Оглавление

Spring

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

Inversion of Control - подход, который позволяет конфигурировать и управлять объектами Java с помощью рефлексии. Вместо ручного внедрения зависимостей, фреймворк забирает
ответственность за это посредством IoC-контейнера. Контейнер отвечает за управление жизненным циклом объекта: создание объектов, вызов методов инициализации и
конфигурирование объектов путём связывания их между собой.
Объекты, создаваемые контейнером, называются beans. Конфигурирование контейнера осуществляется путём внедрения аннотаций, но также, есть возможность, по старинке,
загрузить XML-файлы, содержащие определение bean'ов и предоставляющие информацию, необходимую для создания bean'ов.
Dependency Injection — является одним из способов реализации принципа IoC в Spring. Это шаблон проектирования, в котором контейнер передает экземпляры объектов
по их типу другим объектам с помощью конструктора или метода класса(setter), что позволяет писать слабосвязный код.

2. Что такое IoC контейнер?

Inversion of Control - подход, который позволяет конфигурировать и управлять объектами Java с помощью рефлексии. Вместо ручного внедрения зависимостей,
фреймворк забирает ответственность за это посредством IoC-контейнера.
Контейнер отвечает за управление жизненным циклом объекта: создание объектов,
вызов методов инициализации и конфигурирование объектов путём связывания их между собой.
Объекты, создаваемые контейнером, называются beans. Конфигурирование контейнера осуществляется путём внедрения аннотаций, но также, есть возможность,
по старинке, загрузить XML-файлы, содержащие определение bean'ов и предоставляющие информацию, необходимую для создания bean'ов.
Dependency Injection — является одним из способов реализации принципа IoC в Spring. Это шаблон проектирования, в котором контейнер передает экземпляры
объектов по их типу другим объектам с помощью конструктора или метода класса(setter), что позволяет писать слабосвязный код.

3. Расскажите про ApplicationContext и BeanFactory, чем отличаются? В каких случаях что стоит использовать?

Функционал	BeanFactory	ApplicationContext
Инициализация/автоматическое связывание бина	Да	Да
Автоматическая регистрация BeanPostProcessor	Нет	Да
Автоматическая регистрация BeanFactoryPostProcessor	Нет	Да
Удобный доступ к MessageSource (для i18n)	Нет	Да
ApplicationEvent публикация	Нет	Да

ApplicationContext является наследником BeanFactory и полностью реализует его функционал, добавляя больше специфических enterprise-функций. Может работать с бинами всех скоупов.
BeanFactory - это фактический контейнер, который создает, настраивает и управляет рядом bean-компонентов. Эти бины обычно взаимодействуют друг с другом и, таким образом, имеют зависимости между собой. Эти зависимости отражены в данных конфигурации, используемых BeanFactory. Может работать с бинами singleton и prototype.
BeanFactory обычно используется тогда, когда ресурсы ограничены (мобильные устройства), так как он легче по сравнению с ApplicationContext. Поэтому, если ресурсы не сильно ограничены, то лучше использовать ApplicationContext.
ApplicationContext загружает все бины при запуске, а BeanFactory по требованию.

4. Расскажите про аннотацию @Bean?	**Аннотация @Bean используется для указания того, что метод создает, настраивает и инициализирует новый объект, управляемый IoC-контейнером.** Такие методы можно использовать как в классах с аннотацией @Configuration, так и в классах с аннотацией @Component(или её наследниках). Имеет следующие свойства: **destroyMethod, initMethod** — варианты переопределения методов инициализации и удаления бина, указав их имена в аннотации. **name** — имя бина. По умолчанию именем бина является имя метода. **value** — алиас для name()
5. Аннотация @Component?	@Component - используется для указания класса в качестве компонента spring. Такой класс будет сконфигурирован как spring Bean.
6. Отличия аннотаций @Bean и @Component?	@Bean - ставится над методом и позволяет добавить bean, уже реализованного сторонней библиотекой класса, в контейнер, а @Component используется для указания класса, написанного программистом.
7. Расскажите про аннотации @Service и @Repository. Чем они отличаются?	@Service - указывает, что класс является сервисом для реализации бизнес-логики. @Repository, @Service, @Controller и @Configuration являются алиасами @Component, их также называют стереотипными аннотациями. @Repository - указывает, что класс используется для работы с поиском, получением и хранением данных. Аннотация может использоваться для реализации шаблона DAO. 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.
8. Расскажите про аннотацию @Autowired	@Autowired – автоматическое внедрение подходящего бина: 1) Контейнер определяет тип объекта для внедрения 2) Контейнер ищет соответствующий тип бина в контексте(он же контейнер) 3) Если есть несколько кандидатов, и один из них помечен как @Primary, то внедряется он 4) Если используется @Qualifier, то контейнер будет использовать информацию из @Qualifier, чтобы понять, какой компонент внедрять 5) В противном случае контейнер внедрит бин, основываясь на его имени или ID 6) Если ни один из способов не сработал, то будет выброшено исключение Контейнер обрабатывает DI с помощью AutowiredAnnotationBeanPostProcessor. В связи с этим, аннотация не может быть использована ни в одном BeanFactoryPP или BeanPP. В аннотации есть один параметр **required = true/fals** - указывает, обязательно ли делать DI. По умолчанию true. Либо можно не выбрасывать исключение, а оставить поле с null, если нужный бин не был найден - false. При циклической зависимости, когда объекты ссылаются друг на друга, нельзя ставить над конструктором.

9. Расскажите про аннотацию @Resource	@Resource(аннотация java) пытается получить зависимость: по имени, по типу, затем по описанию (Qualifier). Имя извлекается из имени аннотируемого сеттера или поля, либо берется из параметра name.
	
	@Resource //По умолчанию поиск бина с именем "context"
	private ApplicationContext context;
	
	@Resource(name="greetingService") //Поиск бина с именем "greetingService"
	public void setGreetingService(GreetingService service) {
	this.greetingService = service;
	}
	
	Разница с @Autowired:

	❖ ищет бин сначала по имени, а потом по типу; ❖ не нужна дополнительная аннотация для указания имени конкретного бина; ❖ @Autowired позволяет отметить место вставки бина как необязательное @Autowired(required = false); ❖ при замене Spring Framework на другой фреймворк, менять аннотацию @Resource не нужно.
10. Расскажите про аннотацию @Inject	@Inject входит в пакет javax.inject и, чтобы её использовать, нужно добавить зависимость: <dependency> <groupId>javax.inject</groupId> <artifactId>javax.inject</artifactId> <version>1</version> </dependency> @Inject (аннотация java) аналог @Autowired (аннотация spring) в первую очередь пытается подключить зависимость по типу, затем по описанию и только потом по имени. В ней нет параметров. Поэтому при использовании конкретного имени (Id) бина используем @Named: @Inject @Named("yetAnotherFieldInjectDependency") private ArbitraryDependency yetAnotherFieldInjectDependency;
11. Расскажите про аннотацию @Lookup	Обычно бины в приложении Spring являтся синглтонами, и для внедрения зависимостей мы используем конструктор или сеттер. Но бывает и другая ситуация: имеется бин Car – синглтон (singleton bean), и ему требуется каждый раз новый экземпляр бина Passenger. То есть Car – синглтон, а Passenger – так называемый прототипный бин (prototype bean). Жизненные циклы бинов разные. Бин Car создается контейнером только раз, а бин Passenger создается каждый раз новый – допустим, это происходит каждый раз при вызове какого-то метода бина Car. Вот здесь то и пригодится внедрение бина с помощью Lookup метода. Оно происходит не при инициализации контейнера, а позднее: каждый раз, когда вызывается метод. Суть в том, что вы создаете метод-заглушку в бине Car и помечаете его специальным образом – аннотацией @Lookup. Этот метод должен возвращать бин Passenger, каждый раз новый. Контейнер Spring под капотом создаст подкласс и переопределит этот метод и будет вам выдавать новый экземпляр бина Passenger при каждом вызове аннотированного метода. Даже если в вашей заглушке он возвращает null (а так и надо делать, все равно этот метод будет переопределен).

12. Можно ли вставить бин в статическое поле? Почему?	Spring не позволяет внедрять бины напрямую в статические поля. Это связано с тем, что когда загрузчик классов загружает статические значения, контекст Spring ещё не загружен.
	Чтобы исправить это, создайте нестатический сеттер-метод с @Autowired:
	
	private static OrderItemService orderItemService;
	
	@Autowired
	public void setOrderItemService(OrderItemService orderItemService) {
	TestDataInit.orderItemService = orderItemService;
	}

13. Расскажите про аннотации @Primary и @Qualifier	**@Qualifier** применяется если кандидатов для автоматического связывания несколько, она позволяет указать в качестве аргумента имя конкретного бина, который следует внедрить. Она может быть применена к отдельному полю класса, к отдельному аргументу метода или конструктора:
	public class AutowiredClass { @Autowired //к полям класса @Qualifier("main") private GreetingService greetingService;

	@Autowired //к отдельному аргументу конструктора или метода
	public void prepare(@Qualifier("main") GreetingService greetingService){
	/* что-то делаем... */
	};
	}
	Соответственно, у одной из реализации GreetingService должна быть установлена соответствующая аннотация @Qualifier:
	@Component
	@Qualifier("main")
	public class GreetingServiceImpl implements GreetingService {
	//...
	}

	@**Primary** тоже используется, чтобы отдавать предпочтение бину, когда есть несколько бинов одного типа, но в ней нельзя задать имя бина, она определяет значение по умолчанию, в то время как @Qualifier более специфичен. Если присутствуют аннотации @Qualifier и @Primary, то аннотация **@Qualifier будет иметь приоритет**.

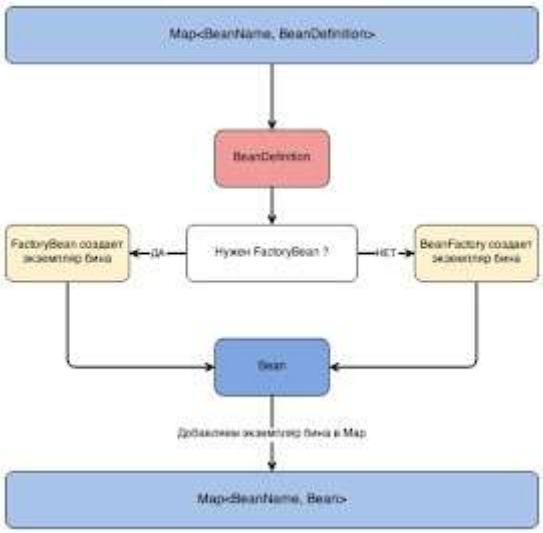
14. Расскажите про аннотацию @Conditional	Spring предоставляет возможность на основе вашего алгоритма включить или выключить определение бина или всей конфигурации через @Conditional, в качестве параметра которой указывается класс, реализующий интерфейс Condition, с единственным методом matches(ConditionContext var1, AnnotatedTypeMetadata var2), возвращающий boolean. Для создания более сложных условий можно использовать классы AnyNestedCondition, AllNestedConditions и NoneNestedConditions. Аннотация @Conditional указывает, что компонент имеет право на регистрацию в контексте только тогда, когда все условия соответствуют. Условия проверяются непосредственно перед тем, как должен быть зарегистрирован 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. Для того, чтобы проверить несколько условий, можно передать в @Conditional несколько классов с условиями: @Conditional(HibernateCondition.class, OurConditionClass.class) 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.
15. Расскажите про 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. Ее можно ставить на @Configuration и Component классы.
16. Расскажите про жизненный цикл бина, аннотации @PostConstruct и @PreDestroy()	**1) Парсирование конфигурации и создание BeanDefinition** После выхода четвертой версии спринга, у нас появилось четыре способа конфигурирования контекста: -Xml конфигурация — ClassPathXmlApplicationContext(“context.xml”) -Конфигурация через аннотации с указанием пакета для сканирования — AnnotationConfigApplicationContext(“package.name”) -Конфигурация через аннотации с указанием класса (или массива классов) помеченного аннотацией @Configuration - AnnotationConfigApplicationContext(JavaConfig.class). Этот способ конфигурации называется — JavaConfig. -Groovy конфигурация — GenericGroovyApplicationContext(“context.groovy”) Цель первого этапа — это создание всех BeanDefinition. BeanDefinition — это специальный интерфейс, через который можно получить доступ к метаданным будущего бина. В зависимости от того, какая у вас конфигурация, будет использоваться тот или иной механизм парсирования конфигурации. Допустим, что наша конфигурация основана на аннотациях. Если заглянуть внутрь AnnotationConfigApplicationContext, то можно увидеть два поля. private final **AnnotatedBeanDefinitionReader** reader; private final **ClassPathBeanDefinitionScanner** scanner; **ClassPathBeanDefinitionScanner** сканирует указанный пакет на наличие классов помеченных аннотацией @Component (или её алиаса). Найденные классы парсируются и для них создаются BeanDefinition. Чтобы было запущено сканирование, в конфигурации должен быть указан пакет для сканирования @ComponentScan({"package.name"}). **AnnotatedBeanDefinitionReader** работает в несколько этапов. 1. Первый этап — это регистрация всех @Configuration для дальнейшего

	парсирования. Если в конфигурации используются Conditional, то будут зарегистрированы только те конфигурации, для которых Condition вернет true. 2. Второй этап — это регистрация BeanDefinitionRegistryPostProcessor, который при помощи класса ConfigurationClassPostProcessor парсирует JavaConfig и создает BeanDefinition. **BeanDefinition** – это объект, который хранит в себе информацию о бине. Сюда входит: из какого класса бин надо создать, scope, установлена ли ленивая инициализация, нужно ли перед данным бином инициализировать другой, init и destroy методы, зависимости. Все полученные BeanDefinition'ы складываются в HashMap, в которой ключём является имя бина, а объект - сам BeanDefinition. При старте приложения, в IoC контейнер попадут бины, которые имеют scope Singleton (устанавливается по-умолчанию), остальные же создаются, тогда когда они нужны.

	2) Настройка созданных BeanDefinition
	Есть возможность повлиять на бины до их создания, иначе говоря мы имеем доступ к метаданным класса.
	Для этого существует специальный интерфейс BeanFactoryPostProcessor, реализовав который, мы получаем доступ к созданным BeanDefinition и можем их изменять. В нем один метод.
	Метод postProcessBeanFactory принимает параметром ConfigurableListableBeanFactory. Данная фабрика содержит много полезных методов, в том числе getBeanDefinitionNames, через который мы можем получить все BeanDefinitionNames,
	а уже потом по конкретному имени получить BeanDefinition для дальнейшей обработки метаданных.
	Разберем одну из родных реализаций интерфейса BeanFactoryPostProcessor. Обычно, настройки подключения к базе данных выносятся в отдельный property файл, потом при помощи PropertySourcesPlaceholderConfigurer
	они загружаются и делается inject этих значений в нужное поле. Так как inject делается по ключу, то до создания экземпляра бина нужно заменить этот ключ на само значение из property файла.
	Эта замена происходит в классе, который реализует интерфейс BeanFactoryPostProcessor. Название этого класса — PropertySourcesPlaceholderConfigurer. Он должен быть объявлен как static
	
	@Bean
	public static PropertySourcesPlaceholderConfigurer configurer() {
	return new PropertySourcesPlaceholderConfigurer();
	}

	3) Создание кастомных FactoryBean
	FactoryBean — это generic интерфейс, которому можно делегировать процесс создания бинов типа .
	В те времена, когда конфигурация была исключительно в xml, разработчикам был необходим механизм с помощью которого они бы могли управлять процессом создания бинов. Именно для этого и был сделан этот интерфейс.
	Создадим фабрику которая будет отвечать за создание всех бинов типа — Color.
	
	public class ColorFactory implements FactoryBean<Color> {
	@Override
	public Color getObject() throws Exception {

	Random random = new Random();
	Color color = new Color(random.nextInt(255), random.nextInt(255), random.nextInt(255));
	return color;
	}
	
	@Override
	public Class<?> getObjectType() {
	return Color.class;
	}
	
	@Override
	public boolean isSingleton() {
	return false;
	}
	}
	
	Теперь создание бина типа Color.class будет делегироваться ColorFactory, у которого при каждом создании нового бина будет вызываться метод getObject.
	Для тех кто пользуется JavaConfig, этот интерфейс будет абсолютно бесполезен.

	4) Создание экземпляров бинов Созданием экземпляров бинов занимается BeanFactory на основе ранее созданных BeanDefinition. Из Map<BeanName, BeanDefinition> получаем Map<BeanName, Bean> Создание бинов может делегироваться кастомным FactoryBean. О их создании читай выше.
	
	Класс, имплементирующий BeanPostProcessor, обязательно должен быть бином, поэтому мы его помечаем аннотацией @Component. SCOPE_SINGLETON — инициализация произойдет один раз на этапе поднятия контекста. SCOPE_PROTOTYPE — инициализация будет выполняться каждый раз по запросу. Причем во втором случае ваш бин будет проходить через все BeanPostProcessor-ы что может значительно ударить по производительности.

17. Расскажите про скоупы бинов? Какой скоуп используется по умолчанию? Что изменилось в Spring 5?	Существует 2 области видимости по умолчанию. Singleton - область видимости по умолчанию. В контейнере будет создан только один бин, и все запросы на него будут возвращать один и тот же бин. Prototype - приводит к созданию нового бина каждый раз, когда он запрашивается. Для бинов со scope “prototype” Spring не вызывает метод destroy(), так как не берет на себя контроль полного жизненного цикла этого бина. Spring не хранит такие бины в своём контексте (контейнере), а отдаёт их клиенту и больше о них не заботится (в отличие от синглтон-бинов). И 4 области видимости в веб-приложении. Request - Область видимости — 1 HTTP запрос. На каждый запрос создается новый бин Session - Область видимости — 1 сессия. На каждую сессию создается новый бин Application - Область видимости — жизненный цикл ServletContext WebSocket - Область видимости — жизненный цикл WebSocket Жизненный цикл web csope полный. В пятой версии Spring Framework не стало Global session scope. И появились Application и WebSocket
18. Расскажите про аннотацию @ComponentScan	Первый шаг для описания конфигурации Spring это добавление аннотаций — @Component или наследников. Однако, Spring должен знаеть где искать их. В @ComponentScan вы указываете пакеты, которые должны сканироваться. Можно указать массив строк. Spring будет искать бины и в их подпакетах. Мы можем расширить это поведение с помощью includeFilters и excludeFilters параметров в аннотации. Для ComponentScan.Filter доступно пять типов фильтров: ANNOTATION ASSIGNABLE_TYPE ASPECTJ REGEX CUSTOM Нужно для того, что например, имея какой-то ненужный класс в не нашей библиотеке, мы можем создать для него фильтр, чтобы его бин не инициализировался.
19. Как спринг работает с транзакциями? Расскажите про аннотацию @Transactional.	**marcobehler.com/guides/spring-transaction-management-transactional-in-depth** **Коротко**: Spring создает прокси для всех классов, помеченных @Transactional (либо если любой из методов класса помечен этой аннотацией), что позволяет вводить транзакционную логику до и после вызываемого метода. При вызове такого метода происходит следующее: - 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. 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. **Что произойдёт, если один метод с @Transactional вызовет другой метод с @Transactional?** Если это происходит в рамках одного сервиса, то второй транзакционный метод будет считаться частью первого, так как вызван у него изнутри, а так как спринг не знает о внутреннем вызове, то не создаст прокси для второго метода.

	Что произойдёт, если один метод БЕЗ @Transactional вызовет другой метод с @Transactional? Так как spring не знает о внутреннем вызове, то не создаст прокси для второго метода. **Будет ли транзакция откачена, если будет брошено исключение, которое указано в контракте метода?** Если в контракте описано это исключение, то она не откатится. Unchecked исключения в транзакционном методе можно ловить, а можно и не ловить. Значения атрибута propagation у аннотации:
	REQUIRED — применяется по умолчанию. При входе в @Transactional метод будет использована уже существующая транзакция или создана новая транзакция, если никакой ещё нет **REQUIRES_NEW** — новая транзакция всегда создаётся при входе метод, ранее созданные транзакции приостанавливаются до момента возврата из метода. **NESTED** — корректно работает только с базами данных, которые умеют savepoints. При входе в метод в уже существующей транзакции создаётся savepoint, который по результатам выполнения метода будет либо сохранён, либо откачен. Все изменения, внесённые методом, подтвердятся только поздее, с подтверждением всей транзакции. Если текущей транзакции не существует, будет создана новая.
	MANDATORY — всегда используется существующая транзакция и кидается исключение, если текущей транзакции нет. **SUPPORTS** — метод с этим правилом будет использовать текущую транзакцию, если она есть, либо будет исполнятся без транзакции, если её нет. **NOT_SUPPORTED** — при входе в метод текущая транзакция, если она есть, будет приостановлена и метод будет выполняться без транзакции. **NEVER** — явно запрещает исполнение в контексте транзакции. Если при входе в метод будет существовать транзакция, будет выброшено исключение.
	Остальные атрибуты: **rollbackFor** = Exception.class - если какой-либо метод выбрасывает указанное исключение, контейнер всегда откатывает текущую транзакцию. По умолчанию отлавливает RuntimeException **noRollbackFor** = Exception.class - указание того, что любое исключение, кроме заданных, должно приводить к откату транзакции. **rollbackForClassName** и **noRollbackForClassName** - для задания имен исключений в строковом виде. **readOnly** - разрешает только операции чтения. В свойстве **transactionManager** хранится ссылка на менеджер транзакций, определенный в конфигурации Spring. **timeOut** - По умолчанию используется таймаут, установленный по умолчанию для базовой транзакционной системы. Сообщает менеджеру tx о продолжительности времени, чтобы дождаться простоя tx, прежде чем принять решение об откате не отвечающих транзакций. **isolation -** уровень изолированности транзакций
	Подробно: Для работы с транзакциями Spring Framework использует AOP-прокси: Для включения возможности управления транзакциями нужно разместить аннотацию @EnableTransactionManagement у класса конфигурации

	@Configuration. Она означает, что классы, помеченные @Transactional, должны быть обернуты аспектом транзакций. Отвечает за регистрацию необходимых компонентов Spring, таких как TransactionInterceptor и советы прокси. Регистрируемые компоненты помещают перехватчик в стек вызовов при вызове методов @Transactional. Если мы используем Spring Boot и имеем зависимости spring-data-* или spring-tx, то управление транзакциями будет включено по умолчанию. Пропагейшн работает только если метод вызывает другой метод в другом сервисе. Если метод вызывает другой метод в этом же сервисе, то используется this и вызов проходит мимо прокси. Это ограничение можно обойти при помощи self-injection.
	Слой логики(Service) - лучшее место для @Transactional. Помечая @Transactional класс @Service, то все его методы станут транзакционными. Так, при вызове, например, метода save() произойдет примерно следующее: 1. Вначале мы имеем: ❖ класс **TransactionInterceptor, у которого вызывается 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.

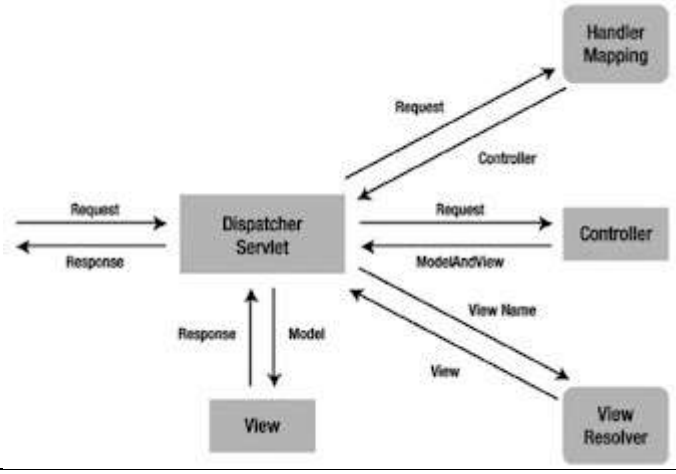
	2. Transaction Interceptor
	В TransactionInterceptor отработает код до работы метода save(), в котором будет определено, выполнить ли метод save() в пределах уже существующей транзакции БД или должна стартовать новая отдельная транзакция.
	TransactionInterceptor сам не содержит логики по принятию решения, 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:
	try {
	transaction.begin(); // логика до
	service.save();
	transaction.commit(); // логика после
	} catch(Exception ex) {
	transaction.rollback();
	throw ex;
	}

	3. TransactionManager Менеджер транзакций должен предоставить ответ на два вопроса: ❖ Должен ли создаться новый EntityManager? ❖ Должна ли стартовать новая транзакция БД? Решение принимается, основываясь на следующих фактах: ❖ выполняется ли хоть одна транзакция в текущий момент или нет; ❖ атрибута «propagation» в @Transactional. 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.

20. Расскажите про аннотации @Controller и @RestController. Чем они отличаются? Как вернуть ответ со своим статусом (например 213)?	@**Controller** - специальный тип класса, обрабатывает HTTP-запросы и часто используется с аннотацией @RequestMapping. @**RestController** ставится на класс-контроллер вместо @Controller. Она указывает, что этот класс оперирует не моделями, а данными. Она состоит из аннотаций @Controller и @ResponseBody. Была введена в Spring 4.0 для упрощения создания RESTful веб-сервисов. @**ResponseBody** сообщает контроллеру, что возвращаемый объект автоматически сериализуется (используя Jackson message converter) в json или xml и передается обратно в объект HttpResponse. ResponseEntity используется для формирования кастомизированного HTTP-ответа с пользовательскими параметрами (заголовки, код статуса и тело ответа). Во всех остальных случаях достаточно использовать @ResponseBody. Если мы хотим использовать ResponseEntity, то просто должны вернуть его из метода, Spring позаботится обо всем остальном. **return ResponseEntity.status(213);**
21. Что такое ViewResolver?	ViewResolver - распознаватель представлений - это способ работы с представлениями(html-файлы), который поддерживает их распознавание на основе имени, возвращаемого контроллером. Spring Framework поставляется с большим количеством реализаций ViewResolver. Например, класс UrlBasedViewResolver поддерживает прямое преобразование логических имен в URL. InternalResourceViewResolver — реализация ViewResolver по умолчанию, которая позволяет находить представления, которые возвращает контроллер для последующего перехода к ним. Ищет по заданному пути, префиксу, суффиксу и имени. Любым реализациям ViewResolver желательно поддерживать интернационализацию, то есть множество языков. Существует также несколько реализаций для интеграции с различными технологиями представлений, такими как FreeMarker (FreeMarkerViewResolver), Velocity (VelocityViewResolver) и JasperReports (JasperReportsViewResolver)
22. Чем отличаются Model, ModelMap и ModelAndView?	**Model** - **интерфейс**, представляет коллекцию пар ключ-значение Map<String, Object>. Содержимое модели используется для отображения данных во View. Например, если View выводит информацию об объекте Customer, то она может ссылаться к ключам модели, например customerName, customerPhone, и получать значения для этих ключей. Объекты-значения из модели также могут содержать бизнес-логику. **ModelMap - класс, наследуется от LinkedHashMap,** тоже используется для передачи значений для визуализации представления. Преимущество ModelMap заключается в том, что он дает нам возможность передавать коллекцию значений и обрабатывать эти значения, как если бы они были внутри Map. **ModelAndView** - это просто контейнер для ModelMap, объект View и HttpStatus. Это позволяет контроллеру возвращать все значения как одно. **View** используется для отображения данных приложения пользователю. Spring MVC поддерживает несколько поставщиков View(они называются шаблонизаторы) — JSP, JSF, Thymeleaf, и т.п. Интерфейс View преобразует объекты в обычные сервлеты.
23. Расскажите про паттерн Front Controller, как он реализован в Spring?	Front controller - обеспечивает единую точку входа для всех входящих запросов. Все запросы обрабатываются одним обработчиком – DispatcherServlet с маппингом “/”. Этот обработчик может выполнить аутентификацию, авторизацию, регистрацию или отслеживание запроса, а затем распределяет их между контроллерами, обрабатывающими разные URL. Это и есть реализация паттерна Front Controller. 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. ❖ Один из контекстов будет корневым, а все остальные контексты будут дочерними.

	❖ Все дочерние контексты могут получить доступ к бинам, определенным в корневом контексте, но не наоборот. ❖ Каждый дочерний контекст внутри себя может переопределить бины из корневого контекста. **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.
24. Расскажите про паттерн MVC, как он реализован в Spring?	MVC — это шаблон проектирования, делящий программу на 3 вида компонентов: **Model** — модель отвечает за хранение данных. **View** — отвечает за вывод данных на фронтенде. **Controller** — оперирует моделями и отвечает за обмен данными model с view. Основная цель следования принципам MVC — отделить реализацию бизнес-логики приложения (модели) от ее визуализации (view). **Spring MVC** - это веб-фреймворк, основанный на Servlet API, с использованием двух шаблонов проектирования - Front controller и MVC. 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: **DispatcherServlet** является главным контроллером, который получает запросы и распределяет их между другими контроллерами. @RequestsMapping указывает, какие именно запросы будут обрабатываться в конкретном контроллере. Может быть несколько экземпляров DispatcherServlet, отвечающих за разные задачи (обработка запросов пользовательского интерфейса, REST служб и т.д.). Каждый экземпляр DispatcherServlet имеет собственную конфигурацию WebApplicationContext. **HandlerMapping**. Выбор класса и его метода, которые должны обработать данный входящий запрос на основе любого внутреннего или внешнего для этого запроса атрибута или состояния. **Controller** — оперирует моделями и отвечает за обмен данными model с view. **ViewResolver**. Выбор, какое именно View должно быть показано клиенту на основе имени, полученного от контроллера. **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. **HandlerAdapter**. Помогает DispatcherServlet вызвать и выполнить метод для обработки входящего запроса. **ContextLoaderListener** - слушатель при старте и завершении корневого класса Spring WebApplicationContext. Основным назначением является связывание жизненного цикла ApplicationContext и ServletContext, а также автоматического создания ApplicationContext. Можно использовать этот класс для доступа к бинам из различных контекстов спринг.

	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 метод. Вызванный метод формирует данные Model и возвращает их в DispatcherServlet вместе с именем View (как правило имя html-файла). ❖ При помощи интерфейса ViewResolver DispatcherServlet определяет, какое View нужно использовать на основании имени, полученного от контроллера. ➢ если это REST-запрос на сырые данные (JSON/XML), то DispatcherServlet сам его отправляет, минуя ViewResolver; ➢ если обычный запрос, то DispatcherServlet отправляет данные Model в виде атрибутов во View - шаблонизаторы Thymeleaf, FreeMarker и т.д., которые сами отправляют ответ. Как видим, все действия происходят через один DispatcherServlet.

25. Что такое АОП? Как реализовано в спринге?	❖ 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 метод. Вызванный метод формирует данные Model и возвращает их в DispatcherServlet вместе с именем View (как правило имя html-файла). ❖ При помощи интерфейса ViewResolver DispatcherServlet определяет, какое View нужно использовать на основании имени, полученного от контроллера. ➢ если это REST-запрос на сырые данные (JSON/XML), то DispatcherServlet сам его отправляет, минуя ViewResolver; ➢ если обычный запрос, то DispatcherServlet отправляет данные Model в виде атрибутов во View - шаблонизаторы Thymeleaf, FreeMarker и т.д., которые сами отправляют ответ. Как видим, все действия происходят через один DispatcherServlet.
26.В чем разница между Filters, Listeners and Interceptors?	**Filter** выполняет задачи фильтрации либо по пути запроса к ресурсу, либо по пути ответа от ресурса, либо в обоих направлениях. Фильтры выполняют фильтрацию в методе doFilter. Каждый фильтр имеет доступ к объекту FilterConfig, из которого он может получить параметры инициализации, и ссылку на ServletContext. Фильтры настраиваются в дескрипторе развертывания веб-приложения. При создании цепочки фильтров, веб-сервер решает, какой фильтр вызывать первым, в соответствии с порядком регистрации фильтров. Когда вызывается метод doFilter(...) первого фильтра, веб-сервер создает объект FilterChain, представляющий цепочку фильтров, и передаёт её в метод. Зависят от контейнера сервлетов. Могут работать с js, css
	интерфейс UserDetailsService - подход к загрузке информации о пользователе в Spring Security. Единственный метод этого интерфейса принимает имя пользователя в виде String и возвращает UserDetails. Он представляет собой принципала, но в расширенном виде и с учетом специфики приложения. В случае успешной аутентификации, UserDetails используется для создания Authentication объекта, который хранится в SecurityContextHolder. Ещё одним важным методом Authentication является getAuthorities() - предоставляет массив объектов GrantedAuthority(роли). Credentials - под ними понимаются пароль пользователя, но им может быть и отпечаток пальца, фото сетчатки и т.п. **preHandle** — метод используется для обработки запросов, которые еще не были переданы в метод 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.
	Listener - это класс, имплементирующий интерфейс ServletContextListener с аннотацией @WebListener. Listener ждет когда произойдет указанное событие, затем «перехватывает» событие и запускает собственное событие. Он инициализируется только один раз при запуске веб-приложения и уничтожается при остановке веб-приложения. Все ServletContextListeners уведомляются об инициализации контекста до инициализации любых

	фильтров или сервлетов в вебприложении и об уничтожении контекста после того, как все сервлеты и фильтры уничтожены
27. Можно ли передать в 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**){...} http://localhost:8080/api/foos?id=1,2,3 public String getFoos(@RequestParam **List<String> id**){...}
28. Как работает Spring Security? Как сконфигурировать? Какие интерфейсы используются?	**В кратце**, основными блоками Spring Security являются: SecurityContextHolder, чтобы обеспечить доступ к SecurityContext. SecurityContext, содержит объект Authentication и в случае необходимости информацию системы безопасности, связанную с запросом. Authentication представляет принципала с точки зрения Spring Security. GrantedAuthority отражает разрешения выданные доверителю в масштабе всего приложения. UserDetails предоставляет необходимую информацию для построения объекта Authentication из DAO объектов приложения или других источника данных системы безопасности. UserDetailsService, чтобы создать UserDetails, когда передано имя пользователя в виде String (или идентификатор сертификата или что-то подобное).
	Подробно: Самым фундаментальным является **SecurityContextHolder**. В нем мы храним информацию о текущем контексте безопасности приложения, который включает в себя подробную информацию о пользователе, работающем с приложением. По умолчанию SecurityContextHolder использует **MODE_THREADLOCAL** для хранения такой информации, что означает, что контекст безопасности всегда доступен для методов исполняющихся в том же самом потоке, даже если контекст безопасности явно не передается в качестве аргумента этих методов: SecurityContextHolder.getContext().getAuthentication().getPrincipal(); UserDetails выступает в качестве принципала. **MODE_GLOBAL** - все потоки Java-машины используют один контекст безопасности. **MODE_INHERITABLETHREADLOCAL** - потоки порожденные от одного защищенного потока, наличие аналогичной безопасности.
	Интерфейс **UserDetailsService** - подход к загрузке информации о пользователе в Spring Security. Единственный метод этого интерфейса принимает имя пользователя в виде String и возвращает **UserDetails.** Он представляет собой принципала, но в расширенном виде и с учетом специфики приложения. В случае успешной аутентификации, UserDetails используется для создания Authentication объекта, который хранится в SecurityContextHolder. Ещё одним важным методом Authentication является getAuthorities() - предоставляет массив объектов GrantedAuthority(роли). Credentials - под ними понимаются пароль пользователя, но им может быть и отпечаток пальца, фото сетчатки и т.п.
	Процесс аутентификации: 1. UsernamePasswordAuthenticationFilter получают имя пользователя и пароль и создает экземпляр класса

	UsernamePasswordAuthenticationToken (экземпляр интерфейса Authentication). 2. Токен передается экземпляру AuthenticationManager для проверки. 3. AuthenticationManager возвращает полностью заполненный экземпляр Authentication в случае успешной аутентификации. 4. Устанавливается контекст безопасности путем вызова SecurityContextHolder.getContext().setAuthentication(...), куда передается вернувшийся экземпляр Authentication. 5. При успешной аутентификации можно использовать successHandler
29. Что такое SpringBoot? Какие у него преимущества? Как конфигурируется? Подробно.	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.
	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. Starter-пакеты можно создавать и свои. **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 springboot-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).** 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. 2. @EnableAutoConfiguration импортирует класс EnableAutoConfigurationImportSelector. Этот класс не объявляет бины сам, а использует фабрики. 3. Класс EnableAutoConfigurationImportSelector импортирует ВСЕ (более 150) перечисленные в META-INF/spring.factories конфигурации, чтобы предоставить нужные бины в контекст приложения. 4. 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), отсутствии бина (@ConditionalOnMissingBean) и т.п. Таким образом наличие конфигурации не значит, что бин будет создан и зачастую конфигурация ничего делать и создавать не будет. 5. Созданный в итоге **AnnotationConfig**EmbeddedWeb**ApplicationContext** ищет в том же DI контейнере фабрику для запуска embedded servlet container. 6. Servlet container запускается, приложение готово к работе
30. Расскажите про нововведения 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 ● Поддержка Kotlin ● Реактивность (веб-инфраструктура с реактивным стеком, «Spring WebFlux») ● 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