Разработка

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

[Основа 1](#_Toc15570515)

[REST слой 1](#_Toc15570516)

[Обработка ошибок 2](#_Toc15570517)

[Гипермедиа (HATEOAS) 2](#_Toc15570518)

[Управление версиями 3](#_Toc15570519)

[XML vs JSON 4](#_Toc15570520)

[Контроль формата 4](#_Toc15570521)

[Удобство для разработчика 4](#_Toc15570522)

[Стоимость 4](#_Toc15570523)

[Простота и функциональность API 4](#_Toc15570524)

[Расширяемость 5](#_Toc15570525)

[Безопасность 6](#_Toc15570526)

[Итого 6](#_Toc15570527)

[Бинарные форматы 6](#_Toc15570528)

[Protobuf 6](#_Toc15570529)

[Avro 7](#_Toc15570530)

[Thrift 7](#_Toc15570531)

[Клиентская часть 7](#_Toc15570532)

[Альтернативы REST 8](#_Toc15570533)

[REST vs RPC 8](#_Toc15570534)

[Идеология 9](#_Toc15570535)

[Маппинг на HTTP 9](#_Toc15570536)

[Использование 9](#_Toc15570537)

[Эволюция API 10](#_Toc15570538)

[Примеры использования 10](#_Toc15570539)

[RPC лучше использовать 10](#_Toc15570540)

[REST лучше использовать 11](#_Toc15570541)

[GraphQL 11](#_Toc15570542)

[Основы 12](#_Toc15570543)

[Схема данных на основе JSON 13](#_Toc15570544)

[Запросы 13](#_Toc15570545)

[Минимальная разработка 13](#_Toc15570546)

[Достоинства 14](#_Toc15570547)

[Недостатки 15](#_Toc15570548)

[WebSocket 16](#_Toc15570549)

[Сервер на основе Jetty 16](#_Toc15570550)

[Сервер на основе Spring / STOMP 16](#_Toc15570551)

[Сравнение REST vs WS: 17](#_Toc15570552)

[Обмен сообщениями 17](#_Toc15570553)

[JMS 17](#_Toc15570554)

[Rabbit MQ 19](#_Toc15570555)

[Apache Kafka 20](#_Toc15570556)

[Валидация данных 20](#_Toc15570557)

[Работа с хранилищами данных 22](#_Toc15570558)

[Из REST в ORM и обратно 23](#_Toc15570559)

[Внутренняя модель данных <=> ORM Entity 23](#_Toc15570560)

[Внутренняя модель данных <=> ORM Query 23](#_Toc15570561)

[Реляционная БД 24](#_Toc15570562)

[Общие принципы 24](#_Toc15570563)

[JDBC 26](#_Toc15570564)

[JPA 27](#_Toc15570565)

[MongoDB 29](#_Toc15570566)

[Neo4J 29](#_Toc15570567)

[REDIS 30](#_Toc15570568)

[Функциональные возможности 30](#_Toc15570569)

[Общий подход 31](#_Toc15570570)

[Кэширование 32](#_Toc15570571)

[Простейшая постановка задачи 32](#_Toc15570572)

[Общий функционал 33](#_Toc15570573)

[Варианты развертывания кэша 33](#_Toc15570574)

[Стратегии кэширования данных 36](#_Toc15570575)

[Работа с данными 37](#_Toc15570576)

[Eviction/ Expiration 38](#_Toc15570577)

[Spring 38](#_Toc15570578)

[Декларативное кэширование 39](#_Toc15570579)

[Программное кэширование 39](#_Toc15570580)

[Infinispan 39](#_Toc15570581)

[Реактивная архитектура 40](#_Toc15570582)

[Основные достоинства 41](#_Toc15570583)

[Основные недостатки 42](#_Toc15570584)

[Взаимодействие с окружением 42](#_Toc15570585)

[Серверные архитектуры 43](#_Toc15570586)

[Сервера 44](#_Toc15570587)

[Netty 44](#_Toc15570588)

[Vertx 44](#_Toc15570589)

[HTTP клиенты 45](#_Toc15570590)

[Неблокирующий HTTP клиент от RxNetty (async-http-client) 45](#_Toc15570591)

[Новый стандартный HTTP клиент, доступный начиная с Java 9 45](#_Toc15570592)

[Retrofit - aсинхронный HTTP клиент со встроенной поддержкой RxJava 46](#_Toc15570593)

[Hystrix – надежная работа в мульти сервисной архитектуре 46](#_Toc15570594)

[Реляционные БД 46](#_Toc15570595)

[NoSQL 47](#_Toc15570596)

[Camel 47](#_Toc15570597)

[Тестирование 48](#_Toc15570598)

[Юнит тесты 48](#_Toc15570599)

[Комплексные тесты 48](#_Toc15570600)

[Срезы 49](#_Toc15570601)

[Тестирование контрактов 49](#_Toc15570602)

[Тестирование целостности данных 49](#_Toc15570603)

[Системные тесты 50](#_Toc15570604)

[Mocking 50](#_Toc15570605)

[Документирование 51](#_Toc15570606)

[Swagger 51](#_Toc15570607)

[Spring RESTDocs 51](#_Toc15570608)

[Приложение 1 – REST 53](#_Toc15570609)

[Основные принципы 53](#_Toc15570610)

[Коды возврата 54](#_Toc15570611)

[Успешное выполнение запрошенной операции 54](#_Toc15570612)

[Перенаправление 54](#_Toc15570613)

[Клиентская проблема 55](#_Toc15570614)

[Серверная проблема 55](#_Toc15570615)

[HTTP методы - стандартные операции над ресурсами 55](#_Toc15570616)

[POST - создание ресурса (или добавление ресурса в коллекцию) 56](#_Toc15570617)

[GET - получение представления ресурса 57](#_Toc15570618)

[HEAD – получение мета-информации 57](#_Toc15570619)

[PUT - обновление состояния ресурса 57](#_Toc15570620)

[DELETE - удаление ресурса 58](#_Toc15570621)

[PATCH - исправление ресурса 58](#_Toc15570622)

[Spring support for REST 58](#_Toc15570623)

[Контроллеры 58](#_Toc15570624)

[Конфигурирование приложения 61](#_Toc15570625)

[Использование RestTemplate в качестве клиента к REST 61](#_Toc15570626)

[Тестирование 62](#_Toc15570627)

[Best Practices & Use Cases 63](#_Toc15570628)

[HTTP запросы 63](#_Toc15570629)

[Stateless сервера 63](#_Toc15570630)

[Сортировка 64](#_Toc15570631)

[Фильтрация запрашиваемых данных: 64](#_Toc15570632)

[Валидация данных 65](#_Toc15570633)

[Поддержка долгих запросов 65](#_Toc15570634)

[Асинхронная работа 65](#_Toc15570635)

[Надежность при передаче сообщений 66](#_Toc15570636)

[Обработка ошибок 67](#_Toc15570637)

[Приложение 2 – работа с БД 69](#_Toc15570638)

[QueryDSL 69](#_Toc15570639)

[Основное API 70](#_Toc15570640)

[Продвинутые фичи 71](#_Toc15570641)

[Кодогенерация 71](#_Toc15570642)

[Интеграция со Spring 72](#_Toc15570643)

[Обновление БД 72](#_Toc15570644)

[Liquibase vs Flyway comparison 72](#_Toc15570645)

[Тривиальная реализация на основе Liquibase 73](#_Toc15570646)

[Полуавтоматическое обновление на основе Liquibase 74](#_Toc15570647)

[Приложение 3 - Bean Mapping 74](#_Toc15570648)

[Orika Bean Mapper 74](#_Toc15570649)

[Простейший случай 75](#_Toc15570650)

[Определение конфигурации маппинга А <=> B 76](#_Toc15570651)

[Определение конфигурации MapperFactory – глобальные настройки 77](#_Toc15570652)

[Кастомные конвертеры 77](#_Toc15570653)

[Кастомные мапперы и фильтры 78](#_Toc15570654)

[Тюнинг и проблемы 78](#_Toc15570655)

[MapStruct Mapper 78](#_Toc15570656)

[Конфигурирование мапперов 79](#_Toc15570657)

[Создание объектов 80](#_Toc15570658)

[Собственно конвертация 81](#_Toc15570659)

[Преобразование типов данных 81](#_Toc15570660)

[Маппинг вложенных объектов 82](#_Toc15570661)

[Повторное использование методов маппинга 83](#_Toc15570662)

[Кастомизация процесса маппинга 83](#_Toc15570663)

[Кастомизация процесса генерации 84](#_Toc15570664)

[Приложение 4 - Bean Validation 85](#_Toc15570665)

[Простейший сценарий 85](#_Toc15570666)

[Проверка бинов 85](#_Toc15570667)

[Определение ограничений 85](#_Toc15570668)

[Выполнение проверки 86](#_Toc15570669)

[Готовые ограничения 86](#_Toc15570670)

[Проверка методов 87](#_Toc15570671)

[Построение сообщений об ошибке 88](#_Toc15570672)

[Группировка ограничений 88](#_Toc15570673)

[Разработка ограничений 89](#_Toc15570674)

[Аннотация 89](#_Toc15570675)

[Реализация валидатора 89](#_Toc15570676)

[Шаблон сообщения об ошибке 90](#_Toc15570677)

[Композиция ограничений 91](#_Toc15570678)

[API для исследования ограничений 91](#_Toc15570679)

[Bootstrapping & конфигурирование 91](#_Toc15570680)

[XML конфигурация 91](#_Toc15570681)

[Программное конфигурирование 91](#_Toc15570682)

[Интеграция с другими фреймворками 93](#_Toc15570683)

[Расширения по сравнению со стандартом 93](#_Toc15570684)

# Основа

* Некоторые наиболее популярные подходы
  + На основе REST
    - Отдельные приложения на произвольных технологиях
    - Сервисы на основе AWS Lambda
  + На основе обмена сообщениями или событиями
    - Тоже REST, но за счет асинхронного вычитывания событий, публикуемых сервисами в виде фидов
    - MQ / Kafka сервер как базовый элемент инфраструктуры для произвольных сервисов
    - Система на основе Vert.x или чего-то подобного
  + В принципе OSGI контейнер вполне может рассматриваться как основа для множества микросервисов, развернутых в рамках одной JVM
* Кроме Spring Boot часто используются в качестве базового фреймворка Dropwizard и Spark (удобный DSL для Java / Kotlin, позволяющий начать с минимум кода)
* Быстрое создание приложения за счет **Spring Initializer**
  + Для эпохи монолитов довольно бесполезная фича – приложения создаются редко и инвестиции в конфигуратор заголовки не окупаются
  + При использовании микросервисов создание нового приложения становится частой операцией. Поэтому важно ввести единый стандарт на его организацию в зависимости от перечня базовых функцинальных возможностей
  + Доступен в виде веб-приложения, а также REST API, которое можно задействовать, например, за счет curl <http://start.spring.io>. Также Spring Tool Suite (IDE на основе Eclipse), предоставляет такую функцию, правда, также использует центральный сервис через интернет
  + Строит приложение, в корне которого находится командный файл maven / gradle wrapper-а. Использование данного wrapper позволяет избежать проблем с несовместимостью версий средства сборки на машинах разных разработчиков, а также на CI нодах.

# REST слой

* На основе **Spring Boot** с зависимостями **Spring Web (spring-boot-starter-web) + Spring Actuator** (подробнее см. приложение)
* Bootstrap класс с **@SpringBootApplication**, main метод которого запускает контейнер при помощи **SpringApplication.run()** метода
* Бины за счет классов с **@Component / @Service / @Repository / @Controller** или класса конфигурации с **@Configuration**, методы-продюсеры которого аннотированы **@Bean**
* REST контроллеры аннотированы **@RestController**
  + На методах **@RequestMapping(method=”…”)** для маппинга методов на шаблоны URL или конкретные аннотации типа **@GetMapping**
  + На аргументах методов **@PathVariable** для вычленения значений из URL, **@QueryVariable** для получения доступа к параметрам запроса, **@RequestBody** чтобы получить доступ к преобразованным к типу данных данным запроса
  + Методы возвращают **ResponseEntity<SomeType>**
    - Можно вернуть **ResponseEntity<Resource>,** взяв в качестве ресурса **FileSystemResource**, **GridFsResource, ClassPathResource, GzipResource** и так далее, что сильно сократит усилия по формированию byte[]
    - Для асинхронного ответа, чтобы не блокировать поток в пуле, ответственном за обработку входящих соединений, можно вернуть
      * **Callable< ResponseEntity<SomeType>>** - в отдельно сконфигурированном для **WebMVC** пуле потоков
      * **DeferredResult< ResponseEntity<SomeType>>** - запоминаем, что отдали, затем сами выполняем работу и вызываем **DeferredResult#setResult**. Особенно удобно, когда задание выталкивается для выполнения в очередь запросов и уже по получению асинхронного ответа мы по **correlationId** находим **deferredResult** и продолжаем выполнение
      * **WebAsyncTask< ResponseEntity<SomeType>>** - можно указать executor и временные ограничения на выполнение
      * Полезно с точки зрения сервера, но клиент все равно будет ждать в блокирующем режиме. Альтернатива – возвращаем **202 + Location,** где клиент со временем может получить статус обработки или конечный результат
* Преобразование данных из / в транспортный формат (обычно JSON, но можно что угодно) за счет различных реализаций **HttpMessageConverter<T>.** Их же надо симметрично регистрировать в **RestTemplate** на стороне клиента.
* Можно сразу репозитории работы с данными, экспортируемые как REST
  + Аннотированны за счет **@RepositoryRestResource**
  + Расширяют **JpaRepository** или что-то родственное

## Обработка ошибок

* **@ExceptionHandler** на метододах самого контролллера или **@ControllerAdvice**, если хочется централизовать логику обработки ошибок
* Официальног стандарта на представление ошибок кроме кодов HTTP не существует, но де-факто
  + MIME тип **application/vnd.error** (константы нет, создавать ручками за счет **MediaType#parseMediaType)**
  + Типы **VndError / VndErrors**, определенные в **spring-boot-starter-hateoas**, возвращаемые как **ResponseEntity<VndError>.** Вопреки общему заблуждению, любой код ошибки может сопровождаться телом ответа

## Гипермедиа (HATEOAS)

* Третий (высший) уровень зрелости сообразно модели зрелости REST Леонардо Ричардсона.
  + Предоставляет описание API, с использованием которого для перемещения между функциями сервиса не требуется знание структуры этого сервиса.
  + Если совсем вкратце, то в описание каждого ресурса входит набор типизированых ссылок, каждая из которых отвечает за какую-то операцию над данным ресурсом или передвижение к связанному ресурсу
  + Позволяет серверу произвольно менять схему путей, разносить реализацию разных операций между сервисами, переименовывать и менять иерархию ресурсов и так далее.
  + Основной минус, клиент теперь должен каждый раз вычитывать для ресурса / корня приложения все ссылки и выбирать из них нужную.
    - Иногда это надо делать много раз для продвижения по цепочке функций / ресурсов - /country/123/actors/search/by-movie?movie=car
    - В общем увеличивается число запросов
* Зависимость **spring-boot-starter-hateoas** предоставляет собственные типы **Resource / Resources**, которые позволяют вернуть объект в сопровождении коллекции типизированных ссылок в виде **Link**. Ссылки как правило формируют при помощи **ServletUriComponentsBuilder**
* Вместо **application/json** ответы имеют теперь тип **application/hal+json**
* Чтобы исследовать неизвестное API применяются HAL браузеры. Например, стандартный за счет зависимостей **spring-boot-starter-actuator + spring-data-rest-hal-browser,** доступен как **/browser**
* Для преобразования данных в ресурсы рекомендуется пользоваться специализироваными конвертерами на основе **ResourceAssembler**.
  + Данные конвертеры хорошее место для формирования того или иного набора ссылок в зависимости от состояния объекта
  + Например, если счет оплачен, то можно добавить ссылку на квитанцию
* Недостатки:
  + не поддерживает определение операций. Но, URL в определении ссылки может быть шаблоном с перечнем возможных параметров
  + Для HAL есть отдельный стандарт (не от автора исходного, автор против) для определения формата форм / операций
* Альтернативы:
  + **WADL**  - Web Application Description Language
  + **JSON-LD + HYDRA** – идеален для расширения уже существующего API с минимальным риском утраты совместимости. **HYDRA** дополняет **JSON-LD** в части описания возможных операций
  + **Collection+JSON** – хорошо определяет как отдельные ресурсы, так и их коллекции, для последних особенно подходит. Отдельно определяет допустимые запросы над коллекциями, а также шаблоны запросов на добавление / изменние элементов коллекции. Идеален для публикации редактируемых наборов данных
  + **SYREN** – хорошая попытка определить общие классы сущностей и возможных операций над ними. В отличии от остальных форматов, здесь у ссылки может быть целый набор **rel** значенией
  + **Mason** – также доступна возможности описания форм, с переченем элементов, методом вызова и типом кодировки. Уникальная особенность – возможность описания структуры возвращаемой ошибки
  + **UBER -** uniform basis for exchanging representations
  + **ALPS -** Application-Level Profile Semantics,автоматически поддерживается Spring Data Rest
  + **RAML** - RESTful Application Modeling Language
* К сожалению, нет никаких стандартных средств документирования структуры самих ресурсов. Варианты: XML Schema, Protobuf определение, JSON Schema

## Управление версиями

* Необходимость корректной поддержки изменений следует рассматривать в позитивном ключе, поскольку они подразумеваеют развитией. Если сервис не именяется, то скорее всего он просто никому не нужен
* Мартин Фаулер выдвинул идею «лояльного читателя» - клиент, который минимально накладывает ограничения на формат потребляемых данных. Например, вместо полной десериализации в объектный тип, чтобы вычитать order-id, применяет JSON-Path или XPath, чтобы игнорировать порядок элементов в документе и глубину расположения.
* Закон Постела, он же принцип надежности: реализации сервисов должны быть консервативными в том, что они производят, но либеральными в том, что они потребляют
* Версионирование сервиса за счет размножение кода достаточно трудоемко, особенно когда в течении переходного периода обе версии эксплуатируются и изменяются вместе. Вместо этого:
  + Обе версии разрабатываются в одном приложении
  + Зачастую, старые версии какое-то время частично маршрутизируют запросы к новым точкам или наоборот, в зависимости от приоритетов
* Способы кодирования версии: в URL, стандартном для приложении HTTP заголовке или в Mime типе, например application/vnd.demoApp-v1+json

## XML vs JSON

### Контроль формата

* Схемы в принципе есть для обоих форматов, но для JSON это скорее экзотика, а для XML хорошо отработанная практика
* Может стать серьезной проблемой для JSON в случае кроссплатформенных систем не совпадающих представлений элементарных типов для различных программных средств генерации / чтения
* В XML ошибки легче найти, сообщение парсеров информативнее
* Автор отказался от присвоения JSON формату номерных версий – отсутствует механизм внесения изменения, существует только один стандарт – текущий
* Стандарт определяется как минимум в шести документах с 2002 п2014 годы

### Удобство для разработчика

* JSON однозначно проще и компактнее.
  + Грамматика столь проста, что в свое время были популярны визитные карточки с ее изложением на обороте
  + Очень трудно в случае сложной XML схемы запомнить какие элементы за какими должны идти
* С другой стороны
  + Так как контролируется это схемой, то различные редакторы весьма облегчают процесс ручного редактирования / создания
  + если данными обмениваются две системы, то какая собственно разница. До момента, когда приходится ручками разбираться в том, что происходит.

### Стоимость

* работа с XML конечно тяжелее, как в смысле трафика, так и в смысле вычислительной нагрузки.
  + Но, как правило, для Enterprise приложений это не проблема, они и так медленные. Если же high load, то это становится проблемой
  + Продвинутые средства типа JAXB позволяют генерировать временные классы для разбора на лету, что сильно оптимизирует процесс
* И все-таки, JSON формат тоже достаточно избыточный, так как наименования свойств повторяются в нем опять и опять. Хотя, сжатие трафика в значительой степени это лечит
* Если компактность / скорость рабора становятся критическими для клиента или сервера, то можно вообще отказаться от текстовых форматов и использовать Avro, Thrift или Protocol Buffers.
  + Правда придется пожертвовать возможностью задействовать для отладки браузер / curl / postman
  + возможно в этом случае лучшим выбором будут также веб-сокеты или простые сокеты

### Простота и функциональность API

* для перелива данных в объекты и обратно есть средства для обоих форматов, а аналогично с Path средствами и работы с деревом представления
* API для работы с XML гораздо сложнее и далеко от интуитивности, но их больше, они стабильнее, предоставляют больше возможностей тюнинга результата
* В браузерах и JS available средах поддержка JSON нативная и быстрая. Правда это в свою очередь оборачивается проблемами с безопасностью.
* Ручная работа с XML в Java достаточно неприятная из-за сложности API, к счастью есть средства типа JDom, Dom4J и так далее, которые в некоторой степени это облегчают
* Для XML есть дополнительные развитые средства типа XSLT, XUpdate, XQuery, XPath и так далее. Мир JSON в этом смысле гораздо беднее
* Автор статьи «Парсинг JSON это минное поле» протестировал порядка 30 парсеров на предмет поведения в пограничных моментах и все они в чем-то да различались. Основные проблемные области парсинга
  + Поддержка скаляров на верхнем уровне – то есть просто «какой-то текст»
  + Замыкающие запятые - { “key” : “valeu”, }
  + Комментарии, замыкающие (//…) или даже встроенные(/\*…\*/) . Автор убрал их из стандарта, но многие парсеры это поддерживают как расширение.
  + Максимальная глубина вложенности. Некоторые парсеры не ограничивают глубину разбора, а просто накрываются к примеру на 1000-ом уровне
  + Специальные представления чисел – например: Nan, Infinity. Стандартом также не поддерживается, но некоторые парсеры…
  + Шестнадцатиричные числа – OxFF
  + Диапазон и точность
    - Представления с е-нотацией дробной части – иногда парсятся с плавающей точкой, иногда с двойной точностью, иногда как строки, иногда падаем.
    - Очень длинные числа могут быть восприняты как с плавающей запятой, так и в виде строки, в зависимости от парсера. Точность может быть урезана
    - Если у нас числа и с дробной частью и без, то в результате будут значения разных типов
  + Объекты с
    - повторяющимися ключами – в результате может браться первое значение, последнее значение или даже все с преобразованим в коллекцию
    - Ключи различные в двоичном выражении, но эквивалетные в строковом после нормализации UnicodeNFC (NFC / NFD представление)
  + Искаженные UNICODE строки и экранированные символы

### Расширяемость

* JSON однозначно лучше, так как
  + клиент спокойно обрабатывает избыточные / недостающие данные
  + если клиент браузер или просто понимает JS, то можно вставлять в документ фрагменты кода, собирающие данные из нескольких источников
* Это же одновременно и проблема, так как негативные последствия изменения модели данных видны далеко не сразу. Если авторы API добавляют / удаляют фрагменты данных, то довольно сложно понять все ли клиенты последовали рекомендациям и соответствующим образом обновились.
* С другой стороны
  + В XML можно добавить дополнительные элементы / атрибуты практически везде. В JSON для этого надо отказаться от скаляров и делать все значения объектами, чтобы было при случае куда подсовывать дополнительные ключи
  + Если добавляемое свойство пересекается по наименованию с уже имеющимся опять-таки проблема – отсутствие пространства имен как понятия
* Основная стратегия для «расширяемых» XML типов («новые» пакеты данных могут быть разобраны «старыми» клиентами и наоборот):
  + Добавляемые жлементы всегда опциональные
  + Последним элементом в последовательность всегда является шаблон расширения:

<**xs:any namespace="##any" processContents="skip" minOccurs="0" maxOccurs="unbounded"**/>

* + Основная проблема: в стандарте схемы 1.0 xs:any не может использовать после опционального параметра. Решение: формальный обязательный параметр «разделитель», обозначающий конец данных определенной версии - <**xs:element name="endOfVersionV1"**/>. Затем уже идет шаблон расширения
  + Таким образом, определение последовательности атрибутов типа выглядит следующим образом: атрибуты версии 1, разделитель версии 1 ... атрибуты версии N, разделитель версии N, шаблон расширения

### Безопасность

* В основном из-за слишком тесной интеграции с движком браузера.
  + Поэтому никогда не парсим его с помощью eval(), хотя это и быстрее всего. Вместо этого есть метод JSON.parse. Однако при этом изнутри документа могут быть вызваны функциии даже выполнен запрос на другой сервер.
  + Частая практика – вместо выполнения запроса со страницы к серверу, встраивать JSON документ в некоторый элемент страницы во время генерации, а потом получать его через innerHTML свойство элемента
* Это не так больно, когда интегрируются два сервера, но очень существенно, когда клиентом API выступает веб или мобильное приложение. Тем не менее, даже для сервера необходимо использовать OWASP JSON Sanitizer или что-то в этом роде
* Примеры
  + Атакующий скрипт переопределяет **Object.prototype.\_\_defineSetter\_\_()** и получает возможность получать в дальнейшем результаты выполнения **GET** запросов веб-приложения к защищенному API
  + Аналогично, но с переопределением конструктора **Array** объекта. Получаем возможность подслушивать процесс парсинга **JSON**

### Итого

* Как бы то ни было, но **Google Trend XML API vs JSON API** напоминает букву Х – последний начиная с 2012 года усточиво вытесняет первый
* XML однозначно имеем смысл выбирать, когда предстоит интеграция с корпоративными заказчиками / унаследованными системами. В других областях предпочтительнее JSON
* Передвать требуемый меди-тип результата можно как за счет постфикса наименования ресурса, так и за счет **Accept** заголовка.
  + Первое удобнее, впрочем можно использовать одновременно оба подхода
  + В заголовке можно указывать несколько типов, разделяемыъ «,», с указанием веса для каждого: **text/html, text/\*; q=0.1.**
  + Дополнительно можно укзывать: **Accept-language, Accept-Charset, Accept-Encoding**
* В итоге **POST / PUT / PATCH** должны требовать **application/json,** в противном случае возвращать 415 ошибку

## Бинарные форматы

* Все три формата (protobuf, avro, thrift) поддерживают схемы данных и процесс их эволюции в том или ином виде.
  + В отличии от JSON, для которого схема это экзотика, а единственная вменяемая стратегия версионирования заключается в том, чтобы только добавлять свойства и никогда их не удалять.
  + Причем схемы данных с набором встроенных типов и возможностью определять новые, что особенно ценно на фоне безграничной гибкости JSON
* Хорошо поддерживаются современными яхыками, типа Scala, Node, Go, Closue, Java и Open Source программными средствами. Плохо устаревшими и корпоративными платформами, веб-клиентами клиентами, хотя и есть некоторые подвижки.
* Это всего лишь распространенное заблуждение, что ресурсы должны быть в XML / JSON, они могут быть в любом удобном для задачи виде.
  + Хотя, конечно, возможность отлаживаться через браузер / postmann увы утрачивается.
  + Кроме того, когда все горит и пропало с JSON / XML обменом гораздо проще разбираться, что немаловажно

### Protobuf

* Идеология: структура сообщения берется из .proto описания, специальный транслятор может строить из него типы данных / конвертеры для всех основных языков программирования
* Компактный (только данные), поддерживает версионирование / слабую связность (данные расположены в пакете по стандартным смещениям, поэтому добавление новых не ломает клиент), если клиент его вообще не поддерживает, то Spring автоматически применит XML / JSON
* Опциональная поддержка схем, однако после генерации кода сериализации / десериализации они становятся не нужны.
  + Соответственно, на ходу обновлять схемы достаточно сложно и если мы захотели схему поменять, например, добавив дефолтное значение, то придется распространять обновление кода
  + Каждому полю класса присваивается соответствующий тег, которым помечаются поля в бинарном потоке.
    - Соответственно, при эволюции типа и поддержке старых данных важно не пере использовать старый тег еще раз.
    - Переименовывать поля можно сколько угодно – на бинарный формат не влияет. А вот менять тип нет – он определяет сколько данных будет считано из потока.
* Одинаково кодируются в потоке обязательные, опциональные и повторяющиеся поля.
  + При сериализации объекта с неопределённым значением обязательного поля будет ошибка
  + Если поле не имеет значения, то в потоке просто не будет фрагмента с соответствующим тегом. Соответственно, такое поле можно безопасно убрать из определения схемы
  + Проблема будет только, если какое-то поле опциональное для отправителя и обязательное для потребителя. Это надо учитывать при эволюции схем
  + Поле может быть превращено из опционального в повторяющееся и обратно без ограничений
* Имеется высокопроизводительный HTTP2 сервер от Google – **gRPC**, с поддержкой автоматической когенерации стабов и клиентов для множества языков.
  + Что особенно интересно, имеется проект **gRPC REST Gateway**, позволяющий автоматически строить шлюз для поддержки мобильных клиентов через **REST**
* Другие возможности для мобильных и веб клиентов
  + В 2016 году появился **proto3 JavaScript codegen**
  + **gRPC Web specification** – совсем навье, насколько оно рабочее не понятно
* Чтобы использовать в Spring контроллерах необходимо просто зарегистрировать **ProtobufHttpMessageConverter** на обоих сторонах

### Avro

* Сериализует / десериализует Java объекты в байты. Формат данных определяется схемой в JSON виде
* Требует наличие схемы И при сериализации И при десериализации
  + в потоке имен полей не содержится и формат очень компактный.
  + Схемы можно подгружать динамически, подстраивая парсер данных на ходу
* Стандарт специально определяет набор правил для правильного версионирование и достижения forward / backward совместимости
  + Чтобы конвертировать старые / новые данные читателю необходим доступ к обоим схемам.
    - Поэтому стандартом требуется иметь общедоступный набор схем. Это не сложно, да и вообще хорошая практика
    - К тому же в JSON схемы данных легко добавлять свои элементы и использовать не только для Avro, но и, например, для биндинга с GUI
  + Совместимость осуществляется за счет дефолтных значений, определенных в схемах. В старые схемы дефолтные значения можно добавлять по мере необходимости.
  + Чтобы переименовать поле надо вначале обновить всех читателей, сохраняя старое наименование поле как алиас. Затем обновляем всех писателей.

### Thrift

* Не просто библиотека сериализации / десериализации, но целый фреймворк RPC вызовов с дефолтной поддержкой двух JSON и трех бинарных.
* В целом подход к кодированию основан на тегах и близок к Protobuf

## Клиентская часть

* Для ручного тестирования обычно используем:
  + Старый добрый curl
  + Poster расширение Firefox / Advanced HTTP Client для Chrome
  + PostMan
  + SoapUI / ReadyAPI
  + HTTPie – Linux /macOS / Windows клиент командной строки с поддержкой:
    - Форматировнный и раскрашенный вывод в терминале
    - Встроенная поддержка JSON
    - Работа с формами и выгрузкой файлов
    - HTTPS, прокси
    - Аутентификация (basic / digest / password / pluging)
    - Произвольные данные запросов / ответов, работа с потоками данных
    - Кастомные заголовки
    - Поддержка сессий
    - Загрузка файлов по типу wget
    - Поддержка плагинов на основе Python
* **RestTemplate**
  + Перехватчики на основе **ClientHttpRequestIntercepter** позволяют применять кросс логику с запросам / ответам. Имеются стандартные для аутентифиакции BASIC / OAuth, GZip сжатия, разрешения имен сервисов по логическим именам и так далее
  + **getForObject()** позволяет преобразовывать ответ как к конкретному классу, так и к общему JSON документу в виде **JsonNode**
  + Чтобы получить HATEOAS ресурс надо
    - Создать параметризованный класс на основе **ParameterizedTypeReference<Resource<SomeType>>** и использовать его в **exchange() / getForObject()**
    - Этот класс позволяет передать информацию об параметризации, которая обычно теряется при компиляции. Шаблон “type token”. Сам **ParameterizedTypeReference** абстрактный, поэтому мы создаем на основе его конкретный, «захватывая» в него иерархию классов. Наиболее общий случай **ParameterizedTypeReference<Map<String, String>>**
  + Решение проблемы с множественными переходами по ссылкам при использовании HATEOAS – клиент **Traverson**, хорошо интегрированный с **RestTemplate**, позволяющий автоматизировать проход по цепочке ссылок с передачей параметров на основе шаблонирования.
* В Spring 5 новая альтернатива для **RestClient – WebClient**
  + Прежде всего поддержка реактивного асинхронного неблокирующего взаимодействия.
    - По умолчанию на основе Netty, но можно использовать и **Jetty reactive HttpClient** или любой другой на основе **ClientHttpConnector**
  + Токены **OAuth от Spring Cloud Security**
  + **HTTP/2**
* Возникает соблазн для каждого микросервиса сразу же реализовывать и предоставлять другим командам собственную реализацию клиента, абстрагирующую использование **RestTemplate**
  + Дополнительные расходы на поддержку его соответствия API сервиса, дополнительный риск ошибок и несоответствий
  + Есть риск сделать его слишком интеллектуальным, засунув в него бизнес-логику, которая должна по идее быть на стороне сервиса. Тогда у идущих напрямую через HTTP будут проблемы с одной стороны, а с другой клиенты реально станут зависеть от деталей бизнес-имплементации, что вообще плохо
  + Хорошая альтернатива – клиент на основе **Netflix Feign** (рассматривается дальше)

# Альтернативы REST

## REST vs RPC

* В принципе никто не мешает смешивать два подхода, хотя есть шанс в итоге собрать недостатки обоих

Есть проекты типа gRPS Gateway, позволяющие автоматически написать шлюз из REST API в RPC сервер

### Идеология

* RPC — прикладная задача моделируется произвольными функциями, методами объектов, обменом сообщениями. Как именно формально не определено.
  + Два основных стиля: URL + имя метода в составе параметра или каждый метод по своему URL. Второй способ предпочтительнее в части прозрачности, разграничения доступа, сбора статистики и так далее
  + Параметры соответственно в URL (для GET это единственная возможность) или в теле, в зависимости от того, какой HTTP метод применяется и насколько много параметров ожидается.
    - Можно в принципе задействовать и заголовки, но чего-то я примеров такого подхода не видел.
    - Соответственно возникает проблема со структурированными параметрами, причем как раз в query, где они широко применяются. В результате, встречается выполнение query через POST, что автоматически отрубает кэширование, даже если его хочется иметь
  + В отличии от RMI и CORBA клиент не получает ссылку на объект, посредством которой может им удаленно манипулировать. Кстати, REST в этом отношении как ни странно идеологически ближе.
* REST — ресурсами, отношениями между ними, выраженными через иерархию и гипермедиа ссылки, и ограниченным набором действий с общеизвестной семантикой и постусловиями.
* Эти способы взаимно выразимы друг через друга (REST можно построить на базе RPC и наоборот), но не одно и то же

### Маппинг на HTTP

* RPC как правило оперирует только GET (для запросов) / POST (для всего остального) методами, REST задействует все HTTP методы
* Соответственно, в случае REST клиент и промежуточные сервера лучше понимают семантику операций и могут использовать кэширование / повторный запрос про сбоях / условное выполнение операций и тому подобные возможности
* В случае RPC это возможно только для запросов, да и то не понятно, так как даже многие весьма уважаемые вендоры используют GET для запроса выполнения действий. В общем от греха подальше результаты RPC лучше вообще никогда не кэшировать.

### Использование

* RPC все методы определены заранее, порядок их вызова и ожидаемые результаты документируется внешними для системы средствами.
* В REST при помощи гипермедиа контролов (их наличия и отсутсвия) *сервер* как владелец ресурса определяет что может делать *клиент* в текущий момент.
  + Также это дает дополнительные возможности для эволюции системы
  + увы, до подобного уровня абстракции доростают не все API
* Последствия вызовов RPC методов не прозрачны и документированы вне системным образом. В случае с REST выполнение метода сразу очевидно отражается на данных ресурса / коллекции – прозрачность состояния приложения.
* Так как REST обменивается целыми представлениями документов, то его трафик обычно массивнее.
  + Но ничего не мешает использовать PATCH операцию и точно так же отправлять запрос только с некоторыми свойствами
  + В случае острой необходимости вызов метода всегда можно представить в ресурсной парадигме – журнал транзакции. Причем с дополнительными вкусными возможностями.
  + Также возникают проблемы, когда ресурс состоит из несколько частей, с различной политикой доступа к ним. Выход – представление их в виде иерархии суб-ресурсов.
* В случае RPC проще начать разработку – функции уже есть, надо их только выставить наружу и для этого есть много средств автоматизации (Spring XML RPC, Thrift, Finalge). Если есть толстый клиент, то можно воспользоваться предназначенными для него функциями.
  + Для REST надо специально подготавливать модель данных и представления бизнес-операций, это дополнительная работа
  + Простота RPC выходит боком
    - функции выставляют какие найдут и потом долго мучаются с неподходящими типами данных, болтливостью интерфейса и так далее
    - автоматические средства построения RPC обычно очень платформенно зависимы, интеграция между различными платформами может со временем обернуться большой проблемой
    - Разработчик становится зачастую не в курсе какие методы локальные, а какие удаленные. В результате возникают попытки вызова удаленного метода в цикле и тому подобные «артефакты»
  + В общем RPC это скорее способ интеграции, нежели построения общедоступного API

### Эволюция API

* в случае REST
  + реализация методов достаточно проста – доступных глаголов мало и идеология их маппинга на бизнес потребности хорошо определена.
  + Как правило, представление изначально содержит достаточно данных для поддержки самых разных, не известных на тот момент, вариантов использования
  + К сожалению это же и минус для REST – объем передаваемых данных и необходимые ресурсы для их обработки
* В случае RPC
  + богатство выбора и легкость реализации приводят к тому, что на руках со временем оказывается целый зоопарк функций, делающих примерно одно и то же.
  + Этому же способствует тенденция передавать / возвращать в методе только необходимые данные. Мнимая оптимальность служит драйвером неконтролируемой мутации
* В результате получается, что архитектура REST систем стандартнее, предсказуемее и прозрачнее для разработчика, так как собрана из практически идентичных компонент, оперирующих моделью данных. Вопрос в том, какой кровью достигается стройность модели данных, позволяющей полностью обслуживать бизнес процесс
* В случае REST разработчик / клиент понимают safe / idempotent / наличие side effect вызываемых методов, в случае RPC необходимо читать документацию
* С REST сложнее вначале, но накладываемая на разработчика модель ограничений позволяет со временем создать более элегантное и цельное API

### Примеры использования

* Опыт Linkedin
  + В ходе первых 7 лет RPC на основе Spring RMI с Java сериализацией. Опыт негативный: формат данных медленный и большой, переход на JSON дал значительный выигрыш, проблема с версионированием API и привязкой к Java платформе
  + В результате разработали собственный движок Rest.li с поддержкой асинхронности, типизированными запросами, автообнаружением ресурсов и так далее
* А вот Amazon AWS напротив от XML-RPC отказываться не собирается. Более того, они даже применяют GET для выполнения команд, что вообще ни в какие ворота не лезет. Однако, так как пользователь все одно работает через предоставляемую библиотеку, то он все этого безобразия не видит.

### RPC лучше использовать

* Для управления процессами
* Когда надо часто получать маленькие фрагменты короткоживущих данных (торговля акциями)
* для расширения унаследованных систем
* для быстро эволюционирующих систем
* Во внутренних API, когда обе стороны под вашим контролем
* при разделении монолита на составные части «малой кровью» и вообще, когда два компонента очень тесно связаны друг с другом, например фронтенд и бэкенд.
* В случае, когда взаимодействие между компонентами заведомо надежное, например, если оба разврачиваются на одном сервере
* Когда обе стороны основаны на одних и тех же технологиях. Это не обязательное требование, но если типы данных сложные, очень ускоряет работу
* когда на клиенте мало памяти / вычислительных возможностей / API для разбора больших документов
* Когда модель данных неочевидна, например, задачи визуализации данных или искусственного интеллекта
* когда логика в основном выражается элементарными операциями / командами – практически любая админка, например, для управления облаком
* требуется большая производительность и минимум передаваемых данных. Впрочем, тогда вообще с HTTP связываться не надо
* Если вы предполагаете, что клиент будет взаимодействовать с вашей системой только через предоставляемый вами SDK
* Когда предполагается использование протоколов, отличных от HTTP (например, веб-сокеты или JMS) , особенно протоколов двунаправленного взаимодействия
* Когда у нас много batch запросов или асинхронных нотификаций

### REST лучше использовать

* Для работы с хранилищами данных, особенно, когда данные надо сделать адресуемыми (банковская система или архив)
* Когда надо достаточно редко оперировать с долгоживущими данными
* В новых проектах
* Для внешнего API, когда вы контролируете только одну сторону и должны уважать клиентов
* В проектах с хорошо определенной моделью данных
* Когда заведомо много CRUD операций, например, управление НСИ
* Если очевидно, что API будет сильно меняться со временем
* Когда предполагается наличие клиентов на всевозможных платформах
* Если необходимо использовать кэширование, выполнение условных операций, отказоустойчивость, балансировку нагрузки, разнесение по географическим точкам и тому подобные встроенные в HTTP протокол возможности
* Если надо выдавать одни и те же данные в разных форматах или в разных представлениях. В случае REST на помощь приходят медиа-типы, в случае RPC получится зоопарк методов или параметров на методах

## GraphQL

* Язык запроса данных, нейтральный как к транспортному протоколу (обычно это RPC при помощи JSON запросов поверх HTTP), так и к реальному механизму постоянного хранения данных (обычно это собственная БД сервиса или обращение к любым другим сервисам)
  + Позволяет клиенту точно указать, какие именно данные ему нужны (smart projection) и за счет одного запроса получить агрегированные данные из нескольких логических источников / ресурсов.
  + Первоначально целью разработки была именно гибкая UI-ориентированная выборка данных, управляемая с клиентской стороны
* Идеология: вместо множества «глупых» endpoint (обычно REST ресурсов) иметь один «умный», который будет работать со многими источниками данных, формируя результат именно в том виде, в котором запрашивает клиент
* Первоначально разработан в facebok в 2012 год с целью оптимизации передачи данных (объем и количество запросов) между различными типами клиентов и серверной части поверх HTTP.
  + В 2015 году как спецификация выпущена в публичный доступ, с тех пор крайне популярен в мире JS, гораздо слабее в Java.
* Постановка проблемы:
  + В REST интерфейс сервиса решает какая именно информация доступна клиенту отдельно по каждому URL
  + Клиент при каждом запросе получает
    - во-первых, все данные ресурса, хотя возможно ему из них нужна только пара полей – «перевыборка».
    - Во-вторых, зачастую ему приходится делать множество запросов ко многим источникам данных – растет нагрузка на сеть. В худшем случае это N+1 запрос, например, перечень сообщений + данные по каждому из авторов для каждого поста. Или наоборот
* Кто использует GraphQL: GitHub, Shopify, Twitter и его платформа для мобильной разработки Fabric, Coursera, Yelp, Wordpress, Drupal, Intuit
* Альтернативный проект: Falcor от Netflix

### Основы

* В отличии от REST единая точка доступа, принимающая в основном POST запросы в виде JSON документов.
  + POST запросы могут как изменять данные, так и запрашивать
  + GET запросы только для query операций. Они мало применимы из-за ограничения на размер собственно запроса, передаваемого в URL
* Сильно типизированный язык - запросы должны соответствовать схеме (язык описания SDL), которая располагается на сервере.
  + Различные фрагменты схемы обрабатываются сконфигурированными для этого бинами – распознавателями (resolvers)
    - Распознаватели либо формируют запрашиваемый фрагмент данных, либо обновляют данные в хранилище. За счет этого естественным образом реализуется CQRS архитектура
    - Схема никак не связана с реальной схемой хранения данных, распознаватели преобразуют одно в другое, а также реализуют различные кросс-концепции, например, безопасность, кэширование или ограничение нагрузки. GraphQL это запрос к данным приложения, а не к данным хранилища
  + Запрос состоит из набора необходимых полей из числа определенных для типа данных и ограничений на них.
    - Например, эквивалентность или размер запрашиваемого массива.
    - Каждое поле в свою очередь может быть также запросом, позволяющим определить вложенный набор дочерних полей. И так далее – определяем иерархический срез от максимально доступного дерева данных.
* Клиенты имеют возможность получить схему для анализа и дальнейшего использования
  + Тестирование и документирование
  + построение UI на основе метаданных. Например, GraphiQL, браузерная IDE для выполнения запроса к произвольным GraphQL сервисам, или GraphQL Playground
  + Агрегирование нескольких GraphQL сервисов в один – stitching, предоставление комплексного API к множеству сервисов, расположенным за единым входом, определяемого консолидированной схемой данных
* Хорошее разделение ответственностей: сервер знает всю схему данных и как с ней работать, клиент знает, какие данные из всего возможного множества нужны именно ему
* Разработка сервера / клиента возможна в обоих направлениях: от кода к схеме и от схемы к генерации кода
* За счет **WebSockets** возможна подписка (непрерывное считывание) на получение данных с сервера
* Так как за получение каждого типа данных отвечает собственный распознаватель, то фактически модель данных описывает модель API приложения.
  + Если в произвольном JSON объекте все поля заменить на методы с пустым списком параметров, то получит описание интерфейса. Правда, для объектных полей будут возвращаться не значения, а ссылки на другие API
  + Если теперь к этим методам добавим параметры, то фактически получим GraphQL модель данных. Получается, что GraphQL позволяет выставить клиентам произвольный объект активной доменной модели, которая в свою очередь будет работать с каким-то хранилищем данных или другими сервисами

### Схема данных на основе JSON

* Перечисление – **enum SomeEnum { ITEM\_1, ITEM\_2 }**
* Стандартные типы: **Int, Float, String, Boolean, ID** (то же самое, что и **String**, но выделяется специально, так как если значение поля в результате работы распознавателя представлено как ID, то для него в свою очередь будет вызван соответствующий распознаватель)
* Определение типа - type **SomeType { propertyName: AnotherType}**
  + **AnotherType** может быть стандартным, например, String или Date, ранее объявленным типом или перечислением
  + **[AnotherType]** – поле является массивом
  + **AnotherType!** – обязательное поле
* DTO аргумент операции – **input InputName {аналогично типу }**
* Два специальных типа
  + **Mutation** – для операций изменения данных
    - Внутри операции типа **newSomething( input : InputName ) : SomeType**
  + **Query** – для операций запроса данных
    - Внутри операции типа **querySomething( input : InputName ) : SomeType**
  + Каждая из операций обрабатывается распознавателями соответствующих типов – **Mutation** или **Query**

### Запросы

* Все запросы начинаются с корневого, определяющего начальный распознаватель, внутри которого перечисляются необходимые поля: **{ something { field1, field2 } }**
  + В данном случае будет выполнено обращение к распознавателю **something**, после чего из результата работы для преобразования в JSON будут использованы только поля **field1 и field2**
  + Ответ будет в виде: **{ data: { something : [ { field1: “…”, field2: “…”}, { … }, …] } }**
  + Если **field3** не литерал, а в свою очередь объект, то
    - для него также можно задать проекцию **field3 { subField1, subField2 }**
    - получение данного поля также обслуживается соответствующим распознавателем
* Распознавателю можно передать параметры, в виде **{ something( id: “1”) { field1, field2 } }**
  + Соответственно, если поле является объектом, то распознавателю также можно передать параметры
  + Кроме того, на основании переменных могут срабатывать директивы, изменяющие схему данных результата выполнения запроса
* Перечень распознавателей на самом деле плоский, но за счет прикрепления к различным полям вложенных типов данных они в свою очередь образуют иерархию выполнения
  + Каждый GraphQL запрос, в отличии от REST запроса, вызывает совместную работу множества распознавателей. Своего рода мультиплексированный REST
* Запрос можно сделать именованным **query getSomething( $id: String ) { …. },** а значение переменной использовать внутри в качестве параметров любого поля, например **something( id: $id )**
  + Наименования запросов и мутаций крайне удобны, например, чтобы разбираться в логе или выполнять отладку
* Аналогично описываются мутации **mutation updateSomething( $id: String ) { updateSomething(id: $id ) { id, name } }**
  + **updateSomething** это соответственно тоже распознаватель
  + в данном случае мы также описываем проекцию результата, возвращаемого операцией обновления

### Минимальная разработка

* Зависимости
  + **graphql-java-tools** – реализация серверной части для Java. На основе библиотеки **graphql-java,** в принципе там есть все необходимое, чтобы поднять GraphQL сервер
  + **graphql-spring-boot-starter** – обертка для инициализации и подвешивания стандартного сервлета, реализующего протокол
  + **graphiql-spring-boot-starter** - для UI работы со схемой данных
* Конфигурационные свойства
  + **graphql.servlet.\*** - определение входной точки для обращения клиентов
  + **graphiql.\*** - определение публикации браузерной IDE для работы с запросами
* Для каждого типа данных / DTO делаем специальный класс данных **с get / set** операциями
* Query распознаватель
  + Реализует **GraphQLQueryResolver**
  + Для каждой из операций соответствующий по аргументам / возвращаемому типу метод
* Mutation распознаватель
  + Реализует **GraphQLMutationResolver**
  + Для каждой из операций соответствующий по аргументам / возвращаемому типу метод
* Бин типа **GraphQLErrorHandler** ответственен за обработку ошибок
  + **boolean isClientError(GraphQLError)**
  + **list<GraphQLError> processErrors(List<GraphQLError>)**
* В мире JVM клиентов мало
  + GQL, который увы на основе Groovy, поэтому подходит не всем
  + Apollo Android, Nodes

### Достоинства

* Более эффективная работа:
  + выбираем только данные, в которых нуждаемся, и только в нужных объемах. Уменьшается объем передаваемых данных
  + Одновременный запрос к нескольким ресурсам, в том числе к произвольно вложенным, уменьшается число запросов
* Наличие формального контракта между клиентом и сервером. Валидация данных, само документирование, использование в качестве метаданных и прочие прелести
* Работа всего API через единственный endpoint – сильно упрощает разработку клиента. С другой стороны, на сервере, вместо расползающегося множества ресурсов единая хорошо контролируемая точка входа
* Встроенное версионирование (точнее его отсутствие) на основе пометки полей как **@Deprected**, что приводит к появлению предупреждений на клиенте и дает возможность разработчикам подготовиться к будущей миграции приложения на обновленную схему
* Возможность естественным образом автоматически объединять элементарные GraphQL сервисы в более высокий уровень – stitching
* За счет гибкости развязывается разработка бэк- и фронтэнда
  + избавляя от необходимости реализовывать специальные ресурсы, оптимизированные под различные варианты использования (UI экраны). Или же сложную логику трансформации подобных ресурсов на основе передаваемых параметров.
  + Позволяя разрабатывать множество клиентов с различными требованиями к данными и единой точкой входа на сервере
* Развязывает между собой тип / представление ресурса и способ его получения. Позволяет на основании одной схемы произвольно комбинировать подходы: либо автор с вложенным набором статей, либо статья с вложенным набором авторов.
* API на основе REST это линейный набор ресурсов. Переход от ресурса к ресурсу возможен только на основе HATEOAS ссылок, которым все пренебрегают и каждый переход по которым – отдельный запрос. На основе GraphQL мы сразу имеем возможность определять произвольные срезы необходимых нам данных

### Недостатки

* В целом организация сервера становится значительно более сложной. Много ранее централизованной логики уходит на уровень отдельных распознавателей, которых в системе может быть тысячи и которые вызываются в контексте друг друга
* За счет гибкости порождает множество принципиально возможных запросов к реальной БД, стоящей за сервисом. Соответственно проблемы с оптимизацией запросов
* Необходимость ограничивать сложность выполняемых запросов не только «вширь», но и «вглубь». В противном случае имеем проблемы с полосой пропускания сервиса
* N+1 проблема остается, но переносится вглубь на уровень имплементации распознавателей.
  + Если распознаватель возвращает объект, свойство которого представлено коллекцией ID, то для каждого элемента коллекции в свою очередь будет вызван распознаватель, который выполнит свой запрос в базу данных
  + Оптимизирующий подход: вместо выполнения каждого запроса индивидуально накапливать идентификатор в промежуточной коллекции и запрашивать их все одним запросом – 1 + 1 запрос.
  + К сожалению такая оптимизация работает только на одном уровне – на каждом уровне вложенности объектов по одному агрегированному запросу, поэтому кэша второго уровня никто не отменял
* Проблемы с кэшированием – нет ключа / идентификатора ресурса, приходится выполнять кэширование на уровне фрагментов данных, что значительно сложнее
* Лишаемся возможности оперировать HTTP кодами, соответственно, вся промежуточная инфраструктура становится бесполезной
* Единая точка входа
  + не позволяет мониторить и управлять нагрузкой отдельно для каждого типа ресурсов. Все это опускается на уровень конкретных распознавателей
  + вся стандартная декларативная JEE безопасность сразу же накрывается медным тазом – надо встраивать во все распознаватели. Проблемы с разграничением доступа для вложенных фрагментов.
* Хорошо поддерживается JS сообществом и гораздо хуже Java. В частности, для JS есть отличные клиенты типа Apollo и Relay, для Java / Android все плохо
* Реализации GraphQL стандарта для разных платформ достаточно отличаются друг от друга
* Транспортный протокол может быть любым, а вот формат передаваемых данных только одним – JSON
* Проблемы с языком описания данных
  + Проблема с null: с серверной точки зрения нет возможности различить команду частичного обновления и команду удаления значения свойства. Приходится определять отдельные мутаторы для каждого такого свойства, используемые при частичном обновлении
  + Каждый объект приходится описывать два раза: один раз в качестве возвращаемого объекта, чтобы делать запрос, и один раз в качестве DTO, чтобы отправлять на сохранение / обновление
  + Интерфейсы поддерживаются, но только для результатов запросов. Поэтому зачастую приходится создавать DTO для обновления данных за счет кодогенерации
  + Дженериков просто нет, поэтому описать контейнер «каких-то» типов невозможно
  + Нет пространств имен, поэтому в реальной систем, когда количество типов составляет несколько сотен, становится крайне недобно
  + В целом это язык прежде всего ориентирован на запрос информации и только потом на несложный ввод

## WebSocket

* Пример: игровая платформа с миллионом подключенных пользователей. Создание HTTP подключения занимает примерно 10 мс. Если каждый пользователь шлет хотя бы десяток сообщений в секунду и для каждого сообщения создается соединие, то понятное дело, что в итоге все просто умрет
  + Можно использовать keep-alive и «постоянные» соединения, но каждый HTTP payload банально занимает больше места и дороже в обработке
  + HTTP/2 позволяет мультеплексирование на уровне соединения и выталкивание данных с сервера при помощи SSE. Если речь идет только о получении серверных уведомлений сойдет, а вот если идет интенсивный взаимный обмен сообщениями – увы не конкурент
* Применяется при создании серверных приложений, обменивающихся в реальном времени даными с миллионами подключенных пользователей. Пример: торговля акциями и колпьютерные игрушки
* Разрабатываются на основе таких технологий как Spring, JAX-RS, GraphQL, Spark, Jetty и так далее
* Двунаправленный, полнодуплексный протокол на основе постоянного TCP соединения.
  + Как правило применяется между веб-браузером и веб-сервером, хотя можно и организовать и сервер-сервер взаимодействие. Но в последнем случае могут быть проблемы с многопоточностью, так как пул клиентов организовать не удастся.
  + Нейтрален по отношению к формату передаваемых данных, часто применяется STOMP (Simple Text Oriented Messaging Protocol) субпротокол

### Сервер на основе Jetty

* Обработчик событий – любой класс с **@WebSocket** с методами
  + **@OnWebSocketConnect onConnect(Session)** – подключение нового пользователя
  + **@OnWebSocketClose onClose(Session, statusCode, reason)** – отключение пользователя
  + **@OnWebSocketMessage (Session, message)** – получение события от пользователя
* При подключении пользователя кладем сессию в **Map<Session, UserInfo>,** при отключении - удаляем, при поступлении сообщения – обрабатываем в контексте данных пользователя
* Рассылка сообщений: **session.getRemote().sendString(),** если надо бродкастить – крутимся по коллекции текущих подключений
* Чтобы протестировать подключение: CURL или веб консоль **www.websocket.org/echo.html**

### Сервер на основе Spring / STOMP

* Зависимость **spring-boot-starter-websocket + @EnableWebSocketMessageBroker** на конфигурации, наследованной от **WebSocketMessageBrokerConfigurer**. В ней переопределяем:
  + В **registerStompEndpoint()** за счет **StompEndpoitRegistery**.addEndPoint() определяем на корневой фрагмент URL, например, /ws => <http://localhost:8080/ws>
  + В **configureMessageBroker()** за счет **MessageBrokerRegistry** определяем
    - **setApplicationDestinationPrefixes()** – фрагменты URL, по которым сервер будет принимать сообщения, например, /app
    - **enableSimpleBroker()** – фрагмент URL, по которому клиенты будут разбирать широковещательные сообщения, например, /topic
* **@Controller** класс с методами
  + Принимают сообщения от клиента: **@MessageMapping(“/hello”)** + аргумент **Message**, могут возвращать результат
  + Выдают сообщения всем клиентам: **@SendTo(“/topic/greetings”)** + возвращают объект
  + Один и тот же метод может как принимать сообщение, так и выдавать его
* В результате, клиент будет подписываться на <http://localhost:8080/ws/topic/greetings> и высылать сообщения в <http://localhost:8080/ws/app/hello>

### Сравнение REST vs WS:

* Вообще-то сравнивать некорректно – REST это не столько протокол, сколько архитектурный стиль. Так что в большей степени мы будем сравнимать его с HTTP
  + Оба являются протоколом уровня приложений, работающим поверх транспортного (TCP), сессионного (Sockets) и наконец презентационного (SSL) уровней
  + WS это появился фактически расширение ограничений HTTP, но сейчас не является его заменой
* REST предлагает уникальные URL для каждого типа ресурсов + HATEOAS, чтобы, начав от стандартного входа, добраться до нужного ресурса или операции над ним. WS это как правило одна входная точка, принимающая все типы сообщений. В общем-то это больше RPC
* REST работает поверх HTTP, это stateless request / response протокол, где на каждую пару устанавливается отдельное TCP соединение, после чего оно закрывается.
  + WS один раз устанавиливает соединение, после чего использует его для обмена множественными запросами / ответами.
  + С одной стороны экономятся ресурсы на установку соединения. С другой HTTP/1.1 вводит концепцию постоянного соединения
* Обмен сообщениями:
  + В рамках REST нет стандартной возможности для сервера инициировать отправку сообщения. Обходные решения: SSE (Server-Sent Event), Comet и так далее.
  + WS позволяет клиенту и серверу по установленному соединению
    - слать сообщения друг другу независимо друг от друга – двунаправленная коммуникация
    - одна сторона не должна ждать окончания получения сообщения, чтобы продолжить взаимодействие – полнодуплексная коммуникация
  + HTTP/2 существенно расширяет возможности HTTP
* Оба позволяют работать поверх TLS за счет https и wss схем соответственно
* Итого:
  + Если надо просто время от времени получать данные с сервера, то лучше REST. Если речь идет о получаения потока данных в режиме реального времени – WS
  + Если инициатива исходит только от клиента и ограничивается простым получением ответа – REST, если инициировать взаимодействие могут оба и взаимодействие протекает одновременно в нескольких контекстах – WS

## Обмен сообщениями

* Cтандартные разновидности:
  + Работа с AMQP брокерами (умный брокер) – ActiveQM, RabbitMQ, QPiD, IronMQ
  + Системы стриминга сообщений (глупый брокер, умный клиент) – Kafka и родственники
  + Коммерческие системы типа Tibco, IBM MQ
  + JMS как одна из старейших и стабильнейших технологий

### JMS

* Верхнеуровневое API подобное JDBC
* **При использовании синхронных вызовов, также упоминаемых как “blocking call”**
  + По мере увеличения нагрузки требуется большая полоса пропускания от сети, следовательно критично добавление нового железа
  + Реализация проще
  + Более чувствительно к проблемам железа, БПС и сети. Если хотя бы одна сторона не активна – взаимодействие установлено быть не может
  + Сообщение должно быть обязательно подтверждено перед выборкой следующего
  + Типовые сценарии - Клиенту требуется результат обработки сообщения для продолжения выполнения операции
  + Используется для транзакционной обработке
  + В основном это REST, RPC, RMI, CORBA, EJB, WS и т.д.
* **При использовании асинхронных вызовов, также упоминаемых как “nonblocking call”**
  + Лучше адаптируется к изменению нагрузки, особенно в части сглаживания пиковой нагрузки
  + Реализация сложнее
  + Легче переживает временные отказы компонентов. Взаимодействие возможно при временной остановке одной из сторон. Информация не теряется, но ее доставка откладывается
  + Типовые сценарии
    - Когда ответ вовсе не требуется или требуется не сразу
    - Планируется обработка большого количества сообщений, возможно с пиками обслуживания
  + Железо должно использоваться более эффективно
  + Асинхронная гарантированная доставка

#### Основные механизмы взаимодействия

* **point-to-point – обмен сообщениями «один к одному»**
  + примеры - взаимный обмен сообщениями, получение транзакции из другой системы, отправка заказа на обработку в другую систему, supply-chain операция
  + подписчик гарантированно получает сообщения в том же порядке, в котором они были отправлены
  + можно просматривать сообщения вперед, например для определения их количества, но потреблять можно только в порядке FIFO
* **publish-subscribe (push-based model) – обмен сообщениями «один к многим»**
  + примеры – многоадресная рассылка в организации, рассылка информации заинтересованным потребителям, причем перечень заранее не известен, рассылка запросов на конкурентную обработку данных
  + Подписка может быть постоянной и временной (по умолчанию)
    - При временной получает сообщения только пока активен
    - При постоянной получает и те сообщения, которые пришли в момент неактивности – сильно нагружает систему обмена сообщениями

#### Apache ActiveMQ

* Наиболее старый и популярный Open Source провайдер службы обмена сообщениями
* Может быть как встроенный (удобно для небольших приложений и запуска тестов), так и развертываться отдельно
* Поддерживает как Java, так и не-Java клиентов по ряду транспортов: прямые вызовы внутри VM, TCP, SSL, NIO, UPD, multicast, JGroups, JXTA
* Поддерживает развертывание в кластере
* Расширенная работа с сообщениями: группировка, виртуальные / композитные **destination**, использование «\*» в наименованиях **destination**, возможность автосоздания **destination** на ходу
* Когда используется вместе со Spring / Camel поддерживает широкий спектр EIP (Enterprise Integration Patterns)

#### Поддержка в Spring Boot

* На классе со **@SpringBootApplication** добавляем **@EnableJms,** который регистрирует все необходимые бины и прежде всего
  + обработчика **@JmsListener** аннотаций
  + **ConnectionFactory и JmsTemplate**
* **@JmsListener(destination, concurrency , id, selector, containerFactory)** на любом методе компонента
  + Класс компонента НЕ должен реализовывать **MessageListener** и в идеале может вообще не быть связанным с JMS API
  + который может принимать
    - бизнес тип, запакоыванный в сообщение
    - **Message** и прочие служебные JMS типы при необходимости, например **Map** заголовков
  + Обработку таких методов осуществляет **JmsListenerAnnotationPostProcessor**
  + Недостаток: создаваемые контейнеры сложно тонко настраивать, а также запускать / останавливать / переконфигурировать по требованию.
    - Поэтому при необходимости стоит обратить внимание на **ServerSessionMessageListenerContainer**
      * Лучше ему сконфигурировать отдельный **ThreadPoolExecutor** с настроенной фабрикой нитей, чтобы нити имели специальные имена и было видно в дампе чем именно контейнер занимается
      * Если нужно подбирать «старые» сообщения, то можно по расписанию пересоздавать контейнер, каждый раз обновляя условие на получение «устаревших» сообщений
* Также необходимо создать бин на основе **JmsListenerContainerFactory**, отвечающий за создание и конфигурирование контейнеров, для этого необходимо:
  + Стандартная **connectionFactory**
  + **DefaultJmsListenerContainerFactoryConfigurer** выполняет конфигурирование фабрики исходя их свойств Spring Boot Based приложения
* **MessageConverter** достаточно выставить как **@Bean,** он автоматически подхватится
* Отправлка сообщений – **JmsTemplate**, основной инструмент для синхронной работы с сообщениями
  + в Spring Boot создается инфраструктурой
  + Предоставляет синхронные методы отсылки / приема сообщений
  + Выполняет за кадром всю тяжелую работу по открытию / закрытию / созданию инфраструктурных объектов
  + Конвертирует данные из объектов в собщения и обратно при помощи экземпляров **MessageConverter**
  + Позволяет использовать destination resolver для динамической маршрутизации сообщений
  + трансляция исключений из checked в runtime **JmsException**
  + позволяет зарегистрировать callback для слушателя
  + Если включить управление качеством за счет **explicitQosEnabled=true** то можно
    - Управлять хранением сообщений – **setDeliveryMode()**
    - Устанавливать TTL – **setTimeToLive()**
  + Недостаток **JmsTemplate** – агрессивной создает / закрывает ресурсы, такие как **connection / session**. Чтобы это вылечить, надо использовать не непосредственно **ConnectionFactory** (например, **ActiveMQConnectionFactory**), а оборачивать ее в **CachingConnectionFactory**, конфигурируемую единственным свойством – **nativeConnectionFactory**
* По дефолту транзакционный доступ
  + управляется за счет **@Transactional** на слушателе
  + Поддерживает распределенные транзации

### Rabbit MQ

* Написан на Erlang, одно из быстрейших решений в своем классе
* Взаимодействие основано на «key of excahnge» - иерархическое значение, за счет фильтрации по частям которого производится маршрутизация
* Топологий
  + Пользователь высылает сообщение в адрес Exchange (обменника), тот его перебрасывает в топики / очереди
  + Во время запуска конфигурируем правила роутинга: сообщения в адрес какого обменика на основании анализа ключа надо высылать в какие очереди / топики
* Реализация в Spring Boot
  + Свойства конфигурации spring.rabbitmq.\* + зависимость Spring boot
  + Бины типа Exchange, Queue – просто new + наименование
  + Бин типа Binding, создаются как BindingBuilder.bind(queue).to(exchange).with(“orders.\*”) и определяют правила роутинга на основе фильтра на ключ
  + Отсылка сообщений за счет RabbitTemplate, прием за счет @RabbitListener(queues)
  + Настройка за счет **RabbitListenerContainerFactory, SimpleRabbitListenerContainerFactoryConfigurer**
    - Можно определить несколько фабрик и в **@RabbitListener** на них ссылаться
* Сырой доступ за счет **AmqpTemplate, AmqpAdmin**

### Apache Kafka

* Может использоваться и как служба рассылки сообщений, но вообще-то средство рапределения потоков упорядоченных событий
* Если RabitMQ / JMS лучше использовать для микросервисных коммуникаций, то Kafka лучше для отправки данных на backend обработку:
  + Постпроцессинг обработанных в системе заказов
  + Расслыка уведомлений пользователям
  + Отправка лога и аналитики
  + Рассылка сообщений об ошибках и предупреждений в больших системах
  + Распределение данных, например, на основе CQRS архитектуры
* Основное достоинства Kafka
  + сообщения ВСЕГДА упорядоченны (в JMS порядок доставки не гаранирован, что впрочем не делает его хуже при правильном применении)
  + учет «полученных» сообщений ведется на клиенте. Соответственно, если он упал и поднялся, то можно посмотреть в собственную базу и дочитать упущенные сообщения. Некоторое сходство с HTTP feeds
* Работа в Spring
  + Свойства **spring.clod.stream.kafka.binder.\***
  + **KafkaTemplate** как способ синхронной работы с сообщениями
  + **@KafkaListener(topics, groupId)** – определение слушателя.
    - Гарантируется, что из всех слушателей с одним **groupId** сообщение получит только один
    - Если члены группы часто умирают / подключаются, то это очень плохо для производительности брокера, так как он при этом каждый раз переупорядочнивает внутренние партиции

# Валидация данных

* Для **POST / PATCH / PUT** запросов, в случае проблемы клиенту возвращается **422 Unprocessable Entity** и перечень ошибок с привязкой к структуре данных переданных аргументов
* Как правило на основе **Hibernate Validator (**см. подробное описание в приложении**)**, но зачастую необходимо строить собственный **Validator** сервис, когда правила проверки требуется делать более гибкими - опциональными / конфигурирумыми администратором
* Валидацию рекомендуется делать отделяемой кросс-концепцией (определять на уровне аргументов методов интерфейса, доступного как клиенту так и серверу), чтобы ее можно было дополнительно вызвать
  + на клиенте перед выполнение запроса (принцип fail fast), а также при вызове mocked сервиса
  + в Unit тесте, чтобы проконтролировать как минимум передачу обязательных параметров при вызове mocked метода
  + соответственно в самом сервисе тогда зачастую можно слегка сэкономить и запустить проверку «до первой ошибки»
  + Если проверки должны обращаться к БД или каким-то сервисам, то
    - Либо выполняем отделение операций через провайдеры данных, которые можно mock в тестах или иначе определеить на клиентской стороне
    - Либо разделяем проверки на две группы: внешние автономные и внутренние, которые могут быть выполнены только внутри соответствующего сервиса. Интерфейс, соответственно, может использовать только внешние проверки. Внутренние будут использоваться на имплементации интерфейса.
* Не рекомендуется делать валидацию централизованным системным сервисом, который будет с необходимостью вызываться в своб очередь каждым сервисом для проверки полученных аргументов. Это строго, красиво и энтерпрайзно, но убивает производительность, делает такой сервис единой точкой отказа / источником общесистемных изменений.
* Common библиотека + доступное всем хранилище конфигураций проверок + кэширование конфигурации - уже значительно лучше, но единый набор правил, зашитый в библиотеку, остается источником частых общесистемных изменений.
  + Возможно разумно разрабатывать и поставлять его в составе клиентского API к сервису. Потому что вообще-то данный набор ограничений и есть часть API – пред / пост условия
  + Проблема в том, что зачастую для проверки требуется удаленные обращения к другим сервисам и все API начинают быть связаными друг с другом. Лечится через разделение таких правил на логику проверки, которую помещаем в API, и логику получения данных, которую оставляем в реализации сервиса.
* Третий вариант это все делать на едином фасаде, через который гонять не только внешние запросы к сервисам, но и запросы сервисов друг к другу.
  + Такой фасад вообще очень полезная штука, так что можно его будет нагрузить еще и такой функциональностью.
  + Правда валидация, связанная с обращениями к другим сервисам, может его (и всю систему вместе с ним) значительно замедлить, а также послужить еще одним путем распространения деградации одиночного сервиса по всей системе. Поэтому необходимо такие обращения активно кэшировать и защищать от сбоев
* Неприятная проблема, когда имеем одну модель между клиентом и UI сервисами и другую между UI сервисами и бизнес сервисами.
  + Чтобы на клиенте показать ошибки в привязке к элементам интерфейса приходится транслировать propery path в описаниях ошибок, возвращаемых бизнес сервисами.
  + Значительно облегчается, если одну модель в другую транслируем программно. Тогда исследовав соответствующие мапперы можно автоматически исправить property path в возвращенном перечне ошибок
* Если проверяем большой граф объектов и имеем при этом много удаленных вызовов, то производительность значительно страдает
  + Хорошо, если данные для проверки можно закешировать и своевременно обновлять
  + В противном случае можно выполнить пакетирование проверок
    - Реализации соответствующих проверок ВСЕГДА возвращают false
    - Затем, получив перечень ошибок, отделяем специфические и выполняем пакетный запрос к удаленному сервису, чтобы проверить все данные одновременно. В итоге удаляем те ошибки, которые не подтвердились сервисом.

# Работа с хранилищами данных

* Наиболее классический подход – реляционная БД, но
  + Платой за универсальность является тяжеловесность
  + Отношения трудно нагружать типами, приходится заводить промежуточные служебные сущности
  + Недостаточно эффективно поддерживаются двоичные данные
  + Геоданные не стандартные
  + Чтобы обеспечить высокую производительность зачастую приходится идти на денормализацию данных и другие ухищрения
* NoSQL
  + Легковесные
  + Больше разнообразие моделей данных, позволяющее выбрать подходящую с минимальными издержками по производительности
* Для работы с хранилищами данных предлагается проект Spring Data
  + Поддержка порядка десятка различных технологий
    - Реляционные БД нативно посредством JDBC и через JPA унифицированно практически все основные БД. Основное достоинство: рабочая система, среда разработки и тесты могут использовать разные реляционные БД без необходимости изменения кода, чисто за счет конфигурации (профилей)
      * При тестировании, как правило, применяется H2 + @ActiveProfiles профиля с определением свойств локальной базы, у нее есть свойства инициализации, позволяющие имитировать некоторые нативные средства, к примеру, Oracle или MSSQL
      * При этом критично избегать нативных средств типа хранимых процедур и материальных представлений. Также надо внимательно смотреть на поддерживаемые транзакционные модели
    - NoSQL типа MongoDB, Redis, Cassandra и так далее более десятка официальных и еще не официальные (например, для MyBatis и JOOQ) проекты
  + корневая абстракция Repository
    - от которой растут более продвинутые: CrudRepositry, PageAndSortingRepository
    - Поверх идут уже конкретные репозитории, типа JpaRepository, MongoRepository
  + Одна из ключевых возможностей – трансляция разнообразных исключений в унифицированный набор
* Чтобы упростить создание служебных методов классов сущностей часто применяется средство шаблонирования исходного кода в процессе сборки Lombok
  + Типовые аннотации
    - @Data – создание get / set методов, toString(), equals() / hashCode() на основе первичного ключа
    - @NoArgsConstructor / @AllArgsConstructor
  + Даже при наличии анотаций можно определять собственные аналогичные методы

## Из REST в ORM и обратно

### Внутренняя модель данных <=> ORM Entity

* Что-то из арсенала **Spring Type Conversion / Orika / Dozer / MapStruct / ModelMapper / JMapper / Transmorph / ModelBridge / Nomin**
  + Неплохой очень краткий обзор со сравнением производительсности - <https://www.baeldung.com/java-performance-mapping-frameworks>
* Достоинства:
  + по сравнению с рукописными преобразованиями сильно компактнее
  + не приводят к слабо контролируемому размножению from / to утилит, код в которых либо дублируют друг друга, либо любое самое мелкое изменение ломает много трансформаций
  + легко читать и отслеживать изменения бизнес-требований
* Основная проблемы:
  + выглядит магически, сложно читать
  + непонятно где находятся в коде, плохо отлаживаются, особенно в случае runtime кодогенерации
  + нет поддрежки компилятора / IDE, проблемы зачастую выясняются только в runtime
  + Большая проблема, когда работаем с параметризованными generics – type erasure, что заставляет использовать весьма уродливые костыли
  + Если используется runtime кодогенерация, то инициализация с несколькими сотнями мапперров занимает много времени. Ленивая слежно реализуется и потенциальные ошибки могут вылезать тоже весьма лениво
* Чтобы проверить полноту маппинга **CDM -> JPA -> DB** и обратно:
  + За счет какого нибудь рандомизатора (например, **io.github.benas:random-beans**) заполняем CDM бины данными
  + Переливаем в JPA сущности, сохраняем в базу, выполняем запрос и обратный маппинг с CDM
  + За счие **DiffBuilder** из **XMLUnit** сравниваем CDM бины «до» и «после»

### Внутренняя модель данных <=> ORM Query

* Обычно вместо «сырого» **API JPA / Hibernate** используем что-то типа **QueryDSL**
* Три основных варианты
  + Запросы приходят уже в достаточно абстрактном виде, например, из UI, тогда построение запроса есть чисто техническая задача трансляции набора пар «поле / значение». Вполне можно строить JPQL запросы из строковых фрагментов
    - Не забываем, что при проходе «a.b.c» могут использоваться как внешние, так и внутренние объединения. В противном случае в результате пропадут объекты с а.b==null
    - Также при построении запросов избегаем Map структур, чтобы результат был детерменированным
    - Слишким гибкий вариант – не сможем построить все возможные индексы, чтобы не сваливаться в full scan. Поэтому применимо для UI, который будет использован ограниченным количеством пользователей
  + Маппинг из аргументов в бины – построители запроса (шаблон **Builder** – создаем бин, устанавливаем свойства – фрагменты запроса, затем метод **build(),** который за счет работы с **QueryDSL** выдает готовый запрос)
    - Недостаток: некоторая подковерная магия
    - Достоинства: все построители можно автоматически протестировать, они могут повторно использоть друг друга, набор возможных запросов ограничен
    - Перед выполнением **build()** маппим user context в уже заполненный построитель запроса. За счет этого применяем политику безопасности. Недостаток – она размазана по многим бинам и смешана внутри построителей с остальным кодов формирования запроса.
  + Ручное построение запроса из аргументов – наиболее гибки, трудоемкий и чреватый ошибками способ:
    - Необходимо все время помнить синтаксис JPA
    - Держать в уме доменную модель: классы, свойства, отношения, типы
    - IDE не помогает, во время компиляции ошибки не выявляются, только в процессе выполнения
    - Рефакторинг превращается в безумие

## Реляционная БД

### Общие принципы

* База данных только хранилище:
  + Не используем view / хранимые процедуры для бизнес транзакций
  + Можно применять для накопления данных для отчетов и интеграции с внешними системами через общее хранилище / db link
* Первичные ключи – UUID
  + Недостатки
    - Не очень удобно для разработки, поэтому для тестового развертывания можно применять отдельную стратегию
    - По некоторым утверждениям замедляют работу из-за повышенной нагрузки на индекс
  + Достоинства
    - ПК сразу доступен, не надо ждать выполнения SQL запроса, чтобы его вычитать
    - снимает все проблемы при репликации между несколькими базами данных
    - безопасность – невозможно осуществить вычитывание данных методом перебора ПК
  + Некоторые БД, например, PostgreSQL имеет специальный тип данных для хранения UUID, который надо явно включать при создании схемы
* По возможность без внешних ключей
  + ORM и так проверяет целостность данных, а они сильно мешаются репликации и обновлении структуры данных
  + Однако, индексы на ONE стороне необходимо создавать в обязательном порядке, чтобы избежать full scan при удалении записи по первичному ключу в дочерней таблице
* Валидация Hibernate / JPA бинов перед выполнением insert / update операции позволяет убрать некоторые ограничения с БД, которые могут
  + снижать производительность операций
  + не могут быть легко реализованы в силу эволюции структуры БД (совместное использование одной БД несколькими версиями ПО), стратегии наследования JPA сущностей или отсутствия доступа к ссылочным данным внутри БД в принципе
  + являются препятствием к репликации данных, налагая требования на точную очередность данных, ссылающихся друг на друга
* Сразу думаем о том, как будем
  + удалять данные
    - Для enterprise систем характерно их вообще не удалять, применяя «логическое» удаление
    - Проблема со справочными данными: удаленные записи «есть» при показе истории и «нету» при редактировании и добавлении данных
  + Версионировать данные
    - Типовой способ: история + единственный актуальный в данный момент экземпляр каждой сущности
    - Иногда приходится применять более сложный подход:
      * У каждой сущности есть два измерения: период актуальности «с» - «по» (логическое время) + время регистрации (физическое время)
      * Соответственно появляется возможность получения представления данных системы с различных точек времени: внесли изменение в данные за прошлый год, они должны учитываться в новых отчетах, но старые отчеты должны перезапускаться без изменения (учета внесенных изменений)
* Работа со сложными документами
  + Каждый документ хранит по возможности полностью собственные данные, копируя их их словарей во время создания / выполнения бизнес операций
    - Содержимое стабильное и в любое время может быть провалидировано за счет ЭЦП
    - Запросы документов проще и быстрее выполняются
    - Можно продублировать часть данных ссылками в целях оптимизации. Например, для raw level security
  + Не использовать наследование на уровне БД - каждый документ и каждый словарь в отдельной таблице. В противном случае имеем:
    - Высоко конкурентные «common» таблицы, данные в которых совместно используются многими документами
    - Сложные запросы к десятками join. Так как оптимизация в данном случае является О2 задачей по сложности, то Oracle быстро отказывается это делать и приходится заниматься ручной оптимизацией через Oracle Outlining
  + Каждый сервис хранит документы с наименьшей возможной гранулярностью:
    - Для поддержки UI вычленяем для отдельного хранения / индексирования только те свойства, которые планируем показывать пользователю. Остальные в CLOB
    - Для биллинга можно практически весь документ хранить в CLOB, все равно мы его в основном получаем по ПК, когда приходит событие перевода со статуса на статус
  + Перевод документа со статуса на статус хорошо делать через сообщение в очереди
    - Соответственно длинна очереди – показатель насколько тормозит операция соответствующей обработки
  + Общая схема работы с документами:
    - Формирование / проверка / подпись документа
    - выполнение учетных операций, в результате которых документ переходит в те или иные статусы, а в специальных таблицах, образующих замкнутый план / планы счетов, накапливаются проводки.
      * Важно, что сам документ при этом НЕ меняется, а проводки образуют историю его обработки
      * Проводки, кроме собственно данных операции обработки документа, содержат также аналитические признаки, которые затем нам понадобятся при составлении отчетов
    - Потом по проводкам собираем другие документы (например, начисления => платежное требование) или же формируем отчеты в различных аналитических разрезах
* Обязательно использовать для ORM кэш второго уровня (например, на основе Ehcache)
  + Либо развертываем как единый общедоступный инфраструктурный сервис, либо конфигурируем рассылку сообщений об инвалидации всем остальным серверам кластера.
    - Второе допустимо, если объем кэшируемых данных не слишком велик, они меняются редко и в кластере не слишком много серверов. В противном случае будет шквал сообщений.
  + относительного процесса кэширования данные разделяем на четыре группы
    - Принципиально не устаревающие системные словари. Например, «типы счетов»
    - Большинство остальных словарей, которые можно просто перечитывать каждые Х минут. От того, что данные будут изменены слегка раньше или слегка позже ничего не поменяется.
    - Операционные словари, например, перечень клиентских счетов. Автоматически инвалидируем кэш при каждой операции. Чтобы часто попадать в кэш надо стараться запросы относительного одного пользователя маршрутизировать на один и тот же сервер – sticky event “session”
    - Операционные данные – большинство документов, не имеет смысла кэшировать
* Все запросы выполняем с временным бюджетом.
  + Причем, применяем его не только на клиенте, но и на сервере / внутри самой БД.
  + Последнее может потребовать (например, для PostgreSQL) выполнения некоторых запросов при выделении каждого соединения с БД из пула
* Генерация отчетов
  + И вообще работа со всеми данными НЕ требующими строгой когерентности: если мы принимаем заказы к обработке, то клиент может слегка подождать, прежде чем у него обновится сумма бонусных балов
  + Эти данные отделяем от оперативных в отдельную базу «только на чтение» куда из основной базы реплицируем данные по мере возможности через асинхронную очередь
  + Отчеты / выгрузки генерируем из этой отдельной базе в основном через JDBC. За счет этого основные бизнес операции не тормозят

### JDBC

* Преимущества:
  + При работе с отчетами и сложными read-only запросами, выгружающими большие таблицы данных
  + ORM вообще-то приносит накладные расходы, как минимум на работу с ЖЦ сущности, а также трансляции JPQL запросов в SQL
  + Гораздо легче применяются оптимизации под конкретные БД, не надо возиться с диалектами, которые далеко не опции предоставляют
  + Так или иначе, но мы всегда нуждаемся в DTO
    - Структуры данных клиента и хранения изменяются независимо и в общем случае никак между собой не связаны
    - По соображениям безопасности, а также для предотвращения N+1 запросов с проходом дальше по графу, автоматическая сериализация крайне нежелательна. Частично помогает **@JsonIgnore**
    - На преобразование **DTO <=> JpaEntity** тратится много ресурсов – метаболизм оперативной памяти
* Конфигурация **spring.datasource.\*** (по умолчанию только единственная) + **JdbcTemplate**
* Чтобы создать и заполнить БД при старте в **src/main/resources** файлы **schema.sql / data.sql**, для тестов аналогично локальные
* Типовые методы
  + **execute()** для DDL и разовых обновлений / удалений / вставок
  + **batchUpdate()** когда надо работать со множественными записями
  + **query() + RowMapper** для выборки и трансформации в объекты. Если производительность не очень критична и структуры **DTO / JpaEntity** совпадают, то можно использовать **BeanPropertyRowMapper**, работающий на основе reflection.
* Имеется доступ **@Inject DataSource,** но лучше туда руками туда лезть только в крайнем случае. Например, для получения метаданных о базе / схеме данных
* Проект Spring Data JDBC поддерживает репозитории на основе упрощенного маппинга
  + Все типы репозиториев, возможность вернуть Optional / Stream, асинхронные запросы и так далее
  + Упрощенные сущности:
    - **@Id** на поле первичного ключа, используется для авто генерации ПК или получения из базы
    - **@PersistenceConstructor** на конструкторе со всеми атрибутами. Данный конструктор будет использоваться для работы с БД, можно определить и любые другие.
    - **SomeType withId(IdType id)** метод, создающий объект с теми же данными и другим первичным ключом, используется когда объект вставляется в базу данных
    - Наименования свойств класса должны соответствовать наименованиям полей в БД
  + Сложные маппинги не поддерживаются, с FK работаем вручную как с обычными полями
  + На методах **репозитория @Query(“select \* from Table where id=:id”),** на аргументах **@Param(“id”)**

### JPA

* Зависимость **spring-boot-starter-data-jpa** + какую базу используем, например, **h2 или mysql-connector-java**
* Конфигурация:
  + определения источника данных как для JDBC + spring.jpa.\*.
  + Также определяем конфигурацию провайдера JPA и в нем опцию создания при старте структуры БД на основании определения сущностей (для тестовой среды как правило, для рабочей возможны варианты)
  + Полезные свойства, устанавливаем в true практически всегда. Во всяком случае в процессе разработки точно:
    - **spring.jpa.hiberntate.show-sql**
    - **spring.jpa.properties.hibernate.format\_sql**
    - **spring.jpa.properties.hibernate.use\_sql\_comments**
    - **spring.jpa.properties.hibernate.generate\_statistics**
* Сущности - аннотации **@Entity, @Id, @GeneratedValue, @Column, @OneToMany** и так далее
  + В принципе эти же сущности поначалу можно использовать для генерации выходного JSON / XML и заменять специальной моделью данных API по мере необходимости и эволюции последней.
    - НО, не забывать в этом случае использовать **@JsonIgnore** на всех свойствах с ленивой загрузкой. Например, для всех коллекций без **Fetch.EAGER**
    - Аналогично данные свойства не должны использоваться в **toString(), equals(), hashCode()**
  + Lombok проект позволяет существенно сэкономить за счет автоматической генерации служебного кода: **@Data, @Builder, @NoArgsConstructor, @AllArgsConstructor, @ToString(exclude), @EqualsAndHashCode(exclude)**
* **Сложные маппинги**
  + **Один-ко-многим**
    - На one стороне где коллекция: **@OneToMany( fetch = LAZY, cascade = ALL, orphanRemoval=true, mappedBy=”свойство на many стороне”)**
    - На many стороне, где свойство – ссылка на родителя: **@ManyToOne( fetch = LAZY, cascade = REFRESH), @JoinColumn(name=”fk\_column\_name”), @JsonIgnore**
    - Грубая ошибка: **cascade = ALL** на many стороне.
      * Тогда при удалении даже одного ребенка будет автоматически удален родитель со всеми его детьми.
      * При последовательном применении можно за счет одной записи удалить всю базу
  + **Много-ко-многим**
    - Основной недостаток: однонаправленная связь
    - На обоих сторонах **@ManyToMany(cascade= {MERGE, REFRESH}, fetch = LAZY), @JoinTable(), @JsonIgnore**
    - Вырожденный случай, так как в большинстве случае в промежуточной таблице нужна какие-нибудь бизнес-поля. Например, дата создания связи. Поэтому вырождается в промежуточную сущность + две **@OneToMany** ассоциации с обоих сторон
  + **Один-к-одному**
    - Двунаправленная
    - С одной стороны: дополнительное поле – FK + **@OneToOne(cascade=ALL, fetch = LAZY), @JoinColumn(name=”собственный FK”, referencedColumnName=”ИД на другой стороне”)** на ссылке. Это будет владелец связи.
    - С другой: **@OneToOne(mappedBy=”имя ссылки на противоположной стороне”)**
    - Есть вариант, когда обе сущности используют один и тот же ИД, реализуется через **@MapsId.** При этом, соответственно, владельцем связи становится противоположная сторона.
  + **Маппинг из / в DTO**
    - Реализуем интерфейс **AttributeConverer<SomeType, String>**
    - В сущности атрибут типа **SomeType**, на нем **@Column** и **@Converter(converter=”реализация интерфейса”)**
    - Соответственно, все содержимое атрибута, представленного в виде DTO, будет храниться в одной колонке БД, как правило строкового типа. Очень удобно, например, при хранении JSON / XML представляемых документов
* Часто для всех сущностей выделяем родительский класс **@MappedSuperclass**, в котором
  + Определяем стандартный ПК, например, типа Long или String c UUID внутри (крайне неудобно при отладке, но неизбежно, когда используем репликацию данных между несколькими хранилищами, в тестовом режиме можно менять на **<ClassName>\_<Long>** для удобства разработчика)
  + Аудит за счет **@EntityListeners(AuditEntityListener.class), @CreateDate / @LastModifiedDate** на соответствующих полях. В конфигурации включаем за счет **@EnableJpaAuditing**
* Создание интерфейса репозитория
  + Расширяем **PagingAndSortingRepository<класс сущности, класс ПК>**
  + Добавляем абстрактные методы типа **findByXxxxAndXxxx** для простейших запросов. Запросы можно переопределить при помощи **@Query**
  + Добавляем дефолтные методы, которые реализуют более сложную логику работы с данными
  + В прикладном коде **@Autowired** интерфейс, бин реализации которого будет создан за кадром Spring Data
* **проблема N + 1 запросов**
  + Если выбираем несколько родителей у каждого у которых есть в свою очередь дети, то в результате будем иметь N+1 запрос – один на всех родителей и по одному для всех детей одного родителя
  + Варианты решения
    - JPQL: select p from Parent p left join fetch p.children
    - Выбираем всех родителей, затем детей при помощи IN (набор идентификаторов родителей) – два запроса. Затем ручками распределяем детей по родителям
      * Если БД преобразует IN в набор OR, то имеем постоянный hard parsing неповторяющихся запросов
      * У некоторых баз, Oracle, например, количество элементов в IN ограничено
    - Делаем нативный запрос совместно поля родителей (значения будут повторяться) и детей, сортируя сначала по ИД родителя, затем по ИД ребенка. Затем ручками создаем родителей (кэшируя уже созданных по идентификаторам) и детей (создавая каждый раз и добавляя к последнему родителю)
* Автоматическое преобразование в DTO
  + в JPQL запросе задать преобразование в DTO через конструктор: **select new DtoClass( t.name, t.id ) from Table t**. Соответственно, конструктор должен содержать аргументы для всех полей класса
  + Определение projection интерфейса
    - В нем get методы, с тем же наименование, что и методы в JPA сущности, а также аннотированные **@Value(“#{ выражение c target.??? элементами}”).**
    - Дальше методы репозиториев могут возвращать на JPA сущности, а projection интерфейсы, конвертирование будет производиться автоматически

## MongoDB

* Документарная база данных, записи хранятся в виде иерархии JSON документов
* Зависимость spring-boot-starter-data-mongodb + конфигурация spring.data.mongodb.\*
* Для разработки используем как правило локальную для разработчика инсталляцию, для запуска тестов – враппер, который на время исполнения тестов распаковывает и запускает локальный экземпляр (<https://www.baeldung.com/spring-boot-embedded-mongodb>), это общий подход для большинства NoSQL средств
* Реализация сущностей:
  + Вместо @Entity используем @Document
  + @Id по-прежнему, связи не аннотируем
  + Чтобы реализовать аудит придется создать @Component на основе AbstractMongoEventListener<базовый класс> в котором ручками обновлять соответствующие поля
* Аналогично репозиторий на основе PagingAndSortingRepository

## Neo4J

* Графовая база данных. Данные хранятся в виде узлов и отношений, которые в свою очередь тоже могут хранить простые и сложные типы значений
* Зависимость spring-boot-starter-data-neo4j + конфигурация spring.data.neo4j.\*
* Язык запросов Cypher, похож на SQL, но позволяет ручками определять план выполнения запроса = порядок обхода графа
  + Это позволяет оптимизировать доступ к данным
  + При определении запроса используется ASCII графика, позволяющая наглядно представить схему
    - Узел как (alias: nodeName)
    - Переход как (откуда)-[ r : тип\_связи { условие на используемую связь} ]->(куда)
    - MATCH (shipment:Shipment)-[r:HAS\_ADDRESS {type: “SHIP\_FROM”} ] -> (address) WHERE shipment.id=??? RETURN address
* Сущность
  + на основе @NodeEntity и @GraphId
  + На переходах @RelationShip(type, direction)
* Репозиторий на основе GraphRepository<класс сущности>, также можно ручками работать с Session, особенно в тестах

## REDIS

### Функциональные возможности

* удаленное хранилище быстро меняющихся активно запрашиваемых данных, только в памяти (виртуальную память было ввели, но потом от нее отказались), высокая производительность, однопоточная атомарная обработка клиентских команд на обработку данных – нет проблем с конкурентным доступом. Если уж совсем необходимо, то можно реализовывать сложные атомарные операции за счет Lua на серверной стороне, но они должны быть очень быстрые
* сотни гигабайт данных, сотни тысяч операций в секунду на экземпляр базы даных, в большинстве случаев чтение медленне чем запись
* Возможность ведения экземпляром до 16 баз данных. Каждая база – файловая структура, которую можно в любой момент скопировать без проблем
* различные стратегии периодического асинхронного сохранения на диск:
  + время от времени или по числу произведенных операций
  + сохранение снапшотов или файлы для постоянного добавления записей о запрошенных операциях.
  + различные режимы сброса на диск файловых буферов ОС (никогда, каждую секунду, после каждой операции).
* репликация (в том числе многоуровневая), средство не только бэкапирования данных, но и повышения производительности – читающих и ищущих клиентов можно запустить на slave. Двунаправленная master – master репликация не поддерживается
* клиентский шардинг (на какой сервер обращать операцию чтения / записи решается на клиенте на основании хэша ключа или других фрагментов даннх). Уже давно ожидается сервер с автоматическим шардингом, но для работы с ним имеются некоторые ограничения, например, при скриптинге
* модель данных не реляционная, но достаточно развитая, что позволяет гибко адаптировать под любую бизнес задачу
  + принцип ключ – значение, может достигать производительность memcached
  + время жизни данных (ключей) с автоматическим удалением
  + хранение значений основанно на пяти типовых структурах данных (string - они же числа, list, set, hash, sorted set – значения строки / числа)
  + предоставление клиенту набора стандартных атомарных операций для каждой из структур.
  + структуры в качестве значений хранят строки, в которые можно запихивать все что угодно, например XML или JSON. Но возможности добраться до отдельных полей этих данных нет.
  + В зависимость от размера экземпляров применяются разные модели хранения с автоматической конвертацией по мере роста
    - упакованный формат с потоковым чтением непрерывного массива данных или на основе различных списочных структур
    - в зависимости от объема данных и количества элементов лучше оптимизировать либо по скорости доступа, либо по объему хранения
* Publish / subscribe операции, блокирование операций чтения до появления в структуре реальных данных
* оптимистические блокировки данных пакеты команд пересылаемые как единое целое и обрабатываемые атомарно с блокировкой или просто так (выигрыш только в уменьшении числа сетевых вызовов).
* Пессимистические блокировки (и более сложные структуры синхронизации типа семафоров) могут быть легко реализованы на клиентском уровне
* скриптинг на LUA
* Используется в Tumblr, Disqus, Skype, StackOverFlow, Instagram, GitHub, Flickr, Digg и так далее
* Области применения:
* Счетчики
* Топы различных рейтингов
* Сессии посетителей и их перечни
* Кэши с инвалидацией по времени и по событиям
* Голосовалки
* Чаты, социальные сети
* Классификаторы
* Кэщирование данных из других хранилищ
* Зависимости **spring-data-redis, redis.clients.jedis, org.appache.commons.commons-pool2** предоставляют все необходимое
  + Поддерживается кластеризация Sentinel
  + Исключения преобразуются в в стандартные для Spring Data Access иерархию
  + Подписка на сообщения стандартным образом через MessageListenerContainer
  + Прозрачная сериализация / десериализация содержимого коллекций посредством JDK, String, JSON, Spring Object/XML
  + Поддержка транзакций за счет @TransactionalSupport
  + Возможность использовать Redis в качестве провайдера кэша

### Общий подход

* + **JedisPoolConfig** с определением как минимум максимального количества соединений
  + **JedisConnectionFactory** с указанием пула соединений, конфигурируем хостом и портом
  + Ссылающийся на нее **RedisTemplate**, используется как универсальный клиент, поддерживает практически все команды, трансляцию исключений и сериализацию / десериализацию
    - Пример - **redisTemplate.boundListOps(userId).leftPush(url.toExternalForm());**
    - Имеется наследниик **StringRedisTemplate** для работы со строками
    - За счет **RedisCallback** можно получить доступ к **RedisConnection**, так же как и для JDBC
  + Можно специальные коллекции в коде сразу связывать с объектами Redis

**@Resource(name="redisTemplate")**

**private ListOperations<String, String> listOps;**

**listOps.leftPush(userId, url.toExternalForm())**

* + Можно атомарные счетчики связывать с объектами Redis
  + Чтобы получать сообщения
    - создаем **RedisMessageListenerContainer**
    - объект со слушателем оборачиваем при помощи **MessageListenerAdapter** и добавляем в контейнер, указывая через **PatternTopic** наименование прослушиваемой очереди.
    - При ручном создании не забывать вызывать **afterPropertiesSet** методы для всех компонентов.

# Кэширование

* До последнего времени
  + Память дорога
  + Маленькое фиксированное количество экземпляров приложений, развернутых на солидных серверах
  + Масштабируемость в основном вертикальная
  + Стабильность каждого экземпляра приложения важна (если не критична) для жизнеспособности всей системы
  + Кэширование большей частью локальное для данных которые сложно вычислить или дорого достать снаружи
  + Реляционная СУБД как стандартное долговременное хранилище данных, разделяемое между всеми серверами системы
  + Каждый запрос клиента обрабатывается одной цепочкой серверов: web -> app -> db
* В настоящее время
  + Память уже достаточно дешева
  + Обычны большие кластеры динамического размера на основе дешевых серверов
  + В основном упор на горизонтальное масштабирование
  + Стабильные системы на основе не-стабильных компонент. Отказ экземпляра любого из сервисов, потеря части запросов / ответов – учитываемая в процессе проектирования норма.
  + Использование оперативной памяти узлов кластера в качестве постоянного хранилища данных вместо привычной файловой системы
  + Масштабирование целостной вычислительной нагрузки за счет разделения на отдельные задачи, распределения между уздами кластера и последующего сбора / обработки результатов (map-reduce)
* Типовая стратегия применения
  + Кэш второго уровняд для Hibernate
  + Обрабатываемые документы не кэшируем, они вряд ли будут часто повторно используемы
  + Основные системные словари кэшируем с длинным TTL, бизнес оперативные с коротким
  + Оперативные бизнес данные (свойства пользователей, эффективную политику безопасности, организационные структуры, перечни счетов и так далее) кэшируем, пока не получаем сообщение о необходимости инвалидации закэшированной информации
  + Кэшируем статические веб-ресурсы, считываемые с файловой системой
* Варианты: Infinispan, Redis, Hazelcast, ehcache, coherence, memcached, gemfire, terracotta

## Простейшая постановка задачи

* Имеется мастер сервис (М) владеющий данными и клиентский сервис (К), который хочет у себя данные кэшировать (например, в развернутым рядом Redis), чтобы
  + Сократить время выполнения операции, как за счет исключения частого вызова достаточно медленного сервиса, так и за счет возможности хранить для себя закэшированные данные в наиболее удобном для использовния виде
  + Защититься от временной недоступности (М) – лучше старые данные чем вовсе их отсутствие
  + Снизить нагрузку на М, в облачном окружении стоимость использования БД может быть сильно выше, чем кэша
* Собственно кэширование тут можно понимать достаточно широко. Под данный кейс подходят любые сценарии, когда два сервиса используют лигически общие данные, НЕ находящиеся в едином shared хранилище
* Основная проблема кэширования – как обновлять кэш, когда М сохраняет новые данные
  + М вызывает К и дает ему команду обновить / сбросить кэш. Достаточно плохо:
    - Два сервиса становятся сильно связанными. Причем как runtime (операция обновления М теперь будет зависеть от здоровья сервиса К), так и в отношении разработки / тестирования
    - Подход очевидно не гибкий – если еще кому-то понадобится кэшировать данные М, то тот теперь обязан знать и вызывать методы уже двух сервисов. На ходу не добавишь, а в результате будет иметь спагетти зависимостей, скорее всего зацикленных
  + М сам обращается в Redis и чистит кэш
    - Немногим лучше, так как М и К теперь зависят через структуру хранения кэшированных данных в Redis.
    - Ну и нарушается один из краеугольных принципов – каждый сервис ПОЛЬНОСТЬЮ владее собственными данными
  + Наиболее оптимальная стратегия – М публикует сообщение для всех, кого оно может потенциально заинтересовать, в общедоступной очереди / топике данных. Получивший сообщение обновляет / чистит локальные данные как ему заблагорассудится
    - Некоторая очень слабая зависимость между М и К сохраняется – формат сообщения. Поэтому лучше использовать форматы изначально поддерживающие версионирование. JSON в этом смысле не очень хорош, а вот Apache Avro, например, гораздо лучше
    - Также они оба зависят от наличия общей очереди / топика, но это уже инфраструктурный уровень, которым в данном случае можно принебречь

## Общий функционал

* Различные реализации могут поддерживать различный набор данных возможностей со своими особенностями и ограничениями. Например, **Guava** поддерживает только локальную изолированную работу, а **Infinispan** практически все дальше перечисленное
* Само приложение, как правило, оперирует только простейшими get / put / evict операциями над именованными сущностями – остальное в экстернализированной конфигурации кэша:
  + Как развертывается и взаимодействует между собой на разных нодах
  + Каким образом пер сущностно осуществляется работа с данными
  + Размеры регионов кэша и настройка количества служебных потоков
  + Транзакционность и так далее….

### Варианты развертывания кэша

* Standalone
  + Каждый экземпляр приложения содержит свой локальный экземпляр кэша, изолированный от других
  + Для хранения данных дорогих с точки зрения вычисления или запроса
  + Как правило совместимо с остальными возможностями, такими как write-through, eviction, участие в JTA транзакциях и так далее
* Cluster
  + Все экземпляры кэша знают друг о друге, способны взаимодействовать между собой с целью поддержания согласованности хранимого содержимого
  + Локальный встроенный режим (P2P):
    - Каждое java приложение содержит встроенный экземпляр кэша, исполняемый в той же JVM, ответственно за bootstrapping и интеграцию с окружением
    - Идеально для уже кластеризованных приложений: размер доступной для хранения данных памяти растет линейно с ростом числа экземпляров приложений
    - P2P взаимодействие вместо использование центрального координатора
    - Пример: распределенный кэш Hibernate второго уровня
  + Клиент / серверный режим
    - Единственно возможный, для не-java приложений
    - Частный вариант: кластер приложений + кластер экземпляров кэша
    - Идеален для
      * Когда накладываем безопасность на работу с кластером
      * Приложениям требуется для хранения данных много больше памяти, чем есть у них на нодах
      * Кэш предоставляется как сервис / инфраструктура, разделяемые между множественными системами
      * Cache as service / common enterprise infrastructure

### Стратегии кэширования данных

* **Инвалидация данных**
  + Когда сущность изменяется в любом экземпляре кэша, все остальные экземпляры информируются о даном событии
  + Снижение объема трафика
    - Multicast сообщения могут нести только перечень ключей
    - Значения могут загружаться экземплярами кэша лениво - “by request” / по необходимости
  + В основном используется в “read-mostly” сценариях и небольших кластерах
  + Недостатки
    - Не подходит для часто изменяемых объектов – слишком большой поток сообщений об инвализации
    - Некоторые среды развертывания затрудняют рассылку Multicast сообщений, приходится применять JMS очереди, например, или как-то еще изворачиваться, что сильно замедляет процесс
  + Пример: hibernate кэш второго уровня для редко изменяемых наборов данных, например: каталог товаров и цен, перечень пользователей и ролей
    - Goods and prices catalog
    - Users and roles
* **Репликация данных**
  + Каждый экземпляр кэша содержит реплики всех данных, хранящихся у соседей. Изменения данных, сделанные в одном экземпляре, должны быть отреплицированы по всему кластеру по всем экземплярам кэшей.
  + Применимо, если размер кластера невелик и каждый экземпляр приложения может получить выгоду от локального хранения в паями всех кэшируемых данных. Например: HTTP сессии в кластере веб-серверов, данные о резервируемых местах в процессе покупки билетов
  + Максимум производительности и живучести, но НЕ масштабируемо по памяти: добавление больше экземпляров кэша не увеличивает потенциальный максимальный объем хранимых данных
  + Запись может быть синхронный (ждем, пока данные не распределятся по всему кластеру, данные надежные и согласованные, но медленно) и асинхронный (запись быстрая, но есть риск потери данных и несогласованного чтения)
* **Распределение данных**
  + Обычно дефолтная стратегия в силу того, что единственная реально масштабируемая
  + Использует специальный алгоритм хэширования, определяющий исходя из ключа, на каких узлах кластера должны храниться данные - Hash Wheel.
    - Основная сложность: должен допускать возможность добавления / удаления серверов в кластере без необходимости перемещения больших объемов данных
    - Вместо хэширования ключа также применяются хэширование различных групп бизнес-данных, чтобы обеспечить совместное хранения одновременно изпользуемых / запрашиваемых данных – группировка данных. Основная проблема – как при этом не свалить все данные на один сервер.
  + Каждый фрагмент данных хранится на нескольких (обычно параметр конфигурации конкретного региона кэша) серверах кластера
    - Компромисс производительность и масштабируемость vs надежность:
      * с одной стороны, чтобы не потеряться, если одновременно выйдут из строя несколько серверов. Кроме того, возможно удастся также поднять производительность чтения за счет разброса запросов по множеству серверов кэша
      * с другой, если число слишком велико, то теряем масштабируемость и фактически переходим к стратегии репликации данных.
    - Часто применятся второй параметра: в скольких дата-центрах должны также храниться реплики данных, чтобы защититься от глоабальных проблем
  + Пример: использование кэша вместо СУБД в качестве единственного постоянного хранилища бизнес данных. Зачастую файловое хранилище при этом вообще не применяется – только одновременное хранилище в памяти множества узлов кластера.
* L1 (near) кэширование
  + Каждый узел кэширует в течении какого-то времени полученные данные локально и слушает события об их изменениях
  + Улучшает производительность для множественных повторных чтений удаленно хранимых данных, но увеличивает потребление памяти
  + Некоторый компромисс между “read-mostly” производительностью инвалидации данных и масштабируемостью рапределения данных
* В одном приложения для разных регионов обычно можно задавать разные стратегии:
  + Системные словари – устаревание на основе TTL
  + Стабильные бизнес словари – инвалидация
  + Нестабильные бизнес данные (обычно еще и с конкурентным доступом) – репликация
  + Стабильные бизнес данные – распределение + L1

### Работа с данными

* Cache-aside
  + Кэш сам по себе не взаимодействует с БД и используется как отдельное хранилище в памяти
  + Подходит для сервисов, которые являются единственными владельцами собственных данных
  + Пример - Hibernate / JPA кэш второго уровня
* Cache-through
  + Кэш является основным хранилищем данных при операциях чтения / записи и в свою очередь взаимодействует с БД
  + Варианты
    - Read-through
    - Write-through
      * Synchronous (клиент ждет вплоть до окончания операции записи в БД) /
      * Asynchronous (только если операция записи в БД не может кончиться ошибкой)
      * компромисс consistence vs latency
    - Cache-behind
      * Асинхронный Write-through
      * Реальная запись в БД выполняется НЕ при каждом обновлении данных в кэше, подходы:
        + В базу сбрасываются наборы измененных данных по планировщику или на основаании размера очереди «старых объектов».
        + Каждый раз в базу сбрасывается полное содержимое кэша / региона кэша, без учета его внутреннего строения
      * Подходит для часто изменяемых данных (сбрасываются не отдельные изменения, а конечный результат на момент синхронизации), значительно снижает нагрузку на БД
* Refresh-ahead
  + Кэш автоматически и асинхронно перезагружает данные из БД на основании TTL
  + Хорошо для часто используемых данных со сложной структурой и медленным хранилищем
  + Редкая возможность, например, реализуемая Terracota

### Eviction/ Expiration

* Цель одна и та же – избежать **OutOfMemory**, но способы принципиально разные
* **Eviction**
  + Каждый регион кэша ограничивается по общей памяти или числу экземпляров
  + Специальный процесс на каждой ноде кластера ответственен за удаление неиспользуемых сущностей из регионов. Иногда данная ответственность возлагается за процесс, обращающийся за данными (piggyback)
  + Стратегии
    - LRU – сохраняются последние используемые
    - LIRS – два региона: LIR для «горячих данных» and HIR для последних используемых, которые могут быть перемещеных в LIR. Сущности удаляются из HIR и хранятся в LIR как можно дольше.
  + Совместимо с хранилищем в БД: сущности могут быть сохранены / считаны из БД в процессе активакции / пассивации
  + Compatible with persisting in cache store – entries can be moved to / from during activation / passivation process
* **Expiration**
  + Сущности могут быть удалены после истечении некоторого периода времени (могут использовать совместно оба), который может быть сконфигурирован для региона или задаваться в ходе put() операции
    - Lifespan – предел времени хранения
    - maxIdle – предел времени не-активности
  + При удаленнии обычно не пассивируются и удаляются глобально, как из памяти, так и из постоянных хранилищ

## Spring

* Результаты выполнения методов бинов могут быть закэшированы.
* Поддерживаются множественные реализации технологии кэширования
  + За счет публикации бина с интерфейсом **CacheManager** при помощи **@Bean,** внутри метода выполняем настройку (например, определяем регионы и их свойства) или подтягиваем конфигурацию, определенную как ресурс
  + Наиболее распространенные
    - **SimpleCacheManager**
    - **EhCacheCacheManager**
    - **GemfireCacheManager**
      * Поддерживает транзакции за счет **GemfireTransactionManagement**
      * Распределенный кэш, поддерживающий одноврменное использование многих экземпляров
* Также поддерживаются технологии**: JCache, Hazelcast, Infinispan, Couchbase, Redis, Guava** (как раз была добавлена в Spring 4).
* Не большая проблема написать свой, например, на основе БД, интерфейс крайне простой для реализации, вопрос, насколько успешным он будет

### Декларативное кэширование

* Реализуется за счет АОП и прокси вокруг методов, которые в свою очередь обращаются к кэшу
* Конфигурирование на основе аннотаций
  + **@EnableCaching** на конфигурации или **<cache:annotation-driven/>**
  + Делаем метод кэшируемым - **@Cacheable** с атрибутами настройки
    - **value / cacheNames** – регионы кэша, в которых будут сохранены значения
    - **key** – SpEL выражение для построения ключа, например, **#person.id** для аргумента типа **Person** с соответствующим наименованием
    - **condition** – SpEL для определения условного кэширования **ДО** выполнения метода
    - **unless** – возможность определить условие кэширование **ПОСЛЕ** выполнения метода, SpEL выражение может использовать **#result** атрибут
    - **cacheManager** – наименование бина, реализующего кэширование, опционально, если используем одновременно несколько технологий
    - **sync** – синхронизация вызовов для предотвращения коллизий
  + Если на методе **@CacheEvict,** то **ПЕРЕД его вызовом** кэш чистится
    - Поддерживаются примерно такие же атрибуты, что и для предыдущей аннотации
    - Наиболее полезные – **value, key, condition**
  + Обновление значение в кэше - **@CachePut,** принципиальное отличие:
    - **@Cacheable –** может пропустить выполнение метода и вернуть закэшированный результат.
    - **@CachePut ОБЯЗАТЕЛНО** выполнит метод и поместит результат в кэш
* Конфигурирование на основе XML
  + Конфигурирование АОП **<aop:config>**
    - **<aop:advisor advice-ref= “cacheAdvice” pointcut=”execution(\* \*..MyService.\*(..))”>**
  + Конфигурирование политики кэширования **<cache:advice id=” cacheAdvice ” cache-manager=”..”>**
    - Внутри по регионам **<cache:caching cache=”…”>**
      * **<cache:cacheable method=”…” key=”…”>**
      * **<cache:cache-evict method=”…” >**

### Программное кэширование

* Получаем через DI доступ к **CacheManager**, у него основные методы
  + **Cache getCache(name)** – получение доступа к региону
  + **Collection<String> getCacheNames()** – получение перечня регионов
* **Cache** предоставляет базовые методы работы с кэшом
  + **Clear**
  + **Evict(key)**
  + **Get(key) / get(key, generator)**
  + **Put(key, value) / putIfAbsent(key, value)** – атомарность второй операции зависит от природы менеджера кэша
  + **getNativeCache()**

## Infinispan

* Declaratively and programmatic configurable cache regions
* Simple key-value access: Map-like API and Tree-like API
* Subscription on events
* Synchronous and asynchronous access
* Distributed ACID transactions, compatibility with any JTA-compliant transaction manager
* Optimistic and pessimistic blocking
* Data eviction, FIFO / LRU based
* Expiration of data based on lifespan and idle time
* Write through / behind cache store
* Persisting state to configurable cache stores, ready high-performance disk, JDBC and cloud storage based implementations
* Clustered and non-clustered (standalone) operation modes
* In cluster
  + embedded and client / server cluster modes
  + invalidation, replication and distribution strategies
  + L1 caching (“near-caching”)
  + synchronous and asynchronous network communications
* Elastic scaling
  + Common configuration: 300 nodes, 3000 caches, 20 TB memory
  + Largest: 1000 nodes
* Multiple access protocols: REST, Memcached, Hot Rod, Web Sockets
* Support for multiple network configurations
* Support for grid computations
* Query support through Hibernate Search / Lucene
* ORM support as NoSQL storage
* JMX statistics and monitoring
* Больше подробностей в отдельном дайджесте

# Реактивная архитектура

* Предназначена для реализации event based асинхронных композиций простых сервисов для получения более сложных, принимая во внимание обработку ошибок, работу с таймаутами, безопасность, высокую масштабируемость/ пропускную способность и так далее
* Основана на формировании композиции функций без глобального состояния , изменяемых данных и побочных эффектов (ну почти), а также на асинхронной обработке потоков сообщений, поступающих от источников данных многочисленным потребителям
* Типичные области применения
  + Неблокирующая работа с I/O диска или сети - неизбежно принимая во внимание задержки по пути и с другой стороны
  + Обработка любых событий / данных, поступающих приложению от производителя, которого оно не может контролировать.
    - Кроме бизнес событий (например, обработка GPS сигналов), сюда относятся инфраструктурные события (например, от файловой системы, пользовельского интерфейса)
    - Интеграция с сообщениями, поступающими от ESB или любых других шин сообщений
  + Поддержка большого количества клиентских соединения (от нескольких тысяч)
  + Сложная логика обработки событий
    - Если поток событий один или даже несколько, но они независимы друг от друга и логика обработки данных проста, то в большинстве случаев использование RxJava особой выгоды не принесет.
    - Но как только их надо комбинировать между собой, корректно обрабатывать ошибки и управлять ресурсами, то имеет смысл смотреть в реактивную область
* Устраняет **“callback hell”** проблему, которая появляется в результате попыток лобового императивного решения задач обработки асинхронных событий:
  + подписываемся на кнопку, в обработчике выполняем вызов удаленного сервиса и подписываемся на результат, в его обработчике в свою очередь подписываемся на возможность обновить интерфейс. Получаем последовательность вложенных друг в друга callback методов, которые трудны для понимания
  + аналогично при необходимости скомбинировать набор последовательных / параллельных вызовов за счет **CompletableFuture<R>.** Дополнительно имеем проблемы с необходимостью управлять (причем динамически, в зависимости от ожидаемой нагрузки) несколькими пулами потоков, чтобы разные по важности группы вызовов правильно разделяли общие ресурсы
  + Избавляемся от необходимости обрабатывать ошибки в КАЖДОМ из callback методе.
    - При лобовом подходе код обработки ошибок размножается. Кроме того, каждая функция обратного вызова должна знать о том, как ей обрабатывать ошибки. То есть она становится сильно связанной с вызывающей ее кодом
    - Все исключения в виде событий отправляются конечному клиенту, которые наверное знает, что надо делать
* Опыт промышленного применения в Java
  + Пошел в гору примерно в 2016 году, когда весь поток данных Netflix стал обрабатываться на основе RxJava (Hystrix, RxNetty, Mantis)
  + Крайне некоторые SQL предлагают RxJava API, например, Mysql и Postgesql
  + Многие NoSQL решения Redis, Couchbase и Splunk предлагают RxJava API для доступа к данным
  + Библиотеки CamelRx, Square Retrofit, Vert.x
* Основные реализации для JVM:
  + Flux и Mono из проекта Spring Reactor
  + RxJava
  + Flow в JDK9

## Основные достоинства

* Из-за массового перехода на микро сервисную архитектуру и повсеместное горизонтальное масштабирование теперь практически любые системы для ответа должны предварительно опросить через сеть какие-то другие сервисы, возможно параллельно или с зависимостями одних вызовов от других.
  + Позволяет комбинировать между собой потоки совершенно различных по природе данных: от REST доступных сервисов, баз дынных, JMS сообщений и так далее
  + Можно работать с данными «по частям» и для обработки, к примеру, первой строки не ждать, пока загрузится весь файл. Ускоряет реакцию приложения, снижает требования к памяти.
  + В итоге хотим от конкурентности перейти к параллелизму, чтобы обеспечить максимальную загрузку процессора имеющимися задачами, вместо того, чтобы блокировать нити и тратить вычислительные ресурсы на ожидание результата.
  + В итоге системы более отзывчивы, больше пропускная способность, ресурсы используются более оптимально
* Позволяет декларативно описывать логику обработки потока данных на основе выборки с таймаутами, кэшированием, обработки ошибок, выделением данных, фильтрацией, анализом состояния и аггрегированием.
  + Отдельное удобство: операторы обработки данных в общем случае не обязаны обрабатывать исключения, порождаемые кодом – источником данных. Это теперь обязанность конечного подписчика.
  + Позволяет легко навешивать на потоки любые необходимые кросс концепции: логирование, кэширование, контроль жизнеспособности, дефолтные значения и так далее. В некотором отношении это напоминает АОП
  + Тем не менее, ошибки можно перехватывать, преобразовывать, замещать дефолтными данными или другими источниками данных, повторять операцию получения данных и так далее
* В большинстве случаев освобождает разработчика от необходимости связываться с конкурентностью
* Новые типы потребителей сервисов для которых данный подход особенно важен: интернет вещей, коммуникация устройство<->устройство, мобильные приложения, потоковая обработка больших массивов данных

## Основные недостатки

* Высокий порог вхождения. Причем подразумевается, что разработчики уже знакомы с функциональным программированием, что добавляет проблем
* Событийно-ориентированные приложения сложно анализировать и отлаживать. Собственно все недостатки, которые мы имеем в Java 8 с переходом от циклов к потокам
* Stacktrace ошибок просто ужасен (реактивный функционал реализуется за счет многочисленных вложенных функций – оберток собственно бизнес-логики, которые его засоряют)
* Тестирование асинхронных приложений это сложно занятие, код теста может быть еще более запутанным, чем тестируемый код
* В некоторых случаях создать правильный Observable достаточно трудоемко из-за проблем с противодавлением и необходимостью очистки ресурсов после окончании работы с потоком.
  + К счастью, эта проблема не стоит на асинхронных по своей природе платформах
  + Кроме того, существует много библиотек и фреймворков, которые поддерживают RxJava изначально и скрывают от нас сложность задачи
* Сами по себе потоки в процессе работы вызывают повышенную нагрузку на GC, так как данные все время переупаковываются – общие издержки функционального программирования с immutable объектами

## Взаимодействие с окружением

* Идеология RxJava – все в мире представляет собой поток данных. Наибольшая выгода получается, когда потоки используются не точечно, а повсеместно – от получения входящих запросов, до клиентских вызовов внешних сервисов.
* Основная проблема современных Java приложений – блокирующее API ввода-вывода
  + Так как весь ввод-вывод упирается в сеть / диск, которые по своей природе асинхронные, то маскировка этого факта для синхронного клиентского кода обходится довольно дорого:
    - потоки регулярно простаивают в ожидании ответа, хотя могли бы в этом время заниматься обработкой ответов другим клиентов.
    - Такие потоки довольно трудно прервать
  + Клиенты, установившее соединение, также могут тормозить с отправкой данных. Из-за перегрузки или злонамеренно атакую. Соответственно поток также ждет готовности постепенно прибывающих данных
  + Как следствие: потоков нужно много.
    - Каждый поток это примерно 50 мб данных, их создание / переключение это значительная нагрузка на процессор и шину памяти (очистить / заполнить кэши, сформировать структуры), расход ресурсов на постоянную сборку мусора
    - Мало потоков – сервер не принимает входящие соединения, много потоков – еще хуже, все ресурсы тратятся на управление ими, соединения принимаются, но запросы обрабатываются очень долго
* Если мы сумеем ограничить число потоков, заставив их роботать постоянно, а не время от времени, то мы сильно снизим требования к памяти / процессору. Логика их работы будет принципиально другой:
  + вместо dedicated работы многих сотен потоков в контексте запросов различных клиентов с постоянным простаиванием в ожидании
  + событийное реагирование на основе нескольких десятков потоков, совместно обрабатывающих запросы всех клиентов
    - увидел, что на сокене готова порция данных - прочитал
    - затем быстро пробросил до прикладного кода, который применил к памяти (особенно для HTTP, где входящий запрос поступает кусочками) / осуществил неблокирующий вызов наружу / вернул ожидающему снаружи клиенту
    - пошел дальше на обработку следующего готового сокета
    - удержание открытого TCP/IP соединения в в смысле памяти обходится довольно дешево –несколько кб против несколких десятков мб под поток, его ожидающий в традиционной модели
  + основные проблемы подхода:
    - такой код писать / отлаживать / мониторить гораздо сложнее.
      * Мы больше не можем притворяться, что получение / отправка данных это то же самое, что вызов локального метода
      * Лобовой подход дает “callback hell” - добро пожаловать в реактивное (а значит и в функциональное) программирование
    - если где-то в клиентском коде затесался синхронный вызов (а это может быть далеко не очевидное место), обработка всех клиентов сразу значительно просаживается.
      * Пример: **URL.equals()** под ковром синхронно обращается к **DNS** серверу по сети. И даже если мы его впрямую не дергаем, а всего лишь имеем дело с **Set<URL>,** то мы уже крупно попали. Кроме того этот неприятный методы выдает разные результаты в зависимости от связности сети на момент вызова
  + Соответственно такой подход можно применить, если вся цепочка обработки клиента будет основана только на неблокирующих вызовах
* Современные новые API хорошо заточены под асинхронный неблокирующий ввод / вывод: серверные события, веб-сокеты (полнодуплексная связь), HTTP/2 (множество запросов / ответов паралельно под одному соединению)

## Серверные архитектуры

* Классическая модель – один поток на одно подключение.
  + Как результат: несколько ГБ оперативной памяти под стеки, все время сборка мусора (хотя память в стеках не очищается – много корневых и живых объектов), очень много процессорного времени тратится на контекстное переключение потоков
  + Хорошо работает до какой-то нагрузки порядка тысячи одновременных подключений. Особенно если применять долгоживущие TCP соединения типа HTTP + keep-alive или веб-сокеты
* Основной шаблон - **Reactor**
  + Классический Reactor: есть некий Event Loop, как правило однопоточный, который отвечает за обработку событий. Все клиентские запросы заходят как события. Далее выполняется обработчик, Handler, который подписан на соответствующие события. При этом будет нехорошо, если обработчик заблокирует Event Loop надолго. Поэтому долгоиграющие задачи делегируются Worker-потокам и выполняются, не блокируя Event Loop. На них повешен некий Callback, который будет вызван, как только задача будет выполнена (или прервется с отчетом об ошибке).
  + В свою очередь, Multi-Reactor расширяет этот шаблон (паттерн), добавляя еще несколько потоков (дополнительные Event Loop-ы). Таким образом, формируется шина событий (Event Bus) которая умеет масштабироваться под ресурсы конкретной машины. Как правило, количество потоков Event Loop определяется по формуле «количество ядер процессора \* 2».

## Сервера

### Netty

* Классический выбор для хорошо понимающих что они делают разработчиков и очень real time приложений. Например, для разработки сервера многопользовательской игры.
* Основан на двух пулах неблокирующихся потоков: один (как правило, в нем один поток) принимает соединения, другой (в нем потоков примерно по числу ядер) – разбрасывает их по обработчикам.
  + Обработчики последовательно совместно собирают данные из отдельных пакетов до полных запросов и обратно
  + В общем все очень по кусочкам, зато с полным контролем на каждом уровне.
* Минусы
  + Высокий порог вхождения
  + Недостаток документации. Есть «Netty in Action», но она уже довольно-таки устарела
  + Изначально рассчитан на работу с асинхронными сокетами. Расстояние отсюда до бизнес абстракций (даже просто до маппинга REST) увы достаточно велико
  + Сложная архитектура с многоуровневым наследованием. Для каждой абстакции четыре типа буфферов. В общем надо основательно въезжать
  + Очень легко устроить утечку ресурсов
* Достоинства
  + полный контроль над сетевым стеком. Возможность включать TCP\_FASTOPEN, TCP\_MD5SIG, TCP\_CORK, SPLICE, поддержку нативного Epoll транспорта, OpenSSL и так далее
  + Поддержка различных протоколов: TCP, UDP, UDT, SCTP? Легко. HTTP, HTTPS, HTTP/2, SPDY, Memcached, MQTT, HTTP/2 и так далее
  + Много готовых обработчиков: SslHandler, JdkZlibEncoder, ZlibEncoder, SnappyFramedEncode, JsonObjectDecoder, ReadTimeoutHandler и так далее. Соответственно пока в этом всем разберешся...
  + Быстрая: 200к на ядро для простейшего HTTP GET с задержко 2-3мс
  + Хорошая поддержка в сообществе
  + Поддерживается в Spring Boot 2

### Vertx

* Оегковесный фреймворк под эгидой Eclispe для разработки полностью реактивных приложений, работающих на основе JVM. В свою очередь основан на Netty и предлагает функционал более высокого уровня.
* Возможности
  + Компоненты приложения могут быть разработаны на Java, JavaScript, Scala, Python, Ruby, Groovy, а также Clojure;
  + Довольно-таки простая модель параллелизма, освобождающая от хлопот многопоточного программирования;
  + Простая модель асинхронного взаимодействия без блокировки, интеграция с RxJava
  + Распределенная шина событий, включающая как клиентскую, так и серверную стороны
  + свои интерфейсы для работы с файловой системой (синхронно и асинхронно), получения доступа к распределенным структурам данных (Map, Lock, Counter), разработки TCP, HTTP и UDP серверов и клиентов
  + Специальный Launcher, который позволяет создавать так называемые fat-jar, где точкой входа будет Main Verticle
  + Умеет работать с OAuth2, Shiro, JWT и так далее
  + Модуль Vert.x Microservices добавляет расширенный service discovery, circuit breaker и получение конфигурацию из множества доступных источников . Умеет интегрироваться с внешним discovery-сервером, например, Consul
  + Кластеризация прямо из коробки на основе Hazelcast, можно использовать Apache Ignite или Zookeeper
  + Более подробно https://dou.ua/lenta/articles/guide-for-vertx/
* Основан на шаблоне Multi-Reactor, весь Vert.x — это один большой Event Bus, с которым мы общаемся посредством Callback-ов.
* Более подробное интро - https://dou.ua/lenta/articles/guide-for-vertx/

## HTTP клиенты

* Стандартные блокирующие клиенты обходятся дешево в смысле потребления CPU, но имитация обычного процедурного вызова значительно ограничивает масштабируемость
* Так как один блокирующий запрос кладет весь неблокирующий сервер, то стандартный поставляемый JVM клиент вообще нам не подходит, так как от него требуется:
  + Поддержка большого количества независимых одномоментных запросов, каждый из которых в свою очередь требует нескольких клиентских обращений к сторонним API
  + Просто выполнение много клиентских HTTP запросов, например, в ходе выполнения пакетных операций. Поисковые роботы / службы индексирования держат постоянно открытыми тысячи / десятки тысяч соединений
  + Поддержка запросов к медленным сторонним API, удерживающим ресурсы клиента в течении долго времени
* В результате применения неблокирующих клиентов можно легко поддерживать десятки тысяч открытых соединений, простаивающих или получающих данные.
  + На поддержку каждого соединения требуется на несколько порядков меньше памяти и ресурсов процессора на обслуживание периодически просыпающегося потока
  + Лимитирующим фактором при неблокирующем подходе является не память, в вычислительная мощность процессора и пропускная способность сети
* Даже при применении неблокирующих HTTP клиентов надо быть очень осторожными, так как блокирующие вызовы могут притаиться в самых неожиданных местах – URL.equals()

### Неблокирующий HTTP клиент от RxNetty (async-http-client)

* **HttpClient().newClient().createGet().flatMap(response::getContent)**
  + В результате вызова HTTP метода получаем **Observable<ByteBuf>,** с которым можно по-разному в дальнейшем оперировать.
  + Например, складывать по порциям для получения всего запроса в виде строки - **response.map(bb -> bb.toString()).collect()**
  + обрабатывать по отдельным кускам при загрузке сохраняемого изображения.
* То есть даже простейший HTML документ по умолчанию скорее всего будет поставляться кусками по мере работы TCP/IP стека. Это и хорошо и плохо, в зависимости от того, что мы с ним дальше делаем
* Если нужны какие-то элементы данных целиком, то доступны операции буфферизации, но их нужно применять самостоятельно
* Еще компактная альтернатива - OkHTTP

### Новый стандартный HTTP клиент, доступный начиная с Java 9

* Это не обновление существующего API, а полностью новое (хотя и использующее некоторые классы старого)
* Поддерживает не только **HTTP/1.1,** но и **HTTP/2, HTTPS/TLS / WebSocket** протоколы
* Легче, проще, широко используются chained методы и построители объектов
* Подписка на асинхронные события, например, получение отдельно заголовки / тело / ошибки / trails
* Поддерживает обработку асинхронных серверных уведомлений (не порожденных запросом клиента)
* Работает как в синхронном, так и в асинхронном режиме

### Retrofit - aсинхронный HTTP клиент со встроенной поддержкой RxJava

* Вопреки распространенному заблуждению НЕ написана специально для Android и может быть использована где угодно
* Минимальные maven зависимости: **retrofit, retrofit-rxjava, converter-jackson**
* Стимулирует к type safe взаимодействию с REST службами за счет того, что требует сначала объявить специальный Java интерфейс для удаленного точки доступа, который потом транслирует в HTTP запросы
  + На методах интерфейса и их параметрах используются аннотации типа **@Get(path/{name}/other), @Query(name), @Path(name)**
    - Синхронное взаимодействие – возвращают **Call< SomeDataType >**
    - Асинхронное взаимодействие
      * Традиционный подход - принимают в качестве аргумента **Callback<SomeDataType>**
      * Реактивный подход - возвращают **Observable<SomeDataType>.** Вместо **Observable** можно использовать любые другие реактивные интерфейсы в зависимости от семантики методов
    - Конвертация данных запросов / ответов осуществляется за кадром
  + Хороший баланс между абстракцией (HTTP запросы как вызовы методов и строгая типизация параметров / результатов) и низкоуровневой детализацией
* Плюсы
  + моделируем неизбежную асинхронность процесса за счет **Observable**, не скрывая ее за дырявой абстракцией блокирующего RPC вызова
  + сложные операции, включающие в себя вызовы нескольких сервисов, могут быть определены за счет одной цепочки операторов

### Hystrix – надежная работа в мульти сервисной архитектуре

* Разработан для управления отказами и их изоляции в распределенных системах за счет оборачивания вызовов сервисами специальной логикой
  + Ограничение уровня конкурентности при вызове внешних сервисов
  + Идентификация некорректных действий за счет учета ошибок и таймаутов
  + Временное прекращение некорректных действий: постоянное и с использованием предохранителя (circuit breaker – прекращаем, периодически тестируем, если сервис возвращается в здоровое состояние - разрешаем)
* Также предоставляет некоторые дополнительные фичи
  + Пакетирование запросов путем объединения нескольких отдельных, поступивших за период времени, в один большой
  + Сбор, публикация и визуализация статистических данных
* В числе прочей полезной функциональности предоставляет **HystrixObservableCommand**

## Реляционные БД

* JDBC был выпущен в 1997 году, когда никто еще не был в курсе подобных проблем. С той поры он остается принципиально блокирующим API
* Объективные причины, почему JDBC остается блокирующим
  + Веб-сервер может поддерживать десятки тысяч соединений, по которым он периодически отправляет небольшие блоки данных, дешевых в получении
    - БД для каждого запроса выполняет следующие операции: разбор запроса (CPU bunded), оптимизация запроса (CPU bunded), исполнение запроса (I/O bounded), сериализация и отправка данных обратно клиенту (net bounded)
    - Большая часть ресурсов сервера БД тратится на исполнение запроса, а диски, даже SSD, плохо поддерживают распараллеливание доступа
  + В результате сервер БД принципиально не может обработать больше нескольких сотен запросов в секунду (ну или больше, но все равно не сравнимо с пропускной способностью HTTP сервера).
* Таким образом, авторы считают (признаться не очень понимаю, чем неблокирующий вызов длинной операции по HTTP принципиально отличается от организации неблокирующего вызова БД) не имеет особого смысла писать неблокирующие полностью реактивные драйверы.
  + Даже в библиотеке Slick из известного реактивного стека Lightbend на основе Akka на нижнем уровне используется стандартный JDBC
  + Хотя они есть, в основном для Postgers, Mysql и не-реляционных БД
* Основная рекомендация:
  + использовать выделенный тщательно настроенный пул потоков, в котором изолировать код, выполняющий блокирующиеся на JDBC операции. Размер пула должен коррелировать с размером пула соединений с БД.
  + Тем не менее, это имеет свое отрицательное влияние: на управление потоками из данного пула будут тратиться дополнительные ресурсы
  + Собственно примерно это делает Node.ja – все блокирующие I/O операции выбрасываются для выполнения в отдельный пул потоков, а потом уведомляют основной рабочий неблокирующийся код о завершении
* Есть стандартные мосты между RxJava и JDBC
  + **rxjava-jdbc**
  + **R2DBC** (Reactive Relational Database Connectivity) – нативно поддерживает PostgreSQL, MSSQL, H2, а также работает в прокси режиме
  + **async-jpa**
  + **Reactive JPA** из последней JEE спецификации
* В мире Spring
  + Подключаем Reactor
  + **Io.r2dbc** + драйвера, например, для **PostgreSQL**
  + Конфигурируем **PostgresqlConnectionConfiguration** для **r2dbc**
  + JPA сущности стандартные, а вот репозиторий теперь расширяет **ReactiveCrudRepository**, методы которого возвращают **Mono<JpaEntity>.** Чтобы их инициализировать на конфигурации определяем **@EnableR2dbcRepositories**
  + Методы контроллера теперь тоже должны возвращать **Mono<SomeDto>**

## NoSQL

* Документарные Couchbase и MongDB предлагают не просто реактивную обертку над блокирующимися вызовами, но полноценный асинхронный доступ к сервису
* Результаты вызова приходят в виде JSON документов, чтобы их анализировать приходится залезать внутрь за счет **map / concatMapIterable / filter**
* Операции обновления также реализованы в виде операторов, возвращающих **Observable<Success>,** поэтому в виде потока можно описать целый процесс выборки, агрегации, вставки, запроса связанных данных и так далее
* Так как работа с каждым типом документов представлена в виде отдельного потока, то их можно произвольно комбинировать между собой и с любыми другими потоками данных не зависимо от природы их источника

## Camel

* Предоставляет богатый набор компонент интеграции, позволяющих унифицированно подключаться к более чем 200 платформам на основе абстракции обмена сообщениями. Например: AMQP, Amazon WS, Cassandra, ElasticSearch, файловая система, FTP, Google API, JDBC, Kafka, MongoDB, SMTP, XMP и так далее.
* Большинство этих компонент может работать в активном режиме, «проталкивая» поступающие сообщения клиенту. Встроенная в Camel поддержка RxJava предоставляет нам идеальный способ построения горячих Observable
  + Строим и конфигурируем **DefaultCamelContext**. В нем можно определить целые цепочки получения и пребразования сообщений. Это все-таки как никак а ESB, пусть и достаточно легковесная
  + Затем **new ReactiveCamel(context).toObservable(url)**
    - Формат URL определяет откуда тянем данные. Например, если никаких цепочек специально не сконфигурировано, то можно просто запросить файловую систему: **file:/home/user/tmp**
    - В результате получим **Observable<Message>** , который будет содержать наименования новых файлов в данной директории
    - В URL можно задать дополнительные параметры типа **?recursive=true&noop=true** и так далее
    - Для каждого типа источника данных есть свое пространство имен, определяющая возможности подключения. Например, для Kafka – сервер, порт, топик, идентификатор группы. В результате имеем опять-таки **Observable<Message>**
* Отличная единая точка для унифицированного подключений множества технологи интеграции в приложение. Скрывает подробности реализации подключения за единообразным интерфейсом. Позволяет за счет Camel функциональности описывать очень сложные технологии интеграции.

# Тестирование

* Разновидности тестов: юнит тесты, комплексные и системные

## Юнит тесты

* Зависимость **spring-boot-starter-test**, на тесте **@RunWith(SpringRunner.class)**
* Контейнер при этом не стартует, контекст приложения не доступен, **@Autowired** не работает
* **@MockBean** на свойствах класса позволяет за счет **Mockito** создать заглушку каких-то сервисов, дальше на их основе кофигрируем тестируемый компонент и прогоняем тесты
* Чтобы иметь возможность сделать DI в тестируемый компонент, он должен принимать подобные зависимости через конструктор. Так как эти зависимости являются для него обязательными, то это нормальная практика

## Комплексные тесты

* Зависимость **spring-boot-starter-test,** на тесте **@SpringBootTest + @RunWith(SpringRunner.class),** чтобы Spring при старте нашел конфигурацию бинов необходимо, чтобы корневой пакет приложения и тестов был один и тот же
  + Атрибут **@SpringBootTest#webEnvironment** определяет как будем эмулировать веб слой
    - **MOCK** – имитация сервлетного контейнера. Быстрее, но часть функционала не доступна
    - **DEFINED\_PORT / RANDOM\_PORT** – реальный сервлетный контейнер, медленне, но надежнее в смысле соответствия реальной работе
    - **NONE** – сервлетный функционал не доступен
* Соответственно при этом инициализируется контекст приложения, работает **@Autowired,** стартуют и загружаются файлы конфигурацию не только из приложения, но и из тестового кода, что дает возможность
  + перманентно подменить сервисы, работающие с внешним окружением, на mock-и
  + за счет активизации тестового профиля заменять целые фрагменты функиональности

### Срезы

* До Spring 1.4 тестовый контекст стартовал целиком, после была возможность инициализировать только часть его – тестовые срезы, что ускоряет процесс. Конфигурируются дополнительной аннотацией на тесте. **@SpringBootTest** при этом НЕ применяется
* **@JsonTest** – тестирование только сериализации / десериализации
  + Внутри теста работаем через **@Autowired JacksonTester<SomeCLass>**
  + Можно перливать данные из строк в бины, тестировать значения в том числе и по path, загружать данные для тестировния из ресурсов приложения и так далее
* **@WebMvcTest(SomeController.class)** – тестирование только отдельно взятых контроллеров, то есть тестирование себя в качестве сервиса
  + Внутри **@Autowired MockMvс,** который мы используем в качестве клиента для вызова контроллера через HTTP
  + Сервисы, которые должны быть в свою очередь вызваны контроллером, подменяем за счет **@MockBean**
  + В итоге смотрим, что вернул клиент и что получили заглушки, вызваные контроллером
  + В данном случае мы только имитируем работу сервлетного контейнера, а на самом деле просто передаем запросы напрямую экземпляру контроллера
* **@DataJpaTest** – тестирование работы с данными
  + В тесте используем специальный **@TestEntityManager,** который для своей работы не нуждается в реальном хранилище данных
* **@RestClientTest** – тестирование обращения к внешним сервисам, то есть себя в качестве клиента
  + Дополнительно на тесте **@AutoConfigureWebClient(registerRestTemplate=true)**
  + Возможно, если мы обращаемся к внешним сервисам только через **@RestTemplate**
  + За счет **@Autowired MockRestServiceServer** имитируем поведение внешнего сервиса, получившего наш запроса.
  + Затем вызываем метод нашего сервиса, который обращается к внешнему, и проверяем чего получилось

### Тестирование контрактов

* Consumer-Driven Constact Testing (CDC-T)
* Основная идея: поставщик микросервиса публикует заглушку (stub), которую его клиенты могут использовать в своих тестах.
  + Заглушка описывает контракт микросервиа и имитирует его поведение в ограниченном scope.
  + Вместе с заглушкой публикуется комплексный тест, работающий с заглушкой и иллюстрирующий использование сервиса реальным клиентом
  + Реальная иллюстрация сервиса и его заглушка на должны исползовать общего кода в виде типов или шаренных библиотек
* **Spring Cloud Contract** – возможность публикации, моделирования и имитации удаленных сервисов с помощью заглушек
  + Поведение сервиса эмулируется DSL на основе groovy скрипта. Логика: если клиентский запрос выглядит так, то высылаем такие-то данные в ответ. Используются регулярные выражения для анализа запроса и генерации данных в ответ.
  + За счет плагина **spring-cloud-contract-maven-plugin** вместе с собственно реализацией меикросервиса публикуется второй артефакт типа **stubs**
  + Клиент может в своем тесте за счет **@AutoConfigureStubRunner** запустить эмуляцию удаленого сервиса из доступного DSL описания, к которому будут обращаться его собственные сервисы

### Тестирование целостности данных

* Одна из основных проблем при изолированном тестировании сервисов – проверка полноты / целостности передаваемых и возвращаемых данных. Возможные ошибки
  + Используем опциональные фрагменты аргументов без проверки – получаем NPE
  + Возвращаем неполные данные – NPE или ошибку валидации получит клиент
  + Переливаем данные в базу / из базы не полностью – данные теряются по дороге
  + Клиент может сформировать данные, которые просто не вместятся в базу
* Все усугубляется тем, что тщательно готовить разнообразные данные для компонентных тестов всем обычно лень. Поэтому передают наиболее очевидные значения, а исключенияиз-за ошибок обработки экзотических вариантов ловят уже пользователи по мере разнообразия вводимых данных
* Чтобы избежать:
  + Аргументы для вызова тестируемого метода готовим при помощи какого-нибудь рандомного генератора
    - Случайные данные должны генерироваться исходя из типов полей и аннотаций на полях. Как минимум необходимо учитывать **JSR 380 (Bean Validation 2.0)** аннотации, например - **@NotNull, @NotEmpty, @Size**, **@Pattern**.
      * Можно добавить собственные
      * Необходима возможность добавления кастомных стратегий генерации рандомных значений для специфических классов
    - Реализуем два режима формирования рандомных данных: бин заполняется полностью или только обязательные поля
  + Каждый тест запускаем два раза с аргументами минимально / максимально заполненными случайными данными
  + Применяем валидацию бинов
    - Для аргументов вызова тестируемого метода / результата возвращаемого метода / аргументов, передавамые mock-ам вызываемых сервисов
    - Особенно удобно это реализовывать, если дергаем сервисы (тестируемые и моск) не непосредственно, а через декораторы (например, на основе RxJava), позволяющие легко встроить опциональную кросс функциональность
* За счет этого для сервисов сразу видим:
  + NPE, если внутри метода обратились к отсутствующим данным
  + Неправильно сформированные данные для вызова внешних сервисов
  + Неправильно сформированные результат вызова сервиса
* На уровне БД сразу видим
  + Если данные не вмещаются в колонки
  + Если обязательные на уровне БД данные в нее в итоге так и не передаются
* Остается проблема с контролем передачи опциональных данных:
  + Чтобы их поймать сравниваем результаты по обоим сторонам цепочки: **DTO1 -> create(dto) -> JPA -> Database -> JPA -> query(query) -> DTO2**
  + Сравнивать DTO1 и DTO2 можно как ручками, так и при помощи средств типа **XmlUnit**

## Системные тесты

### Mocking

* Если все сервисы обращаются друг к другу через Gateway, то он становится идеальным местом для выполнения моск операций, необходимых для:
  + Эмуляции работы еще не разработанных сервисов при параллельной разработке. Особенно при разработке UI
  + Индивидуальное интеграционное тестирование отдельных сервисов или группы сервисов с эмуляцией остальных
* Что требуется (фактически это функционал Mockito, только на уровне HTTP вызовов):
  + Возможность маппить запросы на готовые ответы. Если соответствие найдено – возвращаем моск ответ, если нет – пропускаем к оригинальному сервису
  + Мапинг может быть как статическим, разрабатываемым и поставляемым вместе с сервисом, так и динамическим: тесты при выполнении могут включать / выключать различные готовые маппинги по тегу или регистрировать свои собственные, реагирующие на только что сгенерированные ими данные
  + По окончанию тест может проверить, были ли реально вызваны, сконфигурированные им моск-и и сколько раз
* Весь лог запросов / ответов реальных и моск хорошо при этом записывать.
  + Доступ к ним предоставляется через UI, а также через API, чтобы сбойный тест мог сохранить всю историю своего падения
  + Логи реальных запросов / ответов это хороший исходный материал для формирования собственных mock
* На основе фильтров Zuul такой функционал легко реализуется самостоятельно, но можно прикрутить что-то типа WireMock.

# Документирование

* Сообразно Agile манифесту – сначала добиваемся работоспособности, затем уже документируем. Ну и лучше всего, когда реализация приложения сама себя документирует
* **OPTIONS** позволяет получить информацию о том, какие стандартные HTTP операции можно совершить над ресурсом. MIME тип сообщает что-то о формате представления ресурса. Увы нет способа непосредственно в процессе реализации описать точную бизнес семантику операции, а также поддерживаемый набор параметров / заголовков

## Swagger

* Наиболее стандартное средство для работы со спецификациями в формате OpenAPI.
* Предоставляет несколько утилит
  + **Swagger Editor –** веб редактор для редактирования OpenAPI спецификаций
  + **Swagger UI** – предоставление OpenAPI спецификации в виде интерактивной документации
  + **Swagger Codegen** – генерация серверного и клиентского кода на основе спецификации
* Два направления разработки: либо создаем спецификацию в виде YAML файла и генерируем код проекта из нее, либо аннотируем имеющийся код контроллеров и получаем документацию на выходе
  + Для простых описаний нормально, для сложных - объем необходимых дополнительных аннотаций, использующих текстовые данные, становится слишком значителен
  + В обоих случаях сложно запустить в проект документатора НЕ-программиста
* Инициализация
  + **@EnableSwagger2** на конфигурации
  + Бин **Docket**, посредством которого определяем системные свойства. Например: какие пакеты исследовать на предмет аннотированного кода, путь для публикации интерактивной документации и так далее
  + Бин **ApiInfo** – инфоримация о приложении, контакты разработчиков, общее описание, лицензионная информация и так далее
* На методах контроллеров аннотации
  + **@ApiOperation( description, response )**
  + **@ApiResponses( { @ApiReponse( code, message ) } )**

## Spring RESTDocs

* В процессе выполнения тестов URL и тела запросов / ответов выгружаются на файловую систему. После чего мы подтягиваем их в рукописную документацию на основе Asciidoctor
* Зависимость **spring-restdocs-mockmvc**
* Обычный тест **@SpringBootTest(webEnvironment=MOCK)**
  + Добавляем **@AutoConfigureRestDocs(outputDir = “target/generated-snippets”) –** куда выгружать автогенерируемые фрагменты
  + **@Rule RestDocumentation**
  + В отдельных тестах **mockMvc.peform().addDo(restDocumentation.document(“snippetId”))**
* В рукописной документации ссылаемся на сгенерированные фрагменты как **include::{snippets}/ snippetId/http-request.adoc[]**
* Документацию собираем за счет **asciidoctor-maven-plugin** и затем за счет **maven-resources-plugin** копируем в **src/main/resources/static**
  + Таким образом мы ее делаем частью самого приложения
  + Сгенерированные фрагменты коммитим, что дает нам контроль над изменениями форматов принимаемых / передаваемых данных

# Приложение 1 – REST

## Основные принципы

* **Representational State Transfer** – архитектурный стиль реализации распределенных приложений, основанный на передаче представлений состояний ресурсов. Альтернатива таким технологиям как RPC и SOAP
* Представляет собой **best practices** реализации веб-сервисов, основанных на HTTP. Причем НЕ только как на транспортном протоколе, но и как на протоколе уровня приложений, включающем в себя следующие основные моменты:
  + **HTTP methods** – стандартные глаголы / действия / операции над ресурсами с заранее определенной семантикой: **GET / PUT / PATCH / HEAD / POST / DELETE / OPTIONS**. Данная семантика позволяет реализовать дополнительную функциональность прозрачно для приложения: кэширование, балансирование, обработку ошибок, безопасность и так далее.
  + **URI (Uniform Resource Identifiers)** – уникальные идентификаторы ресурсов.
    - В процессе REST запросов / ответов приложения обмениваются не ресурсами, а предславлениями ресурсов:
      * В различных форматах
      * Обогащенными различными метаданными, необходимыми приложениями, чтобы реализовать обработку информации (HATEOAS)
      * Различной степени гранулярности / подробности в зависимости от реализуемого use case – projection
    - Универсальные в частности означает, что они НЕ зависят от реализации.
      * То есть использовать ID записи базы данных в данном случае не очень хорошо – в случае пересоздания базы он может поменяться.
      * Ну или надо его делать неизменяемым, фактически бизнес идентификатором.
  + **HTTP Responces** – стандартные код ответа и наборы стандарных заголовков, которые должны присутствовать в ответе
* **RESTful архитектура – stateless client / server архитектура**
  + Системы слабо связаны между собой.
  + Клиент не всегда имеет прямое соединение сервером – кэширование и другие промежуточные слои между ними, так что сервер какое-то время может быть вообще не доступен
  + Промежуточные системы могут реализовывать такие концепции как кэширование, балансирование, обработку ошибок, безопасность, мониторинг и так далее
  + Представления ресурсов (тело / заголовки) могут содержать дополнительную мета-информацию
    - Связывающую запрошенные ресурсы с другими
    - Определяющую способ выполнения бизнес-операций над ресурсом в текущем состоянии
  + Так как stateless, то запрос содержит в себе всю информацию, необходимую для его выполнения.
    - С одной стороны, scalability и fail safe, значительное снижение затрачиваемых сервером ресурсов
    - С другой стороны проблемы при реализации, к примеру, безопасности
* REST это не протокол, не стандарт / набор стандартов (диссертация за стандарт не канает), а скорее архитектурный стиль с набором ограничений поверх HTTP протокола:
  + приложение разбито на отдельные ресурсы, каждый из них имеет уникальный ID и адресуем (может быть получен посредством фиксированной ссылки)
    - подразумевается, что ссылку можно заложить, переслать другому пользователю и так далее – это постоянный адрес ресурса в общем информационном пространстве
  + операции над каждым ресурсом описываются с помощью глаголов (методов) HTTP и кодов возврата.
    - Все операции реализуются за счет HTTP методов с общеизвестной семантикой
    - Мы работаем не поверх протокола, а за счет протокола, расширяя его дополнительными соглашениями.
  + Ресурсы могут быть представлены в различных вариантах
    - По ссылке может отдаваться как специфичное для приложения JSON представление, так и его вид в стандартном формате (увы, не для всего есть стандартные форматы) или даже готовый отчет для печати.
  + путь к ресурсу содержит только наименование ресурса и идентификатор, все остальное в параметры запросов. Сохраняем вид пути как можно более простым.
    - Возможна цепочка из нескольких групп наименований / идентификатор - обращение к субресурсу
    - Обычно суб-ресурс трактуется как дочерний, по отношению к родителю. Например, элемент заказа - /orders/15/items/6
  + Сервер stateless, вся информация, необходимая для выполнения запроса, содержится в самом запросе и в ресурсе, которому адресован запрос
* Достоинства REST подхода
  + Простота
  + Широкая поддержка всеми языками и фреймворками
  + Ресурсы могут быть представлены во множестве стандартных и кастомных форматов
  + Для достижения высокой производительности / безопасности / отказоустойчивости можно применять стороннее ПО, выполняющее кэширование / проксирование / ограничение доступа / балансировку
  + Уменьшает связывание клиентов и серверов между собой
  + Браузеры, JS based веб и мобильные приложения могут выполнять репрезентацию полученных данных
  + Новые клиенты легко могут подключаться, даже если изначально приложение не было рассчитано на их поддержку
  + З асчет stateless и балансировки легко достижима высокая производительность и отказоустойчивость, прозрачно для основного приложения
  + Минимум или полное отсутствие документации за счет унификации API
  + Не нужны специальные сложные библиотеки, как в случае, например, SOAP или CORBA

## Коды возврата

### Успешное выполнение запрошенной операции

* **200 OK** – успешный **GET / PUT / PATCH / DELETE**
  + может (для **GET** обязан) содержать представление ресурса
  + может быть использован для **POST**, который ничего при этом не создает
* **201 Created** – в ответ на **POST** с **обязательным** Location заголовком
* **204 Not Content** – успешный статус **без** данных, например после **DELETE / PUT**
* В ответах 2хх с не-пустым телом обязательно наличие заголовка Content-Type: application/json;charset=URF-8

### Перенаправление

* **301 Moved Permanently** – например при версионировании для старой реализации
* **302 Moved Temporarily** – перенаправление на другой ресурс в рамках бизнес процесса
* **303 See Other** – предложение другого способа доступа
* **304 Not Modified** – в ответ на условный **GET**

### Клиентская проблема

* **400 Bad Request** – запрос не понятен серверу в контексте предложенного use case
* **401 Unauthorized** – требуется авторизация
* **403 Forbidden** – нет доступа, доступ не авторизован
* **404 Not Found** – ресурс не найден. Не стоит возвращать данный ответ, если пользователь запросил список с фильтрацией и ни одного ресурса найдено не было – надо вернуть пустой список.
* **405 Method Not Allowed** – применен неправильные HTTP метод
* **409 Conflict** – конфликт, например
  + для конкурентного доступа, когда PUT / POST относятся к предыдущему состоянию ресурса (оптимистическая блокировка на основе ETag)
  + попытка создать ресурс, который для приложения уникален и УЖЕ существует
* **410 Gone** – ресурс больше не доступен, например, чтобы пометить выведенный из употребления метод API
* **414 URI Too Long** – вам пора перейти на использование заголовков
* **415 Unsopported Media Type** – данный формат данных не поддерживается
  + обычно при загрузке файлов на сервер, которые не могут быть обработаны.
  + Также можно возвращать, если не удалось разобрать JSON запрос или сам запрос пришел не в формате JSON
* **418 I’m a teapot** – запрос вообще не удалось разобрать, а вам вообще сюда??? Можно использовать, чтобы
  + указать на несовпадение версий протокола / API клиента и сервера
  + Также чтобы различить реакцию на запросы на не-поддерживаемые URL от запросов не-существующих объектов. Альтернатива это 410, но все-таки это не одно и то же, так как подразумевает, что ресурс тут когда-то существовал
  + В отличии от 404 данный код никогда не бросается никаким промежуточным софтом
* **419 Authentication Timeout** – протухли токен доступа или CSRF, а потому клиент должен повторно авторизоваться. Используется, чтобы предотвратить потерю несохраненных данных клиента, если просто перекинуть его на страницу авторизации
* **422 Unprocessable Entity** – используется для ошибок валидации данных (данные правильно закодированы, корректно распознаны, но с точки зрения принимающего сервиса не валидны). Единственный вариант, когда тело – описание ошибки лучше все-таки выдавать в виде JSON
* **429 Too Many Requests** – слишком большая нагрузка на сервер со стороны ДАННОГО клиента

### Серверная проблема

* **500 Internal Server Error** – что-то совершенно НЕПРЕДВИДЕННОЕ и единичное случилось на сервере. Наличие такой ошибки это признак бага, повод для алерта в системе мониторинга и так далее.
* **501 Not Implemented** – текущий метод к ресурсу не применим или не реализован
* **503 Service Unavailable** – сервер перегружен, зайдите позже. Обязателен заголовок Retry-After
* **504 Gateway Timeout** – тайм-аут при вызове сервером третьей системы
* **502 Bad Gateway** – ошибка при вызове сервером третьей системы

## HTTP методы - стандартные операции над ресурсами

* Все операции над ресурсами, как CRUD, так и любые другие, так или иначе маппятся на основные HTTP методы.
* Если какая-то операция все-таки не маппится. Например, «подсчитать цену общую сумму заказа и передать в дальнейшею обработку». Варианты реализации:
  + Маппим на атрибут сущности, например active, и меняем ее значение через **PATCH** запрос. Просто и прозрачно, но
    - подходит только для самых простых автомарных операций, типа переключения статуса.
    - Не пременимо, когда операция инициирует процесс с сообственным состоянием и сопутствующей дополнительной информацией
  + Маппим аттрибут на квази-суб-ресурсы (/objects/12/activate) и управляем за счет **PUT / DELETE.**
    - Слегка гибче, но в общем то же самое.
    - За счет **GET** на таком ресурсе можно вернуть либо форму с параметрами для ручного вызова либо шаблон команды для заполнения
  + Маппим статус на квази-суб-ресурс (/objects/12/prepared, (/objects/12/passed и так далее)
    - Соответственно, команда на переход ресурса в новое состояние это POST с аргументами. Если такой переход для ресурса не возможен, то возвращается 405
    - **GET** может в частности возвращать информацию о том, кто / когда / с какими аргументами запросил переход и что из этого вышло
    - Если переходы в ЖЦ ресурса не зацикливаются, то есть в каждом состоянии он может побывать только один раз, то вполне нормальная схема
  + Создаем субресурс коллекцию объектов типа «команда» («проводка», «транзакция»). Наиболее общая и гибкая схема.
    - Соответственно, вызов метода ресурса это **POST** в коллекцию документа с аргументами
    - Коллекция является одновременно и логом запрошенных операций и журналом транзакций, которых можно проиграть для воспроизведения состояния ресурса
    - Состояние каждой запрошенной операции является отдельным ресурсом, что позволяет ее гибко контролировать, получать / предоставлять дополнительную информацию / отменять
    - Одновременно ресурс может участвовать в нескольких бизнес процессах

### POST - создание ресурса (или добавление ресурса в коллекцию)

* НЕ- идемпотентный и НЕ-безопасный, так как меняет состояние ресурса
* Представление создаваемого ресурса передается в теле запроса с помощью XML / JSON / YAML / ??? документа, медиа-тип документа в HTTP заголовке Content-Type
* При успешной обработке возврщаются:
  + Обычно ответ пустой, так как у клиента и так уже есть все необходимые данные. Но может возвращаться представление созданного ресурса, возможно обогащенного дополнительной информацией.
  + Основной код ответа **210 (Created)** и **Location** заголовок с URI созданного ресурса, который будет использоваться для дальнейших операций над ним – **GET / PUT / DELETE.** Данный заголовок используется, чтобы выкусить из него и сохранить сгенерированный сервером идентификатор нового ресурса
  + Можно вернуть
    - **301 (Mooved Permanentl) +Location** – клиент автоматически получит представление вновь созданного ресурса за счет клиентского редиректа
    - **204 No Content**, подразумевается, что состояние созданного ресурса в точности такое же, как и было передано. Особенно если размер документа большой
  + Важно, что URI вновь созданного ресурса в данном случае определяет сервис
* При ошибке возвращаются:
  + **400 Bad Request** – когда переданое представление неверно.
    - В этом случае повторять запрос бессмысленно, проблема в нем самом.
    - Может вернуть полученное представление обогащенное информацией, объясняющей почему запрос неверен, например, перечнем обязательных, но отсутствующих полей
    - Можут также вернуть ссылку на документацию (в обогащенном представлении, в заголовке, просто в теле вместо представления)
  + **500 Internal Server Error** – когда сервер не может по внутренним причинам обработать запрос. В этом случае через некоторое время можно попробовать опять
  + Сервер, выдавший **500** или **504** , должен восстановиться после ошибочной попытки обработки и быть готовым получить повторный запрос с теми же данные – идемпотентность.
* В связи с этим имеется проблема – в случае ненадежного взаимодействия и повторения запроса будут созданы два объекта. Возможные решения:
  + Ограничение целостности в БД на бизнес полях – не всегда возможно
  + Представление ресурса, включающее идентификатор, генерируемый при помощи UUID – лучше
    - создает путаницу, даже если ресурс будет создан с соответствующим URL
    - то же самое, но в HTTP заголовке – наверное оптимальное
  + Создавать ресурс за счет **PUT** по URL, включающему UUID идентификатор – в общем-то то же самое, но слегка противоречит идеологи. В общем тоже нормально.
* Так же, по причине отсутствия ограничений на семантику, **POST** используется для реализации всех остальных бизнес-операций, которые плохо укладываются в CRUD модель

### GET - получение представления ресурса

* все данные передаются в URI – важно помнить, что размер ограничен и разные сервера / промежуточные прокси / кэши могут иметь разное представление о максимально возможном размере
* Идемпотентный и не меняет состояние ресурса
* Так как браузеры и промежуточные средства активно кэшируют GET & HEAD запросы, то обязательно указываем политику кэширования. Например, чтобы запретить: **Cache-Control: no-store, no-cache, must-revalidate + Pragma: no-cache**
* В случае ошибки:
  + **500 / 504** в случае невозможности обработать запрос
* Два варианта:
  + /{entity}/:id – запрос конкретного документа
    - В случае успеха код **200** и представление ресурса, иначе **404**
  + /{entity}?param=value&… - запрос коллекции с фильтрацией, сортировкой, разбиением на страницы и так далее
    - В любом случае возвращает **200** + коллекцию с результатами, пустую или нет. Можно более сложный конверт вокруг с номером текущей страницы, данными о фильтрации и так далее

### HEAD – получение мета-информации

* Полный аналог GET с такими же URL, но возвращает только заголовки
* Часто используется браузерами в качестве автоматических pre-flight запросов перед выполнением потенциально опасных операций. Например, для получения политики CORS при кросс-доменных операциях. Ошибка обработки или игнорирование такого запроса приведет к тому, что основной запрос вообще не будет выполнен браузером или другим клиентом
* Может использоваться
  + для проверки существования файла без его передачи
  + для получения мета-информации о ресурсе, специфической для приложения

### PUT - обновление состояния ресурса

* Идентификатор ресурса берется из URI, передавать его еще и в составе документа не рекомендуется.
* Может использоваться также
  + для создания нового с помощью URI, определяемого клиентом
  + для обновления содержания коллекции целиком
* **Идемпотентный**, но меняет состояние ресурса **(не safe)**.
  + Можно применять несколько раз, но только в том случае, если это честный **PUT**, то есть состояние передается полностью.
  + Если хочется только подправить фрагмент, то можно
    - либо использовать **PATCH** метод, спецификация указывает, что он в общем случае не является идемпотентным
    - либо делать части ресурса самостоятельными ресурсами и задействовать уже их
* В случае успеха
  + Как правило, **204** с пустым телом, так как у клиента и так уже есть все необходимые данные, включая идентификатор ресурса
  + **200** с новым представлением (особенно если были автоматически обновлены какие-то метаданные)
* В случае ошибки
  + **404 / 500 / 504**
  + **409 Conflict** – когда не понятно как совместить текущее и переданное представления клиента. Например, заказ уже передан в обработку и не может быть изменен.
  + **405 Method Not Allowed** 
    - казалось бы даже лучше подходит к ситуации **409**, но это невозможность применить операцию к ресурсу «в принципе», а не в данном его состоянии
    - может вернуть заголовок **Allow** со списком наименований доступных для ресурса методов, например **GET**
* При конкурентном обновлении ресурс НЕ может прийти в неконсистенное состояние, но одно из обновлений может потеряться и соответствующий клиент об это не узнает

### DELETE - удаление ресурса

* Можно как конкретного ресурса, так и на коллекции с перечнем параметров для фильтрации предназначенных для удаления элементов. Без параметров это по идее будет удаление всей коллекции - довольно опасный вариант.
* **Идемпотентный**, но меняет состояние ресурса **(не safe)**
  + Вообще-то запрашивается удаление именно представления ресурса, что при этом происходит с самим ресурсом ответственность сервиса. Совершенно не обязательно, что ресурс физически удаляется.
  + Это не означает, что при повторных вызовах он должен выдавать опять **2хх**, лучше **404**. Впрочем, в целях безопасности можно и всегда **2хх**
* В случае успеха - обычно **204** с пустым телом, так как возвращать уже нечего.
* В случае ошибки - **404 / 500 / 504 / 409 /405**

### PATCH - исправление ресурса

* Только с URL формата **/{entity}/:id** так как частично обновляем определенный ресурс
* НЕ safe и НЕ idempotent (хотя часто считают, что idempotent, но пусть идут читают спецификацию)
* Позволяет передать только те фрагменты представления, которые изменились
* Часто служит основой для реализации операций, не укладывающихся в CRUD парадигму
* Обязательно учитывать возможность конкурентного обновления, в результате которого ресурс может приобрести неконсистентное состояние
* Возвращаем **200** + обязательное новое представление ресурса с учетом примененного обновления

## Spring support for REST

* Java EE предлагает собственный рамочный фреймворк для реализации RESTful приложений – JAX-RS, последняя версия 2.0. Имеются различные имплементации – Jersei (RI), RESTEasy, Restlet, CXF
* Spring НЕ реализует JAX-RS стандарт, но начиная с версии 4.0 предлагает различные средства поддержки
  + Для серверной стороны – на основе ранее введенной Spring MVC архитектуры, прежде всего за счет **@RestController и @RequestMapping** аннотаций
  + Для клиентской стороны – прежде всего **RestTemplate**

### Контроллеры

* Как и прежде для Spring MVC приложений необходим декларативно или программно сгенерированный **DispatcherServlet** с дополнительными механизмами
* Вместо **@Controller**, результат которых должен использоваться для поиска и вызова подходящего View, используются **@RestController**
  + Соответственно результат их работы должен быть возвращен пользователю «как есть», без обработки каким-либо представлением.
  + Фактически это сочетание старого **@Controller + @ResponseBody**, который сообщает контейнеру, что результат должен быть отправлен клиенту как есть, без преобразования за счет **View**
  + Это не только определение компонента НО и определение дополнительной функциональности для компонента. Поэтому, данная аннотация НЕ относится к числу стереотипов и вообще определена в другом пакете
* **@RequestMapping** по прежнему является основой маппинг URL запроса на метод контроллера, но теперь в нем активно задействован атрибут **method = RequestMethod.\***
  + является нормой что один URL (адрес ресурса) должен поддерживать несколько операций различных типов (GET/PUT/POST/DELETE) и обслуживать соответственно несколькими методами. В стандартных веб-приложениях как правило используется толькоGET
  + идентификатор ресурса по правилам содержится в URL, поэтому
    - путь запроса содержит шаблон для поиска значения - **value = “/user/${userName}”**
    - аргумент с тем же именем или **@PathVariable(“userName”)**
  + Начиная со Spring 4.3 вводятся упрощающие аннотации - **@GetMapping / @PostMapping / @DeleteMapping / @PutMapping**
    - Не нужно доопределение за счет атрибута method
* **Методы контроллера**
  + Могут применяться все те же аргументы, что и в случае веб-приложений
  + Основной аргумент **- @RequestBody** на аргументе БИЗНЕС-типа.
    - Преобразование данных из XML / JSON в объектное представление производится за кадром
    - Если **@Valid,** то выполняется проверка данных бинов за счет Validation API
  + Также можно определять дополнительные параметры с уже знакомыми аннотациям: **@RequestHeader, @RequestParam** и так далее
  + Возвращает
    - **Void** – если в ответе данных не предусмотрено, например, в случае **POST** запроса и **210-Created**  ответа
    - Объект бизнес-типа, преобразование данных в XML / JSON производится опять-таки за кадром, так как метод скрыто уже аннотирован **@ResponseBody**
  + Возврат дополнительной информации
    - На методе аннотация **@ResponseStatus(HttpStatus.\*)** – декларативное определение стандартного кода возврата, , за счет **reason** аттрибута можно передать дополнительную информацию
    - Параметр типа **HttpServletResponse** – программное определение кода возврата, добавление к ответу произвольных заголовков, например, **Location**
    - Можно вернуть **ResponseEntity**, в котором совместить все возвращаемые данные - представление ресурса + заголовки + статус
  + Если надо вернуть в состве ответа (как правило в виде HTTP заголовка) URL на другой ресурс, то для его создания используется **UriTemplate**
    - Создается как new на основе шаблона, включающем **${someName}** фрагменты
    - затем метод expand(), которому передаются значения фрагментов, которые будут правильным образом преобразованы во фрагменты URL
    - Правильно учитывает текущие application context, port, заголовки проксирования типа **X-Forwarded-Host, X-Forwarded-Port, X-Forwarded-Proto**
* **Обработка ошибок**
  + Проблемы:
    - как превратить исключение в HTTP статус
    - как вернуть вместе с этим статусом соответствующий ответ = тело + заголовки
    - как правильно заакодировать формат тела в соответствии с меди-типом запроса
  + Возбудив исключение, которое наследуется от **RuntimeException** и само в свою очередь аннотировано при помощи **@ResponseStatus**
    - Отвечает за это **ResponseStatusExceptionResolver**, зарегистрированый в **DispatcherServlet**
    - Что делать с уже имеющимися контроллерами, код которых вне нашего контроля и которые возбуждают свои собственные исключения
    - Ничего кроме статуса вернуть не получится, остальное вне нашего доступа
  + При помощи **@ExceptionHandler({перечень обрабатываемых исключений})** на методах-обработчиках контроллера
    - которые могут
      * ничего не делать и только возвращать статус за счет **@ResponseStatus**
      * выполнить редирект на представление, обрабатывающее ошибку
      * вернуть представление ошибки в виде **@ResponseBody**
      * полностью сформировать правильный HTTP ответ в виде **HttpEntity<?> or ResponseEntity<?>**
    - принимают правктически все те же параметры, что и методы обработчики запросов, плюс **Exception** и соответственно его наследников
    - Основные проблема:
      * на каждом контроллере надо определять заново или иметь общего предка для всех котроллеров, что не всегда возможно.
      * Плюс те же самые проблемы, что и в предыдущем случае: ничего больше не возвращается и непонятно что делать с уже имеющимися контроллерами
  + Реализоватьинтерцептор контроллера
    - Класс, аннотированный **@ControllerAdvice**
      * За счет атрибутов определяем к каким контроллерами применяем
        + **basePackages** – ко всем в определенных пакетах
        + **basePackageClasses** – к определенному классу контроллера и наследникам
      * Можно расширить **ReponseEntityExceptionHandler,** который выполняет обработку стандартных исключений, например **HttpMessageNotReadableException** .
    - На нем метод-перехватчик с собственными исключениями
      * Аннотированный с помощью **@ExceptionHandler({перечень обрабатываемых исключений})**
      * Принимающий любые необходимые параметры и возвращающий **ResponseEntity** с которым мы имеем полный контроль над результатом
* **Конвертация HTTP сообщений**
  + Проблема
    - как определить путь, которым **@ResonseBody / @RequestBody** преобразуют XML / JSON документ из запроса в объектное представление и обратно
    - как замапить метод-обработчик на соответствующий запроса
  + Определяется для **@RequestMapping** аннотаций за счет атрибутов **produces / consumes**
    - Отбор подходящих запросов выполняется на основе HTTP заголовков
      * **Content-Type** – в каком формате выслали данные
      * **Accept** – в каком формате ожидаем ответ
    - Стандартные форматы (меди-типы) – **MediaType.\***
  + Конвертация данных выполнеяется за счет реализаций **HttpMessageConverter<T>**
    - реализации данного интерфейса автоматически ищутся и регистрируются за счет **<mvc:annotation-driven> или @EnableWebMvc**
    - стандартно выполняется классами типа **XxxHttpMessageConverter**, где Xxx может быть:
      * **ByteArray, String, Resource**
      * **Form** - в MultyValueMap<String, String>
      * **Source** -в javax.xml.transform.Source
      * Если соответствующие библиотеки подключены
        + Jaxb2RootElement
        + MappingJackson2 и MappingJackson
        + AtomFeed, RssChannel
  + Чтобы создать свой конвертер:
    - Реализовать интерфейс **GenericHttpMessageConverter<T>** с методами
      * **boolean canRead(Type, Class<?>, MediaType)**
      * **boolean canWrite(Type, Class<?>, MediaType)**
      * **T read(Type, Class<?>, HttpInputMessage)**
      * **void write(T , Type, MediaType, HttpOutputMessage)**
    - подключить
      * наследовать конфигурцию от **WebMvcConfigurerAdapter** и переопределив метод **configureMessageConverters(List< HttpMessageConverter>)** подредактировать стандартный перечень.
      * Также можно опубликовать свою реализацию **RequestMappingHandlerAdapter**, набив ее необходимыми конвертерами
      * Также надо подключить его к используемому клиенту за счет **RestTemplate.setMessageConverters().** Для тестов это делается автоматически

### Конфигурирование приложения

* **@EnableMvc + @ComponentScan** на классе, расширяющем **WebMvcCOnfigurerAdapter**
* За счет переопределения методов:
  + **addResourceHandlers()** – определяем где находятся статические ресурсы
  + **addViewControllers()** – регистрируем логическое наименование единственной веб-страницы приложения, которую вешаем на «/»
    - За счет бина **@Bean InternalResourceViewResolver** определяем префикс и постфикс для резолвинга этой страницы по логическому наименованию
  + **configureDefaultServletHandling()** – вешаем стандартный **DispatcherServlet**
  + **configureMessageConverters(List<HttpMessageConverters>)** – регистрируем JSON/ XML маппинг
    - вызываем родителя!!!
    - Создаем **MappingJackson2HttpMessageConverter** как new и добавляем в список
      * Определяем опции конвертации **(prettyPrint, defaultCharSet)**
      * Создаем и устанавливаем экземпяр **ObjectMapper**
        + Конфигурируем ему **SerializationFeature** и другие детали процесса

### Использование RestTemplate в качестве клиента к REST

* Thread-safe, можно иметь только один экземпляр в системе
* HTTP methods can be combined to together using RestTemplate
* В случае проблем обертывает исключения транспортного уровня в **RestClientException**
* По умолчанию использует стандартный HTTP клиент из java.net, предоставляемый JDK
* Много методов для различных методов HTTP и доступных аргументов, все они работают через два семейства метов
  + **public <T> T execute(url, method, requestCallback, responseExtractor, uriVariables)**
  + **<T> ResponseEntity<T> exchange(url, method, requestEntity, responseType, uriVariables)** и другие сочетания типов аргументов
* Обязательные параметры всех методов
  + первый параметр – шаблон URI, например **«…/hotels/{hotel}/bookings/{booking}»,** или **java.net.URI**
  + аргументы шаблона - либо **String** …, тогда подставляются по очереди, либо **Map<String,String>,** тогда по имени. Последнее удобне, когда параметров много или они повторяются
  + класс объекта, если рузультат надо сразу же конвертировать в объектное представление
* Для поддержки маппинга из/в объектное представление, необходимо зарегистрировать конвертеры, это можно сделать за счет **RestTemplate.getMessageConverters()**
* **Основные СИНХРОННЫЕ операции**
  + **GET**
    - **ResponseEntity** **getForEntity(url, args).**
      * Затем **entity.getStatusCode(), entity.getBody(),** а также все остальные HTTP подробности
    - **Object getForObject(url, args, class) –** получение только объектного представления результата
  + **HEAD**
    - **HttpHeaders headForHeaders(url, args)**
  + **POST**
    - Обязательный аргумент – объект, данные которого надо конвертировать в тело запроса, или уже подготовленный **RequestEntity**, тогда возможна работа с дополнительными параметрами
    - Варианты
      * **postForLocation()** - возвращает URL только что созданного ресурса из **Location** заголовка
      * **postForObject()** – возвращает объект
      * **postForEntity()** – возвращает ResponseEntity
  + **PUT**
    - void **put(…)** – ничего не возвращает
    - Аналогично можно отправить объект или **RequestEntity**
  + **OPTIONS**
    - **Set<HttpMethod> optionsForAllow()**
  + **DELETE**
    - **void delete(…)**
    - так как ничего не возвращает, то единственный способ узнать результат – запросить удаленный ресурс и проверить получение 404 ответа
* **Общее API**
  + Синхронная работа
    - **ResponseEntity exchange(url, method, request | object, class)**
  + Асинхронная работа
    - создаем **RequestCallback** с методом **void doWithRequest(ClientHttpRequest)**
    - создаем **ResponseExtractor** с методом <**T> extractData(ClientHttpResponse)**
    - вызываем **template.execute(url, method, callback, extractor).**
      * Если результат не важен, то последний аргумент может быть null
* **AsyncRestTemplate** – вариант для поддержки асинхронных операций с возможностями Java8. Возвращает результаты в виде типизированных **ListenableFuture<T>**

### Тестирование

* За счет конфигурирование Mock веб-сервера и использования для работы контроллеров mock зависимостей, поставляющих сервисы
  + Задействуется **DispatchServlet** для предоставления всей Spring MVC функциональности, но не запускается сервлетный контейнер.
  + Поэтому часть функциональности может просто не работать
    - Нету - contextPath по умолчанию, jsessionid cookie, JSP страниц (шаблонные механизмы типа freemaker продолжают работать)
    - Не работают асинхронные вызовы
    - Также нету классических HTTP механизмов: фильтров / слушателей, развертываемых через **web.xml**
    - Не работают **forwarding, redirection**. Придется их ручками вынимать как URL из ответа и проверять на соответствие
  + К счастью
    - практически все, относящееся к REST работать продолжает – рендеринг JSON / XML, **@ResponseBody** и так далее
    - Можно подсунуть свои релизации системных механизмов, например **HandlerExceptionResolver** и затем заглянуть внутрь, проверив как они сработали для того или иного запроса
* Конфигурирование
  + **@RunWith(SpringJUnit4ClassRunner) + @ContextConfiguration + @WebAppConfiguration**
  + За счет фрагмента тестовой конфигурации подсовываем **@Bean**, публикующие mock сервисы
  + В **@Setup** методе
    - Создаем **RestTemplate** как new
    - Создаем **MockRestServiceServer.bindTo(restTemplate)….build()**
* Реализация теста
  + за счет **server.expect(requestTo(“…”)).andResponse()** определяем ожидаемую последовательность запросов / ответов
  + Запускаем клиента, проверяем результаты его работы
  + Проверяем за счет mock сервисов какие данные реально были получены ими
  + За счет **server.verify()** проверяем, что запросы имели место быть

## Best Practices & Use Cases

* Стандарт HTTP (а REST работает поверх стандарта HTTP) это стандарт, который определяет дефолтное поведение всех агентов, которые могут встретиться между сервером и клиентом – балансиры / прокси / кэширующие прокси / средства сетевой защиты / средства мониторинга / браузеры и мобильные платформы, используемые разработчиками, средства CI / тестирования / клиентские библиотеки и так далее. Несоблюдение может приводить к неочевидным проблемам, поэтому любые велосипеды могут крайне неожиданно выйти боком
  + Сразу установить единый стандарт на схему построения URL, дополнительные параметры, наименование сущностей, работу со страницами, обработку ошибок и так далее. Не так страшно, что она будет не оптимальная, главное, чтобы прозрачная и понятная.
  + Схема URL должна отражать логическую иерархию между ресурсами / отношение родитель – ребенок и так далее. Держим путь как более простым, все дополнительное в параметры
  + Сразу вводить схему версионирования API, потом это будет гораздо больнее. Обычно версия вводится как стандартный префикс в URL
  + Аналогично сразу закладывать идемпотентность всех вызовов. Обычно, через параметр **requestUid**, генерируемый и передаваемый клиентом.

### HTTP запросы

* + Кодировка для всех запросов / ответов только UTF-8. Некоторые браузеры и библиотеки могут игнорировать заголовок charset. Будут ли средства защиты предоставлять средства перекодировки и анализа для произвольно кодировки не понятно.
  + Обязательно указывать заголовок **“Accept: application/json, \*/\*;q=0.01”** – именно так мы обозначаем, что обращаемся к API, а не пытаемся запросить обычную HTML страницу. Кстати, разные браузеры форматируют данную строку немного по-разному, так что на сервере лучше проверить просто вхождение токена **«application/json»**
  + При наличии тела запроса обязателен заголовок **Content-Type: application/json;charset=URF-8**.
  + При загрузке файла обязателен заголовок **Content-Type: multipart/form-data** и дальше
    - в первой части **Content-Type: application/json;charset=URF-8** и **Content-Disposition: form-data; name=”date”**
    - Для каждого файла **Content-Type: image/jpeg** и **Content-Disposition: form-data; name=”avatar”; filename=”user.jpg”**
  + Если используем CSRF токен (а его лучше использовать), то только в отдельном заголовке кросс-функционально для всех запросов. В куках не безопасно – можно украсть.
* JSON - разные клиенты и разные парсеры могут интерпретировать их по-разному
  + Использование кодов символов и html-сущностей не допускается, то есть применяем режим JSON\_UNESCAPED\_UNICODE.
  + Для строк используем только двойные кавычки

### Stateless сервера

* каждый запрос должен быть независимым от других и быть обрабатываемым произвольным сервером кластера, промежуточное хранение данных между отдельными запросами не допускается
* это дает возможность достижения надежности / масштабирования путем раскидывания запросов по серверам, а также возможности в любой момент поднять / опустить произвольный сервер
* Возможные проблемы
  + В этом случае размер запроса может существенно увеличиться, например, когда каждый раз отправляем полностью состояние формы с данными, а не событие перехода пользователя из одного поля в другое.
    - Нет предыдущего состояния в сессии на сервере – нет возможности оптимизировать взаимодействие.
    - Слегка лечится комбинацией
      * **statefull сервера -** к ним подключаются пользователи, на них хранятся пользовательские сессии и осуществляются простейшие операции: обогащение запроса данными сессии, выполнение запроса за счет stateless сервера, обновление сессионной информации, конвертирования данных ответа и возврат пользователю.
        + Используются в основном как временное хранилище а оперативной памяти сессионной информации.
      * **stateless сервера -** выполняют реальные бизнес операции на основе «тяжелых» запросов, сформированных statefull серверами. Несут в основном вычислительную нагрузку или как конвертеры данных, при обращении к БД
      * Слегка, потому что в случае чего сессией пользователя придется все-таки пожертвовать - реплицировать ее между серверами очень дорого
  + Если используется Hibernate / JPA кэширование второго уровня, то рандомный разброс запросов по кластеру обработчиков приведет к тому, что все кэши на всех серверах будут пытаться сохранить все данные из базы.
    - Решение – слабое связывание: по мере возможности пытаемся обрабатывать запросы от одного пользователя на одном и том же сервере. Если нет или он тормозит – на каком придется.
    - Это требует хитрого входного балансира, мне не известно решение, поддерживающее подобное из коробки. Делали сами на основе JMS между web и app серверами, как именно – выходит за пределы дайджеста

### Сортировка

* Общий подход: набор наименований полей через «,», перед именем может быть префикс «-». Недостаток – слишком гибко, сложно будет создать все возможные наборы поддерживающих индексов в БД
* Набор именованных вариантов сортировок – более подходит для высоконагруженных систем, потому что не возможно поддержать все возможные варианты сортировки + фильтрации индексами на базе данных. А любой fullscan убивает всю систему нафиг
* Хорошо реализована в OData стандарте

### Фильтрация запрашиваемых данных:

* либо для каждого поля отдельный параметр, либо один параметр с закодированным запросом. Если пред
* Что делать, если предполагаются сложные и большие запросы
  + предусмотреть возможность расширения за счет специального заголовка, дополняющего параметры, переданные в URL. HTTP не устанавливает ограничение на размер подобного заголовка, но от сервера к серверу он разный: Apache - 8k, Nginx – 4/8K, IIS – 8/16K, Tomcat 8/4K, в случае переполнения получим 413 Entity Too Large или 4хх ошибку
  + В крайнем случае выполняем запросы через POST, где нет ограничений на размер, НО теряем дефолтное кэширование / автоматическое перевыполнение в случае сбоя и так далее. С другой стороны при подобного рода операциях кэширование нам особо и не нужно.
  + Два запроса:
    - POST на ресурс, представляющий собой коллекцию запросов, в ответ получаем хэш содержания запроса. Затем GET query=<полученный хэш>. Параметры, связанные с передвижением по курсору / сортировки, в запрос понятное дело не включаем, только данные фильрации
    - С одной стороны возможность длинного запроса, с другой нормально работает промежуточное ПО и прочие приятности транспортного протокола.
    - Увы, вместо одного запроса два и хранить эти запросы придется на быстром расшаренном хранилище – частичный stateful. Если в результате второго GET не был найден фильтр (хранилище перезагрузилось), то просто повторяем операцию.
* Опять таки: либо позволяем клиенту конструировать что хочется, либо предоставляем выбор из предопределенного набора фильтров и имеем потенциальный fullscan на базе
* Типовые запросы можно выставлять отдельными методами – аналог VIEW в базах данных
* Хорошо реализована в OData стандарте
* Полезно также предоставлять клиенту возможность определить перечень полей / встроенных сущностей, получаемых в запрошенном представлении ресурса / коллекции – projection.
  + Оптимально вообще предоставлять некоторое ограниченное представление + механизм запроса его расширения / ограничения
  + Projection может быть
    - статический (несколько именованных профилей данных) и динамический (клиент формирует профиль из произвольного набора расширений)
    - Выполняемым как за счет сериализации, так и за счет модификации запроса к БД.
      * Последний наиболее эффективен, так как позволяют избежать ненужных join и соответственно траты вычислительных ресурсов на всех уровнях приложения
      * Не забываем контролировать планы запросов + проблема N+1 запроса
  + Оба,Spring и JAX-RS, предоставляют средства статического projection за счт сериализации.
  + Если нужно больше – ручками. Интересная технология, которую в данном случае можно применить - **GraphQL**
* В случае сильно нагруженных систем лучше предоставлять клиенту не конструктор на основе мета-описания ресурса, а ограниченных набор профилей (сортировок / фильтрации / projection), которые он может комбинировать между собой
  + Имеем все-таки ограниченный набор возможных запросов к БД, есть теоретическая возможность их все оптимизировать
  + Можно написать автоматический интеграционный тест, который переберет и выполнит все комбинации, протестировав систему насквозь: от контроллера до построения и выполнения SQL запроса. Сложно в реализации, но потом экономит очень много сил при эволюции / исправлении системы.

### Валидация данных

* Для **POST / PATCH / PUT** запросов, в случае проблемы клиенту возвращается **422 Unprocessable Entity**
* Полезно возвращать верхнеуровнюю ошибку с кодом «Validation Failed», а в качестве вложенных данных
  + перечень реальных ошибок, каждая из которых описывается отдельным кодом (Too long, mast present, illegal charactes и так далее) и ссылается на определенный путь в полученном в запросе бине данных
  + Исходный документ, элементы которого вместо данных содержат ошибки. Или продублированы элементами с соответствующими именами – somePropertyError

### Поддержка долгих запросов

* Запросы клиента, которому нужен результат, всегда должны выполняться синхронно и быстро.
  + Если клиенту результат не нужен, то его можно отпустить и продолжить выполнение.
  + Успешность выполнения это тоже результат и требует синхронности / блокировки клиента
* Варианты с долгими запросами
  + Запрос в хранилище, уникальный ключ возращаем клиенту и отпускаем его. Затем он ходит к нам на стандартный URL и предъявляя ключ получает результат или сообщение, что он еще не доступен
  + В случае взаимодействия двух систем: в качестве одного из аргументов клиент предлагает callback URL или уникальный ключ, с использованием которого сервер потом вызовет клиента самостоятельно. Доступно, только когда клиент также является адресуемым сервером.
  + Используем Comet / SignalR / WebSocket / Server Side Event и все такое

### Асинхронная работа

* Как правило оптимистическая блокировка на основе **Entity Tags (ETag)**
  + Заголовок **ETag** с уникальным идентификатором состояния ресурса возвращается вместе с ответом.
    - Идентификатор может быть полных хэшем состояния, что достаточно дорого.
    - Можно (и лучше) использовать время последнего обновления / версию состояния в БД / UUID, который хранится в составе ресурса и перегенерируется при любой операции над
    - Важно: тэг отражает состояние ресурса, а НЕ сгенерированного представления. Соответственно удобные средства автоматической генерации тэга на основе HTTP фильтров / интерцепторов ответа на этом лажают, так как они вычисляют его как хэш возвращаемых данных. Кроме того, они жрут достаточно много ресурсов, что тоже не очень хорошо.
  + Последующий запрос на изменение должен содержать заголовок **If-Match** с соответствующим полученным значением
  + Заголовок **If-None-Match** применяется обычно с **GET**, когда генерация представления ресурса занимает большое время и хочется сэкономить ресурсы сервиса. Соответственно, если условие выполнено, то вместо 200 с данными возвращается просто 204
  + Также могут использоваться
    - для управления кэшированием
    - для ускорения операций обработки, так как для возврата 409 не надо парсить весь ответ и производить дорогостоящее сравнение данных
  + Клиент не обязан использовать **ETag** , даже если ему его передали. Сервер должен быть к этому готов. Соответственно данная информация может также ходить в составе состояния ресурса.
  + Если выявлена коллизия, то сервер должен вернуть **412 Precondition Failed**. В этом случае клиент должен выполнить **GET**, получить новый тег, выполнить операцию над полученным обновленным состоянием ресурса и повторить запрос или отказаться от операции вовсе
  + С ответом 204 сервер может также вернуть **ETag** , не возвращая новое состояние ресурса
  + Вместо конкретного значения оба заголовка могут содержать \*
    - Для **If-None-Match** это означает применить запрос, если ресурса не существует (**POST** конкуренция)
    - Для **If-Match** соответственно применить запрос только в том случае, если ресурс существует (**PUT / DELETE** конкуренция)
  + Некоторые тонкости
    - Ресурс может поставляться в различных представлениях, как с точки зрения протокола (XML/JSON), так и с бизнес точки зрения (full/detail, различные виды поставки одного и того же товара и так далее). В этом случае генерировть ETag автоматически на основаннии содержимого ответа нельзя, надо использовать бизнес информацию, например, поле version БД.
    - Если применятся сжатие данных, то ZIP формат имеет внутри метку времени, соответственно вычислять ETag надо до сжатия, а не после
    - ETag может использовться серверми кэширования. Далеко не всеми, но например Squid его понимает.
* Аналогично с метками времени
  + Заголовки **Last-Modified / If-Modified-Since / If-Unmodified-Since**
  + Главным образом применяются для кэширования
  + Для контроля конкурентного обновления применимы плохо, так как метка времени указывается с точностью до секунды, что зачастую недостаточно
  + Кроме того, **Last-Modified** может быть в трех разных форматах, что тоже не облегчает его использования
* Смешивать два подхода крайне НЕ рекомендуется, результат может оказаться совершенно непредсказуемым

### Надежность при передаче сообщений

* Основная проблема использования НЕ-idempotent метода **POST** при создании нового ресурса – что делать, если был потерян РЕЗУЛЬТАТ операции
  + Если был потерян запрос – ничего страшного, повторим. А вот повтор при потерянном результате предыдущей попытки может выйти боком. Например, для счета будет зарегистрировано два идентичных платежа
  + С НЕ-idempotent **PATCH /POST** при обновлении все проще – если ресурс уже существует, то это всего лишь частный случай конкурентного обновления. Закрывается, например, за счет ETag с версией содержания ресурса
  + **GET, DELETE и PUT** вообще проблемы не представляют, при правильой реализации их можно повторять сколько угодно
* Типовые варианты решения:
  + Не использова подобные операции
  + у каждого запроса уникальный UUID, генерируемый пользователем – плохо, мы пользователю не доверяем - вполне может «по-умному» его перегенерировать при повторном отправлении запроса
  + вычисляем хэш его данных на основании бизес параметров – не всегда применимо, вполне возможно, что он действительно хочет послать подряд два идентичных платежа – имеет право
  + Одноразовый защищенный токен
    - Сначала пользователь запрашивает у коллекции токен, который генерируется, например, в виде подписанного UUID. Затем предъявляет его в качестве данных POST или в виде заголовка
    - Сервер сохраняет токен в составе ресурса, а потому может отследить был он уже раз использован или нет. Во втором случае вместо выполнения операции сервер возвращает уже имеющийся результат.
    - Клиент сам сгенерировать токен не может
      * Хорошее место для контроля доступа и ограничения потока данных
      * Можно предотвратить DDOS атаку
        + чем больше имеющейся поток запросов, тем медленнее генерятся токены для следующих запросов
        + Сервер может проверить валидность токена очень быстро и без обращения к хранилищу данных
    - Достоинство – сгенерированные токены хранить (а также неизбежно удалять запрошенные и не использованные) не надо
  + Вариант токена – генерация черновиков
    - Клиент выполняет **POST** некоторых минимальных данных для создания ресурса в коллекцию
    - Ему возвращается **Location** с временным URL созданного черновика ресурса
      * Хорошее место, чтобы определить дефолтные данные
      * URL может вообще не относиться к данной коллекции, например, быть случайным или в специальной суб-коллекции
    - На данный URL он уже
      * заливает данные (или читает + заливает, не важно) при помощь **PUT**, которая является идемпотентной операцией.
      * выполняет команду публикации ресурса, смены статуса с «черновик» на «создан»
    - Проблема, что делать с забытыми черновиками
  + Специальные многостадийные протоколы передачи данных, например **HTTPRL** - протокол надежной передачи данных поверх HTTP.
    - Описывает два процесса: надежную передачу с клиента на сервер (uploading) и c сервера на клиент (downloading).
    - 100% защиты данные методы все равно не дают, потому что клиент вместо сбойного HTTP запроса вполне может повторить всю цепочку вызовов с самого начала. Но это уже вполне намеренное поведение и виноват в этом только он сам

### Обработка ошибок

* Общий принцип – как можно больше придерживаться стандарта HTTP, чтобы дать возможность нормально работать промежуточным серверам и используемым клиентским библиотекам
  + Не следует увлекаться экзотическими кодами просто потому, что их описание кажется знакомым – на них может быть навешено широко распространенная дефолтная обработка. Например
    - 401 запускает HTTP аутентификацию, которая будет странно смотреться
    - Балансировщик, получив 503 от сервера, будет считать, что нода временно не доступна
  + Другой анти паттерн – отвечать 200 на все и возвращать ошибку в теле ответа. Соответственно отключаем целый слой дефолтного поведения системы и усложняем себе жизнь
* Развернутую информацию при любой ошибке можно вернуть в теле, лучше в формате text/plain + UTF\_8.
  + Устойчивое заблуждение, что только вместе с 200 можно вернуть тело и поэтому его нужно использовать в том числе и при обработке ошибок
  + JSON рекомендуется не использовать, разве что в случае возвращения ошибок валидации данных запроса
* 500 это код НЕПРЕДВИДЕННОЙ ошибки, с которой ничего нельзя сделать и которая либо никогда не повторится, либо возникает всегда.
  + Типовая реакция балансировщика – пропускает на клиента, нода не отключается даже временно. А вот если балансировщик от какой-то ноды получает ее часто, то может ноду отключить нафиг
  + Выбрасывание 500 это дефолтная реакция системы на ошибку и признак именно непредвиденной проблемы. Соответственно, если ошибка хоть как-то обрабатывается или сознательно порождается, то лучше использовать другие коды. Это код НЕ ПРИКЛАДНОГО уровня.
  + Кроме того, это алерт для системы мониторинга и повод админам вмешаться.
* 503 – код локальной ошибки
  + Например, это недоступность третьей системы к которой производится обращение. Или отсутствие необходимых данных на данной ноде
  + Соответственно, получив 503 балансировщик пойдет на другую ноду и в результате клиент получит 200 вместо ошибки
* 422 – код ошибки валидации данных сервером, в теле выдаем перечень ошибочных полей с комментариями. Нормальная ситуация с точки зрения сервера. А вот попытаться обработать запрос, получить NPE и вылететь с 500 – это совершенно справедливо есть признак наличия бага.
* Единственный вариант, когда 200 используется для возвращения ошибок это batch операции.
  + Например, когда мы загружаем одновременно несколько документов
  + Тогда для каких-то элементов задания нам надо вернуть ОК, а для каких-то ошибочный результат с комментариями. В целом возвращаем 200 + JSON описание результата и пусть клиент разбирается

# Приложение 2 – работа с БД

## QueryDSL

* Средство построение статически типизированных звпросов при помощи fluent API, относительно JPA это альтернатива JPQL запросам и использованию Criteria API
  + Недостатки ручной сборки JPQL понятны
  + Criteria API вроде предлагает type safety и тоже построена на генерируемой в процессе сборки метамодели, но получающийся в результате код очень сложный и непонятный
* Основные достоинства подхода:
  + Поддержка IDE
  + Контроль синтаксиса вплоть до невозможности построить синтаксически неправильный запрос
  + вместо конкатенации строк оперирование доменными типами и свойствами, представленными в виде специальных классов и их статичских свойств
  + Значительно облегчается рефакторинг
  + Большинство ошибок выявляются во время компиляции
  + Хорошо поддерживается динамическое построение запросов
* Единая идеология + API для работы с различными источниками данных: JPA / Hibernate / JDO / JDBC / Lucene / Hibernate Search / MongoDB / Collections / RDFBean
* Поддержка языковых расширений для Scala
* Основные зависимости:
  + **query-dsl-jpa** – собственно runtime библиотека, для остальных технологий есть аналоги
  + **query-dsl-apt** – APT (annotation processing tool) препроцессор JPA сущностей, запускаемый при помощи **apt-maven-plugin**
    - ищет **@Entity** аннотированные классы и на основании их анализа строит метаданные в виде специальных java классов, которыми оперирует библиотека
    - зависимость только во время сборки, для работы не нужен
* Идеология:
  + **JPAAnnotationProcessor** в процессе сборки приложения ищет все JPA сущности
  + Для каждой JPA сущности, например, **Customer**, строится класс метаданных, располагающийся в том же пакете, с префиксом “Q”. В нашем случае это **QCustomer**. Классы по умолчанию располагаются в **target/generated-sources/java** и доступны для анализа
  + Класс содержит по умолчанию одноименное статическое поле **QCustomer.customer,** которое будет использоваться как вместо алиса сущности при построении запроса
    - **Customer cust = queryFactory.selectFrom(QCustomer.customer).fetchOne()**
    - Для большинства простых запросов данной константы вполне достаточно. Если же в запросе надо оперировать одновременно с несколькими сущностями одного типа, то можно создавать вручную через конструктор
  + Свойства класса являются элементами метаданных, описывающими структуру JPA сущности и содержащие методы формирования фрагментов запроса на их основе. Например, если имеется **Customer.getFirstName(),** то отбор по данному полю будет выглядеть как
    - **Customer cust = queryFactory.selectFrom(QCustomer.customer). where(QCustomer.customer.firstName().eq(“someName”) ).fetchOne()**
    - Соответственно поля имеют типы **NumberPath, StringPath, QSomeClass, SetPath< QSomeClass>** и подобные, методы которых служат для определения условий филтрации, группировки и так далее
  + Методы, которыми можно оперировать в **where** представлены **SImpleExpression**. Например: **eq, between, goe, loe, in(обычно подзапрос для выбора коллекции), ne**
  + Для комбинирования условий между собой также используются методы элементов Q-модели
    - **customer.firstName().eq(“someName”).or( customer.firstName().eq(“anotherName”) )**
* Недостатки
  + Каждый раз, когда обновляем структуру сущностей, необходимо сразу же пересобирать проект
  + Необходимо специально настраивать IDE, чтобы классы сгенерированной модели были нормально доступны. Иногда, она их теряет или кэширует старые без возможности обновления
  + Для сложных запросов бывает нелегко понять, что за SQL получится в итоге. Впрочем, дебаг выдача JPA провайдера сильно помогает
  + Нет insert функциональности. Намеренное решение разработчиков, предлагают использовать стандартную функциональность JPA

### Основное API

* Основные абстракции – **Query, Fetchable, Expression**
* Фабрики запросов: **JPAQueryFactory / HubernateQueryFactory**, обычно одна на все приложение, инициируется за счет **EntityManager / SessionFactory**
* Для работы с одной сущностью используем **selectFrom(),** для нескольких
  + **from(customer, company)**
  + **xxxJoin(customer.company, company).on(<условие на основе company>).** 
    - И customer и company это экземпляры Q модели.
    - Соответственно первый аргумент xxxJoin это источник данных, второй – алиас. В общем это аналог «xxx join customer.company as company on company.xxx==yyy»
    - Соответственно, если надо сделать join с двумя компаниями, то придется создавать через конструктор два экземпляра QCompany с различными идентификаторами
* Что выбираем
  + **selectFrom(customer)** – сущности
  + **select(customer.firstName, company.name).from()** – ряд свойств
    - в результате имеем **List<Tuple>,** от каждой **get()** опять-таки с использованием методанных. Например, **row.get(company.name)**
    - можно **select(new QTuple(customer.firstName, company.name))** – типизированная строка
  + в конце:
    - **fetchOne()** – если не найден, то null, если больше одного - исключение
    - **list(customer) или list(customer.firstName, company.name)**
    - **uniqueResult()** – исключение, если не один результат
* Общий поток вызовов:
  + Запрос - **queryFactory().selectFrom().where().orderBy().groupBy()**
  + Удаление - **queryFactory().delete().where().execute()**
  + Обновление - **queryFactory().update().where().set(qProperty, value).execute()**
  + Если мы хотим осуществить тюнинг запроса, то вместо выполнения вызываем **createQuery()** и имеем привычный JPA Query, который можно интегрировать с любым стандартными API
* Подзапросы
  + На основе статических **JPAExpressions**. Например, **JPAExpressions.select().from()** и так далее
  + Или **query = new JPASubQuery().from().uniqueResult(),** а затем в основном запросе в **where()** в функциях от свойств Q модели сущности
* Выбор литералов **в select()**
  + За счет **Expressions.constant(),** часто применяется в подзапросах
* Группировка
  + **groupBy(customer.company.name).list(customer.count(), customer.age.max())**
* Сортировка
  + **orderBy(customer.name.asc())**

### Продвинутые фичи

* **BooleanBuilder** – способ быстро строить однотипные запросы из наборов данных
  + **new BooleanBuilder(),** **затем builder.or().and()** и так далее
  + в конце **where(builder())**
* Можно мешать с нативным SQL
  + Для этого применяем **querydsl-maven-plugin**, который конфигурируем ссылкой на схему БД
  + В результате имеем Q классы, описывающие таблицы БД и их колонки
  + Для построения совместного запроса используется **JPASQLQuery**, в котором можно мешать Q эелементы, относящиеся к JPA и у JDBC
  + Методы для работы с SQL в том числе и весьма продвинутые. Например, работы с окнами данных
* Работа с коллекциями
  + В данном случае коллекция это источник данных, то есть обращения к внешнему серверу в приниципе не происходит
  + Кодогенерация и JPA / Hibernate фабрики запросов тут в простейшем случае не применяются, вместо этого используются статические методы класса Alias
  + Q модели соответственно тоже нет, вместо этого оперируем специальными экземплярами самих бинов
    - Получение мета описания: **Customer customer = alias(Customer.class, “customer”)**
    - Там, где раньше использовали **QCustomer.customer.firstName()** теперь применяем **$(customer.getName()),** который использует проксирование, чтобы превратить метод бина в экземпляр метаданных
  + Дальше примерно так же: **select ($(customer.getName()) ).from($(customer),** **customers).where($(customer.getName()).eq(“someName”)).fetchAll()**
  + Можно сгенерировать метаданные стандартным способом за счет **querydsl-apt + querydsl-collections + QueryDslAnnotationProcessor**
* Формирование готовых бинов из результатов выборки
  + В качестве аргумента **select()** используем **Projections.bean(Bean.class, qexpression1, qexpression2 и так далее).** В данном случае используются одноименные set-методы. Если **Projections.fields()** – будут использоваться свойства бина.
  + Аналогично **Projections.constructor()** будет строить экземпляры бинов на основании конструктора с соответствующим числом / типами аргументов. Можно указать какой именно конструктор за счет**@QueryProjection**
* Аггрегация результатов в памяти
  + Это НЕ SQL GROUP BY, выполняемая на сервере
  + за счет статических методов класса **GroupBy**
  + Пример: **query().from().where().transform( groupBy(post.id).as(list(post.comment))**

### Кодогенерация

* По умолчанию Q-модель для каждой сущности генерируется только на два уровня вниз. Чтобы продлить - **@QueryInit(“somePath”)** на необходимом свойстве сущности. В пути можно использовать «\*»
* При помощи **@Config** аннотации на пакетах / типов можно определить методы доступа к данным бинов
* За счет **@QueryType(SIMPLE|COMPARABLE|NONE)** можно, например, запретить сравнение по строкам, содержащим дату, или вообще убрать поле из Q-модели
* За счет статических классов с **@QueryEntity** и методов с **@QueryDelegate(SomeEntity.class)** можно добавлять свои методы к генерируемой Q-модели. Например, для расширения набора условий над стандартными типами данных – добавление **inPeriod() к QDate**
* За счет **GenericExporter** можно выполнять кодогенерацию самостоятельно, тонко настраивая кучу параметров. Все APT плагины используют в конце концов его

### Интеграция со Spring

* Делаем заготовку в виде **@MappedSuperclass AbstractEntity** со стандартным полем **“Long id”** и методом **equals(),** который использует его для сравнения сущностей
* Интерфейс абстрактного репозитария для работы с QueryDsl
  + Переметры дженерика: **T extends AbstractEntity, P extends EntityPathBase<T>, ID extends Serializable**
  + Расширяем **JpaRepository<T, ID>, QueryDslPredicateExecutor<T>, QueryDslBinderCustomizer<P>**
  + Реализуем дефолтный метод **customize()** как пустой
* Конкретный репозиторий **@Repository,** расширяющий абстрактный как **<Customer, QCustomer, Long>,** который нам предоставляет как минимум методы (находятся в **QueryDslPredicateExecutor**, полезно посмотреть как реализованы):
  + **findOne()** – один, null или исключение. Есть продолжение **orElse(),** позволяющее заменить null дефолтным значением
  + семейство **findAll()**
  + **count()**
  + **exists()**

## Обновление БД

* Даже самая первая версия, устанавливаемая для штатной эксплуатации, должна содержать функционал, контролирующий обновление схемы БД
* Рукописные решения, например, ручные скрипты обновления, рано или поздно жестоко выходят боком. Особенно, когда разработчики не имеют доступа к локальной копии «живой» БД для проверки очередных версий приложения / обновлений

### Liquibase vs Flyway comparison

|  |  |  |
| --- | --- | --- |
| Functionality | Liquibase | Flyway |
| Validation of migration scripts to prevent their altering | **+** | **+** |
| Types of migrations |  |  |
| DSL based migrations (XML, Groovy, Java) | **+** | **-** |
| Native SQL based migrations | + | + |
| Java code based migrations | + | + |
| Substitution of parameters | **+** | **+** |
| Execution of migration |  |  |
| Only once | + | + |
| Always | + | - |
| Only changed | + | - |
| With precondition | + | - |
| Smart error handling | + | - |
| Depending on context | + |  |
| Code organisation | Flexible, include based, structure | Flat list of scripts |
| Common operations |  |  |
| Update DB | + | + |
| Rollback update | + | - |
| Validate current state | + | + |
| Calculation difference with another DB | + | - |
| Calculation difference with JPA Eclipse Link based model | - | - |
| Generation initial migration | + | - |
| Generation update SQL for target DB | + | No, but can be easy implemented |
| Execution in offline mode | + | - |
| Calculation the difference between installed and current DB structure | No, but can be easy implemented | - |
| Initialization based on existing DB structure | + | + |
| Replacing set of small migrations with one big | - | - |
| Repairing installation | - | +- |
| Tools integration |  |  |
| Standalone tool | + | + |
| Maven | + | + |
| Ansible | + | + |
| SDK | + | + |

### Тривиальная реализация на основе Liquibase

* Оракловская специфика – одна миграция === одна транзакция, соответственно, невозможно в одной миграции атомарно создать таблицу и одновременно добавить FK
* В результате выполнения одной задачи появляется один файл с несколькими миграциями. Через некоторое время их становится столько, что инициализация с нуля / переход между двумя большими версиями начинает занимать много времени. Соответственно надо заводить «кумулятивные» файлы с обновлениями между конкретными версиями
* Пример структуры хранилища и наименований:

**db-migrations**

**/v-1.0**

**/sprint-1**

**/2013-03-02--01-developer1- some-change.xml**

**/2013-03-02--02-developer2-some-another-change.xml**

**/…**

**/sprint-2**

**/…**

**/changelog-v.1.0-cumulative.xml**

**/v-2.1**

**/...**

**/changelog-v.2.0-cumulative.xml**

**/changelog.xml**

* Перед применнием на боевой базе необходимо убедиться, что структура БД не была изменена вручную с последней процедры обновления
  + Руками за счет CRC от diff, сгенерированного между текущей структуры БД и базы соответствующей версии, созданной «с нуля»
  + Если мы сохраняем эту CRC в базе данных, то можем защитить приложение от старта на неправильной базе
* Крайне полезно с самого начала реализовать процедуру «анонимизации» содержимого реальной БД. Ряд проблем могут быть решены только потем анализа реальных наборов данных. Например:
  + Тестирование скриптов миграции данных
  + Проблемы с планами запросов, например, «full scan». Оптимизатор Oracle руководствуется содержанием БД и реальным распределением данных в индексах при принятии решений
  + Анализ данных при создании индексов / оптимизации запросов в процессе разработки

### Полуавтоматическое обновление на основе Liquibase

* Основная идея:
  + мы не описываем большинство миграций. Вместо этого мы генерируем их на основе сравнения текущей структуры БД и задаваемой набором JPA сущностей, используемых приложением (с учетом всех установленных плагинов / расширений)
  + в процессе развития приложения мы только расширяем структуру БД и никогда ее не сужаем
* Специальное веб-приложение с Liquibase, кодом генерации обновления и доменной моделью приложения
* Крайне простой UI, позволяющий
  + Получение описания текущей схемы БД
  + Построение разницы за счет
    - Auto-ddl + использование второй схемы
    - Собственный компаратор на основе, например, <https://github.com/ezhuravlev/org.liquibase.eclipselink>
  + Просмотр сгенерированного обновления в виде набора SQL запросов
  + Применение обновления с выводом лога выполнения запросов
* Данный подход позволяет
  + В большинстве случаев не писать DDL запросы / описания обновлений вручную совсем.
  + При старте приложения обновлять текущую БД прозрачно для разработчика, котому нужно только определить JPA сущность
  + Существенно уменьшить число специально реализованных миграций
    - Например, после 7 лет разработки сложной банковской системы с примерно 1000 таблицами было зарегистрировано только порядка 30 миграция. 92% из них было связано с преобразованием данных при рефакторинге сущностей

# Приложение 3 - Bean Mapping

## Orika Bean Mapper

* Основные характеристики
  + Конвертирует одно дерево объектов в другое, корректно отрабатывая циклические зависимости
  + Код генерируется в процессе инициализации за счет программно формируемых метаданных.
    - За счет этого конвертация выполняется значительно быстрее, чем в случае **Dozer**, например.
    - Но старт приложения и генерация мапперов в случае их большого количества занимает некоторое время. Можно самостоятельно реализовать lazy инциализацию мапперов
  + API для формирования метаданных fluent, оперирует String константами для наименования свойств
  + Использует reflection для доступа к properties бинов, также может задействовать public НЕ статические поля. Имеется возможность работы со свойствами, методы работы с которыми не соответствуют Java Beans соглашениям
  + Корректно отрабатывает работу с примитивами, их обертками, перечислениями, коллекциями, стандартными дженериками и так далее. Имеет для этого богатый набор встроенных конвертеров
  + Может работать с immutable типами за счет регистрации специальных конвертеров
  + В случае, если свойства двух бинов имеют одинаковые имена, то обрабатывает их в автоматическом режиме (если они одинакового типа или доступен конвертер типов)
  + Все не стандартные случаи легко могут быть кастомизируемы за счет собственных фабрик объектов, мапперов, конвертеров и так далее
  + Корректно отрабатывает маппинг наследников, а также маппинг в реализации интерфейсов и абстрактные классы
* Недостатки
  + Время старта приложение растет с числом мапперов
  + Ошибки маппинга трудно отлаживать
  + Для идентификакции свойств API оперирует строковыми константами и ошибки будут выявлены только в процессе старта приложения при попытке сгенерировать и скомпилировать мапперы. Разумеется не автоматического рефакторинга.
  + Среднее время выполнение операции и пропускная способность на несколько порядков ниже, чем у **MapStruct**
  + Интеграция самой библиотеки, а также генерируемого кода с DI контейнером требует специальных усилий
  + В целом технология маппинга не слишком прозрачная
* Зависимости: **orika-core**
  + С собой транзитивно подтягивает **javassist, slf4j, paranamer**
  + **javassist** является очень распространенной инструментальной библиотекой, поэтому контейнеры часто предоставляют свою версию и имеем с несовместимостью. Поэтому, имеется сборка с уже включенным javassist, классы которого перемещены во внутренние пакеты Orika
* Ключевые абстракции
  + **MapperFactory** – ключевой интерфейс за которым скрывается текущая конфигурация, загруженные метаданные и так далее. Является источником **MapperFacade** – уже конкретного инструмента преобразования данных.
  + **ObjectFactory** – создание объектов. Часто заменяется, когда создание подразумевает не просто вызов конструктора но и какую-то пост-обработку. Например, выполнение DI за счет **SpringBeanAutowiringSupport.processInjectionBasedOnCurrentContext(this)**
  + **Mapper** – копирование значений свойств из одного объекта в другой. Важно, что мы с самого начала имеем два объекта. Наиболее простой и часто применяемый способ преобразования объектов.
  + **Converter** – комбинация двух предыдущих случаев. Создается новый destination объект и затем заполняется значениями из исходного объекта. Используется для конвертации примитивных значений типа строки в объекты, а также когда надо не просто создать / сконфигурировать объект, но и после выполнить над ним какие-то еще действия.

### Простейший случай

* Создание новой фабрики - **new DefaultMapperFactory.Builder()… установка глобальных свойств…build()**
* Дальше мы должны задать метаданные
  + как маппить нестандартные поля классов (одноименные поля одних типов мапятся автоматом), конвертеры, полностью кастомные программные мапперы, какие типы использовать при маппинге на интерфейсы / абстрактные классы и так далее.
  + Например, **mapperFactory.classMap(fromClass, toClass).field(fromField, toField).byDefault(). Register().** Два последних метода важны – включение дефолтного маппинга и регистрация метаданных, описывающих маппинг двух классов
* Дальше получаем **MapperFacade** – средство маппинга
  + Простейший случай – **mapperFactory.getMapperFacade().** Он за счет **map(fromObject, toClass)** позволяет преобразовать бин в один из заданных классов.
    - Не очень оптимальный – каждый раз вычисляет дерево преобразования заново.
    - Есть обратный метод **mapReverse()**
  + Если нам надо постоянно преобразовывать данные из / в известной пары типов, то можно воспользоваться **mapperFactory.getMapperFacade(fromClass, toClass),** который даст **BoundMapperFacade** с улучшенной производительностью. Соответственно туда это **map(source)** и обратно **mapReverse()**

### Определение конфигурации маппинга А <=> B

* Начинаем с **mapperFactory.classMap(classFrom, classTo),** заканчиваем обязательно **register()**
* **byDefault()** – автоматически найдет и зарегистрирует маппинг одноименных (регистер учитывается) свойств
  + вглубь не идет
  + Можно задать собственный за счет **builder.classMapBuilderFactory().** Например, **ScoringClassMapBuilder**, который пытается угадать парность за счет близости наименований и типов и даже работает с коллекциями / проходами внутрь. Магия в общем…
* **field(«from», «to»)** – добавление двунаправленного маппинга двух свойств для пары классов
  + Если хочется одно направленного, то есть **fieldAToB() / fieldBToA()**
  + **«from» / «to»** это не обязательно только наименования полей, но и выражения, на основе полей
    - **from[n]** – значение из массива / списка по индексу
    - **from[‘key’]** – значение из Map по ключу
    - **from.p1.p2.p3** – проход внутрь по цепочке свойств
    - Разворачивание коллекции в Map за счет двух вызовов **field(),** определяющих
      * Мапинг ключей
        + **from{name}** – для каждого элемента коллекции from взять свойство name
        + **to{key}** – использовать значение слева как ключ в Map to
      * Маппинг значений
        + **from{}** – для каждого элемента коллекции from
        + **to{value}** – использовать значение слева как value в Map to
    - Аналогичные преобразования доступны для работы с другими типами коллеций
  + Если использовать **fieldMap((«from», «to»)…add(),** то можно задать дополнительные параметры работы с данной парой ствойств
* Исключение свойства из маппинга – **exclude(),** если этого не сделать, то выругается о том, что не знает как преобразовать данное свойство
* Какие конструкторы использовать для создания бинов А и В – **constructorA() / constructorB(),** в качестве параметров методов используются наименования параметров конструкторов
* Маппинг null значений:
  + По умолчанию null переносится как обычное значение
  + Можно поменять на нескольких уровнях
    - Глобально - **DefaultMapperFactory.Builder().mapNulls(false)**
    - Для класса – **classMap().mapNulls(),** действует на следующие маппинги свойств, можно менять туда-обратно по мере перечисления свойств
    - Для свойства – **fieldMap().mapNulls().mapNullsInReverse().add()**
* Доопределение маппинга классов – **classMap().customize( CustomMapper<A,B>).** Применяется уже после переноса данных. Может быть использован для пост процессинга конвертации данных
* Гибкая идентификация свойств бина
  + Позволяет переопределить, что есть **get / set методы**, сами имена, какие типы при этом использовать и так далее, Может применяться слева / справа в **field()** методе
  + Общий синтаксис: **name : { getter | setter [ | type=< типа значения>] }**
  + Например,
    - если вместо набора свойств у нас есть два метода **getAttribute(name) / setAttribute(name, value)**
    - то задать свойств employment мы можем за счет следующего выражения - **employment: { getAttribute(‘employment’) | setAttribute(‘employment’, %s) | type = SomeDataType}**
    - соответственно %s это процессируемое значение
  + Строчку можно построить также за счет **PropertyBuilder**
  + Если у нас таких типов, в которых надо специально определять свойства, много, то можно реализовать собственный **PropertyResolver** на основе **IntrospectorPropertyResolver**
    - Для каждого типа и наименование свойства должен сформировать объект **Property**, который используется для получения / установления значения данного свойства. Если super конкретным именем свойства лажает, то пробуем super еще раз уже со специальным выражением для того же свойства
    - Сконфигурировать его глобально за счет **factoryBuilder.propertyResolverStrategy()**

### Определение конфигурации MapperFactory – глобальные настройки

* **constructorResolverStrategy()**
* **compilerStrategy()**
  + дефолтная **JavassistCompilerStrategy() -** оптимальна по производтительности, но в случае чего проблемы с отладкой
  + более удобная для отладки и разбора проблем **– EclipseJdtCompilerStrategy(),** зависимость **orika-eclipse-tools**
* **propertyResolverStrategy()** – как преобразовать аргументы метода **field()** в средства получения / установки свойств бинов
  + по умолчанию **IntrospectorPropertyResolver** – базируется на java beans спецификации
  + можно применять **RegexPropertyResolver**, позволяющий задать кастомные выражения для распознавания **get / set** методов
* **classMapBuilderFactory()** – как сформировать дефолтный маппинг для пары классов
* **setAutoMapping()** – включение / выключение дефолтного маппинга
* **mapNulls()**
* **registerObjectFactory(factory, class)** – как создавать объекты специальных классов
* **codeGenerationStrategy()** – создает набор **Specification**, каждая из которых для маппинга **FieldMap** возвращает в виде String код **equalit()** и собственно код самого маппинга

### Кастомные конвертеры

* Реализация
  + расширяем **CustomConverter<A,B>** - это однонаправленный по умолчанию в одним методом **convert()**
  + Если хочется двунаправленный, то **BidirectionalConverter<A,B>** с двумя методами **convertTo() / convertFrom()**
  + **Protected mapperFacade** дает возможность пользоваться всей остальной функциональностью, например, для конвертации отдельных свойств
  + **canConvert(from, to)** – применим ли конвертер в конкретном случае
* Регистрация
  + На глобальном уровне:
    - **mapperFactory.getConverterFactory().registerConverter(converter)**
    - Перебираются от долее специфичным к более общим. Выигрывает первый, у кого **canConvert()=true**
  + На уровне свойства
    - **mapperFactory.getConverterFactory().registerConverter(converterId, converter)**
    - **fieldMap().converter(converterId).add()**

### Кастомные мапперы и фильтры

* Применяется, если мы не хотим определять для каждой **classMap()** пары
* Реализация
  + Расширяем **CustomMapper<from,to>.** Используем точные классы, если определим как **CustomMapper<?,?>** получим маппер из всего во все
  + Внутри ТОЛЬКО копируем значения из исходного бина в готовый бин назначения.
  + **Protected mapperFacade** дает возможность пользоваться всей остальной функциональностью, например, для конвертации отдельных свойств
* Регистрация
  + **mapperFactory.registerMapper()**
* Чтобы подправить дефолтный процесс маппинга можно использовать фильтры
  + Реализация интерфейса **Filter, CustomFilter / ничего не делающий NullFilter** - заготовки
  + регистрируется за счет **mapperFactory.registerFilter()**
  + Что можно:
    - Решить, маппим ли мы данную пару свойств вообще
    - Подправить значения from свойства после получением даннх
    - Подправить будущее значение to свойства перед установкой данных

### Тюнинг и проблемы

* Производительность:
  + Хранить **MapperFactory** как синглетон, его инициализация занимает много времени
  + Кэшировать **Type** классы, при маппинге дженериков
  + Использовать **BoundMapperFacade** если все время маппим одно и то же
  + Использовать **BoundMapperFacade**, получаемый как **mapperFactory.getMapperFacade(a,b,false),** если граф объектов заведомо не содержит циклов
* За счет уровня логирования DEBUG на пакете orika видно: создание classMap, генерация маппера, идентификация маппера для пары классов
* Если применяем стратегию **EclipseJdtCompilerStrategy**, то автоматически генерируемый в рантайм код маппера можно отладить по шагам
  + Зависимость **orika-eclipse-tools**
  + Активизируем либо за счет **MapperFactory.Builder**, либо устанавливаем системное свойство **OrikaSystemProperties.COMPILER\_STRATEGY** в наименование соответствующего класса
  + Сгенерированные исходники можно найти в **target/test-classes.** Принципиальные момент: файлы НЕ появятся до момента инициализации фабрики и соответственно формирования исходников мапперов
  + В принципе с этой же стратегией можно и штатно работать, но надо тогда для производительности запретить сохранять **исходники / class файлы** на диск за счет свойств **WRITE\_SOURCE\_FILES / WRITE\_CLASS\_FILES**

## MapStruct Mapper

* На основе Java Annotation процессора, который создает мапперы реализации интерфейсов / абстрактных классов с абстрактными методами конвертации, снабженными специальными аннотациями.
  + Затем эти методы могут быть использованы обычным образом для выполнения конвертации
  + Реализации методов генерируются в процессе сборки приложения и доступны для исследования / отладки
  + За счет этого процедуры инициализации маппинга и собственно самого маппинга НЕ используют reflection и выполняются максимально быстрым образом
* Достоинства
  + Прозрачность за счет доступности исходного кода
  + Быстрое выполнение за счет полного отсутствия динамических вызовов
  + Быстрая идентификация ошибок маппинга в процессе сборки приложения
  + Возможность подмешивать функциональность маппинга к любому другому коду
  + Инфраструктурный код минимален
  + Изначально хорошо совместим со всеми DI контейнерами типа CDI / Spring без необходимости выполнения дополнительной кастомизации
  + Совместим со многими нестандартными моделями бинов, например, Lombok
  + Много крайне нестандартных дефолтных конвертеров типа Joda-Time, JAXBElement и так далее
  + Легкое взаимодействие с ЖЦ маппинга за счет **@BeforeMapping / @AfterMapping**
* Недостатки
  + Кодогенерация
  + Не поддерживает immutable поля / объекты
  + При определении маппинга наименования свойств указываются как строки – нет автоматического рефакторинга
* Зависимости
  + **mapstruct** – набор необходимых для определения маппинга аннотация типа **@Mapping** и крайний минимум инфраструктурного кода
  + **mapstruct-processor** – препроцессор аннотаций
    - вызываемый обычно **maven-compiler-plugin**
    - конфигурируемый свойствами
      * **mapstruct.defaultComponentModel** – генерировать мапперы как компоненты. Варианты: **default / cdi / spring / jsr330**. Можно переопределить за счет **@Mapper#componentModel**
      * **mapstruct.unmappedTargetPolicy** – что делать во время препроцессинга со свойствами, не указанными при маппинге. Варианты: **ERROR / WARN (дефолтный) / IGNORE**. Можно переопределить за счет **@Mapper# unmappedTargetPolicy**

### Конфигурирование мапперов

* Интерфейс или абстрактный класс аннотируем **@Mapper**
* Определяем абстрактные методы конвертации **B aToB(A)**
  + На них за счет **@Mapping( source= “…”, target = “…”)** уточняем как конвертировать нестандартные свойства (одноименные поддерживаются автоматически)
  + Свойста должны быть реализованы как java beans get / set методы (fluent set – методы также поддерживаются) или public НЕ static поля
    - target поля соответственно не могут быть final)
    - Поддерживаются цепочки свойств
  + Методы конвертации могут принимать несколько параметров.
    - В этом случае, когда отображение не однозначно в **@Mapping#source** надо использовать префикс с именем параметра метода
    - Такие методы возвращают null если все переданные параметры null
    - **@Mapping#source** может ссылаться и просто на параметр метода
* Дополнительные возможности маппинга свойств
  + Вместо source ссылок на свойства можно использовать
    - **Ignore** – игнорировать target свойство при маппинге
    - **constant** – фиксированные значения
    - **expression** – выражения в виде java(код, который может быть вставлен в соответствующием месте при кодогенерации)
      * в принципе могут использоваться разные языки, но сейчас поддерживается только java
      * Во время кодогенерации корректность кода не проверяется, но при компиляции сгенерированного соответственно могут быть ошибки
      * в коде могут использоваться при помощи имен параметры метода конвертации
      * обычтно используется для вызова конструктора с параметрами
      * Если задействуются другие классы, то надо использовать либо полные имена классов, либо **@Mapper#imports** на интерфейсе
  + Вместе с **source / target** ссылок на свойства можно использовать
    - **defaultVaule** – что вместо null значения
    - **defaultExpression** - аналогично
* Если надо обновить бин вместо того, чтобы создавать новый, то добавляем единственный параметер типа B, аннотированный **@MappingTarget**. Метод при этом может возвращать void или по-преждениму В.
  + Что делать, когда исходное значение свойства source для метода обновления null
    - По умолчанию **target** свойство устанавливается в null
    - При помощи **nullvaluePropertyMappingStrategy=IGNORE** можно игнорировать подобные свойства
    - Аналогично при помощи **SET\_TO\_DEFAULT** можно установить инициализацию в только что созданные бин / дефолтное значение примитива пустую поллекцию и так далее
* Если метод возвращает супер-тип, то на нем при помощию **@BeanMapping#resultType** определяем какой именно из наследников создавать.
  + Аналогичные свойства для **@IterableMapping / @MapMapping**
* Тут же можно реализовать любой другой код в виде default или обычных методов. Это могут быть как
  + кастомные мапперы, которые будут задействованы при конвертации. В качестве кастомных мапперов будут использованы любые методы с любым названием, принимающие один параметр нужного типа и возвращающие результат нужного типа
  + так и любой другой бизнес код, главное, чтобы он не воспринимался препроцессором в качестве кастомного маппера
* Проверки на null в сгенерированном коде
  + Дефолтное поведение **(nullValueCheckStrategy=ON\_IMPLICIT\_CONVERSION)** это генерировать если
    - **target** это примитив, а source нет
    - применяется преобразование, создающее **targtet**, а затем **set** метод на **target**
  + Можно **ALWAYS** – всегда на не примитивном source
* Работа с исключениями
  + Можно самим определять на методах маппинга
  + Если верхнеуровневый метод не поддерживает подобное исключение, то оно будет обернуто в **RuntimeException**

### Создание объектов

* По умолчанию используются дефолтные конструкторы.
* Использование построителей
  + Задействуются для построенияimmutable объектов
  + Реализованы через собственный **BuilderProvider** SPI. Если для какого-то типа имеется построитель, то он будет задействован при маппинге
  + Дефолтный **BuilderProvider** реализует следующую логику
    - Тип должен реализовать статический метод без параметров, предоставляющий построитель
    - Построитель должен предоставлять метод без параметров, возвращающий исходный тип.
      * Если таки[ методов несколько, то используется **“build”**
      * Наименование метода может быть задано за счет **@Builder** вместе с аннотациями **@BeanMapping, @Mapper, @MapperConfig**
    - Если подобных методов несколько построитель не используется вообще, в процессе компиляции возвращается предупреждение
  + Соответственно в процессе маппинга target свойства используются не на самом классе, а на построителя. По окончанию вызывается build-метода и возвращается его результат
  + Шаблон **Object Factory** также подпадает под определение построителя
  + При наличии в **classpath** соответствующих классов поддерживаются построители: **Lombok, AutoValue, Immutables, FreeBuilder.**
  + Чтобы запретить использование построителей вообще надо через SPI задействовать **NoOpBuilderProvider**
* Фабрики объектов
* Но иногда конструкторов просто нет или как, например, в случае с JAXB необходимо пользоваться специально сгенерированными **ObjectFactory** классами
* Фабрика объектов это метод, обычно без параметров, аннотированный как **@ObjectFactory**, возвращающий нужный тип и определенный либо на самом интерфейсе маппинга , либо на подлюченном через **@Mapper.uses**
  + Может принимать параметр **@TargetType**
  + Может принимать дополнительные параметры - объекты ИЗ которых производится конвертация
* Что делать, когда на вход метода маппинга приходит null
* По умолчанию возвращается null
* При помощи **@BeanMapping#nullValueMappingStrategy=RETURN\_DEDAULT** можно вернуть свежий незаполненный бин target типа
* Поддерживается для примитивов, **iteratble, Map** и так далее
* Аналогичные атрибуты есть у остальных аннотаций маппинг

### Собственно конвертация

* Вне зависимости от заданной компонентной модели – **Mappers.getMapper(MapperClass.class).** Рекомендуется внутри маппера определять статическую константу **INSTANCE** с данным значением
* В компонентной модели
  + Если задана компонентная модель через свойство проепроцессора иди **@Mapper#componentModel,** то препроцессор генерирует маппер уже в качестве соответствующего компонента. Например, для Spring в виде **@Singleton** бина
  + Соответственно:
    - В клиенте **@Autowired** и используем
    - Внутри маппера также можно использовать **@Autowired** других сервисов и задействовать их в кастомных мапперах
* Стратегия DI
  + В **@Mapper** за счет аттрибута **injectionStrategy** определяем как будем доставляеть зависимости. Дефолтная это **InjectionStrategy.FIELD,** можно также **CONSTRUCTOR** для облегчения тестирования
  + Для абстрактных классов и декораторов надо использовать set методы для работы с DI

### Преобразование типов данных

* Очень много по умолчанию, например **String <=> int**
  + Автоматически между примитивами и врапперами, при этом null обрабатывается корректно
  + Если конвертация происходит с потерей точности, то можно за счет **@Mapper / @MapperConfig атрибута typeConversionPolicy** определить возможность контроля. По умолчанию игнорируется
  + **@Mapping / @IterableMapping** поддерживают аттрибуты **numberFormat** / **dateFormat** для работы со строками
* Много дефолтных конвертеров для всякой экзотики типа **JAXBElement** или **XmlGregorianCalendar**
* Дополнительные параметры
  + Методы подключенных мапперов могут принимать параметры **@TargetType типа Class**, чтобы выяснить, во что надо конвертировать значение. Часто применяется вместе с JPA, для загрузки сущности по первичному ключу во время преобразования DTO объекта. **EntityManager** для выполнения работы должен точно знать класс сущности.
  + Дополнительный контекст (состояние процесса конвертации) может быть передан по цепочке вызываемых методов маппинга за счет параметра, аннотированного как **@Context**
    - Передаются в также методы **@ObjectFactory, @BeforeMapping, @AfterMapping**
    - Создание контекста – ответственность вызывающего. Если в цепочке встретился метод без контекстного параметра, то контекст соответственно теряется и дальше соответственно null
* Маппинг коллекций
  + Любые **Iterable / Stream<?> / []** между собой автоматически, условные **iterable <=> non-iterable** не разрешены
  + При маппинге двух **Map** между собой отдельно преобразуются ключи и значения
  + **@Mapper / @MapperConfig** позволяют при помощи **CollectionMappingStrategy** определить, как именно будет формироваться коллеция на target.
    - Например, в случае JPA сущностей необходимо при добавлении элементов пользоваться **add** методом коллекции, чтобы правильно установить **parent / child** взаимоотношения между бинами
    - Варианты: **ACCESSOR\_ONLY, SETTER\_PREFERED, ADDER\_PReFFERED, TARGET\_IMMUTABLE**
* Маппинг перечислений
  + Просто определяем метод, конвертирующих одно перечисление в другое
  + Одноименные константы маппятся автоматически
  + Чтобы замаппить «разные» на методе **@ValueMappings** с коллекцией **@ValueMapping(source, destination)** внутри
  + В качестве source могут выступать специальные значения из **MappingConstants**
    - **ANY\_REMAINING** – в что маппить все остальные, то есть не одноименные и не специально замапленные
    - **ANY\_UNMAPED** – в что маппить все не специально замапденные. Отличие: сюда попадают и одноименые тоже!
    - **NULL** – понятно что

### Маппинг вложенных объектов

* По умолчанию всегда пытается идти внутрь, чтобы запретить **@Mapper#disableSubMappingMethodsGeneration**
* Использует для рекурсивного маппинга другие методы, определенные в этом же классе или в классах, подключенных за счет **@Mapper#uses**, выбирая при этом наиболее специфические по типам.
* Выбор вложенных методов мапинга можно уточнить за счет квалификаторов
  + Используются, когда для преобразования **А=>В** доступно несколько методов с эквивалентными сигнатурами
  + При помощи **@Qualifier** из пакета **org.mapstruct** определяем несколько аннотаций для вложенного интерфейса маппинга, присоединяемого через uses
    - Одна с **@Target(TYPE)** собственно на интерфейс
    - Несколько с **@Target(METHOD)** для методов интерфейса
  + В основном интерфейсе на методах в **@Mapping#qualifiedBy** передаем массив из аннотаций, определяющий какие именно из аналогичных методов вложенного интерфейса будут использованы
  + В принципе можно свои аннотации не определять, а использовать **@Named** аналогично на классе / методах вложенного интерфейса маппинга. Тогда в на методах основного интерфейса за счет **@Mapping#qualifiedByName** определяем какие именно методы будут использоваться для вложенного маппинга

### Повторное использование методов маппинга

* Подключение к конкретному интерфейсу маппинга других за счет **@Mapper#uses**
* Наследование конфигурации маппинга для методов
  + Для однотипных методов с **@Mapping / @BeanMapping / @IterableMapping**
  + Типовая ситуация: есть метод конвертации на котором используется **@Mapping,** а есть аналогичный метод обновления. Чтобы в нем не дублировать определения добавляем к нему **@IngeritConfiguration** и он наследует конфигурацию маппинга первого метода
  + Методы, конфигурацию которых можно наследовать должны располагаться в этом же классе, супер классах или shared конфигурациях
  + При помощи **@IngeritConfiguration.name** можно указать с какого точно метода берем конфигурацию
  + Унаследованная конфигурация может быть уточнена по мере необходимости
  + Если метода А наследует В, который наследует С, то в итоге А получает совместный результат со всеми промежуточными уточнениями
  + За счет **@IngeritInverseConfiguration** можно воспользоваться конфигурацией, чтобы определить «обратный» маппинг
    - Выражение и константы при этом понятное дело не наследуются, так как их нельзя «развернуть»
* Совместно используемые shared конфигурации
  + Определяем интерфейс, на нем **@MapperConfig**
    - Там же определяем глобальные стратении типа **unmappedTargetPolicy** и так далее
    - в uses перечисление нескольких мапперов
  + В конкретном маппере используем за счет **@Mapper.config**
  + В интерфейсе shared конфигурации можно определять методы конвертации. Они не будут реализованы при кодогенерации, но их конфигурация может быть унаследована

### Кастомизация процесса маппинга

* Иногда необходимо применять некоторую логику до / после процесса конвертации
* Декораторы
  + Применение type safe логики к конкретным методам
  + На интерфейсе маппера совместно с **@Mapper** используем **@DecoratorWith(класс декоратора)**
  + Класс декоратора должен реализовывать интерфейс маппера (расширять, если маппер на основе абстрактного класса)
  + Связь с исходным маппером - **delegate**
    - Для **componentModel=defult** – в конструкторе принимаем **delegate** – ссылку на сгенерированный маппер
    - Для Spring - **@Autowired + @Qualifier(“delegate”)** на соответветствующем поле
  + в методах маппинга вызываем пред-логику, соответствующией метод конвертации от delegate, пост-логику
* **before / after callback методы**
  + для перехвата вообще всех методов маппинга в общем стиле
  + определяеются на самом интерфейсе маппинга, подключенных при помощию @Mapper.uses или на **@Context** параметрах
  + Аннотируются при помощи **@BeforeMapping / @AFterMapping**
  + Принимают параметры
    - **@MapperTarget / @TargetType**
    - **@Context**
    - Любые параметры, которые воспринимаются как source данные для маппинга
  + НЕ-null возвращаемое значение используеются как результат маппинга

### Кастомизация процесса генерации

* Все подключаются через SPI интерфейс – файлы с **META-INF/services**, именованые в соответствии с интерфейсом сервиса и содержащие наименование имплементации
* **AccessorNamingStrategy** – как распознавать get / set методы и выделять из них наименования свойств
* **MappingExclusionProvider** – где останавливать процесс маппинга вложенных бинов
* **BuilderProvider** – как разпознавать **Builder** шаблон и строить при помощи него объекты

# Приложение 4 - Bean Validation

* Обычная задача, которая встречается на каждом слое (клиент, презентация, бизнес, работа с данными).
  + В худшем случае логика проверки данных реализуется каждый раз отдельно для каждого слоя, что требует времени и чревато ошибками
  + Чтобы избежать дублицирования кода можно включить проверку данных в классы доменной модели, но это смешение концепций, потому что проверки это на самом деле метаданные, а доменная модель – реализация. Кроме того, зачастую классы доменной модели генерируются, например из XSD, что крайне затрудняет подобный подход.
* **JSR 380 (Bean Validation 2.0)** определяет модель метаданных и API для проверки бинов и аргументов / результатов вызываемых методов.
  + Дефолтные метаданные на основе аннотаций, но можно пере / доопределять при помощи XML и вызовов API.
  + API универсальное и может быть использовано в любом слое приложений, на сервере и на java based клиенте в том числе.
* Зависимости
  + Само API - **Javax.validation:validation-api**
  + Реализация API - **org.hubernate.validator:hibernate-validator**
    - Чтобы строить динамические сообщения об ошибках нужна реализация **Unified Expression Language (JSR 341).** Обычно она поставляется контейнером, но для SE надо добавить руками **org.glassfish:javax.el**
    - Если используем в CDI окружении, то дополнительно нужна интеграция hibernate c CDI – **org.hibernate.validator:hibernate-validator-cdi**

## Простейший сценарий

* На полях / свойствах (get-методах) / классах используем аннотации, например, **@NotNull / @Size / @Max**
* Строим реализацию **Validator** за счет **Validation.buildDefaultValidationFactory().getValidator()**
  + И валидатор и фабрика повторно используемые, **ThreadSafe**, можно спокойно засунуть их в статическую переменную для глобального доступа
  + Создание фабрики требует ресурсов (она собирает, загружает и кэширует метаданные), поэтому ее лучше кэшировать.
  + Создание валидатора с расширениями достаточно дешево, так как это фактически контекст процесса валидации
* **Validator.validate(bean)** дает нам **Set<ConstraintValidation>,** если все нормально, то пустую, в противном случае по экземпляру на ошибку валидации
* За счет **ConstraintValidation** выясняем в каком свойстве бина (переданного или дочернего) имеем ошибку, какую именно, каково ошибочное значение и так далее
* В простейшем случае нет необходимости использовать какие-либо Hibernate специфические классы

## Проверка бинов

### Определение ограничений

* Основной способ определения ограничений – за счет аннотаций на полях, свойствах (get – методах), классах, классах – параметрах дженериков
  + Чтобы понять где применимо смотрим значение **@Target,** используемое в определении аннотации ограничения
  + Для полей:
    - в процессе валидации движок прямо обращается к полям минуя методы
    - поля могут быть любого типа доступа, но НЕ могут быть static
    - Если используются enchanced objects, то поля более не доступны (реально движок имеет дело с прокси оберткой), поэтому надо применять ограничения на свойствах
  + Для свойств
    - Наличие set-методов не обязательно
    - Так как применяем к get-методам, то не возможно проверить значения в процессе установки. Для этого необходимы ограничения на уровне вызываемых методов
  + Для классов
    - Чтобы проверить взаимосвязь между несколькими свойствами бина
    - Hibernate поставляет только один готовы - ???
  + Для параметров дженериков
    - **@Target = ElementType.TYPE\_USE**
    - Пример: **Set<@Size(min=2, max=14) String>.** Поддерживаются также и для вложенных дженериков: **List< Set<@Size(min=2, max=14) String>>**
    - Автоматически поддерживается для **Iterable, Map, Optional, OptionalXxx**
    - для остальных надо реализовывать способ экстракции данных из контейнера – дженерика в виде класса, расширяющего **ValueExtractor<SomeContainer< @ExtractedValue?>>**
* Не рекомендуется смешивать аннотирование полей и свойств – некоторые проверки могут быть вызваны дважды
* Стандартные ограничения, предусмотренные API, не включают в себя применимые на уровне класса
* Если класс реализует интерфейс или наследуется от другого класса, то ограничения наследуются также. Для set-методов ограничения аггрегируются.
* Можно потребовать проверить также свойства вложенного бина (каскадирование) за счет **@Valid** на поле / свойстве / параметре дженерика. Null значения при этом игнорируются. Движок корректно отрабатывает возможность зацикливания структуры

### Выполнение проверки

* методы validator:
  + Все методы validator имеют var-args параметры для определения групп валидаторов, которые должны учитываться. По дефолту используется группа **Default**
  + **validate(bean**) – проверка собственно бина, простейший случай
  + **validateProperty(bean, “propertyName”)** – проверка только одного свойства бина
  + **validateValue(bean, “propertyName”, value)** – проверка возможного значения одного свойства бина
  + Два последних метода НЕ учитывается **@Valid,** соответственно каскадирования проверки НЕ производится
* Свойства **ConstraintViolation**
  + **message / messageTemplate** – готовое сообщение об ошибке / шаблон сообщения
  + **rootBean / leafBean** – бин переданный методу проверки / бин на котором зарегистрирована ошибка. Аналогичные методы с Class постфиксом
  + **propertyPath** – путь к ошибочному значению от **rootBean**. Тип **Path**, содержащий коллецию **Node**, каждая из которых фрагмент пути.
  + **invalidValue**
  + **constratintDescriptor**

### Готовые ограничения

* В основном это предусмотренные API, но есть и Hibernate расширения
* Некоторые также используются Hibernate для выполнения DDL в процессе применения ORM
* Для большинства ограничений Hibernate расширяет применимость по типам. Например, **@Max** может быть применим к String свойствам, при этом значение конвертируется в числовое
* **Стандартные**
  + **@AssertFalse, @AssertTrue**
  + **@DecimalMax, @DecimalMin** – проверяемое значение приводится к **BigDecimal**
  + **@Digits**
  + **@Email**
  + **@Future, @FutureOrPresent, @Past, @PastOrPresent**
  + **@Max, @Min, @Negative, @NegativeOrZero, @Positive, @PositiveOrZero**
  + **@NotBlank** – только к строкам, предварительно удаляются пробельные символы
  + **@NotEmpty** - к строкам (в отличии от предыдущего случая оставляются как есть) и коллекциям
  + **@NotNull, @Null**
  + **@Patern**
  + **@Size** – к строкам и коллекциям
* **Дополнительные**
  + Бизнес проверки - **@CreditCardNumber, @Currentcy, @EAN, @ISBN, @LuhnCheck**
  + **@DurztionMax, @DurationMin**
  + **@Length** – то же самое, что Size
  + **@CodePointLength** с примением кастомизируемой normalizationStrategy
  + **@Mod10Check, @Mod11Check**
  + **@Range**
  + **@SafeHtml**
  + **@ScriptAssert** – единственный уровня класса, можно настраивать язык, собственно скрипт и поле на котором будет ошибка. Пример значения **– ‘’property1 + property3 < property2**”
  + **@UniqueElements** – на любой коллекции. Чтобы в результате был перечень дубликатов – добавить в шаблон сообщения **{duplicates}**
  + **@URL** – можно также зафиксировать протокол, хост, порт, потребовать соответствия **regexp**. Использует URL конструктор, поэтому выполняет сетевое обращение. Если хочется без этого, то надо использовать **RegexpURLValidator**
  + Проверки, локальные для стран типа **@CNPJ – Brazilian corporate taz payer registry number**

## Проверка методов

* Можно определять и проверять ограничения на значения параметров методов / конструкторов и возвращаемые ими значения – пре / пост условия вызова методов
* НЕ работают для static методов
* Автоматически НЕ вызываются - главным образом для использования внутри АОП / EJB / CDI интерцепторов, прокси и так далее
  + Не надо вызывать руками, о чем легко забыть
  + Будут автоматически отражены в документации JavaDoc (не забывать **@Documented** для собственных проверок)
  + Соответственно вызывающий такие методы код получает **ConstraintViolationException**
* Чтобы отличить подобные проверки от проверок бинов:
  + На реализации **ConstraintValidator** добавляется **@SupportedValidationTarget**
  + На самой аннотации проверки свойство **validationAppliesTo = ConstraintTarget.PARAMETERS** или **RETURN\_VALUE.** Если метод не принимает параметров или возвращает void, то есть значение очевидно – можно опускать.
* Аналогично поддерживается **@Valid,** чтобы запустить проверку вложенных бинов
* При наследовании: пред-условия не могут быть уже, пост – соответственно шире (Liskov Substitution Principle).
  + Соответственно, если на методе родителей не было ограничения вовсе, то на методе наследника добавить его не получится
  + К конструкторам это все вообще не относится – они не наследуются
  + Можно настроить за счет **HibernateValidationConfiguration.methodValidationConfiguration**
* Чтобы выполнить нужен специальный валидатор – **ExecutableValidator**, получаемый за счет **factory.getValidator().forExecutables()**
  + Для данного валидоатора можно завести отедльную фабрику со специальнуми настройками
  + **validateParameters() / validateReturnValue()** – передается соответствено object, Method и массив параметров / возвращенное значение
  + **validateConstructorParameters() / validateConstructorReturnValue()** – аналогично, но вместо **Method** соответственно **Constructor**
* в возвращаемых **ConstraintViolation** при этом используются методы **getExecutableParameters() / getExecutableReturanValue().** В **getPropertyPath()** можно узнать имя класса / метода / индекс ошибочного параметра
* Единственный готовый валидатор - **@ParameterScriptAssert**
* При разработке такие валидаторы:
  + Должны быть аннтированы **@SupportedValidtionTarget(ValidationTarget.PARAMETERS)**
  + Обрабатывать типа дынных **Object[] / Object**

## Построение сообщений об ошибке

* Определение каждого ограничения содержит шаблон сообщения об ошибке в свойстве аннотации
  + Во время применения аннотации можно переопределить за счет атрибута message
* Шаблон содержит ссылки на свойства ограничения в виде **{property},** а также выражения в виде **${}**
  + **{value}** – это свойство ограничение (всегда есть, единственное, которое можно использовать в аннотации без атрибута), **{validatedValue}** – проверяемое значение
  + Можно использовать **${ formatter.format(….) }**
* Развертывание шаблона осуществляет **MessageInterpolator**
  + Обычно, свойство аннотации ограничения возвращает ключ, а дефолтный **ResourceBundleMessageInterpolator** используется, чтобы по ключу достать шаблон и потом его раскрыть
  + За счет **Validation.byDefaultProvier().configure().messageInterpolator()** можно подсунуть свой собственный, который будет грузить шаблоны откуда-то еще или реализовывать расширенный синтаксис шаблонов. Реализация должна быть Thread Safe.

## Группировка ограничений

* Позволяет ограничить перечень ограничений, применяемых в конкретном случае при проверке бина.
  + Каждая группа это просто маркерный интерфейс без каких-либо дополнительных требований
* Простейший случай
  + При применении ограничения к свойству в аннотации можно указать группу за счет атрибута groups. По умолчанию используется группа **Default** – дефолтная группа ограничений
  + При выполнении проверки указываем перечень групп - **validator.validate(bean, Group1.class, Group2.class),** выполняются только ограничения, относящиеся к любой из групп, в произвольном порядке
* Комбинирование групп
  + Группы могут расширять друг друга просто за счет наследования интерфейсов.
  + Группы можно объединять за счет определения группового интерфейса с **@GroupSequence( {…} )** на нем.
    - Соответственно, когда используем групповой интерфейс в **validate()** ограничения выполняются сначала для первой группы, потом для второй и так далее
    - Если просто передаем перечень групп в **validate(),** то порядок выполнения НЕ определен
    - Соответственно надо следить, чтобы не было зацикливания
* За счет **@GroupSequence** на уровне класса можно переопределить дефолтную группу ограничений для класса
  + Применяется ТОЛЬКО для данного класса, не влияет на проверку вложенных бинов. Чтобы переопределить что-то для них можно на свойствах использовать **@ConvertGroup(from, to).** Некоторые ограничения
    - Применяется только вместе с **@Valid**
    - Можно несколько аннотаций с одним и тем же **to**, но они все должны иметь разные **from**
    - From не может указывать на группу аннотаций
  + Это статическое переопределение, можно выполнить динамическое за счет **@GroupSequenceProvide( класс, реализующий DefaultGroupSequenceProvider<BeanClass> ).** В процессы выполнения экземпляр провайдера получает ссылку на бин и возвращает перечень групп в зависимости от состояния бина

## Разработка ограничений

* Из трех частей: аннотация, определяющая статические и динамические свойства ограничения, валидатор, реализующий логику проверки, шаблон сообщения об ошибке

### Аннотация

* Определяется при помощи **@Interface** стандартным образом и несет на себе следующие аннотации
  + **@Target({…})** – к каким элементам применимо ограничение.
    - Стандартные значения это **FIELD, METHOD, PARAMETER, TYPE\_USE**
    - **ANNOTATION\_TYPE** – возможность использования при создании композиции ограничений
  + **@Retention(RUNTIME)**
  + **@Constraint( validatedBy = <класс валидатора> )** – ссылка на реализацию. Если ограничение применимо к нескольким типам данных, то перечень валидаторов
  + **@Documented**
  + **@Repeatable** – возможность применения аннотации несколько раз
* Должна иметь следующие обязательные **default {}** методы
  + **String message** – в случае **MessageInterpolator** ключ шаблона сообщения об ошибке
  + **Class[] groups – либо {},** либо конкретную группу, если не предполагается использование других групп для данного ограничения, что редко
  + **Class<? Extends Payload>** - возможность за счет атрибута payload при применении аннотации передать некоторое значение, которое будет доступно в **ConstraintViolation**.
    - Сам движок проверки ничего с данным значением не делает
    - Как правило, за счет данного атрибута передают в **ConstraintViolation** «серьезность» нарушения – **INFO / WARN / ERROR / FATAL\_ERROR** (не ошибка данных, а ошибка выполнения самого ограничения)
* Остальные **default {}** методы (например, **value** – можно определять значение без указание атрибута) служат для определения атрибутов, кастомизирующих поведение ограничения

### Реализация валидатора

* Должен реализовать **ConstraintValidator< интерфейс ограничения, тип проверяемого значения>**
  + Соответственно, если ограничение применимо к нескольким типам данных, то реализуем несколько валидаторов
* Методы
  + **void initialize( экземпляр ограничения)** – возможность получить доступ к свойствам сконфигурированного на бине / свойстве бина и так далее ограничения.
  + **Boolean isValid(bean, ConstraintValidationContext)** – выполнение проверки
    - В простейшем случае просто возвращаем true / false, построение **ConstraintViolation** с сообщением об ошибки формируется автоматически
    - Рекомендовано null значение трактовать как true. В противном случае использовать дополнительную **@NotNull** аннотацию на свойстве
* Наличие **initialize()** метода делает валидатор НЕ Thread **Safe**, соответственно для каждого места применения ограничения создается отдельный экземпляр валидатора
* **ConstraintValidationContext** – возможность кастомизировать сообщение об ошибки
  + **context.disableDefaultConstratinViolation()** – отключаем стандартное поведение
  + **context.buildConstraintViolationWithTemplate()**
    - в принципе можно вообще без шаблона вернуть готовое сообщение, но шаблонирование позволяет безопасно обработать пользовательские данные (escape потенциального вредоносного кода, например). Аналогично можно ссылаться на параметры ограничения. НЕ допускать попадания в обобщение чувствительных данных.
    - Возвращает **ConstraintViolationBuilder** с цепочкой методов, в конце ОБЯЗАТЕЛЬНО вызывать **addConstraintViolation().** Только после этого новое состояние ограничения целостности будет зафиксировано
  + **builder.addPropertyNode()** – подмена наименования свойства, на котором будет зарегистрирована ошибка
  + Подмена НЕ означает регистрации ошибки – если метод в итоге вернет true, то **ConstratinViolation** создан НЕ будет
  + **unwrap(HibernateContraintValidatorContext.class)** – получение специфической для Hibernate реализации контекста с дополнительными функциями
* Можно в качестве базового класса использовать **HibernateConstraintValidator**
  + Тогда методу **initialize()** будут переданы два параметра: **ConstraintDescriptor и HibernateConstraintValidatorInitializationContext**
  + Доступ к аннотации сохраняется через **constraintDescriptor.getAnnotation()**
  + Контекст предоставляет расширенный доступ к хелперам, clock provider (тестирование ограничений, использующий **@Future**, например) и тому подобным глобальным сервисам
* Когда мы строим **ValidatorFactory**, при помощи **Validation.byProvider(),** или **Validator**, при помощи **HibernateValidatorFactory.usingContext(),** можно за счет **constraintValidatorPayload()** определить объект с разделяемыми между всеми валидаторами данными
  + Доступ – **hibernateContraintValidatorContext.getConstraintValidatorPayload()**
  + Это НЕ тот payload, который доступен за счет атрибута аннотации. Соответственно он НЕ доступен за счет **ConstraintViolation** при обработке сообщений об ошибках
  + Можно положить любые глобальные настройки, ссылку на Spring / CDI контекст, в общем это точка расширения глобального контекста валидации произвольными данными и сервисами

### Шаблон сообщения об ошибке

* По умолчанию заводим файл **ValidationMessaes.properties** в корне, туда свойства **<класс аннотации>=<шаблон сообщения>.** Соответственно можно иметь локализованные версии данного ресурса
* Если используем собственную версию **MessageInterpolator**, то можно хранить где угодно и в какой угодно форме. Например, в базе данных

### Композиция ограничений

* Если несколько ограничений все время используются совместно, то их можно заменить одним кастомным – композицией
* Просто на интерфейс аннотации композиционного ограничения навешиваем все ограничения, которые хотим совместить
* Композиционное ограничение может определять, а может и не определять собственный дополнительный валидатор, в этом случае **@Constratin( validatedBy = { } )**
* При нарушении такого ограничения получаем коллекцию **ConstraintViolation** под одной на члена композиции. Чтобы только одно единственное – добавляем **@ReportAsSingleViolation**

### API для исследования ограничений

* Вход через **BeanDescriptor validator.getConstraintsForClass()**
  + **isBeanConstrained()** – есть ли на нем вообще какие-либо ограничения
  + **getConstraintDescriptors()** – получение всех ограничений / **getConstraintsForXxxx()** – конкретных типов ограничений – свойств, методов, конструкторов. Возвращаются ограничения как самого класса, так и его родителей
  + **getConstrainedXxxx()** – получение свойств, методов, конструкторов, на которых есть ограничения
* Описания ограничений представлены различными наследниками **ElementDescriptor** – **PropertyDescriptor, MethodDescriptor, ConstructorDescriptor** и так далее....

## Bootstrapping & конфигурирование

* Для получения реализаций **ValidationFactory** & **Validator**, предоставляемых текущим провайдером, используются статические методы **Validator**
  + **ValidationFactory** - thread safe, повторно используемая, можно кэшировать в статической переменной и шарить между всем остальным кодом, хранит в себе все метаданные, достаточно медленно инициализируется
  + **Validator** - thread safe, повторно используемый, достаточно легковесный, при создании можно заменять многие дефолтные компоненты и настройки, предоставляемые **ValidationFactory**

### XML конфигурация

* За счет **META-INF/validation.xml.** Только один такой файл может содержаться в текущем classpath, если несколько – будет ошибка.
* Основные элементы
  + **validation-config** – определение реализации и свойств основных компонент: провайдера, построителя сообщений, фабрики валидаторов и так далее
  + **constraint-mapping** – определения ограничений и какие ограничения с какими свойствами применяем для каких бинов / методов.
    - Это аналог определения ограничений за счет аннотации, применяемый, когда исходный код модели данных не доступен или не может быть отредактирован. Например, сгенерированная JAXB модель
    - Крайне неудобно, лучше заменять программным конфигурированием
* Главная проблема «сырого» использования: каким образом поставить библиотеку валидаторов для повторного использования не вынуждая разработчика копировать фрагменты из одного конфигурационного файла в другой

### Программное конфигурирование

* **Validator.buildDefaultValidatorFactory()** работает за счет **META-INF/validation.xml** и найденного в classpath провайдера
  + Если провайдеров несколько, какой будет выбран не определено. Каждый провайдер декларирует свою реализацию за счет содержимого файла **META-INF/services/** **javax.validation.spi.ValidationProvider**
  + Поэтому лучше **Validator.byProvider(конкретный провайдер).configure(),** затем цепочкой методов конфигурируем будущую фабрику и наконец строим при помощи **buildValidatorFactory()**
  + Еще вариант**: Validator.byDefaultProvider().providerResolver().configure()** и так далее. Передаваемая как аргумент реализация **ValidationProviderResolver** может помочь при поиске в специфическом окружении.
  + По окончании использования фабрику лучше закрыть за счет **close(),** чтобы высвободить ресурсы приложения
* Когда начинаем конфигурировать уже имеем начальные настройки из **META-INF/validation.xml.** Чтобы сбросить – **ignoreXmlConfiguration(),** чтобы посмотреть изначальные значения - **getBootstrapConfiguration()**
* Что можно настроить:
  + **MessageInterpolator** – как получаем шаблон сообщения об ошибке и складываем его с данными
  + **TraversableResolver** – как перемещаемся по графу в проверяемом бине. Необходимо, когда какие-то свойства, к примеру, lazy инициализируемые или в случае JPA ведут к выполнению SQL запросов. Соответственно, для JPA можно переходить только по свойствам, уже загруженным и каскадируемым.
  + **ConstraintValidatorFactory** – каким образом инициализируем конкретные экземпляры ограничений на основе конкретных валидаторов.
    - Например, когда внутри валидатора используются **@Inject / @Autowired** конструкции и мы должны их дополнительно инициализировать по сравнению просто с вызовом конструктора
    - Данная фабрика НЕ должна кэшировать создаваемые ограничения
  + **ParamaterNameProvider** – как при проверке методов извлекать наименования параметров. Дефолтная реализация использует reflection API - либо компилируем проект с –parameters, либо будем видеть arg0 и так далее...
  + **ClockProvider** – поставщик текущего времени.
    - В основном, чтобы тестировать ограничения по времени и иметь возможность переопределить, что такое **now().** Для этого соответствующие валидаторы должны использовать **ConstraintValidatorContext.getClockProvider()** вместо **new Date()**
    - Также можно сконфигурировать **temporalValidationTolerance(),** чтобы противодействовать разбросу текущего времени между различными серверами кластера. В валидаторе получаем значение через **HibernateConstraintValidatorContext**
  + **ValueExtractor** – каким образом извлевать данные из generic типов, можно вызывать несколько раз для каждого из кастомных экстракторов
  + **ScriptEvaluatorFactory** – как загружать и исполнять скрипты, используемые валидаторами. Например, **@ScriptAssert / @ParameterScriptAssert**
  + **addMapping(InputStream)** – добавление кастомых фрагментов XML конфигурации
  + **addProperty()** – определение значения конфигурационных свойств
  + если применяем **byProvider(класс провайдера),** то становятся доступными методы, специфичные для конкретного провайдера. В данном случае:
    - **failFast()** – выполняем проверку до получения первой ошибки
* Когда получаем валидатор за счет **factory.usingContext()….getValidator()** многие перечисленные опции можно локально переопределить для данного валидатора. Например, **MessageInterpolator** или **TraversableResolver**

## Интеграция с другими фреймворками

* ORM
  + Так как активно используется проксирование, то лучше ограничения определять на get-методах
  + По умолчанию Hibernate ORM использует ограничения для формирования DDL определений. Отключаемо и настраиваемо для разных типов ограничений
  + Как правило настраивается за счет стандартных JPA **свойств javax.persistence.validation в файлах hibernate.cfg.xml / persistence.xml**
    - **Mode** - активация
    - **Group.pre-xxxx** – перечень групп, выполняемых перед вставкой, обновление, удаление
* CDI
  + Если есть зависимость **org.hibernate.validator:hibernate-validator-cdi,** то больше ничего делать не надо
  + Конфигурируется из XML
  + Можно **@Inject** фабрику и валидатор
  + В самих валидаторах можно **@Inject** остальные сервисы приложения и исользовать **@PostConstruct**
  + Если на методах (кроме get) есть ограничения аргументов / результата, то их вызов будет автоматически проверяться. Для get если хочется, то надо добавить **@ValidateOnExecution( type = ALL)** на метод или на класс целиком
* Spring
  + Поставляется с готовой интеграцией
  + Валидация вызываемого кода
    - По умолчанию используется Spring AOP, так что валидация будет работать только для методов, не для конструкторов
    - Класс / методы аннотируем **@Validated**
    - Создаем конфигурацию
      * На ней **@ComponentScan** с пакетом внути которого проверяемый код
      * Поставляем бин **MethodValidationPostProcessor**
  + Разработка валидаторов
    - Специльно подключать не надо, Spring автоматически ищет все реализации **ConstraintValidator** при инициализации фабрики
    - Внутри валидаторов моожно использовать **@Autowired** и прочие средства контейнера. До этого приходилось вызывать **SpringBeanAutowiringSupport.processInjectionBasedOnCurrentContext(this)** в собственной **ConstraintValidatorFactory** фабрике

## Расширения по сравнению со стандартом

* **Fail fast mode**
  + Конфигурируется через **Validation.byProvider().configure()**
  + В основном для проверки вызовов между сервисами, когда точный перечень ошибок не важен, а больше важен сам факт корректности аргументов вызова
  + Или когда объекты очень большие и проверка требует значительных ресурсов
  + Нет никакой гарантии какая из сбойных проверок в результате сработает
* Смягчение требований к наследованию при валидации методов
  + Аналогично, через методы **configure()**
  + В результате код становится не переносимым и с другим провайдером валидации будет исключение в процессе инициализации фабрики
* Программное определение ограничений
  + Получаем **ConstraintMapping** за счет **configure(),** набиваем определениями привязкой ограничений к конкретным бинам /методам, затем используем с **configure.addMapping()**
    - Для стандартных ограничений есть Def-классы. Например, для **NotNull – NotNullDef**. Для собственных можно использовать **GenericConstraintDef**, метод **param()** которого позволяет для произвольной аннотации задачть последовательно значения всех атрибутов
    - Также поддерживаются метод **valid()** вместо **@Valid** и так далее…
  + Каждый валидируемый элемент данных может быть сконфигурирован только раз. Не разрешается помещать ограничение на родителя, если ребенок не переопределяет свойство / метод
* Программное доопределение **Validation Factory** с целью добавление собственных ограничений
  + Получаем **ConstraintMapping** за счет **configure()** и аналогично предыдущему случаю
  + Реализуем **ConstraintMappingContributor**
  + Если используем дефолтную инициализацию, то прото добавляем его в **META-INF/validation.xml** в свойстве **hibernate.validator.constraint\_mapping\_contributors**
  + Неочевидное применение: можно заменить класс валидатора для стандартного ограничения. Часто используется для **@URL,** чтобы выполнять проверку только за счет **regexp** – валидатор **RegexpURLValidator**
* Композиция валидаторов
  + За счет **@ConstraintComposition(CompositionType.\*)** можно определить логику композиции – **AND / OR / ALL\_FALSE**
* Передача дополнительных данных с сообщением об ошибке
  + Можно не только за счет payload свойства аннотации ограничения, но и внутри валидатора за счет **HibernateConstraintValidatorContext.withDynamicPayload()**
  + Дополнительные значения / переменные для раскрытия шаблона сообщения об ошибке. Тоже методы **HibernateConstraintValidatorContext: addExpressionVariable() / addMessageParameter().** Переменые отличаются тем, что их можно использовать внути **${} EL** выражений, сообщения это просто фрагменты текста для **{msg}**
* Процессор аннотаций **hibernate-validator-annotation-processor**, подключаемый через **maven-compiler-plugin** и проверяющий корректность найденных ограничений на доступных классах.
  + Также его можно запустить при помощи javac и интегрировать с IDE
  + Проверяет, например
    - Соответствие аннотаций типам данных элементов, на которых они применены
    - Корректность ограничений свойств за счет set-методов
    - Корректность ограничений на методы с учетом наследования
    - Невозможность проверки static элементов
    - И до хрена всего другого…