** с даным квалификатором**),** то
  + к нему автоматически применяется специальная обертка на основе интерцептора HTTP запросов, позволяющая в URL вместо имени хоста использовать идентификатор сервиса
  + автоматический round robin запросов с исключением из перечня сбойных (высокий процент ошибок или таймауты) нод. Данный сервис реализуется Ribbon по умолчанию без необходимости дополнительной конфигурации
  + собирает статистику по производительности / запросам / ошибкам и так далее
* **Netflix Feign** – некоторый аналог **Retrofit**, работает поверх **Ribbon API**
  + Необходимо добавить аннотация **@EnableFeignClient**
  + Дальше для каждого сервиса определяем интерфейс
    - Аннотированный **@FeignClint(serviceName)**
    - На методах знакомые аннотации **@RequestMapping / @PathVariable** и так далее
    - Методы возвращают конкретные бизнес-типы вместо **ResponseEntity**, так что никакие загадочные шаблоны токена супертипа больше не нужны
  + Затем делаем **@Autowired** полей данного типа, реализация клиента поставляется системой
  + Ошибки доступны в виде **FeignException** с дополнительными свойствами, в том числе тело с разобранным JSON сообщения об ошибке. К сожалению по умолчанию маппинг различных ошибок на различные прикладные исключения не выполняется, но в документации есть примеры, как это можно обойти
* В принципе можно иметь всю функциональность Ribbon не используя соединение с Eureka
  + **ribbon.eureka.enable = false**
  + **serviceId.ribbon.listOfServices = url1,url2…**
* Чтобы дать возможность не JVM based сервисам работать с Eureka используется проект **Spring Cloud Sidecar**

## Consul

* Обычно на каждой ноде вместе с сервисом запускается приложение **envconsul**, которое полностью управляет сервисом и отвечает за его запуск / остановку / предоставление конфигурационных параметров и так далее
* Можно напрямую обращаться к консулу изнутри приложения
  + Стартер **spring-cloud-starter-consul-all**
  + На приложении нужно определить **@EnableDiscoveryClient**, чтобы соединиться с агентом
  + За счет свойств **spring.cloud.consul.discovery.\*** можно определить URL, за счет которого консул будет проверять жизнеспособность приложения (обычно ставим на **health** endpoint)

## Best practices

* Типовой вариант использования: локализация проблемы на работающей и тяжело нагруженной системе, генерирующей тонны логов и в принципе не позволяющей подключиться для дебага
  + Запускаем подозрительный сервис на машине разработчика, регистрируем в том же Eureka сервере, что обслуживает боевую систему, начинаем получать часть реальных боевых запросов
  + В случае наличия фасада на основе Zuul , регистрируем под специальным именем и динамически добавляем фильтр, который будет отправлять к нам только запросы, подозрительные на возможность возникновения проблемы
* Вопрос курицы и яйца: как компонентам окружения найти eureka сервер?
  + Фиксированное наименование в hosts
  + Запускаем eureka сервер первым, адрес кладем в переменную окружения, оттуда внутрь контейнера
* Увы, не все наши сервисывсегда JVM based. Также имеются клиенты граничных сервисов, основанные, к примеру, на iOS или Java Script (хотя имеются специализированные решения, например, для Eureka, позволяющие и в этом случае осуществить клиентскую балансировку).
  + В общем хочется иметь возможность осуществлять балансировку запросов клиентов всех типов к сервисам всех типов независимо от технологии реализации с единым набором задаваемых нами правил
  + Единственный вариант – введение граничных серверов, осуществляющих исключительно маршрутизацию / балансирование входящих запросов
  + Cloud Foundry предоставляет такие сервера как стандартные сервисы с некоторыми ограничениями. Например, не поддерживаются цепочки подобных серверов, осуществляющие иерархическую маршрутизацию
  + Увы, Spring Cloud на этом месте никакого готового функционала не предоставляет. Но это довольно легко реализовать такое самим.
    - Это будет HTTP сервис, принимающий все запросы для данного хоста, а затем, например, использующий первый сегмент URL как идентификатор сервиса, которому необходимо перебросить запрос при помощи **DiscoveryClient** и соответствующей политики балансировки
    - Необходимо специально сконфигурировать **RestTemplate** на доверие всем получаемым запосам и игнорирование ошибок
      * Создаем «доверяющую всему» реализацию **X509TrustManager**
      * Создаем «доверяющую всему» реализацию **SimpleClientHttpRequestFactory**, которая в операции **openConnection()** в случае, когда super метод возвращает **HttpsURLConnection** устанавливает ей **SSLContext** на основе «доверяющего» **X509TrustManager** и **HostnameVerifier** всегда возвращающий **true**
      * При создании **RestTemplate** устанавливаем ему «доверяющую» **SimpleClientHttpRequestFactory** и релизацию **DefaultResponseHabdler,** hasError() метод которой всегда возвращает **false**
    - Сам контроллер
      * просто извлекает RequestEntity из входного запроса и с помощью него выполняет запрос к нужному сервису, возвращая клиенту получаемый в итоге ResponseEntity
      * Необходимо уважать заголовки X-CF-Forwarded-Url, X-CF-Proxy-Metadata, X-CF-Proxy-Signature
  + Подобные граничные сервера также могут выполнять много всего: аутентификацию или преобразование аутентификации, ограничение трафика, разброс различным обработчикам в зависимости от типа клиента, обогащение запроса, логирование и мониторинг, а также многое чего другого
    - Лучше особо не разращивать их функциональность (особенно кросс-функциональность), оставляя ее для шлюза
  + Чтобы такие сервисы легко могли нести значительную входную нагрузку из хорошо делать на основе асинхронного, неблокирующего, реактивного API
* Каким образом регистрировать в Service Discovery сервисы не на основе JVM
  + Ручками, выведя наружу API регистрации / дерегистрации и предоставив со стороны сервиса стандартный endpoint для выполнения heartbeat
  + Решения партнерские для Service Eureka, например, Netflix Prana

# Client resiliency

* В распределенных системах в любой момент во время выполнения любого удаленного вызова надо иметь в виду возможность получения отказа или не получения ответа вовсе.
  + Это правило, а не исключение
  + Соответственно в случае проблемы вместо стека исключения клиенту необходимо показать что-то приличное, пусть и стандартное / устаревшее.
* Традиционный подход:
  + учитываем возможность полного отказа какого-нибудь сервиса в любой момент времени.
  + Соответственно лечим за счет исбыточности на каждом уровне – кластеризация, балансир запросов, распределение инфраструктуры на несколько площадок
  + Основная проблема: для не полученного ответа мы принципиально не можем определить был ли обработан запрос
    - Любой сбойный запрос повторяем, до тех пор, пока наконец не получаем ответ.
    - Сами сервисы делаем идемпотентными, рассчитанными на повторное получение запросов
* Однако, когда какой-либо сервис не прекращает полностью отвечать / начинает отвечать только ошибками, а всего лишь замедляет работу, то можно столкнуться с многочисленными проблемами:
  + Такая ситуация может случаться только время от времени и мгновенно распространяясь по системе (клиенты такого сервиса в свою очередь замедляются, а затем и их клиенты тоже), преводят к полному отказы всего
  + Если клиент начинает безконтрольно ждать результатов все новых блокирующих выполнение вызовов деградирующего сервиса и никакой защиты в него не встроено:
    - Новые запросы порождают все новые ждущие потоки, число потоков увеличивается, дополнительная память выделяется, все больше ресурсов тратится на переключение между ними и сбор мусора. JVM клиента (счастье, если в этой JVM только один клиент) замедляется в целом
    - Если такие долгоие запросы включены в локальную БД транзакцию, то
      * пул соединений с БД быстро переполняется (если нет пула или он не ограничен, то просто убиваем БД). Соответственно все другие процессы на клиенте, даже не связанные с обращением к деградирующему сервису, начинают тормозить в ожидании доступного соединения
      * База переполняется длинными транзакциями начинает сама по себе тормозить, тормозя всех ее клиентов (например, других, вообще не относящихся к делу сервисов)
    - Все аналогичные проблемы с ускоренным источщением ресурсов получает, разумеется, и сам деградирующий сервер, так что ситуация ухудшается с отрицательной обратной связью.
    - Пользователи, не получая вовремя ответы, начинают жать на кнопки с удвоенной энергией, перезагружать страницы и в результате нагрузка на систему еще возрастает
  + Что самое обидное, в результате долго ожидания такой клиент скорее всего получит таймаут или ошибку. То есть ожидание и все сопутствующие проблемы были совершенно зря
  + То есть, деградация распространяется по системе не только непосредственно через цепочки вызова, упирающиейся в деградирующий сервис, но и на процессы, происходящие «рядом». Ну то есть косвенно практически на всю систему и течением времени ситуация может болько автокаталитически ухудшаться.
* Необходимо отметить, что если клиент применяет асинхронные вызовы, то подобных проблем нет.
  + Но только настоящие асинхронные, то есть автоматическая конвертация синхронного вызова в JSM request / reply не подходит – клиентский поток все равно продолжает ждать и потреблять ресурсы.
  + Только «настоящий» реактивный стиль, при котором деградирует только сетевой уровень, который в большинстве случае достаточно мощный, чтобы выдержать размножение занятых сокетов

## Spring Retry

* Первоначально был частью Spring Batch, но теперь можно использовать как библиотеку отдельно
* За счет автоматического перезапуска фрагмента кода лечит проблемы
  + сервис не ответил. Например, потому что упал и еще не поднялся заново (Spring Cloud атоматически перезапускает упавшие сервисы) или просто еще не стартовал в процессе запуска системы
  + До сервиса еще не доехали необходимые данные вследствие асинхронной природы приложения
* Зависимость **spring-retry**, на конфигурации **@EnableRetry**
* На защищаемом методе **@Retryable** с основными свойствами
  + **Include** – для каких исключений повторять
  + **maxAttempts** – по умолчанию 3
  + **backoff** – управление задержками между попытками
* На резервном методе **@Recover**, затем параметры (все опциональные)
  + Тип обрабатываемого исключения
  + Параметры исходного метода
* Программный способ: конфигурируем **RetryTemplate** и используем его метод **execute** для выполнения фрагментов кода, обернутых в **RetryCallback**
* В обоих случаях на выполнение можно навешивать обертки в виде **RetryListener**, которые позволяют выполнять фрагменты кода до / после успешного / после ошибочного вызова защищаемого метода
* Основная проблема: будет тупо ломиться в сбойный сервис, укладывая и его и себя
* Достоинства:
  + Более понятный API
  + определение резервных методов за счет типизации – устойчивость к рефакторингу
  + на основе семафоров – не будет проблем с контекстом при выполнении кода в другом потоке
  + Требуется меньше вычислительных ресурслв и потоков (для Hystrix исходный поток стоит, а код выполняется в другом потоке, предоставляемом сконфигурированном пулом)
  + Не собирает статистики и соответственно нет способов визуализации происходящего

## “Fail Fast” на основе Hystrix

* Идея –обрывать вызовы к деградирующим серверам как можно раньше, не допуская распространения процесса деградации по всей системе. То есть это упреждающее отклонение потенциально сбойных запросов.
* **Client-side load balancing** с автоматическим исключением тормозящих нод
* **Периодический опрос сервисов** со стороны Service Discovery с автоматическим исключением больных серверов из конфигурации и отправкой их не перезагрузку.
  + Последнее может быть источником проблем: остановка больного сервера вызывает рост нагрузки на остальные, они в свою очередь деградируют и начинают перезагружаться. Свежие экземпляры сталкиваются с усилинной нагрузкой и не выдерживают. В результате весь кластер лихорадит
* **Circuit breaker** – мониторинг запросов, обязательное прерывание по таймауту (соответственно блокированные ресурсы высвобождаются) и временное прекращение вызовов тормозящего сервиса вообще, если процент сбойных / прерыванных запросов привысил заданный.
  + Периодически он таки пропускает запросы, тестируя здоровье сервиса, и в случае его выздоровление автоматически восстанавливает работу без необходимости вмешательства администратора
  + Сервис при этом может как «выздороветь» сам за счет временного снижения нагрузки на него, так и быть просто перезапущенным. Важно, что администратору при этом не надо предпринимать дополнительные ручные усилия
* **Fallbacks** – если за счет Circuit breaker прекращаем вызывать какой-то сервис, то начинаем вместо использовать резевную реализацию
  + Варианты альтернативного поведения - локальный кэш, дефолтные данные, отправка запроса пользователя на отложенную обработку в очередь и так далее
  + Да, какой-то фрагмент функциональности системы конечно при этом страдает, но система в целом выживает
* **Bulkheads** – выполнение удаленных вызовов рассаживаем по отдельным пулам потоков, чтобы в случае деградации сервера пострадали только операции, выполняемые в отдельном пуле.
  + Каждый такой пул это одельный изолированный отсек, обеспечивающий жизнеспособность всей системы
  + Переполнение пула сначала их замедляет, а затем он прекращает принимать новые запросы и срабатывает Circuit breaker
* Дополнительная валидация аргументов запроса к сервису перед отправлением, то есть еще на стороне клиента. Запрос все равно будет отвергнут, но мы сохраним время на удаленных вызов и пропускную способность сети. Также это позволяет выявить много ошибок на этапе юнит тестирования кода клиентов.

### Минимум

* Maven зависимости: **spring-cloud-starter-hystrix + hystrix-javanica**
* На конфигурации **@EnableCircuitBreaker,** без этого никакой защитный функционал просто абсолютно молча работать не будет
  + Как правило там же будет **@EnableHystrixDashboard,** чтобы анализировать текущее состояние сервисов
* Все методы, обращающиеся к потенциально могущим деградировать сервисам аннотируем с помощью **@HystrixCommand,** который создает соответствующие контролирующие прокси
  + Ну то есть вообще все удаленные вызовы
  + По умолчанию без дополнительной настройки все вызовы будут свалены в один пул, то есть **bulkheads** реально не будет
  + Соответственно, все операции с **RestTemplate / JpaTemplate / JdbcTemplate** также надо изолировать в отдельные методы
* В результате, если вызов такого метода длится более секунды (дефолтный таймаут), то получаем **HystrixRuntimeException**

### Настройка

* **Резервный метод**
  + За счет аттрибута **fallbackMethod**, которому передается наименование метода, располагающегося обязательно в том же классе и имеющий ту же самую сигнатуру.
    - Данное ограничение накладывается библиотекой Javanica, используемой «под ковром»
    - Spring Retry в этом смысле гораздо удобнее, так как ориентируется на типизацию и поэтому устойчив к рефакторингу
  + Если в этом методе тоже выполняем удаленный вызов, то его в свою очередь надо защищать при помощи **@HystrixCommand**
  + Если резервное поведением разное в зависимости от клиента метода, то просто try / catch на **HystrixRuntimeException** и не городим огород
* **Таймауты** 
  + За счет аттрибута аннотации **commandProperties**, которому передается массив **@HystrixProperties** с аттрибутами name / value
  + **execution.isolation.thread.timeoutInMilliseconds** – таймаут выполнения операции, по умолчанию 5 секунд.
    - Если требуется больше, то скорее всего имеем проблемы с производительностью в вызываемом сервисе – не надо поддаваться соблазны и увеличить таймаут, а надо исследоват потенциальную проблему.
    - Если же все-таки увеличиваем, то обязательно отсаживаем длинные операции в отдельный пул потоков, чтобы длинные операции не вытесняли из него короткие
* **Bulkhead**
  + Если не конфигурируем, то используется дефолтный пул из 10 потоков, через который пропускаются все запросы – веб, базы данных и так далее. Подходит только для приложений с малой интенсивностью запросов, редкий случай
  + Принцип: запросы к каждому удаленному сервису (или запросы, порожденные одним бизнес-процессом, реализуемым клиентом) в отдельный пул. Если запросы бывают короткие и длинные, то рассаживаем по разным пулом, чтобы длинные не вытесняли короткие из пула
  + Наименование пула в атрибуте **threadPoolKey**
  + Свойства пула за счет **threadPoolProperties** (тоже коллекция **@HystrixProperties**):
    - **coreSize** – по умолчанию 10
      * оптимальный размер: (maximum requests per second \* 99th percentile latency in seconds) + some extra for overhead
      * признак того, что размер слишком маленький – клиент тормозит при вполне здоровом сервисе, который он вызывает
    - **maxQueueSize** – по умолчанию -1, то есть использование **SynchronousQueue** и выбрасывание всех запросов, не помещающихся в размер пула. Если устанавливаем >1, то используется **LinkedBlockingQueue**
    - **queueSizeRejectionThreshold** – минимальный размера очереди, можно только если >0
* **circuit breaker**
  + в течение 10 секундного интервала измеряет, как часто вызов накрывается. Причем начинает измерять только после того, как прошло некоорое минимальное число вызовов удаленного сервиса
  + Если привышен процент сбойных (таймаут или ответ 500) , то отключает и начинает новое окно статистики. Каждые пять секунд пропускает вызов. Если он прошел – включает и начинает новое окно
  + Конфигурируется за счет аттрибута **commandPoolProperties** (тоже коллекция **@HystrixProperties**), основные имеют префикс circuitBreaker:
    - **requestVolumeThreshold** – после скольких вызовов в окне начинаем считать процент сбойных, по умолчанию 20
    - **errorThresholdPercentage** – процент сбойных, после которых отрубается сервис, по умолчанию 50
    - **sleepWindowInMilliseconds** – через какое время пробуем вызвать сервис опять, по умолчанию 5 секунд
  + Дополнительные, относящиеся ко сбору статистики
    - **metrics.rollingState.timeInMilliseconds** – размер окна в котором считаем статистику, по умолчанию 10 секунд
    - **metrics.rollingState.numBuckets** – число аккумуляторов статистики, используемых для анализа окна (размер окна должен нацело делиться на данное значение), по умолчанию 10
    - Чем меньше окно и больше число аккумуляторов, тем точнее будет собираемая статистика и тем больше ресурсов будет тратиться на ее анализ
* **Уровни конфигурирования**
  + Для приложения в целом через системные свойства
  + Для всех методов класса через **@DefaultProperties** на классе
  + Для отдельных методов внутри класса можем переопределить через параметры конкретного пула

### Кастомизация

* **HystrixPlugins.getinstance()** предлагает набор статических методов для доступа к основным компонентам:
  + Основные: **HystrixEventNotifier, HystrixMetricsPublisher, HystrixPropertiesStrategy, HystrixCommandExecutionHook, HystrixConcurencyStrategy**
  + В прикладной код Hystris попадает уже сконфигурированный, поэтому сначала сохраняем старые плагины, затем вызываем **HystrixPlugins.reset(),** затем за счет статических методов **HystrixPlugins.getinstance()** начинаем обратно регистрировать сохраненные или обертки над сохраненными, кастомизирующие их поведение
* Работа с потоками
  + Две стратегии, которые можно определить через **@HystrixCommand** свойство **execution.isoaltion.strategy**
  + Дефолтная **THREAD** – запросы выполняются в отдельном потоке в одном из сконфигурированных пулов.
    - Плюс – их легко прервать и код лучше изолирован
    - минус – это другой поток и имеем проблемы с **ThreadLocal** переменными, определенными в потоке, вызывающем метод – команду. Кроме того, излишние ресурсы – число задествованных потоков реально умножается на два.
  + **SEMAPHORE** – запрос выполняется в том же потоке, что и клиент запроса.
    - Нет проблем с **ThreadLocal** контекстом, на выполнение тратится меньше ресурсов.
    - Но в обычных синхронных контейнерах типа Tomcat прерывание такого метода может выбросить исключение, которое прикладной код не сможет перехватить. Поэтому рекомендовано к использованию только в Netty и тому подобных асинхронных средах, когда требуется очень большая пропускная способность.
* Обработка **ThreadLocal** контекста
  + Проблема: с дефолтной стратегией код команды выполняется в другом потоке, нежели клиент команды, и не имеет доступа к клиентским **ThreadLocal** переменным
  + **ThreadLocal** переменные увы часто используются, например, чтобы хранить контексты: транзакционный, безопасности, correlationID, получаемый с клиента и связывающий все запросы в одну логическую транзакцию и так далее.
  + Частый шаблон в традиционных не-асинхронных веб приложениях: фильтр на входе вычленяет какие-то загловки из HTTP запроса и кладет в **ThreadLocal** переменные.
    - Эти переменные доступны всему коду вниз по стеку, могут быть автоматически подхвачены HTTP клиентом и подмешены к исходящим удаленным вызовам, переданы при вызове в БД, использованы при авторизации запрошенной операции и так далее
    - Если команда выбрасывается в отдельный поток, то при формировании вложенного удаленного вызова контекст увы более не пробрасывается, так как он остался в вызывающем потоке
  + Hystrix предлагает механизм для пересаживания контекста в поток-исполнитель внутри пула – **HystrixConcurrencyStrategy**
    - Увы, можно посредством **HystrixPlugins** определить только одну глобальную стратегию и это уже сделал Spring, чтобы пересаживать собственный контекст безопасности. Но мы можем обернуть ее своей собственной (делегат)
    - Создаем собственную стратегию, расширяя стандартную **HystrixConcurrencyStrategy**.
    - В конструкторе принимаем параметр **delegate типа HystrixConcurrencyStrategy** и сохраняем. Этот nullable параметр – опциональная переопределенная ранее стратегия
    - Логика практически всех методов: если **delegate==null,** то вызываем super метода, если нет – соответствующий метод на **delegate**
    - Интересующий нас метод – **Callable wrapCallable(callable).**
      * Он вызывается в клиентском потоке и создает **callable**, которая будет отправлена в пул и выполнена в отдельном потоке.
      * Соответственно обертываем параметр своим **Callable** классом, который принимает два аргумента – старый **callable** и набор параметров, вытащенных из текущей **ThreadLocal**. В методе call, который будет вызван в отдельном потоке, мы
        + сначала устанавливаем в поток необходимые **ThreadLocal** переменные,
        + затем выполняем исходный **callable**
        + затем в final блоке чистим эти переменные, чтобы их по ошибке не использовал другой код, который будет исполняться данным потоком (потоки в пуле кэшируются и используются повторно)
    - Теперь осталось в процессе старта приложения (обычно **@PostConstruct** на классе конфигурации)
      * получить за счет **HystrixPlugins** текущую «старую» стратегию
      * создать «новую» собственную стратегию, передав старую в качестве параметра – делегата
      * установить «новую» стратегию в процессе переинициализации **HystrixPlugins**

### Объединение команд

* Пример: пользователи просматривают перечни книг и для каждой надо запросить «оценку» из внешнего сервиса
  + Так как для каждого вызова данного сервиса имеется фиксированная задержка, то дергать его для каждой книги / несколько книг расточительно
  + Вообще-то такая задача решается гораздо эффективнее через внутренний кэш, но с ним будут проблемы с актуальностью данных, что не всегда приемлимо
* Если каким-то умудремся группировать запросы, то
  + Время выполнение каждого отдельного запроса, конечно, слегка возрастает на длинну «окна» группировки. Эгоистично применяемое распараллеливание запроса снижает задержку для одного запроса, но потенциально ухудшает производительность системы в целом из-за исчерпания ресурсов
  + В целом пропускная способность системы возрастает – меньше потоков ожидающих ответа, число соединений на сетевом уровне, эффективнее используется полоса пропускания сети, пакетирование запросов значительно повышает эффективность использования базы данных
  + Ресурсы распределяются между запросами более равномерно и поведение системы становится более предсказуемым
* Реализация
  + Разумеется, что сервис должен поддерживать подобные “batch” запросы. Кроме того, каждый «результат» должен каким-то образом быть коррелируемым с запросом
  + Общий принцип: в течении времени накапливаем «похожие» команды, сворачиваем их в одну, выполняем, затем общий результат выполнения разбиваем на отдельные ответы. Вручную реализовывать довольно геморно: глобальный реестр запросов, межпотоковая синхронизация и всякое такое
  + Hystrix предлагает дефолтную реализацию процесса на основе **HystrixObservableCollapser**
    - В конструкторе запоминаем аргумент и задаем свойства свертывания: длинна окна в мс и максимальное число аргументов в свертке. По умолчанию 10 мс и 50 запросов
    - Дальше, в **createCommand()** мы ручками объединяем отдельные запросы в групповой запрос к серверу
    - Разборка отдельных результатов по отделным запросам поддерживается следующими методами:
      * **getRequestArgumentKeySelector()** – функция, выделяющая из аргумента каждого запроса логический ключ
      * **getBatchReturnTypeKeySelector()** – функция, выделяющая ключи из ответов
      * **getBatchReturnTypeToResponseTypeMapper()** – функция, которая на основе выделенных ключей, сопоставляет один эдемент ответа одному исходному запросу
* Недостатки
  + Очевидный – пакетирование вносит дополнительную задержку на стороне клиента в среднем на половину длинны окна пакетирования
  + Пакетирование не имеет смысла при низкой нагрузке: каждый пакет будет состоять из одного запроса и мы только добавим задержку на процессы пакетирования / разбора
  + API должно быть адаптировано под процесс пакетирования

### Мониторинг

* Чтобы работать Hystrix в любом случае собирает статистическую информацию. Поэтому очень соблазнительна возможность ее использовать
* Методы доступа:
  + Программный
    - Например, **HystrixCommandCompletionStream** позволяет подписаться на события окончания выполнения каждой команды с определенным груповым ключом.
      * Событие содержит в себе информацию о выполнении. Например: успешность, число попыток и так далее.
      * Благодаря этому легко подсчитать число отказов в секунду: **filter().window().flatMap( Observable::count() )**
    - Аналогичные: **HystrixCommandStartStream, HystrixCollapserEventStream** и так далее.
  + Модуль **hystrix-metrics-event-stream** предоставляет средства отправки аггрегированных метрик посредством HTTP
    - **HystrixMetricsStreamServlet** – предоставляет поток статистики в режиме реального времени на основе SSE (выталкивает данные на клиента). По умолчанию, данный поток данных доступен как **http://localhost:8000/hystrix.stream**
* Визуализация
  + Модуль **hysteric-dashboard –** автономное приложение на основе JS,
    - Это JS приложение , которое получает поток SSE данных и для каждой команды обновляет и публикует в реальном режиме времени статистическую информацию:
      * Количество команд по типу завершения
      * Процентили задержки
      * Состояние прерывателя и пропускная способность
      * Статистика пула потоков, если используется блокирующий класс **HystrixCommand**
    - Чтобы активизировать **@EnableHystrixDashboard**, после чего UI доступен как **/hystrix**
    - Проблема в том, что данное JS приложение может получать данные только из одного источника
      * По умолчанию, панель может работать с любым источником данных, который конфигурируется пользователем при обращении. Поэтому рекомендуется сразу же создать URL, который откроет ее уже направленной на текущий сервис
      * Таким образом, для каждого экземпляра сервиса имеем свою панель с информацией о здоровье удаленных вызовов. Это полезно, но неудобно.
  + Проект **Turbine** позволяет аггрегировать потоки данных от нескольких серверов и поставлять в качестве единого потока SSE событий для JS приложения визуализации
    - Можно в конфигурации статически перечислить URL, откуда брать данные, а можно получать их через **DiscoveryClient**, в данном случае перечень источников будет актуально обновляться. Также можно сконфигурировать получение данных через stream на основе RabbitMQ / Kafka
    - Конфигурирование аналогичное
      * Всем модуль **spring-starter-cloud-stream-rabbit**
      * Turbine сервер на основе **spring-cloud-starter-turbinne-stream** модуля + **@EnableTurbineStream**
      * Для всех сервисов добавляем модуль **spring-cloud-netflix-hystrix-stream**, который отправляет данные в поток на turbine сервер

# Service routing на основе Zuul

* Был разработан компанией Netflix в качестве единой точки доступа (service gateway, шлюза) ко всем ее продуктам
  + изначально жестоко тестировался в боевых условиях и остается адекватным современным требованиям
  + В настоящее время за счет более 20 кластеров серверов поддерживает доступ примерно тысячи различных типов устройств, сотни протоколов, порядка 100 миллионов клиентов с интенсивностью порядка 50 тысяч запросов в секунду и десятки миллиардов запросов в день
  + Наименование из фильма «Охотники за приведениями» - Gatekeeper of Gozer
* Основной функционал это встраиваемый stateless реверс-прокси, предоставляющий унифицированный интерфейс (единую точку доступа) ко всем сервисам системы.
* Основная задача шлюза – реализация кросс концепций.
  + Можно конечно это делать на основе программных интерцепторов, индивидуальных для каждого сервиса и стандартных, поставляемых библиотекой, совместно используемой всеми сервисами, но
    - Сервисы могут быть не java based или даже java based, но унаследованные
    - Трудно обеспечить эквивалентную реализацию общей логики всеми командами. Особенно, если учесть, что эти кросс концепции достаточно трудны в реализации,

прямого отношения к бизнесу не имеют и будут реализовывать по остаточному принципу

* + инфраструктурная библиотека это уже лучше, но
    - все равно встает вопрос ее правильного внедрения в код
    - также проблема нестабильности – любое изменение в ней потребует повторного тестирования и развертывания всех остальных сервисов. Соответственно мы теряем возможность гибко управлять правилами обработки.
  + Можно реализовать ряд граничных сервисов, но тоже получим в итоге некоторый зоопарк. Особенно, когда речь пойдет о функциональности, которая должна быть реализована во всех граничных сервисах одинаково
* Соответственно возникает идея реализовать примение кросс концепций в качестве прокси, перехватывающего / маршрутизирующего все запросы к сервисам и имеющего возможность произвольно их менять
  + Интересная серия статей по развитию API, использованию Zuul, исследованным подводным камням и тому подобного от создателей “Riot Games API”
  + Одна из альтернатив – облачное решение Apigee от Google, ничего не надо ставить локально, дружит с основными известными облачными провайдерами
  + В настоящее время на основе Spring 5 реактивного доступа предлагается Spring Cloud Gateway как более адаптивный. Другие популярные альтернативы – Mashery, Apigee, Kong, WSO2, 3scale и до хрена облачных провайдеров
* Проблемы с многообразием клиентов
  + До хрена различных типов клиентов: HTML5, Android, iOS, PlayStation, Xbox, различные SmartTV и так далее
  + Каждый тип клиентов обладает своими особенностями
    - Ограничение объема памяти или вычислительной мощности – часть операций надо перекладывать на сервер, может справиться с меньшим объемом получаемых данных
    - Особые типы содержимого или кодировки
    - Различные принимаемые модели данных, метаданные запросов / ответов
    - Различная площадь экрана может требовать загрузки данных по частями, а не сразу
    - Для одних клиентов более эффективная потоковая модель доставки данных, для других порционная
    - Различные сценарии взаимодействия с пользователем
  + Если подгонять каждый микросервис под каждый клиент, то получится зоопарк и значительное усложнение системы. Значительно легче вынести обработку специфики клиентов на граничные сервисы

## Основной функционал

* Всего в zuul кроме ядра содержится порядка 40 готовых для использования компонент (в большинстве своем фильтров), хотя обычно используется гораздо меньше.
  + Также поддерживает разработку собственных фильтров, реализуемых либо как Java классы, либо как Groovy скрипты, динамические подгружаемые без перезагрузки сервера из сконфигурированного источника
  + Вообще это фактически больше конструктор фасада, нежели готовая их коробки к использованию система
* Аутентификация и авторизация запросов от многих типов клиентов к многочисленным сервисам – Policy Enforcement Point (PEP)
* Снятие SSL
* мониторинг трафика
  + Исследование потока входных данных
  + выделение статистики, позволяющей понять, что происходит со всей системой, отдельными компонентами, различными типами трафика / потоков данных и так далее
  + логирование запросов
  + трекинг запросов: внедрение correlationId, который впоследствии полуавтоматически подмешивается во все вызовы сервисов друг друга
* Динамический роутинг запросов
  + Функциональное тестирование отдельных компонент за счет перенаправление на них трафика выделенных для тестирования клиентов
  + Нагрузочное тестирование отдельных компонент за счет перенаправление на них выверенной доли входного трафика
  + Ответвление части трафика для исследования / отладки сбойных операций (SurgicalDebugFilter).
    - Обычно мы знаем признаки подобных запросов (бизнес данные, типы клиентов, определенные операции) и можем ориентировать добавляемые фильры именно на них
    - Очень сложно ловить подобные ошибки в потоке остальных запросов, логировать их. До этого Netflix использовал Hadoop для исследования биллионов строк записанных логов запросов / ответов.
    - С фильтрами подобного типа появляется даже возможность поставить отдельный сервер и осуществить на нем пошаговую отладку сбойного запроса
  + Быстрое исправление проблем
  + Canary testing – периодическое выполнение тестовых запросов, тестирующих определенные маршруты вызовов
  + Автоматическое подключение / отключение серверов кластера (registry / discovery)
  + Защита чисто внутренних сервисов от внешнего доступа
* Определение правил роутинга
  + Статическая конфигурация - определяемая локально или загружаемая извне
  + Программная реализация - за счет фильтров, которые имеют возможность в выполнять роутинг как за счет данных, обрабатываемого запроса, так и за счет информации, получаемой от других сервисов
  + Автоматический раутинг - за счет получения информации от серверов Eureka / Consul, в которых регистрируются экземпляры сервисов при старте
* Сокрытие реального разбиения системы по компонентам и подробностей развертывания.
  + Клиент видит единый адрес, из которого торчат все интерфейсы. За ним может быть произвольный зоопарк, эволюционирующий по мере необходимости
  + Сервисы опять-таки могут ничего не знать о развертывании друг друга и обращаться за всеми зависимостями на единую входную точку
* Аггрегация вызовов нескольких сервисов для одного запроса клиента, изменение грануляции методов API / используемых ими данных, миграция сервисов между серверами и прочая эволюция реализации
* Решение проблем CORS (cross origin resources sharing, простейший случай – UI клиент поддерживается одним сервисом, а данные пытается подтягивать с другого)
* Произвольная трансформация запросов / ответов, например, трансформация API
  + Унификация API, предоставляемого несколькими провайдерами
  + Поддержка версионирования
  + Усечение / исправление URL, заголовков, кодировок
* Предоставление статического контента
* Распределение нагрузки
  + по серверам / типам запросов
  + квотирование доступа пользователей
  + отсечение излишней нагрузки и сброс нагрузки в случае возникновения воронки ошибок
* Диспетчеризация запросов по региональным центрам обработки

## Архитектура

* **ZuulFilter runner** – обработчик входящих запросов на основе цепочки фильтров
  + Фильтры реализуются на основе подгружаемых Groovy скриптов (в процессе эксплуатации копируются в виде исходного кода в определенные директории, откуда подхватываются, компилируются и встраиваются в цепочку обработки) или доступных статически Java классов
  + Каждый фильтр характеризуется
    - Тип – на каком этапе обработки запроса фильтр должен быть использован, основные
    - Порядок вызова
    - Критерии – фильтр на свойства запроса / ответа
    - Собственно логика выполнения фильтрации (а можно ничего не фильтровать, а сразу сформировать ответ)
  + Фильтр может либо выполнять все необходимую работу сам, либо пользоваться услугами других серверов API. Например,
    - Можно выделить отдельную группу серверов для учета лимита частоты запросов и ведения black листов клиентов. Все они будут хранить общую информацию в Redis базе данных. Это синхронная операция и она должна быть выполнена маскимально быстро, чтобы не тормозить процесс фильтрации. Выполнение необходимо тщательно мониторить.
    - Сбор и отсылка метрик в хранилище напротив есть асинхронная операция, поэтому ее можно разбивать в целях безопасности на отдельные этапы с промежуточным хранилищем, защищающем основной процесс
      * Собирать можно в глобальную для сервера коллекцию и выделить отдельный поток на периодическую отсылку на сервер статистики.
      * Там реальную запись от фактического выполнения запросов к БД также можно отделить очередью в памяти, локальной или на основе REDIS.
      * Таким образом, в случае проблем с производительностью будет страдать не Zuul, а сторонний сервис, реализующий дополнительный сервис.
* **Filter File Manager** – опрашивает директории на предмет добавленных / удаленных / измененных скриптов с определениями фильтров
* **Filter Loader** – создает экземпляр фильра и добавляет в цепочку, управляемую ZuulFilter runner
* За пределами остается система, которая предоставляет администратору услуги для управления в терминах обслуживаемого бизнеса.
  + Данная система как-то хранит описания, конфигурирует на ее основе фильтры и выталкивает в директории, откуда их подхватывает **Filter File Manager**
  + Разработка подобной системы – зона ответственности поставщиков API
  + Фильтры в процессе своей работы могут также нуждаться в свежей конфигурации
    - Простейшая реализация – они запрашивают систему управления. Плохо, так как она обычно не сильно быстрая и в результате производительность системы в целом будет зависеть от нее
    - Хорошо: push подход, все изменения в конфиругации выталкиваются в высокопроизводительное и тупое внешнее хранилище, например, в S3 buckets, откуда ее запрашивают фильтры и другие компоненты
    - Через экстернализованную конфигурацию самого Zuul, загружаемую из Eureka
* Так как цепочка фильров может быть перестроена в любой момент, то у администраторов появляется мощный инструмент управления системой в целом, за счет возможности быстро добавить фикс, перебросить потоки обработки, ответвить поток сбойных запросов для исследования и так далее.

## Серверная конфигурация

* Как обычно это не готовый к использованию сервер (в отличии от, к примеру, NGINX), а Spring Boot компонент, который может встраиваться в любые приложения и работать как прокси.
  + Вполне допустимо, когда в одном приложении запускается и основное приложение, например пользовательский интерфейс, и zuul based прокси, позволяющее ему без проблем использовать API, расположенное где-то еще. В этом случае UI запрашиет все сервисы по относительному URL, как будто они развернуты на localhost, а zuul выполняет проксирование
* .Основная зависимость **spring-cloud-starter-zuul** + аннотация на конфигурации **@EnableZuulProxy**
  + Есть еще аннотация **@EnableZuulServer**, но она стартует «пустой» сервер без загруззки reverse proxy фиильтров и использовании Eureka. Используется, когда мы полностью самостоятельно хотим построить функционал шлюза или завязать его с другим discovery сервисом, напримр, Consul
  + Автоматически подключается набор дефолтных фильров, а также Ribbon и Hystrix для выполнения запросов к проксируемым сервисам
  + Перечень стандартных фильтров в пакете [org](https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-core/src/main/java/org)/[springframework](https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-core/src/main/java/org/springframework)/[cloud](https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-core/src/main/java/org/springframework/cloud)/[netflix](https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-core/src/main/java/org/springframework/cloud/netflix)/[zuul](https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-core/src/main/java/org/springframework/cloud/netflix/zuul)/filters модуля [spring-cloud-netflix-core](https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-core). Чтобы запретить выполнение какого-то фильтра: **zuul.<Simple Class Name>.<filter type>.disable=true**
* По умолчанию умеет автоматически пробрасывать запросы к сервисам на основании информации, загружаемой из Eureka по serviceId, выделяемым как первый фрамент URL перехватываемого запроса
* Минимальная конфигурация
  + **eureka.instance.preferIpAddress=true**
  + **client.registerWithEureka=true**
  + **client.fetchRegistry=true**
  + **client.serviceUrl.defaultZone=…**

## Роутинг входных запросов

* Для всех источников конфигурации роутинга эффективный результат доступен в виде единого перечня как **host:5555/routes** в виде пар **<фильтр на входящий URL> / <идентификатор сервиса, куда будет правлен запрос>**
  + Ингда хочется разнести подобные служебные URL шлюза и обращения к проксируемым сервисам. Для этого **zuul.prefix=/api**
* Так как сам zuul использует Ribbon + Hystrix, то после выяснения маппинга запрос к сервису будт отправлен с использованием клиенского балансира, а также всех средств client resilency
* Атоматический при помощи service discovery
  + В качестве serviceId используется первый фрагмент входящего URL, соответственно запрос на **/organizationservice/\*\*** будет отправлен сервису, зарегистрировавшемуся в Eureka как **organizationservice**
  + Если в Eureka есть хотя бы один зарегистрировавшийся экземпляр какого-то сервиса, то соответствующая строка маппинга появится в **«/routes».**
  + Это не очень хороший вариант – все сервисы, зарегистрированные в Eureka, могут быть вызваны. Лучше все-таки:
    - таки определять сервисы один за другим ручками и в них за счет свойства **serviceId** прописывать связь с именем сервиса в Eureka
    - За счет свойства **ignoredServices: \*** запретить всем сервисам авто резолвиться за счет Eureka. Можно также запртить авто резолвиться конкретным сервисам, перечислив их через запятую
* Ручной
  + за счет определения свойства конфигурации
    - **zuul.routes.organizationservice = /organization/\*\***
    - или пары
      * **zuul.routes.someId.path = /organization/\*\***
      * **zuul.routes.someId.serviceId = organizationservice**
  + Такие маппинги всегда видны в **/routes** независимо от наличия зарегистрированных в Eureka экземпляров сервиса. Еслиже таковых нет, то обращение вернет 500
  + Это добавит еще одно правило, автоматическое при этом никуда не денется и так же будет выполнять роутинг (если конечно хоть один экземпляр сервиса зарегистрирован). Соответственно автоматический маппинг для таких сервисов надо отключать
* Статический маппинг – на сервисы, которые НЕ зарегистрировавны в Eureka
  + В основном это НЕ JVM based сервисы
  + **zuul.routes.someService**
    - **path: /serviceid/\*\* -** как вычленяем запрос
    - **url:** [**http://some-server:8080/api**](http://some-server:8080/api)
  + в **/routes** в правой части вместо **serviceId**, используемого при обращении к Eureka, будем иметь конкретные URL
  + Недостаток – так как не используется Eureka, то имеем роутинг только на одну точку. Можно выключить интеграцию Ribbon с Eureka и ручками настроить ему перечень серверов для баланса
    - **ribbon.eureka.enable = false**
    - **zuul.routes.someService.serviceId = someServiceId**
    - **someServiceId.ribbon.listOfServices = url1,url2…**
  + Проблема, когда в одной конфигурации используем и статический и автматический / ручной маппинг.
    - Так как Ribbon выключен, то данные с Eureka не кэшируются и для не-статического маппинга проксирование каждого запроса означает предварительное обращение к Eureka за lookup реального URL сервиса. Поэтому для проксирования не-JVM сервисов рекомендуется разворачивать отдельный Zuul кластер
    - Другое решение это Spring Cloud Sidecar, позволяющий не-JVM сервисам регистрироваться / де-регистрироваться в Eureka
  + **zuul.routes.\*** свойства обрабатываются **ProxyRouterLocator**. У него есть еще много интересных свойств, например
    - **retryable** + остальные связанные – ribbon клиент будет пытаться повторять сбойные запросы несколько раз
* Перезагрузка конфигурации
  + Конфигурация маппинга это такая же конфигурация, как и любая другая, а значит мы можем ее экстернализировать в Eureka для сервиса zuulservice (стандартное наименование, можно соответственно переопределить)
  + Соответственно настраиваем Eureka работать с git, коммитим новыую конфигурацию и дергаем **/refresh** на всех zuul серверах
  + Можно аналогично сделать POST на /routes, что перегрузит не все refresh scope бины, а только конфигурацию маршрутизации
* Таймауты
  + Так как по умолчанию используем Hystrix, то если проксирумый сервис не уложился в дефолтную секунду, то клиент получает 500. Чтобы переопределить
    - **hystrix.command.default.execution.isolation.thread.timeoutInMiliseconds = xxx** – для всех сервисов
    - Если нужно для какого-то отдельного, то вместо **default** используем ИД сервиса
  + Так как по умолчанию используем Ribbon, то если проксирумый сервис не уложился в пять секунд, то тоже имеем 500
    - Лучше не переопределять, а выяснять, почему же сервис так тормозит
    - **ribbon.ConnectTimeout и ribbon.ReadTimeout**
    - **serviceId.ribbon.ReadTimeout и serviceId.ribbon.ConnectTimeout**
  + таким образом, если сервис не укладывается в пять секунд, то надо переопределять ОБА таймаута
* Загрузка файлов через Zuul
  + Маленькие файлы можно загружать стандартным образом, ничего не поменялось. С большими файлами возникают проблемы при передаче через стандартный Spring **DispatcherServlet**: хочется избежать multipart processing, так как в данном случае он не имеет большого смысла
  + Zuul представляет собственный механизм, оптимизирующий загрузку больших файлов, реализованный через префкис пути **/zuul/:**
    - для сервиса, определенного как **zuul.routes.customers = /customers/\*** загрузка больших файлов возможна через **/zuul/customers/\***
    - Префикс можно настраивать посредством **zuul.servletPath**
  + Загружать большие файлы надо с **Transfer-Encoding: chunked**
* Фильтрация критических для безопасности HTTP заголовков
  + Речь идет о трех: **Cookie, Set-Cookie, Authorization**. По умолчанию они убираются из проксируемых запросов
  + Можно точно настроить их список за счет свойства **zuul.sensitiveHeaders**. Пока свойство не определено, то фильтруются все три, если перечислены через «,» конкретные, то только они.

## Фильтры

* Ручное проксирование запросов, фактически это функционал аналогичный JEE Servlet Filter / Spring Aspect, но реализуемый без вмешательства в код конкретных сервисов

### Типы фильтров

* Обычные **javax.servlet.Filter**, обрабатывающие запросы к самому **Zuul Servlet**
  + Например, чтобы добавлять CORS специфичные заголовки при работе JS приложений с некоторыми граничными сервисами. Автоматически добавляет к ответу, направляемому прилжению, заголовки типа **Access-Control-\***
  + Функциональность стандартная, но достаточно ограниченная
  + В принципе за счет **RouteLocator**, можно разобраться какой именно проксируемый сервис мы вызываем, но это не слишком просто. Приходится адаптировать схему маршрутизации под логику фильтра, что не очень правильно
  + Такой фильтр требует статической линковки кода фильтра с приложением самого сервера. Остальные фильтры можно делать на основе Groovy скриптов динамически подгружаемыми / выгружаемыми из сконфигурированной директории
* **PRE** – применяются к влиентскому запросу самыми первыми
  + для контроля формата клиентского запроса, аутентификации /авторизации
  + имеют возможность разбирать и модифицировать входной запроса, не могут выполнять его перенаправление
* **POST**– применяются после получения ответа от проксируемого сервиса
  + для логирования, мониторинг, обработки ошибок, аудита
* **ROUTING** – после pre-filter для произвольного перенаправления запроса
  + обычно для манипуляции трафиком. Например, для ответвлении запросов определенной группы клиентов на выделенные сервера, потениально проблемных запросов на машину разработчика дла анализа, отдельных тестовых пользователей на сервера с новым функционалом и так далее
  + она НЕ выполняют перенаправление в HTTP смысле (то есть посредством 302 заголовка), вместо этого они формируют другой запрос, выполняют его посредством Ribbon или Apache HTTP Client и на основании полученных данных формируют ответ клиенту.
  + имеют максимальный контроль над трафиком, однаго это компенсируется трудоемкостью реализации
* **ERROR** – обработка ошибок на одной из предыдущих стадий
* **STATIC** – немедленная генерация ответа клиенту вместо вызова сервиса

### Общие моменты реализации

* Наследуемся от **ZuulFilter**, перекрываем методы **filterType(), filterOrder(), shouldFilter()** и наконец самый главный **run()** – собственно выполнение фильтрации
* За счет **RequestContext.getCurrentContext()** (это не стандартный Spring **RequestContext,** а собственный Zuul с дополнительными свойствами) имеем возможность доступа к данным и выполнения необходимых операций
  + **getRequest() / getResponse()**
  + **setResponseBody()**
  + **addZuulRequestHeader() / addZuulResponseHeader()** – нам нельзя изменять исходные заголовки, поэтому мы добавляем заголовки в эту коллекцию, а Zuul перед вызовом сервера / возвратом клиенту их правильно складывает.
    - Соответственно, если в фильтре ищем какой-то заголовок, то его надо искать в двух местах
    - Аналогичные get-методы
  + **setRequestXxxx() / setResponseXxxx()** – подмена запроса / ответа, целиком или по частям – тело, параметры запроса и так далее
  + **setSendZuulResponse()** – ответ должен дальше быть непосредственно отправлен клиенту
  + **setRouteHost()** – явная маршрутизация запроса к определенному хосту
* По умолчанию Zuul добавляет заголовки **X-Forwarded-Host / X-Forwarded-Prefix**, поэтому если сервисы используют для построения URL стандартные утилиты Spring, то получающией ссылки будут нормально указывать на фасад, вместо реального расположения сервиса
* По умолчанию маршрутизация stickness
  + в принципе можно сделать фильтр, который будет подмешивать в ответ cookie с наименованием экземпляра сервиса и пока экземпляр здоров стараться отправлять полученные запросы с меткой по соответствующиему адресу.
  + Критически важно, когда используется Hibernate кэш второго уровня и не хочется, чтобы все сервера кластера хранили все метаданные / основные данные всех пользователей
* **@Autowired RouteLocator** дает возможность программного доступа к текущей конфигурации роутинга
* Доступна вся функциональность, предоставляемая **DiscoveryClient**
* За счет **@EventListener** методов можно получить доступ к событиям
  + **HearbeatEvent** – добавление / удаление экземпляра сервиса. Например, можно кэшировать **Map<serviceId, List<ServiceInstance>>** и при получении данного события обновлять через **discoveryClient.getServices()**
  + **RoutesRefreshedEvent** – изменение схемы маршрутизации
* Весь код фильтра хорошо оборачивать **catch(Exception ex)** с **ReflectionUtils.rethrowRuntimeException(ex)** внутриц

### Распространение по системе correlationId

* Основное назначение: по единому **correlationId**, присваемому запросу, поступившему от клиента, собрать все данные из логов все сервисов, которые, так или иначе, приняли участие в его обработке
  + Аналогично распространяем по системе **userId, organizationId, securityToken, transactionId** и тому подобные глобальные свойства запросов
* Будем передавать по системе в виде некоторого HTTP заголовка
* **PRE фильтр**
  + Получает запрос за счет **RequestContext.getCurrentContext().getRequest()**
  + Исследует заголовки запроса (а также **getZuulRequestheaders()** заголовки) на предмет наличия нужного.
  + Если не найден (то есть это внешний запрос), то добавляем в исходящий запрос к сервису за счет **addZuulRequestheader().** Иначе ничего не делаем и его не меняем (это запрос от сервиса, который уже пороэден внешним запросом клиента).
* **Сервис**
  + Данную функциональность вполне допустимо выполнять в качестве common library – она достаточно трудоемка, но в то же время ее реализация достаточно стабильна. Обычно применяет шаблон **UserContext + UserContextHolder** со статическими методами для установки / получения контекста для текущего потока
  + Получив запрос, за счет стандартного HTTP фильтра выделяет данные, оформляет в виде **UserContext** и кладет в **ThreadLocal**
  + Использует при логировании (аутентификации / авторизации и так далее). Можно ручками, а можно добавить в **Spring Mapped Diagnostic Context (MDC)**
  + Подмешивает во все исходящие HTTP запросы
    - При использовании thread режима Hystrix правильно перекладывает в потоки пула, реально выполняющие запроса
    - Расширяет **ClientHttpRequestIntercepter** и перед выполнением запроса, получив доступ к **Request** объекту, добавляет туда считанный из **ThreadLocal** заголовок
    - В процессе формирования **@LoadBalanced** бина **RestTemplate**, использует **template.getIntercepters()** чтобы добавить данный интерцептор в конфигурацию. В приложении как правило имеется только один такой бин, совместно используемый всеми компонентами, осуществляющими удаленные вызовы
* **POST фильтр**
  + Считываем значение из **getZuulRequestheaders() / getRequest().getheaders()** и пересаживаем обратно в ответ клиенту

### Фильтр динамической маршрутизации

* Заготовку исходного кода лучше брать из **SimpleHostRoutingFilter**, потому что особой функциональной поддержки zuul на этом месте не предоставляет, а логика обработки достаточно сложна
* Единственный набор утилит – **ProxyRequestHelper**, при помощи него
  + Выделяем из исходного запроса uri, заголовки и параметры (http метод и тело в виде InputStream получаем непосредственно из запроса)
  + Пересаживаем результат из отдельно выполняемого сихронного запроса к проксируемому сервису
* Чтобы вызвать проксируемый сервис используем Apache HttpClient безо всяких оберток

### Фильтр ограничениия нагрузки

* Возможные алгоритмы
  + Маркерной корзины (token bucket) – корзина фиксированной емкости, в которую маркеры (блоки байт / пакеты фиксированного размера) добавляются с фиксированной скоростью
  + Протекающего ведра (leaky bucket) – протекающее ведро будет переполнено, если средняя скорость, с которой наливают воду, привысит норму протечки или в ведро сразу нальют объем, превышащий его емкость
* Широко используемая реализация – Guava RateLimiter (<https://www.baeldung.com/guava-rate-limiter>)
* Фильтр типа PRE, если не получилось пропихнуть очередной запрос, то
  + В response устанавливаем статус **HttpStatus.TOO\_MANY REQUESTS**
  + **RequestContext.getCurrentContext().setSendZuulResponse(false)**
  + throw new **ZuulException()** с данными из HttpStatus.TOO\_MANY REQUESTS

## Неизбежные проблемы

* Большее время ответа по сравнению с прямым вызовом сервиса
* Опасность реализовать за счет него слишком много логики. Ее надо выносить отдельно и делать фильтры всего лишь способом подключения.
* Должен быть очень устойчивым и масштабируем, в противном случае неизбежно становится единой точкой отказа и бутылочным горлом системы.
  + Причем все проблемы будут выявляться именно тут, а для разработчиков это поначалу будет черный ящик с не очень понятным поведением.
  + С другой стороны эти сервера практически ничего не делают, поэтому обеспечить их жизнеспособность гораздо проще
  + Поэтому весь код внутри stateless и легковесный по-максимому, никаких вызовов реляционных БД и прочих тяжелых операций
* Кластеризация
  + На входе системы кластер gateway серверов, перед ним балансир – железный / HAProxy /NetScaler / NGINX (наверное, самый распространенный вариант сегодня)
  + Более хитрый вариант:
    - Все экземпляры Zuul регистрируем в Eureka / Consul сервере и заставляем клиента его обнаруживать динамически
    - Для браузерных приложений клиентк Eureka есть, так что самый распространенный вариант использования закрывается
    - Минусы:
      * для каких-то клиентов (например, для PL/SQL) клиентов к Eureka просто нет.
      * Стандарты безопасности организации могут запрещать использование client side балансиров
* Тюнинг gateway серверов под повышенную нагрузку на сетевом уровне (см. приложение).

## Zuul-2

* все то же самое, но плюс асинхронно в неблокирующей манере на основе Netty
* В результате применения асинхронного подхода
  + Снизилось использование CPU, значительно увеличилось число удерживаемых входящих соединений
  + Появилась возможность широкого использования push / bi-derectional взаимодействия с клиентами
  + За счет этого появилась возможность избавиться от клиенских chatty проктоколов, когда клиенты раз за разом переспрашивают сервер на предмет поступивших для них сообщений. Такие протоколы обычно составляют изрядную часть клиентского трафика, например, для мобильных систем.
  + Система легче переживает кризисы и “retry storms”
* Одако все это только для систем с малой вычислительной нагрузкой, основная работа которых состоит в подготовке запроса, например, в базу данных и ожидании его выполнения
  + Пример подобной системы – запись потока входящих сообщений в лог. С асинхронным подходом можно добиться значительного снижения расхода СPU и увеличения пропускной возможности системы
  + Для в основном вычислительных систем, например, аггрегация / сжатие данных, шифрование / работа с электронными подписями, особого выигрыша получить не удастся.
* Общий принцип – чем меньше система сама по себе делает, тем больший выигрыш она получает от асинхронной архитектуры. Zuul-2 в этом отношении практически идеальная система. Вопрос о переводе реализации API на асинхронный подход это вопрос баланса сложности реализации и ожидаемого выигрыша.

# Безопасность

* Чем плохо передавать логин / пароль в каждом запросе
  + Чем чаще передаем, тем больше риск утечки
  + При утечке одного запроса к одному сервису сразу надо полностью запрещать все запросы пользователя ко всем сервисам и менять этот пароль всюду
  + Если скомпрометирован сам сервис, то попадаем на замену всех паролей всех пользователей
  + Люди задействуют одни и те же пароли в разных системах
  + Смена пароля это дорого и больно. Люди склоны его слегка варьировать и противостоять этому трудно
* Каким образом в одной системе избежать необходимости:
  + Для клиента системы – предоставлять credentials при каждом вызове
  + Для сервисов – повторно реализовывать хранение данных логику идентификации / аутентификации, а также выполнять последующие вызовы других сервисов от имени, обратившегося пользователя
* Каким образом обеспечить безопасность при взаимодействии изначально разных систем. Обычно, когда третья сторона должна предоставить пользователю системы некоторый сервис, а для этого ей нужен авторизованный доступ к услугам системы в контексте полномочий пользователя
  + Пользователь один, а данные его разные и централизованное хранение их затруднено или невозможно в принципе. И ни одна система для другой не является главной, то есть master системой, они равноправны.
  + Пример: сервис продажи еды и электронный бумажник, которым пользователь может воспользоваться для оплаты заказа. Данные пользователя в обеих системах совершенно различны, а взаимодействие систем «принять ваучер в качестве оплаты» должно осуществляться неотрекаемо и защищенно для обеих сторон.

## Введение (более подробно в приложении)

* **Oauth2 (Open Authority)** это token based (рассчитан на использование Bearer токена в виде SAML2 или JWT) открытый расширяемый протокол авторизации (отвечает на вопрос «что разрешено пользователю»), позволяющий
  + сервисам делегировать задачу авторизации пользователей на специальный инфраструктурный сервис, которым обе стороны доверяют.
  + Отвязать ЖЦ записи пользователя (логин / пароль) от ЖЦ токена безопасности (утверждения о том, что пользователь осуществил процесс аутентификации)
  + Избежать блокировки учетной записи пользователя при компрометации клиента
  + Сокращают время / scope открытой передачи чувствительных данных пользователя, позволяя осуществлять их обработку в одном, заведомо безопасном компоненте
  + Полностью разделить задачи авторизации и аутентификации
    - Первая теперь будет происходить единым, стандартным для всех компонентов системы образом
    - форму и конкретную реализацию осуществления второй можно выбирать независимо и даже вообще варьировать от пользователя к пользователю.
* OAuth2 определяет следующих участников процесса
  + **Resource Owner** – организация, отвещающая за реализацию сервиса, отвечает за регистрацию **Resource Server на Authorization Server**
  + **Client** – приложение, которому нужен доступ к ресурсу. Это может быть либо мобильное приложение, либо сервер, обслуживающий веб-приложение.
  + **Resource Server** – сервис, предоставляющий публичное API, посредством которого UserAgent можно оперировать ресурсом. В нашем случае это любой защищаемый сервис
  + **Authorization Server – IdP (identification provider).** Соответственно, чтобы это все работало, все остальные участники процесса должны ему доверять.
  + **UserAgent** – приложение, которое использует конечный пользователь.
    - Что-то, что посылает HTTPS запросы и может обрабатывать редиректы, возвращаемые сервером авторизации или клиентом
    - Это может быть либо браузер, в случае веб-приложения, либо компонента внутри Client, в случае, например, мобильного приложения
* Обязательный предварительный этап регистрации Resource Server на IdP. В результате обоим сторонам становятся известными
  + **callback** – куда возвращать аутентифицированного клиента, опциональный параметр, так как в принципе клиент и сам знает куда ему потом идти
  + **client\_id и client\_secret** – обязательные параметры.
    - В данном случае секрет не используется при подписи запросов. Это практически логин / пароль клиента на провайдера при сервер-сервер взаимодействии
    - Секрет может быть как симметричным, так и ассиметричным, зависит от настроек обоих сторон. Однако второй вариант не стандартизован и реализуется специальным образом.
* **Resource Owner Password Credemtials Grant** – основной вариант использования в случае REST
  + Совсем просто: клиент сам запрашивает у пользователя логин+пароль, идет с ним на IdP и обменивает на токен за счет GET grant\_type=password & client\_id & client\_secret & username & password
  + После чего вкладывает токен в каждый запрос к Resourse Server
    - Resourse Server (он stateless) идет с ним на IdP и получает всю необходимую информацию о пользователе, его правах, длинне сессии и так далее
* SAML2 токен в принципе это подписанный XML с некоторым набором информации, разнящимся от IdP к IdP
  + в том числе Subject confirmation method (тип токена) - каким образом подтверждается, что предъявитель запроса сего действительно Subject
  + В случае REST это Bearer – никаким, то есть, кто предъявил валидный токен, тот и будет выступать в качестве Subject. Соответственно, если токен украден и вставлен в любой чужой запрос, то RS этого не заметит
  + В случае Oauth2 это даже не XML документ, а просто уникальный идентификатор, предъявляемый IdP для получения информации о сессии пользователя, его свойствах, ролях и так далее. Тип, то есть Bearer, передается как префикс в HTTP заголовке Authorization
* JWT токен это BASE64 кодированный подписанный JSON, в котором уже есть вся необходимая информация
  + То есть обращаться каждый раз на IdP за его проверкой и получении данных о пользователе не надо. Таким образом значительно снижаем нагрузку на IdP, а также уменьшаем время обработки запроса пользователя к сервису.
  + Однако возникает проблема - такой самодостаточный токен при этом невозможно отозвать – он валиден до даты, указанной в составе подписанных свойств. Что крайне неудобно.
    - Соотвественно применяют комбинацию двух токенов: коротко живущий, для обращения к Resource Server, и длинно живущий, чтобы чтобы автоматически получить короткоживущий у IdP без прохождения процедуры идентификации
    - В момент получения очереднго короткоживущего токена можно обновить информацию, которую будет в дальнейшем использоваться Resource Server
    - Неизбежный компромисс: чем быстрее в случае чего мы хотим иметь возможность отключить однажды авторизовывашегося пользователя, тем чаще нам надо обновлять короткоживущий токен, тем больше нагрузка на сервисы и IdP
  + Расширяемый, при реализации IdP можно добавить в него любое количество кастомных свойств, которые также будут подписаны и переданы на клиента

## Архитектура Spring Security

* Процесс аутентификации (получение достоверных данных о пользователе) полностью отделен от процесса авториации (на основении достоверных данных вынесение решения, имеет ли пользователь доступ к защищаемому ресурсу)
* Аутентификация выполняется реализацией **AuthenticationManager**, которая получает на вход аргумент **Authentication** и возвращает его в случае успеха со свойством **authenticated=true**
  + В случае неуспеха – **AuthenticationException / null**
  + На даном уровне ничего не известно о природе запроса (HTTP / PRC / локальный метод) и о данных, предоставленных пользователем (логин / пароль, сертификаты, токены)
* **ProviderManager** – реализация **AuthenticationManager,** хранящая цепочку менеджеров и работающая с одним видом **Authentication**
  + Сам **ProviderManager** может содержать код авторизации, срабатывающий в ситуации, когда не один менеджер цепочки не дал ответа
  + Соответственно имеем иерархическую структуру менеджеров, делегирующих ответственность друг другу, но могущих работать и самостоятельно.
  + Разные менеджеры могут работать с разными по природе источниками идентификационной информации (БД, LDAP)
    - Каждый источник абстрагируется за счет AuthenticationProvider. Например, в случае БД это **DaoAuthenticationProvider**, который ищет экземпляр **UserDetailsService** для загрузки данных пользователей
* Решение о доступе идентифицированного пользователя к конкретному ресурсу выполняется экземпляром **AccessDecisionManager**, который работает за счет цепочки ответственности на основе **DecisionVoter**
  + Данные для принятия решения: **Authentication**, контекст принятия решения, коллекция **ConfigAttribute** атрибутов ресурса
  + Разные **AccessDecisionManager**, декларируют возможность работать с разными контекстами / аттрибутами

## Oauth2 + SAML

### Сервер авторизации

* Зависимости **spring-cloud-security + spring-cloud-oauth2**
* На конфигурации **@EnableAuthorizationServer** (сама процедура выдачи токена авториации) + **@EnableResourceServer** ( процедура выдачи информации о пользователе по токену – это такой же защищенный ресурс, как и обрабатываемые остальными сервисами)

#### Конфигурируем собственно AuthorizationServer

* Выполняется в классе конфигурации, наследуемом от **AuthorizationServerConfigurationAdapter** двух **@Autowired** **configure()** методах, принимающих разные параметры
* За счет параметра **ClientDetailsServiceConfigure** можно сконфигурировать, откуда брать информацию о защищаемых сервисах . Фактически это регистрация resource server на IdP
  + варианты
    - **inMemory()** – определяем прямо на месте и дальше храним в памяти
    - **jdbc()** – указываем как загрузить из ДБ
    - **withClientDetails(ClientDetailsService)** – предоставляем собственную реализацию сервиса
  + Для каждого подключаемого сервсиса предоставляем как минимум следующие данные:
    - **clientId / secret** – для авторизации сервиса на IdP во время получения данных о пользователе.
    - **authorizedGrantTypes** – каким сценарии Oauth2 поддерживаем, в нашем случае это **“refresh\_token”, “password”, “client\_credentials”**
    - **scopes** – типы клиентов сервиса, например, “webclient”, “mobileclient”. В принципе это ограничить разные типы пользователей в доступе, но на деле, так как они сами предоставляют информацию о своем типе, это не слишком полезно
    - URL на который будет перенаправлен запрос клиента в случае положительной аутентификации. Лучше всего использовать **service discovery + discovery client**
  + В принципе можно реализовать автоматическое подключение отдельных сервисов приложения через REST в процессе стартовой регистрации. Это внутренний защищенный обмен и достаточно безопасно
* За счет параметра **AuthorizationServerEndpointConfigurer** подключаем его к стандартной Spring авторизации за счет установки свойств **authenticationManeger** и **userDetailsService** (это уже не про клиентов, а про пользователей, которые будут авторизоваться, чтобы затем обратиться к клиентам)

#### Конфигурируем собственно Spring Security

* Наследуем конфигурацию от **WebSecurityConfigurerAdapter**
* **@Autowired** **void configure(AuthenticationManagerBuilder auth)** – строим хранилище данных для аутентификации пользователе
  + В простейшем вариант **auth.inMemoryAuthentication(),** затем цепочка методов набивающих хранилище информацией о пользователях
  + Более продвинутые варианты:
    - **jdbcAuthentication()** – работа со встроенным хранилищем на основе БД, затем аналогично цепочка методов, чтобы сконфигурировать работу с таблицами / колонками, шифрование пароля для хранения в БД и так далее
    - **ldapAuthentication()**
    - **userDetailsService(T userDetailsService)** - использование собственного локального сервиса работы с пользователями
* **@Autowired** **void configure(HttpSecurity http)** – настройка доступа к различным URL в зависимости от полномочий пользователя
  + в принципе нам это не нужно, так как бизнес-точек доступа у нас тут просто нет, только стандартные для авторизации и получения доступа к данным пользователю по токену
  + все будет работать по умолчанию
* за счет **@Bean** публикуем бины типа **AuthenticationManager** и **UserDetailsService**, получаемые как вызов super методов. Это нужно для конфигурирования **AuthorizationServerEndpointConfigurer** в предыдущем разделе

#### Контроллер выдачи информации о пользователе

* Данный метод будет вызываться при каждом вызове любого метода защищаемого сервиса (Resource Server)
* Метод **/user,** принимающий параметр **Oauth2Authentication** и возвращающий **Map<String, Object>**
  + Под ковром происходит валидация **access token** и получение данных о пользователе
  + В возвращаемый объект перекладываем информацию из **Oauth2Authentication** с ключами **«user» и «authorities».** Можем добавить любую свою дополнительную информацию

### Схема авторизации пользователя

* Пользователь обращается на сервер авторизации **/auth/oauth/token** как POST и предъявляет **grant\_type=password, scope, username, password**. В отличии от остальных обращений это не JSON, а **form data** передаваемые данные
* В ответ получает JSON со следующими данными: **access\_token, token\_type=bearer, refresh\_token, expires\_in (по умолчанию 12 часов), scope**
* Теперь любой участник (то есть как сам пользователь, так и любой сервис, получивший каким-либо способом **Access Token**) может обратиться на **/auth/user** с GET запросом и HTTP заголовком **Authorization** установленным в **Bearer:<access\_token>.** В ответ он получит JSON документ с описанием пользователя, сессией, правами и так далее

### Защита сервисов (Resource Servers)

* Общий принцип: любой запрос к сервису должен содержать токен доступа. Выделив его идем к IdP, валидируем токен, получаем перечень ролей и из этого исходим
* Зависимости **spring-cloud-security + spring-cloud-oauth2**
* Свойство конфигурации **security.auth2.resource.userInfoUrl** направляем на сервер IdP в контроллер получения информации о клиенте, в URL естественно используем идентификатор сервиса вместо реального хоста
* На классе конфигурации **@EnableResourceServer**

#### Безопасность на уровне HTTP

* Какие именно ресурсы и как защищаем, определяется в конфигурации, расширяющей **ResourceServerConfigurerAdapter в configure()** методах, принимающих следующие параметры
  + **HttpSecurity** – какие именно URL и как защищаем (цепочка вызовов от аргумента, чем-то напоминает **switch-case-default** конструкцию)
    - Определяем тип запроса - **http.authorizeRequest(),**  соответственно если пользователь не приложил соответствующий Authorization заголовок, то он немедленно получает 401 “unauthorized”
      * Затем от него шаблон на URL запроса - **antMatchers(“\*\*/user/edit”)**
      * И требуемые для авторизации к нему роли доступа **hasRole() / hasAnyRole()**, если их нет даже идентифицированный пользователь получает 401 “acess denied” – код тот же, но ошибка другая
      * Таких групп определений (отдельный “case”) может быть много
    - в заключении все остальные запросы **anyRequest(),** “default” case и результат «авторизовать» - **authenticated()**
    - Также отключаем сессию, так как она нам не нужна **httpsessionManagement().sessionCreationPolicy( SessionCreationPolicy.STATELESS )**

#### Безопасность на уровне методов

* Определяем **@EnableGlobalMethodSecurity(securedEnabled=true)** на конфигурации **ResourceServerConfigurerAdapter**
* Методы аннотируем **@Secured(“ROLE\_NAME”).**
  + Данная аннотация доступна со Spring 2.0
  + Используется старый синтаксис, то есть не только роли, но и, например, константы типа **IS\_AUTHENTICATED\_FULLY**, конструкции типа **hasRole(…),** SpEL выражения и так далее
* Если в аннотаации конфигурации применить **jsr250Enabled=true**, то Методы аннотируем **@RolesAllowed({“ROLE\_NAME”})**
  + Здесь могут применяться ТОЛЬКО роли
  + Вариант более стандартный, но менее продвинутый
* За счет **prePostEnabled=true** можно включить ACL безопасность дополнительно к разграничению по ролям
  + Все эти аннотации конфигурируются SpEL выражениями, в которых могут использоваться
    - Уже знакомыми выражениями - **hasRole('ADMIN') AND hasRole('DBA')**
    - Переменные контекста, например **authentication** – авторизованный пользователь
    - аргументы вызова, например **#user.userName = authentication.name** –только для пользователя с тем же именем, что и авторизованный
    - Наименования бинов с методами: **@A.maySayHelloTo(principal, #name)**
    - Могут использоваться как на методах, так и на классах / интерфейсах
  + **@PreAuthorize** - для проверки полномочий пользователя перед входом в метод
    - Оперирует данными пользователя и параметрами запроса
    - Если выражение возвращает false,то метод не вызывается
  + **@PostAuthorize -** для проверки полномочий пользователя после выполнения метода.
    - Может дополнительно оперировать **returnObject** – данными, возвращенными методом, например, **returnObject.type == authentication.name**
    - Если выражение возвращает false, то результат не возвращается пользователю
  + **@PostFIlter и @PreFilter** – для фильтрации перечня объектов на основании правила, задаваемого SpEL выражением
    - **Pre** – фильтрует перечень, передаваемый в качестве аргумента, **Post** – соответственно возвращаемый методом
    - Для написания предиката дополнительно ко всем ранее описанным переменным используется **filterObject**, например: filte**rObject.assignee == authentication.name**
    - Проблема с **@PreFIlter** – если запрос может вернуть много данных, то он сработает как модификатор **Eager** и вычитает их все, даже если методу реально нужны только несколько первых (выдача страниц)
* Ну и конечно можно получить доступ к сырой информации об авторизованном пользователе и работать с ней программно
  + **@Autowired** или аргумент метода контролллера **Authentication authentication**
  + **SecurityContextHolder.getContext().getAuthentication()**
  + В контроллере **request.getUserPrincipal()**

#### Распространение контекста безопасности на вложенные вызовы

* Сервис, к которому обратился авторизованный пользователь, в свою очередь делает вызовы других сервисов, то надо автоматически пересаживать Authorization заголовок во все исходящие HTTP запросы
* Настройка zuul
  + Как минимум **zuul.sensitiveHeaders: Cookie, Set-Cookie,** чтобы разрешить передачу **Authorization** заголовка в проксируемые сервисы
  + Можно **@EnableOUuth2Sso** на конфигурации Zuul, тогда он также будет пропускать **Authorization** заголовок внутрь и плюс к этому авторизовать все входящие запросы
* Настройка сервиса
  + Зависимость **security-autoconfiguration** с **TokenRelayAutoConfiguration** внутри, можно посмотреть как это реализовано
  + Ручками реализовывать фильтр для работы с **Authorization** заголовком не надо – это делается за счет **@EnableOAuth2Cleint**
    - вместо **RestTemplate** необходимо самому опубликовать и использовать **OAuth2RestTemplate**, который возьмет на себя всю работу.
    - **RestTemplate** остается доступным, им можно делать запросы, не требующие распространения контекта безопасности
    - **Feign** клиенты также поддерживают распространение
    - В методах контроллера можно использовать параметры **Principal**

## OAuth2 + JWT

* Spring Cloud из коробки поддерживает использование JWT в качестве Oauth2 токена, но необходимо специально сконфигурировать его использование на IdP и на сервисах
* Основное достоинство: JWT токен самодостаточен - можно проверить и извлечь из него необходимую информацию прямо на сервисе без обращения к IdP.
  + Таким образом, значительно снижаем нагрузку на IdP, а также уменьшаем время обработки запроса пользователя к сервису.
  + Однако получаем проблему в виде необходимости иметь два токена: долго и коротко живущие

### Сервер авторизации

* Зависимость **spring-security-jwt**
* Конфигурация с определением необходимых бинов:
  + **TokenStore** на основе **JwtTokenStore**
  + **@Primary DefaultTokenService,** конфигурируемый **tokenStore** и **supportRefreshToken**=true. Используеся для чтения и формирования токена
  + **JwtAccessTokenConverter** - работает как транслятор между JWT / OAuth2 токенами
    - **singingKey = serviceConfig.getJwtSigningKey()** или просто строковая константа
      * по умолчанию используется симметричный ключ, который должен быть определен как свойство signing.key одинаково на IdP и на всех сервисах.
      * Разумеется, лучше использовать ассиметричные ключи за счет **converter.setKeyPair()** на сервере авторизации и converter.setVerifierKey() на сервисе. Детальная инструкция на <https://www.baeldung.com/spring-security-oauth-jwt>
  + **JWTTokenEnhancer** – наша собственная реализация **TokenEnhancer**, позволяет кастомизировать токен путем добавления собственных свойств, опциональный бин
* Подключение в конфигурацию IdP на основе **AuthorizationServerConfigurationAdapter**
  + В методе **configure(AuthorizationServerEndpointConfigurer endpoints)**
    - Создаем **TokenEnhancerChain** на основе **JWTTokenEnhancer** и **JwtAccessTokenConverter** (порядок важен)
    - Конфигурируем **endpoints** дополнительно за счет **tokenStore(JwtTokenStore) и accessTokenConverter(JwtAccessTokenConverter)**
* Если теперь авторизуемся, то оба токена будут гораздо длиннее.
  + Чтобы их декодировать и проверить целостность можно воспользоваться jsonwebtoken.io, которые преобразуют Base64 строку в JSON документ
* Расширение содержания токена
  + Собственная реализация **TokenEnhancer**
  + В методе **Oauth2AccessToken enhance(Oauth2AccessToken)** используем **setAdditionalInformation()** чтобы дополнить токен **Map<String, Object>** кастомных данных
  + Токены подписаны, но НЕ зашифрованы. Поэтому, расширяя их содержание, не надо вкладывать туда чувствительную информацию

### Сервисы

* Зависимость **spring-security-jwt**
* Аналогичная конфигурация с определением бинов **TokenStore, @Primary DefaultTokenService и JwtAccessTokenConverter**
* Проблема: **Oauth2RestTemplate** ничего про JWT токен не знает, поэтому придется
  + Определить входной фильтр для разбора HTTP запроса и сохранения данных в **ThreadLocal**
  + сконфигурировать собственный бин **RestTemplate** с интерцептором, коорый пересаживает значения JWT заголовков из **ThreadLocal** в создаваемые запрос
* **Zuul** также может работать с JWT токенами, но для этого ему не нужна вся эта сложная обвязка
  + Зависимость **io.jsonwebtoken.jjwt**
  + Используем **Jwts.parser()** для разбора Authorization заголовка из которого предварительно обрали **Bearer** префикс

## Общие рекомендации

* Требовать наличие заголовка **Content-Length,** если нет – **411 Length Required,** проверять соответствие реальному размеру, отвергать запросы с **Transfer-Encoding: chunked** вообще
* Не выпускать наружу тексты исключений, позволяющих проанализировать устройство кода
* Вместо различных ошибок о недопустимых действиях по отношению к ресурсу всегда выдавать только 404 – защита против сканирования, мы скрываем само существование ресурса. Увы, с авторизацией так не получится.
* Генерировать URI случайного вида, например, используя UUID, чтобы затруднить их воспроизведение сообразно некоторому шаблону
* Валидировать аргументы, особенно максимальные размеры полей
* Всегда использовать при роботе с БД **PreparedStatement** и аналоги
* Меньше кода – меньше забытых фич, которые могут послужить объектом атаки
* Не запускать из-под root или других пользователей, имеющих существенные права
* Очищать JSON от возможных внедрений при помощи **OWASP JSON Sanitizer**
* Логи тоже утекают, в логах не должно быть **PersonallyIdentifiableInformation (PII)**
* Используем HTTPS/SSL при всех взаимодействиях, как снаружи внутрь системы, так и внутри системы при взаимодействии сервисов между собой
  + Да, это осложняет отладку / конфигурирование и создает дополнительную нагрузку, но меньше риск что-то забыть, а также позволить маленькой уязвимости расшириться
  + Между собой внутри системы можно даже ввести взаимную аутентификацию на основе сертификатов
* Все обращения к сервисам должно производиться через единый входной шлюз, гарантирующий обязательное и единое применение политик безопасности
  + Все endpoint сервисов должны вызываться только через шлюз, соответственно сервисы должны принимать запросы только от шлюза
  + Шлюз выставляет наружу только endpoints необходмые клиентам. Остальные endpoints, посредством которого сервисы вызываютдруг друга не должны быть доступны снаружи
  + Часто, особенно в параноидальных случаях типа PCI DSS, организуем два шлюза с двумя IdP
    - Один для сервисов, предоставляющих внешнее API, предназначенное для вызова клиентами. Только усеченный набор данных / операций, необходимых для клиентских workflow. Работают как фасады для остальных сервисов
    - Другой (private zone) для сервисов, вызываемых из первой (public) зоны. Все сервисы из публичной зоны должны для вызова дополнительно авторизоваться на IdP приватной зоны. Именно тут должны храниться пользовательские данные и реализовываться подробные операции по их обработке. В принципе, тут уже обязательное применение HTTPS может быть излишним, так как оно не бесплатное
* Уменьшаем пространство для атак на серверах
  + Опускаем все лишние сервисы
  + Закрываем все лишние входные и выходные порты, не используемые нашими сервисами и общесистемной инфраструктурой
  + Black list серверов от которых принимаем запросы

# Интеграция и обмен сообщениями

* **Event Driven Architecture (EDA)** – обмен сообщениями, содержащими изменение в состоянии данных. Позволяет строить приложения из слабо связанных между собой компонент, каждый из которых с одной стороны публикует события об изменении собственных данных, а с другой, реагирует на аналогичные события других компонент.
* Основные кейсы:
  + Уведомление о событии: одна система уведомляет остальные об изменениях в предметной области. Сообщение отправляется в пространство, исходная система не ожидает, что его кто-то обработает и не рассчитывает на получение ответа. Система получатель может сделать на основании уведомления все что угодно, например, запросить у исходной системы свежую копию измененых данных.
  + Передача полного состояния за счет события: в данном случае все необходимые для получателя данные находятся в самом сообщении. С одной стороны хорошо – получателю не надо обращаться к владельцу данных, в другой, если два сообщения поменялись местами / последнее потерялось и так далее, то получатель сохранит у себя старые данные
  + Передача только измененного состояния за счет события: сообщение меньше, обращения к источнику данных также не требуется, но риск получить проблемы еще больше. Соответственно надо на приемнике контролировать порядок и идемпотентность применения
  + Журнал событий: источник сохраняет у себя все эмитированные события в некотором логе, который может быть считан другой системой для воспроизведения у себя полной истории хранилища данных
* Наиболее частый кейс применения – кэширование
* Достоинства EDA архитектуры
  + **Loose coupling** – любой синхронный HTTP запрос это жесткая зависимость между сервисами. В данном случае сервисы ничего друг о друге не знают
  + **Durability** – сообщение может быть доставлено, даже если получатель временно недоступен
  + **Scalability** – отправитель не ждет, пока получатель подтвердит получение сообщений. Если сообщений слишком много, то можно поставить несколько получаетелей, а на каждом из них гибко управлять числом потоков слушателя в зависимости от доступного железа. Размер очереди дает очевидную оценку производительности потребителя. Очередб позволяет выравнять пики нагрузки
  + **Flexibility** – отправитель ничего не знает о том, кто и как потребляет сообщения, что дает возможность легко наращивать функционал
* Недостатки – возрастание сложности
  + часть времени хранилища находятся в несогласованном состоянии. В общем-то это не большая проблема, даже в случае очень важных данных.
    - Например, при редактировании какой-нибудь кредитной политики как правило можно принебречь временем несогласованности в несколько минут: клерк может нажать кнопку «применить» в некотором диапазоне времени и требовать точного соответствия с временем поступления кредитной заявки достаточно глупо.
    - От привычных ACID транзакций приходится переходить к BASE (Basically Available, Soft-state, Eventual consistency – доступность в большинстве случаев; неустойчивое состояние; согласованность в конечном счете), что трудно для понимания и непривычно
  + становится трудно понять очередность выполнения операций. Тем более, что из-за недетерминированого порядка потребления сообщений (а также потерянных, дважды доставленных сообщений, принятых с ошибкой) она может варьироваться.
  + При потреблении сообщения теряется контекст исходной операции – трудно понять чем именно было вызвано изменение данных и вообще в контексте какого глобального бизнес процесса производится выполнение. Поэтому correlationID должен быть обязательным компонентом сообщения. Увы, это помогает при разборе логов, но слабо при попытке локальной отладки какого-то сервиса, тем более, на активно работающей системе
  + Проблемы с конкурентным обновлением данных, потерянными обновлениями, обновлениями доставленными в неправильном порядке и доставленными несколько раз. Особенно, когда сообщения несут только изменения, а не все данные, а обновляем мы реляционную СУБД в активно применяемыми FK
* Один из существенных вопросов – что помещаем в сообщение?
  + Обязательно **correlationId** и версию протокола
  + Можно конечно сразу запихнуть измененные данные или все данные
    - тогда получателю не придется самому идти к источнику данных.
      * А если все данные, то его копия не сможет оказаться в неконсистентном состоянии. Правда, может в устаревшем, если сообщения поменяются местами или задвоятся.
      * Кроме того, нагрузка на источник данных снизится, ему не придется отвечать на множество вопросов всех потребителей.
      * Хорошо, когда всех потребителей интересуют все данные в сообщении и все сообщения
    - однако это слишком сильно свяжет между собой источник и потребителей данных. И если структура данных будет меняться, то потребуются изменения во всех местах и причем одновременно
  + Поэтому лучше только ИД и выполненную операцию и пусть потребитель сам сходит к источнику за обновлением сообразно той версии протокола, которая ему доступна.
    - Нагрузка на источник данных конечно будет выше
    - Но зато потребители получат последнюю версию, даже если какие-то сообщения потерялись или поменялись местами
    - Если данные интересуют не всех, то меньше объем передаваемых данных
  + Еще вариант: данные помещаем в некоторое централизованное хранилище, а в сообщении только ссылку на запись
    - Достоинства: гарантированная сохранность. Если среду распространения перезапускаем, то неполученные сообщения можно перевыслать. Даровой лог работы. Единстыенный вариант, когда payload очень большой
    - Недостатки: узкое место (хотя и легко масштабируемое, так как единственная операция это получение записи по ПК)
* Одно из наиболее частых возражений против введения асинхронных сообщений в проект – порядок передачи сообщений не гарантируется
  + Есть решения гарантирующие порядок передачи – kafka
  + TCP на самом деле тоже не гарантирует порядок, просто проявляется это при высоких наргузках и сложной конфигурации сети
  + В большинстве случае это принципиально не возможно:
    - При преобразовании асинхронного вызова в синхронный
    - Если документ асинхронно переводится последовательно через несколько состояний и каждый переход
  + Основной кейс, когда это действительно часто всречается – передача большого объема данных, разбитого на отдельные порции. Для этого применяется специальный шаблон «цепочка сообщений»
* В реальности как правило не удается перевести систему полностью на асинхронное взаимодействие. Например: размещение заказа приходится выполнять синхронно, чтобы пользователь затем его увидел в своем перечне.
  + А вот дальнейшею его обработку можно выполнять уже асинхронно, периодически высылая пользователю уведомления о изменении его статуса / выявленных ошибках / дополненной информации и так далее. Соответственно, первичная операция создания должна выполняться максимально быстро и не включать в себя дорогих функций
  + Как правило в микросервисных системах UI отделен от сервиса процессинга заказа. Соответственно можно завести в UI хранилище данных о заказе с минимальным набором свойств (в идеале только те, которые показываем или по которым ищем), а сам заказ хранить в виде документа, асинхронно высылаемого в сервис процессинга

## Spring integration

* В принципе любая интеграция это обмен сообщениями между двумя точками. Но, если это реализовывать в лоб на основе очередей, то получится архитектура типа «спагетти», где сложно понять кто, куда и в какой последовательность отправляет сообщения. Вместо этого предлагается описывать интеграционные потоки.
* Другая альтернатива: множество отдельных сервис и мастер-сервис интеграции, который их всех использует. Основной недостаток: сервис получается зависимым буквально от всего и крайне хрупким.
* В чем-то это похоже на Unix идеологию каналов и фильтров: cat input.txt | grep ERROR | wc –1 > output.txt. При этом отдельные этапы обработки ничего друг о друга не знают и могут участвовать в различных интеграционных потоках.
* Достоинства:
  + Весь код разбит на стандартные фрагменты, понятные всем разработчикам
  + Интеграционные компоненты это обычные POJO объекты, представленные в качестве бинов, специально аннотированных или описанных в XML схеме интеграциии
  + Поддерживаются практически все шаблоны Enterprise Application Integration
* Зависимости: spring-integration-core + множество spring-integration-\* по числу технологий, задействованных в интеграционном процессе.
  + Для Java конфигурации - на отдельной конфигурации с интеграционными компонентами **@EnableIntegration**, тогда дальше используем DSL для определения процесса. В данной конфигурации публикуются бины, реализующие отдельные интеграционные компоненты: каналы сообщений, адаптеры каналов, активаторы сервисов и так далее. Каждый такой бин несет две анотации: **@Bean** и, например, **@ServiceActivator**
  + Можно описать в виде XML со схемами **/integration** и, например, **/integration/redis**, тогда задействуем на конфигурации **@ImportResource(classpath:WEB-INF/integration.xml)** и **@AutoConfigurationAfter(RedisConfiguration.class),** чтобы поднять процесс интеграции уже после инфраструктуры Redis

### Основные абстракции

* **Message<T>** - сообщение, содержит собственно payload, а также метаданные в виде **MessageHeaders** набора заголовков
* **Message Channel** – именованный канал передачи сообщений между двумя компонентами обработки
  + некоторая абстракция типа **Queue** – сообщения посредстом данного канала потребляются «из» и отправляются «в» каких-то систем.
  + Служат для развязки между собой различных потоков интеграции. При этом отдельные потоки интеграции можно использовать как функции, которые повторно используются потоками более высокого порядка за счет применения к каналам сообщений
  + По количеству читателей: Point-To-Point в виде **DirectChannel** и Publish-Subscribe в виде **PublishSubscribeChannel**
  + Основные операции с точки зрения интеграции: **send, subscribe**
  + Создаются просто конструктором (опционально с указанием **Executor**), публикуются как **@Bean** методы, возвращающие **MessageChannel**, наименование канала == наименование бина
  + Чтобы создать P2P канал, присоединенный к PS, используется **@BridgeFrom** бин на основе **DirectChannel**
* **Inbound Channel Adapter** – компонент, который присоединяет интеграционный процесс к внешней системе, которая служит источником сообщений
  + Фактически это однонаправленный транслятор между стандартным протоколом и интеграционным процессом
  + Имеются стандартные для Http, Amqp, Jdbc, Redis, WebSocket и так далее
  + Бывают опрашивающие и управляемые событиями
  + Создаются как **@InboundChannelAdapter(channel=”наименование канала в который отправлять сообщение”)** бин, возвращающий реализацию **MessageSource<T>**
    - Например, **FileReadingMessageSource**, для которой можно определить директорию, фильтр на наименование файла, необходимость блокирования найденного файла и так далее
    - Дополнительные свойства аннотации: **autoStartup, poller** (возможность за счет **@Poller** определить периодичность опроса)
* **Inbound Gateway** – двунаправленный адаптер, транслирует не только запрос внешней системы в мообщение интеграционного процесса, но и обратно. Аналогично есть реализации для множества транспортов
* **Outbound Channel Adapter** – транслирует сообщение интеграционного процесса в вызов внешней системы
* **Outbound Gateway** – аналогично, двунаправленный адаптер для вызова внейшней системы
* **Message Handler** – вариант **Channel Adapter** для соединения с Java кодом
  + Имеется стандартные для различных технологий, расширяющие **MessageHandler**, например, **FileWritingMesssageHandler** или **MethodInvokingMessageHandler**
* **Message Gateway** – автоматический конвертер для формирования сообщения из аргумента вызова бизнес-метода, входная точка в интеграционный процесс для Java кода
  + Интерфейс с **@MessagingGateway**
  + Внутри методы принимающие аргументы бизнес типов, возвращающие **void** и аннотированные как **@Gateway(requestChannel = “ appChannelId ”),** которые берут на себя формирование и отсылку сообщения
* **Service Activator** – напротив, выход из интеграцинного процесса в Java код локальный или с вызовом внешней системы
  + Создаются как **@ServiceActivator(inputChannel)** бин, который возвращает реализацию **MessageHandler**
* **Transformer** – преобразователь сообщения, например: **HeaderEnricher, ContentEnricher**
* И до фига всего остального: преобразователи, фильтры, маршрутизаторы, разделители, агрегаторы и так далее
* **IntegrationFlow** – поток интеграции, граф компонент, соединенных между собой
  + Spring Integration предлагает DSL на основе статических методов **IntegrationFlows** для сборки данных абстракций между собой в описание интеграционного потока. В общем ничего нового по сравнению с Camel.
  + Важно, что интеграционный компоненты не имеют никакого понятия об интеграционном потоке (потоках) в рамках которого они вызываются

### Реализация

* Публикуем бин типа **IntegrationFlow**, создаваемый за счет последовательности методов **IntegrationFlows**
  + Последовательность обработки – цепочка методов построителя, например (методов гораздо больше, просто для получения представления о подходе)
    - **from( адаптер | другой поток интеграции | канал сообщений)**
      * например **Files.inboundAdapter(dir)**
        + где **dir** конфигурируем как **@Value(“путь к директории”) File dir;**
        + можно конфигурировать, например, опции опроса или автосоздание директории
      * Дальше идут многочисленные методы **IntegrationFlowBuilder**
    - **transform( тип результата, код преобразования)**
    - **aggregate( код конфигурирования AggregatorSpec )**
    - **handle( метод промежуточной обработки)**
      * может быть несколько последовательно
      * должен вернуть сообщение текущее или другое, создаваемое при помощи **MessageBuilder.** Возврат **null** завершает обработку.
      * в принципе на этом можно заканчивать
    - **routeToRecipients(построитель условного разветвителя)**
      * **builder.recipient( поток интеграции | канал сообщений, условие на сообщение).recipient()… get()**
    - В конце **get()**
  + При этом автоматически создается канал сообщения с идентификатором **“<ИД потока>.input”**. Его можно достать из контекста как именованный бин и отправить в него сообщение или использовать с **@Gateway(requestChannel =”<ИД потока>.input”)**
* Spring сам собирает определения по всем опубликованным бинам и запускает процесс интеграции

## Spring Cloud Stream

* Надстройка на Spring Integration, в которой делается упор на легкость экстернализации конфигурации
* Абстрактная обертка над различными технологиями обмена сообщениями. Например, **AMQP** ( например **RabbitMQ), Apache Kafka, JMS (**например **ActiveMQ), REDIS, WAS Simple Queueing Service, Apache Geode** и так далее. Как обычно: с одной стороны скрывает сложность, с другой не позволяет легко релизовать нестандартные сценарии
* Основные примитивы:
  + **Source** – коллекция методов, позволяющая публиковать сообщения в различные каналы. Каждый метод принимает аргумент – сообщение, реализуя перекодировку самостоятельно
  + **Binder** – связка со средой передачи сообщений
  + **Sink** – коллекция методов, позволяющих получать сообщения
  + **Source / Sink** это аннотированные интерфейсы, на основании которых Spring поставляет реализации бинов, обрабатывающих сообщения.
  + Связь с реальными каналами осуществляется через идентификаторы в аннотациях и свойства конфигурации, что позволяет переконфигурировать сервис на различные каналы / среды без необходимости перекомпиляции
* Зависимости: **spring-cloud-stream** и в зависимости от выбранной среды передачи сообщений, например, **spring-cloud-starter-stream-kafka**
  + kafka тут используется как средство доставки сообщений, но вообще-то это средство доставки логов:
    - сообщения строго упорядоченны и клиент в случае необходимости может прочитать не только последние сообщения, но и историю вглубь
    - в отличии от JMS и тем более ESB это концепция “dump channel + smrat client”, а не наоборот
  + В случае RabbitMQ можно spring-cloud-starter-stream-rabbit, в том числе становятся доступными и низкоуровневые примитивы типа **ConnectionFactory, RabbitTemplate, AmqpOperations** и так далее

### Отправка сообщений:

* На конфигурации **@EnableBinding(Source интерфейс** с каналами вывода, интерфейс отправки сообщений**)**
* Каждый интерфейс с каналами вывода определяет множество методов принимающих тип сообщения, аннотированных **@Output (“appChannelId”)** и возвращающих **MessageChannel**.
  + Можно использовать дефолтный интерфейс **Source**, который определяет метод **@Output(“output”) output(Object),** в данном случае **appChannelId===output**
  + Соответственно, данный интерфейс это коллекция (в терминах JMS и серверов приоложений) **Destinations**, в которые можно отправлять сообщения
  + **appChannelId** – это внутренний локальный идентификатор канала, используемый при конфигурировании, может быть любым
* Дальше **@Autowired** данный интерфейс в код и вызываем **source.someMethod().send(msg).** Чтобы построить сообщение есть **MessageBuilder** с удобными методами
* Осталось сконфигурировать
  + Канал за счет **spring.cloud.stream.bindings.appChannelId**
    - **destination** – нативный идентификатор канала, оба потребитель и поставщик сообщений должны, соответственно, использовать один и тот же нативный идентификатор, чтобы наконец встретиться на сервере сообщений
    - **content-type** – как кодировать сообщение
  + Среду передачи сообщения за счет **stream.bindings.kafka.binder** и так далее, набор свойств зависит от конкретной среды
* Не забывать переносить в сообщение **correlationId, Authorization** и другие инфраструктурные заголовки из входящего HTTP вызова или сообщения

### Получение сообщений

* На конфигурации **@EnableBinding(Sink интерфейс с каналами ввода)**
* Каждый интерфейс с каналами ввода определяет множество методов без аргументов, аннотированных **@Input(“appChannelId”)** и возвращающих **SubscribableChannel**
  + Можно использовать дефолтный интерфейс **Sink** с идентификатором **«input»,** в данном случае **appChannelId===input**
* Реализация потребителя
  + Можно просто навесить слушатель на **SubscribableChannel**
  + Можно на каком-то методе произвольного бина использовать **@StreamListener(“appChannelId”),** и принимать аргументы типа сообщения
  + Можно использовать **SubscribableChannel** при создании потока интеграции за счет **IntegrationFlows.from(subscribableChannel)**
* Осталось сконфигурировать
  + Канал за счет **spring.cloud.stream.bindings.appChannelId**
    - **destination** – нативный идентификатор канала
    - **content-type** – как кодировать сообщение
    - **group** – из всех слушателей с одним **groupId** сообщение будет доставлено только одному
    - **durableSubscription**
  + Среду передачи сообщения за счет **stream.bindings.kafka.binder** и так далее, набор свойств зависит от конкретной среды
* Если в процессе обработки сообщения осуществляется HTTP вызов, то **correlationID** (а также **Authorization**, ИД транзакции и так далее) в запрос надо подмешивать ручками

## EDA на основе HTTP

* В принципе EDA архитектура это совсем не обязательно про очереди собщений, ее вполне можно выстроить и на основе чистого HTTP + Atom Syndication Format или аналоги
* В данном же случае, источник публикует упорядоченные перечни сообщений в виде feed ленты, а слушатели периодически опрашивая перечень, со временем их получают
* Есть мнение, что atom это идеальный формат для высоко масштабируемой event-driven архитектуры. Тем не менее, когда требуется очень низкая latency, данный подход вряд ли применим или вызывает излишнее потребление ресурсов
* Типовой пример: распределенная система, где один из компонентов владеет reference данными, а остальные подписаны на их получение.
  + Например, отдельный сервис управления каталогом товаров, предоставляемый внутри компании остальным системам
  + Когда появляется / удаляется новый товар или обновляется его описание, сервис публикует соответствующее событие либо с новым описанием, либо со ссылкой на ресурс товара
  + Клиенты регулярно опрашивают ленту за счет считывания первого динамического фрагмента ленты, долистываются до последней обработанной ими записи и начинают последовательно считывать и применять к собственной базе все предыдущие.
  + Можно хранить также ссылку на последний считанный фрагмент ленты и от него идти вперед по времени. Но надо помнить, что URL данного фрагмента может быть изменен из-за рефакторинг приложения. Соответственно, если 404, то идем в начало ленты и назад по ссылкам
  + Хорошо работает, когда лента обновляется достаточно медленно, а остальные системы устойчивы к задержке обновления. Для них задержка в час для получения новой цены на товар, к примеру, должна быть не существенна. А вот гарантия получения этого обновления, рано или поздно, напротив важна.
  + Клиенты сервиса, за счет кэширования локального или промежуточного (например, для отдельных филиалов), получают возможность хранить необходимые данные близко от себя и использовать их максимально эффективно
* В данном случае веб-архитектура
  + предоставляет возможность
    - масштабирования сервиса, вплоть до многих тысяч индивидуальных клиенттов
    - устойчивости к падениям собственно самого сервиса, а также сети в целом.
    - устойчивать к падениям и откатам во времени клиентов: стартовав заново, они просто опять вычитают и пременят «забытые» обновления
    - заодно появляется возможность логирования изменений ресурса, в принципе если вся лента сохраняется, то ее можно даже накатить на пустую базу и таким образом восстановить сам ответственный сервис
    - гарантируется порядок доставки и обработки сообщений, что кстати для JMS неверно и иногда является проблемой
  + Хорошо разделяет ответственности по определению данных и их последующему использованию.
  + Любопытно, что по сравнению с JMS системами ответственности инвертированы: не поставщик (или промежуточное ПО) выталкивает обновление и ответственен за его благополучную доставку, а клиент вытягивает его по собственной инициативе.
* Atom Publishing Protocol (или соответственно аналоги) позволяет подписчикам конкурентно разбирать публикуемые задания для выполнения и последующией обратной публикации результатов
  + Источник публикует перечень заданий, каждое из которых снабжено ETag для работы с конкурентностью
  + Исполнитель конкурентно лочит задание. Операция легковесная: не получилось – пошли попробуем залочить другое.
  + Затем исполнитель имеет длинный timeout на выполнение задания, после которого он должен его завершить.
    - Если умирает – задание опять становится доступным для остальных исполнителей
    - Если не успевает – может продлить период за счет отдельного запроса
  + Пример: Camunda BPM публикует перечень external task для внешних асинхронных исполнителей

## Spring Cloud Data Flow

* Платформа для построения (визуального, терминальный клиент, API) потоков данных - цепочек приложений на основе Spring Cloud Stream, обменивающихся друг с другом через каналы данных на основе стандартных для Spring серверов очередей сообщений. Позволяет динамически создавать и управлять сложными архитектурами, управляемыми событиями (SEDA, Staged Event-Driven Architectures)
* Фактически это менеджер приложений, позволяющий по описанию развернуть множество приложений, определив входные / выходные каналы сообщений для каждого таким образом, чтобы сформировать интеграционный сценарий следующего уровня. Примеры сценариев: обработка показаний датчиков, трейдинг, анализ логов и так далее
* Для развертывания приложений полагается на Spring Cloud Deplyer, который абстрагирует задачу развертывания для таких сред как
  + Из коробки – локальный сервер (для тестового использования), Cloud Foundry, Apache YARN, Mesos, Kubernetes. Сторонние реализации для Hashicorp Nomad, RedHat OpenShift
  + Ключевые элементы API
    - AppDeployer – поддержка ЖЦ приложений: развертывание, получение информации о состоянии и так далее
    - TaskLauncher – ЖЦ выполнения задач на основе Spring Batch / Spring Cloud Task: запуск / остановка, очистка данных запущенных экземпляров, мониторинг и так далее
* Основные абстракции
  + Приложение
    - Обыное приложение на основе Spring Cloud Stream
    - В простейшем случае это @ServiceActivator со свойствами inputChannel = Processor.INPUT и outputChannel=Processor.OUTPUT – один канал ввода / один вывода
    - Перечень каналов в конфигурации приложения выливается в перечень портов, которые могут быть сконфигурированы для связи нескольких приложений в поток
  + Определение приложения в каталоге зарегистрированых приложений
    - Свойства
      * Уникальный идентификатор
      * Роль: источники данных / процессоры / приемники данных.
      * Ссылка на maven / docker артифакт, из которого будет развернут экземпляр приложения
      * Опциональные дополнительные данные
    - Регистрировать приложения в каталоге можно ручками, программно, а также за счет специальных файлов определения приложения
  + Схема потока:
    - Каждый компонент схемы выливается в несколько запущенных экземпляров приложения, сконфигурированного в части входны / выходных потоков
    - Состоит из описания последовательность компонент, разветвлений потоков и так далее
    - имеется DSL язык для описания, визуальный редактор Spring Flo
  + Задача
    - Перечень задач автоматически получается из запущенного приложения на основе Spring Integration
    - Среда Spring Cloud Data Flow позволяет запустить ее с некоторыми параметрами один раз / несколько по расписанию / определенное количество экземпляров и так далее
    - Параметры при запуске в итоге будут переданы как аргументы в CommandLineRunner, ApplicationRunner / Job
* Управляющий сервер Spring Cloud Data Flow с хранилищем данных предоставляет
  + Открытое REST API для управления с java based API доступа, основным элементом которого является DataFlowTemplate, который в свою очередь использует RestTemplate
  + Веб-приложение
  + Интерактивная коммандная оболочка на основе Spring Shell

# Выполнение заданий (пакетная обработка данных)

* Первоначально было частым способом утилизировать простаивающие мощности в периоды неактивных пользователей. Сейчас системы глобальны, пользователи активны всегда, а кластера умеют ужиматься в соответствии со снизающейся нагрузкой. Так что сейчас не столь актуально.
* Однако все еще используется для регулярной обработки больших по природе массивов последовательно организованных данных: наборы файлов, результаты SQL выборок и так далее.
* Такие данные, как правило, одновременно в память не влезают, а потому делятся на блоки и обрабатываются один за другим. Соответственно такая обработка может быть распараллелена как локально, так и глобально на кластере.
* Основной недостаток: при подобной обработке мы неизбужно сталкиваемся с устареванием данных, поэтому в основном применяют к диапазонам данных, вышедшим из бизнес-процесса. Например, статистика по продажам за предыдущую неделю.
* Spring Batch - предлагаемое Spring решение для поддержки обработки большого количества записей с данными. Включает как библиотеку, так и UI для работы с заданиями (запуск, получение статистики, перезапуск, пропуск и так далее)

## Основные элементы архитектуры

* **JobRepository** – хранилище данных
  + Среда Spring Batch работает с сохранением состояния запущенных / остановленных / прерванных задач
  + Можно, в частности, посмотреть, были ли и почему пропущены в обработке отдельные элементы и привело ли это к отмену задания в целом
* **JobOperator** – оператор задания
* **Job** – собственно задание
  + Имеет наименование и состоит из набора также именованных этапов. Все это видно в интегрфейсе.
  + выполнение задания разбивается последовательно на несколько этапов – **Step**. Каждый этап может быть выполнен либо на одном и том же, либо на различных узлах кластера, в зависимости от настройки процесса
  + В контексте каждого этапа либо работает произвольный код – **Tasklet**, либо специализированне компоненты: **ItemReader, ItemProcessor, ItemWriter**
* **ItemReader** – умеет считывать элементы данных из внешнего мира, аккумулируя их для бальнейшей обработки в блоки. Так как данные считываются блоками, то система ввода / вывода используется максимально эффективно.
* **ItemProcessor** – обработка данных НЕ связанная с вводом / выводом: валидация, трансформация и так далее
* **ItemWriter** – умеет выгружать обработанный блок (это важно) данных во внешнюю систему. Так как данные выгружаются блоками, то система ввода / вывода используется максимально эффективно.

## Локальное выполнение

* На конфигурации **@EnableBatchProcessing**
* Доступное оператору для запуска задание публикуется как **@Bean** типа **Job**, который строится за счет **JobBuilderFactory**
  + Типовая последовательность: **jbf.get(наименование).incrementer(как генерить последовательные ИД). start(someStep).next(someStep)…build()**
  + Можно сконфигурировать **TaskExecutor** для распараллеливания, но тогда сбойное задание нельзя будет перезапустить с определенного смещения в наборе данных
* Отдельные этапы создаются за счет **StepBuilderFactory – sbf.get(наименование)...build().** Типовые операции:
  + **tasklet()** – выполнение произвольного кода и возврат статуса результата
  + **chunk()** – определение размера блока данных
  + **faultToTolerant().skip(SomeException).retry(SomeException).retryLimit(n)** – политика отказоустойсивости при обработке каждого элемента данных. Ошибка обработки очередной записи НЕ обязательно ведет к краху всего задания в целом
  + **reader().processor().writer()** – реализации основных этапов обработки, если не был определен **tasklet**
    - Чтение / запись за счет специализированных компонент, ориентированных на определенную технологию.
      * Например, **FlatFileItemReader и FlatFileItemWriter**, которые создаются за счет **FlatFileItemReaderBuilder и FlatFileItemWriterBuilder** соответственно. Определяем куда / откуда читать / писать, с каким разделителями, какие свойства эдемента данные использовать и в каком виде, ну и так далее
      * Другой частый пример: **JdbcCursorItemReader и JdbcBatchItemWriter**
    - Обработка производится сконфигурированыым или созданным на ходу специально **ItemProcessor<F,T>.** Здесь мы валидируем данные и при необходимости обращаемся к внешним сервисам для их обработки.
    - В качестве аргументов используются бины с **@StepScope**, экземпляры которых создаются заново под каждый экземпляр задания. Параметры задания передаются в них как параметры **@Value(“… #{jobParameters[‘paramName’]}”)**
* Запуск при помощи **JobLauncher.run(job, params),** который запускается в бине типа **CommandLineRunner / ApplicationRunner**, автоматически стартуемые при запуске Spring Boot приложения
  + Есть возможность передать **TaskExecutor** и запустить процесс в асинхронном режиме
  + Параметры строятся при помощи **JobParametersBuilder**
  + Возвращается **JobExecution** при помощи которого можно отслеживать состояние и выяснить чем в итоге завершилось
* Работа по расписанию
  + Можно самому ручками внутри приложения за счет JDK **ScheduledExecutorService / @Scheduled** или даже классическим **cron**, если процесс должен запускаться всегда только на одном сервере. Не слишком частый, но возможный вариант. Однако способ явно не соответствует динамической схеме развертывания современных приложений
    - что делать, если выделенный под выполнение пакентных заданий узел упал или вообще не запущен?
    - Если на нескольких, то как обеспечить выполнение в каждый момент времени только на одном узле кластера?
  + Коммерческие диспетчеры типа BMC, Flux Scheduler, Autosys или известное открытое решение, хорошо интегрированное со Spring – Quarz Enterprise Job
  + Spring Integration предоставляет механизм выбора лидера и осуществления распределенных блокировок на основе использования Hazelcast, Apache Zookeeper или Redis. Для этого необходимо предоставить бины типа **Candidate и LeaderInitiator**, соединенные с инфраструктурой обмена сообщениями. В принципе ничего сложного.

## Распределенное выполнение

* Локально на каждом узле можно распралелить выполнение за счет конфигурируемого для **Job** экземпляра **TaskExecutor**, но это вертикальное маштабирование, а мы хотим горизонтального – разбрасывать задания и отдельные этапы заданий по узлам кластера
* Возникают узлы двух типов: **лидирующий** (управляющий, в каждый момент времени единственный, формирует и рассылает задания) и **рабочие** (выполняют задания)
  + Между собой они соединены через среду рассылки сообщений (то есть посредством **MessageChanels**), а также через общее хранилище данных, которое используется **JobRepository.** Всего используются три канала: **leaderRequsts, workerRequests, workerReplies**
  + Бизнес код при этом состоит из трех частей: выполняемый только на лидирующем узле / только на рабочих / и там и там. Простейший механизм для этого Spring профили, однако когда лидирующая роль мигрирует по кластеру, то приходится переключать активности динамически
* Два типа распределения заданий
  + **Remote Partitioning** – распределение с разделением.
    - На рабочие узлы рассылаются сообщения с диапазоном записей для считывания. После чего уже там запускаются **ItemReader, ItemProcessor и ItemWriter**.
    - Данный подход требует, чтобы все рабочие узлы имели доступ к ресурсам, контролируемым лидирующим узлом.
    - Оптимально, когда основная работа выполняется в **ItemReader / ItemWriter** и необходимость распределения связана больше всего с проблемами вывода / вывода
    - Наиболее частый случай, посколько большинство пакетных заданий так или иначе связаны с объемным вводом / выводом
  + **Remote Chunking** – распределение с фрагментированием
    - данные считываются на лидирующем узле и рассылаются на рабочие для обработки
    - Оптимально, когда основная нагрузка приходится на **ItemProcessor** и имеет счетную природу / занимает большой объем памяти
    - Фактически, это **map - reduce**
  + В любом случае, результат удаленного выполнения этапа задания возвращается на лидирующий узел и сохраняется в хранилище
* Реализация лидера
  + Конфигурируется **Job**, который может включать в себя как локальные, так и удаленные этапы
  + Локальные этапы конфигурируются обычным образом, для удаленных
    - Вместо локальной логики исполнения за счет **StepBuilderFactory** конвигурируем эккземпляры **Partitioner и PartitionHandler**
    - **PartitionHandler** – ответственен за формирование и рассылку сообщений **StepExecution** с диапазоном обрабатываемых записей.
    - Обычно на основе **MessageChannelPartitionHandler**, который связывается с **JobExplorer** и конфигурируется
      * Числом рабочих узлов, между которых осуществляется распределение гнагрузки
      * Каналом приема сообщений от лидера, который уходит в **JobRepository**
      * наименованием этапа, который должен быть выполнен удаленно на рабочем узле
    - **Partitioner** – ответственен разделение рабочей нагрузки - формирование наборов идентификаторов обрабатываемых записей исходя из числа рабочих узлов
* Рабочего узла
  + Посредством **StepExecutionRequestHandler** получают работу у лидера через **MessageChannel** в виде отдельных объектов **StepExecutionRequest**, которые передаются за счет **StepLocator** для выполнения локально описанному этапу, на сей раз содержащему нормальную бизнес логику
  + Бин типа **StepExecutionRequestHandler**, конфигурируется ссылками на **JobExplorer** и **StepLocator**, отвечает за выполнение **StepExecutionRequest**
  + Бин типа **StepLocator** за счет стандартного **BeanFactoryStepLocator**, который ищет бин типа **Step**, одноименный наименованию этапа, указанного в **StepExecutionRequest**, который в состоянии обработать данные, указанные в запросе
  + Бин типа **IntegrationFlow** за счет **IntegrationFlows**
    - **from(канал получения StepExecutionRequest от лидера для выполнения)**
    - **handle()** с передачей запроса в **StepExecutionRequestHandler** для выполнения
    - **channel(канал выброса полученного результата обработки в виде StepExecutionResponse обратно лидеру)**
  + Бин типа **Step**, требуемого наименования, строится за счет **StepBuilderFactory** обычным образом.
    - В реализации логики доступ к данным запроса осуществляется за счет **StepExecutionContext**
    - Каждый бин, используемый конфигуратором, по-прежнему **@StepStone**,параметры могут использовать **@Value(“#{stepExecutionContext[‘minValue’]}”),** например

## Управление задачами

* Задача это фрагмент кода, на выполнение которого расходуется достаточно большое количество времени по сравнению с обычным пользовательским запросом. Обычно запускается с различными параметрами (ключевой момент – наличие интерфейса параметризации) один или несколько раз по расписанию / событию. Например:
  + Обработка свалившегося в директорию файла с данными
  + Очистка мусора
  + Периодическое формирование и рассылка отчета
* Spring Cloud Task позволяет параметризировать и запускать задачи в виде **CommandLineRunner, ApplicationRunner**, Srping Batch **Job**
* Зависимость **spring-cloud-task-starter + @EnableTask** на конфигурации
* В результате получаем возможность **@Inject TaskExplorer**, позволяющий получать информацию о запущенных задачах

# Обеспечение целостности данных

* В условиях распределенной системы актуальнейшая и сложнейшая задача. Эвристическая теорема **CAP** ( теорема Брюера) утверждает, что любая распределенная система может иметь только два из следующих свойств
  + **C – consistency,** согласованность данных, в каждый момент времени данные на всех узлах системы не противоречат друг другу
  + **A – availability,** доступность данных, любой запрос к системе завершается корректным откликом, однако без гарантии, что ответы ото всех узлов системы совпадут
  + **P – network partitioning tolerance,** расщепление распределённой системы на несколько изолированных секций не приводит к некорректности отклика от каждой из секций.
* Варианты систем (чем жертвуем):
  + **CA** – может развалиться на две части, каждая из которых будет выдавать свой результат. Реализуемо на основе ACID транзакций, например, кластерная СУБД
  + **CP** – может при каких-то условиях не выдавать ответа вообще. Реализуемо на основе распределенных песимистических блокировок
  + **AP** – данные на различных узлах могут не соответствовать друг другу. Пример – кэши или узлы DNS
* Так как **tolerance** это обычно предъявляемое требование, то вопрос чем будем жертвовать A или С? Обычно жертвуем consistency, переходя от постоянно согласованных систем к в «конечном итоге согласованным».
  + **BASE архитектура -** ***Basically Available, Soft-state, Eventually consistent*** — базовая доступность, неустойчивое состояние, согласованность в конечном счёте, прямое противоречие с ACID
  + **Базовая доступность** - сбой в некоторых узлах приводит к отказу в обслуживании только для незначительной части сессий при сохранении доступности в большинстве случаев
  + **Неустойчивое состояние** подразумевает возможность жертвовать долговременным хранением состояния сессий (таких как промежуточные результаты выборок, информация о навигации, контексте), при этом концентрируясь на фиксации обновлений только критичных операций.
* Типы транзакций:
  + **1PC – вырожденный случай**
    - Координатор сначала выполняет операции на всех ресурсах. Затем последовательно так же последовательно закрепляет
  + **2PC – двухфазная транзакция**
    - Координатор (компонент, начинающий транзакцию и регистрирующий у себя все ресурсы, подвергающиеся обновлению) опрашивает всех на предмет готовности, если кто-то не готов – откат
    - Затем ресурам предлагается зафиксировать транзакцию, если кто-то не сумел – откат
  + **3PC - трехфазная транзакция**
    - Добавляется еще один этап опроса – **preCommit**
    - В принципе более устойчив к отказам ресурсов, в JEE мире это экзотика
* Зачастую необходимость в **shared / distributed** транзакциях вообще порождена неправильным дизайном или желанием в лоб оптимизировать систему. Пример:
  + Пользователи отправляют сообщения, в профиле пользователя хочется выводить время последнего принятого / отправленнного сообщения
  + Если у нас есть два сервиса, один из которых занимается сообщениями, а второй пользователями, то лобовое добавление сообщения выливается в необходимость согласованного обновления данных двух источников: добавить сообщение в коллекцию + обновить свойство пользователя
  + Варианты решения:
    - При выводе профиля пользователя отдельно запрашивать время у сервиса сообщений.
      * Удобно и логично , но +1 запрос и время. Более того, если контроллер формирования страницы с профилем запрашвает оба сервиса (или что еще хуже, сервис пользователя запрашивает сервис сообщений), то тормозящий (а это очень вероятно) сервис сообщений будет укладывать за собой выдачу профиля
      * Лечится асинхронным запросом уже из страницы, выданой пользователю.
      * Более того, это позволяет динамически обновлять поле на уже открытой страницы без необходимости ее перезагружать
      * Плюс разумное кэширование на query операции, потому что точность до секунды явно никому не сдалась.
    - Сервис пользователей ведет у себя локальную реплику статистики сообщений по пользователям.
      * При добавлении сообщения, на шину данных выбрасывается сообщение, которое подхватывается сервисом пользователей и обновляет реплику
      * Имеется некоторое неустранимое временной рассогласование обоих наборов данных, но оно вполне терпимо
  + В общем решения не без недостатков, но это гораздо лучше, чем откатывать операцию добавления сообщения только потому, что сервис пользователей не ответил вовремя. Бизнес-ценность его ответа ничтожна, а проблем получаем масса

## Проблемы:

* Что делать, если кто-то не сумел откатиться? А если координатор умер посреди процесса, что делать источникам посредине процесса обновления?
* Значительная нагрузка на сеть в процессе переговоров с ресурсами. Особенно, когда в транзакции участвуют не два и не три сервиса / ресурса, а больше. При росте числа ресурсов, задействованных в транзакции, производительность падает не линейно.
* Распределенная транзакция занимает значительно большее время. Соответственно, повышенная нагрузка на всех участников:
  + В координаторе большее количество потоков заблокировано. На ресурсах данных повышенные требования к локальным механизмам типа «сегмент отката», которые ответственны за блокировки отдельных записей
  + если какой-то участник тормозит, то начнут по timeout откатываться остальные. Особенно это чувствительно, когда они это начнут делать на этапе подтверждения / отката глобальной транзакции. Причем тормоза распространяются по системе недетерминировано вне всякой связи с тем, какие сервисы от каких зависят.
  + традиционные реляционные СУБД принципиально не рассчитаны на работу под нагрузкой в условиях, когда продолжительность типовой локальной транзакции привышает 30 ms. Для NoSQL, зачастую работающих в однопоточном режиме, требования еще жестче. Поэтому они, кстати, достаточно редко предоставляют подобную возможность
* Вообще, при использовании глобальных транзакции существенно расширяются фрагменты кода, которые могут быть выполнены только последовательное. А это согласно закону Амдала всегда режет масштабируемость системы в целом
* Заблокированный ресурс в это время не может участвовать в других бизнес транзакциях,
  + распределенные транзакции дополнительно начинают ждать друг друга, таким образом получает отрицательная обратная связь
  + повышается вероятность непредсказуемых deadlock
* Даже в рамках глобальной транзакции на уровне отдельных ресурсов мы имеем отдельные изолированные транзакции (даже если это таблицы в одном экземпляре БД, «принадлежащие» различным микросервисам)
  + Соответственно изменения в одном сервисе не видят изменения в другом. В общем-то это не большая проблема, так как предполагается, что у каждого сервиса изолированное хранилище и декларативная ссылочная целостность за счет ключей между ними не используется
  + Точные моменты времени, в котором завершатся эти локальные транзакции на отдельных ресурсах в процесс второй фазы не известны, соответственно мы получаем race condition.
    - Например, если вторая система вслед за завершением операции пытается обратиться к первой за только что вставленными / обновленными данными (для BPMN процесса, в который в ходе операции вбросили сообщений это частый случай), то ее может ожидать отказ.
    - Особенно, если одна из систем или обе находятся под значительной нагрузкой
* Очень сложно работать с транзакциями когда
  + Хочется асинхронной работы
  + Необходимо выполнять операции повторно в случае timeout ответа
* В общем распределенные транзакции в распределенной системе убивают hight load, режут масштабируемость и вносят неустранимую неопределенность в поведении, с этим приходится смириться

## X / Open XA транзакции

* Поддерживаются в Java мире при помощи **JTA API**
* Обычно это 2PC транзакции на несколькими базами данных или базой данных и сервером обмена сообщениями
* Для их осуществления необходимо использование
  + специального менеджера транзакций **JtaTransactionManager** и специальных реализаций **DataSource / ConnectionFactory**, рассчитанных на взаимодействие с таким менеджером
  + Если развертываемся на сервере приложений это означает, что и то и другое надо брать с него при помощи JNDI.
  + Если чисто Spring приложение, то надо использовать опять-таки специальные **DataSource / ConnectionFactory** (обычно предоставляются провайдерами ресурсов) + сторонний транзакционный менеджер, например, **Atomikos или Bitronix**
* Частые ошибки:
  + Hibernate + DataSourceTransactionManager
  + административный персонал сервера приложений снимает где-нибудь галочки и ресуры становятся не XA совместимыми. Иногда, приложение этого не замечает и мы имеем дело с восхитительно странными постэффектами
  + В результате имеем дело со стратегией **Wing-and-a-Prayer**

### Дополнительные недостатки:

* чтобы их использовать, ВСЕ ресурсы должны быть реализованы на одной JTA платформе или приходится реализовывать нетривиальные преобразователи между их данными
* Для общения между микросервисами приходится применять протоколы, рассчитанные на перенос транзакционного контекста.
  + Например, WS-Transaction над Xml Web Services, RMI, Corba – все они (ну разве частично кроме первого) для микросервисов достаточно экзотические и тяжеловесные
  + RedHat предложил REST-AT
    - спецификация до сих пор не вышла из состояния черновика
    - протокол рассчитан на использование сервера приложений, выступающего в качестве координатора, со специальными API, а также требует от всех микросервисов поддержки также специального API

### Шаблоны оптимизации

* могут опционально реализовываться менеджерами транзакций. В принципе, универсальные подходы, полезные и в других случаях.
* По мере изложения надежность снижается, а достигаемый выигрыш закономерно растет
* **автоматическиая редукция к 1PC** в случае, когда менеджер видит, что в транзакции реально задействован только один ресурс
* **Last Resource Gambit** – в распределенную транзакцию можно безопасно подмешать один НЕ-XA-совместимый ресурс, если операцию его обновления вызывать последней
* **Shared Transasction Resource**
  + Можно вообще не связываться с XA транзакцией, если оба отдельных ресурса на самом деле реализованы одим и тем же ресурсом
  + Примеры:
    - Совместное использование одного и того же соединения с БД компонентами, основанными на ORM и JDBC
    - Все сервисы системы обращаются в одну и ту же БД, поэтому можно им позволить работать с одной и той же нативной транзакцией, ИД которой передавать через аргумент вызова
    - «синтетическая» схема БД, аггрегирующая в себя несколько отдельных схем или даже экземпляров БД посредством синонимов или dblink (Oracle, PostgreSQL)
  + Особенно частный случай: совместная работа со службой сообщений и БД оптимизированная за счет того, что что сообщения хранятся в той же самой базе данных и обрабатываются той же самой локальной для БД транзакцией
    - Не все вендоры это поддерживают, но вот, например, к Apache ActiveMQ можно прикрутить свой адаптер хранения
      * Встроенный брокер + ActiveMQConnectionFactory c brokerUrl=”…&async=false”
      * JDBCPersistenceAdapter + TransactionAwareDataSourceProxy(dataSource, jmsTemplate)
      * JmsTemplate.sessionTransacted=true
  + Значительно уменьшает сложность системы и повышает пропускную способность. Однако не всегда применим.
* **Best Efforts 1PC**
  + Если у нас есть два (несколько) ресурса, транзакция на одном из которых практически всегда завершается успешно («надежный» ресурс, как правило, инфраструктурный. Второй, «менее надежный», это как правило бизнес-ресурс, с которым возможны всевозможные неприятности), то можно вообще обойтись без XA транзакции, выполняя закрепление на надежном ресурсе последним
  + В общем-то это опять-таки оптимистическая транзакция. Все, чем мы рискуем, причем с довольно низкой вероятностью, это необходимость повторить операцию еще раз с повторным вызовом бизнес операции на менее надежном ресурсе. Поэтому идеально, если это идемпотентная операция
  + Классический пример: обработка сообщения в Spring с записью данных в БД
    - Ключевой момент: в результате сначала закрепляем локальную транзакцию в БД и только затем в системе очередей. В каком порядке из получать – безразлично
    - Так как каждое сообщение имеет уникальный ИД, то организовать на данном месте идемпотентность бизнес операции достаточно элементарно
    - TransactionAwareConnectionFactoryProxy над ActiveMQConnectionFactory с synchedLocalTransactionAllowed=true
    - Слушатель JSM используется самый обычный (не JTA!!!) DataSourceTransactionManager на базой данных
  + Другие способы:
    - Каскадирование менеджеров транзакций
      * В общем-то это такая SAGA без компенсирующих операций: если все хорошо, то всем commit, если нет – оставшимся rollback
      * Для Spring это реализуется за счет ChainedTransactionManager. Commit / Rollback производится в обратном порядке, относитель того, в котором обычные TransactionManager добавляются в конфигурацию
    - Ручное использование Transaction Synchronization API
      * Когда в текущую транзакцию добавляет обновляемый ресурс, регистрируем слушатель, который в случае успеха закрепит его локальную транзакцию
      * Собственно комбинация TransactionAwareConnectionFactoryProxy + DataSourceTransactionManager именно это и делает
      * По сравнению с предыдущим это оптимизация, так как глобальная транзакция будет иметь дело не со всеми возможными менеджерами транзакций, а только с относящимися к реально затронутым ресурсам
* **Nontransactional Access**
  + Некоторые ресурсы маргинальны относительно бизнес процессов и их мало вероятной «порчей» можно попросту пренебречь. Или же вызвать отложенную (иногда сильно отложенную, вплоть до ежедневной batch) операцию очистки
  + Реализация: основная операция под контролем PlatformTransactionManager, маргинальная за счет соединения, ручками взятого из DataSource c autoCommit=true
  + Примеры: аудит, версионность, логирование
* Итого
  + Начать можно с **Wing-and-a-Prayer** и на основании получаемых проблем рано или поздно придти к необходимости в некоторых местах добавить синхронизацию. Это частый, интуитивно получающийся из-за непонимания платформы, вариант
  + Лучше всего по возможности использовать **Nontransactional Access**  или **Best Efforts 1PC**
  + **Shared Transasction Resource –** штука очень эффективная, но с трудом применимая
  + **Last Resource Gambit –** практически так же безопасно и функционально, как и 2PC, но экономит достаточно мало
  + **2PC** это наиболее общий, корректный и универсальный способ, а потому наиболее дорогой

## SAGA шаблон

* Изначально был введен аж в 1987 году для обработки длинных транзакций
  + Из-за продолжительности такой транзакции сервер СУБД успеет ее несколько раз оборвать, потому что он не может столько времени блокировать собственные ресурсы, например, для оракла размер rollback segment не позволяет.
  + Кроме того, наличие подобные транзакций просто убьет производительность системы в целом.
* Соответственно надо разделить большую логическую на набор множества физически / нативных – запросов к отделным ресурсам / сервисам
  + Запросы различных физических транзакций должны быть перемежаемы между собой, не могут зависеть друг от друга и могут быть выполнены друг относительно друга в любом порядке
  + В результате выполнения каждой физической транзакции должна быть получена и сохранена минимальная информация, необходимая для ее отката.
  + Для каждой физической транзакции долна быть определена компенсирующая операция, возвращающая соответственный ресурс в адекватное состояние.
    - Это НЕ обязательно исходное, так как в условиях конкурентной работы за время логической транзакции к незаблокированному ресурсу могут быть применены другие изменения.
    - Компенсирующая операция это такая же бизнес операция со своей логикой, которая уникальна для природы сервиса
* Ограничения на архитектуру системы:
  + Основные операции с ресурсами должны
    - быть реализованы по принципу «максимум один раз».
    - возвращать данные, необходимые для собственной отмены. Например, ПК созданной сущности или ее версии, созданной в ходе обновления. Возможно, часть данных будет относиться не к самому сервису, а к тому, который он в свою очередь вызвал
  + Компенсационные должны
    - Быть реализованы по принципу «хотя бы один раз»
    - быть идемпотентными
    - лучше всего, если они могут быть выполнены в любом порядке (наиболее «чувствительные» данные компенсируем первыми) и параллельно, это ускоряет процедуру отмена логической транзакции
* В **SAGA** шаблоне доступность достигается за счет изолированности отдельных транзакций друг от друга – **AC(-I)D**:
  + побочные эффекты от каждого запроса / отказа видны всей системе вплоть до завершения выполнения / отмены всей логической транзакции
  + то есть: грязное чтение, неповторяющееся чтение, перезапись изменений между отдельными бизнес транзакциями и так далее…
* Достоинства
  + Поддерживает длинные транзакции, асинхронные операции, перезапрос данных, интеллектуальный recovery отдельных проблем (в том числе, с участием пользователя и / или административного персонала) и так далее
  + Отдельные ресурсы, задействованные в рамках глобальной транзакции, блокируются на минимальное время, что сохраняет производительность и масштабируемость системы
  + Обычно применяется в системах с большим количеством запросов,например, почтовые сервисы или социальные сети
* Проблемы:
  + Что делать, когда координатор выполнения накрывается посредине этапа, когда ответ от сервиса уже получен, а состояние **SAGA** / процесса / логической транзакции еще не сохранено
  + Не избавляет от необходимости делать вызываемые операции сервисом идемпотентными. Однако сильно облегчает, так как теперь есть контекст для постоянного хранение requestUid, числа уже выполненных перезапросов и тому подобных данных
  + Схемы данных сервисов и сами сервисы существенно усложняются за счет необходимости реализовывать операции отката
  + Трудно отлаживать, особенно когда реализуется на основании асинхронного обмена сообщениями
  + Остальной код также усложняется, так как посредине любой операции с данными может прилететь операция отката, которая их поменяет. Но это вообще свойственно асинхронным системам, так что не большое усложнение
* Если распределенные транзакции по природе своей пессимистические, то SAGA это пример оптимистического подхода

### Принципы реализации

* **Events / Choreography**
  + Нет центрального координатор, каждый сервис публикует на шине сообщения и в свою очередь, на основании получаемых сообщений, индивидуально решает что с ними делать
  + SAGA процесс завершается благополучно, когда очередной сервис не публикует сообщения. Или когда в ответ на его сообщение никто больше не производит операций в бизнес контексте SAGA
  + SAGA процесс завершается неявно: очередной сервис публикует сообщение о неудачной обработке очередного сообщения. В ответ на него остальные сервисы начинают откатываться. Последовательно или одновременно ему не ведомо
  + В данном случае критично, чтобы сообщения об откате содержали ИД транзакции, чтобы сервисы знали к каким именно данным они относятся
  + Достоинства:
    - просто в реализации и понимания общего принципа
    - все участники слабо связаны и не знают друг о друге.
    - Если бизнес процесс включает несколько шагов, то возможно это наилучший вариант
  + Недостатки
    - При возрастании числа участников становится сложно понять, какие сервисы кого слушают
    - Логика процесса / отката процесса распределена по участникам и не прозрачна. Совершенно не понятно, кто же в итоге завершает процесс. Число участников не определено и сложно определяемо
    - Возможно возникновение сложно детектируемых циклических зависимостей между сервисами и возникновение «вечных» процессов
    - Сложно тестировать, так как при этом необходимо симулировать наличие всех участников
* **Commands / Orchestration**
  + Реализуется за счет координатора выполнения (**SAGA execution coordinator, SEC**), концентрирующего в себе логику выполнения экземпляра SAGA
  + В нем отслеживается прогресс выполнения отдельных физических транзакции в контексте логической, производится сохранение состояния логической транзации, а также даных для отката выполненных физических транзакций.
    - Данные при этом долговременно сохраняются в базе, а сам по себе координатор stateless. Поэтому в случае сбоя на смену опущенному экземпляру координатора всегда можно поднять новый, который продолжит работу
    - При сбое в выполнении очередной физическая транзакции логическая откатывается за счет выполнения всех компенсационных операций для уже ранее выполненных физических транзакций. Если же компенсационная дает сбой, то она запускается повторно снова и снова.
  + Два подход: push – координатор вызывает обработчиков и pull – координатор публикует задания для выполнения, а обработчики их разбирают. Второй вариант сложнее, но лучше масштабируется и более отказоустойчив.
  + Достоинства
    - Циклические зависимости не возможны, так как сервисы не вызывают друг друга. Оркестратор в данном случае единственная точка активности
    - Снижается сложность сервисов
    - Для сложных процессов проще в понимании и реализации. Сложность процесса растет линейно с увеличением числа участников. Легче управлять откатами.
    - Легче тестировать, так как mock сервисов в данном случае ничего делать не должен
    - Если мы обнаруживаем, что прилетает следующая транзакция для уже обрабатываемого объекта, то ее проще поставить на паузу
  + Недостатки
    - Риск сконцентрировать слишком много бизнес логики в оркестраторе и получить «глупые» рудиментальные сервисы, в пределе выполняющие CRUD операции
    - Дополнительный сервис
  + Фактически мы имеем дело с машиной состояний, где прием / обработка каждого сообщения / команды приводит к выполнению некоторого бизнес-кода и переключению в следующее состояние
    - Хорошо организовывать переключения также на основе отсылки / приема сообщений из очередей. Это позволяет значительно масштабировать нагрузку и обеспечить отказоустойчивость кластера координаторов. Альтернативный процесс – планировщик на основе периодического опроса базы – проще и понятнее, но масса недостатков.
    - При последовательном обогащении функциональными возможностями данный шаблон превращается в BPMN движок
      * там уже есть поддержка логических / бизнес транзакций на уровне примитива языка описания процесса
      * можно реализовать более сложную логику отката, включающую, например, дальнейшее взаимодействие с пользователем: если зарезервировать выбранную комнату в отеле не получилось, то вместо того, чтобы отменять резервацию авиабилета, можно предложить пользотелю выбрать другую комнату
* Важно для реализации
  + Каждая транзакция должна иметь уникальный ИД, который должен фигурировать во всех сообщениях, как прямых, так и компенсаторных
  + Вместо того, чтобы конфигурировать reply очереди для всех компонент можно передавать им их как свойство в сообщении. Таким образом
    - компоненты будут способны участвовать в работе с несколькими координаторами
    - за счет одного сообщения можно будет последователь задействовать несколько компонент. Что, впрочем, делает систему опять запутанной, от чего мы хотим уйти
  + Избегать всеми силами синхронных взаимодействий
    - Любой компонент в любое время может упасть, поэтому лучше pull организация, когда колмпоненты вытаскивают задания для обработки
    - Если компоненту для обработки сообщения нужно синхронно обратиться к источнику данных, то возможно лучше сделать это самостоятельно и отсылать компоненту уже обогащенное данными сообщение
  + Каждую отдельную операцию выполнять с timeout и разумной retry policy

### Стандартные фреймворки

* **Narayana LRA**
  + Разработан в сотрудничестве с Eclipse MicroProfile
  + Основной акцент на предоставления API для координации долговременных процессов в условиях необходимости достижения конечной консистенности состояния и без использования блокировок
  + Предоставляет разработчику централизованный координатор процессов. Каждый участник прцоесса может присоединить к нему, путем выполнения HTTP запроса к координетору, который также предоставляет HTTP API для завершения / отката процесса. LRA предоставляет специальный клиент, облегчающий выполнений HTTP запросов к координатору
  + Состав SAGA в данном случае определяет инициатор процесса посредством вызова методов остальных участников под контролем CDI аннотаций. Имеется также и API для ручной регистрации участников процесс.
  + Чтобы завершить экземпляр SAGA инициатор вызывает REST URL координатора, а тот, в свою очередь, вызывает URL участников.
  + Порядок вызова участников при заврешении процесса фиксированный, таким образом, отдельные участники могут зависеть друг от друга. Вызовы при откате выполняются в обратной последовательности.
  + Кластеризованное решение, состояние координатора может быть реплицировано или распространяться через транзакционный лог. Смерть участника детектируется на основании timeout его вызова, после чего весь экземпляр процесса автоматически откатывается
* **Axon**
  + Платформа для разработки масштабируемых высокопроизводительных приложений
  + Базируется на CORS шаблоне (обсуждается дальше), предусматривающем две шины событий: одну для create / update запросов и другую для query. События хранятся в виде распределенног лога, что обеспечивает консистентность состояния узлов в случае их краха
  + Экземпляры процессов (экземпляры SAGA) реализованы как аггрегаты: satateful бины, которые инициализируются / меняют свое состояние на основании набора событий, относящихся к одной бизнес-сущности. Завершение / откат SAGA в данной идеологии реализуются как специальные события, стандартная логика обработки которых может быть кастомизирована разработчиком
  + Специальная аннотация @Saga помечает агрегат как SAGA бин со стандартными обработчиками сообщений. Также имеются специальные аннотации для того, чтобы отметить событий создающие / завершающие процесс
  + Порядок вызова участников при завершении процесса может быть любой
* **Eventuate.io**
  + Платформа для EDA распределенных приложений
  + Также базируется на CORS шаблоне, промежуточном хранилище события в MySQL базе данных и распределении их по узлам посредством Kafka
  + Специальный тип API (Eventuate Tram Core) для работы с SAGA. Позволяет приложениям расссылать через именованные каналы асинхронные сообщения, являющиейся частью БД транзакции
  + Логика SAGA строится на основании специального программного билдера, позволяющей присоединять участников, а также регистрировать обработчики завершения / прерывания процесса
  + Порядок вызова участников при завершении процесса фиксированный, таким образом, отдельные участники могут зависеть друг от друга. Вызовы при откате выполняются в обратной последовательности.
* **Итого:**
  + Два последних средства тяжеловаты для реализации только SAGA функциональности, но если уж мы их взялись использовать...
  + Narayana LRA может быть прозрачно внедрена в любой приложение в качестве инфраструктуры. В чем-то это напоминает очень легковесный BPMN сервер

### Простейшая реализация

* SAGA реализуем как сущность со следующими свойствами
  + Уникальный идентификатор. Например, наименование основной сущности процесса + ее ИД.
  + Наименование бизнес сервиса, которому передаются события для обработки.
    - Удобно автоматически вызывать на данном сервисе
      * **on<наименование события>** методы – для нормального процессинга
      * **compensate<Наименование события>** методы, принимающие как сами события, так и возвращенные ими на этапе выполнения результаты, для компенсации сбойного процесса
  + Наименования **cancel события**, завершающего обработку SAGA
  + Лог обработанных событий с аргументами и результатами их обработки, которые передаются компенсационным методам.
    - Отличные исходные данные для администрирования и прочих разборов полетов.
* Для инициализации сущности делаем построитель с цепочкой методов, **build()** метод сохраняет экземпляр в базе
* JMS слушатель
  + По начальному событию инициализирует SAGA экземпляр, по всем остальным – достает из базы на основании УИД и применяет событие
  + SAGA экземпляр в свою очередь прокидывает событие дальше, вызывая для события **onXxx** метод на бизнес-сервисе
    - Понятно, что заниматься этим должна не сама сущность, а специальный SagaService, но это уже не принципиальные детали
    - Обработчик события в результате выполнения бизнес операции публикует в очереди сообщение о выполнения следующей операции.
      * Ну или это делает UI, с получеными от пользователя данными
      * Автоматически получаем масштабируемость и отказоустойчивость обработки документов в рамках кластера
    - Обновление SAGA экземпляра, подтверждение полученного сообщения и отсылка следующего осуществляется, как наиболее надежная операция, ПОСЛЕ вызова бизнес-сервиса - **«Best Efforts 1PC»** шаблон
  + В случае ошибки при примении или поступлении cancel события мы просто достаем лог поступивших событий и проигрываем его в обратном порядке вызывая **compensateXxx** методы
* В качестве расширения
  + Дополнительные методы бизнес сервиса для обработки событий: инциализация SAGA / завершение (придется также конфигурировать наименование commit события) / отмена
  + Кроме компенсационных методов можно определить «закрепляющие» методы для поддержки TCC шаблона (см дальше)
  + Перечень допустимых событий, для каждого возможные дальнейшие – простейшая машина состояний. Дальше этот функционал можно расширять и придти к полноценному самописному BPMN движку!
  + Timeout-ы на события и на SAGA процесс в целом
  + Один слушатель per бизнес процесс с выделенным пулом потоков это уже неплохо, гораздо лучше, кстати, чем единственный дефолтный JobExecutor любого BPMN движка. Но совсем хорошо, если создается (queue + pool) per event. В этом случае мы можем для каждого процесса настраивать производительность каждого источника обработки и по числу сообщений в очереди видеть, что происходит.

## CQRS шаблон

* **Command Query Responsibility Segregation** - метод должен быть либо **командой**, выполняющей какое-то действие, либо **запросом**, возвращающим данные, но не одновременно
  + Все методы, возвращающие значения, не должны иметь побочных эффектов. Побочный эффект: невозможность отследить количество вызовов подобных методов
  + Упрощение программы за счет доступ к её состоянию (через запросы) и изменение её состояния (через команды) более понятными
* Отдельные микросервисы системы имеют собственный контекст: полностью владеют своими локальными данными.
  + К сожалению это делает невозможным интеграцию с другими сервисами через общую базу данных. Ну и это, конечно, грубое нарушение инкапсуляции. Хотя если мы пишем распределенный монолит….
  + Однако интегрироваться по данным с другими сервисами как-то надо. Рассылать им адресно изменившиеся данные достаточно неудобно и медленно. К тому же данные им могут понадобиться для работы с каком-нибудь совсем ином разрезе, например, для полнотекстового поиска.
* Соответственно возникает идея: в процессе применения команды на обновления данных рассылать ее (или другое ее представление) broadacast по шине приложения, чтобы каждый из заинтересованных систем могли бы ее применить к собственной реплике совместно используемых данных.
* Обычно реализуется на основе рассылки через службу сообщений, но можно и просто публиковать посредством HTTP ленту команд.
  + Каждый сторонний сервис, поднявшись, прежде всего проверяет номер последней примененной команды, а затем скачивает и применяет к себе все новые.
  + Соответственно, если хранилище сервиса опустили не аккуратно или вообще восстановили из старой копии, то достижение актуального состояния есть только вопрос времени
* Достоинства:
  + Сохраняя команды отдельно всегда можно воссоздать хранилищей или докатить до актуального с сохраненной репликой – порождение событий
  + Каждая система, получив событие, может применить данные из него в той форме, в которой ей это необходимо. Мы даже зачастую просто не можем и предположить как именно
  + В некоторых ситуациях команду быстрее сохранить, чем применить. Соответственно состояние хранилища данных докатываем по мере возможности, но рано или поздно все команды будут применены и хранилище станет актуальным первичному источнику данных
  + Писать можно в одно хранилище, а читать из нескольких других, оптимизированных под необходимые поисковые запросы. При этом процедура чтения будет хорошо масштабироваться. Процедура записи, увы, по-прежнему нет.
  + На основании обработки одних команд можно выбрасывать на шину другие и так далее.
* Недостатки
  + С точки зрения читателя и писателя относительно каждого элемента информации система какое-то время будет в несогласованном состоянии
  + Если масштабирем чтение, то и разные читатели могут получить в один момент разные данные.
  + Бизнес-логика, реализованная подобным образом, становится децентрализованной и слабо пригодной для археологического анализа
* Хранилища, оптимизированные под подобные операции: Kafka, Event Store, Eventuate

### Реализация на основе Axon

* Тесно интегрированное со Spring Boot средство, с течение времени включает в себя все больше составляющих из Spring Cloud
* Основная входная точка: **CommandGateway**, в асинхронном режиме отправляет команды к обработчикам.
  + Обработчики могут быть как локальными, так и на других узлах кластера, организованного на основе **JGroups** или **DiscoverClient**
  + В результате маршрутизации имеем **CompletableFuture**, что очень хорошо сочетается с асинхронными Servlet 3 и экономит входные потоки HTTP контейнера
* Обработчик: любой метод в Spring бине, аннотированный **@CommandHandler** и принимающий аргумент соответствующего типа
  + Он конечно может и сам записать данные в базу, но архитектурно лучше сразу опубликовать сообщение на шине за счет **apply(),** один из слушателей которой сохранит данные
  + Все опубликованные сообщения записываются в хранилище для повторного использования (воссоздания агрегатов)
  + Как правило, обработчики это агрегаты, так как последовательные команды зачастую связаны между собой ЖЦ бизнес-сущности
* Команда это просто класс, один из свойств которых аннотирован **@TargetAggregateIdentifier** для отправки сообщения соответствующему агрегату
* Агрегат – stateful компонент с аннотацией **@Agregate** и свойством, аннотированным **@AggregateIdentifier.** Представление бизнес-сущности в оперативной памяти, реализующее ее ЖЦ в целях правильной интерпретации применяемых команд / событий
  + Если у нас имеется команда для какого-то агрегата (типы и значения атрибутов с **@AggregateIdentifier и @TargetAggregateIdentifier** должны совпадать), то
    - создается бин, его состояние либо восстанавливается из базы, либо на него накатываются все предыдущие команды
    - после чего бин остается в памяти и к нему маршрутизируются все вновь получаемые шлюзом команды
  + За счет этого можно, например, отслеживать ЖЦ сущности, к которой последовательно применяются команды без необходимости выполнять запросы к БД. Например, открытие заказа, добавление элементов (если заказ еще не закрыт), закрытие заказа
  + Типовое поведение при обработке сообщения: проверяем его валидность относительно состояния агрегата (если это первое сообщение в ЖЦ и поле с **@AggregateIdentifier** пусто, то инициализируем идентификатор агрегата и остальные обязательные поля), а затем создаем и публикуем сообщение для всех заинтересованных слушателей. Зачастую сообщение просто повторяет по структуре команду, но использовать **@TargetAggregateIdentifier** нет необходимости – обработка в агрегате уже завершена
* Обработка сообщения:
  + Получается любыми бинами с методами соответствующих типов, аннотированными
    - **@EventSourcingHandler** – с привязкой к соответствующему агрегату
    - **@EventHandler** – просто слушатель без состояния
  + Соответственно нам теперь надо создать бин с методом **@EventHandler**, принимающим аргумент с типом события, который
    - Сохранит данные в БД
    - Разошлет событие по кластеру. Текущая для выпуска версия содержала облегчающие механизмы на основе AMQP, но вроде обещано использования стандартных для Spring универсальных MessageChannel
* Остальные микросервисы, получив сообщение, либо игнорируют их, либо обновляют хранимые реплики интересующих их фрагментов данных

## TCC ( Try-Confirm / Cancel )

* Частая область применения транзакций это «распределенные системы резервирования», например:
  + Зарезервировать билет на самолет и номер в отеле
  + Выбрать место в кинотеатре с оплатой в независимом сервисе
* Общая логика на примере REST
  + Резервируем билет на самолет за счет POST
    - Операция легко делается идемпотентной, можно как на нашей стороне, лучше на стороне сервиса резервации
    - в ответ
      * получаем URL для выполнения confirm / cancel операции
      * билет резервируется за нами на определенный срок
  + Аналогично с номером в отеле
  + В случае благополучного итога делаем PUT на оба ресурса с подтверждением. Если что-то пошло не так, то DELETE с анулированием операции. Обе операции идемпотентны по определению
  + Если мы что-то продолбали, то билеты по истечению срока блокировки автоматически становится доступным для продажи снова
* Дополнительные возможности:
  + На стороне сервиса регистрации лучше отслеживать историю статусов. Например, чтобы начислить штрафы клиентам, которые в массовом порядке «забывают» анулировать резервации
  + GET по полученному URL позволяет получить информацию о резервации, также можно предоставить сервис расширения периода резервации, возможно, тоже за отдельную плату
* В общем это типичная оптимистичная транзакция. А с точки зрения сервиса резервации это BPMN External Task, относительно которого есть тоже множество интересных трюков по реализации

## REST + ACID на коленке

* Иногда увы это все-таки необходимо
* Основной сценарий:
  + Сервис при получении запрос
    - Создает локальную ACID транзакцию и выполняет запрошенную операцию
    - оставляет транзакцию незавершенной и возвращает в составе результатов URL для ее управления (AcidParticipationLink). Данный URL может просто откатывать транзакцию, а может также и выполнять какие-то компенсирующие действия.
  + Координатор
    - Выполняет все операции на коллекции сервисов и получает взамен коллекцию URL
    - Проходится по коллекции и выполняет commit.
      * Возможно сначала запрос подтверждения возможности (на той стороне, например, session.flush() ), а только потом уже commit.
      * Неустранимые конфликты будут увы в любом случае
* Дополнительные возможности:
  + НЕ-завершение транзации вовсе не означает необходимость подвешивать поток, в котором выполнялись операции с ресурсом.С точки зрения Java приложения транзакция это просто уникальный идентификатор, передаваемый при каждом обращении к БД / серверу очередей. Но придется повозиться с ThreadLocal переменными, в которых он хранится:
    - Считывася и очищая их при завершении обработки запроса
    - Выставляя заново из какого-то хранилища или аргументов при поступлении команды на commit / rollback операции
    - Совершенно аналогичные проблемы, когда мы хотим совместить транзакционность с распараллеливанием обработки
  + Если сервис в рамках логической транзакции запрашиваем несколько раз, то можно уже полученный AcidParticipationLink передавать в качестве аргументов, чтобы продолжить обработку данных в уже начатой транзакции
  + Если сервисы при этом в свою очередь начинают вызывать друг друга, то они также могут использовать AcidParticipationLink в качестве одного из аргументов
  + В принципе можно объединять данный подход с SAGA, то есть часть ресурсов блокировать пессимистически (ACID), а часть оптимистически (за счет возможности вызвать компенсирующую операцию)
* Из коробки подобный сценарий поддерживается **Atomikos**
  + Приложение использует встроенные менеджер транзакций, ответственный за
    - начало / закрепление глобальной транзакции, и использующий для этого URL, возвращаемые обновляемые ресурсами
    - начало / закрепление локальной транзакции, формирование возвращаемых URL, обработку поступающих на них запросов со стороны инициатора транзакции
  + Функционал основан на JAR-RX / Spring Rest интерцепторах, прозрачно добавляющих / экстрагирующих служебную информацию из HTTP заголовков
  + Клиентский интерцептор:
    - определяет наличие незаконченной транзакции и добавляет заголовок с URL
    - если запрос, уже содержит заголовок с данными активной транзакции, присодиняет запрос к локальной транзакции
  + Серверный интерцептер
    - Вычленяет и сохраняет URL из результатов запроса к ресурсам
    - Добавляет в запросы к ресурсам транзакционную информацию
    - Выполняет закрепление / откат глобальной транзакции с использованием перечня URL
* Достоинства
  + Прозрачность для реализаций микросервисов
  + Если мы вызываем сервис и НЕ получаем ответа, то мы можем быть уверенными, что данные не закреплены
* Проблемы
  + Что делать, если на финальной стадии некоторые ресурсы все-таки начали отваливаться? Вот тут очень могут пригодиться компенсирующий операции, «подчищающие» за собой данные или наоборот возможность быстренько повторить сбойную операцию, в зависимости от специфики ресурса
  + Долговременные ACID транзакции на ресурсах убивают производительность, соответственно, подходит применяется в слабо нагруженных и локальных системах, в основном на уровне отдельной компании

# Промышленная эксплуатация

* Какие требования обычно стоят
  + Масштабируемость в смысле способности справляться с бизнес требованиями. Увы, это не универсальное понятие и меняется от системы к системе.
    - Если мы не понимаем бизнес системы, то мы не можем предсказать по каким направлениям она будет развиваться и как именно предстоит масштабироваться
  + Способность справляться с неожиданными ситуациями
* Чтобы оценить текущее состояние и что-то спрогнозировать нам нужно заранее вставлять в приложение снимаемые показатели работоспособности. Причем, как системные, так и имеющие бизнес-значение (см книгу Release It! от Pragmatic)
* Завершение процесса разработки приложения еще не означает готовности ее к практической эксплуатации
  + Раньше ПО переваливали на службу эксплуатации в виде более или менее черного ящика и предоставляли ей крутиться как хочет. Недостатки:
    - они не хотят брать неизвестно что без длительного тестирования и вообще хотят что-то менять как можно реже - режется динамичность разработки.
    - Разработчики не хотят тратить время и деньги на функционал, не имеющий непосредственно бизнес-ценности
  + Сейчас идеология DevOps: достижение бизнес результатов есть общая ответственность обоих команд и даже фактическое объединение данных команд.
    - Соответственно, разработчики становятся кровно заинтересованы, чтобы снабдить выпускаемой ПО средствами, облегчающими его эксплуатацию - им же самим разбираться со всем возможным последующим гавном.
    - Особено это становится важным для распределенных систем, где количество точек отказа и путей распространения последствий этого отказа резко возрастает
* Основные моменты, на которые необходимо обратить внимание при вводе приложение в эксплуатацию:
  + Планирование и управление процессами: кто запускает / останавливает приложение, как гарантирует уникальность запущенного на хосте / в кластере
  + Восстановление работоспособности при сбое: как админы могут оценить текущее состояние экземпляра, какие показатели при этом используются, каким образом они могут быть проанализированы в текущий момент / в историческом разрезе, что происходит при его внезапной смерти / отказе хоста
  + Управление логами: как осуществляет формирование / сбор с узлов кластера / первичная обработка, как админы могут использовать лог для анализа происходящего, насколько информация чувствительна и как защищается
  + Управление маршрутизацией: доступно ли оно пользователю, как балансируется нагрузка, перераспределяется ли она при перезапуске компонентов приложения
  + Распределенная трассировка: какие запросы поступают в систему, какие сервисы участвуют в каждой бизнес транзакции, каковы параметры основных запросов (частота, длительность), как выглядят усредненные показатели сейчас и меняют с течением времени

## Диагностика – отслеживаемость состояния системы

* С точки зрения инфраструктуры сбора информации о состоянии два подхода (большинство развитых средств могут использовать оба режима)
  + Извлечение (pull) – инфраструктура сама периодически опрашивает состояние компонент. Ключевой момент: периодичность, если она слишком низкая, то инфраструктура будет не замечать вовремя изменений в компонентах. Поэтому данный подход используется реже.
  + Внедрение (push) – компоненты сами периодически оповещают инфраструктуру о своем состоянии
* На платформе Spring Boot основная технология публикации данных для Spring Boot Actuator, собирающая по всем компонентам приложения информацию и публикующая их в виде множества REST ресурсов

### Spring Actuator – основной источник диагностической информации

* Зависимость **spring-boot-starter-actuator** и **spring-boot-starter-actuator-web** для REST доступа
* Можно настроить также доступ через JMX, чтобы соединять с промышленными системами мониторинга.
  + Свойства **endpoints.jmx.enabled и spring.jmx.enabled** для контроля экспорта API актуатора и вообще разрещения работы с JMX
  + Можно автоматически построить HTTP API работы с всеми JMX бинами за счет **org.jolokia.jolokia-core.\*,** соответственно возможностей гораздо больше, чем просто получение информации из актуатора
* Можно получить возможность соединяться через SSH
  + Возможность обеспечивается за счет встроенной среды на сонове **CraSH**, чтобы ее активировать надо добавить зависимость **spring-boot-starter-remote-shell**
  + Соединяться на порт 2000 как user/<автосгенерированный пароль, доступный через лог>
    - Можно задать фиксированные логин / пароль через свойства **shell.auth.simple.user.\*,** хотя чаще используют **shell.ssh.key-path**
    - Можно разрешить также и доступ через telnet
  + Перечень команд можно расширять самому - **Groovy** класс в пакете **src/main/resources/commands**, **src/main/resources/crash/commands** или бин, расширяющий CrashPlugin
* Типовая кастомизация
  + Переименование точек доступа.В принципе можно за счет уникального сложного значения частично решать проблемы безопасности, но очень частично. Можно сдвинуть всем разом весь контекстактуатора.
  + Разрешение / запрещение всех или отдельных точек доступа
  + Пересаживание на определенный порт или даже определенный сетевой интерфейс
  + Безопасность настраивается стандартным образом

#### Стандартные точки REST доступа

* **/autoconfig -** Provides an auto-configuration report describing what auto configuration conditions passed (раздел **positiveMatches**) and failed (раздел **negativeMatches**). Используется чтобы понять какие в итоге бины вошли / не вошли в конфигурацию и почему
* **/configprops** - Describes how beans have been injected with configuration properties (including default values).
* **/beans** - Describes all beans in the application context and their relationship to each other. Каждое описание состоит из
* **/dump -** Retrieves a snapshot dump of thread activity, а также для каждой нити перечни блокировок и точек синхронизации.
* **/env** - Retrieves all environment properties
  + дефолтные, из внутренних конфигов, внешних конфигов, аргументов командной строки, переменных окружения, системных переменных JVM, параметров сервлетного контейнера и так далее, в общем все зарегистрированные в системе **PropertySource** источники
  + значения ключей, включающих слова типа password, secret, key заменяются на «\*», остальные выводятся как есть
  + Хорошая возможность понять какие вообще параметры приложения могут быть конфигурированы
* **/env/{name}** - Retrieves a specific environment value by name.
* **/health -** Reports health metrics for the application
* **/info** - Retrieves custom information about the application
* **/mappings** - Describes all URI paths and how they’re mapped to controllers (including Actuator endpoints).
* **/metrics** – доступ к набору показателей, собираемым по всем доступным компонентам
  + **Garbage - collector gc.\*** - The count of garbage collections that have occurred and the elapsed garbage collection time for both the mark-sweep and scavenge garbage collectors (from **java.lang.management.GarbageCollectorMXBean**)
  + **Memory - mem.\*** The amount of memory allotted to the application and the amount of memory that is free (from **java.lang.Runtime**)
  + **Heap - heap.\*** The current memory usage (from **java.lang.management.MemoryUsage**)
  + **Class loader - classes.\*** The number of classes that have been loaded and unloaded by the JVM class loader (from **java.lang.management.ClassLoadingMXBean**)
  + **System** - processors.uptime instance.uptime systemload.average System information such as the number of processors (from **java.lang.Runtime**), uptime (from **java.lang.management.RuntimeMXBean**), and average system load (from **java.lang.management.OperatingSystemMXBean**)
  + **Thread pool - threads.\*** The number of threads, daemon threads, and the peak count of threads since the JVM started (from **java.lang.management.ThreadMXBean**)
  + **Data source - datasource.\*** The number of data source connections (from the data source’s metadata and only available if there are one or more **DataSource** beans in the Spring application context)
  + **Tomcat sessions - httpsessions.\*** The active and maximum number of sessions in Tomcat (from the embedded Tomcat bean and only available if the application is served via an embedded Tomcat server)
  + **HTTP - counter.status.\* gauge.response.\*** Various gauges and counters for HTTP requests that the application has served
* **/metrics/{name}** - Reports an individual application metric by name.
* **/trace** - Provides basic trace information (method, path, timestamp, headers, and so on) for HTTP requests.
* **/auditevents –** данные аудита
* **/docs –** документация для входных точек актуатора, для активизации добавить spring-boot-actuator-docs
* **/loggers –** текущая конфигурация логирования и возможность ее поменять на ходу
* Специфичные для добавленных в приложение расширений, например
  + **/flyway|liquivase –** лог последней примененной процедуры миграции структуры БД во время старта приложения
* **GET/logfile –** содержимое, доступно если определены переменные **logging.file или logging.path**, поддерживается параметр Range для листания

#### Формирование показателей

* Основные разновидности:
  + Счетчики – за один раз записываем некоторую разницу, например, «плюс один запрос», а система сама обновляет текущее значение показателя
  + Датчики – записываем непосредственно текущее значение показателя, например, «в данный момент в системе залогинено ХХ пользователей»
* Работа со счетчиками
  + В свой сервис вставляем **CounterService** и за счет следующих методов накапливаем информацию «по-тикам»
    - **void increment(String metricName);**
    - **void decrement(String metricName);**
    - **void reset(String metricName);**
* Работа с датчиками
  + Или периодически сами **GaugeService. submit(String metricName, double value)**
  + Илибин реализующий интерфейс **PublicMetrics** с единственным методом **Collection<Metric<?>> metrics().**
    - Будет дергаться при каждом запросе статистики, поэтому надо аккуратно полдходить к мультипоточности и кэшированию
    - Все основные показатели собираются от компонент через реализации данного интерфейса. Например, **SystemPublicsMetrics**.

#### Аггрегация показателей

* Все это замечательно, но как правило интерес представляет не текущее значение показателя, а результат его некоторой статистической обработки
* Помогает библиотека **Dropwizard Metrics**, зависимость **io.dropwizard.metrics:metrics-core**
* К нашим счетчика добавляем префикс **«meter.»**, после чего они начинают промежуточно обрабатываться данной библиотекой.
  + Вычисляются:
    - Средняя интенсивность поступления событий, экспоненциально-взвешенные скользящие средние в 1/5/15 минутных интервалах и так далее
    - Полного хранения накопленных данных для вычисления не производится, так что проблем с памятью не имеем
  + В результате в **/metrics**  появляются множественные варианты нашего свойства с постфиксами **meanRate / fiveMinuteRate** и так далее
* Если к датчикам добавляем префикс **«histogram.»,** то аналогично начинает считаться максимум / минимум / среднее / медиана / процентили
* Также библиотека предоставляет таймеры с дополнительной статистической обработкой:
  + Класс **StopWatch** с методами **start() / stop(),** обычно используем по обе стороны вызова внешнего сервиса
  + Результат **getLastTaskTimeMillis()** отправляем в **GaugeService** с наименованием, начинающимся на **«timer.»**

#### Сбор и централизованная обработка показателей

* Данные по одиночным экземплярам сервисов вещь интересная, но не не слишком полезная в контексте всей системы: надо где-то агрегировать, а также связывать данные по вложенным вызовам
* Основное средство: Time Series Database (TSDB), оптимизированная под сбор, анализ и визуализацию систем показателей, изменяющихся с течением времени
  + Популярные: Ganglia, Graphite, Open TSDB, InfluxDB, Prometheus
  + Обычно используется в тандеме с системой визуализирования. Например, Graphana или Graphite
  + Данные представляют себе пары ключ / значения, привязанные к определенному времени
* Для экспорта накопленных данных из систем на основе Spring применяется реализация интерфейса MetricsWriter с аннотацией @ExportMetricsWriter
  + Из коробки можно публиковать в Redis, JMX или по стандартному протоколу StatsD - StatsMetricWriter в различные системы
  + StatsD изначально был создан специально для выбрасывания данных в Graphite / Carbon, но потом стал популярен и теперь его используют InfluxDB, TSDB, Ganglia
  + Dropwizard также поддерживает публикацию данных для анализа через \*Reporter классы. Например, посредством бина GraphiteReporter, сконфигурированного URL базы и ссылкой на MetricsRegistry – источник данных. Созданный reporter конфигурируется периодичность исполнения, запускается и затем публикуется как бин.
* Некоторые базы позволяют привязывать к значениям ключа различные метаданные, но это не общее свойство. Поэтому ключ должен иметь иерархическое устройство, позволяющее посредством группировки по его фрагментам получать

#### Общая информация об экземпляре сервиса - /info

* В отличии от показателей данная информация обновляется достаточно редко и служит в основном в справояных целях: кокая версия кода запущена, когда он был собран, какой коммит в GIT привел к развертыванию данной версии сервиса и так далее
* Автоматически туда попадают любые свойства, начинающиеся с info.\* вне зависимости от места определения
  + Можно, например, включить плагин pl.project13.maven:git-commit-id-plugin (уже сконфигурирован в родительском профиле), который создаст файлы git.branch и git.commit, свойства из которых пойдут в /info
  + Ну или сконфигурировать фильтрацию \*.properties файлов при сборке, чтобы задействовать стандартные maven переменные типа project.artifactId / project.version за счет @...@ шаблонов

#### Текущее состояние сервиса и отдельных его компонент - /health

* Например, может использоваться балансировщиком для определения жизнеспособности данного узла кластера
* За счет конечной точки HealthEndpoint, собирающей информацию со всех реализаций HealthIndicator.
  + Как правило, каждый инфраструктурный модуль уже несет в себе подобный поставщик. Например для Java Mail, MongoDB, Cassandra, JDBC, Redis и так далее.
  + Цель – предоставить информацию о жизнеспособности как самого сервера, так и всех важных для него зависимостей
  + Кроме собственно status = UP | DOWN может выводиться и любая дополнительная информация, например, количество доступного пространства для файловой системы
* Реализация HealthIndicator представляет собой бин с единственным методом **Health health()**
* Внутри за счет билдера **Health.Builder** создаем индикатор
  + **builder.up().build() или builder.down(),** затем **build()**
  + Есть методо **withDetail()** позволяющий добавить дополнительную информацию
* Можно в одном бине вернуть множество индикаторов
  + Получаем доступ к системному **HealthAggregator** или создаем свой на основе **AbstractHealthAggregator**, если системный **OrderedHealthAggregator** с принципом агрегации «использовать последнее значение» не подходит
  + За счет него создаем **CompositeHealthIndicator**, который заполняем данными и возвращаем
  + Содержание индикаторов кэшируется. По умолчанию секунда, можно поменять через **endpoints.health.time-to-live**
* Подобная «лобовая» реализация связывает собственно индикатор с кодом системы, за счет опроса его сотояние, что зачастую нежелательно
  + В реализации индикатора посредством @EventListener на методе с параметров – типом сообщения осуществляем подписку на сообщение, меняющее его состояние и публикующее дополнительную информацию
  + Во всех остальных местах при помощи ApplicationEventPublisher.publishEvent() высылаем индикатору сообщение
  + Приятный побочный эффект – выполнение прикладного / инфраструктурного кода при этом не замедляется

#### Публикация событий аудита - /auditEvents

* Например, события безопасности (аутентификации и авторизации) по умолчанию и так аудируются, чтобы подписаться на получение
  + реализовать **AbstractAuthenticationAuditListener** или **AbstractAuthorizaionAuditListener**
  + или @EventListener на методе, принимающем AuditApplicationEvent
* Чтобы использовать данный механизм для собственных событий:
  + Публикуем через **ApplicationEventPublisher**, поставляемый интерфейсом **ApplicationEventPublisherAware** или за счет @Autowired
  + Они накапливаются для показа в реализации **AuditEventRepository.** Дефолная реализация в памяти, поэтому рекомендуется предоставлять собственную (отдельное хранилище, не-транзакционная асинхронная запись, append-only доступ и так далее)

### Системные средства диагностики

* Позволяют дагносцировать проблемы на низком системном уровне за счет анализа взаимодействия между компонентами приложения. Обычно за счет java агентов или инструментирования кода.
* Коммерческие – New Relic или App Dynamics.
  + Одно из приемуществ – совместимость с Pivotal Cloud Fiundry и другими распространенными облачными инфрастуктура «из коробки», позволяя анализировать изменения в поведении компонент на низком уровне в контексте выполняемых бизнес операций, что особенно важно
* Неплохой обзор - <https://habr.com/ru/company/pc-administrator/blog/304356/>

#### Java Melody

* Легко встраивается в приложение за счет добавление нескольких зависимостей и сервлета для генерации GUI
* Позволяет получать за счет веб GUI и генерируемых PDF отчетов множество системной информации, в частности:
  + статистика http запросов;
  + статистика обращений в базу данных с детализаций от HTTP запроса до плана SQL запроса к базе
  + срабатывание сборщика мусора (garbage-collector);
  + количество потоков;
  + статистика использования памяти;
  + количество транзакций в базу данных;
  + ошибки в логах.
  + Статистику по вызовам на границах бинов
  + Выявление «горячик» методов за счет периодического сохранения thread dump
  + Статистика по JSF / JSP / Servlet / Struts
  + Анализ работы Quartz
* Имеется возможность собирать аггрегированную статистику с нескольких узлов кластера и анализировать вместе
* Практически не замедляет работу приложения, поэтому может быть использована в production режиме
* В основном полезна для приложений на основе Spring, но для CDI based тоже может быть использована, хотя и с меньшими возможностями
* Куча плагинов для анализа поведения различных инфраструктурных компонентов. Можно добавлять собственные

#### Glowroot

* Примерно то же самое, но посвежее и больше ориентированное на JEE мир
* Основные возможности:
  + Захват трассировки для медленных запросов и ошибок.
  + Непрерывное профилирование (с очень удобной фильтрацией).
  + Временные диаграммы времени отклика. Временные диаграммы процентилей.
  + Захват и агрегация SQL.
  + Захват и агрегация вызовов компонент.
  + Запись атрибутов и диаграмм. Атрибуты по процентилям времени ответа и атрибутам MBean.
  + Исторический накопитель всех данных (1m, 5m, 30m, 4h) с настраиваемой задержкой.
  + Полная поддержка для асинхронных запросов, которые охватывают несколько потоков.
  + Центральный центральный коллектор / центральный интерфейс.
* Поддержка дополнительных технологий: Apache Camel, Cassandra (DataStax client), EJB, Elasticsearch client, Java Executors, Grails, Hibernate, различные HTTP clients, JavaMail, JAX-RS, JAX-WS, Jdbc, JMS, JSF, JSP, Kafka, различные Loggers , MongoDB, Netty (HTTP codec) , Play Framework (1 and 2) , Quartz Scheduler , Redis (Jedis client) , Servlets , Spring Framework ,Vert.x

## Логирование

* так как имеем дело с распределенной и зачастую асинхронной системой, то попытка debug-а превращается в кошмар, потому что одну логическую операцию приходится отслеживать по нескольким сервисам / машинам / потокам / хранилищам данных
* В высоко конкурентной асинхронной среде возможность поставить break point зачастую бессмысленна, так как пауза вносит существенное искажение в исследуемый процесс. Поэтому проблемы типа race condition могут быть проанализированы только по косвенным следам
* Соответственно единственный вариант это работа с логами, при условии, что у нас есть
  + **correlationID** связывает воедино все строчки лога, относящиеся к одной логческой транзакции
  + логи автоматически собираются из всех мест в одно хранилище, преобразуются в вид, пригодный для анализа, и индексируются для быстрого поиска
    - хотя встречал уникумов, которые годами ручками последовательно смотрели во все доступные логи в поисках интересующего фрагмента
    - очень частичное облегчение для сбора и просмотра логов с нескольких серверов - <https://github.com/pschweitz/insidelog>
  + поток запросов, составляющих единую транзакцию, легко вычленяется и визуализируется с целью поиска проблемных мест
* Для этого необходимы три инфраструктурные компоненты: сборщик логов, хранилище и агрегатор логов и средство визуального анализа потока запросов в составе транзакции
* Для микросервисов инфраструктура работы с логами является критичной и должна разрабатываться с самого начала параллельно с основной системой

### Формирование логов

* Сам Spring Boot использует Commons Logging, оставляя возможность выбора поставщика сервиса за приложением.
  + Поставляются дефолтные конфигурации для **Java Util Loggin, Log4J2, Logback.**
  + Для каждой конфигурации включен дефолтный вывод на консоль с возможностью быстрого подключения файла через свойства **logging.file / logging.path** с ротацией через каждые 10Мб вывода.
    - В облачном окружении вывод в файл как правило не применяется вовсе – доставка лога из STDOUT до продвинутый средств анализа это задача инфраструктуры. А файлы логов имеют неприятную особенность выжирать все место в контейнере
    - Однако для локальной разработки / запуска часто применяется
  + Обычно для приложения при запуске определяют файл с настройками логгирования через **logging.config**
  + Дополнительные возможности: **logging.pattern.console, logging.pattern.file, logging.pattern.level**
  + Чтобы увидеть текущуюконфигурацию логгеров или ее изменить за счет POST используется **/loggers** точка доступа актуатора. Меняется она таким образом, соответственно, только на данном узле.
    - Поэтому если хочется выполнить изменение во всем кластере, то придется перебрать все узлы.
    - Среды Pivotal Web Services / Pivotal Cloud Foundry предоставляют соответствующую функциональность из коробки
* Внедрение **correlationID** и другой дополнительной информации выполняется **Spring Cloud Sleuth,**который релизован на основе HTTP фильтров и прокси обвязок над остальными Spring компонентами
* Собственно раньше мы уже занимались пересаживанием **correlationID** ис потока в поток, а также генерацией его в фильтре Zuul и возвратом клиенту. Sleuth делает нечто подобное, но более комплексно
  + Генерируя **correletionID** за счет Zuul
  + передавая в составе **Spring Mapped Diagnostic Context (MDC)** по всей системе и пересаживая во все выполняемые HTTP / MQ / DB / Kafka и так далее вызовы.
    - Под каждую технологию реализация состоит из перехватчика вызова и экземпляра **SpanInjector**, который добавляет метаданные в описание текущей транзакции
    - Увы, JMS из коробки не поддерживается и каждый получаетель сообщения генерирует **correlationID** заново
  + автоматически подмешивая во все логируемые сообщения, если они выполняются за счет Spring Boot инфраструктуры (SL4J или Logback имплементация без разницы)
  + опционально выбрасывая в средство анализа Zipkin
* Зависимость **spring-cloud-starter-sleuth и spring-cloud-sleuth-core** (опциональная, позволяющая работать с API Sleuth)
* Терминология:
  + **traceId** – то же самое, что **correlationId** – идентификатор глобальной транзакции (трассы)
  + **spanId** – идентификатор части транзакции, которая относится к одному сервису (промежуток).
  + Описание промежутка может сопровождаться дополнительными тегами / , например, идентификаторами затрагиваемых бизнес-объектов
  + Если выполнение запроса к системе осуществляется за счет последовательного вызова двух компонент (A-> B, B->DB), то мы будем иметь одну глобальную транзакцию из двух частей. Для первой метаданными будут характеристики HTTP запроса, для второго – соответственно SQL
* В типичной строке лога мы можем видеть 4 поля: **<application name>, < traceId >, < spanId >, <была ли строка выброшена в Zipkin для анализа>**
* Небольшая проблема: разработчики Sleuth считают, что **correlationId** не должен покидать систему, что зачастую крайне не удобно, так как клиенты не могут идентифицировать ответ, процесс получение которого они бы хотели проанализировать
  + При сборе Zuul добавляем **spring-cloud-starter-sleuth**
  + Делаем post-фильтр, в нем
    - **@Autowired Tracer –** доступ к контексту запроса
    - В ответ добавляем заголовок (обычно **tmx-correlation-id**), который читаем из **tracer.getCurrentSpan().traceIdString()**
* Если обрабатывать все запросы, то нагрузка на систему может быть слишком высока, а собираемая информация попросту избыточна.
  + Например, нас может интересовать только подробные данные по сбойным запросам, некоторой части всех запросов или только относящимся к тестовым версиям компонент
  + За счет интерфейса **Sampler** с методом **Boolean isSampled(Span)** можно управлять отбором запросов для анализа

### Аггрегация логов

* Инфраструктура сборки, хранения и анализа логов достаточно сложная, подразумевающая возможность обработки, хранения и анализа очень больших объемов данных. Поэтому зачастую ее используют как услугу, предоставляемую облачным провайдером.
  + То есть наша система генерит логи, передает куда-то для хранения, а мы пользуемся веб интерфейсом для анализа.
  + С одной стороны, наши данные куда-то утекают и поэтому требуется изрядный upload канал в интернет. С другой, от нас требуется гораздо меньше усилий для начала эксплуатации.
* Важно: с этого момента начинаем рассматривать записи лога в качестве событий. Поэтому они должны иметь структурированную форму, допускающую выделение из них данных для анализа
  + Частая проблема: одно событие представляется несколькоими строчками лога – крайне неудобно для разбора агрегатором
  + Соответственно баланс между читабельностью и структурным видом в идеале контролируемым формальной схемой
* Наиболее распространенные решения
  + **ELK** = Elasticsearch (хранение и поиск на Lucene) + Logstash (сбор логов) + Kibana (веб интерфейс) + Grafana (витрины данных)
    - open source решение, есть коммерческая версия, локальная установка
    - наиболее гибкое, но и наиболее трудоемкое в боевом развертывании (дефолтный вариант ставится мгновенно, но его надо вдумчиво тюнить под большую нагрузку) и поддержке. В общем-то, это конечно больше конструктор с развитой экосистемой, нежели готовое решение.
    - Основное достоинство – его используют все, а потому много руководств по любому тюнингу. Качественная документация и много книг.
    - Elasticsearch с плагинами это очень функциональное решение в том числе, что немаловажно, поддерживающее SQL. Нет необходимости знать о структуре получаемых данных до начала из сбора. Можно начать агрегировать события как они есть, а затем по мере понимания задач задавать структуру не теряя обратную совместимость
    - Основной недостаток – частые релизы ломают совместимость между основными компонентами и между компонентами и дополнениями. Ну и частичная платность (Z-Pack для Elasticsearch ), причем купить можно только большими кусками, а не только необходимые фичи.
  + **Graylog** - open source решение, есть коммерческая версия, локальная установка, достаточно похоже на предыдущий случай, но гораздо легче развернуть «из коробки». По степени готовности напоминает Splunk. Тоже поддерживается logstash, веб интерфейс, разграничение доступа, уведомления и так далее.
  + **Splunk** – только коммерческая версия (цена зависит от объема обрабатываемых данных), есть локальный и облачные варианты в 10 регионах AWS. Старейшее и наиболее функционально богатое решение, обработка логов из множества различных по типу источников, мощная экосистема дополнений. Есть пробный доступ 15 гб в течении 15 дней. Сложный и не всегда интуитивный интерфейс. Полностью закрытое решение.
  + **SumoLogic** – freemium / tiered pricing версии, только облачное решение. На сервера ставится агент, который передает логи в облако, причем не только из микросервисов, но из MongoDB, Docker, Linux / Windows, Nginx, Apache, MySql и так далее. В бесплатной версии 500 мб в день с 7 дневным хранением данных.
  + **Papertrail** – аналогично, собирает логи из syslog, text, apache, mysql, heroku, журналов событий windows, маршрутизаторов, брандмауеров, RubyOnRails и так далее. Доступ к данным может осуществляться из браузера, командной строки, API или с помощи оповещений по HTTP / почте или Slack.
  + **Logz.io** – предоставляет ELK как услугу, тоже сбор данных со всего чего только можно, бесплатная версия 3Гб с хранением 3 дня
  + **Timber** – тоже облако, библиотеки для Elixir / Node / Ruby / JS / Python, платформы AWS / heroku / Docker / Kubernetes, источники данных Filebeat / Fluentd / Syslog / Logstash, различные операционные системы, протоколы HTTP / UPD. До 1 гб и хранения логов в течение дня бесплатно, потом гибкая политика ценообразования. Куча всяких средств анализа, оповещения, достки с диаграммами и так далее
  + **Logentries** – аналогично, 5 гб в течении 7 дней хранения бесплатно
  + Облачные платформы как правило предоставляют собственные сервисы агрегации, например, **Cloud Foundry Loggregator.** К числу достоинств последнего относится возможность приема сообщений через WebSocket
  + **SyslogD** – это и система сбора и протокол, который поддерживается многими другими системами, например, **Cloud Foundry Loggregator**
* Важная функциональная возможность подобных систем: отображение событий лога и значений показателей в едином контексте
* Использование Papertail
  + Создаем аккаунт в системе и конфигурируем syslog коннектор, который будет принимать от нас данные
    - В результате мы имеем URL, который принимает syslog трафик
  + Собираем Logspout Docker контейнер, который будет принимать логи со контейнеров всех наших сервисов и отправлять в Papertail
    - Для этого он читает данные из docker.sock стандартного unix сокета, предоставляемого Docker инфраструктурой, и высылает в облако через syslog
    - Обязательная операция: в docker-compose.yaml свойство command надо определить в URL приема syslog
    - Основные достоинства Logsoupt в отличии от docker log driver
      * Можно высылать данные одновременно в несколько систем, например, в средство анализа и в мониторинг безопасности, проверяющий, чтобы чего лишнего не выслали
      * Можно конфигурировать сбор и фильтрацию централизованно, а не в каждом контейнере индивидуально
      * Можно высылать данные через разные протоколы: все через syslog в Papertail и фильтрованные через HTTP в домашнюю систему индексации. Также доступны UDP / TCP протоколы
  + Конфигурируем сбор логов в Docker контейнерах остальных сервисов и Zuul
    - Тут ничего делать не надо, кроме stdout логи автоматом отправляются в docker.sock, откуда их подбирает Logspout для дальнейшей обработки

### Визуализация логов

* Выполняется с помошью **Open Zipkin,** который помогает визуализировать процесс выполнения транзакции, проходящей через множественные сервисы.
* Дает только достаточно грубые статистические оценки основных этапов выполнения, поэтому не может полноценно выступать средством Application Performance Management (APM), позволяющим совместно анализировать данные по загрузке CPU, объему памяти, I/O и так далее, а также анализировать этапы выполнения в контексте отдельных программных компонент, составляющих микросервисы. Для этого могут подойти средства типа, например, Java Melody.
* Инструментирование сервисов и Zuul
  + Зависимости **spring-cloud-starter-sleuth + spring-cloud-sleuth-core, spring-cloud-sleuth-zippkin**
  + Свойство конфигурации **spring.zipkin.baseUrl** для отправки данных через HTTP
    - В принципе можно еще через RabbitMQ или Kafka, что гораздо удобнее при большой нагрузке, а также позволяет при необходимости перезгружать сервер
    - Для эксплуатации можно просто передавать переменную окружения **ZIPLIN\_URI** или воспользоваться средством Service Discovery для обнаружения сервера
  + Какой процент запросов отправлять на Zipkin
    - Свойство **spring.gleuth.sampler.percentage** между 0 и 1
    - Или публикуем как бин не дефолтный **Sampler**, а **AlwaysSampler**
  + Сбор статистики на кастомных вызовах.
    - Например, обращение к БД. В принципе легко пишется абстрактный интерцептор, который будет собирать статистику на границе всех Spring компонент
    - **@Autowired Tracer**
    - Перед выполнением фрагмента кода: **tracerCreateSpan(name)**
    - После, обязательно в **finally** методе для предотврашения утечки ресурсов
      * **span.tag(“peer.service”, “callType”)**
      * **span.logEvent(Span.CLIENT\_RECV)** – вычисление времени окончания вызова
      * **tracer.close(span)** – очищаем ресурсы
* Zippkin сервер
  + Зависимости **io.zipkin.java.zipkin-server и io.zipkin.java.zipkin-autoconfigure-ui**
  + Две аннотации на конфигурации
    - **@EnableZipkinServer** – это аннотация самого zipkin, простейший вариант развертывания с приемом информации через HTTP
    - **@ EnableZipkinStreamServer** – аннотация SpringCloud Sleuth, позволяет использовать Zipkin с RabbitMQ и Kafka
  + Как кранить данные
    - По умолчанию в памяти, что допустимо только для тестового использования. После рестарта контейнера все данные соответственно утрачиваются
    - Имеются реализации на основе MySQL, Cassandra, Elasticsearch
* Анализ результатов
  + Для Zuul будет показывать на каждый внешний запрос два локальных вызова, так как он не перенаправляет запрос, а выполняет собственный и пересаживает результаты в исходный
  + Операции отправки исходящих сообщений видны в контексте бизнес транзакции, а вот получения и дальнейшей обработки нет – traceId автоматом не пересаживается и входящие сообщения появляются со свежим traceId. Лечится только ручками. Ну или можно выстроить операции в порядке очередности, что для трекинга единичных транзакций дает вполне понятный результат.
  + Также по умолчанию вложенные операции, не связанные с HTTP / сообщениями вообще не видны, если эти вызовы специально не оборачиваются для сбора статистики
  + Так как zipkin знает как сервисы вызывают друг друга, то он также предоставляет данные по топологии системы
* В будующем ожидается, что Sleuth будет взаимодействовать также со стандартом Open Tracing, позволяющим определять общие словари метаданных для различных платформ и языков

### Дополнительные элементы лога

* ИД пользовательской «сессии»
  + У нас stateless система, но каждый сеанс работы конечного клиента тем не менее может быть идентифицирован отдельно, потому что клиент авторизуется.
  + Соответственно, если этот токен достаточно быстро «протухает», то это вполне себе идентификатор конкретной сессии, к которому могут прявязываться те или иные события
  + То есть это еще один УИД уровнем выше traceId
* Информация об исключении
  + Для каждого исключения генерируем и сохраняем в логе вместе со стеком - UUID + CRC( содержимое стенка исключения из которого убираются инфраструктурные и динамические фреймы)
  + Пользователю для обращения в службу поддержики выдаем UUID:CRC
  + Сотрудники по CRC сразу видят с каким типовым случаем проблемы они имеют дело, система анализа лога позволяет по CRC оценить частоту встречаемости, корреляцию с бизнес-данными / типами клиентов и так далее
* Изменяемая подробность логирования
  + При аутентификации пользователя кладем в контекст и дальше передаем по системе уровень логирования, позволяющий для некоорых клиентов писать в лог существенно больше данных. Вплоть до полного лога содержания всех запросов / ответов всех сервисов, а также промежуточных аргументов, передаваемых внутри сервисов между отдельными компонентыми
  + Может включаться:
    - Службой поддержки при разборе обращения клиента, постоянной получающего те или иные ошибки
    - Автоматически для клиентов, ответвляемых на тестируемый перед вводом в эксплуатацию функционал
* Подробное логирование ошибочных запросов
  + Для каждой глобальной транзакции логируем данные с максимальной подробностью во временном расшареном быстром хранилище (REDIS) или индивидуально на каждом сервисе в памяти (сложнее, как «дальние» сервисы узнают о необходимости удалить / выслать собранные данные)
  + Если транзакция завершилась нормально, то отправляем в сервис аггрегации логов минимально необходимый объем данных. Если с ошибкой – все собранные данные со всех задействованных в транзакции сервисов с максимальной подробностью
* Исключение чувствительных данных
  + Используется, если в лог надо включить чувствительные данные (например, полные дамп запросов / ответов к сервису с персональной информацией)
  + Проще всего:
    - заменяем «личные данные» на «типа данных»-«CRC от чувствительных данных + соль из данных самого пользователя»
    - недостатки: требует написания отдельного обработчика для каждого типа данных, зная наименование и способ вычисления CRC можно отследить примерную историю работы, выполненные операции и даже попытаться восстановить наиболее вероятные аргументы
  + Правильно, но трудоемко и требует больше ресурсов
    - Аналогично, но данные выкидываем через Security Logger в отдельное защищенное хранилище, а в лог пишем полученный оттуда УИД
    - Перед сбрасыванием лога обратно в локальную защищенную систему агрегации логов производим обратную операцию

## Администрирование

* **Spring Boot Admin** – проект от компании Codecentric
  + Предоставляет общий взгляд на систему, позволяя проваливаться до конкретных экземпляров сервисов, поставляющих информацию посредством Spring Actuator
  + Сервер строится аналогично: завиисимости + @EnableAdminServer + @EnableDiscoveryClient, чтобы получать данные о доступных узлах
  + Сервисы поставляют информацию серверу администрирования
    - Всем сервисам просто добавляем зависимость и они при помощи DiscoveryClient получают доступ к серверу администрирования для публикации данных.
    - Или определяем у них в конфигурации spring.boot.admin.url, поставляемый как переменная окружения
  + Чтобы открыть сервисы для управления со стороны сервиса администрирования в простейшем случае устанавливаем им **management.endpoints.web.exposure.include = “\*”**
* Аналог - **Ordina Microservices Dashboard,** поволяет получить совокупное представление системы на следующих уровнях
  + UI компоненты - например, директивы Angular
  + Resources- информация, извлекаемая их /mappings сервисов
  + Microservices – информациях об экземплярах на основе Spring Boot или других технологиях, если есть соответствующие адаптеры
  + Backends – все третьи серверы, от которых зависят микросервисы,информация о которых получается из /health раздела актуатора
* AppManager от Pivotal Cloud Foundry

# Развертывание

* Создание полностью автоматизированной процедуры сборки, развертывания и тестирования всего приложения от каждого коммита через все среды (dev / test / release / prod) это сложная, но одна из самых важных частей архитектуры проекта, основанного на микросервисах
  + Скорость процедуры – ключевое требование, обеспечивающее гибкость системы и возможность опертативной обратной связи, как от клиентов, так и от заказчиков
  + Время реализации нового или фикса бага должно измеряться минутами, максимум часами, а не днями
* Соответственно основные требования к процедуре развертывания
  + **Полная автоматизация** – после коммита разработчика никаких ручных вмешательств. Разве что финальное одобрение перед выкладкой в эксплуатацию.
  + **Воспроизводимость** – каждый раз выполняется одна и та же последовательность действий. Все зависимости (в том числе от всех вспомогательных утилит. генераторов кода, например) фиксируются на неизменяемых релизах компонент. Сборка системы из любого состояния CVS каждый раз должны преводить к одним и тем же конечным артефактам. Даже если мы решили собрать самый первый релиз системы или состояние на «начало времен».
  + **Завершенность** – на выходе готоый к эксплуатации артефакт, не требующий выполнения над ним никаких ручных операций. Либо образ виртуальной машины, либо Docker контейнер. Процедура сборки данного артефакта также разумеется должна быть полностью автотизирована, а скрипты лежать под управление системы контроля версий, как и любой другой исходный код
  + **Неизменяемость** – после сборки артефакты ни в коем случае, ни при каких обстоятельствах не должны изменяться вручную. Если что-то нужно поправить – меняем в исходных кодах / скриптах, коммитим и ждем результата. Необходимые runtime параметры (настройка GC, активные профили Spring Boot, адреса серверов инфраструктуры и так далее) должны поставляться запускаемым образам извне, через переменные окружения. Разумется, они должны также храниться под CVS и их «доставка» также должны быть автоматизирована
* К сожалению Spring Clound / Boot это фреймворк для разработки и на данном месте никаких средств автоматизиации нам не представляет
* Интересное средство – Concourse, позволяет создавать короткоживущие среды для сборки / тестирования / развертывания на основе конвееров задач, запускаемых в динамически запускаемых Docker контейнерах. Имеет java коннектор, CLI клиента и веб интерфейс.

## Развертывание в Amazon Cloud

* Минимальный инструментарий:
  + **Amazon Web Services (AWS)** аккаунт
  + **Amazon AWS console** - веб-консоль для создания и конфигурирования серверов. В принципе через нее можно делать вообще все, но не удобно
  + **Amazon ECS command line client** – во-первых, удобнее, во-вторых, средство автоматизации работы с Docker образами.
    - Предоставляет усеченный набор команд, поэтому необходимо использовать SSH, чтобыиметь возможность зайти и посмотреть ручками
    - Соответственно заранее генерируем и прикапываем пару ключей
  + **Amazon Cloud Formation scripring DSL** для автоматизации всех задач, связанных с управлением Amazon инфраструктурой
    - Хороший аналог это **HashiCorp TerraForm**

### Конфигурирование Amazon инфрастуктуры

* Можно как ручками (для первого раза), так и за счет Cloud Formation DSL, что является основным вариантом
* Убираем из приложения Docker обраы инфраструктурных серверов, которые предодоставляются Amazon, и конфигурируем посредством веб-консоли
  + Вместо PostgreSQL – Amazon RDS
    - Поддеживаются различные типы баз, например, классический MySQL
    - Минималка это db.t2.micro (1 vCPU, 1 Gb RAM. 5 GB SSD storage)
    - В результате процесса конфигурирования получаем URL для подключения
  + Вместо Redis – Amazon ElastiCache
    - Есть также MemCache, оба только в памяти
    - Минималка это cache.t2.micro (0.5 Gb памяти)
    - В результате процесса конфигурирования получаем URL для подключения
  + Для нужд разработки есть бесплатные конфигурации серверов, соответственно используем plan = Dev/Test
* Чтобы запускать Docker остальные образы - single-node Amazon ECS cluster
  + Конфигурируется также через веб-консоль, раздел Amazon EC2 Container Service Link, лчше сразу прервать визар и вместо этого исползовать «create cluster» кнопку
  + Инфраструктура Amazon будет сама контролировать состояние запущенных контейнеров и перезапускать при необходимости
  + К сожалению, даже минимальный требуемый сервер t2.large (8 Gb RAM, 22 Gb storage) уже платный (на момент написания книги), примерно 0.1 цент в час, поэтому прекратив работу его надо опускать, чтобы счетчик не тикал
  + Docker образов будет увы много: Spring Cloud Config. Eureka, Kafka, Oauth2, Zuul, Logspout Docker, Zipkin + собственно прикладные сервисы системы. К счастью, Docker образы весьма компактные, чего не скажешь о полноценных виртуальных машинах
  + Наружу за счет применения VPS Security Group settings открываем только два порта: 5555 для Zuul и 22 для SSH, чтобы иметь возможность зайти и посмотрет чего происходит ручками.
    - Образы внутри узла видят друг друга без ограничений.
    - Если для простоты использовать одну и ту же VPS Security Grooup для PostgreSQL / Redis / ECS cluster, то проблем с видимость не будет хотя это и не рекомендуется.
  + Конфигурируем Amazon IAM, чтобы тот мониторил состояние Docker образов

### Запуск системы ручками

* Все достаточно просто, обычно делается пару раз в целях отладки, затем в обязательном порядке автоматизируется
* Конфигурируем Amazon ECS command line client, определяем: регион, идентификатор нашего кластера, открытый / закрытый ключи доступа (хорошо хранить как переменные окружения, так как они часто нужны)
* Затем ecs-cli compose –file …. up (при условии конечно, что образы уже собраны и каходяться в Docker Hub репозитории), чтобы отправить контейнеры на кластер и запустить.
  + За счет ecs-cli ps можно затем посмотреть, что в итоге получилось.
  + В выдаче видим IP адреса (у всех будет один – IP кластера) и порты всех запущенных контейнеров

### Автоматизация развертывания

#### Автоматизация build / deployment pipeline

* Continuous Integration + Continuous Delivery (CI / CD)
* Последовательность операций, запускаемых CI после каждого commit разработчика:
  + компиляция, unit / интеграционное тестирование, сборка Docker image и публикация его в репозитарии
    - интеграционное тестирование в данном случае означает тестирование процессов в рамках одного сервиса, при условии mock остальных и может быть каких-то компонент внутри (в общем не желательно)
  + Для каждой из сред (dev / test )
    - Развертывание и выполнение платформенных тестов
      * тесты отдельных сервисов, не использующие mock других сервисов и направленные в первую очередь на проверку интеграции сервисов в развернутом приложении между собой
      * так как это тестирование «черного ящика», то лучше тесты реализовывать вообще на другом языке. Например, python или groovy
    - Выполнение системных тестов – тестируются основные общесистемные варианты использования всего приложения в целом
    - если тесты проходят нормально, то выполнение переносится на следующую среду
  + Развертывание на prod + финальное выполнение избранных общесистемных тестов (опционально)
    - может выполняться CI как автоматически, так и по команде оператора
    - в данном случае тесты гарантируют, что все компоненты системы по-прежнему нормально взаимодействуют между собой. Иногда такие тесты выполняют регулярно для раннего обнаружения проблем
  + Выполнение monkey тестов – случайное отключение произвольных серверов и
    - раннее обнаружение ошибок за счет проверки того, что
      * система нормально справляется с неизбежными произвольными отказами
      * нигде случайно не образовался stateful функционал, как в сервисах, так и в инфраструктуре
      * любой сервер может быть в любой момент убит без последствий (для него самого / обрабатываемых данных / зависящих от него сервисов) и поднят из свежего образа
      * инфраструктура и остальные сервера нормально отрабатывает дополнительную нагрузку, связанную с мерцанием серверов – дополнительная нагрузка не приводит к снижение уже их собственной стабильности
    - Семейство средств Netflix Chaos Monkey
    - Всегда лучше обнаружить отдельную проблему в нормальном состоянии, когда есть время с ней разбираться. А не тогда, когда вылезла как последствие удара нагрузки / других проблем и у всех жопа в мыле и нет времени
* Все скрипты / конфигурационные настройки, реализующие pipeline, соответственно должны версионироваться и быть под контролем CVS

#### Инструментарий

* В принципе можно полностью воспользоваться облачной инфраструктурой и ничего у себя локально не развертывать
* GitHub - git репозиторий, предоставляет события + REST API для интеграции в CI/CD процесс
* Moven Spotify Docker плагин – для автоматизации построения Docker контейнера
* Docker + Docker Compose – средство легковесной виртуализации
* Docker Hub – публичный репозитарий Docker контейнеров
* Travis CI – облачнаый CI сервис
  + хорошо интегрирован с GitHub (также с SubVersion и Mercurial) и Docker, не столь полно функционален как Jenkins, например, но для наших целей подойдет
  + Очень легко настраивается – travis.yml файл в корне git репозитария каждого сервиса, который описывает процесс сборки, запускаемый по уведомлению с GitHub
  + Для каждой сборки стартует отдельную виртуальную машину, которая конфигурируется travis файлом (sudo, выполнение sh файлов, установка основного языка – jdk и необходимых сопутствующих утилит и библиотек типа ecs-cli, использование системных сервисов и так далее) и используется для выполнения сборки

#### Работа с Docker + Docker Compose

* За счет первого собираем образ легковесной виртуальной машины, который может быть запущен на большинстве операционных систем поверх основной. В отличии от классичесих виртуальных машин он очень компактен, поэтому на типовой машине разработчика с 16 гб оперативки легко может быть запущен одновременно десяток Docker контейнеров
* За счет второго автоматизируем конфигурирование и запуск групп Docker образов, составляющих приложение. Compose поставляется в составе Docker инсталляции и довольно ограничен в функцинальности. Если хочется большего, то Kubernetes или Mesos
* Построение Docker образа выносим в отдельный maven модуль (если в одном контейнере, что не рекомендуется, запускаем несколько сервисов – в отдельный maven проект). Минимально необходимы:
  + **Spotify docker-maven-plugin в pom.xml,** которому передается наименование образа (в него обычно включаем уникальный тэг, которым помечаем код из которого был создан сервис) и набор ресурсов, который должен быть скопирован во временную директорию, в которой будет строиться образ. Минимальный набор ресурсов это запускаемый jar сервиса, sh файл для его запуска и собственно **dockerfile** – конфигурационный файл, который будет использован для создания Docker образа.
  + **dockerfile** – набор инструкций по построению Docker файла, по одной в строке. Наиболее распространенные:
    - **FROM** – какой образ использовать в качесве базового. Практически наследование, что крайне снижает время построения образов для ряда сервисов, потому что общую часть можно вынесить в редко изменяемый родительский образ. Например, в качестве базового можно взять Alpine Linux с предустановленным JDK
    - **RUN** – выполнение команды на конфигурируемом образе. Обычно это либо инсталляция дополнительного ПО, либо создание структуры директорий и установка атрибутов файлов, либо препроцессинг конфигурационных файлов
    - **ADD** – копирование ресурсов внуть конфигурируемого образа
    - **CMD** – определение команды, которая будет выполнена при старте образа. В данном случае мы запускаем sh файл, управляющий запуском нашего сервиса. Такая инструкция может быть только одна.
    - **EXPOSE** – документирование перечня портов, которые будут использоваться запущенным внутри сервисом. Обычно это порты для debug / jmx мониторинг, а также HTTP порт на котором случает сервис.
    - **ENV** – установка значений локальных переменных окружения
    - **HEALTHCHECK** – определение каким образом система будет тестировать жизнеспособность запущенного контейнера
  + **SH файл запуска сервиса**
    - В качестве ресурса должен быть скопирован внутрь контейнера при сборке и использоваться за счет **CDM** инструкции
    - В принципе опционален, так как **CDM** может содержать просто java команду. Но лучше определить, так как появляется возможность войти через SSH внутрь и запустить / остановить ручками, причем с различными аргументами. Неприемлемо в качестве штатной процедуры, но допустимо в процессе отладки
    - Основные этапы:
      * За счет **while + nc (netcat-openbsd)** ждем, пока необходимые сетевые зависимости появятся на своих портах. Для Spring Boot сервисов можно с помощью **curl проверять /health** точки доступа.
      * Запуск сервиса за счет java с –D маппингом переменных окружения внутрь JVM
    - Хорошая практика класть рядом properties файл с дефолтной для запуска внутри контейнера конфигурацией. Тогда, в случае необходимости в процессе отладки что-то поменять не надо будет лезть в код и вспоминать какие дополнительные перменные конфигурирования доступны
* Запуск всей системы вместе за счет **Docker Compose**
  + Средство оркестрации группы Docker образов, которые запускаются / останавливаются все вместе как единая единица развертывания. Позволяет определить для каждого образа свою индивидуальную конфигурацию
  + На основе **docker-compose.yml** со следующими основными инструкциями:
    - **services.<serviceId>**
      * **image** – идентификатор образа, который будет использоваться при запуске. Ищется сначала в локальном Docker репозитории, затем в глобальном DockerHub
      * **ports** – маппинг внутренних портов образа наружу
      * **environment** – определение переменых окружения
        + плохо, что шифрование переменных при этом не поддерживается. Но есть технология **«Docker secrets»** для управления чувствительными данными
        + каждый **serviceId** будет использован как DNS наименование внутри системы виртуализации, поэтому связывать сервисы между собой можно через URL вида <http://serviceId:xxxx>
  + Дальше очень просто **docker-compose –f compose.yml up**
    - В ответ получаем на консоль совместный вывод всех перечисленных контейнеров в процессе запуска. В начале каждой строки вывод – идентификатор контейнера.
    - Можно запустить в detached режиме с –d опцией, а потому переодически подключаться, чтобы продолжать видеть весь лог или отфильтрованный по serviceId

#### Основные этапы CI / CD на основе Travis (разделы в travis.yml файле):

* Устанавливаем JDK
* Получаем sudo и доступ к сервисам типа docker
* Настраиваем уведомления на emal
* Устанавливаем глобальные системные переменные
  + В том числе и чувствительные: login / password для Docker Hub, ключи к Amazon, токен к GitHub и так далее
  + Можно, конечно понадеяться, на безопасноть GitHub репозитария, но лучше не надо. К тому же, он может быть вообще публичным
  + Поэтому лучше их зашифровать локально при помощи travis утилиты и в скрипт поместить результат в виде env.global.secure=<зашифрованное представление>.
  + Основной недостаток: не видно название переменной и мы сами уже обратно восстановить в случае чего не сможем
  + Для шифрования каждого travis файла используется своя пара ключей, поэтому copy / paste между ними с целью повторного использования не возможен
* Выполняем ряд sh команд по инсталляции необходимого нам кастомного ПО
  + Travis rest client – для того, чтобы потом запустить задачу выполнения платформенных тестов из другого репозитория
  + ECS клиент – скачиваем через curl и инсталлируем
  + Экспорт переменных, конфигурирующих работу скриптов сборки
* Выполняем отдельные sh скрипты сборки
  + Устанавливаем TAG на исходный код, данный тэг будет поток использоваться как идентификатор Docker контейнера и служить сквозным идентификатором всего процесса – очень удобно для аудита
  + Запускаем Maven, строим сервис и Docker образ
  + Публикуем собранный Docker образ в Docker Hub
  + За счет ECS клиента развертываем образ на кластере
  + За счет Travis rest client стартуем платформенные тесты

## Cloud Foundry

* PaaS решение с открытым кодом, позволяющее специалистам при развертывании сосредоточиться исключетельно на самом приложении и необходимых ему сервисах. Основное достоинство: процедуры развертывания всех приложений вне зависимости от их природы теперь унифицированы и могут быть автоматизированы.
* Есть несколько реализаций, например, общедоступный Pivotal Web Services с ограниченным набором функциональности, а также Pivotal Cloud Foundry, который может быть установлен заказчиком самостоятельно поверх многих популярных IaaS, например, AWS
* Типовая работа через интерфейс командной строки
  + Логин с указанием организации и текущей среды, в которой осуществляется развертывание
  + Создание сервиса (внешней зависимости приложения) с указанием типа, вместимости (тарифного плана), идентификатора (фактически это DNS)
  + Выталкивание исполнимого артефакта без старта
    - Исполнимые артефакты могут быть очень разными: одиночные java файлы, jar / war, Play, Grails, приложение на основе RoR, Node.js, docker контейнеры (не рекомендуется, так как это фактически двойная контейнеризация) и так далее
    - Один из пареметров это buildpack – сценарий того, каким образом надо обернуть исполняемый артефакт, чтобы потом обеспечить его запуск на одном из серверов облака.
      * Идея повзаимствована у Heroku.
      * За счет этого им все равно, что за приложение они запускают, главное чтобы на выходе был Linux контейнер. Имеется, например, сценарий, который выполняет компиляцию проекта из исходников.
    - Сценарий сам по себе может быть развернут гле угодно и даже содержаться внутри самого артефакта
  + Биндинг запускаемого артефакта с сервисами
    - Фактически это ознаачает поставка переменной окружения со ссылкой на DNS имя сервиса
    - Для доставки перечня внешних зависимостей используется переменная VCAP\_SERVICES
  + Запуск артефакта
  + Все предыдущие этапы можно выполнить одной командой выталкивания, если в качестве параметра передать ей манифест – yml файл с описанием требуемых зависимостей, переменных среды и так далее
* Имеется Spring проект Java Cloud Foundry Client, позволяющий выполнять эти и многие другие операции программно
  + В том числе настраивать подсистемы идентификации / авторизации, общесистемный сервис регистрации и так далее
  + Основан на Reactor библиотеке, что позволяет эффективно описывать последовательности выполняемых операций, а также по возможности их распараллеливать
  + Особенно удобно для реализации общесистемных текстов, предусматривающих переконфигурирование системы в процессе запуска

## Типовые проблемы с пермещением приложений в облако

* Избавление от внешней зависимости в виде JEE сервера, а также из дополнительных артефактов его конфигурирования (обычно в виде специальных дескропторов развертывания).
  + Существуют специальные сценарии развертывания, рассчитанные, например, на WebSphere – WebSphere Liberty
  + Но лучше приложение перерабатывать, чтобы сделать переносимым
* Доступ к внешним зависимостям через JNDI
  + Тут только локализация доступа в одном месте и миграция на использование переменных окружения
* Репликация HTTP сессий
  + Обычно используется широковещательная рассылка сообщений всем нодам кластера. Увы, в облаках это не работает
  + Замена – Spring Session, позволяющий хранить сессии в Redis / Apache Geode / Hazelcast
  + Делать ничего не надо
    - Зависимости spring-boot-starter-redis + spring-session
    - Параметры подключения к Redis
  + Система становится особенно чувствительной к тому, чтобы все данные в сессии были бы сериализованы
  + Дополнительные фичи
    - Поддержка работы с HTTP сессией на WebSocket
    - Single Sign-Off – возможность разлогиниться сразу на всех серверах
    - Разные приложения могут совместно использовать сессии друг друга
* Работа с сообщениями
  + С JMS все плохо, поэтому AMQP, например, RabbitMQ
  + Вообще-то, если последовательно применять идеологию Spring для работы с сообщениями, то придется менять не так уж много
  + Есть мост в виде RabbitMQ\_JMS
* Распределенные транзакции
  + Это зло
  + Опять-таки, это зло
  + Spring поддерживается за счет XSDataSource / XAConnectionFactory
  + Можно подключать различные координаторы: Bitronix, Atomikos
  + Основная проблема: записи о транзакциях надо хранить на файловой системе вечно, а облака как правило такой функции не предоставляют - контейнер может умереть в любой момент. Решение: выделенная машина с SSH доступом + SSHFS файловая система
* Облачные файловые системы
  + Обычно облака не предоставляют долговременные файловые системы для совместного использования
  + Можно использовать файловые системы на FUSE основе, например, SSHFS. Они предоставляют API на основе HTTP, SSH подключения, MongoDB файлового доступа
  + Пример: MongoDB GridFS + @Autowired GridFsTemplate для работы с ней
* Email
  + Например, SendGrid, обращение к которому в простейшем случае реализуется за счет new SendGrid.Email() + конфигурационные свойства подключения к сервису, поставляемые средой. Есть также стандартный JavaMail API доступ.
* Идентификация / авторизация
  + Cloud Foundry предоставляет UUA сервис, который хорошо интегрируется с Spring Security
  + Альтернатива – Stormpath, который работает фасадом перед другими средствами ИА или выступать как подобное средство самостоятельно

# Приложение 1 – тюнинг серверов

* Число потоков и объем доступной памяти НЕ должны быть слишком большие. В противном случае CPU будет тратиться на частые переключения, а сборка мусора запускаться редко и длиться долго, заставляя сервер «икать»
  + Можно выделать отдельные входные сервера, преобразующие синхронные внешние вызовы в асинхронные запросы, например, через JMS, с ожиданием ответа. Таким образом мы размениваем таймаут подключения на таймаут запроса за счет затрат на управление ожидающими потоками
  + Если все операции сервиса,так или иначе, упираются в базу, то не имеем смысла делать допустимое число потоков большим чем размер пула соединений с БД
* Частая проблема с сокетами – их не хватает и входящие соединения отваливаются
  + Много CLOSE\_WAIT – проблемы в исходном коде, утечка сокетов, что-то где-то не закрываем. Временное решение – автоматически перегружать сервер, когда число таких сокетов превышает допустимое
  + Много TIME\_WAIT – надо тюнить систему, чтобы она могла использовать их повторно или как можно быстрее прибивать нафиг
* За счет /etc/sysctl.conf
  + increase max open files
    - fs.file-max = 1000000
  + increase the port range (to allow more sockets to be opened)
    - net.ipv4.ip\_local\_port\_range = 9000 65535
  + allows OS to reuse sockets in TIME\_WAIT state for new connections when it is safe from a protocol viewpoint.
    - net.ipv4.tcp\_fin\_timeout = 15
    - net.ipv4.tcp\_tw\_reuse = 1
    - net.ipv4.tcp\_tw\_recycle = 1
* за счет /etc/security/limits.conf:
  + soft nproc 1000000
  + hard nproc 1000000
  + soft nofile 1000000
  + hard nofile 1000000
* За счет /etc/pam.d/su:
  + session required pam\_limits.so

# Приложение 2 – Oauth 2.0 и вокруг

* Основное отличие от предыдущего стандарта OAuth 1.0
  + Изменились наименования участников
  + Процесс стал проще
    - раньше когда пользователь обращался к сервису, то сначала сервис обращался по защищенному каналу к Oauth провадйеру за специальным одноразовым токеном (Request Token), который подмешивался в последующий запрос пользователя за идентификацией
    - теперь отсутствует предварительная операция запроса Request Token и ВСЕ токены, так или иначе, ходят через публичную сеть (защита, наверное, стала слабее)
  + Нет обязательного требования подписи передаваемой информации.
    - Поэтому использование HTTPS стало обязательным, а сервис каждый раз должен проверять полученный SAML2 токен на центальном сервисе авторизации
    - Это может быть проблемой для высоконагруженных проектов, при балансировке и тому подобных особенностях развертываия. Частично лечится JWT стандартом

## Основные варианты взаимодействия участников (Types of grants)

### Authorization Code Grant Flow (three legged OAuth)

* основной и наиболее часто используемый вариант использования при интеграции двух систем. Наиболее сложный тип, остальные это различные упрощения на основе данного
* Основная цель: владелец ресурса (пользователь) хочет предоставить любому другому (заранее не определенному и не доверенному) приложению-клиенту доступ к ресурсу, хранимому и защищаемому на определенном сервере. При этом он естественно не хочет передавать этому приложению свои логин / пароль
* Единственный вариант, когда авторизация происходит полностью на клиенте и не возможна подмена одного приложения (клиента) другим. Во всех остальных случаях ситуация хуже, так как IdP в принципе можно подменить.
* Пользователь инциирует операцию доступа к защищенному ресурсу. Например, нажав кнопку на клиенте, которая выполнит GET защищаемого ресурса
* Запрос пользователя редиректится на IdP
  + Например, через GET с параметрами client\_id, redirect\_uri (куда возвращать потом пользователя), response\_type=code, scope. Крайне рекомендуется в redirect\_uri добавлять уникальный для каждого пользователя параметр. Используется для предотвращения CSRF атак
    - Там он аутентифицируется, не важно, при этом как – это специфика реализации IdP
    - Опционально у него могут запросить дополнительную информацию, например, к какому именно экземпляру ресурса он предоставляет доступ или какие операции над ресурсом он разрешает выполнить клиенту
    - После чего пользователя редиректят обратно на клиента, вложив в запрос Authorization Code, например, при помощи GET запроса с параметром code
* Клиент вычленяет код авторизации из полученного запроса.
  + В принципе, вместо редиректа пользвателю могут просто выдать этот самый код на экран / прислать по другому каналу, после чего он сам вручную донесет этот код авторизации до клиента и продолжит выполнение запрошенного процесса
* Клиент обращается с Authorization Code на IdP и обменивает на токен доступа.
  + Как правило, это POST запрос с параметрами grant\_type=authorization\_code и client\_id, client\_secret, code
  + В ответ приходят
    - access\_token – токен доступа
    - token\_type = “bearer”
    - expires\_in
    - refresh\_token – опциональный долгоживущий, в случае если основной токен доступа короткоживущей. Используется для получения следующего токена доступа без участия клиента
  + Как правило токены сохраняют клиентом для постоянного хранения. Особенно, долгосрочный.
  + Обращаться к IdP может
    - как сам клиент (мобильное приложение, SPA веб приложение), в этом случае все обращения к IdP идут с одного адреса
    - так и сервер, обслуживающий веб-приложение с более традиционной архитектурой.
      * Обращения теперь идут уже с двух адресов и мы имеем взаимодействие сервер-сервер, которое может быть защищено дополнительно
      * Важно, что в данном случае токен доступа никогда не попадает к коду Java Script в браузере на стороне пользователя и поэтому не может быть скомпрометирован.
* С токеном доступа клиент идет на сервер ресурса и выполняет необходимые операции над защищенным ресурсом
  + В принципе клиент может работать не с одним сервером ресурсов, а с несколькими (или с различными типами ресурсов / операций доступа). В этом случае, ему понадобится несколько обращений с Authorization Code на IdP за несколькими токанами доступа

### Implicit Grant

* Практически то же самое, но вместо Authorization Code клиенту выдают сразу токен доступа, с которым он может прямо обращаться за ресурсом
* Применяется в ситуации, когда у клиента нет серверной части и возможности сохранить надолго полученный токен.
  + Клиент открывает встроенный браузер со страницей авторизации сервис провайдера – GET response\_type=token&client\_id
  + Пользователь авторизауется сервис-провайдером
  + Затем, встроенный браузер редиректится СП на фейковый URL, содержащий в адресе токен доступа и прочую сопутствующую информацию.
    - это может быть, например, страница со статическим контентом, располагающаяся на самом IdP
    - В последнее время это как правило 302 обратно на URL самого приложения вида <http://someHost/appPage#token=xxxxx>, получение которого не вынуждает браузер делать новый запрос к серверу за страницей, но JS имеет доступ к информации после # и может считать токен
    - Пример использования: приложение с ФБ авторизацией
  + Клиент перехватывает запрос редиректа, вычленяет токен доступа и продолжает запрошенную операцию по доступу к ресурсу
* Соответственно, запросов сервер-сервер в данной схеме просто нет

### Resource Owner Password Credemtials Grant

* Совсем просто: клиент сам запрашивает у пользователя логин+пароль, идет с ним на IdP и обменивает на токен / токены, который предъявляет защищаемому ресурсу
* Используется, когда оба, клиент / ресурс доверяют друг другу. Обычно они при этом принадлежат одной и той же организации
* Запрос к IdP выглядит как GET grant\_type=password&client\_id&client\_secret&username&password
* Oauth2 access token при этом обычно НЕ попадает к пользователю,а оседает в сессии клиента. Если же клиент stateless, то увы…

### Client Credential Grant

* Пользователь вообще в принципе не задействован. Клиент сам по себе авторизуется на IdP в качестве Subject и получает токен / токены.
* Запрос к IdP выглядит как GET grant\_type=client&client\_id&client\_secret&username&password
* Используется, когда оба, клиент / ресурс доверяют друг другу. Обычно они при этом принадлежат одной и той же организации
* Типовой случай: клиент периодически собирает какую-то статистику / выполняет отчеты на основании данных, пренадлежащих ресурсу

### Refresh tokens

* Суб-процесс описывает работу с двумя токенами: длинным токеном авторизации и коротким токеном доступа, который необходимо периодически обновлять, предъявляя токен авторизации
* GET grant\_type=refresh\_token&client\_id&client\_secret&refresh\_token

## Недостатки

* Процесс формирования спецификации осуществлялся совместно представителями интернет сообщеста, которым нужен был улучшенный по сравнению с первой версией стандарт, и представителями корпораций, которым был нужен пригодный для расширения фреймворк, на основе которого уже каждый производитель конкретного решения мог развить собственную версию. В результате спецификация получилась столь расплывчатая, что отдельные имплементации оказались практически не совместимые между собой.
  + Даже ФБ, где реализацией занимался один из авторов спецификации, в итоге игнорирует / специфически трактует многие ее части
  + Сврехгибкость ведет не только к несовместимости отдельных реализаций между собой, но также и к тому, что многие реализации просто дырявые и высоко уязвимые.
  + Правильная реализация решения на освное спецификации OAuth2 требует большого количества специальных знаний, отсутствующих у большинства разработчиков
* В результате один из авторов OAuth1 и активный участник процесса разработки второй версии Эран Хаммер отозвал свое имя из числа авторов OAuth2 и порекомендовал тем, кто успешно реализовал первую версию, на ней и оставаться
* Принципиальная проблема с безопасностью пользователей мобильных приложений, авторизующихся через OAuth2
  + Слишком широкие возможности для авторов приложения не прозрачные для пользователей
    - Свободный доступ к стеку TCP/IP
    - Возможность перехвата нажатий клавишей
    - Встроенный в большинство приложений браузер, предлагаемый ими как средства аутентификации в IdP
  + Все это приводит к тому, что возможности для фишинга открываются воистину безграничные.
  + Слегка лечит то, что большинство IdP, например ФБ, также предоставляют свои собственные приложения + SDK, позволяющией другим программам пользоваться тем фактом, что пользователь уже залогинен в фирменном приложении. Но увы, пользователи столь привыкли к кнопкам «Зайти через», что тычат в них и вводят пароль при помощи опасного мобильного браузера.
* Принципиальная проблема у провайдеров с различением между собой различных мобильных клиентов
  + Авторизация самого клиента по client\_id / client\_secret имеет существенную проблему – эта пара одинакова для всех экземпляров клиентов, как правило, зашита в клиента и легко извлекаема.
    - Соответственно кто угодно может прикинуться клиентом и получив каким-либо образом данные пользователя (см. предыдущий раздел) вломиться в защищаемое API
    - Также не понятно, каким образом отличать серверных клиентов от мобильных.
  + Если у нас только серверные клиенты все просто: привязка по IP (перечню IP), взаимная аутентификация через сертификаты и все такое прочее
  + Если приложение наше
    - Если мы берем с пользователя приложения плату (даже нулевую, все равно ему придется осуществить платеж с карточки), то тоже все неплохо – AppStore / PlayMarket предоставляют счета покупки, содержащие Bundle ID приложения, привязанные к конкретному пользователю. Их можно проверить, за счет запроса к Apple / Google в момент выдачи / обновления токена
    - Мы можем сами его распространять среди ограниченного круга лиц, производя инициализацию каждого поставленного экземпляра генерируемым сертификатом. Муторно, но можно. Правда в таком случае, лучше сразу закрыть весь трафик за счет TLS и отказаться от OAuth в принципе
  + Google предоставляет сервис SafetyNet, который позволяет получить отпечаток ключа, которым было подписано приложение, а также его BundleID
    - К сожалению Apple такой информации не предоставляет
    - Опять-таки, эта информация предоставляется только авторам приложения. А если к нам стучится сторонний клиент?
  + Соответственно, мы не может доверять данным из scope / client\_id а также различать различные мобильные приложения между собой. Единственный выход – поставлять собственные мобильные SDK, тесно интегрированные с мобильной операционной системой
* Типовые атаки
  + веб-приложение редиректит пользователя на сервер IdP, добавив в конце @some.server.com – адрес прокси, который будет перехватывать все общение пользователя с IdP и, разумеется, пароль / логин. Если URL достаточно длинный, то дописанный прокси будет практически не заметен
  + Пользователя просто редиректят на фишинговый сайт, замаскированный под ОР. Общения с реальным ОР в принципе не происходит или происходит от имени пользователя (MiTM). URL аналогично маскируется
* Технологии дополнительной защиты
  + Разовый пароль
    - SSL канал между IdP и пользвателем
    - При регистрации пароль высылается пользователю в письме
    - Когда пользователь авторизуется на IdP в первый раз с данным паролем, то ему кладется специальная кука и ее копия сохраняется на сервере IdP
    - При последующих аутентификациях на IdP, затребованных различными клиентами
      * Проверяется наличие и состав куки
      * Пользователю при аутентификации выводят специальную уникальную только для него картинку, которая служит свидетельством того, что идет реальное общение с IdP
      * Пароль вводится на графической клавиатуре, где каждый раз разкладка новая
  + Специальные плагины в браузере, берущие на себя работу с IdP
  + Взаимная аутентификация пользователя и IdP на основе двух сертификатов. С одной стороны пользователю не надо теперь воодить пароль, с другой сертификат может быть украден трояном

## Веб токены

* Параметр запроса, который содержит в зашифрованном / подписанном виде всю информацию, необходимую для авторизации доступа к ресурсу.
* Выдается сервером на основании процедуры аутентификации.
  + Либо симметричный алгоритм шифрования и все сервера шарят один и тот же секрет, либо ассиметричный, тогда сервер аутентификации обладает закрытым ключем, а все остальные открытым
  + Лучше, когда ЗК не хранится на сервере авторизации, а все операции подписи / шифрования выносятся на отдельный сервер, в отдельном домене, лучше вообще аппаратный, с неизвлекаемым ключем и так далее.
* В простейшем случае в состав токена входит срок годности и по его истечении клиент идет авторизоваться заново.
  + Соответственно, чем должен этот срок, тем меньше нагрузка на сервер аутентификации
  + Основная проблема: в течении этого срока невозможно
    - Управлять привилегиями пользователя, если они зашиты в токен
    - Отозвать токен, если он был украден
* Поэтому вводят схему из двух токенов:
  + Долгосрочный – выдается на основании процедуры аутентификации и позволяет получить короткосрочный токент
  + Короткосрочный
    - выдается на основании долгосрочного и позволяет получить доступ к ресурсу
    - Основной достоинство – как правило подписанный и мы его можем быстро проверить НЕ обращаясь к базе данных
  + В момент очередного запроса короткосрочного токена мы имеем возможность вмешаться
    - И либо прекратить этот цирк вовсе
    - либо сменить пользователю перечень назначенных ролей
  + Недостаток: необходимо хранить перечень долгосрочных токенов с привязкой к пользователям, то есть опять-таки statefull.
    - Чем меньше срок обмена – тем больше нагрузка на расшаренное хранилище пользователей и дорогую процедуру аутентификации
    - Если же мы делаем процедуру продления токена автоматической, то ресурсов тратим меньше, но не можем вмешаться и много возни с двумя токенами
* Гибридный токен
  + В одном токене две даты – expiration / refresh, после истечения последней сервер ресурса сам пойдет и обновит токен, вернув клиенту свежий
  + У пользователя есть свойство minRefreshDate – минимальная refresh дата, валидная для токена, который предъявляется к обновлению
    - В процессе аутентификации это поле заполняется текущей датой
    - В процессе обмена токента
      * проверяется, что это поле не пусто и меньше чем refresh дата обмениваемого токена
      * опять-таки заполняться текущей датой
  + Если надо прибить пользователя / изменить содержимое токена – обнуляем ему minRefreshDate и таким образом заставляем его через некоторое авторизоваться.
    - В процессе данной процедуры minRefreshDate соответственно меняется
  + Если атакующий пробует обновить старый токен
    - либо minRefreshDate==null – авторизация еще не прошла
    - либо minRefreshDate > refresh даты токена – пользователь уже авторизовался и токен устаревший
  + Соответственно, пока не истекла expiration, мы ничего сделать не можем. Но потом, имеем возможность вмешаться в автоматическую процедуру продления токена не тратя ресурсы на полную процедуру аутентификации
* Обычно готовим на базе JWT токенов
  + особенно, когда IdP и все SP под нашим контролем, а потому можно применить любую схему авторизации и навязать ее пользователям.
  + Если же требуется интеграция посредством веб-токенов с другими системами, то придется уважать стандарты, которые скорее всего в том или ином виде выросли из мира SAML

### SAML токен

* Стандартный XML based фреймворк для обмена security информацией между сторонами
* SAML токен это достаточно пространный XML документ, содержащий следующие данные
  + Subject – описание аутентифицированного клиента (principal).
  + claims - аттрибуты клиента, его роли, LDAP аттрибуты
  + issuer – описание IdP, то есть издателя токена
  + audience - информация относительно SP: кому предназначается токен, условия применения, ограничения по использованию
  + времена создания, начала действия, окончания действия
  + ЭЦП всей этой информации. Причем это XML подпись, поэтому внутри куча информации о томкакие алгоритмы используются для хэширования данных, выполнения подписи, сбора подписываемой информации, ее трансформации, данные сертификата. Соответственно задействованы стандарты: XML Signature, XML Encryption
* Типы токенов (Subject confirmation method – каким образом подтверждается, что предъявитель запроса сего действительно Subject)
  + Bearer – никаким, то есть кто предъявил валидный токен, тот и будет выступать в качестве Subject.
    - Соответственно, если токен украден и вставлен в любой чужой запрос, то SP этого не заметит.
    - Зачастую содержит в себе только самый минимум информации, остальную сервер ресурса может получиь обратившись (каждый раз) с полученным токеном на IdP
    - В REST мире приемущественно используется эта схема за счет ее простоты
  + Holder-of-keys
    - IdP вкладывает публичный ключ клиента в токен и подписывает его
    - Клиент, отправляя запрос RS, подписывает его целиком своим закрытым ключем
    - RS, получив запрос, вначале проверяет валидность публичного ключа клиента за счет подписи IdP, затем проверяет целостность запроса за счет подписи клиента и его публичного ключа
    - Имеется также схема с симметричными ключами
  + Sender vouches – когда промежуточный сервис выступает в качестве представителя клиента. Примерно аналогичная схема с множественным подписыванием
* Проблемы
  + Длинны он при этом получается просто необычайной, что накладывает ограничения, если передавать в параметре запроса
  + Подразумевается, что стандартный, но на самом деле разные IdP и сервера приложений зачастую друг друга из коробки просто не понимают и требуется долгая конфигурационных настройка параметров. Особенно в части правил / проверки формирования подписей.
  + За исключением Bearer, остальные варианты требуют от клиента выполнения криптографических операций и вообще все становится очень и очень сложно
  + Далеко не все системы понимают все виды токенов, например
    - SAP (SP) не принимает Bearer, Active Directory Fideration Server (IdP) не понимает Sender vouches
    - Соответственно, клиент должен: авторизоваться на ADFS, затем с этим bearer токеном саходить к Data Power, обменять его на Sender vouches и уже с ним идети к SAP
    - Соответственно, если надо обеспечить доступ посредством REST к SAP, то приходится играться с подобными интеграционными задачами

### JWT токен

* Сам по себе стандарт не особенно полезен, так как содержит описание только структуры данных и не содержит определения протоколов использования.
  + Именно поэтому, может легко быть использван в самопальных протоколах идентификации / аутентификации и очень популярен
  + Широко используется в сочетании со стандартами OAuth2 и OpenID
  + Использует стандарты JWS (JSON Web Signature) и JWE (JSON Web Encryption)
* В соответствии со стандартом представляет собой разновидность Bearer токена
  + Часто пересылается в заголовке запроса Authorization: Bearer <token>
  + Переносит внутри себя примерно ту же информацию, что и SAML токен.
    - В частности, содержит утверждения (claims) в виде <ключ>:<значения>, которые могут использоваться в RBAC (Role-Based Access Control) или ABAC (Attribute-Based Access Control) моделях безопасности.
    - Стандартные ключи имеют длинну наименования в 3 символа
  + Может быть использован как самим API, для восстановления контекста безопасности, так и при обращении к другим API от имени обратившегося пользователя (secure identity propagation)
* Состоит из основных частей <метаданных (header)>.<тела (payload)>.<подписи (singature) закодировнных заголовка + тела>, каждая часть кодируется в base64url
  + Метаданные
    - alg – алгоритм подписи
    - “typ”: “JWT”
  + Тело – любые данные
    - Часто встречаются:
      * Iss – issuer, опциональный, например, идентификатор сервера IdP
      * Exp – expiration time, обычно число секунд с начала времени
      * Sub – subject, уникальный идентификатор пользователя в контексте IdP (локальный), никогда не переназначаемый, может быть уточнен за счет
        + upn – user principal name, например, в виде user@domain
        + unique\_name
        + family\_name, given\_name
      * Aud – audience, предмет утверждения, что именно содержит очень специфично для того или иного приложения. Например, может быть одним на все API или только на конкретную функцию / набор ресурсов.
      * nbf, iat – начало использования и время издания, обычно это одно и то же значение
      * jti – идентификатор данного токена в контексте одного IdP
      * amr – authentication method, например, login+password
      * roles – массив строковых наименваний ролей, присвоенных subject
      * appid – уникальный идентификатор приложения, запросившего токен
      * scp – scope, набора привилегий, предоставленных приложению вне зависимости от пользователя, описываемого токеном
      * auth\_time – время когда пользователь реально авторизовался. В случае, например, использования перезапрашиваемого токена доступа это время может очень сильно отличаться от остальных
      * nonce – строкое значение, ассоциирующее токен с клиентской сессией, используется для предотвращения replpay аттак
      * at\_hash – хэш токена доступа, выполненный с помощью алгоритма, указанного в заголовке
      * sub\_jwk – открытый ключ, который может быть использован для проверки подписи токена. Используется, self-Issued IdP
    - Для остального наименования из IANA JSON Web Token Registry или свои собственные на свой страх и риск
  + Подпись
    - Поддерживается подпись в виде HMAC на секретном ключе (например, SHA 256) или RSA на паре
    - Открытый ключ, которым выполняется проверка подписи, может содержаться как в токене, так и доставляться независимым путем
  + В принципе стандарт подразумевает различные варианты, в том числе неподписанные токенты и зашифрованные токенты.
    - Неподписанный токен может использоваться, например, при переносе информации через защищенный другими механизмами канал
    - В наиболее часто используемом варианте применяется только подпись, так как обмен информацией все равно происходит по защищенному каналу
  + На сайте jwt.io есть сервис расшифровки токена
* Простейшие варианты использования для доступа ка API
  + Первый шаг примерно один и тот же: клиентское приложение получает токен от IdP
  + Частые варианты дальше
    - API имеет открытый ключ IdP, соответственно сам проверяет токен и восстанавливает контекст безопасности
    - Перед API стоит Gateway, который выполняет всю работу за API
      * В этом случае API и Gateway доверяют друг другу, канал взаимодействия между ними защищен другими средствами (континент, mutual SSL, JWT токен, авторизующий сам Gateway)
      * Gateway передает API не весь токен, а только необходимую информацию, выделенную из него и возможно измененную / обогащенную. Например, в части маппинга ролей, определенных в IdP, на роли, специфичный для данного API
      * В принципе Gateway может время от времени связываться с IdP, и запрашивать от него дополнительную информацию, которой он обогащает утверждения, полученные за счет токена
      * Пример такого взаимодействия общедоступными средствами: Apigee Gateway + Azuer AD.
        + Кстати, Apigee по умолчанию не предоставляет стандартных операций работы с JWT, но это можно сделать за счет заливаемого Java класса, который возьмет на себя эту функцию
* Spring API
  + Jwts.builder() производит экземпляр JwtBuilder с методами
    - Установки основных данных (expiration) и произвольных данных (claims)
    - Подписи – signWith()
    - Формирования – compact()
  + Jwts.parser() c методами
    - setSingingKey()
    - parse(token)
    - DefaultClaims getBody() – по сути дела Map данных

# Приложение 3 – Advanced Messagin Patterns

* Только наиболее актуальные, более полный перечень в в дайджесте «Шаблоны интеграции» или одноименной широко известной книге

### Синхронный вызов через асинхронный обмен сообщениями

* Много клиентов и много серверов, каждый метод интерфейса взаимодействия представлен двумя очередями inq / outq
* Каждый клиент имеет уникальный clientId и подписан на сообщения в outq с фильтром client=<clientId>
* Клиент, посылая сообщение
  + Генерирует сообщение с уникальным messageId и устанавливает source=clientId
  + Добавляет сообщение в глобальную коллекцию, высылает в inq и блокируется за счет wait() на сообщении (обязательно с таймаутом)
* Любой из серверов, получив сообщение из inq
  + Обрабатывает его
  + Формирует ответное сообщение с correlationId=<messageId исходного сообщения> и client = <source исходного сообщения>. В ответном сообщении может быть либо результат, либо текст исключения (сериализовать само исключения для пересылки вряд ли удастся)
  + Высылает сообщение в outq
* Клиент, получив сообщение из outq
  + По correlationId получает из коллекции исходное сообщение, переносит полученные данные в секцию «ответа» и выполняет notify() на сообщении, высвобождая поток, который ожидает выполнения запроса.
  + Клиенский поток либо десериализует и возвращает результат, либо возбуждает исключение на основании присланного описания
* Плюсы
  + В лобой момент можем добавить любое количество клиентов / серверов без необходимости переконфигурирования системы
  + Любой сервер работает в комфортном для себя режиме (особенно если зажать число потоков в пуле подписчика), а также может в любое время умереть и быть перезапущенным
  + Можем осуществлять обновление системы по частям, перезапуская сервис за сервисом
  + Мониторинг производительности данного бизнес процесса элементарен - длинна очереди
  + Если между собой взаимодействуют, к примеру, веб и апп сервер, то сервер очередей можно разместить в DMZ. В результате имеем сетевые вызовы только изнутри защищенного сетевого домена
* Недостатки
  + Принципиально невозможно сделать синхронный вызов и получить результат в рамках одной транзакции: пока транзакция не завершится, сообщение не уйдет, а значит ответ не придет – ошибки не будет, но ожидание будет вечным
  + Запросы клиентов равномерно размешаиваются по всем серверам
    - Hibernate кэши второго уровня с данными пользователей на всех серверах будут содержать одни и те же данных и в перспективе переполняться
    - Медленные сервера будут замедлять всех клиентов за счет того, чтобы сообщения из очереди вычитываются последовательно с головы
    - В процессе обновления системы запросы новых клиентов могут обрабатываться старыми серверами и наоборот

### Sticky Event “Session”

* «продвинутый» вариант предыдущего шаблона
* Когда мы получаем первый запрос пользователя (в сессии или просто храним некоторый локальный кэш), то высылаем сообщение с destination=null. Оно будет получено случайным сервером
* Каждый сервер
  + Отвечая на сообщение, устанавливает destination=serverId
  + Подписан на inq с двумя фильтрами, работающими как OR
    - destination = serverId
    - destination = null
    - now – messsageDtCreate > someTTL – фильтр для обработки «старых» сообщений, его придется время от времени обновлять
* Клиент, получив сообщение, запоминает у себя сервер, обработавший запрос клиент, и в дальнейшем, высылая запрос устанавливает соответственный destination= serverId, чтобы запрос был обработан тем же самым сервером
  + Если соответствующий сервер жив, то запрос будет на нем нормально обработан и за данными пользователя в базу лазить не придется
  + Если сервер умер или тормозит, то за счет последнего фильтра его поднимет другой или более быстрый сервер и в результате обслуживание клиента «переедет» на него
* В результате очередь сообщений разбирается с двух сторон, с хвоста и с головы. То есть более быстрые сервера воруют работу у более медленых.

### Развертывание event driven GUI в console-like стиле

* Традиционный подход (VB, консольные приложения, Delphi и так далее) в написании интерфейса:
  + onButtonPress() { … if( openForm(formId, arg1, arg2).someProperty = xxx) { …} …}
  + Разработчики, привыкшие к подобному, с трудом переучиваются на собременный подход с callback методами
* Если обращение к каждому примитиву GUI оформлять как команду, то можно устроить своего рода концевую рекурсию
  + onButtonPress ничего не находит в аккумуляторе, кладет аргументы свего вызова в стек и начинает выполняться
  + openForm() открывает окно диалога, выполняет обмен с сервером, получает результат и кладет в аккумулятор. Затем она (обычно за счет выбрасывания исключения) перезапускает обработку предыдущего фрейма. Хотя в принципе может повторить все тот же фокус уже от своего имени
  + заново перезапущенный onButtonPress, находит в аккумуляторе результат уже выполненного openForm() и вместо того, чтобы открывать форму идет выполняться дальше. До конца или пока не встретится с очередным дочерним диалогом
* Основной недостаток: вызываемые промежуточные методы должны быть идемпотентными или откатывать транзакцию каждый раз, когда выполнение прерывается

### Обратный адрес (Return Address)

* уведомление получателя о месте, куда надо направить ответ на сообщения
* Можно задать канал ответов прямо в коде получателя
  + но это снижает гибкость системы, а также лишает возможность получателя обслуживать много инициаторов запроса
  + возможно, что результат обработки запроса интересует не отправителя, а другой (следующий) компонент
  + у инициатора может быть несколько процессоров результатов обработки, например в зависимости от типа результата
* логика определения дальнейшего обработчика результата выносится из получателя сообщения в отправителя за счет использования поля ReplyTo. Как правило сообщение-ответ связывается с сообщением-вопрос за счет идентификатора корреляции –CorrelationID

### Цепочка сообщений (Message Sequence)

* передача посредством сообщений сколь угодно большого объема данных
* Ограничения на объем сообщения зависит от вендора, как правило есть некоторый оптимум между объемом и производительностью БПС
* Большой объем данных передается последовательностью, в каждом сообщении последовательности – ИД цепочки, ИД позиции, Индикатор длинны или конца последовательности
* Для отправки или приема последовательности лучше пользоваться транзакционным клиентом.
  + Вся цепочка обрабатывается в рамках одной транзакции.
  + При этом во многих системах гарантируется еще и порядок сообщений в рамках одной транзакции – облегчение работы по восстановлению данных
* Часто ИД цепочки является одновременно и ИД корреляции
* Несовместимо с конкурирующими потребителями или диспетчером сообщений – все части должны собираться в одном месте. Поэтому используется только с каналом «точка-точка»
* Вместо цепочки можно использовать шаблон квитанции: данные в базу, ид от базы в сообщение. При этом обе системы должны иметь общий доступ в базу
* Может реализовываться разветвителем совместно с агрегатором, особенно когда между ними данные надо протолкнуть в канал с ограничениями

### Преобразователь порядка (Resequencer)

* восстановление исходного порядка ответов в соответствии с порядком запросов
* Основная причина нарушения порядка – асинхронная выборка сообщений для обработки + разное время на обработку каждого сообщения + разная скорость каналов передачи
* Можно отправлять сообщения по одному по мере обработки предыдущего, но
  + Значительно замедляется работа системы. Не допускается параллельная обработка сообщений. Большое время простоя компонент
  + Приходится завязаться на получение сообщений от обрабатывающих компонент – сложно и возрастает связанность системы. Не всегда можно контролировать исходный поток
* Фильтр с сохранением состояния
  + получает сообщения, хранит их во внутренних буферах до полного получения каждой последовательности, публикует содержимое буфера в правильной последовательности
  + может хранить только часть последовательности, относительно которой выявлен неправильный порядок - последний полученный и уже отправленный 9, очередной 11, значит ждем 10, по его получению отправляем в правильном порядке
* В каждом сообщении – ИД последовательности (часто он же ИД корреляции) и порядковый номер сообщения в последовательности.
* Проблема генерация порядковых номеров.
  + Как ид можно использовать GUID или MAC+время обработки. С порядковым номером сложнее – они должны образовывать непрерывную последовательность
  + Может быть узким местом процесса разветвления, лучше встраивать непосредственно в разветвитель
* Проблема переполнения буфера, источники:
  + длинный по своей природе поток сообщений в транзакции,
  + сбой компонента, обрабатывающего только сообщения определенного типа – тормозит все последовательности
* решения
  + при работе с очередью иногда БПС позволяет выбирать «задние» не трогая очередные. Стандартным способом – только просматривать, выбирать нельзя
  + контролировать работу поставщика сообщений используя уведомления о старте / остановке работы. Можно уведомлять поставщика об объеме свободного буфера, чтобы тот мог сгенерировать определенный объем сообщений
  + генерация заменителя отсутствующего сообщения – если потребитель и природа данных подобное позволяют, например, при передаче голоса по IP сетям

### Квитанция (Claim Check) – сокращение объема данных в пересылаемом сообщении

* Основные компоненты: фильтр содержимого, сохраняющий удаляемое в базе данных, поле с ИД данных в сообщении, расширитель содержимого на основании базы и ИД данных
* Особенно удобное, когда XML сообщение с внедренными CDATA, особенно бинарниками
* Плюсы (кроме объема трафика)
  + Снижает зависимость промежуточных систем от информации для них не предназначенной
  + Увеличивает безопасность
  + Увеличивается производительность, так как однократное чтение / запись в базу более эффективно.
    - при передаче между компонентами важные и большие данные все равно ложатся в базу
    - затраты на сериализацию / десериализацию, шифрование / дешифрование и т.д.
  + Особенно удобно когда сообщение пройдя через цепочку обработки возвращается отправителю. А основная часть данных нужна ТОЛЬКО отправителю
* Выбор ключа
  + В теле сообщения уже есть бизнес ключ, например ИД клиента. Плюс – унификация обработки таких ключей
  + Ключ в базе = ИД сообщения: это делает ненужным поле в сообщении, но
    - может встретиться вложенное «сокращение – расширение» или сообщение пройдет через один редуктор несколько раз и в базе будет конфликт.
    - Сообщения могут упаковываться друг в друга, меняя ИД. Или порождать друг друга
  + УИД
* Удаление данных из базы
  + Компонент, расширяющий сообщение. Строго, но нельзя использовать одни и те же данные несколько раз
  + По мере устаревания
  + Не удалять вовсе, например для систем бухучета
* Специальный вариант использования – безопасность
  + можно скрывать конфиденциальные данные на время обработки, заменяя ее на абстрактные УИД-ы. Например – ИД кред. Карточек, имена, адреса и т.д.
  + контроль за доступом к закрытой информации локализуется в расширителе
  + политика сокрытия информации локализуется в редукторе
* диспетчер процессов тоже использует квитанцию для хранения состояния