Доп.вопросы:

1. Аннотация @Autowire на поля/конструктор/методы. Почему лучше на конструктор. Про существующие классы.
2. Репозиторий и его исключения. Какие и почему нужны.
3. Параметры и хиддеры у http запросов
4. put идемпотент и нет - спорно
5. Минусы Spring Boot - про вес и скорость работы приложений

Содержание (подготовка к собесам):

1. Что такое инверсия контроля (IoC) и внедрение зависимостей (DI)? Как эти

принципы реализованы в Spring?

2. Что такое IoC контейнер?

3. Что такое Bean в спринге?

4. Расскажите про аннотацию @Bean?

5. Расскажите про аннотацию @Component?

6. Чем отличаются аннотации @Bean и @Component?

7. Расскажите про аннотации @Service и @Repository. Чем они отличаются?

8. Расскажите про аннотацию @Autowired

9. Расскажите про аннотацию @Resource10. Расскажите про аннотацию @Inject

11. Расскажите про аннотацию @ Lookup

12. Можно ли вставить бин в статическое поле? Почему?

13. Расскажите про аннотации @Primary и @Qualifier

14. Как заинжектить примитив?

15. Как заинжектить коллекцию?

16. Расскажите про аннотацию @ Conditional

17. Расскажите

про аннотацию @Profile

18. Расскажите про ApplicationContext и BeanFactory, чем отличаются? В каких

случаях что стоит использовать?

19. Расскажите про жизненный цикл бина, аннотации @PostConstruct и

@PreDestroy()

20. Расскажите про скоупы бинов? Какой скоуп используется по умолчанию? Что

изменилось в пятом спринге?

21. Расскажите про аннотацию @ComponentScan

22. Как спринг работает с транзакциями? Расскажите про аннотацию

@Transactional.

23. Расскажите про аннотации @Controller и @RestController. Чем они отличаются?

Как вернуть ответ со своим статусом (например 213)?

24. Что такое ViewResolver?

25. Чем отличаются Model, ModelMap и ModelAndView?

26. Расскажите про паттерн MVC, как он реализован в Spring?

27. Расскажите про паттерн Front Controller, как он реализован в Spring?

28. Что такое АОП? Как реализовано в спринге?

29. В чем разница между Filters, Listeners and Interceptors?

30. Можно ли передать в запросе один и тот же параметр несколько раз? Как?

31. Как работает Spring Security? Как сконфигурировать? Какие интерфейсы

используются?

32. Что такое SpringBoot? Какие у него преимущества? Как конфигурируется?

Подробно.

33. Расскажите про нововведения Spring 5.

1 часть

1. Что такое бин?

В Spring объекты, образующие основу приложения и управляемые контейнером Spring

IoC, называются бинами.

Бин — это объект, который создается, собирается и управляется контейнером Spring IoC.

Бины и их зависимости отражаются в метаданных конфигурации, используемых контейнером.

@Bean - Это аннотация Spring Framework, она используется над методом для указания

того, что данный метод создает, настраивает и инициализирует новый объект, управляемый

Spring IoC контейнером. Такие методы можно использовать как в классах с аннотацией

@Configuration, так и в классах с аннотацией @Component (или её наследниках).

Позволяет дополнительно определить у бина:

❖ name - имя (уникальный идентификатор) бина;

❖ initMethod - имя метода для вызова во время инициализации бина;

❖ destroyMethod - имя метода для вызова во время удаления бина из контекста;

❖ autowireCandidate - является ли этот бин кандидатом на автоматическое внедрение в

другой бин.

Классы, аннотированные @Configuration, проксируются через CGLIB.

Классы @Component или обычные классы не проксируются и не перехватывают вызовы методов с

аннотациями @Bean, что означает, что вызовы не будут маршрутизироваться через контейнер

и каждый раз будет возвращаться новый экземпляр бина.

Также методы бинов, вызывая друг друга в таких классах, не будут создавать бины, а

будет просто выполняться код метода, ведь в данном случае они отработают не через прокси.

CGLIB (Code Generation Library) - Это библиотека инструментария байтов, используемая

во многих средах Java, таких как Hibernate или Spring. Инструментарий байт-кода позволяет

манипулировать или создавать классы после фазы компиляции программы.

**Имена бинов**

Имя бина, которое в контейнере является одновременно и его уникальным

идентификатором, по умолчанию соответствует имени метода, аннотированного @Bean. Но

если требуется указать иное имя, то можно использовать атрибут name, который принимает

String. Однако, атрибут name также может принимать массив String, что позволяет использовать

несколько имен. Первый элемент массива будет являться именем и уникальным идентификатором бина, а остальные будут его псевдонимами.

**Можно ли вставить бин в статическое поле? Почему?**

Spring не позволяет внедрять бины напрямую в статические поля. Чтобы исправить это, создайте нестатический сеттер-метод.

**Как заинжектить примитив?**

Внедрить в поле примитив можно с помощью аннотации @Value на уровне параметров

поля или конструктора/метода.

Нам понадобится файл свойств (\*.properties), чтобы определить значения, которые мы

хотим внедрить аннотацией @Value. Сначала в нашем классе конфигурации нам нужно указать

аннотацию @PropertySource с именем файла свойств. Если из файла не подтянутся значения по тем или иным причинам, то можно указать значения, которые будут внедрены по умолчанию.

Кроме того, для внедрения значений мы можем использовать язык SpEL (Spring

Expression Language).

Мы также можем использовать аннотацию @Value для добавления свойств в Map. Для

начала нам нужно определить свойство в формате {key: ‘value '} в нашем файле свойств.

**Как заинжектить коллекцию?**

Array Injection

Мы можем вставлять массивы примитивов и ссылочных типов. Со всеми массивами и

коллекциями мы можем использовать внедрение через конструкторы, сеттеры или поля.

Также мы можем собрать все бины одного типа, находящиеся в контейнере, и внедрить

их в коллекцию или массив.

Использование @Qualifier

Методы класса JavaConfig (те, которые аннотированы @Bean) могут быть объявлены с

определенным квалифицирующим типом, используя @Qualifier. Мы использовали параметр

'name' у аннотации @Bean, чтобы указать конкретный классификатор для бина. Но элемент

'name', на самом деле, является не столько именем, сколько идентификатором бина, который

должен быть уникальным, потому что все бины хранятся в контейнере в Map.

В случае с коллекцией мы хотим, чтобы несколько бинов имели одно и то же имя

квалификатора, чтобы их можно было внедрить в одну коллекцию с одним и тем же квалификатором. В этом случае мы должны использовать аннотацию @Qualifier вместе с @Bean

вместо элемента name.

Определение бина содержит метаданные конфигурации, которые необходимы управляющему контейнеру для получения следующей информации: как создать бин, информацию о жизненном цикле бина, зависимости бина.

В Spring Framework существуют такие свойства, определяющие бины:

* class - Этот атрибут является обязательным и указывает конкретный класс Java-приложения, который будет использоваться для создания бина.
* name - Уникальный идентификатор бина. В случае конфигурации с помощью xml-файла, вы можете использовать свойство “id” и/или “name” для идентификации бина.
* scope - Это свойство определяет область видимости создаваемых объектов.
* constructor-arg - определяет конструктор, использующийся для внедрения зависимости.
* properties - Определяет свойства внедрения зависимости.
* initialization method - Здесь определяется метод инициализации бина
* destruction method - Метод уничтожения бина, который будет использоваться при уничтожении контейнера, содержащего бин.
* autowiring mode - Определяет режим автоматического связывания при внедрении зависимости.
* lazy-initialization mode - Режим ленивой инициализации даёт контейнеру IoC команду создавать экземпляр бина при первом запросе, а не при запуске приложения.

2. Виды бинов? Расскажите про scope бинов. **Что изменилось в пятом Spring?**

Spring Framework поддерживает шесть scopes:

1. singleton
2. prototype
3. request
4. session
5. application
6. websocket
7. не активированный по умолчанию Custom thread scope.

С 3 по 6 доступны только в веб-приложениях. Мы также можем создать свой

собственный scope.

**Singleton**

Является дефолтным scope. В контейнере будет создан только один бин, и все запросы

на него будут возвращать один и тот же бин. Этот бин хранится в контейнере, и все запросы и

ссылки на этот бин возвращают закэшированный экземпляр.

Singleton Scope потокобезопасный.

Singleton-бины обычно создаются сразу при сканировании.

Singleton bean можно внедрять в любой другой бин. В сам singleton можно внедрить только *prototype* или *singleton*.

**Prototype**

Scope “prototype” приводит к созданию нового бина каждый раз, когда он

запрашивается.

Для бинов со scope “prototype” Spring не вызывает метод destroy(). Spring не берет на

себя контроль полного жизненного цикла бина со scope @prototype”. Spring не хранит такие

бины в своём контексте ( контейнере), а отдаёт их клиенту и больше о них не заботится (в

отличие от синглтон-бинов).

Не потокобезопасный, т.к. он не гарантирует что один и тот же экземпляр будет вызываться только в 1 потоке.

Prototype-бины обычно создаются только после запроса.

Prototype может быть зависимостью для любого бина.

Внедрять можно только singleton или prototype.

Если внедрять prototype, то для каждого singleton будет создан уникальный prototype.

**Request**

Контейнер создает новый экземпляр для каждого HTTP-запроса. Таким образом, если

сервер в настоящее время обрабатывает 50 запросов, тогда контейнер может иметь не более

50 бинов, по одному для каждого HTTP-запроса. Любое изменение состояния одного

экземпляра не будет видимо другим экземплярам. Эти экземпляры уничтожаются, как только

HTTP-запрос завершен.

**Session**

Бин создается в одном экземпляре для одной HTTP-сессии. Таким образом, если сервер

имеет 20 активных сессий, тогда контейнер может иметь не более 20 бинов, по одному для

каждой сессии. Все HTTP-запросы в пределах времени жизни одной сессии будут иметь доступ

к одному и тому же бину.

**Application**

Бин со scope “application” создается в одном экземпляре для жизненного цикла

ServletContext. Виден как атрибут ServletContext. Синглтон - в одном экземпляре для

ApplicationContext.

**Websocket**

Бин со scope “websocket” создается в одном экземпляре для определенного сеанса

WebSocket. Один и тот же бин возвращается всякий раз, когда к нему обращаются в течение

всего сеанса WebSocket.

**Custom thread scope**

Spring по умолчанию не предоставляет thread scope, но его можно активировать.

Каждый запрос на бин в рамках одного потока будет возвращать один и тот же бин.

**В пятой версии Spring Framework не стало Global session scope.**

3. Чем бин отличается от POJO-класса?

По словам Мартина Фаулера, POJO - это объект, который инкапсулирует бизнес-логику, а Bean (за исключением определения, уже указанного в других ответах) - это немного больше, чем контейнер для хранения данных, а операции, доступные на объекте, просто устанавливаются и получить данные.

Класс pojo - это обычный класс без каких-либо специальностей, класс, полностью не связанный с технологией /framework. Класс не реализуется из технологии/структуры и не распространяется на технологию/инфраструктуру api, этот класс называется классом pojo.

POJO может реализовывать интерфейсы и расширять классы, но суперкласс или интерфейс не должны быть технологией/каркасом.

**JavaBean - это объект Java, который удовлетворяет определенным соглашениям о программном обеспечении:**

- поля должны быть приватными

- у полей должны быть геттеры/сеттеры

- должен быть публичный конструктор без аргументов

- является сериализуемым классом.

- не является динамическим Proxy-объектом.

4. Inversion of Control. Как Spring реализует этот принцип?

***IoC (Inversion of Control)*** — инверсия управления. Это принцип в разработке программного обеспечения, при котором **управление объектами или частями программы передается контейнеру или фреймворку**. Не вы управляете процессом выполнения кода/программы, а фреймворк это делает за вас. Вы передали ему управление (инверсия управления).

В отличие от традиционного программирования, в котором наш пользовательский код

обращается напрямую к библиотекам, IoC позволяет фреймворку контролировать ход

программы и обращаться к нашему коду, когда это необходимо. Для этого, фреймворки

используют абстракции со встроенным дополнительным поведением. Если мы хотим добавить

наше собственное поведение, нам нужно расширить классы фреймворка или подключить наши

собственные классы.

Преимущества этой архитектуры:

❖ отделение выполнения задачи от ее реализации;

❖ легкое переключение между различными реализациями;

❖ большая модульность программы;

❖ более легкое тестирование программы путем изоляции компонента или проверки его

зависимостей и обеспечения взаимодействия компонентов через контракты.

Центральной частью Spring является подход Inversion of Control, который позволяет конфигурировать и управлять объектами Java с помощью рефлексии.

Вместо ручного внедрения зависимостей, фреймворк забирает ответственность за это посредством контейнера.

Контейнер отвечает за управление жизненным циклом объекта: создание объектов, вызов методов инициализации и конфигурирование объектов путём связывания их между собой.

Объекты, создаваемые контейнером, также называются управляемыми объектами (beans).

Обычно, конфигурирование контейнера, осуществляется путём внедрения аннотаций (начиная с 5 версии J2SE), но также, есть возможность, по старинке, загрузить XML-файлы, содержащие определение bean’ов и предоставляющие информацию, необходимую для создания bean’ов.

Объекты могут быть получены одним из двух способов:

1. Dependency Lookup (поиск зависимости) — шаблон проектирования, в котором вызывающий объект запрашивает у объекта-контейнера экземпляр объекта с определённым именем или определённого типа.
2. Dependency Injection (внедрение зависимости) — шаблон проектирования, в котором контейнер передает экземпляры объектов по их имени другим объектам с помощью конструктора, свойства или фабричного метода.

Инверсия управления может быть достигнута с помощью различных механизмов, таких

как: шаблон проектирования “Стратегия”, шаблон “Локатор служб”, шаблон “Фабрика” и

внедрение зависимостей (DI).

***Dependency Injection*** *-* это шаблон проектирования для реализации IoC, где

инвертируемым (переопределяемым) элементом контроля является настройка зависимостей

объекта. Соединение объектов с другими объектами или «внедрение» объектов в другие

объекты выполняется контейнером IoC, а не самими объектами.

В Spring Framework инверсия контроля достигается именно внедрением зависимостей.

Инверсия контроля и внедрение зависимостей считаются одним и тем же.

В Spring Framework внедрение зависимостей описывается как процесс, посредством

которого объекты определяют свои зависимости (то есть другие объекты, с которыми они

работают) только через аргументы конструктора, аргументы фабричного метода или свойства, которые устанавливаются в экземпляре объекта после того, как он создан или возвращен из

метода фабрики. После чего контейнер IoC внедряет эти зависимости в компонент при его

создании.

5. ApplicationContext

ApplicationContext - это главный интерфейс в Spring-приложении, который предоставляет информацию о конфигурации приложения. Так же, как BeanFactory, ApplicationContext загружает бины, связывает их вместе и конфигурирует их определённым образом. Но кроме этого, ApplicationContext обладает дополнительной функциональностью.

**Это мапа!**

ApplicationContext предоставляет:

* Фабричные методы бина для доступа к компонентам приложения
* Возможность загружать файловые ресурсы в общем виде
* Возможность публиковать события и регистрировать обработчики на них
* Возможность работать с сообщениями с поддержкой интернационализации
* Наследование от родительского контекста

Зачем так много? Чтоб можно было задавать разные способы кофигурирования?

Есть несколько основных реализаций интерфейса ApplicationContext:

* FileSystemXmlApplicationContext — в конструкторе необходимо указать полный путь к XML-файлу с конфигурацией бинов - для автономных приложений;
* ClassPathXmlApplicationContext — необходимо поместить XML-файл с конфигурацией бинов в CLASSPATH - для автономных приложений;
* XmlWebApplicationContext — загружает XML-файл с метаданными бинов в веб-приложении.
* AnnotationConfigApplicationContext - - для обычной Java-конфигурации, в качестве

аргумента которому передается класс, либо список классов с аннотацией

@Configuration, либо с любой другой аннотацией JSR-330, в том числе и @Component.

* WebApplicationContext - для веб-приложений;

ApplicationContext ctx = new FileSystemXmlApplicationContext("c:/bean\_properties.xml");

- пример создания.

**ApplicationContext** — это набор бинов (объектов). Обращаясь к контексту — мы можем получить нужный нам бин (объект) по его имени например, или по его типу, или еще как-то.

Кроме того, мы можем попросить спринг самого сходить поискать в своем контексте нужный нам бин и передать его в наш метод.

**IoC Container**

В Spring Framework контейнер отвечает за создание, настройку и сборку объектов,

известных как бины, а также за управление их жизненным циклом. Он (контейнер) представлен

интерфейсом ApplicationContext.

Контейнер получает инструкции о том, какие объекты создавать, настраивать и

собирать, через метаданные конфигурации, которые представлены в виде XML, Java-аннотаций

или Java-кода:

❖ **XML** - Метаданные считываются из файла с расширением \*.xml;

❖ **Java-аннотации** - В Spring 2.5 появилась поддержка метаданных конфигурации на основе аннотаций, которая использует данные байт-кода для подключения компонентов.

Разработчик перемещает конфигурацию в сам класс компонента, используя аннотации

к соответствующему классу, методу или полю. При этом, сам XML-файл с базовыми

настройками остаётся. Контейнер считывает аннотации перед считыванием XML,

поэтому, если бин конфигурируется и через аннотации и через XML-файл, то настройки

XML переопределят настройки аннотаций.

❖ **Java-код** - Начиная со Spring 3.0, используя Java-код, а не файлы XML, мы можем

определять настройки в специальном классе, помеченном аннотацией @Configuration.

Появились аннотации @Configuration, @Bean, @Import и @DependsOn и т.д.

**Основные этапы поднятия ApplicationContext:**

1 этап: **Парсирование** конфигурации (xml, groovy, JavaConfig и пр.) и создание всех BeanDefinition (AnnotatedBeanDefinitionReader, BeanDefinitionReader, ClassPathBeanDefinitionScanner)

2 этап: **Настройка** созданных BeanDefinition (BeanFactoryPostProcessor)

3 этап: **Создание** кастомных FactoryBean (FactoryBean<T>)

4 этап: **BeanFactory** создает экземпляры бинов, при необходимости делегируя создание FactoryBean (BeanFactory)

5 этап: **Настройка** созданных бинов (BeanPostProcessor)

**ApplicationContext и BeanFactory, чем отличаются? В каких случаях что стоит использовать?**

BeanFactory — это интерфейс, который предоставляет механизм конфигурации,

способный управлять объектами любого типа. В общем, BeanFactory предоставляет

инфраструктуру конфигурации и основные функциональные возможности.

BeanFactory легче по сравнению с ApplicationContext.

ApplicationContext является наследником BeanFactory и полностью реализует его

функционал, добавляя больше специфических enterprise-функций.

**Как получить ApplicationContext в интеграционном тесте:**

Если вы используете JUnit 5, то вам нужно указать 2 аннотации:

* @ExtendWith(TestClass.class) — используется для указания тестового класса
* @ContextConfiguration(classes = JavaConfig.class) — загружает java/xml конфигурацию для создания контекста в тесте

Можно использовать аннотацию [@SpringJUnitConfig](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/context/junit/jupiter/SpringJUnitConfig.html), которая сочетает обе эти аннотации.

Для теста веб-слоя можно использовать аннотацию [@SpringJUnitWebConfig](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/context/junit/jupiter/web/SpringJUnitWebConfig.html).

**Завершить работу ApplicationContext**

Если не веб-приложение:

* Регистрация shutdown-hook с помощью вызова метода registerShutdownHook(), он также реализован в классе AbstractApplicationContext. Это предпочтительный способ.
* Можно вызвать метод close() из класса AbstractApplicationContext.

В Spring Boot:

* Spring Boot самостоятельно зарегистрирует shutdown-hook за вас.

6. Как можно связать бины?

Связывание бинов или внедрение зависимостей происходит по следующему алгоритму:

1. Описание бинов. Производится

- либо полностью в .xml файле,

- либо в джава коде с использованием аннотаций (@Configuration + методы помеченные аннотацией @Bean, которые возвращают бины)

- либо полностью с помощью аннотаций (@Configuration + @ComponentScan + @Component)

2. Собственно внедрение зависимостей, может происходить через конструктор, через сеттеры,через поле над которыми ставиться аннотация @Autowire

7. Что такое Dependency Injection?

Dependency Injection (внедрение зависимости) - это набор паттернов и принципов разработки программного обеспечения, которые позволяют писать слабосвязанный код. В полном соответствии с принципом единой обязанности объект отдаёт заботу о построении требуемых ему зависимостей внешнему, специально предназначенному для этого общему механизму.

***Dependency Injection*** *-* это шаблон проектирования для реализации IoC, где

инвертируемым (переопределяемым) элементом контроля является настройка зависимостей

объекта. Соединение объектов с другими объектами или «внедрение» объектов в другие

объекты выполняется контейнером IoC, а не самими объектами.

В Spring Framework инверсия контроля достигается именно внедрением зависимостей.

Инверсия контроля и внедрение зависимостей считаются одним и тем же.

В Spring Framework внедрение зависимостей описывается как процесс, посредством

которого объекты определяют свои зависимости (то есть другие объекты, с которыми они

работают) только через аргументы конструктора, аргументы фабричного метода или свойства, которые устанавливаются в экземпляре объекта после того, как он создан или возвращен из

метода фабрики. После чего контейнер IoC внедряет эти зависимости в компонент при его

создании.

Во время DI происходит создание спрингом бинов и внедрение их в другие бины. Всего есть 3 варианта внедрения: через конструктор, через сеттеры, через аннотацию над полем.

К достоинствам применения DI можно отнести:

* Сокращение объема связующего кода.
* Упрощенная конфигурация приложения.
* Возможность управления общими зависимостями в единственном репозитории.
* Улучшенная возможность тестирования.
* Стимулирование качественных проектных решений для приложений.

Реализация DI в Spring основана на двух ключевых концепциях Java — **компонентах JavaBean** и **интерфейсах**.

При использовании Spring в качестве поставщика DI- гибкость определения конфигурации зависимостей внутри приложений разнообразными путями (т.е. внешне в XML-файлах, с помощью конфигурационных Java классов Spring или посредством аннотаций Java в коде).

Компоненты JavaBean (также называемые POJO (Plain Old Java Object — простой старый объект Java)) предоставляют стандартный механизм для создания ресурсов Java, которые являются конфигурируемыми множеством способов.

За счет применения DI объем кода, который необходим при проектировании приложения на основе интерфейсов, снижается почти до нуля. Кроме того, с помощью интерфейсов можно получить максимальную отдачу от DI, потому что бины могут использовать любую реализацию интерфейса для удовлетворения их зависимости.

К типам реализации внедрения зависимостей в Spring относят:

* Constructor Dependency Injection — это тип внедрения зависимостей, при котором зависимости компонента предоставляются ему в его конструкторе (или конструкторах). Рекомендуется как основной способ, т.к. даже без спринга внедрение зависимостей будет работать корректно.
* Setter Dependency Injection — контейнер IoC внедряет зависимости компонента в компонент через методы установки в стиле JavaBean. В основном через сеттеры. При модификации не создает новые экземпляры, в отличии от конструктора. Он при каждой модификации создаёт новый экземпляр.

8. Какие бины будут использоваться для настройки приложения?

Чтобы создать класс с конфигурацией на основе Java-кода, нужно аннотировать его с помощью

@Configuration.

Этот класс будет содержать фабричные методы для создания бинов в контейнере.

Эти методы должны быть аннотированы аннотацией @Bean.

Аннотация @Configuration, прописанная перед классом, означает, что класс может быть использован контейнером Spring IoC как конфигурационный класс для бинов.

Аннотация @Bean, прописанная перед методом, информирует Spring о том, что возвращаемый данным методом объект должен быть зарегистрирован, как бин.

Аннотация @ComponentScan("path") задает путь по которому спринг будет искать классы для создания бинов.

9. Контейнеры Spring

Container создаёт объекты, связывает их вместе, настраивает и управляет ими от создания до момента уничтожения. Spring Container получает инструкции какие объекты инстанцировать и как их конфигурировать через метаданные: XML, Аннотации или Java код.

**Spring BeanFactory Container** - это самый простой контейнер, который обеспечивает базовую поддержку DI и который основан на интерфейсе org.springframework.beans.factory.BeanFactory. Такие интерфейсы, как BeanFactoryAware и DisposableBean всё ещё присутствуют в Spring для обеспечения обратной совместимости.

Бины создаются при вызове метода getBean().

Наиболее часто используемая реализация интерфейса BeanFactory – XmlBeanFactory. XmlBeanFactory получает метаданные из конфигурационного XML файла и использует его для создания настроенного приложения или системы. BeanFactory обычно используется тогда, когда ресурсы ограничены (мобильные устройства). Поэтому, если ресурсы не сильно ограничены, то лучше использовать ApplicationContext.

**Spring ApplicationContext Container**. ApplicationContext является более сложным и более продвинутым Spring Container-ом. Наследует BeanFactory и так же загружает бины, связывает их вместе и конфигурирует их определённым образом. Но кроме этого, ApplicationContext обладает дополнительной функциональностью: общий механизм работы с ресурсами, распознавание текстовых сообщений из файлов настройки и отображение событий, которые происходят в приложении различными способами. Этот контейнер определяется интерфейсом org.springframework.context.ApplicationContext.

Бины создаются при "поднятии" контекста все сразу. Если не указана стратегия инициализации.

Чаще всего используются следующие реализации AppicationContext:

* FileSystemXmlApplicationContext - Загружает данные о бине из XML файла. При использовании этой реализации в конструкторе необходимо указать полный адрес конфигурационного файла.
* ClassPathXmlApplicationContext - Этот контейнер также получает данные о бине из XML файла. Но в отличие от FileSystemApplicationContext, в этом случае необходимо указать относительный адрес конфигурационного файла (CLASSPATH).
* AnnotationConfigApplicationContext — метаданные конфигурируются с помощью аннотаций прямо на классах.
* GenericGroovyApplicationContext - эта конфигурация работает по сути так же, как и Xml, только с Groovy-файлами. К тому же, GroovyApplicationContext нормально работает и с Xml-файлом. Принимает на вход строку с конфигурацией контекста. Чтением контекста в данном случае занимается класс GroovyBeanDefinitionReader.

При этом мы можем указать несколько файлов конфигурации Spring.

По своей сути IoC, а, следовательно, и DI, направлены на то, чтобы предложить простой механизм для предоставления зависимостей компонента (часто называемых коллабораторами объекта) и управления этими зависимостями на протяжении всего их жизненного цикла. Компонент, который требует определенных зависимостей, зачастую называют зависимым объектом или, в случае IoC, целевым объектом. Вполне уместно сейчас заявить, что IoC предоставляет службы, через которые компоненты могут получать доступ к своим зависимостям, и службы для взаимодействия с зависимостями в течение их времени жизни.

В общем случае IoC может быть расщеплена на два подтипа: инверсия управления (Dependency Injection) и поиск зависимости (Dependency Lookup). Инверсия управления — это крупная часть того, делает Spring, и ядро реализации Spring основано на инверсии управления, хотя также предоставляются и средства Dependency Lookup. Когда платформа Spring предоставляет коллабораторы зависимому объекту автоматически, она делает это с использованием инверсии управления (Dependency Injection). В приложении, основанном на Spring, всегда предпочтительнее применять Dependency Injection для передачи коллабораторов зависимым объектам вместо того, чтобы заставлять зависимые объекты получать коллабораторы через поиск.

10. Модули Spring

Контейнер **Core** Container (основной) 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.

Контейнер **Data Access**/Integration состоит из JDBC, ORM, OXM, JMS и модуля Transactions.

* JDBC обеспечивает абстрактный слой JDBC и избавляет разработчика от необходимости вручную прописывать монотонный код, связанный с соединением с БД.
* ORM обеспечивает интеграцию с такими популярными ORM, как Hibernate, JDO, JPA и т.д.
* Модуль OXM отвечает за связь Объект/XML – XMLBeans, JAXB и т.д.
* Модуль JMS (Java Messaging Service) отвечает за создание, передачу и получение сообщений.
* Transactions поддерживает управление транзакциями для классов, которые реализуют определенные методы.

Контейнер **Web**. Этот слой состоит из Web, Web-MVC, Web-Socket, Web-Portlet

* Модуль Web обеспечивает такие функции, как загрузка файлов и т.д.
* Web-MVC содержит реализацию Spring MVC для веб-приложений.
* Web-Socket обеспечивает поддержку связи между клиентом и сервером, используя Web-Socket-ы в веб-приложениях.
* Web-Portlet обеспечивает реализацию MVC с среде портлетов.

Spring также включает в себя ряд других важных модулей, таких как **AOP, Aspects, Instrumentation, Messaging и Test**

* AOP реализует аспекто-ориентированное программирование и позволяет использовать весь арсенал возможностей АОП.
* Модуль Aspects обеспечивает интеграцию с AspectJ, которая также является мощным фреймворком АОП.
* Instrumentation отвечает за поддержку class instrumentation и classloader, которые используются в серверных приложениях.
* Модуль Messaging обеспечивает поддержку STOMP.
* И наконец, модуль Test обеспечивает тестирование с использованием TestNG или JUnit Framework.

1. Inversion of Control - контейнер: конфигурирование компонентов приложений и управление жизненным циклом Java-объектов.
2. Фреймворк аспектно-ориентированного программирования: работает с функциональностью, которая не может быть реализована возможностями объектно-ориентированного программирования на Java без потерь.
3. Фреймворк доступа к данным: работает с системами управления реляционными базами данных на Java-платформе, используя JDBC- и ORM-средства и обеспечивая решения задач, которые повторяются в большом числе Java-based environments.
4. Фреймворк управления транзакциями: координация различных API управления транзакциями и инструментарий настраиваемого управления транзакциями для объектов Java.
5. Фреймворк MVC: каркас, основанный на HTTP и сервлетах, предоставляющий множество возможностей для расширения и настройки (customization).
6. Фреймворк удалённого доступа: конфигурируемая передача Java-объектов через сеть в стиле RPC, поддерживающая RMI, CORBA,HTTP-based протоколы, включая web-сервисы (SOAP).
7. Фреймворк аутентификации и авторизации: конфигурируемый инструментарий процессов аутентификации и авторизации, поддерживающий много популярных и ставших индустриальными стандартами протоколов, инструментов, практик через дочерний проект Spring Security (ранее известный как Acegi).
8. Фреймворк удалённого управления: конфигурируемое представление и управление Java-объектами для локальной или удалённой конфигурации с помощью JMX.
9. Фреймворк работы с сообщениями: конфигурируемая регистрация объектов-слушателей сообщений для прозрачной обработки сообщений из очереди сообщений с помощью JMS, улучшенная отправка сообщений по стандарту JMS API.
10. Тестирование: каркас, поддерживающий классы для написания модульных и интеграционных тестов.

11. @Transactional Как спринг работает с транзакциями?

Spring поддерживает два типа управления транзакциями:

**Программное** управление транзакциями: вы должны управлять транзакциями с помощью программирования. Это способ достаточно гибкий, но его сложно поддерживать. Либо через использование TransactionTemplate, либо через реализацию PlatformTransactionManager напрямую. Используется, если нужно работать с небольшим количеством транзакций.

**Декларативное** управление транзакциями: вы отделяете управление транзакциями от бизнес-логики. Вы используете только аннотации @Transactional и конфигурацией на основе XML для управления транзакциями. Наиболее предпочтительный способ.

Для работы с транзакциями Spring Framework использует AOP-прокси.

Для включения возможности управления транзакциями первым делом нужно

разместить аннотацию @**EnableTransactionManagement** у класса-конфигурации @**Configuration**.

Аннотация @EnableTransactionManagement означает, что классы, помеченные

@Transactional, должны быть обернуты аспектом транзакций. Однако, если мы используем

Spring Boot и имеем зависимости spring-data-\* или spring-tx, то управление транзакциями будет

включено по умолчанию.

@EnableTransactionManagement отвечает за регистрацию необходимых компонентов

Spring, таких как TransactionInterceptor и советы прокси (proxy advices- набор инструкций,

выполняемых на точках среза - Pointcut). Регистрируемые компоненты помещают перехватчик

в стек вызовов при вызове методов @Transactional.

Spring создает прокси для всех классов, помеченных @Transactional (либо если любой из

методов класса помечен этой аннотацией). Прокси-объекты позволяют Spring Framework вводить транзакционную логику до и после вызываемого метода - главным образом для запуска

и коммита/отката транзакции.

Если мы разместим аннотацию @Transactional над классом @Service, то все его методы

станут транзакционными. Так, при вызове, например, метода save() произойдет примерно

следующее:

**1.** Вначале мы имеем:

❖ класс TransactionInterceptor, у которого основной метод invoke(...), внутри

которого вызывается метод класса-родителя TransactionAspectSupport:

invokeWithinTransaction(...), в рамках которого происходит магия транзакций.

❖ TransactionManager: решает, создавать ли новый EntityManager и/или

транзакцию.

❖ EntityManager proxy: EntityManager - это интерфейс, и то, что внедряется в бин в

слое DAO на самом деле не является реализацией EntityManager. В это поле

внедряется EntityManager proxy, который будет перехватывать обращение к

полю EntityManager и делегировать выполнение конкретному EntityManager в

рантайме. Обычно EntityManager proxy представлен классом SharedEntityManagerInvocationHandler.

**2.** Transaction Interceptor

В TransactionInterceptor отработает код до работы метода save(), в котором будет

определено, выполнить ли метод save() в пределах уже существующей транзакции БД

или должна стартовать новая отдельная транзакция. TransactionInterceptor сам не

содержит логики по принятию решения, решение начать новую транзакцию, если это

нужно, делегируется TransactionManager. Грубо говоря, на данном этапе наш метод

будет обёрнут в try-catch и будет добавлена логика до его вызова и после.

**3.** TransactionManager

Менеджер транзакций должен предоставить ответ на два вопроса:

❖ Должен ли создаться новый EntityManager?

❖ Должна ли стартовать новая транзакция БД?

TransactionManager принимает решение, основываясь на следующих фактах:

❖ выполняется ли хоть одна транзакция в текущий момент или нет;

❖ атрибута «propagation» у метода, аннотированного @Transactional (для примера, значение REQUIRES\_NEW всегда стартует новую транзакцию).

Если TransactionManager решил создать новую транзакцию, тогда:

❖ Создается новый EntityManager;

❖ EntityManager «привязывается» к текущему потоку (Thread);

❖ «Получается» соединение из пула соединений БД;

❖ Соединение «привязывается» к текущему потоку.

И EntityManager и это соединение привязываются к текущему потоку, используя

переменные ThreadLocal.

**4.** EntityManager proxy

Когда метод save() слоя Service делает вызов метода save() слоя DAO, внутри которого

вызывается, например, entityManager.persist(), то не происходит вызов метода persist()

напрямую у EntityManager, записанного в поле класса DAO. Вместо этого метод вызывает

EntityManager proxy, который достает текущий EntityManager для нашего потока, и у него

вызывается метод persist().

**5.** Отрабатывает DAO-метод save().

**6.** TransactionInterceptor

Отработает код после работы метода save(), а именно будет принято решение по

коммиту/откату транзакции.

Кроме того, если мы в рамках одного метода сервиса обращаемся не только к методу

save(), а к разным методам Service и DAO, то все они буду работать в рамках одной транзакции,

которая оборачивает этот метод сервиса.

Вся работа происходит через прокси-объекты разных классов. Представим, что у нас в

классе сервиса только один метод с аннотацией @Transactional, а остальные нет. Если мы

вызовем метод с @Transactional, из которого вызовем метод без @Transactional, то оба будут

отработаны в рамках прокси и будут обернуты в нашу транзакционную логику. Однако, если мы

вызовем метод без @Transactional, из которого вызовем метод с @Transactional, то они уже не

будут работать в рамках прокси и не будут обернуты в нашу транзакционную логику.

**У @Transactional есть ряд параметров:**

❖ @Transactional (isolation=Isolation.READ\_COMMITTED) - уровень изоляции.

❖ @Transactional(timeout=60) - По умолчанию используется таймаут, установленный по

умолчанию для базовой транзакционной системы. Сообщает TransactionManager-у о

продолжительности времени, чтобы дождаться простоя транзакции, прежде чем

принять решение об откате не отвечающих транзакций.

❖ @Transactional(propagation=Propagation.REQUIRED) - (Если не указано,

распространяющееся поведение по умолчанию — REQUIRED.) Указывает, что целевой

метод не может работать без другой транзакции. Если до вызова этого метода уже была

запущена транзакция, то метод будет работать в той же транзакции, если транзакции не

было, то будет создана новая.

REQUIRES\_NEW -Указывает, что новая транзакция должна запускаться каждый раз при

вызове целевого метода. Если транзакция уже идет, она будет приостановлена, прежде

чем будет запущена новая.

MANDATORY - Указывает, что для целевого метода требуется активная транзакция. Если

активной транзакции нет, метод не сработает и будет выброшено исключение.

SUPPORTS - Указывает, что целевой метод может выполняться независимо от наличия

транзакции. Если транзакция работает, он будет участвовать в той же транзакции. Если

транзакции нет, он всё равно будет выполняться, если не будет ошибок. Методы,

которые извлекают данные, являются лучшими кандидатами для этой опции.

NOT\_SUPPORTED - Указывает, что целевой метод не требует распространения контекста

транзакции. В основном те методы, которые выполняются в транзакции, но выполняют

операции с оперативной памятью, являются лучшими кандидатами для этой опции.

NEVER - Указывает, что целевой метод вызовет исключение, если выполняется в

транзакционном процессе. Этот вариант в большинстве случаев не используется в

проектах.

❖ @Transactional (rollbackFor=Exception.class) - Значение по умолчанию:

rollbackFor=RunTimeException.class В Spring все классы API бросают RuntimeException, это

означает, что если какой-либо метод не выполняется, контейнер всегда откатывает

текущую транзакцию. Проблема заключается только в проверяемых исключениях.

Таким образом, этот параметр можно использовать для декларативного отката

транзакции, если происходит Checked Exception.

❖ @Transactional (noRollbackFor=IllegalStateException.class) - Указывает, что откат не

должен происходить, если целевой метод вызывает это исключение. Если внутри

метода с @Transactional есть другой метод с аннотацией @Transactional (вложенная

транзакция), то отработает только первая (в которую вложена), из-за особенностей

создания proxy.

**Распространение транзакций**

Когда вызывается метод с @Transactional происходит особая уличная магия: proxy, который создал Spring, создаёт persistence context (или соединение с базой), открывает в нём транзакцию и сохраняет всё это в контексте нити исполнения (натурально, в ThreadLocal). По мере надобности всё сохранённое достаётся и внедряется в бины. Привязка транзакций к нитям (threads) позволяет использовать семантику серверов приложений J2EE, в которой гарантируется, что каждый запрос получает свою собственную нить.

Таким образом, если в вашем коде есть несколько параллельных нитей, у вас будет и несколько параллельных транзакций, которые будут взаимодействовать друг с другом согласно уровням изоляции. Но что произойдёт, если один метод с @Transactional вызовет другой метод с @Transactional? В Spring можно задать несколько вариантов поведения, которые называются правилами распространения.

2 часть

1. Как получить данные из файла .property?

1. В SpringBoot заранее прописан файл application.properties, если вы используете просто Spring, то надо указывать конфигурацию @PropertySource (classpath:application.properties" ). Обращаясь к property.**getProperty**({**ключ**}) – вы получаете его значение.
2. @Value на параметрах конструктора
3. @Autowired private Environment env; ... env.getProperty("name");

2. Как запустить Спринг-приложение из-под сервера Tomcat?

1. Добавить конфигурацию Tomcat
2. Расширяем main-класс с помощью SpringBootServletInitializer.
3. В поме указываем <packaging>war</packaging>
4. добавляем зависимость spring-boot-starter-tomcat

3. Что такое Artifacts?

Это проект со всеми его зависимостями: библиотеками, плагинами, статическими ресурсами предназначенный для тестирования, деплоя, дистрибуции. Артефакт может быть в виде архива jar, war etc.. либо в виде директории в которой хранится то же что и в архиве т.е. скомпиливанный исходный код, библиотеки, включенные в зависимости, статические ресурсы (веб страницы, картинки, js файлы...), другие артефакты.

4. В чем отличие артефакта war от war exploded?

war артефакт упакованный в архив т.е. все файлы проекта находятся в одном архивном файле, с которым работает контейнер.

war exploded этот тот же проект только в не заархивированном виде.

5. Какая разница между аннотациями @Component, @Repository и @Service в Spring?

@Component Это аннотация Spring Framework, ею мы помечаем класс, если хотим, чтобы из этого

класса был создан бин. Именно эту аннотацию ищет Spring Framework, когда сканирует наши

классы. Можно указать имя (Id) для создаваемого бина, а можно не указывать, тогда по

умолчанию именем будет название класса с маленькой буквы.

Аннотация @Component имеет наследников: @Repository, @Service и @Controller.

Все они являются частными случаями использования @Component для слоёв DAO, сервиса и

контроллера MVC соответственно. Технически они одинаковы, но мы используем их для разных целей.

Задача @**Repository** заключается в том, чтобы отлавливать определенные исключения

персистентности и пробрасывать их как одно непроверенное исключение Spring Framework. Для

этого в контекст должен быть добавлен класс PersistenceExceptionTranslationPostProcessor.

Мы помечаем бины аннотацией @**Service**, чтобы указать, что они содержат бизнес-

логику. Так что нет никакого другого предназначения, кроме как использовать ее на уровне

сервиса.

Итог:

❖ @Component - Spring определяет этот класс как кандидата для создания bean.

❖ @Service - класс содержит бизнес-логику и вызывает методы на уровне хранилища.

Ничем не отличается от классов с @Component.

❖ @Repository - указывает, что класс выполняет роль хранилища (объект доступа к DAO).

❖ @Controller - указывает, что класс выполняет роль контроллера MVC. DispatcherServlet

просматривает такие классы для поиска @RequestMapping.

@RequestMapping используется для мапинга (связывания) с URL для всего класса или для

конкретного метода обработчика.

Аннотация **@Component** (как и @Service и @Repository) используется для

автоматического обнаружения и автоматической настройки бинов в ходе сканирования путей к

классам.

Аннотация **@Bean** используется для явного объявления бина, а не для того, чтобы Spring

делал это автоматически в ходе сканирования путей к классам:

❖ прописываем вручную метод для создания бина;

❖ делает возможным объявление бина независимо от объявления класса, что позволяет

использовать классы из сторонних библиотек, у которых мы не можем указать

аннотацию @Component;

❖ с аннотацией @Bean можно настроить initMethod, destroyMethod, autowireCandidate,

делая создание бина более гибким.

Аннотация @Autowired

Это аннотация Spring Framework, ею помечают конструктор, поле, сеттер-метод или

метод конфигурации, сигнализируя, что им обязательно требуется внедрение зависимостей.

Если в контейнере не будет обнаружен необходимый для вставки бин, то будет

выброшено исключение, либо можно указать @Autowired(required = false), означающее, что

внедрение зависимости в данном месте не обязательно.

Аннотация @Autowired является альтернативой Java-аннотации @Inject, не имеющей

required = false (зависимость должна быть обязательно внедрена).

Начиная со Spring Framework 4.3, аннотация @Autowired для конструктора больше не

требуется, если целевой компонент определяет только один конструктор. Однако, если

доступно несколько конструкторов и нет основного/стандартного конструктора, по крайней

мере один из конструкторов должен быть аннотирован @Autowired, чтобы указать контейнеру,

какой из них использовать.

По умолчанию Spring распознает объекты для вставки по типу. Если в контейнере

доступно более одного бина одного и того же типа, будет исключение. Для избежания этого

можно указать аннотацию Spring Framework - @**Qualifier**("fooFormatter"), где fooFormatter — это имя (Id) одного из нескольких бинов одного типа, находящихся в контейнере и доступных для

внедрения.

При выборе между несколькими бинами при автоматическом внедрении используется

имя поля. Это поведение по умолчанию, если нет других настроек.

Аннотация @Resource

Java-аннотация @Resource может применяться к классам, полям и методам. Она

пытается получить зависимость: сначала по имени, затем по типу, затем по описанию (Qualifier).

Имя извлекается из имени аннотируемого сеттера или поля, либо берется из параметра name.

При аннотировании классов имя не извлекается из имени класса по умолчанию, поэтому оно

должно быть указано явно.

Указав данную аннотацию у полей или методов с аргументом name, в контейнере будет

произведен поиск компонентов с данным именем, и в контейнере должен быть бин с таким

именем.

Если указать её без аргументов, то Spring Framework поможет найти бин по типу.

Если в контейнере несколько бинов-кандидатов на внедрение, то нужно использовать

аннотацию @Qualifier.

Разница с @Autowired:

❖ ищет бин сначала по имени, а потом по типу;

❖ не нужна дополнительная аннотация для указания имени конкретного бина;

❖ @Autowired позволяет отметить @Autowired(required = false); место вставки бина как необязательное

❖ при замене Spring Framework на другой фреймворк, менять аннотацию @Resource не

нужно.

Аннотация @Inject

Размещается над полями, методами, и конструкторами с аргументами. @Inject как и

@Autowired в первую очередь пытается подключить зависимость по типу, затем по описанию и

только потом по имени. Это означает, что даже если имя переменной ссылки на класс

отличается от имени компонента, но они одинакового типа, зависимость все равно будет

разрешена.

Аннотации @Primary и @Qualifier

Если есть два одинаковых бина (по типу и имени) спринг не знает какой именно использовать и выдает exception. Если над одним из этих бинов установлена @Primary, то его использовать предпочтительнее (может использоваться на этапе конфигурации с соответствующими аннотациями). Но если нам нужно использовать в работе оба этих бина, можно над каждым поставить @Qualifier и задать имя, для идентификации этих бинов (работает только с @Autowired).

Мы используем @Primary, чтобы отдавать предпочтение бину, когда есть несколько

бинов одного типа. Эта аннотация полезна, когда мы хотим указать, какой компонент

определенного типа должен внедряться по умолчанию.

Когда есть несколько бинов одного типа, подходящих для внедрения, аннотация

@Qualifier позволяет указать в качестве аргумента имя конкретного бина, который следует

внедрить.

Стоит отметить, что если присутствуют аннотации @Qualifier и @Primary, то аннотация

@Qualifier будет иметь приоритет. По сути, @Primary определяет значение по умолчанию, в то

время как @Qualifier более специфичен.

Аннотация @ComponentScan

Аннотация @ComponentScan используется вместе с аннотацией @Configuration для

указания пакетов, которые мы хотим сканировать на наличие компонентов, из которых нужно

сделать бины.

@ComponentScan без аргументов указывает Spring по умолчанию сканировать текущий

пакет и все его подпакеты. Текущий пакет - тот, в котором находится файл конфигурации с этой

самой аннотацией @ComponentScan. В данном случае в контейнер попадут:

❖ бин конфигурационного класса;

❖ бины, объявленные в конфигурационном классе с помощью @Bean;

❖ все бины из пакета и его подпакетов.

Аннотация @SpringBootApplication включает в себя аннотации @ComponentScan,

@SpringBootConfiguration и @EnableAutoConfiguration, но это не мешает разместить её ещё раз

отдельно для указания конкретного пакета.

* Если указать @ComponentScan с атрибутом basePackages, то это изменит пакет по

умолчанию на указанный.

* Если указать @ComponentScan с атрибутом excludeFilters, то это позволит использовать

фильтр и исключить ненужные классы из процесса сканирования.

Первый шаг для описания Spring Beans это добавление аннотации — @Component, или @Service, или @Repository.

Однако, Spring ничего не знает об этих бинах, если он не знает где искать их. То, что скажет Spring где искать эти бины и называется Component Scan. В @ComponentScan вы указываете пакеты, которые должны сканироваться.

Аннотацию @Profile

Профили — это ключевая особенность Spring Framework, позволяющая нам относить

наши бины к разным профилям (логическим группам), например, dev, test, prod.

Мы можем активировать разные профили в разных средах, чтобы загрузить только те

бины, которые нам нужны.

Используя аннотацию @Profile, мы относим бин к конкретному профилю. Её можно

применять на уровне класса или метода. Аннотация @Profile принимает в качестве аргумента

имя одного или нескольких профилей. Она фактически реализована с помощью гораздо более

гибкой аннотации @Conditional.

Рассмотрим базовый сценарий - у нас есть бин, который должен быть активным только

во время разработки, но не должен использоваться в продакшене. Мы аннотируем этот

компонент с профилем «dev», и он будет присутствовать в контейнере только во время

разработки - во время продакшена профиль dev просто не будет активен.

Аннотация @Conditional

Часто бывает полезно включить или отключить весь класс @Configuration, @Component

или отдельные методы @Bean в зависимости от каких-либо условий.

Аннотация @Conditional указывает, что компонент имеет право на регистрацию в

контексте только тогда, когда все условия соответствуют. Может применяться:

❖ над классами прямо или косвенно аннотированными @Component, включая классы

@Configuration;

❖ над методами @Bean;

❖ как мета-аннотация при создании наших собственных аннотаций-условий.

Условия проверяются непосредственно перед тем, как должно быть зарегистрировано

BeanDefinition компонента, и они могут помешать регистрации данного BeanDefinition. Поэтому

нельзя допускать, чтобы при проверке условий мы взаимодействовали с бинами (которых еще

не существует), с их BeanDefinition-ами можно.

Условия мы определяем в специально создаваемых нами классах, которые должны

имплементировать функциональный интерфейс Condition с одним единственным методом,

возвращающим true или false. Создав свой класс и переопределив в нем метод matches() с нашей логикой, мы должны передать этот класс в аннотацию @Conditional в качестве параметра.

Если класс @Configuration помечен как @Conditional, то на все методы @Bean,

аннотации @Import и аннотации @ComponentScan, связанные с этим классом, также будут

распространяться указанные условия.

Для более детальной настройки классов, аннотированных @Configuration, предлагается

использовать интерфейс ConfigurationCondition.

6. Структура MVC-приложения? Паттерн MVC, как он реализован в Spring?

MVC (Model-View-Controller)

Это шаблон проектирования программного обеспечения, который делит программную

логику на три отдельных, но взаимосвязанных компонента: модель, представление и

контроллер — таким образом, что модификация каждого компонента может осуществляться

независимо.

**Модель** (Model) предоставляет данные и реагирует на команды контроллера, изменяя

своё состояние. Она содержит всю бизнес-логику приложения.

**Представление** (View) отвечает за отображение пользователю данных из модели в

нужном формате.

**Контроллер** (Controller) содержит код, который отвечает за обработку действий

пользователя и обменивается данными с моделью (любое действие пользователя в системе

обрабатывается в контроллере).

Основная цель следования принципам MVC — отделить реализацию бизнес-логики

приложения (модели) от ее визуализации (вида). Такое разделение повысит возможность

повторного использования кода.

Польза применения MVC наиболее наглядна в случаях, когда пользователю нужно

предоставлять одни и те же данные в разных формах. Например, в виде таблицы, графика или

диаграммы (используя различные виды). При этом, не затрагивая реализацию видов, можно

изменить реакции на действия пользователя (нажатие мышью на кнопке, ввод данных).

**Spring Web MVC**

Spring MVC - это оригинальный веб-фреймворк, основанный на Servlet API,

предназначенный для создания веб-приложений на языке Java, с использованием двух самых

популярных шаблонов проектирования - Front controller и MVC.

**Front controller** (Единая точка входа) - паттерн, где центральный сервлет,

DispatcherServlet, принимает все запросы и распределяет их между контроллерами,

обрабатывающими разные URL.

**Spring MVC** реализует четкое разделение задач, что позволяет нам легко разрабатывать

и тестировать наши приложения. Данные задачи разбиты между разными компонентами:

Dispatcher Servlet, Controllers, View Resolvers, Views, Models, ModelAndView, Model and Session

Attributes, которые полностью независимы друг от друга, и отвечают только за одно

направление. Поэтому MVC дает нам довольно большую гибкость. Он основан на интерфейсах

(с предоставленными классами реализации), и мы можем настраивать каждую часть

фреймворка с помощью пользовательских интерфейсов.

Основные интерфейсы для обработки запросов:

* **HandlerMapping**. По запросу определяет, какие перехватчики (interceptors) с пре- и пост-

процессорной обработкой запроса должны отработать, а затем решает, какому контроллеру

(обработчику) нужно передать данный запрос на исполнение. Процесс их определения основан

на некоторых критериях, детали которых зависят от реализации HandlerMapping.

Двумя основными реализациями HandlerMapping являются RequestMappingHandlerMapping

(который поддерживает аннотированные методы @RequestMapping) и SimpleUrlHandlerMapping (который поддерживает явную регистрацию путей URI для обработчиков).

* **HandlerAdapter**. Помогает DispatcherServlet вызвать обработчик, сопоставленный с

запросом. Для вызова аннотированного контроллера необходимо прочитать аннотации над

методами контроллера и принять решение. Основная цель HandlerAdapter - избавить

DispatcherServlet от этой рутины.

* **ViewResolver**. Сопоставляет имена представлений, возвращаемых методами

контроллеров, с фактическими представлениями (html-файлами).

Все платформы MVC предоставляют способ работы с представлениями. Spring делает это

с помощью ViewResolver, который позволяет отображать модели в браузере, не привязывая

реализацию к определенной технологии представления.

Spring Framework поставляется с довольно большим количеством ViewResolver, например

InternalResourceViewResolver, XmlViewResolver, ResourceBundleViewResolver и несколькими

другими. По умолчанию реализацией интерфейса ViewResolver является класс InternalResourceViewResolver.

Любым реализациям ViewResolver желательно поддерживать интернационализацию, то

есть множество языков.

* **View**. Отвечает за возвращение ответа клиенту в виде текстов и изображений.

Используются встраиваемые шаблонизаторы (Thymeleaf, FreeMarker и т.д.), так как у Spring нет

родных. Некоторые запросы могут идти прямо во View, не заходя в Model, другие проходят

через все слои.

* **LocaleResolver**. Определение часового пояса и языка клиента для того, чтобы

предложить представления на его языке.

* **MultipartResolver**. Обеспечивает Upload — загрузку на сервер локальных файлов

клиента. По умолчанию этот интерфейс не включается в приложении и необходимо указывать

его в файле конфигурации. После настройки любой запрос о загрузке будет отправляться этому

интерфейсу.

* **FlashMapManager**. Сохраняет и извлекает «входной» и «выходной» FlashMap, который

можно использовать для передачи атрибутов из одного запроса в другой, обычно через

редирект.

Ниже приведена **последовательность событий, соответствующая входящему HTTP-**

**запросу**:

❖ После получения HTTP-запроса DispatcherServlet обращается к интерфейсу

HandlerMapping, который определяет, какой Контроллер (Controller) должен быть

вызван, после чего HandlerAdapter, отправляет запрос в нужный метод Контроллера.

❖ Контроллер принимает запрос и вызывает соответствующий служебный метод,

основанный на GET, POST и т.д. Вызванный метод формирует данные Модели

(например, набор данных из БД) и возвращает их в DispatcherServlet вместе с именем

Представления (View) (как правило имя html-файла).

❖ При помощи интерфейса ViewResolver DispatcherServlet определяет, какое

Представление нужно использовать на основании полученного имени и получает в

ответе имя представления View.

➢ если это REST-запрос на сырые данные (JSON/XML), то DispatcherServlet сам его

отправляет;

➢ если обычный запрос, то DispatcherServlet отправляет данные Модели в виде

атрибутов в Представление (View) - шаблонизаторы Thymeleaf, FreeMarker и т.д.,

которые сами отправляют ответ.

Все действия происходят через один единственный DispatcherServlet.

Сконфигурировать наше Spring MVC-приложение мы можем с помощью Java-config,

добавив зависимость spring-webmvc и установив над классом конфигурации @EnableWebMvc,

которая применит дефолтные настройки - зарегистрирует некоторые специальные бины из

Spring MVC и адаптирует их к нашим бинам. Но, если требуется тонкая настройка, то мы можем

имплементировать интерфейс WebMvcConfigurer и переопределить необходимые методы.

Теперь нужно зарегистрировать конфигурацию в Spring Context это позволит сделать

созданный нами класс MyWebAppInitializer, который нужно унаследовать от

AbstractAnnotationConfigDispatcherServletInitializer, и передать в его методы классы нашей

конфигурации RootConfig.class и App1Config.class.

Своими внутренними методами он создает два экземпляра WebApplicationContext в

виде объектов класса AnnotationConfigWebApplicationContext.

Если же у нас только один класс конфигурации, то его нужно передать в метод

getRootConfigClasses(), а getServletConfigClasses() должен возвращать null.

7. Чем контроллер отличается от сервлета?

Web – это система интернет-серверов, которая поддерживает форматированные документы. Документы форматируются с использованием языка разметки, называемого HTML (HyperText Markup Language), который поддерживает ссылки на другие документы, такие как графические, аудио- и видеофайлы и т. д.

Сервлет – это программный модуль Java на стороне сервера, который обрабатывает запросы клиентов и реализует интерфейс сервлета. Могут отвечать на любые запросы, и они обычно используются для расширения приложений, размещаемых на веб-серверах.

**Жизненный цикл**

Весь жизненный цикл сервлета управляется контейнером сервлета, который использует интерфейс javax.servlet.Servlet для понимания объекта сервлета и управления им.

Жизненный цикл сервлета проходит в основном четыре этапа:

1. Загрузка. Когда сервер запускается, контейнер сервлета развертывается и загружает все сервлеты.
2. Инициализация. Затем сервлет инициализируется путем вызова метода init(). Метод Servlet.init() вызывается контейнером Servlet для уведомления о том, что этот экземпляр Servlet успешно создан и готов к запуску.
3. Обработка запросов. Затем сервлет вызывает метод service() для обработки запроса клиента и вызывается для информирования сервлета о клиентских запросах.
4. Уничтожение. Наконец, сервлет завершается вызовом destroy(). Метод запускается только один раз за время существования сервлета и сообщает об окончании экземпляра сервлета.

Методы init() и destroy() вызываются только один раз. Наконец, сервлет – это сборщик мусора сборщиком мусора JVM. Итак, на этом завершается жизненный цикл сервлета.

**Шаги по созданию**

Чтобы создать сервлет, нам нужно выполнить несколько шагов по порядку. Они заключаются в следующем:

1. Создать структуру каталогов.
2. Создать сервлет.
3. Скомпилировать сервлет.
4. Добавить сопоставления в файл web.xml.
5. Запустить сервер и развернуть проект.
6. Доступ к сервлету.

Для запуска программы сервлета у нас должен быть установлен и настроен сервер Apache Tomcat.

Отслеживание сеансов.

Сессия просто означает определенный интервал времени. Отслеживание сеанса – это метод поддержания состояния (данных) пользователя, также известный как управление сеансом в сервлете. Таким образом, каждый раз, когда пользователь запрашивает сервер, сервер обрабатывает запрос как новый запрос.

Чтобы распознать конкретного пользователя, нам нужно отслеживать сеанс.

Файлы cookie

Файл cookie – это небольшой фрагмент информации, который сохраняется между несколькими клиентскими запросами. Имеет имя, одно значение и дополнительные атрибуты, такие как комментарий, путь и квалификаторы домена, максимальный возраст и номер версии.

Поскольку это метод отслеживания сеансов, по умолчанию каждый запрос рассматривается как новый запрос.

Страницы сервера Java

JSP или Java Server Pages – это технология, которая используется для создания веб-приложений точно так же, как технология сервлетов. Это расширение Servlet – поскольку оно предоставляет больше функциональных возможностей, чем сервлет, такой как язык выражений, JSTL и т. Д. Страница JSP состоит из тегов HTML и тегов JSP. Страницы JSP легче поддерживать, чем сервлет, потому что мы можем разделить проектирование и разработку.

**Сервлет** представляет специальный тип классов Java, который выполняется на веб-сервере и который обрабатывает запросы и возвращает результат обработки.

**Контроллер** это часть архитектурного шаблона MVC отвечающая за обработку входящих запросов.

В контексте Spring MVC в качестве сервлета выступает dispatcher servlet, реализующий паттерн FrontController, который принимает все входящие запросы и перенаправляет их на соответствующие контроллеры.

В приложении Spring MVC может существовать произвольное количество экземпляров DispatcherServlet, предназначенных для разных целей (например, для обработки запросов пользовательского интерфейса, запросов веб-служб REST и т.д.). Каждый экземпляр DispatcherServlet имеет собственную конфигурацию WebApplicationContext, которая определяет характеристики уровня сервлета, такие как контроллеры, поддерживающие сервлет, отображение обработчиков, распознавание представлений, интернационализация, оформление темами, проверка достоверности, преобразование типов и форматирование и т.п.

Сервлет принимает запросы от пользователя и перенаправляет их на определенный контроллер.

Контроллер:

* Обрабатывает запросы от пользователя
* Обменивается данными с моделью
* Показывает пользователю правильное представление
* Переадресовывает пользователя на другие страницы

8. Основная зависимость фреймворка Спринг? Почему во многих сборках она не указывается явно?

Spring-core. Не указывается потому, что она включается потому что включается по умолчанию.

9. Как вернуть страницу в контроллере? Как вернуть данные?

- В методе контроллера, в возвращаемом значении указать имя представления

- model.addAttribute("var\_name", data)

10. Model, ModelMap и ModelAndView?

**Model**. Интерфейс, лежит в пакете spring-context. В методах контроллера мы можем

использовать объекты Model для того, чтобы складывать туда данные, предназначенные для

формирования представлений. Кроме того, в Model мы можем передать даже Map с

атрибутами.

**ModelMap.** Этот класс наследуется от LinkedHashMap<String, Object> и по сути служит общим контейнером модели для Servlet MVC, но не привязан к нему, и лежит в пакете spring-context. Имеет все преимущества LinkedHashMap плюс несколько удобных методов

**ModelAndView**. Этот класс лежит в пакете spring-webmvc и может одновременно хранить модели и представление, чтобы контроллер мог отдавать их в одном возвращаемом значении. Внутри

содержит поле private Object view, куда записывает нужное представление, а также поле private

ModelMap model, куда и складывает все атрибуты модели

Spring MVC поддерживает несколько поставщиков View(они называются шаблонизаторы) — JSP, JSF. Интерфейс View преобразует объекты в обычные сервлеты.

11. **Аннотации @Controller и @RestController. Чем они отличаются? Как вернуть ответ со своим статусом (например 213)?**

@**Controller** помечает класс как контроллер HTTP-запросов. @Controller обычно

используется в сочетании с аннотацией @RequestMapping, используемой в методах обработки

запросов. Это просто дочерняя аннотация аннотации @Component и позволяет автоматически

определять классы при сканировании пакетов.

Аннотация @**RestController** была введена в Spring 4.0 для упрощения создания RESTful

веб-сервисов. Это удобная аннотация, которая объединяет @Controller и @ResponseBody, что

устраняет необходимость аннотировать каждый метод обработки запросов аннотацией

@**ResponseBody** сообщает контроллеру, что возвращаемый объект автоматически

сериализуется в json или xml и передается обратно в объект HttpResponse. Контроллер

использует Jackson message converter для конвертации входящих/исходящих данных. Как

правило целевые данные представлены в json или xml.

**ResponseEntity**

Данный класс используется для формирования ответа HTTP с пользовательскими

параметрами (заголовки, код статуса и тело ответа). ResponseEntity необходим, только если мы

хотим кастомизировать ответ. Во всех остальных случаях достаточно использовать @ResponseBody.

Если мы хотим использовать ResponseEntity, то просто должны вернуть его из метода,

Spring позаботится обо всем остальном.

Если клиент ждет от нас JSON/XML, мы можем параметризовать ResponseEntity

конкретным классом и добавить к ответу заголовки и Http статус.

12. Прокси-объекты

Прокси это специальный объект, который имеет такие же публичные методы как и бин, но у которого есть дополнительная функциональность.

Два вида прокси:

* JDK-proxy — динамическое прокси. API встроены в JDK. Для него необходим интерфейс
* CGLib proxy — не встроен в JDK. Используется когда интерфейс объекта недоступен

Плюсы прокси-объектов:

* Позволяют добавлять доп. логику — управление транзакциями, безопасность, логирование
* Отделяет некоторый код(логирование и т.п.) от основной логики

3 часть

1. Принципы работы Spring.

Spring Framework — один из самых популярных фреймворков для создания приложений на Java. Вы просто пишете какие-то свои классы, прописываете там какую-то часть логики, а создает объекты ваших классов и вызывает методы за вас уже сам фреймворк. Чаще всего, ваши классы имплементируют какие-то интерфейсы из фреймворка или наследуют какие-то классы из него, таким образом получая часть уже написанной за вас функциональности. Но не обязательно именно так. В спринге например стараются по максимуму отойти от такой жесткой связности (когда ваши классы напрямую зависят от каких-то классов/интерфейсов из этого фреймворка), и используют для этой цели аннотации.

Но важно понять, что *спринг* — это просто набор каких-то классов и интерфейсов, которые уже написаны за вас.

**Структура.** Фреймворк состоит как-бы из нескольких модулей:

* 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);
* core;
* и других.

**Конфигурация Spring - приложения:**

Существует *три основных способа конфигурации приложения* (то-есть, указания спрингу какие именно объекты нам нужны для работы):

1. при помощи xml файлов/конфигов;
2. при помощи java-конфигов+аннотации;
3. автоматическая конфигурация+autowired.

*Разработчики спринга выстраивают их в таком порядке приоритетности:*

* наиболее приоритетный способ, которому стоит отдавать предпочтение — это автоматическая конфигурация;
* если при помощи автоматической конфигурации нет возможности правильно настроить все возможные бины — использовать джава-конфигурацию (создание объектов используя джава код);
* ну и самый низкоприоритетный способ — это по-старинке, используя xml конфиги.

Кроме того, спринг позволяет комбинировать эти способы. Например, все то, что может быть настроено автоматически — пусть спринг сделает сам, там где надо указать какие-то особые параметры — сделать при помощи джава-конфигов, и кроме того, можно подключить какие-то легаси конфиги в xml формате.

2. Связывание бинов и их жизненный цикл.

Связыванием бинов в спринг называется внедрение в бины зависимостей при их инициализации. Это можно делать явно, указывая в конфигурационном xml файле зависимости с помощью тегов <constructor-arg> и <property>, а также в спринге есть механизм автоматического связывания по имени, типу, конструктору.

Жизненный цикл Spring бина — время существования класса. Spring бины инициализируются при инициализации Spring контейнера и происходит внедрение всех зависимостей. Когда контейнер уничтожается, то уничтожается и всё содержимое. Если нам необходимо задать какое-либо действие при инициализации и уничтожении бина, то нужно воспользоваться методами init() и destroy(). Для этого можно использовать аннотации @PostConstruct и @PreDestroy().

Запуск Spring приложение - запускается Spring контейнер, далее проходится по всем файлам и если видит аннотацию Bean - создается объект бина, затем внедряются зависимости (Dependency Injection) - затем вызывается init метод у данного бина - бин будет готов к использованию и пользователь получит доступ к нему - когда пользователь завершит приложение - спринг вызовет дестрой метод - и затем приложение остановится.

Жизненный цикл бина:

1. создание объекта бина, через констуктор new()
2. внедрение зависимостей
3. нотификация aware-интерфейсов (реализация данных интерфейсов может добавлять бинам какой-то специфичный функционал)
4. пред-инициализация в методах .PostProcessBeforeInitialization() интерфейса BeanPostprocessor
5. инициализация (может быть разными способами) - @PostConstract

- .afterPropertiesSet() бина под интерфейсом InitializingBean

- Init-метод (либо прописать в xml, либо @Bean(initMethod = "init"))

1. пост-инициализация (донастройка бина) в методах .PostProcessAfterInitialization() интерфейса BeanPostprocessor
2. использование бина
3. уничтожение бина при завершении работы IoC контейнера (также может быть разными способами):

- Метод бина с аннотацией @PreDestroy

- Метод указанный в свойстве destroyMethod определения бина

- Метод destroy() интерфейса DisposableBean

**Жизненный цикл бинов:**

1. *Парсирование конфигурации и создание BeanDefinition* — это создание всех BeanDefinition. Объекты BeanDefinition — это набор метаданных будущего бина, макет, по которому нужно будет создавать бин в случае необходимости. То есть для каждого бина создается свой объект BeanDefinition, в котором хранится описание того, как создавать и управлять этим конкретным бином. Проще говоря, сколько бинов в программе - столько и объектов BeanDefinition, их описывающих. eanDefinition содержат (среди прочего) следующие метаданные: имя класса, scope, обратные вызовы, ссылки на другие бин-компоненты, другие параметры конфигурации (размер пула, кол-во соединений).

При конфигурации через аннотации с указанием пакета для сканирования или JavaConfig

используется класс AnnotationConfigApplicationContext. Регистрируются все классы с

@Configuration для дальнейшего парсирования, затем регистрируется специальный

BeanFactoryPostProcessor, а именно BeanDefinitionRegistryPostProcessor, который при помощи

класса ConfigurationClassParser парсирует JavaConfig, загружает описания бинов (BeanDefinition),

создаёт граф зависимостей (между бинами) и создаёт:

Map<String, BeanDefinition> beanDefinitionMap = new ConcurrentHashMap<>(256);

в которой хранятся все описания бинов, обнаруженных в ходе парсинга конфигурации.

1. *Настройка созданных BeanDefinition*. После первого этапа у нас имеется коллекция Map, в которой хранятся BeanDefinition-ы. BeanFactoryPostProcessor-ы на этапе создания BeanDefinition-ов могут их настроить как нам необходимо. BeanFactoryPostProcessor-ы могут даже настроить саму BeanFactory ещё до того, как она начнет работу по созданию бинов. В интерфейсе BeanFactoryPostProcessor всего один метод - postProcessBeanFactory.
2. *Создание кастомных FactoryBean* (только для XML-конфигурации)
3. *Создание экземпляров бинов*. Сначала BeanFactory из коллекции Map с объектами BeanDefinition достаёт те из них, из которых создаёт все BeanPostProcessor-ы, необходимые для настройки обычных бинов. Создаются экземпляры бинов через BeanFactory на основе ранее созданных BeanDefinition.
4. *Настройка созданных бинов*.
5. На данном этапе бины уже созданы, мы можем лишь их донастроить. Интерфейс BeanPostProcessor позволяет вклиниться в процесс настройки наших бинов до того, как они попадут в контейнер. ApplicationContext автоматически обнаруживает любые бины с реализацией BeanPostProcessor и помечает их как “post-processors” для того, чтобы создать их определенным способом. Например, в Spring есть реализации BeanPostProcessor-ов, которые обрабатывают аннотации @Autowired, @Inject, @Value и @Resource. Интерфейс несет в себе два метода: postProcessBeforeInitialization(Object bean, String beanName) и postProcessAfterInitialization(Object bean, String beanName). У обоих методов параметры абсолютно одинаковые. Разница только в порядке их вызова. Первый вызывается до init-метода, второй - после.Как правило, BeanPostProcessor-ы, которые заполняют бины через маркерные интерфейсы или тому подобное, реализовывают метод postProcessBeforeInitialization (Object bean, String beanName), тогда как BeanPostProcessor-ы, которые оборачивают бины в прокси, обычно реализуют postProcessAfterInitialization (Object bean, String beanName). **Прокси** — это класс-декорация над бином. Например, мы хотим добавить логику нашему бину, но джава-код уже скомпилирован, поэтому нам нужно на лету сгенерировать новый класс. Этим классом мы должны заменить оригинальный класс так, чтобы никто не заметил подмены. Есть два варианта создания этого класса: 1. либо он должен наследоваться от оригинального класса (CGLIB) и переопределять его методы, добавляя нужную логику; 2. либо он должен имплементировать те же самые интерфейсы, что и первый класс (Dynamic Proxy). По конвенции спринга, если какой-то из BeanPostProcessor-ов меняет что-то в классе, то он должен это делать на этапе postProcessAfterInitialization(). Таким образом мы уверены, что initMethod у данного бина, работает на оригинальный метод, до того, как на него накрутился прокси.

Хронология событий:

1. Сначала сработает метод postProcessBeforeInitialization() всех имеющихся

BeanPostProcessor-ов.

1. Затем, при наличии, будет вызван метод, аннотированный @PostConstruct.
2. Если бин имплементирует InitializingBean, то Spring вызовет метод afterPropertiesSet() - не рекомендуется к использованию как устаревший.
3. При наличии, будет вызван метод, указанный в параметре initMethod аннотации @Bean.
4. В конце бины пройдут через postProcessAfterInitialization (Object bean, String beanName).

Именно на данном этапе создаются прокси стандартными BeanPostProcessor-ами.

Затем отработают наши кастомные BeanPostProcessor-ы и применят нашу логику к

прокси-объектам. После чего все бины окажутся в контейнере, который будет

обязательно обновлен методом refresh().

1. Но даже после этого мы можем донастроить наши бины ApplicationListener-ами.
2. Теперь всё.

6. *Бины готовы к использованию* Их можно получить с помощью метода ApplicationContext#getBean().

7. *Закрытие контекста*

Когда контекст закрывается (метод close() из ApplicationContext), бин уничтожается.

Если в бине есть метод, аннотированный @PreDestroy, то перед уничтожением

вызывается этот метод.

Если бин имплементирует DisposibleBean, то Spring вызовет метод destroy() - не

рекомендуется к использованию как устаревший.

Если в аннотации @Bean определен метод destroyMethod, то будет вызван и он.

@PostConstruct

Spring вызывает методы, аннотированные @PostConstruct, только один раз, сразу после

инициализации свойств компонента. За данную аннотацию отвечает один из BeanPostProcessor-ов.

Метод, аннотированный @PostConstruct, может иметь любой уровень доступа, может

иметь любой тип возвращаемого значения (хотя тип возвращаемого значения игнорируется

Spring-ом), метод не должен принимать аргументы. Он также может быть статическим, но

преимуществ такого использования метода нет, т.к. доступ у него будет только к статическим

полям/методам бина, и в таком случае смысл его использования для настройки бина пропадает.

Одним из примеров использования @PostConstruct является заполнение базы данных.

Например, во время разработки нам может потребоваться создать пользователей по

умолчанию.

@PreDestroy

Метод, аннотированный @PreDestroy, запускается только один раз, непосредственно

перед тем, как Spring удаляет наш компонент из контекста приложения.

Как и в случае с @PostConstruct, методы, аннотированные @PreDestroy, могут иметь

любой уровень доступа, но не могут быть статическими.Целью этого метода может быть освобождение ресурсов или выполнение любых других задач очистки до уничтожения бина, например, закрытие соединения с базой данных.

Обратите внимание, что аннотации @PostConstruct и @PreDestroy являются частью

Java EE, а именно пакета javax.annotation модуля java.xml.ws.annotation. И поскольку Java EE устарела в Java 9, то с этой версии пакет считается устаревшим (Deprecated). С Java 11

данный пакет вообще удален, поэтому мы должны добавить дополнительную зависимость

для использования этих аннотаций.

3. Паттерны Spring.

**DI, IoC,**

**Proxy,**

**Singleton,**

**Factory,**

**Template,**

**MVC,**

**Front Controller,**

**Service locator**

**Singleton - Creating beans with default scope.**

**Factory - Bean Factory classes**

**Prototype - Bean scopes**

**Adapter - Spring Web and Spring MVC**

**Proxy - Spring Aspect Oriented Programming support**

**Template Method - JdbcTemplate, HibernateTemplate etc**

**Front Controller - Spring MVC DispatcherServlet**

**Data Access Object - Spring DAO support**

**Dependency Injection and Aspect Oriented Programming**

**Proxy** (Заместитель) - представляет функциональные возможности другого класса. объект-оболочка или агент, который вызывается клиентом для доступа к реальному обслуживающему объекту за кулисами. Обеспечивает защиту исходного объекта от внешнего мира.

**Singleton** (Одиночка) - гарантирует, что в памяти будет существовать только один экземпляр объекта, который будет предоставлять сервисы. Рекомендуется использовать бинов без состояния.

**Factory** (Фабрика) -позволяет инициализировать объект через фабричный метод(BeanFactory - Простой контейнер, который обеспечивает базовую поддержку DI (Dependency Injection, инъекция зависимостей). Для работы с этим контейнером используется интерфейс org.springframework.beans.factory.BeanFactory;

Этот паттерн позволяет инициализировать объект через публичный статический метод, называемый фабричным методом.

Spring использует паттерн Factory для создания объекта бина с использованием следующих двух подходов: BeanFactory и ApplicationContext.

**ApplicationContext** - ApplicationContext является центральным интерфейсом в приложении Spring для предоставления информации о конфигурации приложения. Объявляет специфичные enterprise-функции. Эти функции включают в себя возможность чтения параметров из property-файлов и публикацию событий приложения для слушателей событий (предназначен для обработки какого-либо события - он«слушает» событие, например щелчок мышью пользователя или нажатие клавиши, а затем реагирует соответствующим образом.). Для работы с этим контейнером используется интерфейс org.springframework.context.ApplicationContext.

**Template** (Шаблон) - название говорит само за себя. Пример использования -

запрос к базе данных: установка соединения ~> запрос ~> получить всех

0ff ~> закрыть соединение. Этот паттерн широко используется для работы с повторяющимся бойлерплейт кодом (таким как, закрытие соединений и т. п.).

**Model View Controller** (Модель-Представление-Контроллер) - основная задача - отделить логику от визуализации. Паттерн основан на Servlet API. В данном паттерне компоненты не зависят друг от друга и каждый отвечает за определенную задачу.

**Front Controller** (Контроллер запросов) - Spring предоставляет DispatcherServlet, чтобы гарантировать, что входящий запрос будет отправлен вашим контроллерам. Паттерн используется для того, чтобы все запросы обрабатывались одним обработчиком и передавались в main, что так же в свою очередь обеспечивает дополнительную безопасность и удобство для разработчика. Паттерн Front Controller используется для обеспечения централизованного механизма обработки запросов, так что все запросы обрабатываются одним обработчиком. Этот обработчик может выполнить аутентификацию, авторизацию, регистрацию или отслеживание запроса, а затем передать запрос соответствующему контроллеру.

**View Helper** (Вспомогательный компонент представления) - отделяет статическое содержимое в представлении, такое как JSP, HTML, от обработки бизнес-логики. У спринга есть своя библиотека для инкапсуляции логики обработки в хелперах.

**Service Locator** (Локатор служб) - ServiceLocatorFactoryBean сохраняет информацию обо всех бинах в контексте. Когда клиентский код запрашивает сервис (бин) по имени, он просто находит этот компонент в контексте и возвращает его. Клиентскому коду не нужно писать код, связанный со Spring, чтобы найти бин.

Паттерн Service Locator используется, когда мы хотим найти различные сервисы, используя JNDI. Учитывая высокую стоимость поиска сервисов в JNDI, Service Locator использует кеширование. При запросе сервиса первый раз Service Locator ищет его в JNDI и кэширует объект. Дальнейший поиск этого же сервиса через Service Locator выполняется в кэше, что значительно улучшает производительность приложения.

**Observer-Observable** (Наблюдатель) - Используется в механизме событий ApplicationContext.

Определяет зависимость "один-ко-многим" между объектами, чтобы при изменении состояния одного объекта все его подписчики уведомлялись и обновлялись автоматически.

**Context Object** (Контекстный объект) - Паттерн Context Object, инкапсулирует системные данные в объекте-контексте для совместного использования другими частями приложения без привязки приложения к конкретному протоколу.

**Chain of Responsibility** - это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи. Ему Spring Security.

**Шаблон проектирования Front Controller, как он реализован в Spring?**

Паттерн Front Controller обеспечивает единую точку входа для всех входящих запросов.

Все запросы обрабатываются одним фрагментом кода, который затем может делегировать

ответственность за обработку запроса другим объектам приложения. Он также обеспечивает

интерфейс для общего поведения, такого как безопасность, интернационализация и передача

определенных представлений определенным пользователям.

В Spring в качестве Front Controller выступает DispatcherServlet, все действия проходят

через него. Как правило в приложении задаётся только один DispatcherServlet с маппингом “/”,

который перехватывает все запросы. Это и есть реализация паттерна Front Controller.

Однако иногда необходимо определить два и более DispatcherServlet-а, которые будут

отвечать за свой собственный функционал. Например, чтобы один обрабатывал REST-запросы с

маппингом “/api”, а другой обычные запросы с маппингом “/default”. Spring предоставляет нам

такую возможность, и для начала нужно понять, что:

❖ Spring может иметь несколько контекстов одновременно. Одним из них будет корневой

контекст, а все остальные контексты будут дочерними.

❖ Все дочерние контексты могут получить доступ к бинам, определенным в корневом

контексте, но не наоборот. Корневой контекст не может получить доступ к бинам

дочерних контекстов.

❖ Каждый дочерний контекст внутри себя может переопределить бины из корневого

контекста.

Каждый DispatcherServlet имеет свой дочерний контекст приложения.

**DispatcherServlet** по сути является сервлетом (он расширяет HttpServlet), основной целью которого является обработка входящих веб- запросов, соответствующих настроенному шаблону URL. Он принимает входящий URI и находит правильную комбинацию контроллера и вида. Веб-приложение может определять любое количество DispatcherServlet-ов. Каждый из них будет

работать в своем собственном пространстве имен, загружая свой собственный дочерний

WebApplicationContext с вьюшками, контроллерами и т.д.

Например, когда нам нужно в одном Servlet WebApplicationContext определить обычные контроллеры, а в другом REST-контроллеры.

**WebApplicationContext** расширяет ApplicationContext (создаёт и управляет бинами и т.д.),

но помимо этого он имеет дополнительный метод getServletContext(), через который у него есть

возможность получать доступ к ServletContext-у.

**ContextLoaderListener** создает корневой контекст приложения и будет использоваться всеми дочерними контекстами, созданными всеми DispatcherServlet. Напомню, что корневой контекст приложения будет общим и может быть только один.

**Root WebApplicationContext** содержит компоненты, которые видны всем

дочерним контекстам, такие как сервисы, репозитории, компоненты инфраструктуры и т.д.

После создания корневого контекста приложения он сохраняется в ServletContext как атрибут,

имя которого: WebApplicationContext.class.getName() + ".ROOT"

Чтобы из контроллера любого дочернего контекста обратиться к корневому контексту

приложения, мы можем использовать класс WebApplicationContextUtils, содержащий

статические методы.

**ContextLoaderListener vs DispatcherServlet**

1. ContextLoaderListener создает корневой контекст приложения.
2. Каждый DispatcherServlet создаёт себе один дочерний контекст.
3. Дочерние контексты могут обращаться к бинам, определенным в корневом контексте.
4. Бины в корневом контексте не могут получить доступ к бинам в дочерних контекстах (напрямую).
5. Все контексты добавляются в ServletContext.
6. Мы можем получить доступ к корневому контексту, используя класс WebApplicationContextUtils.

4. SpringBoot. Какие у него преимущества? Как конфигурируется?

Spring Boot - это модуль Spring-а, который предоставляет функцию RAD для среды Spring

(Rapid Application Development - Быстрая разработка приложений). Он обеспечивает более

простой и быстрый способ настройки и запуска как обычных, так и веб-приложений. Он

просматривает наши пути к классам и настроенные нами бины, делает разумные

предположения о том, чего нам не хватает, и добавляет эти элементы.

Spring Boot представляет собой комбинацию Spring Framework и встроенного

контейнера сервлетов и отсутствие (или минимальное наличие) конфигурации приложения.

**Ключевые особенности и преимущества Spring Boot:**

1. **Простота управления зависимостями** (spring-boot-starter-\* в pom.xml).

Чтобы ускорить процесс управления зависимостями Spring Boot неявно

упаковывает необходимые сторонние зависимости для каждого типа приложения на

основе Spring и предоставляет их разработчику в виде так называемых starter-пакетов.

Starter-пакеты представляют собой набор удобных дескрипторов зависимостей,

которые можно включить в свое приложение. Это позволяет получить универсальное

решение для всех технологий, связанных со Spring, избавляя программиста от лишнего

поиска необходимых зависимостей, библиотек и решения вопросов, связанных с

конфликтом версий различных библиотек.

Например, если вы хотите начать использовать Spring Data JPA для доступа к базе

данных, просто включите в свой проект зависимость spring-boot-starter-data-jpa (вам не

придется искать совместимые драйверы баз данных и библиотеки Hibernate). Если вы

хотите создать Spring web-приложение, просто добавьте зависимость spring-boot-starter-

web, которая подтянет в проект все библиотеки, необходимые для разработки Spring

MVC-приложений, таких как spring-webmvc, jackson-json, validation-api и Tomcat.

Другими словами, Spring Boot собирает все общие зависимости и определяет их

в одном месте, что позволяет разработчикам просто их использовать. Также при

использовании Spring Boot, файл pom.xml содержит намного меньше строк, чем в Spring-

приложениях.

2. **Автоматическая конфигурация**.

Автоматическая конфигурация включается аннотацией @EnableAutoConfiguration (входит в состав аннотации @SpringBootApplication).

После выбора необходимых для приложения starter-пакетов Spring Boot

попытается автоматически настроить Spring-приложение на основе выбранных jar-

зависимостей, доступных в classpath классов, свойств в application.properties и т.п.

Например, если добавим spring-boot-starter-web, то Spring boot автоматически

сконфигурирует такие бины как DispatcherServlet, ResourceHandlers, MessageSource итд

Автоматическая конфигурация работает в последнюю очередь, после регистрации пользовательских бинов и всегда отдает им приоритет. Если ваш код уже зарегистрировал бин DataSource — автоконфигурация не будет его переопределять.

3. **Встроенная поддержка сервера приложений/контейнера сервлетов** (Tomcat, Jetty, итд).

Каждое Spring Boot web-приложение включает встроенный web-сервер. Не нужно

беспокоиться о настройке контейнера сервлетов и развертывания приложения в нем.

Теперь приложение может запускаться само как исполняемый .jar-файл с

использованием встроенного сервера.

4. **Готовые к работе функции**, такие как метрики, проверки работоспособности, security и

внешняя конфигурация.

5. **Инструмент CLI** (command-line interface) для разработки и тестирования приложения

Spring Boot.

6. **Минимизация boilerplate кода** (код, который должен быть включен во многих местах

практически без изменений), конфигурации XML и аннотаций.

Как происходит **автоконфигурация в Spring Boot**:

1. Отмечаем main класс аннотацией @SpringBootApplication (аннотация инкапсулирует в себе:

@SpringBootConfiguration, @ComponentScan, @EnableAutoConfiguration), таким образом

наличие @SpringBootApplication включает сканирование компонентов, автоконфигурацию и показывает разным компонентам Spring (например, интеграционным тестам), что это Spring Boot приложение.

@EnableAutoConfiguration импортирует класс EnableAutoConfigurationImportSelector. Этот класс не объявляет бины сам, а использует так называемые фабрики.

1. Класс EnableAutoConfigurationImportSelector смотрит в файл META-INF/spring.factories и

загружает оттуда список значений, которые являются именами классов (авто)конфигураций,

которые Spring Boot импортирует. Т.е. аннотация @EnableAutoConfiguration просто импортирует ВСЕ (более 150) перечисленные в spring.factories конфигурации, чтобы предоставить нужные бины в контекст приложения.

1. Каждая из этих конфигураций пытается сконфигурировать различные аспекты приложения

(web, JPA, AMQP и т.д.), регистрируя нужные бины. Логика при регистрации бинов управляется набором @ConditionalOn\* аннотаций. Можно указать, чтобы бин создавался при наличии класса в classpath (@ConditionalOnClass), наличии существующего бина (@ConditionalOnBean), отсуствии бина (@ConditionalOnMissingBean) и т.п. Таким образом наличие конфигурации не значит, что бин будет создан, и в большинстве случаев конфигурация ничего делать и создавать не будет.

1. Созданный в итоге AnnotationConfigEmbeddedWebApplicationContext ищет в том же DI

контейнере фабрику для запуска embedded servlet container.

1. Servlet container запускается, приложение готово к работе!

4 часть

1. Spring security.

Spring Security обеспечивает всестороннюю поддержку аутентификации, авторизации и

защиты от распространенных эксплойтов. Он также обеспечивает интеграцию с другими

библиотеками, чтобы упростить его использование.

Spring Security - это список фильтров в виде класса FilterChainProxy, интегрированного в

контейнер сервлетов, и в котором есть поле List<SecurityFilterChain>. Каждый фильтр реализует

какой-то механизм безопасности. Важна последовательность фильтров в цепочке.

Когда мы добавляем аннотацию @EnableWebSecurity добавляется DelegatingFilterProxy,

его задача заключается в том, чтобы вызвать цепочку фильтров (FilterChainProxy) из Spring

Security.

В Java-based конфигурации цепочка фильтров создается неявно.

Если мы хотим настроить свою цепочку фильтров, мы можем сделать это, создав класс,

конфигурирующий наше Spring Security приложение, и имплементировав интерфейс

WebSecurityConfigurerAdapter. В данном классе, мы можем переопределить метод:

@Override

protected void configure(HttpSecurity http) throws Exception

{ http .csrf().disable() .authorizeRequests(); }

Именно этот метод конфигурирует цепочку фильтров Spring Security и логика, указанная

в этом методе, настроит цепочку фильтров.

**Основные классы и интерфейсы**

**SecurityContext** - интерфейс, отражающий контекст безопасности для текущего потока.

Является контейнером для объекта типа Authentication. (Аналог - ApplicationContext, в котором

лежат бины).

По умолчанию на каждый поток создается один SecurityContext. SecurityContext-ы

хранятся в SecurityContextHolder.

Имеет только два метода: getAuthentication() и setAuthentication(Authentication

authentication).

**SecurityContextHolder** - это место, где Spring Security хранит информацию о том, кто

аутентифицирован. Класс, хранящий в ThreadLocal SecurityContext-ы для каждого потока, и

содержащий статические методы для работы с SecurityContext-ами, а через них с текущим

объектом Authentication, привязанным к нашему веб-запросу.

**Authentication** - объект, отражающий информацию о текущем пользователе и его

привилегиях. Вся работа Spring Security будет заключаться в том, что различные фильтры и

обработчики будут брать и класть объект Authentication для каждого посетителя.

Кстати объект Authentication можно достать в Spring MVC контроллере командой

SecurityContextHolder.getContext().getAuthentication().

Authentication имеет реализацию по умолчанию - класс UsernamePasswordAuthenticationToken, предназначенный для хранения логина, пароля и коллекции Authorities.

**Principal** - интерфейс из пакета java.security, отражающий учетную запись пользователя.

В терминах логин-пароль это логин. В интерфейсе Authentication есть метод getPrincipal(),

возвращающий Object. При аутентификации с использованием имени пользователя/пароля

Principal реализуется объектом типа UserDetails.

**Credentials** - любой Object; то, что подтверждает учетную запись пользователя, как

правило пароль (отпечатки пальцев, пин - всё это Credentials, а владелец отпечатков и пина -

Principal).

**GrantedAuthority** - полномочия, предоставленные пользователю, например, роли или

уровни доступа.

**UserDetails** - интерфейс, представляющий учетную запись пользователя. Как правило

модель нашего пользователя должна имплементировать его. Она просто хранит пользовательскую информацию в виде логина, пароля и флагов isAccountNonExpired, isAccountNonLocked, isCredentialsNonExpired, isEnabled, а также коллекции прав (ролей) пользователя.

Данная информация позже инкапсулируется в объекты Authentication.

**UserDetailsService** - интерфейс объекта, реализующего загрузку пользовательских

данных из хранилища. Созданный нами объект с этим интерфейсом должен обращаться к БД и

получать оттуда юзеров.

**AuthenticationManager** - основной стратегический интерфейс для аутентификации.

Имеет только один метод, который срабатывает, когда пользователь пытается

аутентифицироваться в системе.

Наиболее часто используемая реализация AuthenticationManager - родной класс

ProviderManager, который содержит поле private List<AuthenticationProvider>

providers со списком AuthenticationProvider-ов и итерирует запрос аутентификации по этому

списку AuthenticationProvider-ов. Идея такого разделения - поддержка различных механизмов

аутентификации на сайтах.

**AuthenticationProvider** - интерфейс объекта, выполняющего аутентификацию. Имеет

массу готовых реализаций. Также можем задать свой тип аутентификации. Как правило в

небольших проектах одна логика аутентификации - по логину и паролю. В проектах побольше

логик может быть несколько: Google-аутентификация и т.д., и для каждой из них создается свой

объект AuthenticationProvider.

AuthenticationProvider немного похож на AuthenticationManager, но у него есть

дополнительный метод, позволяющий вызывающей стороне спрашивать, поддерживает ли он

переданный ему объект Authentication, возможно этот AuthenticationProvider может

поддерживать только аутентификацию по логину и паролю, но не поддерживать Google-

аутентификацию.

**PasswordEncoder** - интерфейс для шифрования/расшифровывания паролей. Одна из

популярных реализаций - BCryptPasswordEncoder.

В случае, если нам необходимо добавить логику при успешной/неудачной

аутентификации, мы можем создать класс и имплементировать интерфейсы

AuthenticationSuccessHandler и AuthenticationFailureHandler соответственно, переопределив их

методы.

**Как это работает с формой логина и UserDetailsService:**

❖ Пользователь вводит в форму и отправляет логин и пароль.

❖ UsernamePasswordAuthenticationFilter создает объект Authentication

* UsernamePasswordAuthenticationToken, где в качестве Principal - логин, а в качестве Credentials - пароль.

❖ Затем UsernamePasswordAuthenticationToken передаёт объект Authentication с логином и паролем AuthenticationManager-у.

❖ AuthenticationManager в виде конкретного класса ProviderManager внутри своего списка объектов AuthenticationProvider, имеющих разные логики аутентификации, пытается аутентифицировать посетителя, вызывая его метод authenticate(). У каждого AuthenticationProvider-а:

➢ Метод authenticate() принимает в качестве аргумента незаполненный объект

Authentication, например только с логином и паролем, полученными в форме

логина на сайте. Затем с помощью UserDetailsService метод идёт в БД и ищет

такого пользователя.

➢ Если такой пользователь есть в БД, AuthenticationProvider получает его из базы

в виде объекта UserDetails. Объект Authentication заполняется данными из

UserDetails - в него включаются Authorities, а в Principal записывается сам объект

UserDetails, содержащий пользователя.

➢ Затем этот метод возвращает заполненный объект Authentication

аутентификацию). Вызывается AuthenticationSuccessHandler.

➢ Если логин либо пароль неверные, то выбрасывается исключение. Вызывается

AuthenticationFailureHandler.

❖ Затем этот объект Authentication передается в AccessDecisionManager и получаем

решение на получение доступа к запрашиваемой странице (проходим авторизацию).

Как подключить Spring Security к проекту?

- нужно добавить зависимости:

spring-security-web,

spring-security-config,

thymeleaf-extras-springsecurity5

- создать public class SpringSecurityInitializer extends AbstractSecurityWebApplicationInitializer

- создать конфигурационный файл @Configuration @EnableWebSecuritypublic class SecurityConfig extends WebSecurityConfigurerAdapter в котором переопределить метод .configure(), где указывается конфигурация логина/логаута (можно указать по какому адресу находится форма логина, по какому адресу происходит обработка запроса логина, логику обработки при логине(хэндлеры), названия полей с формы логина).

Настраиваются параметры доступа по определенным урл, для ролей и авторити.

2. Авторизация, аутентификация.

**Авторизация** - предоставить разным пользователям разный доступ в зависимости от их

полномочий (Authorities) или ролей (Roles), нужна для того, чтобы ограничить доступ к

некоторым частям приложения.

**Аутентификация** - процедура проверки подлинности, то, как мы проверяем личность

того, кто пытается получить доступ к определенному ресурсу.

*Аутентификация* — это проверка, что пользователь есть тот, за кого себя выдает. Чтобы выполнить проверку, надо:

* Извлечь имя и пароль из HTTP-запроса. За это отвечает *UsernamePasswordAuthenticationFilter* (конкретно в нашем приложении с Form-Based аутентификацией).
* Сравнить их с реальными именем и паролем, хранящимся где-то (в базе, на LDAP-сервере, во временной памяти приложения и т.д. где угодно). Это делает *AuthenticationManager* в методе *authenticate()*.  
  Вызывается *authenticate()* из фильтра *UsernamePasswordAuthenticationFilter сразу* после извлечения имени/пароля из HTTP-запроса.

Метод *authenticate(Authentication authentication)* интерфейса.

*AuthenticationManager —* проверка пароля. Как видно в коде выше, метод *authenticate()* получает на вход объект *Authentication* с именем и паролем, полученными от клиента и требующими проверку. Имя хранится в *principal*, а пароль в *credenticals* (до проверки, после проверки будет иначе). Если аутентификация не прошла (имя и пароль неверны), то выбрасывается исключение *BadCredentials*.В случае же успеха возвращается тоже объект *Authentication,* но заполненный по-другому.

После аутентификации в поле *Principal* объекта *Authentication* будет реальный пользователь в виде *UserDetails.* При этом поле Credentials обнуляется, а isAuthenticated() меняется с false на true. То есть имя и пароль перемещаются объект *Principal.* Как же AuthenticationManager в authenticate() решает, правильный пароль, или нет? Очевидно, для этого надо сравнить переданный пароль с реальным. А для этого по переданному имени надо извлечь реального пользователя. И вот тут дальнейшее зависит от того, где этот пользователь хранится.

**Типы аутентификации в Spring Security**

Есть несколько стандартных типов хранения и извлечения пользователя, и за каждый из них отвечает свой *AuthenticationProvider. AuthenticationManager* делегирует провайдеру извлечь данные их хранилища. В Spring Security реализованы несколько стандартных провайдеров, все они задаются в методе *configure().*

Итак, провайдеры:

* In-Memory
* Jdbc
* LDAP

Все упирается в получение реального пользователя по его имени — в метод *loadUserByName* интерфейса *UserDetailsService.* Где UserDetails содержит информацию о реальном пользователе (точнее, нам надо составить эту информацию из того, что есть в базе, например).

И составляем мы ее как раз в методе loadUserByUsername(), если реализовываем его вручную. Важно заполнить password, username и authorities (права) объекта UserDetails .

SecurityContext — хранилище объекта Authentication.

Допустим, аутентификация прошла успешно — это значит, имя и пароль верные.

Тогда объект *Authentication* сохраняется в *SecurityContext*, а тот, в свою очередь, — в *SecurityContextHolder.* Таким образом, SecurityContext используется для хранения объекта Authentication.

Аутентификация в нашем примере происходит только раз. Коль скоро она прошла успешно, a*uthentication* восстанавливается из контекста, а в итоге из сессии при последующих запросах. Происходит это в *SecurityContextPersistenceFilter*.

Сессии в нашем примере включены — они включены по умолчанию, если их специально не отключить. То есть после аутентификации в HTTP-ответе клиенту отправляется уникальный *JSESSIONID*, который он отправляет во всех последующих запросах. По *JSESSIONID*  восстанавливается сессия, из нее берется *SecurityContext*, а из него *Authentication*.

3. Principal, Authorities, Authentication.

**Principal** - это текущий вошедший в систему пользователь. Привязан к текущему потоку,

запросу и сеансу. Фактически принципалом пользователя может выступать его логин,

пароль, или роль и т.д.

**Authorities** - представляют полномочия(разрешения), которые были предоставлены

принципалу (или например роли).

**Authentication** - объект Authentication с именем и паролем, полученными от клиента и

требующими проверку. Имя хранится в principal, а пароль в credenticals (до проверки,

после проверки будет иначе).

После аутентификации в поле Principal объекта.

Authentication будет реальный пользователь в виде UserDetails, поле Credentials обнуляется, то есть имя и пароль перемещаются объект Principal.

**SecurityContextHolder** Используется для хранения контекста безопасности (security context). Кто является действующим в данный момент пользователем, прошел ли он проверку подлинности, какие полномочия у него есть.

Spring Security автоматически связывает информацию аутентификации с текущим потоком, когда пользователь входит в систему, и автоматически очищает информацию аутентификации текущего потока, когда пользователь выходит из системы.

* getAuthentication () возвращает информацию об аутентификации,
* getAuthorities (), список информации об авторизации, по умолчанию использует некоторые классы реализации интерфейса GrantedAuthority, обычно ряд строк, представляющих информацию об авторизации.
* getCredentials (), информация о пароле и строка пароля, введенная пользователем, обычно удаляются после аутентификации для обеспечения безопасности.
* getDetails (), детали, интерфейс реализации в веб-приложении обычно WebAuthenticationDetails, который записывает IP-адрес посетителя и значение sessionId.
* getPrincipal (), возвращает информацию об идентификаторе. Наиболее важная информация об идентичности в большинстве случаев возвращает класс реализации интерфейса.

Аутентификация - это интерфейс в весеннем пакете безопасности, напрямую унаследованный от класса Principal, и Principal находится в **java.security**.

Аутентификация - это высший уровень абстракции идентификации / аутентификации в весенней безопасности. Из этого интерфейса верхнего уровня мы можем получить список информации о правах доступа, принадлежащей пользователю, пароль, данные пользователя, информацию об идентификаторе пользователя и информацию об аутентификации.

1. Имя пользователя и пароль получены фильтром и инкапсулированы в **Authentication** И обычно это **UsernamePasswordAuthenticationToken** Это класс реализации.
2. **AuthenticationManager** Менеджер по идентификации отвечает за проверку этого **Authentication**
3. После успешной аутентификации,**AuthenticationManager** Диспетчер идентификации возвращает сообщение, заполненное информацией (включая информацию о разрешениях, информацию об идентификации и подробную информацию, упомянутую выше, но пароль обычно удаляется) **Authentication** Instance.
4. **SecurityContextHolder** Контейнер контекста безопасности будет заполнен информацией на шаге 3.**Authentication**, Через SecurityContextHolder.getContext (). SetAuthentication (...) метод, установите его.

Это абстрактный процесс аутентификации, и во всем процессе, если вы не запутались в деталях, остается только один **AuthenticationManager**.

Общий класс реализации интерфейса AuthenticationManager

**ProviderManager** - **List<AuthenticationProvider>** Список, хранит несколько методов аутентификации. Фактически это приложение режима делегата (Delegate). Другими словами, всегда есть только одна основная запись аутентификации: AuthenticationManager, разные методы аутентификации: Имя пользователя + пароль (UsernamePasswordAuthenticationToken), почтовый ящик + пароль, номер мобильного телефона + пароль для входа соответствуют трем провайдерам аутентификации.

UserDetails, который также является одним из наиболее часто используемых интерфейсов в платформе. UserDetails - это интерфейс, в который Spring инкапсулирует информацию об идентификаторе. Этот интерфейс охватывает некоторые необходимые поля информации о пользователе, и конкретный класс реализации расширяет его.

Он очень похож на интерфейс аутентификации: например, все они имеют имя пользователя и полномочия, и разграничение их также является одним из ключевых элементов этой статьи.

Аутентификационные getCredentials () и getPassword () в UserDetails должны обрабатываться по-разному: первый - это учетные данные, предоставленные пользователем, а второй - правильный пароль пользователя. Аутентификатор - это фактически сравнение двух. GetAuthorities () в Authentication фактически формируется getAuthorities () из UserDetails. Помните метод getUserDetails () в интерфейсе аутентификации? UserDetails заполняется после прохождения через AuthenticationProvider.

4. InMemoryAuthentication vs basicAuthentication?

**BasicAuthentication** это механизм чтения имени пользователя и пароля, а In-Memory Authentication - это механизм хранения.

**InMemoryAuthentication** - означает, что пользователь хранится не в базе, а в

оперативной памяти приложения до тех пор, пока оно запущено. И чтобы

отредактировать пользователя, придется заново запускать приложение. Этот вариант

полезен для экспериментов во время разработки.

**basicAuthentication** - аутентификация путем сравнения пары идентификатор/пароль,

зашифрованные с использованием кодировки base64 с теми, что хранятся в системе.

Чтобы получить доступ к ресурсам (Resource), защищенные Basic Authentication,

пользователь должен отправить request и в том request должна содержаться информация

username/password прикрепленная на Header.

5. Секьюрность к контроллеру? (минимум 2 способа).

- через antMather

- @EnableGlobalMethodSecurity над файлом конфигурации, @Secured(value="ADMIN") над методом в контроллере.

- @EnableGlobalMethodSecurity(prePostEnabled = true)

@PreAuthorize("hasAuthority('ROLE\_ADMIN')")

@PreAuthorize("isAuthenticated()") // в скобках SpEL выражение

* @**Secured** используется для указания списка ролей в методе. Таким образом,

пользователь может получить доступ к этому методу только в том случае, если у него

есть хотя бы одна из указанных ролей.

* @**RoleAllowed** является аннотацией, эквивалентной аннотации @Secured в JSR-250

( Запрос Спецификации Java целью которого является разработка аннотаций ).

* @PreAuthorize проверяет данное выражение перед вводом метода, в то время как

аннотация

* @PostAuthorize проверяет его после выполнения метода и может изменить

результат.

* Аннотации @**PreFilter** и @**PostFilter** используются для фильтрации списков объектов на

основе определяемых нами пользовательских правил безопасности.

@PostFilter определяет правило для фильтрации списка возврата метода, применяя это

правило к каждому элементу в списке . Если оцененное значение истинно, элемент будет

сохранен в списке. В противном случае элемент будет удален.

@PreFilter работает очень похожим образом, однако фильтрация применяется к списку,

который передается в качестве входного параметра аннотированному методу.

**Другие виды аутентификации:**

* Тoken Authentication (Проверка подлинности токена. Токен выдается сервером.)
* Digest authentication(Дайджест-аутентификация. Данный метод отправляет по сети хеш-

сумму логина, пароля, адреса сервера и случайных данных, и предоставляет больший

уровень защиты, чем базовая аутентификация, при которой данные отправляются в

открытом виде.)

* Digital Signature- Пример: флешка, при поступлении на кату.)
* OAuth2 authorization(Авторизация OAuth2. Схема авторизации, обеспечивающая

предоставление третьей стороне ограниченный доступ к защищенным ресурсам

пользователя без передачи ей (третьей стороне) логина и пароля)

6. Many-to-many vs one-to-one

@**OneToOne**

Для связи один к одному в обоих классах к соответствующим полям добавляется

аннотация @OneToOne. Параметр optional говорит JPA, является ли значение в этом

поле обязательным или нет. Связанное поле в User объявлено с помощью аннотации

@JoinColumn, параметр name которой обозначает поле в БД для создания связи. Для

того, чтобы объявить сторону, которая не несет ответственности за отношения,

используется атрибут mappedBy

Со стороны владельца к аннотации @OneToOne добавляется параметр cascade. В

однонаправленных отношениях одна из сторон (и только одна) должна быть владельцем

и нести ответственность за обновление связанных полей. Каскадирование позволяет

указать JPA, что необходимо «сделать со связанным объектом при выполнении операции

с владельцем».

@**ManyToMany**

В случае отношения «многие ко многим» обе стороны могут относиться к нескольким

экземплярам другой стороны.

Моделирование отношений «многие ко многим» с помощью POJO очень просто. Мы

должны включить в оба класса коллекцию , которая содержит элементы других.

После этого нам нужно пометить класс @Entity и первичный ключ @Id , чтобы сделать их

правильными объектами JPA.

Кроме того, мы должны настроить тип отношения чтобы Hibernate понял, что мы хотим

создать именно двустороннее отношение нам нужно указать, какая из сторон является

владельцем отношений, а какая является обратной стороной. Это делается при помощи

атрибута mappedBy. Важно отметить, что указывается этот атрибут в аннотации, которая

находится на противоположной стороне отношения.

Для отношения многие ко многим любая из сторон может быть владельцем. В случае с

ролями и пользователями выберем сущность пользователя в качестве владельца.

В зависимости от cascade будет либо учитываться связь между сущностями для

выполнения операции, либо нет.

**cascade**=ALL - на все операции

cascade={PERSIST, MERGE, REMOVE, REFRESH, DETACH} - на определенные операции,

те что есть в перечислении будут учитываться.

**FetchType** - Определение стратегии для извлечения данных из базы данных.

EAGER - стратегия, которая предусматривает получение полной связи между

сущностями, и последующих обращениях к связям не будет выполнять запрос на

получение данных, так как данные изначально были получены полностью.

LAZY - стратегия, которая не предусматривает получение полной связи сущностей, и при

первом обращении к связи будет выполняться запрос на получение данных с БД.

7. Каскады для таблиц и какие они бывают.

Каскады позволяют нам настроить наши связи между сущностями. Если быть

конкретнее, то мы можем выстроить зависимость между сущностями. Одна из

сущностей будет выступать в роли стержневой, а другая в роли ассоциации.

CascadeType.ALL означает, что необходимо выполнять каскадно сразу все

операции:

CascadeType.PERSIST -создание новой сущности, которая ещё не была

представлена в БД.

CascadeType.MERGE - слияние.

CascadeType.REMOVE - удаление.

CascadeType.REFRESH - обновление.

CascadeType.DETACH - отсоединить сущности друг от друга.

8. Bootstrap.

Свободный набор инструментов для создания сайтов и веб-приложений. Включает в себя

HTML- и CSS-шаблоны оформления для типографики, веб-форм, кнопок, меток, блоков

навигации и прочих компонентов веб-интерфейса, включая JavaScript-расширения.

**Основные инструменты Bootstrap:**

Сетки — заранее заданные размеры колонок, которые можно сразу же использовать,

например, ширина колонки 140 px относится к классу .span2 (.col-md-2 в третьей версии

фреймворка), который можно использовать в CSS-описании документа.

Шаблоны — фиксированный или резиновый шаблон документа.

Типографика — описания шрифтов, определение некоторых классов для шрифтов, таких как

код, цитаты и т. п.

Медиа — предоставляет некоторое управление изображениями и видео.

Таблицы — средства оформления таблиц, вплоть до добавления функциональности

сортировки.

Формы — классы для оформления форм и некоторых событий, происходящих с ними.

Навигация — классы оформления для панелей, вкладок, перехода по страницам, меню и

панели инструментов.

Алерты — оформление диалоговых окон, подсказок и всплывающих окон.

9. Rest-сервисы. Их преимущества и недостатки

REST API (Representational State Transfer API) - архитектурный стиль взаимодействия компонентов распределенного веб-приложения. В широком смысле, компоненты в REST взаимодействуют в сети Интернет в качестве клиентов и серверов. Грубо говоря REST - набор соглашений взаимодействия с HTTP. Определяет стиль взаимодействия между разными компонентами системы.

REST означает передачу репрезентативного состояния. REST — это архитектурный стиль для веб-разработки. Архитектура REST излагает рекомендации по передаче представлений ресурсов между клиентами и серверами в сети.

REST API или RESTful API — это веб-API, соответствующий стилю архитектуры REST.

Настоящий RESTful API должен соответствовать пяти архитектурным ограничениям REST:

* Единый интерфейс - необходим для отделения клиента от сервера.
* Клиент-сервер
* Без гражданства - Отсутствие состояния означает, что клиент и сервер не хранят информацию о состоянии друг друга. Поскольку сервер не хранит никакой информации, он обрабатывает каждый запрос клиента как новый запрос.
* Кэшируемый
* Многоуровневая система

Код по запросу — это необязательное ограничение архитектуры RESTful. Код по запросу позволяет серверу отправлять исполняемый код (скрипты или апплеты) клиенту по запросу клиента.

### **Расскажите о методах HTTP-запросов, поддерживаемых REST, и о том, когда они используются.**

REST API основаны на HTTP-запросах или командах, каждая из которых выполняет свою задачу.

REST поддерживает следующие HTTP-запросы:

* Метод GET : запрос данных с сервера.
* Метод POST : отправьте данные для создания нового ресурса по указанному сервером URL-адресу.
* Метод PUT : отправьте данные для создания нового ресурса по указанному клиентом URL-адресу.
* Метод DELETE : удалить ресурс с сервера.
* Метод OPTIONS : возвращать методы запроса, поддерживаемые службой.
* Метод HEAD : возвращает метаинформацию, такую ​​как заголовки ответа.
* Метод PATCH : изменить часть ресурса на сервере.

### **В чем разница между методами PUT и POST?**

Этот вопрос может поставить некоторых разработчиков в тупик. Способность объяснить это поможет вам выделиться как человеку, который действительно знает, о чем говорит.

Вот различия между PUT и POST:

**PUT**:

* Идемпотент (т. е. несколько запросов дадут один и тот же результат).
* Ответы PUT не кэшируются.
* Обновляет или заменяет целевой ресурс полезной нагрузкой запроса.

**POST**:

* Не идемпотент (т. е. несколько запросов дадут несколько одинаковых результатов).
* Ответы POST могут кэшироваться при условии правильного заголовка управления кешем.
* Полезная нагрузка запроса обрабатывается веб-сервером на основе целевого ресурса.

Понимание идемпотентности. Примером идемпотентной операции может быть операция умножения числа на единицу. Независимо от того, сколько раз вы умножаете пять на один, вы получите тот же результат.

### **В чем разница между AJAX и REST?**

Клиент AJAX может сделать RESTful-запрос к REST API (например, запрос на получение), но AJAX не является архитектурным стилем. Это метод веб-разработки для клиентских приложений. Клиенты AJAX могут получить доступ к REST API, но они не реализованы с помощью AJAX.

AJAX – технология обращения к серверу без перезагрузки страницы.

### **В чем разница между SOAP и REST?**

Хотя некоторые REST API используют протоколы SOAP, REST и SOAP — это совершенно разные подходы к созданию API. Интервьюеры могут задать этот вопрос, чтобы оценить глубину вашего понимания.

Вот некоторые различия между SOAP и REST.

SOAP:

* Протокол
* Формат данных ограничен XML
* Тяжелый и требует большей пропускной способности
* Звонки не могут кэшироваться

REST:

* Архитектурный стиль
* Позволяет использовать различные форматы данных, включая обычный текст, HTML, XML, JSON и YAML.
* Легкий и требует меньше пропускной способности
* Звонки могут кэшироваться
* Объясните коды состояния ответа HTTP.

@Controller - это обычная аннотация, которая используется для пометки класса

как Spring MVC Controller, тогда как @RestController — это специальный контроллер, используемый в веб-сервисах RESTFul, и эквивалент @Controller + @ResponseBody .

Аннотация @ResponseBody ставится на методы, которые работают с данными, а не с моделями. Ее не требуется указывать явно, если используется @RestController.

Обычные методы возвращают Model, а методы аннотированные @ResponseBody возвращают объекты, которые конвертируются в медиа-файлы с помощью HttpMessageConverter.

Если разрабатывается публичное API и логика взаимодействия во многом покрывается четверкой методов CRUD - выбирается REST. Он наиболее популярен в WEB. Яндекс, Google и другие используют именно его для своего API.

Representational - ресурсы в REST могут быть представлены в любой форме - JSON, XML, текст, или даже HTML- зависит от того, какие данные больше подходят потребителю.

State - при работе с REST вы должны быть сконцентрированы на состоянии ресурса, а не на действиях с ресурсом.

Transfer - REST включает себя передачу ресурсных данных, в любой представленной форме, от одного приложения другому.

**Преимущества**:

* Масштабируемость
* Простота использования
* Поддержка многих форматов (JSON, XML, YAML и т.д.)
* Разделение клиента и сервера
* Может быть закеширован, экономичность ресурсов
* Предназначен для вызова клиентским приложением
* Мультиязычность

**Недостатки**:

* Полный ответ, включая метаданные
* Необходимость единого шаблона запроса под ответ
* Не применяет методы безопасности
* Метод HTTP ограничивает вас синхронными запросами
* Из-за отсутствия состояния вы не сможете поддерживать состояние (например, в сеансах).
* Отсутствие спецификации;
* Неоднозначность методов управления данными.

10. Форматы данных в REST-сервисах

Задавать то в каком виде будут возвращаться форматы на клиент можно несколькими

способами. На самом деле в REST считается что один и тот же ресурс может быть в нескольких представлениях. Например один и тот же запрос а один и тот же URI может возвращать данные в разных форматах. Самым распространённым вариантов выбор является выбор между XML и JSON.

Например GET запрос на URI http://myadressbook.com/contacts/25 /может вернуть нам XML, JSON, vCard или например изображение. Чаще всего управляют тем какие данные возвращаются с помощью HTTP Header - Accept. В качестве параметров для Accept заголовка используется один из MIME типов либо свой кастомный MIME тип (Multipurpose Internet Mail Extensions - многоцелевые расширения интернет-почты).

Иногда тип возвращаемых данных задают посредством URI, например:

http://myadressbook.com/contacts/25?return=json

http://myadressbook.com/contacts/25.json

Данные которые идут в сообщении на сервер тоже могут быть разных форматов и сервер

вполне себе их может понимать. Аналогично формат данных которые идут на сервер

задается заголовком Content-Type.

11. responseBody, requestBody, ResponseEntity

@**ResponseBody** дает фреймворку понять, что объект, который вы вернули из

метода надо прогнать через HttpMessageConverter, чтобы получить готовое к

отправке на клиент представление.

@**RequestBody** используется для возврата и записи объекта прямо в тело

HTTP-ответа, в соответствии с заголовком HTTP-запроса.

**ResponseEntityResponseEntityResponseEntityResponseEntity** - специальный класс для возврата ответов. С помощью него

можно вернуть клиенту HTTP статус код.

Аннотация @**RestController** была введена в Spring 4.0 для упрощения создания RESTful

веб-сервисов. Это удобная аннотация, которая объединяет @Controller и @ResponseBody, что

устраняет необходимость аннотировать каждый метод обработки запросов аннотацией

@**ResponseBody** сообщает контроллеру, что возвращаемый объект автоматически

сериализуется в json или xml и передается обратно в объект HttpResponse. Контроллер

использует Jackson message converter для конвертации входящих/исходящих данных. Как

правило целевые данные представлены в json или xml.

@ResponseBody сообщает контроллеру, что возвращаемый объект автоматически сериализуется в JSON и передается обратно в объект HttpResponse.

**ResponseEntity**

Данный класс используется для формирования ответа HTTP с пользовательскими

параметрами (заголовки, код статуса и тело ответа). ResponseEntity необходим, только если мы

хотим кастомизировать ответ. Во всех остальных случаях достаточно использовать.

@ResponseEntity необходим, только если мы хотим кастомизировать ответ, добавив к нему статус ответа (1хх - информационные, 2хх - успешные, 3хх - перенаправления, 4хх - ошибки клиента, 5хх - ошибки сервера). Во всех остальных случаях будем использовать @ResponseBody.

Если мы хотим использовать ResponseEntity, то просто должны вернуть его из метода,

Spring позаботится обо всем остальном.

Если клиент ждет от нас JSON/XML, мы можем параметризовать ResponseEntity

конкретным классом и добавить к ответу заголовки и Http статус.

12. AJAX/fetch?

**AJAX** означает асинхронный JavaScript и XML и позволяет асинхронно обновлять веб-

страницы, обмениваясь данными с веб-сервером во время работы приложения. Короче

говоря, это означает, что можно обновлять части веб-страницы без перезагрузки всей

страницы (URL-адрес остается прежним).

Метод **fetch**() - это XMLHttpRequest нового поколения. Он предоставляет улучшенный

интерфейс для осуществления запросов к серверу: как по части возможностей и контроля

над происходящим, так и по синтаксису, так как построен на промисах (объект,

представляющий результат успешного или неудачного завершения асинхронной

операции.).

13. RestController vs Controller

@Controller: эта аннотация - всего лишь специализированная версия @Component, и она

позволяет автоматически обнаруживать классы контроллеров на основе сканирования classpath.

@RestController: эта аннотация является специализированной версией @Controller, которая

автоматически добавляет @Controller и @ResponseBody аннотации, поэтому нам не нужно

добавлять @ResponseBody к нашим методам отображения.

Основная разница между традиционным Spring MVC контроллером и RESTfull веб-сервис

контроллером заключается в способе создания тела HTTP ответа. MVC контроллер опирается на

технологию View, а RESTfull веб сервис контроллер возвращает объект, который представляется

в HTTP ответе в виде JSON или XML. @RestController превращает помеченный класс в Spring-

бин. Этот бин для конвертации входящих/исходящих данных использует Jackson message

converter.

@Controller - Возвращает страницу и данные при помощи шаблонизаторов JSP вроде Thymeleaf.

@RestController - Возвращает данные. Используется с JSON или XML. Позволяет применять технологии без перезагрузки веб страниц, на основе JS.

Для того, чтобы настроить @RestController на возвращение JSON или XML используются Jackson зависимости.

При отправке данных с фронта в рестконтроллер нужно в заголовке(header) указать тип возвращаемого/отправляемого тела запроса.

Например,ContentType или accept: "application/json"

14. RestTemplate и его методы

Класс RestTemplate это специальный клиент в Spring для отправки http-запросов (что-то

вроде встроенной программы POSTMAN). Он предоставляет удобные API для легкого

вызова конечных точек (эндпойнтов) REST'а в одну строку.

RestTemplate restTemplate = new RestTemplate();

String fooResourceUrl = "http://localhost:8080/spring-rest/foos";

ResponseEntity response = restTemplate.getForEntity(fooResourceUrl + "/1", String.class);

**Методы**:

1 delete () Удалить ресурсы по указанному URI.

2 exchange () Выполнить определенный HTTP-метод для URL-адреса и вернуть ResponseEntity,

содержащий объект, который взят из тела ответа

3 execute () Выполнить определенный HTTP-метод для URL-адреса и вернуть объект,

отображенный из тела ответа

4 doExecute() Выполните данный метод на предоставленном URI.

5 getForEntity () отправляет HTTP-запрос GET, а возвращенный ResponseEntity содержит объект,

отображенный в теле ответа

6 postForEntity () POST-данные в URL-адрес и возвращение ResponseEntity, содержащего объект,

который отображается из тела ответа

7. getForObject () отправляет HTTP-запрос GET, и возвращаемое тело запроса будет сопоставлено

с объектом

8 postForObject () Отправьте данные POST в URL и верните объект, сформированный в

соответствии с соответствием тела ответа

9. patchForObject () Обновите ресурс, вставив данный объект в шаблон URI, и верните

представление, найденное в ответе.

10 headForHeaders () отправляет HTTP-запрос HEAD и возвращает HTTP-заголовок,

содержащий URL-адрес определенного ресурса.

11. optionsForAllow () отправляет запрос HTTP OPTIONS и возвращает информацию заголовка

Allow для определенного URL

12. postForLocation () POST-данные в URL-адрес и возвращение URL-адреса вновь созданного

ресурса

13. put () Создайте или обновите ресурс, поместив данный объект в URI.

14. httpEntityCallback () Возвратите RequestCallback реализацию, которая записывает данный

объект в поток запроса.

15 acceptHeaderRequestCallback() Возвращает RequestCallback, который устанавливает Accept

заголовок запроса на основе заданного типа ответа, перекрестно сверяясь с настроенными

преобразователями сообщений.

16. handleResponse () Обработайте данный ответ, выполняя соответствующую регистрацию и

вызывая, ResponseErrorHandler если необходимо.

17. getErrorHandler() Возвращает обработчик ошибок

18. setErrorHandler() Установите обработчик ошибок.

19. setDefaultUriVariables() Настройте значения переменных URI по умолчанию.

20 headersExtractor() Вернуть экстрактор ответа для HttpHeaders.

21. responseEntityExtractor() Вернуть a ResponseExtractor, который готовит файл ResponseEntity.

22. getUriTemplateHandler() Возврат настроенного обработчика шаблона URI.

23. setUriTemplateHandler() Настройте стратегию расширения шаблонов URI.

24. getMessageConverters() Возвращает список преобразователей тела сообщения.

25. setMessageConverters() Установить преобразователи тела сообщения для использования

15. HTTP протокол

HTTP — широко распространённый протокол передачи данных, изначально предназначенный для

передачи гипертекстовых документов (то есть документов, которые могут содержать ссылки,

позволяющие организовать переход к другим документам).

Задача, которая решается с помощью протокола HTTP — обмен данными между

пользовательским приложением, осуществляющим доступ к веб-ресурсам (обычно это веб-

браузер) и веб-сервером. На данный момент именно благодаря протоколу HTTP обеспечивается

работа Всемирной паутины.

Сам по себе протокол HTTP не предполагает использование шифрования для передачи

информации. Тем не менее, для HTTP есть распространённое расширение, которое реализует

упаковку передаваемых данных в криптографический протокол SSL или TLS.

Название этого расширения — HTTPS (HyperText Transfer Protocol Secure). Для HTTPS-соединений обычно используется TCP-порт 443 HTTPS широко используется для защиты информации от перехвата, обеспечивает защиту от атак вида man-in-the-middle — в том случае, если сертификат проверяется на клиенте, и при этом приватный ключ сертификата не был

скомпрометирован, пользователь не подтверждал использование неподписанного сертификата, и на компьютере пользователя не были внедрены сертификаты центра сертификации злоумышленника.

На данный момент HTTPS поддерживается всеми популярными веб-браузерами.

Методы:

GET /books/ - получает список всех книг. Как правило, это упрощенный список,

т.е. содержащий только поля идентификатора и названия объекта, без

остальных данных.

GET /books/{id} - получает полную информацию о книге.

POST /books/ - создает новую книгу. Данные передаются в теле запроса.

PUT /books/{id} - изменяет данные о книге с идентификатором {id}, возможно

заменяет их. Данные также передаются в теле запроса.

OPTIONS /books - получает список поддерживаемых операций для указанного

ресурса (практически не используется)

DELETE /books/{id} - удаляет данные с идентификатором {id}.

PUT /books/{id} - изменяет данные с идентификатором {id}.

**Безопасный** запрос - это запрос, который не меняет состояние приложения.

**Идемпотентный** запрос - это запрос, эффект которого от многократного выполнения равен эффекту от однократного выполнения.

Безопасный / Идемпотентный:

get + / +

put - / +

delete - / +

post - / -

options + / +

1хх - INFORMATIONAL - Запрос был получен и процесс продолжается

2хх - SUCCESS - Запрос был успешно получен, понят и принят.

3хх - REDIRECTION - Для выполнения запроса необходимо предпринять

дальнейшие действия

4хх - CLIENT ERROR - Запрос содержит неверный синтаксис или не может

быть выполнен.

5хх - SERVER ERROR - Серверу не удалось выполнить корректный запрос.

Основные:

200 - OK - успешный запрос. Если клиентом были запрошены какие-либо

данные, то они находятся в заголовке и/или теле сообщения.

201 - OK - в результате успешного выполнения запроса был создан новый

ресурс.

204 - OK - ресурс успешно удалён.

304 - Not Modified - клиент может использовать данные из кэша.

400 - Bad Request - запрос невалидный или не может быть обработан.

401 - Unauthorized - запрос требует аутентификации пользователя.

403 - Forbidden - сервер понял запрос, но отказывается его обработать или

доступ запрещен.

404 - Not found - ресурс не найден.

500 - Internal Server Error -внутренняя ошибка сервера.

16. Можно ли передать в GET-запросе один и тот же параметр несколько раз? Как?

Да, можно принять все значения, используя массив в методе контроллера:

http://localhost:8080/login?name=Ranga&name=Ravi&name=Sathish

public String method(@RequestParam(value="name") String[] names){...}

или

http://localhost:8080/api/foos?id=1,2,3

----

IDs are [1,2,3]

@GetMapping("/api/foos")

@ResponseBody

public String getFoos(@RequestParam List<String> id) {

return "IDs are " + id;

}ИдемпотентныйИдемпотентныйИдемпотентныйИдемпотентный

5 часть - дополнительно

1. **SpringData**

Spring Data — дополнительный удобный механизм для взаимодействия с сущностями базы данных, организации их в репозитории, извлечение данных, изменение, в каких то случаях для этого будет достаточно объявить интерфейс и метод в нем, без имплементации.

Основное понятие в Spring Data — это репозиторий. Это несколько интерфейсов которые используют JPA Entity для взаимодействия с ней.

Так например интерфейс

public interface CrudRepository<T, ID extends Serializable> extends Repository<T, ID>

обеспечивает основные операции по поиску, сохранения, удалению данных (CRUD операции), так же есть PagingAndSortingRepository, JpaRepository.

Создание нативного запроса в SpringData: использование аннотаций @Modifying, @Transactional, и в @Query пишем наш запрос.

2. Нововведение Spring 5

* Используется JDK 8+ (Optional, CompletableFuture, Time API, java.util.function, default methods)
* Поддержка Java 9 (Automatic-Module-Name in 5.0, module-info in 6.0+, ASM 6)
* Поддержка HTTP/2 (TLS, Push), NIO/NIO.2, Kotlin
* Поддержка Kotlin
* Реактивность (Web on Reactive Stack)
* Null-safety аннотации(@Nullable), новая документация
* Совместимость с Java EE 8 (Servlet 4.0, Bean Validation 2.0, JPA 2.2, JSON Binding API 1.0)
* Поддержка JUnit 5 + Testing Improvements (conditional and concurrent)
* Удалена поддержка: Portlet, Velocity, JasperReports, XMLBeans, JDO, Guava

3. Spring Cloud (Data Flow).

Это инструменты для создания сложных топологий для потоковой и пакетной передачи данных.

Spring Cloud предоставляет разработчикам инструменты для быстрого создания некоторых распространенных шаблонов в распределенных системах (например, управление конфигурацией, обнаружение сервисов, автоматические выключатели, интеллектуальная маршрутизация, микропрокси, шина управления, одноразовые токены, глобальные блокировки, выборы руководства, распределенные сеансы, состояние кластера). Координация распределенных систем приводит к стандартным шаблонам, и с помощью Spring Cloud разработчики могут быстро создавать сервисы и приложения, реализующие эти шаблоны. Они будут хорошо работать в любой распределенной среде, включая собственный ноутбук разработчика, центры обработки данных без операционной системы и управляемые платформы, такие как Cloud Foundry.

Spring Cloud Data Flow предоставляет инструменты для создания сложных топологий для потоковых и пакетных конвейеров данных. Конвейеры данных состоят из приложений [Spring Boot](https://projects.spring.io/spring-boot/) , созданных с использованием платформ микросервисов [Spring Cloud Stream](https://cloud.spring.io/spring-cloud-stream) или [Spring Cloud Task .](https://cloud.spring.io/spring-cloud-task/)

Spring Cloud Data Flow поддерживает ряд вариантов использования обработки данных, от ETL до импорта/экспорта, потоковой передачи событий и прогнозной аналитики.

Используется для микросервисов. См.док по микросервисам.

<https://spring.io/projects/spring-cloud-dataflow>

4. Spring Integration.

Spring Integration обеспечивает легкий обмен сообщениями в приложениях на базе Spring и поддерживает интеграцию с внешними системами через декларативные адаптеры. Эти адаптеры обеспечивают более высокий уровень абстракции по сравнению с поддержкой Spring для удаленного взаимодействия, обмена сообщениями и планирования. Основная цель Spring Integration - предоставить простую модель для построения корпоративных решений по интеграции, сохраняя при этом разделение задач, что важно для создания поддерживаемого, тестируемого кода.

<https://spring.io/projects/spring-integration>

5. Spring Batch.

Spring Batch предоставляет многократно используемые функции, которые необходимы для обработки больших объемов записей, включая ведение журнала / трассировку, управление транзакциями, статистику обработки заданий, перезапуск заданий, пропуск и управление ресурсами. Он также предоставляет более продвинутые технические услуги и функции, которые позволят выполнять пакетные задания чрезвычайно большого объема и с высокой производительностью благодаря методам оптимизации и разделения. Простые и сложные пакетные задания большого объема могут использовать платформу с высокой степенью масштабируемости для обработки значительных объемов информации.

<https://spring.io/projects/spring-batch>

6. Spring Hateoas.

Spring HATEOAS предоставляет некоторые API для упрощения создания REST-представлений, которые следуют принципу HATEOAS при работе с Spring и особенно Spring MVC. Основной проблемой, которую он пытается решить, является создание ссылки и сборка представления.

<https://spring.io/projects/spring-hateoas>

7. Spring Rest Docs.

Spring REST Docs поможет вам документировать сервисы RESTful.

Он сочетает в себе рукописную документацию, написанную с помощью [Asciidoctor,](http://asciidoctor.org/) и автоматически генерируемые фрагменты, созданные с помощью [Spring MVC Test](https://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle#spring-mvc-test-framework) . Такой подход освобождает вас от ограничений документации, создаваемой такими инструментами, как [Swagger](http://swagger.io/) .

Это помогает вам создавать документацию, которая является точной, краткой и хорошо структурированной. Затем эта документация позволяет вашим пользователям получать необходимую информацию с минимальными усилиями.

<https://spring.io/projects/spring-restdocs>

8. Spring AMQP.

Проект Spring AMQP применяет основные концепции Spring для разработки решений для обмена сообщениями на основе AMQP. Он предоставляет «шаблон» как абстракцию высокого уровня для отправки и получения сообщений. Он также обеспечивает поддержку управляемых сообщениями POJO с «контейнером слушателя». Эти библиотеки облегчают управление ресурсами AMQP, способствуя использованию внедрения зависимостей и декларативной конфигурации. Во всех этих случаях вы увидите сходство с поддержкой JMS в Spring Framework.

Проект состоит из двух частей; spring-amqp - это базовая абстракция, а spring-rabbit - это реализация RabbitMQ.

<https://spring.io/projects/spring-amqp>

9. Spring Web Flow.

Spring Web Flow основан на Spring MVC и позволяет реализовать «потоки» веб-приложения. Поток заключает в себе последовательность шагов, которые направляют пользователя при выполнении какой-либо бизнес-задачи. Он охватывает несколько HTTP-запросов, имеет состояние, обрабатывает транзакционные данные, может использоваться повторно и может быть динамичным и долгосрочным по своей природе.

<https://spring.io/projects/spring-webflow>

10. Spring Kafka.

Проект Spring for Apache Kafka (spring-kafka) применяет основные концепции Spring для разработки решений для обмена сообщениями на основе Kafka. Он предоставляет «шаблон» в качестве высокоуровневой абстракции для отправки сообщений. Он также обеспечивает поддержку управляемых сообщениями POJO с @KafkaListener аннотациями и «контейнером слушателя». Эти библиотеки способствуют использованию инъекций зависимостей и декларативных. Во всех этих случаях вы увидите сходство с поддержкой JMS в Spring Framework и поддержкой RabbitMQ в Spring AMQP.

<https://spring.io/projects/spring-kafka>

11. АОП - аспектно-ориентированное программирование

Аспектно-ориентированное программирование (АОП) — это парадигма

программирования, целью которой является повышение модульности за счет разделения

междисциплинарных задач. Это достигается путем добавления дополнительного поведения к

существующему коду без изменения самого кода.

АОП предоставляет возможность реализации в одном месте сквозной логики - т.е.

логики, которая применяется к множеству частей приложения - и обеспечения автоматического

применения этой логики по всему приложению.

Подход Spring к АОП заключается в создании "динамических прокси" для целевых

объектов и "привязывании" объектов к конфигурированному совету для выполнения сквозной

логики.

Аспектно-ориентированное программирование (АОП) — это парадигма программирования, целью которой является повышение модульности за счет разделения междисциплинарных задач. Это достигается путем добавления дополнительного поведения к существующему коду без изменения самого кода.

ООП, AOP и Spring - взаимодополняющие технологии, которые позволяют решать сложные проблемы путем разделения функционала на отдельные модули. АОП предоставляет возможность реализации сквозной логики - т.е. логики, которая применяется к множеству частей приложения - в одном месте и обеспечения автоматического применения этой логики по всему приложению.

Подход Spring к АОП заключается в создании "динамических прокси" для целевых объектов и "привязывании" объектов к конфигурированному совету для выполнения сквозной логики.

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

**Объединённая точка** (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.

* before - Запускает совет перед выполнением метода.
* after - Запускает совет после выполнения метода, независимо от результата его работы (кроме случая остановки работы JVM).
* after-returning - Запускает совет после выполнения метода, только в случае его успешного выполнения.
* after-throwing - Запускает совет после выполнения метода, только в случае, когда этот метод “бросает” исключение.
* around - Запускает совет до и после выполнения метода.

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

**Введение** (Introduction) - Это сущность, которая помогает нам добавлять новые атрибуты и/или методы в уже существующие классы.

**Целевой объект** (Target object) - Это объект на который направлены один или несколько аспектов.

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

С помощью АОП мы можем прописать, например, что будет выполняться до или после какого-то действия. Прописываем это один раз и этот функционал будет работать везде. Например нам нужно сделать логирование во всех методах @Service, с ООП нам бы пришлось прописывать этот функционал в каждом методе для всех @Service. А с АОП мы можем в конфигах прописать для @Service что будет происходить с каждым вызовом его методов, - в нашем случае писать логи. Элементы АОП такие как аспекты также используются в транзакциях спринга.

12. Как сделать локализацию в приложении

Spring MVC предоставляет очень простую и удобную возможность локализации приложения. Для этого необходимо сделать следующее:

* Создать файл resource bundle, в котором будут заданы различные варианты локализированной информации.
* Определить messageSource в конфигурации Spring используя классы ResourceBundleMessageSource или ReloadableResourceBundleMessageSource.
* Определить localeResolver класса CookieLocaleResolver для включения возможности переключения локали.
* С помощью элемента spring:message DispatcherServlet будет определять в каком месте необходимо подставлять локализированное сообщение в ответе.

13. Spring MVC interceptor

Перехватчики в Spring (Spring Interceptor) являются аналогом Servlet Filter и позволяют перехватывать запросы клиента и обрабатывать их. Перехватить запрос клиента можно в трех местах: preHandle, postHandle и afterCompletion.

**preHandle** — метод используется для обработки запросов, которые еще не были переданы в метода обработчик контроллера. Должен вернуть true для передачи следующему перехватчику или в handler method. False укажет на обработку запроса самим обработчиком и отсутствию необходимости передавать его дальше. Метод имеет возможность выкидывать исключения и пересылать ошибки к представлению.

**postHandle** — вызывается после handler method, но до обработки DispatcherServlet для передачи представлению. Может использоваться для добавления параметров в объект ModelAndView.

**afterCompletion** — вызывается после отрисовки представления.

Для создания обработчика необходимо расширить абстрактный класс HandlerInterceptorAdapter или реализовать интерфейс HandlerInterceptor. Также нужно указать перехватчики в конфигурационном файле Spring.

14. Реактивное программирование и 4 принципа

Реактивное программирование — это программирование в многопоточной среде.

Реактивный подход повышает уровень абстракции вашего кода и вы можете сконцентрироваться на взаимосвязи событий, которые определяют бизнес-логику, вместо того, чтобы постоянно поддерживать код с большим количеством деталей реализации. Код в реактивном программировании, вероятно, будет короче.

Поток — это последовательность, состоящая из постоянных событий, отсортированных по времени. В нем может быть три типа сообщений: значения (данные некоторого типа), ошибки и сигнал о завершении работы. Рассмотрим то, что сигнал о завершении имеет место для экземпляра объекта во время нажатия кнопки закрытия.

Мы получаем эти cгенерированные события асинхронно, всегда. Согласно идеологии реактивного программирования существуют три вида функций: те, которые должны выполняться, когда некоторые конкретные данные будут отправлены, функции обработки ошибок и другие функции с сигналами о завершении работы программы. Иногда последнее два пункта можно опустить и сосредоточится на определении функций для обработки значений. Слушать(listening) поток означает подписаться(subscribing) на него. То есть функции, которые мы определили это наблюдатели(observers). А поток является субъектом который наблюдают.,

Критерии реактивного приложения:

1. Responsive. Разрабатываемая система должна отвечать быстро и за определенное заранее заданное время. Кроме того система должна быть достаточно гибкой для самодиагностики и починки.

Что это значит на практике? Традиционно при запросе некоторого сервиса мы идем в базу данных, вынимаем необходимый объем информации и отдаем ее пользователю. Здесь все хорошо, если наша система достаточно быстрая и база данных не очень большая. Но что, если время формирования ответа гораздо больше ожидаемого? Кроме того, у пользователя мог пропасть интернет на несколько миллисекунд. Тогда все усилия по выборке данных и формированию ответа пропадают. Вспомните gmail или facebook. Когда у вас плохой интернет, вы не получаете ошибку, а просто ждете результат больше обычного. Кроме того, этот пункт говорит нам о том, что ответы и запросы должны быть упорядочены и последовательны.

1. Resilient. Система остается в рабочем состоянии даже, если один из компонентов отказал.

Другими словами, компоненты нашей системы должны быть достаточно гибкими и изолированными друг от друга. Достигается это путем репликаций. Если, например, одна реплика PostgreSQL отказала, необходимо сделать так, чтобы всегда была доступна другая. Кроме того, наше приложение должно работать во множестве экземпляров.

1. Elastic. Система должна занимать оптимальное количество ресурсов в каждый промежуток времени. Если у нас высокая нагрузка, то необходимо увеличить количество экземпляров приложения. В случае малой нагрузки ресурсы свободных машин должны быть очищены. Типичный инструменты реализации данного принципа: Kubernetes.
2. Message Driven. Общение между сервисами должно происходить через асинхронные сообщения. Это значит, что каждый элемент системы запрашивает информацию из другого элемента, но не ожидает получение результата сразу же. Вместо этого он продолжает выполнять свои задачи. Это позволяет увеличить пользу от системных ресурсов и управлять более гибко возникающими ошибками. Обычно такой результат достигается через реактивное программирование.

Spring 5 WebFlux - поддерживает стек реактивного программирования.