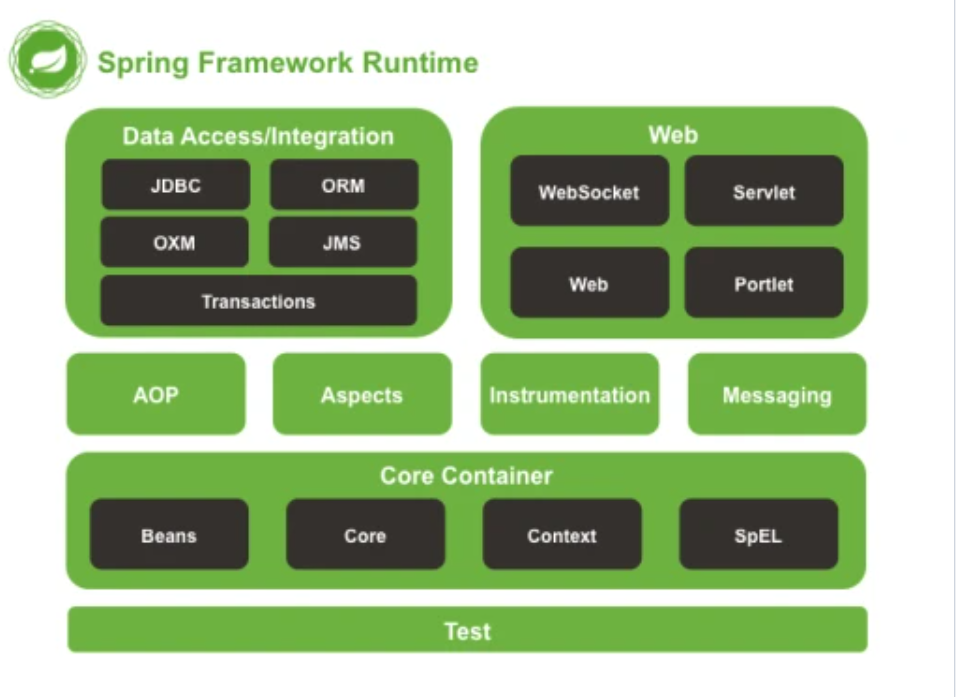
1. Spring как семейство проектов. SpringFramework - состав и назначение.

**Spring Boot** — комплексный фреймворк для создания и запуска приложений с минимальными усилиями и настройками. Но спринг — это не один какой-то конкретный фреймворк. Это скорее общее названия для целого ряда небольших фреймворков, каждый из которых выполняет какую-то свою работу.



***На изображении видно, что спринг фреймворк состоит как-бы из нескольких модулей:***

* data access;
* web;
* core;
* и других.

Как можно было догадаться, модуль data access содержит в себе средства для работы с данными (в основном, с базами данных), web — для работы в сети (в том числе и для создания веб-приложений, о которых будет позже).

**Основной контейнер (Core Container)**включает в себя Beans, Core, Context и SpEL (expression language).

Beans отвечает за BeanFactory которая является сложной реализацией паттерна Фабрика (GoF).

Модуль Core обеспечивает ключевые части фреймворка, включая свойства IoC и DI.

Context построен на основе Beans и Core и позволяет получить доступ к любому объекту, который определён в настройках. Ключевым элементом модуля Context является интерфейс ApplicationContext.

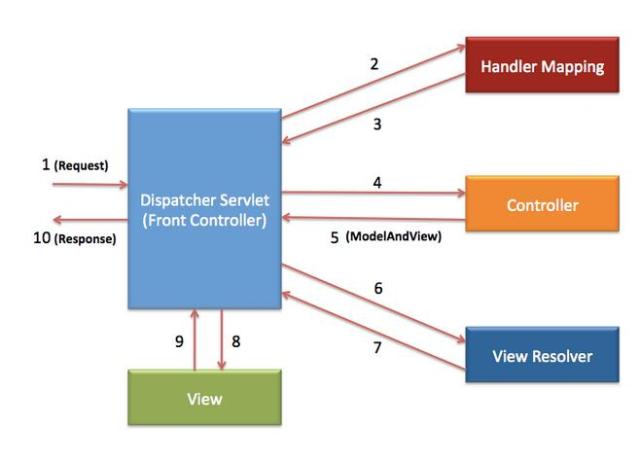
Модуль SpEL обеспечивает мощный язык выражений для манипулирования объектами во время исполнения.

1. Жизненный цикл запроса в MVC Spring. Диспетчеризация. Настройка контекста

***Жизненный цикл запроса разберем все поэтапно:***

* Фронт-контроллер DispatcherServlet получает запрос.
* DispatcherServlet передает этот запрос HandlerMapping — интерфейсу, реализуемому объектами, определяющими отображение (маппинг) между запросами для поиска подходящего контроллера.
* HandlerMapping отправляет сведения о контроллере назад в DispatcherServlet.
* DispatcherServlet вызывает контроллер, ранее идентифицированный через HandlerMapping. Выбранный контроллер обрабатывает запрос, вызвав соответствующий метод для подготовки данных и создав некую бизнес-логику (или напрямую извлекает информацию из базы данных).
* Контроллер возвращает DispatcherServlet необходимые данные модели и информацию по UI для подбора внешнего вида их отображения.
* Как только DispatcherServlet получает объект ModelAndView, он передает его ViewResolver — интерфейсу, способному находить (резолвить) вид по его имени (View Name), чтобы затем найти соответствующий вариант отображения View.
* Определившись с выбором, ViewResolver отправляет нужные сведения обратно фронт-контроллеру.
* DispatcherServlet вызывает соответствующий View (определенный ViewResolver).
* View создает отклик в виде HTML-кода и отправляет его назад.
* В конце главный контроллер предоставляет инструкции браузеру для настройки данного HTML и последующего отображения результата конечному пользователю.

1. Spring MVC архитектура. Front Controller. Создание контроллера.



Паттерн Front Controller используется для обеспечения централизованного механизма обработки запросов, так что все запросы обрабатываются одним обработчиком.

Аннотация @Controller определяет класс как контроллер Spring MVC.

1. Конфигурация Spring. WebMvcConfigurer. Аннотации:@ Controller, @Repository, @Service.

**JavaConfig** — это альтернатива привычной нам XML конфигурации, но используя не xml разметку, а синтаксис Java.( @Configuration)

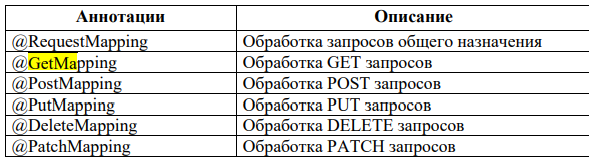
Класс конфигурации WebMvcConfigurer позволяет **избавиться от настройки xml и web.xml** и полностью перейти на java классы. Имеет множество методов, которые при желании можно переопределить.

@service – сервис (тоже самое что компонент, просто для вида)

@repository - репозиторий

1. Адресация в Контроллере. @RequestMapping, @GetMapping и др.

Аннотация @Controller определяет класс как контроллер Spring MVC., @RequestMapping указывает, что все методы в данном Контроллере относятся к одному URL-адресу



1. Понятие Inversion of Control-контейнер (IoC) и Dependency Injection (DI)

**Inversion of Control (инверсия управления)** — это некий абстрактный принцип, набор рекомендаций для написания слабо связанного кода. Суть которого в том, что каждый компонент системы должен быть как можно более изолированным от других, не полагаясь в своей работе на детали конкретной реализации других компонентов.  
**Dependency Injection (внедрение зависимостей)** — это одна из реализаций этого принципа (помимо этого есть еще [Factory Method](http://en.wikipedia.org/wiki/Factory_pattern), [Service Locator](http://en.wikipedia.org/wiki/Service_locator_pattern)).  
Реализуется инверсия управления несколькими способами, среди которых есть внедрение зависимостей (Dependency Injection, DI) и поиск зависимостей (Dependency Lookup, DL).

Внедрение зависимостей реализуется несколькими способами, среди которых можно выделить:

Внедрение через конструктор

Внедрение через set-метод

Внедрение через интерфейс

В случае с поиском зависимостей класс должен самостоятельно реализовывать логику получения зависимостей извне. Для этого он должен иметь доступ к некоему источнику зависимостей.

7. JavaBean . Правила описания и использование JavaBean. Области действия управляемых бинов, аргументы, свойства. @Autowired, @Primary, @Qualifier, @Inject.

**JavaBean** – **это** одноуровневые объекты, использующиеся для того, чтобы инкапсулировать в одном объекте код, данные или и то и другое. Компонент **JavaBean** может иметь свойства, методы и события, открытые для удаленного доступа.

Правила описания гласят: • Класс должен иметь паблик конструктор без параметров. • Свойства класса должны быть доступны через get, set и другие методы которые должны подчиняться стандартному соглашению об именах. • Класс должен быть сериализуем. • Класс должен иметь переопределенные методы equals(), hashCode() и toString().

Все компоненты вашего приложения: @Component, @Service, @Repository, @Controller и т.д.) автоматически регистрируются как Spring Beans.

Свойства класса с аннотацией @Autowired заполняются соответствующими значениями сразу после создания bean'а и перед тем, как любой из методов класса будет вызван.

Аннотация @Primary задает бин, который будет внедрен по умолчанию (при отсутствии других указаний).

Аннотация @Qualifier позволяет уточнить **имя бина**, который надо внедрить. Используется прямо перед аргументом.

@**Inject** является дополнительным для общедоступных, конструкторов без параметров, когда никакие другие конструкторы не присутствуют. Это позволяет инжекторам вызвать конструкторов по умолчанию.

Области видимости:

**Singleton** определяет один единственный бин для каждого контейнера Spring IoC (используется по умолчанию).  
**Prototype** позволяет иметь любое количество экземпляров бина.  
**request** создаётся один экземпляр бина на каждый HTTP запрос. Касается исключительно ApplicationContext.  
**session** создаётся один экземпляр бина на каждую HTTP сессию. Касается исключительно ApplicationContext.  
**global-session** создаётся один экземпляр бина на каждую глобальную HTTP сессию. Касается исключительно ApplicationContext.

1. Жизненный цикл Bean Spring. @ComponentScan.

Через следующие этапы проходит каждый отдельно взятый бин:

1. Инстанцирование объекта.

2. Установка свойств из конфигурации бина, внедрение зависимостей;

3. Нотификация aware-интерфейсов. BeanNameAware, BeanFactoryAware и другие.

4. Пре-инициализация

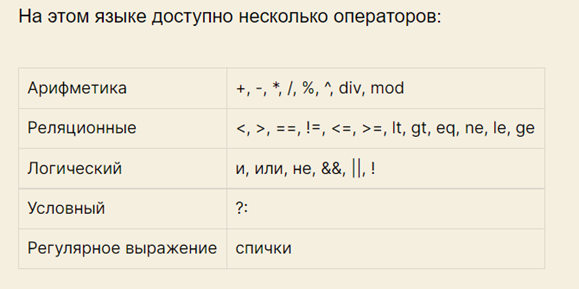
5. Инициализация. Разные способы применяются в таком порядке:

6. Пост-инициализация

В @**ComponentScan** вы указываете пакеты, которые должны сканироваться. Spring будет искать бины не только в пакетах для сканирования, но и в их подпакетах.

1. Spring Expression Language (SpEL): особенности и область использования.

Spring  [Expression Language](https://translate.googleusercontent.com/translate_c?depth=1&pto=aue&rurl=translate.google.ru&sl=en&sp=nmt4&tl=ru&u=http://en.wikipedia.org/wiki/Unified_Expression_Language&usg=ALkJrhgJT-q4G7ZGjSza3o5L9xhOmpOycg)  (сокращенно SpEL) — это мощный язык выражений, который поддерживает запросы и манипулирование  [графом объектов](https://translate.googleusercontent.com/translate_c?depth=1&pto=aue&rurl=translate.google.ru&sl=en&sp=nmt4&tl=ru&u=http://en.wikipedia.org/wiki/Object_graph&usg=ALkJrhhzWvJ_c45W1SsFGPLlIQK1u5klXQ)  во время выполнения, а также оценку логических и  [математических выражений](https://translate.googleusercontent.com/translate_c?depth=1&pto=aue&rurl=translate.google.ru&sl=en&sp=nmt4&tl=ru&u=http://en.wikipedia.org/wiki/Expression_%2528mathematics%2529&usg=ALkJrhjyS-8seza46qcNy3lrFGmSQfNaIQ)  и многие другие функции. SpEL может использоваться независимо, независимо от того, использует ли ваша [среда приложения](https://translate.googleusercontent.com/translate_c?depth=1&pto=aue&rurl=translate.google.ru&sl=en&sp=nmt4&tl=ru&u=http://en.wikipedia.org/wiki/Application_framework&usg=ALkJrhinSHhtRS5F1DgrDJGgu4gI7-SSCw) Spring   или нет.



10.Spring FrameworkValidation. Интерфейс Validator.

Идея Bean Validation в том, чтобы определять такие правила, как «Это поле не может быть null» или «Это число должно находиться в заданном диапазоне» с помощью аннотаций. Это гораздо проще, чем постоянно писать условные операторы проверок.

Валидировать можно сущности, контроллеры(тело запроса, переменные пути, параметры запроса) и сервисные слои.

В Spring есть так же свой интерфейс Validator и именно его имплементация выполняет проверку данных. Это уже не декларативный подход, но в нем есть своя гибкость и расширяемость. Чтобы его использовать, нужно переопределить методы validate и supports.

11.Правила валидации и ограничения.

Правила валидации в Bean Validation задаются при помощи ограничений (constraints), аннотаций, расположенных в пакете **javax.validation.constraints**. Ограничения могут применяться к свойствам классов, аргументам методов и конструкторов, их возвращаемым значениям, а так же к типам обобщений.

Числовые, даты и времени, строчные, булевые.

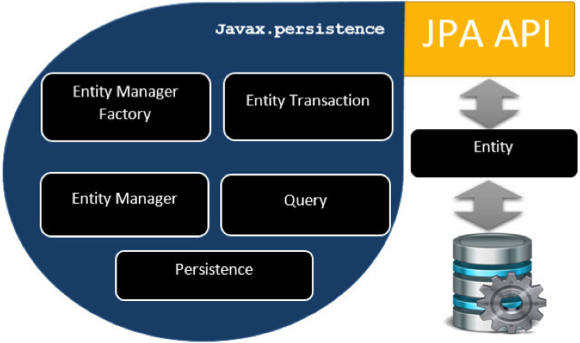
12.Создание пользовательского валидатора.

Последовательность действий по пунктам:

1. Создадим **аннотации** для валидации полей (проверка на пустоту строки и на регулярное выражение), а также аннотацию-триггер, которая будет запускать валидацию на уровне параметров метода.
2. Распишем **классы** ошибок.
3. Спроектируем интерфейсы и реализуем классы с логикой для валидации объектов, используя **Java Reflection API**.
4. Приведем в действие написанный функционал с помощью **Spring AOP(Aspect)**.
5. Обработаем **ошибки** валидации и отдадим клиенту соответствующее **DTO**.

13.Понятие ORM. Архитектура JPA: EntityManager, Persistence, …

ORM (Object-Relational Mapping или объектно-реляционное отображение) — технология для отображения объектов в структуры реляционных баз данных, чтобы представить java-объект в виде строки таблицы. Таблица обычно соответствуют классу, строки таблицы - экземплярам этого класса, колонки таблицы - столбцам.





14.Требования в Entity. Жизненный цикл Entity. Типы связей.

1) Entity класс должен быть отмечен аннотацией Entity или описан в XML файле конфигурации JPA,  
2) Entity класс должен содержать public или protected конструктор  
3) Entity класс должен быть классом верхнего уровня (top-level class),  
4) Entity класс не может быть enum или интерфейсом,  
5) Entity класс не может быть финальным классом (final class),  
6) Entity класс не может содержать финальные поля или методы, если они участвуют в маппинге (persistent final methods or persistent final instance variables),

У Entity объекта существует **четыре статуса жизненного цикла**: new, managed, detached, или removed. Их описание  
1) new — объект создан, но при этом ещё не имеет сгенерированных первичных ключей и пока ещё не сохранен в базе данных,  
2) managed — объект создан, управляется JPA, имеет сгенерированные первичные ключи,  
3) detached — объект был создан, но не управляется (или больше не управляется) JPA,  
4) removed — объект создан, управляется JPA, но будет удален после commit'a транзакции.

**OneToOne, ManyToMany, OneToMany, ManyToOne.**

15.Spring Data Annotations.

|  |
| --- |
| @Entity |
| @Table |  |
| @Column |  |
| @Id |  |
| @GeneratedValue |  |

@OrderBy

@Transient + связи, что выше

16. JPA механизм обратных вызовов (@Pre… @Post…). Запросы

Всего, как следут из кода выше, есть 4 типа *callbacks,*три из которых вызываются до и после изменения.

* @PrePersist — вызывается как только инициирован вызов persist() и исполняется перед остальными действиями.
* @PostPersist  — вызывается когда сохранение в базу завершено и оператор INSERT выполнен.
* @PreUpdate  — вызывается перед сохранением изменений в сущности в базу.
* @PostUpdate — вызывается, когда данные сущности в базе обновлены и оператор UPDATE  выполнен.
* @PreRemove — вызывается как только инициирован вызов remove() и исполняется перед остальными действиями.
* @PostRemove — вызывается, когда операция удаления из базы завершено и оператор DELETE выполнен.
* @PostLoad — вызывается после загрузки данных сущности из БД.

17.Паттерн Service, Repository, Controller.

* **Controller** — отвечает за взаимодействие с пользователем посредством IO (ввод-вывод).
* **Service** — отвечает за бизнес логику и ее переиспользование между компонентами.
* **Repository** — отвечает за получение данных из внешних источников, такие как база данных, api, локальное хранилище и пр.

18.Аспектно-ориентированное программирование. Понятие аспекта, совета, срез и точки соединения, вплетение.

Итак, **АОП** — **аспектно-ориентированное программирование** — это парадигма, направленная на повышение модульности различных частей приложения за счет разделения сквозных задач. Для этого к уже существующему коду добавляется дополнительного поведение, без изменений в изначальном коде.

Ключевой единицей в ООП является “объект”, а ключевой единицей в АОП – “аспект”. В качестве примера “аспекта” можно привести безопасность, кэширование, логирование и т.д.

**Аспект (aspect)** — Это модуль, который имеет набор программных интерфейсов, обеспечивающих сквозные требования. К примеру, модуль логирования будет вызывать АОП аспект для логирования. В зависимости от требований, приложение может иметь любое количество аспектов.

**Точка соединения (join point)** — Это такая точка в приложении, где мы можем подключить аспект. Другими словами, это место, где начинаются определённые действия модуля АОП в Spring.

**Совет (advice)** — Это фактическое действие, которое должно быть предпринято до и/или после выполнения метода. Это конкретный код, который вызывается во время выполнения программы.

**Срез (pointcut)** — Срезом называется несколько объединённых точек (join points), в котором должен быть выполнен совет.

**Плетение (weaving)** — Это процесс связывания аспектов с другими объектами приложения для создания совета. Может быть вызван во время компиляции, загрузки или выполнения приложения.

19.Архитектура АОП в Spring. ProxyFactory. AOP frameworks

Как уже было отмечено, реализация аспектно-ориентированного программирования в Spring основана на использовании объектов-посредников (proxy). Создание посредников возможно программным образом, используя класс [ProxyFactory](http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/aop/framework/ProxyFactory.html). Он представляет собой реализацию простейшего профайлера, который можно использовать для того, чтобы узнать, сколько времени выполняется код метода.

20.Конфигурации Spring AOP. Пример определения аспекта. Аннотации и правила настройки @Pointcut @Before @AfterReturning @Around и др.

Определить аспект можно с помощью xml или @AspectJ(используя аннотации).

Возможные **виды советов**:

1. **Перед (Before)** — советы данного типа запускаются перед выполнением целевых методов — точек соединения.
2. **После (After)** — советы, которые выполняются после завершения выполнения методов — точек соединения, как в обычных случаях, так и при бросании исключения.
3. **После возврата (After Returning)** — данные советы выполняются только в том случае, когда целевой метод отрабатывает нормально, без ошибок.
4. **После бросания (After Throwing)** — данный вид советов предназначен для тех случаев, когда метод, то есть точка соединения выдает исключение. Мы можем использовать этот совет для некой обработки неудачного выполнения (к примеру, для отката всей транзакции или логирования с необходимым уровнем трассировки).
5. **Вокруг (Around)** — пожалуй, один из самых важных видов советов, который окружает метод, то есть — точку соединения, с помощью которого мы можем, к примеру, выбрать, выполнять данный метод точки соединения или нет.

21. Понятие SPA и MPA приложений.

SPA — это одностраничное приложение, содержащее HTML-страницу, которая динамически (без полной перезагрузки) обновляется в ответ на действия пользователя.

MPA или Multi Page Applications по принципу работы полностью противоположны SPA. MPA — это многостраничные приложения, работающие, как привычные нам веб-сайты. Они отправляют запрос на сервер и полностью обновляют страницу, когда с ней совершается какое-либо действие (переход на другую страницу, внесение и изменение данных).

22.Entity – DTO конвертация. Модель Mapper (конфигурация, правила)

Data Transfer Object

В ходе разработки любого приложения программист сталкивается с необходимостью работать с моделями различных объектов, созданных для разных целей. И соответственно, с необходимостью конвертировать их между собой. Если ваш проект на начальном этапе развития, то можно, конечно, использовать рукописные конверторы. Но рано или поздно проект станет больше, и вы столкнётесь с необходимостью использовать уже готовое решение для конвертации моделей.

Одним из таких решений является МodelMapper. Его очень просто использовать в самом начале проекта без особых знаний. Попытаюсь разобрать основные моменты использования фреймворка.

Для того, чтобы начать маппить entity в dto нам достаточно написать вот такой простой конвертор:

23.Понятие REST.Требования к RESP архитектуре.

**Representational State Transfer** — это архитектурный стиль взаимодействия компонентов распределённого приложения в сети. Архитектурный стиль – это набор согласованных ограничений и принципов проектирования, позволяющий добиться определённых свойств системы.

Но зачем нам REST? Зачем нам этот стиль? Что нам даст применение принципов REST?

Если мы обратимся опять же к первоисточнику — к работе Филдинга, то мы выясним, что назначение REST в том, чтобы придать проектируемой системе такие свойства как:

* Производительность,
* Масштабируемость,
* Гибкость к изменениям,
* Отказоустойчивость,
* Простота поддержки.

Это наиболее ценные свойства, с которыми встречается, например, аналитик при проектировании систем. В действительности их намного больше. Если внимательно посмотреть на эти свойства, то мы увидим ни что иное, как нефункциональные требования к системе, которых мы на своих проектах стремимся достичь.

24.HTTP-методы REST.

В итоге, совместив имеющуюся спецификацию HTTP и REST-подход наконец-то обретают смысл различные HTTP-методы. GET — возвращает ресурс, POST — создает новый, PUT — обновляет существующий, DELETE — удаляет.

25.REST контроллер. Отображения запросов. Параметры запроса и ответа.

Поясним аннотации: @RestController — говорит спрингу, что данный класс является REST контроллером. Т.е. в данном классе будет реализована логика обработки клиентских запросов

@Autowired — говорит спрингу, что в этом месте необходимо внедрить зависимость. В конструктор мы передаем интерфейс ClientService. Реализацию данного сервиса мы пометили аннотацией @Service ранее, и теперь спринг сможет передать экземпляр этой реализации в конструктор контроллера.

Метод возвращает ResponseEntity<?>. ResponseEntity — специальный класс для возврата ответов. С помощью него мы сможем в дальнейшем вернуть клиенту HTTP статус код.

26.Отображение кодов ответа HTTP. Сопоставленные и не сопоставленные запросы.

1. Информационные 100 - 199
2. Успешные 200 - 299
3. Перенаправления 300 - 399
4. Клиентские ошибки 400 - 499
5. Серверные ошибки 500 - 599

мера: сопоставление HTTP-запроса с методом с использованием некоторых основных критериев.

### @RequestMapping — по пути

### @RequestMapping — метод HTTP

**@RequestMapping С атрибутом headers**

Сопоставление можно сузить еще больше, указав заголовок для запроса

**@RequestMapping Потребляет и производит**

Особого внимания заслуживает отображение **типов носителей, создаваемых контроллером** методом.

@RequestMapping(value = "/ex/foos", method = RequestMethod.GET)

27.Настраиваемые исключения при ошибках запроса REST. Форматы данных.

мы можем пойти на меньшее вмешательство и использовать реализацию [**DefaultErrorAttributes,**](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/ResponseStatus.html) либо добавив в исключения аннотацию [**@ResponseStatus,**](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/ResponseStatus.html)либо позволив всем исключениям расширять [**ResponseStatusException**](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/server/ResponseStatusException.html) . Оба способа позволяют настроить статус ответа и значение **сообщения**. К сожалению, большинство исключений, создаваемых на уровне инфраструктуры, предоставляются фреймворком и не могут быть настроены, поэтому нам нужно другое решение. Одна из возможностей для аннотированных контроллеров - использовать [**@ExceptionHandler**](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/ExceptionHandler.html) для отдельных исключений. Тогда мы могли бы создать ответ с нуля, но это пропустило бы обработку исключений по умолчанию, и мы хотели бы иметь одинаковую обработку для каждого исключения.

 С другой стороны, обработка исключений по умолчанию довольно сложна, и вы можете начать вмешиваться на многих уровнях, сверху вниз:

* Непосредственно в контроллере с помощью **try/catch** (MVC) или **onErrorResume()** (Webflux). Я не рекомендую это в большинстве случаев, потому что сквозная проблема, такая как обработка исключений, должна быть определена глобально, чтобы гарантировать согласованное поведение.
* Перехватить исключения в **функциях @ExceptionHandler** . Создайте свои собственные ответы с помощью @ExceptionHandler (Throwable.class) для случая по умолчанию.
* Или **повторно генерируйте исключения** , аннотируйте их с помощью **@ResponseStatus**или **расширяйте ResponseStatusException,** чтобы настроить ответ для определенных случаев.

28.Тестирование REST. POSTMAN.

29.Понятие HATEOAS REST сервиса (swagger)

**HATEOAS** (Hypermedia as the Engine of Application **State**  (Гипермедиа как двигатель состояния приложения)) — архитектурные ограничения для **REST**-приложений. С помощью **HATEOAS** клиент взаимодействует с сетевым приложением, сервер которого обеспечивает динамический доступ через гипермедиа. **REST**-клиенту не требуется заранее знать, как взаимодействовать с приложением или сервером за пределом гипермедиа.

30.Документирование REST на основе Open API. Аннотации.

@OpenAPIDefinition

<dependency>  
 <groupId>org.springdoc</groupId>  
 <artifactId>springdoc-openapi-ui</artifactId>  
 <version>1.6.4</version>  
</dependency>  
<dependency>  
 <groupId>io.springfox</groupId>  
 <artifactId>springfox-swagger2</artifactId>  
 <version>2.6.1</version>  
 <scope>compile</scope>  
</dependency>  
  
<dependency>  
 <groupId>io.springfox</groupId>  
 <artifactId>springfox-swagger-ui</artifactId>  
 <version>3.0.0</version>  
 <scope>compile</scope>  
</dependency>

31.Spring Security Framework. Request Security. Servlet filters. Security setting.

#### **Аутентификация**

(1) Пользователю будет предложено войти в систему предоставив имя (логин или email) и пароль. Имя пользователя и пароль объединяются в экземпляр класса UsernamePasswordAuthenticationToken(экземпляр интерфейса Authentication) после чего он передается экземпляру AuthenticationManager для проверки.  
  
(2) В случае если пароль не соответствует имени пользователя будет выброшено исключение BadCredentialsException с сообщением “Bad Credentials”.  
  
(3) Если аутентификация прошла успешно возвращает полностью заполненный экземпляр Authentication.  
  
(4) Для пользователя устанавливается контекст безопасности путем вызова метода SecurityContextHolder.getContext().setAuthentication(…), куда передается объект который вернул AuthenticationManager.

Подключение поддержки Security для SpringFramework приложения:

<**dependency**>

<**groupId**>org.springframework.security</**groupId**>

<**artifactId**>spring-security-config</**artifactId**>

<**version**>${spring.version}</**version**>

</**dependency**>

@EnableWebSecurity  
@RequiredArgsConstructor  
@EnableGlobalMethodSecurity(prePostEnabled = true)  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
 private final CustomUserDetailsService userDetailsService;  
  
 @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 http  
 .authorizeRequests()  
 .antMatchers("/", "/product/\*\*", "/images/\*\*", "/registration", "/user/\*\*", "/static/\*\*","/swagger-ui.html")  
 .permitAll()  
 .anyRequest().authenticated()  
 .and()  
 .formLogin()  
 .loginPage("/login")  
 .permitAll()  
 .and()  
 .logout()  
 .permitAll();  
 }  
  
 @Override  
 protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.userDetailsService(userDetailsService)  
 .passwordEncoder(passwordEncoder());  
 }  
  
 @Bean  
 public PasswordEncoder passwordEncoder() {  
 return new BCryptPasswordEncoder(8);  
 }  
}

32.Authentication and authorization.

В Java есть несколько технологий, которые обеспечивают безопасность:

* "**Java SE Platform Security Architecture**", подробнее про которую можно прочитать в Guide от Oracle: "[JavaTM SE Platform Security Architecture](https://docs.oracle.com/javase/8/docs/technotes/guides/security/spec/security-specTOC.fm.html)". Эта архитектура описывает то, как необходимо защищать наши Java приложения в Java SE среде выполнения. Но это не является темой нашего сегодняшнего разговора.
* "**Java Cryptography Architecture**" — расширение (Java Extension), которое описывает шифрование данных. Подробнее про это расширение можно прочитать на JavaRush в обзоре "[Java Cryptography Architecture : Первое знакомство](https://javarush.ru/groups/posts/2277-java-cryptography-architecture--pervoe-znakomstvo)" или в Guide от Oracle: "[Java Cryptography Architecture (JCA) Reference Guide](https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html)".

## JAAS "Java Authentication and Authorization Service (JAAS)".

**JAAS** — это расширение Java SE и оно описано в документе "[Java Authentication and Authorization Service (JAAS) Reference Guide](https://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/JAASRefGuide.html)". Как следует из названия технологии, JAAS описывает то, как нужно выполнять аутентификацию и авторизацию:

* "**Аутентификация**": в переводе с греческого "authentikos" означает "реальный, подлинный". То есть аутентификация — это проверка подлинности. Что тот, кто проходит аутентификацию, действительно тот, за кого себя выдаёт.
* "**Авторизация**" : в переводе с английского означает "разрешение". То есть авторизация — это контроль доступа, выполняемый после успешного прохождения аутентификации.

33.Interception of requests. Перехват запросов

Написать три метода: preHandle(), postHandle(), afterCompletion(), которые вызываются перед обработкой запроса актуальным классом, после этой обработки, а так же после окончания запроса соответственно.

34.Security support in Spring Security at the method level.

В Java есть несколько технологий, которые обеспечивают безопасность:

* "**Java SE Platform Security Architecture**", подробнее про которую можно прочитать в Guide от Oracle: "[JavaTM SE Platform Security Architecture](https://docs.oracle.com/javase/8/docs/technotes/guides/security/spec/security-specTOC.fm.html)". Эта архитектура описывает то, как необходимо защищать наши Java приложения в Java SE среде выполнения. Но это не является темой нашего сегодняшнего разговора.
* "**Java Cryptography Architecture**" — расширение (Java Extension), которое описывает шифрование данных. Подробнее про это расширение можно прочитать на JavaRush в обзоре "[Java Cryptography Architecture : Первое знакомство](https://javarush.ru/groups/posts/2277-java-cryptography-architecture--pervoe-znakomstvo)" или в Guide от Oracle: "[Java Cryptography Architecture (JCA) Reference Guide](https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html)".

35.Configure Spring Security for OAuth 2.0 Login and Resource Server

## **OAuth2**

OAuth2 — это протокол авторизации, который позволяет клиенту (третьей стороне) получить доступ к ресурсам вашего приложения. Для построения OAuth2-приложения нам необходимо знать Grant Type (код авторизации), Client ID и Client secret.

Полное имя сервера ресурсов - **OAuth2 Resource Server**, который на самом деле является частью протокола OAuth 2.0 и обычно реализуется с помощью веб-токена Json (на самом деле существует еще один, называемый Непрозрачными токенами, который также можно использовать). Сервер авторизации OAuth 2.0 отправляет клиенту **веб-токен Json**, который используется для проверки того, что у клиента есть разрешение на доступ к ресурсу.

Когда отдельное приложение преобразуется в микросервис, служба авторизации на самом деле лучше централизована, унифицирует управление авторизацией аутентификации пользователя и выдает токен клиенту. каждая конкретная служба также является ресурсным сервером, нам нужно только абстрагироваться от того, каким может быть интерфейс управления доступом. Здесь следует принципу: **обычная инкапсуляция в библиотеку классов (jar), персонализированная абстракция для настройки.**Общая блок-схема выглядит следующим образом.

36.Spring Cloud

Cоздав приложение и освоив контейнеризацию, вы увидите как легко, одной цифрой в конфиг-файле, настраивается запуск контейнеров во множестве экземпляров, для отказоустойчивости и масштабирования.

Но возникает проблема, – потребитель сервисов не умеет работать с множеством экземпляров. Для связи прикладных сервисов в целостное приложение, необходимы инфраструктурные сервисы.

Подсистема Spring Cloud состоит из:

1. [Зависимостей, адаптирующих](https://habr.com/ru/post/674882/#%D0%97%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B8%D0%92%D1%81%D1%82%D1%80%D0%B0%D0%B8%D0%B2%D0%B0%D0%B5%D0%BC%D1%8B%D0%B5) Spring Boot приложения к облачной среде.
2. [Инфраструктурных сервисов](https://habr.com/ru/post/674882/#%D0%98%D0%BD%D1%84%D1%80%D0%B0%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D1%8B%D0%B5%D0%A1%D0%B5%D1%80%D0%B2%D0%B8%D1%81%D1%8B%D0%9F%D0%B5%D1%80%D0%B5%D0%BD%D0%BE%D1%81%D0%B8%D0%BC%D1%8B%D0%B5), в виде готовых Spring-приложений, переносимых​ между облачными платформами​.
3. [Клиентов сетей Service Mesh](https://habr.com/ru/post/674882/#ServiceMesh), реализующих инфраструктурные сервисы.
4. [Клиентов облачных платформ](https://habr.com/ru/post/674882/#%D0%98%D0%BD%D1%84%D1%80%D0%B0%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D1%8B%D0%B5%D0%A1%D0%B5%D1%80%D0%B2%D0%B8%D1%81%D1%8B%D0%98%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5), реализующих инфраструктурные и коммерческие сервисы.
5. [Микро-фреймворков](https://habr.com/ru/post/674882/#%D0%9C%D0%B8%D0%BA%D1%80%D0%BE%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA%D0%B8) для разработки микросервисов и их оркестровки.