Оглавление

[IOC, DI, бины 5](#_Toc45003105)

[Bean Factory. 6](#_Toc45003106)

[ApplicationСontext 6](#_Toc45003107)

[Dependency Injection. 7](#_Toc45003108)

[@Qualifier и @Primary 12](#_Toc45003109)

[@Bean и @Component 12](#_Toc45003110)

[@Component, @Service, @Repository 13](#_Toc45003111)

[@Controller и @RestController 14](#_Toc45003112)

[@ResponseBody и @ResponseEntity 15](#_Toc45003113)

[Транзакции в Spring. @Transactional 15](#_Toc45003114)

[Жизненный цикл бина в Spring 27](#_Toc45003115)

[Model, ModelMap, ModelAndView 30](#_Toc45003116)

[Какие паттерны используются в Spring (singleton, prototype, builder, proxy, chain of responsibility, dependency injection)? 31](#_Toc45003117)

[Bean scopes 36](#_Toc45003118)

[Spring Data 37](#_Toc45003119)

[Spring Security 38](#_Toc45003120)

[@Recource, @Autowired, @Inject 40](#_Toc45003121)

[@Conditional 41](#_Toc45003122)

[@ComponentScan 41](#_Toc45003123)

[Spring Boot 42](#_Toc45003124)

[Нововведение Spring 5 45](#_Toc45003125)

[Spring Cloud (Data Flow) 46](#_Toc45003126)

[Spring Integration 46](#_Toc45003127)

[Spring Batch 47](#_Toc45003128)

[Spring Hateoas 47](#_Toc45003129)

[Spring Rest Docs 48](#_Toc45003130)

[Spring AMQP 48](#_Toc45003131)

[Spring Web Flow 49](#_Toc45003132)

[Spring Kafka 49](#_Toc45003133)

[Контейнеры спринга 50](#_Toc45003134)

[Части спринга, модули 54](#_Toc45003135)

[АОР 58](#_Toc45003136)

[Spring Framework 61](#_Toc45003137)

[BeanFactory и ApplicationContext разница 62](#_Toc45003138)

[Работа спринга с ДАО 63](#_Toc45003139)

[Что такое контроллер 64](#_Toc45003140)

[View-resolver 64](#_Toc45003141)

[model.put разница model.addAttribute 65](#_Toc45003142)

[Form Binding 66](#_Toc45003143)

[Как сделать локализацию в приложении 66](#_Toc45003144)

[Spring MVC interceptor 67](#_Toc45003145)

[parent in pom.xml 68](#_Toc45003146)

[@Sheduler 68](#_Toc45003147)

[Внедрение в синглтон prototype 69](#_Toc45003148)

[@Profile 69](#_Toc45003149)

[Транзакция с аннотацией @Transactional вызывает метод без аннотации 70](#_Toc45003150)

[@Around 70](#_Toc45003151)

[SPELL 70](#_Toc45003152)

[Container — как устроен (Map) 70](#_Toc45003153)

[FileSystemApplicationContext 70](#_Toc45003154)

[Bean Definition 71](#_Toc45003155)

[Статические методы в бинах 71](#_Toc45003156)

[ContextLoaderListener 71](#_Toc45003157)

[HandlerException 71](#_Toc45003158)

[@ControllerAdvice 72](#_Toc45003159)

[@ModelAttribute над методом 73](#_Toc45003160)

[Реактивное программирование и 4 принципа 73](#_Toc45003161)

[WebSocket 76](#_Toc45003162)

[@Lookup 76](#_Toc45003163)

[@Target и @Retention 77](#_Toc45003164)

[Bean 79](#_Toc45003165)

[Прокси 82](#_Toc45003166)

[Spring MVC 83](#_Toc45003167)

[Starter-pack Spring Boot — как создать? 88](#_Toc45003168)

[SOAP и REST 88](#_Toc45003169)

[Spring Data — что под капотом 90](#_Toc45003170)

[Spring Security — что под капотом 90](#_Toc45003171)

[@Secured 90](#_Toc45003172)

[Как исключить класс, автоконфигурацию класса в Spring Boot 90](#_Toc45003173)

[Как заинжектить примитив 91](#_Toc45003174)

[@Value 91](#_Toc45003175)

[Как заинжектить коллекцию 91](#_Toc45003176)

[BeanPostProcessor и BeanFactoryPostProcessor 91](#_Toc45003177)

[Часто используемые аннотации спринга 91](#_Toc45003178)

# IoC, DI, бины

Инверсия управления (Inversion of Control, IoC) позволяет конфигурировать и управлять объектами Java при помощи рефлексии. Этот контейнер забирает на себя ответственность за внедрение зависимостей и отвечает за управление жизненным циклом объекта.

Бины — объекты, создаваемые контейнером (IoC). Конфигурирование контейнера осуществляется при помощи аннотаций или XML‑файла.

Внедрение зависимости (Dependency Injection, DI) — процесс, когда один объект реализует свой функционал через другой. Можно сказать, что внедрение зависимости является частью инверсии управления и согласно принципу единой обязанности забирает на себя одну обязанность — внедрение зависимости.

Не стоит путать с инверсией зависимостей (Dependency Inversion). Инверсия зависимостей — набор паттернов и принципов разработки ПО для написания слабосвязанного кода.

Преимущества DI

∙ Сокращение объема связующего кода. Одним из самых больших плюсов DI является возможность значительного сокращения объема кода, который должен быть написан для связывания вместе различных компонентов приложения. Зачастую этот код очень прост — при создании зависимости должен создаваться новый экземпляр соответствующего объекта.

∙ Упрощенная конфигурация приложения. За счет применения DI процесс конфигурирования приложения значительно упрощается. Для конфигурирования классов, которые могут быть внедрены в другие классы, можно использовать аннотации или XML‑файлы.

∙ Возможность управления общими зависимостями в единственном репозитории. При традиционном подходе к управлению зависимостями в общих службах, к которым относятся, например, подключение к источнику данных, транзакция, удаленные службы и т. п., вы создаете экземпляры (или получаете их из определенных фабричных классов) зависимостей там, где они нужны — внутри зависимого класса. Это приводит к распространению зависимостей по множеству классов в приложении, что может затруднить их изменение. В случае использования DI вся информация об общих зависимостях содержится в единственном репозитории (в Spring есть возможность хранить эту информацию в XML‑файлах или Java классах), что существенно упрощает управление зависимостями и снижает количество возможных ошибок.

∙ Улучшенная возможность тестирования. Когда классы проектируются для DI, становится возможной простая замена зависимостей. Это особенно полезно при тестировании приложения.

∙ Стимулирование качественных проектных решений для приложений. Вообще говоря, проектирование для DI означает проектирование с использованием интерфейсов. Используя Spring, вы получаете в свое распоряжение целый ряд средств DI и можете сосредоточиться на построении логики приложения, а не на поддерживающей DI платформе.

Как реализуется DI в Spring Framework?

Реализация DI в Spring основана на двух ключевых концепциях Java — компонентах JavaBean и интерфейсах. При использовании Spring в качестве поставщика DI вы получаете гибкость определения конфигурации зависимостей внутри своих 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 — 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 создает новый экземпляр.

# Bean Factory.

BeanFactory — контейнер, который создает, настраивает и управляет бинами. Эти бины взаимодействуют друг с другом и имеют зависимости между собой. Эти зависимости отражены в данных конфигурации, используемых BeanFactory.

BeanFactory используется, когда ресурсы ограничены (мобильные устройства).

Если ресурсы не сильно ограничены — лучше использовать ApplicationContext.

# ApplicationСontext

ApplicationContext — главный интерфейс в Spring-приложении, который предоставляет информацию о конфигурации приложения. Как и BeanFactory загружает бины, связывает их вместе и конфигурирует их определенным образом, а также обладает дополнительной функциональностью.

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.

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)

# @Qualifier и @Primary

Два бина одного типа — exception.

|  |  |  |
| --- | --- | --- |
|  | @Primary | @Qualifier |
| где ставится | над классом (@Bean, @Component и сервис?) | при создании объекта с аннотацией @Autowired |
| значение | предпочтительный бин | указывает бин |
| атрибуты | — | value |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @Target({ElementType.TYPE, ElementType.METHOD})  @Retention(RetentionPolicy.RUNTIME)  @Documented  public @interface Primary {  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | @Target({  ElementType.FIELD,  ElementType.METHOD,  ElementType.PARAMETER,  ElementType.TYPE,  ElementType.ANNOTATION\_TYPE})  @Retention(RetentionPolicy.RUNTIME)  @Inherited  @Documented  public @interface Qualifier {  String value() default "";  } |

# @Component, @Service и @Repository

Обозначают класс как бин. У всех единственный атрибут value (что делает?).

|  |  |
| --- | --- |
| @Component | универсальный компонент |
| @Service | содержит бизнес логику и вызывает методы хранилища |
| @Repository | роль хранилища |

У @Repository есть специфика работы с исключениями:

автоматически перехватывает специфические Java исключения и пробрасывает их дальше как неконтролируемые исключения доступа к данным Spring. Для этого в контексте прописывается класс PersistenceExceptionTranslationPostProcessor. В @Repository будет полный стектрайс со всеми методами откуда летит эксепшн, в @Component/@Service будет один эксепшн.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | @Target({ElementType.TYPE})  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Indexed  public @interface Component {  String value() default "";  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | @Target({ElementType.TYPE})  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Component  public @interface Service {  @AliasFor(  annotation = Component.class  )  String value() default "";  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | @Target({ElementType.TYPE})  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Component  public @interface Repository {  @AliasFor(  annotation = Component.class  )  String value() default "";  } |

# @Bean и @Component (@Service, @Repository)

|  |  |
| --- | --- |
| @Bean | явное объявление бин‑компонента и его настройка (внедрить объект сторонней закрытой б |
| @Component, @Service и @Repository | автоматические обнаружение и настройка бин‑компонента (сканирование пути к классам) |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21. | @Target({  ElementType.METHOD,  ElementType.ANNOTATION\_TYPE})  @Retention(RetentionPolicy.RUNTIME)  @Documented  public @interface Bean {  @AliasFor("name")  String[] value() default {};  @AliasFor("value")  String[] name() default {};  @Deprecated  Autowire autowire() default Autowire.NO;  boolean autowireCandidate() default true;  String initMethod() default "";  String destroyMethod() default "(inferred)";  } |

# @Controller и @RestController

@Controller — класс является контроллером MVC. Диспетчер сервлетов просматривает такие классы для поиска @RequestMapping (@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PatchMapping).

@Controller помечает класс как контроллер HTTP запросов. @Controller обычно используется в сочетании с аннотацией @RequestMapping, используемой в методах обработки запросов.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | @Target({ElementType.TYPE})  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Component  public @interface Controller {  @AliasFor(  annotation = Component.class  )  String value() default "";  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23. | @Target({ElementType.TYPE, ElementType.METHOD})  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Mapping  public @interface RequestMapping {  String name() default "";  @AliasFor("path")  String[] value() default {};  @AliasFor("value")  String[] path() default {};  RequestMethod[] method() default {};  String[] params() default {};  String[] headers() default {};  String[] consumes() default {};  String[] produces() default {};  } |

На случай, если нужно что ‑то вернуть используется @RestController (Spring 4.0).

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | @Target({ElementType.TYPE})  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Controller  @ResponseBody  public @interface RestController {  @AliasFor(  annotation = Controller.class  )  String value() default "";  } |

Применив ее к контроллеру, автоматически добавляются аннотации @RestController, а также @ResponseBody применяется ко всем методам. Аннотация @ResponseBody сообщает контроллеру, что возвращаемый объект автоматически сериализуется в JSON и передается обратно в объект HttpResponse.

@RestController = @Controller + @ResponseBody.

@RestController превращает помеченный класс в Spring‑бин. Этот бин для конвертации входящих/исходящих данных использует Jackson message converter. Как правило целевые данные представлены в json или xml.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | @Controller  @RequestMapping("books")  public class SimpleBookController {  @GetMapping("/{id}", produces =  "application/json")  public @ResponseBody Book getBook(  @PathVariable int id) {  return findBookById(id);  }  private Book findBookById(int id) {  // ...  }  } |

или так:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | @RestController  @RequestMapping("books-rest")  public class SimpleBookRestController {  @GetMapping("/{id}", produces =  "application/json")  public Book getBook(@PathVariable int id) {  return findBookById(id);  }  private Book findBookById(int id) {  // ...  }  } |

# @ResponseBody и @ResponseEntity

@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. Будет передаваться ответ, состоящий из заголовка, тела и статуса.

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

# Транзакции в Spring. @Transactional

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.

Аннотация сама по себе определяет область действия одной транзакции БД. Транзакция БД происходит внутри области действий persistence context.

Persistence контекстом в JPA является EntityManager, который использует внутри класс Session ORM-фреймворка Hibernate (при использовании Hibernate как persistence провайдера). Persistence контекст — объект-синхронайзер, который отслеживает состояния ограниченного набора Java объектов и синхронизирует изменения состояний этих объектов с состоянием соответствующих записей в БД.

Один объект Entity Manager не всегда соответствует одной транзакции БД. Один объект Entity Manager может быть использован несколькими транзакциями БД. Самый частый случай такого использования — когда приложение использует шаблон «Open Session in View» для предотвращения исключений «ленивой» инициализации. В этом случае запросы, которые могли быть выполнены одной транзакцией при вызове из слоя сервиса, выполняются в отдельных транзакциях в слое View, но они совершаются через один и тот же Entity Manager.

При этом @PersistenceContext не может внедрить entity manager напрямую. Entity Manager это интерфейс, и то, что внедряется в бин, не является самим по себе entity менеджером, это context aware proxy, который будет делегировать к конкретному entity менеджеру в рантайме.

Но прокси persistence контекста, которое имплементирует EntityManager не является достаточным набором компонентов для осуществления декларативного управления транзакциями. На самом деле нужно три компонента:  
∙ прокси Entity менеджера;  
∙ аспект транзакций;  
∙ менеджер транзакций.

Аспект транзакций — «around» аспект, который вызывается и до и после выполнения аннотированного бизнес метода. Конкретный класс для имплементации этого аспекта — это TransactionInterceptor.

Аспект транзакций имеет две главные функции:

∙ В момент «до» аспект определяет выполнить ли выполняемый метод в пределах уже существующей транзакции БД или должна стартовать новая отдельная транзакция. В момент «до» аспект сам не содержит логики по принятию решения, решение начать новую транзакцию, если это нужно, делегируется Transaction менеджеру.

∙ В момент «после» аспект решает, что делать с транзакцией, делать коммит, откат или оставить незакрытой.

Transaction менеджер. Менеджер транзакций должен предоставить ответ на два вопроса: должен ли создаться новый Entity Manager? Должна ли стартовать новая транзакция БД?

Ответы необходимы предоставить в момент, когда вызывается логика аспекта транзакций в момент «до». Менеджер транзакций принимает решение, основываясь на следующих фактах: выполняется ли хоть одна транзакция в текущий момент; нет ли атрибута «propagation» у метода, аннотированного @Transactional (для примера, REQUIRES\_NEW всегда стартует новую транзакцию).

Если менеджер решил создать новую транзакцию, тогда:  
∙ создается новый entity менеджер;  
∙ «привязка» entity менеджера к текущему потоку (Thread);  
∙ «взятие» соединения из пула соединений БД;  
∙ «привязка» соединения к текущему потоку.

И entity менеджер и это соединение привязываются к текущему потоку, используя переменные ThreadLocal. Они хранятся в потоке, пока выполняется транзакция, и затем передаются менеджеру транзакций для очистки, когда они уже будут не нужны. Любая часть программы, которой нужен текущий entity manager или соединение, может заполучить их из потока. Этим компонентом программы, который делает именно так является Entity Manager Proxy.

EntityManager proxy. Когда бизнес метод делает вызов, например, entityManager.persist(), этот вызов не вызывается напрямую у entity менеджера. Вместо этого бизнес метод вызывает прокси, который достает текущий entity менеджер из потока, в который его положил менеджер транзакций.

Как использовать:

В файле конфигурации нужно определить менеджер транзакций transactionManager для DataSource.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | <bean id="transactionManager" class="  org.springframework.jdbc.datasource  .DataSourceTransactionManager">  <property name="dataSource" ref="dataSource">  </property></bean> |

Включить поддержку аннотаций, добавив запись в контекстном xml файле вашего spring-приложения ИЛИ добавьте @EnableTransactionManagement в ваш конфигурационный файл.

Добавить аннотацию @Transactional в класс (метод класса) или интерфейс (метод интерфейса).

У @Transactional есть ряд параметров:

∙ уровень изоляции:

|  |  |
| --- | --- |
| 1.  2. | @Transactional(isolation =  Isolation.READ\_COMMITTED) |

∙ 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:

|  |  |
| --- | --- |
| 1. | @Transactional(timeout=60) |

∙ (Если не указано, распространяющееся поведение по умолчанию — REQUIRED) Указывает, что целевой метод не может работать без другой tx. Если tx уже запущен до вызова этого метода, то он будет продолжаться в том же tx, или новый tx начнется вскоре после вызова этого метода.

|  |  |
| --- | --- |
| 1. | @Transactional(propagation=Propagation.REQUIRED) |

∙ REQUIRES\_NEW — указывает, что новый tx должен запускаться каждый раз при вызове целевого метода. Если уже идет tx, он будет приостановлен, прежде чем запускать новый.  
∙ NESTED.  
∙ MANDATORY — указывает, что для целевого метода требуется активный tx. Если tx не будет продолжаться, он не сработает, выбросив исключение.  
∙ SUPPORTS — указывает, что целевой метод может выполняться независимо от tx. Если tx работает, он будет участвовать в том же tx. Если выполняется без tx, он все равно будет выполняться, если ошибок не будет. 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.

∙ 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 проверенных исключениях. Т. о., этот параметр можно использовать для декларативного отката транзакции, если происходит Checked Exception.

|  |  |
| --- | --- |
| 1. | @Transactional (rollbackFor=Exception.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. Но у аннотации @Transactional можно указать параметры.

|  |  |
| --- | --- |
| 1.  2. | @Transactional (noRollbackFor =  IllegalStateException.class) |

Распространение транзакций

Когда вызывается метод с @Transactional происходит особая уличная магия: proxy, который создал Spring, создает persistence context (или соединение с базой), открывает в нем транзакцию и сохраняет все это в контексте нити исполнения (натурально, в ThreadLocal). По мере надобности все сохраненное достается 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.

Т. о, 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 произойдет, если один метод с @Transactional вызовет другой метод с @Transactional? В Spring можно задать несколько вариантов поведения, которые называются правилами распространения.

∙ Propagation.REQUIRED — применяется по умолчанию. При входе в @Transactional метод будет использована уже существующая транзакция или создана новая транзакция, если никакой еще нет

∙ Propagation.REQUIRES\_NEW — второе по распространенности правило. Транзакция всегда создается при входе метод с Propagation.REQUIRES\_NEW, ранее созданные транзакции приостанавливаются до момента возврата из метода.

∙ Propagation.NESTED — корректно работает только с БД, которые умеют savepoints. При входе в метод в уже существующей транзакции создается savepoint, который по результатам выполнения метода будет либо сохранен, либо откачен. Все изменения, внесенные методом, подтвердятся только позднее, с подтверждением всей транзакции. Если текущей транзакции не существует, будет создана новая.

∙ Propagation.MANDATORY — обратный по отношению к Propagation.REQUIRES\_NEW: всегда используется существующая транзакция и кидается исключение, если текущей транзакции нет.

∙ Propagation.SUPPORTS — метод с этим правилом будет использовать текущую транзакцию, если она есть, либо будет исполнятся без транзакции, если ее нет.

∙ Propagation.NOT\_SUPPORTED — одно из самых забавных правил. При входе в метод текущая транзакция, если она есть, будет приостановлена и метод будет выполняться без транзакции.

∙ Propagation.NEVER — правило, которое явно запрещает исполнение в контексте транзакции. Если при входе в метод будет существовать транзакция, будет выброшено исключение.

Все эти правила действуют как при вызове метода в текущем потоке, так и выполнения в другом потоке. В случае другого потока транзакция будет относится к нему.

Куда же ставить @Transactional?

Классическое приложение обычно имеет многослойную архитектуру: контроллеры > слой логики > слой доступа к данным > слой ORM.

Где здесь место для @Transactional? Слой ORM обычно никто не пишет сам и использует какое-либо стандартное решение, в которое аннотации не вставишь.

Слой доступа к данным обычно представляет собой набор классов, методы которых реализуют тот или иной запрос. Получается, что если каждый метод аннотировать @Transactional, то, с одной стороны, работать это конечно будет, а с другой стороны теряется смысл транзакций, как логического объединения нескольких запросов в одну единицу работы. Ведь в таком случае у каждого метода, т. е. у каждого запроса, будет своя, собственная, транзакция.

Слой логики представляется идеальным местом для @Transactional: именно здесь набор запросов к базе оформляется в единую осмысленную операцию в приложении. Зная, что делает ваше приложение, вы можете четко разграничить логические единицы работы в нем и расставить границы транзакций.

Слой контроллеров тоже может быть неплохим местом для @Transactional, но у него есть два недостатка, по сравнению со слоем логики. Во‑первых, он взаимодействует с пользователем, напрямую или через сеть, что может делать транзакции длиннее: метод будет ждать отправки данных или реакции пользователя и в этом время продолжать удерживать транзакцию и связанные с ней блокировки. Во‑вторых, это нарушает принцип разделения ответственности: код, который должен быть ответственен за интерфейс с внешним миром, становится ответственен и за часть управления логикой приложения.

И последнее — никогда не аннотируйте интерфейсы. Аннотации не наследуются и поэтому, в зависимости от настроек Spring, вы можете внезапно оказаться фактически без своих @Transactional.

Транзакциями в Spring управляют с помощью Declarative Transaction Management (программное управление). Используется аннотация @Transactional для описания необходимости управления транзакцией. В файле конфигурации нужно добавить настройку transactionManager для DataSource.

# Жизненный цикл бина в Spring

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 все 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 бинов:

∙ Загрузка описаний бинов, создание графа зависимостей (между бинами).

∙ Создание и запуск BeanFactoryPostProcessors.

∙ Создание бинов.

∙ Spring внедряет значения и зависимости в свойства бина.

∙ Если бин реализует метод setBeanName() из интерфейса NameBeanAware, то ID бина передается в метод.

∙ Если бин реализует BeanFactoryAware, то Spring устанавливает ссылку на bean factory через setBeanFactory() из этого интерфейса.

∙ Если бин реализует интерфейс ApplicationContextAware, то Spring устанавливает ссылку на ApplicationContext через setApplicationContext().

∙ BeanPostProcessor — специальный интерфейс, и Spring позволяет бинам имплементировать этот интерфейс. Реализуя метод postProcessBeforeInitialization(), можно изменить экземпляр бина перед его (бина) инициализацией (установка свойств и т. п.).

∙ Если определены методы обратного вызова, то Spring вызывает их. Например, это метод, аннотированный @PostConstruct или метод initMethod из аннотации @Bean.

∙ Теперь бин готов к использованию. Его можно получить с помощью метода ApplicationContext#getBean().

∙ После того, как контекст будет закрыт (метод close() из ApplicationContext), бин уничтожается.

∙ Если в бине есть метод, аннотированный @PreDestroy, то перед уничтожением вызовется этот метод. Если бин имплементирует DisposibleBean, то Spring вызовет метод destroy(), чтобы очистить ресурсы или убить процессы в приложении. Если в аннотации @Bean определен метод destroyMethod, то вызовется и он.

Интерфейс BeanPostProcessor позволяют разработчику самому имплементировать некоторые методы бинов перед инициализацией и после уничтожения экземпляров бина. Имеется возможность настраивать несколько имплементаций BeanPostProcessor и определить порядок их выполнения. Данный интерфейс работает с экземплярами бинов, а это означает, что Spring IoC создает экземпляр бина, а затем BeanPostProcessor с ним работает. ApplicationContext автоматически обнаруживает любые бины, с реализацией BeanPostProcessor и помечает их как “post-processors” для того, чтобы создать их определенным способом.

Интерфейс BeanPostProcessor имеет всего два метода: postProcessBeforeInitialization и postProcessAfterInitialization

# Model, ModelMap, ModelAndView

Интерфейс Model инкапсулирует (объединяет) данные приложения. ModelMap реализует этот интерфейс, с возможностью передавать коллекцию значений. Затем он обрабатывает эти значения, как если бы они были внутри Map. Следует отметить, что в Model и ModelMap мы можем хранить только данные. Мы помещаем данные и возвращаем имя представления.

С другой стороны, с помощью ModelAndView мы возвращаем сам объект. Мы устанавливаем всю необходимую информацию, такую как данные и имя представления, в объекте, который мы возвращаем.

Model — интерфейс, он определяет держатель для атрибутов модели и в первую очередь предназначен для добавления атрибутов в модели. В то время как ModelMap — класс, реализация Map для использования при построении данных модели для использования с инструментами пользовательского интерфейса. Поддерживает цепные вызовы и генерацию атрибута модели имена.

ModelAndView — просто контейнер для ModelMap и объект представления. Это позволяет контроллеру возвращать оба как одно значение.

# Какие паттерны используются в Spring (singleton, prototype, builder, proxy, chain of responsibility, dependency injection)?

Вот некоторые известные паттерны, используемые в Spring Framework:

∙ Chain of Responsibility  — 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.

∙ Singleton (одиночка) — Паттерн Singleton гарантирует, что в памяти будет существовать только один экземпляр объекта, который будет предоставлять сервисы. Spring область видимости бина (scope) по умолчанию равна singleton и IoC‑контейнер создает ровно один экземпляр объекта на Spring IoC‑контейнер. Spring-контейнер будет хранить этот единственный экземпляр в кэше синглтон-бинов, и все последующие запросы и ссылки для этого бина получат кэшированный объект. Рекомендуется использовать область видимости singleton для бинов без состояния. Область видимости бина можно определить как singleton или как prototype (создается новый экземпляр при каждом запросе бина).

∙ Model View Controller (Модель-Представление-Контроллер) — преимущество Spring MVC в том, что ваши контроллеры являются POJO, а не сервлетами. Это облегчает тестирование контроллеров. Стоит отметить, что от контроллеров требуется только вернуть логическое имя представления, а выбор представления остается за ViewResolver. Это облегчает повторное использование контроллеров при различных вариантах представления.

∙ Front Controller (Контроллер запросов) — Spring предоставляет DispatcherServlet, чтобы гарантировать, что входящий запрос будет отправлен вашим контроллерам. 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 отделяет статическое содержимое в представлении, такое как JSP, от обработки бизнес-логики.

∙ Dependency injection и Inversion of control (IoC) (внедрение зависимостей и инверсия управления) — IoC‑контейнер в Spring, отвечает за создание объекта, связывание объектов вместе, конфигурирование объектов и обработку всего их жизненного цикла от создания до полного уничтожения. В контейнере Spring используется инъекция зависимостей (Dependency Injection, DI) для управления компонентами приложения. Эти компоненты называются "Spring-бины" (Spring Beans).

∙ 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 кэширование. 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. ApplicationContext является центральным интерфейсом в приложении Spring для предоставления информации о конфигурации приложения.

∙ Proxy (заместитель) — позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что‑то до или после передачи вызова оригиналу.

∙ Factory (фабрика) — определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

∙ Template (шаблон) — этот паттерн широко используется для работы с повторяющимся бойлерплейт кодом (таким как закрытие соединений и т. п.).

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

# Bean scopes

В Spring предусмотрены различные области времени действия бинов:

∙ singleton — может быть создан только один экземпляр бина. Этот тип используется спрингом по умолчанию, если не указано другое. Следует осторожно использовать публичные свойства класса, т. к. они не будут потокобезопасными.

∙ prototype — создается новый экземпляр при каждом запросе.

∙ request — аналогичен prototype, но название служит пояснением к использованию бина в веб приложении. Создается новый экземпляр при каждом HTTP request.

∙ session — новый бин создается в контейнере при каждой новой HTTP сессии.

∙ global-session: используется для создания глобальных бинов на уровне сессии для Portlet приложений.

В Spring 5:  
∙ singleton  
∙ prototype  
∙ request  
∙ session  
∙ application - один экземпляр будет создан и доступен в течение всего жизненного цикла ServletContext  
∙ websocket - один экземпляр будет создан и доступен в течение всего жизненного цикла WebSocket.

# Spring Data

Spring Data — дополнительный удобный механизм для взаимодействия с сущностями БД, 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 — репозиторий. Это несколько интерфейсов которые используют JPA Entity для взаимодействия с ней. Так, например, интерфейс

|  |  |
| --- | --- |
| 1.  2.  3.  4. | 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.

# Spring Security

Spring Security предоставляет широкие возможности для защиты приложения. Кроме стандартных настроек для аутентификации, авторизации и распределения ролей и маппинга доступных страниц, ссылок и т. п., предоставляет защиту от различных вариантов атак

∙ SecurityContextHolder, в нем содержится информация о текущем контексте безопасности приложения, который включает в себя подробную информацию о пользователе (Principal) работающем в настоящее время с приложением. По умолчанию SecurityContextHolder использует ThreadLocal для хранения такой информации, что означает, что контекст безопасности всегда доступен для методов исполняющихся в том же самом потоке. Для того чтобы изменить стратегию хранения этой информации можно воспользоваться статическим методом класса SecurityContextHolder.setStrategyName(String strategy).

∙ SecurityContext, содержит объект Authentication и в случае необходимости информацию системы безопасности, связанную с запросом от пользователя.

∙ Authentication представляет пользователя (Principal) с точки зрения Spring Security.

∙ GrantedAuthority отражает разрешения выданные пользователю в масштабе всего приложения, такие разрешения (как правило называются «роли»), например ROLE\_ANONYMOUS, ROLE\_USER, ROLE\_ADMIN.

∙ UserDetails предоставляет необходимую информацию для построения объекта Authentication из DAO объектов приложения или других источников данных системы безопасности. Объект UserDetails содержит имя пользователя, пароль, флаги: isAccountNonExpired, isAccountNonLocked, isCredentialsNonExpired, isEnabled и Collection — прав (ролей) пользователя.

∙ UserDetailsService, используется чтобы создать UserDetails объект путем реализации единственного метода этого интерфейса

Позволяет получить из источника данных объект пользователя и сформировать из него объект UserDetails который будет использоваться контекстом Spring Security.

Аннотации:

∙ @EnableGlobalMethodSecurity — включает глобальный метод безопасности.

∙ @EnableWebMvcSecurity — "включает" Spring Security. Не будет работать, если наш класс не наследует WebSecurityConfigurerAdapter

∙ @Secured используется для указания списка ролей в методе.

∙ @PreAuthorize и @PostAuthorize обеспечивают контроль доступа на основе выражений. @PreAuthorize проверяет данное выражение перед входом в метод, тогда как аннотация @PostAuthorize проверяет его после выполнения метода и может изменить результат.

∙ @PreFilter для фильтрации аргумента коллекции перед выполнением метода

# @Recource, @Autowired, @Inject

Аннотации для внедрения зависимостей.

@Resource (java) пытается получить зависимость: по имени, по типу, затем по описанию. Имя извлекается из имени аннотируемого сеттера или поля, либо берется из параметра name.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31. | package javax.annotation;  import java.lang.annotation.\*;  import static java.lang.annotation.ElementType.\*;  import static  java.lang.annotation.RetentionPolicy.\*;  @Target({TYPE, FIELD, METHOD})  @Retention(RUNTIME)  public @interface Resource {  String name() default "";  String lookup() default "";  Class<?> type() default java.lang.Object.class;  enum AuthenticationType {  CONTAINER,  APPLICATION  }  AuthenticationType authenticationType() default AuthenticationType.CONTAINER;  boolean shareable() default true;  String mappedName() default "";  String description() default "";  } |

@Inject (javax.inject!!!) или @Autowired (spring) в первую очередь пытается подключить зависимость по типу, затем по описанию и только потом по имени.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | package javax.inject;  import java.lang.annotation.Documented;  import java.lang.annotation.ElementType;  import java.lang.annotation.Retention;  import java.lang.annotation.RetentionPolicy;  import java.lang.annotation.Target;  @Target({  ElementType.METHOD,  ElementType.CONSTRUCTOR,  ElementType.FIELD})  @Retention(RetentionPolicy.RUNTIME)  @Documented  public @interface Inject {  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20. | package org.springframework.beans.factory  .annotation;  import java.lang.annotation.Documented;  import java.lang.annotation.ElementType;  import java.lang.annotation.Retention;  import java.lang.annotation.RetentionPolicy;  import java.lang.annotation.Target;  @Target({  ElementType.CONSTRUCTOR,  ElementType.METHOD,  ElementType.PARAMETER,  ElementType.FIELD,  ElementType.ANNOTATION\_TYPE})  @Retention(RetentionPolicy.RUNTIME)  @Documented  public @interface Autowired {  boolean required() default true;  } |

Для @Inject аналог @Named, для @Autowired идет @Qulifier.

# @Conditional

@Conditional позволяет накладывать определенные условия, выполнив которые, можно, например, инициализировать бин. Пример: нам нужен определенные бин только когда выполняется некое условие. Мы прописываем в этом бине @Conditional, и, если оно выполняется, (true) бин создастся. Существует много готовых решений @Conditional:

ConditionalOnBean — условие выполняется, если присутствует нужный бин в BeanFactory;

ConditionalOnClass — условие выполняется, если нужный класс есть в classpath;

ConditionalOnCloudPlatform — условие выполняется, когда активна определенная платформа;

ConditionalOnExpression — условие выполняется, когда SpEL выражение вернуло положительное значение;

ConditionalOnJava — условие выполняется, когда приложение запущено с определенной версией JVM;

ConditionalOnJndi — условие выполняется, только если через JNDI доступен определенный ресурс;

ConditionalOnMissingBean — условие выполняется, если нужный бин отсутствует в BeanFactory;

ConditionalOnMissingClass — условие выполняется, если нужный класс отсутствует в classpath;

ConditionalOnNotWebApplication — условие выполняется, если контекст приложения не является веб контекстом;

ConditionalOnProperty — условие выполняется, если в файле настроек заданы нужные параметры;

ConditionalOnResource — условие выполняется, если присутствует нужный ресурс в classpath;

ConditionalOnSingleCandidate — условие выполняется, если bean-компонент указанного класса уже содержится в BeanFactory и он единственный;

ConditionalOnWebApplication — условие выполняется, если контекст приложения является веб контекстом.

# @ComponentScan

Первый шаг для описания Spring Beans — добавление аннотации @Component, @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, называется @ComponentScan. В @ComponentScan вы указываете пакеты, которые должны сканироваться.

Spring будет искать бины не только в пакетах для сканирования, но и в их подпакетах.

Аннотация @SpringBootApplication включает в себя три аннотации:

|  |  |
| --- | --- |
| 1.  2.  3. | @Configuration  @EnableAutoConfiguration  @ComponentScan |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.  37.  38.  39.  40.  41.  42.  43.  44.  45.  46.  47.  48.  49.  50.  51.  52.  53.  54.  55.  56.  57.  58.  59.  60.  61.  62.  63.  64.  65.  66. | package org.springframework.context.annotation;  import java.lang.annotation.Documented;  import java.lang.annotation.ElementType;  import java.lang.annotation.Repeatable;  import java.lang.annotation.Retention;  import java.lang.annotation.RetentionPolicy;  import java.lang.annotation.Target;  import org.springframework.beans.factory.support  .BeanNameGenerator;  import org.springframework.core.annotation  .AliasFor;  @Retention(RetentionPolicy.RUNTIME)  @Target({ElementType.TYPE})  @Documented  @Repeatable(ComponentScans.class)  public @interface ComponentScan {  @AliasFor("basePackages")  String[] value() default {};  @AliasFor("value")  String[] basePackages() default {};  Class<?>[] basePackageClasses() default {};  Class<? extends BeanNameGenerator>  nameGenerator()  default BeanNameGenerator.class;  Class<? extends ScopeMetadataResolver>  scopeResolver()  default  AnnotationScopeMetadataResolver  .class;  ScopedProxyMode scopedProxy()  default ScopedProxyMode.DEFAULT;  String resourcePattern() default "\*\*/\*.class";  boolean useDefaultFilters() default true;  ComponentScan.Filter[] includeFilters()  default {};  ComponentScan.Filter[] excludeFilters()  default {};  boolean lazyInit() default false;  @Retention(RetentionPolicy.RUNTIME)  @Target({})  public @interface Filter {  FilterType type()  default FilterType.ANNOTATION;  @AliasFor("classes")  Class<?>[] value() default {};  @AliasFor("value")  Class<?>[] classes() default {};  String[] pattern() default {};  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.  37.  38.  39.  40.  41.  42.  43.  44.  45.  46.  47.  48.  49.  50.  51.  52.  53.  54.  55.  56.  57.  58.  59.  60.  61.  62.  63.  64.  65.  66.  67.  68.  69.  70.  71.  72.  73.  74.  75.  76.  77.  78. | package org.springframework.boot.autoconfigure;  import java.lang.annotation.Documented;  import java.lang.annotation.ElementType;  import java.lang.annotation.Inherited;  import java.lang.annotation.Retention;  import java.lang.annotation.RetentionPolicy;  import java.lang.annotation.Target;  import org.springframework.beans.factory.support  .BeanNameGenerator;  import org.springframework.boot  .SpringBootConfiguration;  import org.springframework.boot.context  .TypeExcludeFilter;  import org.springframework.context.annotation  .ComponentScan;  import org.springframework.context.annotation  .Configuration;  import org.springframework.context.annotation  .FilterType;  import org.springframework.context.annotation  .ComponentScan.Filter;  import org.springframework.core.annotation  .AliasFor;  @Target({ElementType.TYPE})  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Inherited  @SpringBootConfiguration  @EnableAutoConfiguration  @ComponentScan(  excludeFilters = {@Filter(  type = FilterType.CUSTOM,  classes = {TypeExcludeFilter.class}  ), @Filter(  type = FilterType.CUSTOM,  classes =  {AutoConfigurationExcludeFilter.class}  )}  )  public @interface SpringBootApplication {  @AliasFor(  annotation = EnableAutoConfiguration.class  )  Class<?>[] exclude() default {};  @AliasFor(  annotation = EnableAutoConfiguration.class  )  String[] excludeName() default {};  @AliasFor(  annotation = ComponentScan.class,  attribute = "basePackages"  )  String[] scanBasePackages() default {};  @AliasFor(  annotation = ComponentScan.class,  attribute = "basePackageClasses"  )  Class<?>[] scanBasePackageClasses()  default {};  @AliasFor(  annotation = ComponentScan.class,  attribute = "nameGenerator"  )  Class<? extends BeanNameGenerator>  nameGenerator()  default BeanNameGenerator.class;  @AliasFor(  annotation = Configuration.class  )  boolean proxyBeanMethods() default true;  } |

# Spring Boot

По сути, Spring Boot — это просто набор классов конфигурации, которые создают нужные бины в контексте. Точно так же их можно создать руками, просто Boot это автоматизирует. При этом помогая решить проблему конфликтов разных версий компонентов.

Чтобы ускорить процесс управления зависимостями, Spring Boot неявно упаковывает необходимые сторонние зависимости для каждого типа приложения на основе Spring и предоставляет их разработчику посредством так называемых starter-пакетов (spring-boot-starter-web, spring-boot-starter-data-jpa и т. д.)

Starter-пакеты представляют собой набор удобных дескрипторов зависимостей, которые можно включить в свое приложение. Это позволит получить универсальное решение для всех, связанных со Spring технологий, избавляя программиста от лишнего поиска примеров кода и загрузки из них требуемых дескрипторов зависимостей. Например, если вы хотите начать использовать Spring Data JPA для доступа к БД, просто включите в свой проект зависимость spring-boot-starter-data-jpa и все будет готово (вам не придется искать совместимые драйверы БД и библиотеки Hibernate)

В основе "магии" Spring Boot нет ничего магического, он использует совершенно базовые понятия из Spring Framework. В кратком виде процесс можно описать так:

∙ Аннотация @SpringBootApplication включает сканирование компонентов и авто-конфигурацию через аннотацию @EnableAutoConfiguration.

∙ @EnableAutoConfiguration импортирует класс EnableAutoConfigurationImportSelector.

∙ EnableAutoConfigurationImportSelector загружает список конфигураций из файла META-INF/spring.factories.

∙ Каждая конфигурация пытается сконфигурировать различные аспекты приложения (web, JPA, AMQP etc), регистрируя нужные бины и используя различные условия (наличие/отсутствие бина, настройки, класса и т. п.)

∙ Созданный в итоге AnnotationConfigEmbeddedWebApplicationContext ищет в том же DI контейнере фабрику для запуска embedded servlet container.

∙ Servlet container запускается, приложение готово к работе!

Важное понятие Spring Boot — автоконфигурация. По сути, это просто набор конфигурационных классов, которые создают и регистрируют определенные бины в приложении. По большому счету, даже сам Embedded Servlet Container — это просто еще один бин, который можно сконфигурировать! Пара важных моментов, которые важно знать об автоконфигурации:

∙ Включается аннотацией @EnableAutoConfiguration.

∙ Работает в последнюю очередь, после регистрации пользовательских бинов.

∙ Принимает решения о конфигурации на основании доступных в classpath классов, свойств в application.properties и т. п.

∙ Можно включать и выключать разные аспекты автоконфигурации, и применять ее частично (например, только MySQL + JPA, но не веб)

∙ Всегда отдает приоритет пользовательским бинам. Если ваш код уже зарегистрировал бин DataSource — автоконфигурация не будет его перекрывать

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.

Отключить ненужные автоконфигурации можно при помощи свойств exclude и excludeName аннотаций @EnableAutoConfiguration, @ImportAutoConfiguration и @SpringBootApplication. Или в property задать SpringAutoconfiguration exclude и передать имена классов.

Можно отказаться от использования механизма автоконфигурации, вместо этого указывая необходимые автоконфигурации вручную. Для этого надо избавиться от аннотаций @SpringBootApplication и @EnableAutoConfiguration в коде вашего проекта, а для указания нужных конфигурационных классов использовать аннотации @SpringBootConfiguration и @ImportAutoConfiguration. Однако стоит помнить, что используемые автоконфигурации все еще могут содержать неиспользуемые компоненты.

# 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

# Spring Cloud (Data Flow)

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

# 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.

# 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.

# 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.

# Spring Rest Docs

Spring REST Docs поможет вам документировать сервисы RESTful.

Он сочетает в себе рукописную документацию, написанную с помощью Asciidoctor, и автоматически генерируемые фрагменты, созданные с помощью Spring MVC Test. Такой подход освобождает вас от ограничений документации, создаваемой такими инструментами, как Swagger.

Это помогает вам создавать документацию, которая является точной, краткой и хорошо структурированной. Затем эта документация позволяет вашим пользователям получать необходимую информацию с минимальными усилиями.

# 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.

# 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.

# 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.

# Контейнеры спринга

Container создает 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 — самый простой контейнер, который обеспечивает базовую поддержку DI и который основан на интерфейсе org.springframework.beans.factory.BeanFactory. Такие интерфейсы, как BeanFactoryAware и DisposableBean все еще 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 определенным 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.

Groovy — объектно-ориентированный язык программирования разработанный для платформы Java как альтернатива языку Java с возможностями Python, Ruby и Smalltalk. Groovy использует Java-подобный синтаксис с динамической компиляцией в JVM байткод и напрямую работает с другим Java кодом и библиотеками. Язык может использоваться в любом Java проекте или как скриптовый язык.

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 — 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.

# Части спринга, модули

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 определен 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.

Inversion of Control — контейнер: конфигурирование компонентов приложений и управление жизненным циклом Java-объектов.

Фреймворк аспектно-ориентированного программирования: работает с функциональностью, которая не может быть реализована возможностями объектно-ориентированного программирования на Java без потерь.

Фреймворк доступа к данным: работает с системами управления реляционными БД на Java-платформе, используя JDBC- и ORM-средства и обеспечивая решения задач, которые повторяются в большом числе Java-based environments.

Фреймворк управления транзакциями: координация различных API управления транзакциями и инструментарий настраиваемого управления транзакциями для объектов Java.

Фреймворк MVC: каркас, основанный на HTTP и сервлетах, предоставляющий множество возможностей для расширения и настройки (customization).

Фреймворк удаленного доступа: конфигурируемая передача Java-объектов через сеть в стиле RPC, поддерживающая RMI, CORBA,HTTP-based протоколы, включая web‑сервисы (SOAP).

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).

Фреймворк удаленного управления: конфигурируемое представление и управление Java-объектами для локальной или удаленной конфигурации с помощью JMX.

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.

Тестирование: каркас, поддерживающий классы для написания модульных и интеграционных тестов.

# АОР

Аспектно-ориентированное программирование (АОП) — 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) — 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.

Объединенная точка (Join point) — такая точка в приложении, где мы можем подключить аспект. Другими словами, это место, где начинаются определенные действия модуля АОП в Spring.

Совет (Advice) — 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 объединенных точек (join points), в котором должен быть выполнен совет.

Введение (Introduction) — сущность, которая помогает нам добавлять новые атрибуты и/или методы в уже существующие классы.

Целевой объект (Target object) — объект на который направлены один или несколько аспектов.

Плетение (Weaving) — 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.

# Spring Framework

Spring Framework (или коротко Spring) — универсальный фреймворк с открытым исходным кодом для Java-платформы. Spring можно использовать для построения любого приложения на языке Java (т. е. автономных, веб‑приложений, приложений JEE и т. д.). Характеристика “облегченная” в действительности не имеет никакого отношения к количеству классов или размеру дистрибутива; напротив, она определяет принцип всей философии Spring — минимальное воздействие. Платформа Spring является облегченной в том смысле, что для использования ядра Spring вы должны вносить минимальные (если вообще какие-либо) изменения в код своего приложения, а если в какой‑то момент вы решите больше не пользоваться ядром Spring, то и это сделать очень просто.

# BeanFactory и ApplicationContext разница

Разница между (BeanFactory) и контейнером Advanced J2EE (ApplicationContext) заключается в следующем:

BeanFactory создаст объекты для bean-компонентов (т. е. для классов POJO), упомянутых в файле spring.xml (<bean></bean>) только когда вы вызываете метод .getBean(), но ApplicationContext создает объекты для всех bean-компонентов (<bean></bean> если его область явно не указана как «Prototype»), настроенная в spring.xml при загрузке самого файла spring.xml.

BeanFactory: (Ленивый контейнер, потому что он создает объекты для bean-компонентов только при явном вызове из пользовательского / основного класса)

ApplicationContext: (нетерпеливый контейнер из‑за создания объектов всех одноэлементных компонентов при загрузке самого файла spring.xml)

Технически рекомендуется использовать ApplicationContext, поскольку в приложениях реального времени объекты bean-объектов будут создаваться во время запуска приложения на самом сервере. Это сокращает время ответа на запрос пользователя, поскольку объекты уже доступны для ответа.

# Работа спринга с ДАО

Spring DAO предоставляет возможность работы с доступом к данным с помощью технологий вроде JDBC, Hibernate в удобном виде. Существуют специальные классы: JdbcDaoSupport, HibernateDaoSupport, JdoDaoSupport, JpaDaoSupport.

В Spring DAO поддерживается иерархия исключений, что помогает не обрабатывать некоторые исключения.

# Что такое контроллер

Ключевым интерфейсом в Spring MVC является Controller. Контроллер обрабатывает запросы к действиям, осуществляемые пользователями в пользовательском интерфейсе, взаимодействуя с уровнем обслуживания, обновляя модель и направляя пользователей на соответствующие представления в зависимости от результатов выполнения. Controller — управление, связь между моделью и видом.

Основным контроллером в Spring MVC является org.springframework.web.servlet.DispatcherServlet. Задается аннотацией @Controller и часто используется с аннотацией @RequestMapping, которая указывает какие запросы будут обрабатываться этим контроллером.

# View-resolver

ViewResolver — распознаватель представлений. Интерфейс ViewResolver в Spring MVC (из пакета org.springframework.web.servlet) поддерживает распознавание представлений на основе логического имени, возвращаемого контроллером. Для поддержки различных механизмов распознавания представлений предусмотрено множество классов реализации. Например, класс UrlBasedViewResolver поддерживает прямое преобразование логических имен в URL. Класс ContentNegotiatingViewResolver поддерживает динамическое распознавание представлений в зависимости от типа медиа, поддерживаемого клиентом (XML, PDF, JSON и т. д.). Существует также несколько реализаций для интеграции с различными технологиями представлений, такими как FreeMarker (FreeMarkerViewResolver), Velocity (VelocityViewResolver) и JasperReports (JasperReportsViewResolver).

InternalResourceViewResolver — реализация ViewResolver, которая позволяет находить представления, которые возвращает контроллер для последующего перехода к нему. Ищет по заданному пути, префиксу, суффиксу и имени.

# model.put разница model.addAttribute

Метод addAttribute отделяет нас от работы с базовой структурой hashmap. По сути addAttribute это обертка над put, где делается дополнительная проверка на null. Метод addAttribute в отличии от put возвращает modelmap.

model.addAttribute(“attribute1”,”value1”).addAttribute(“attribute2”,”value2”);

# Form Binding

Нам это может понадобиться, если мы, например, захотим взять некоторое значение с HTML страницы и сохранить его в БД. Для этого нам надо это значение переместить в контроллер Спринга.

# 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.

# 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.

# parent in pom.xml

В pom.xml дочерних проектов необходимо ввести секцию <parent> и определить GAV-параметры родительского проекта.

# @Sheduler

Шедулер — управляет таймерами запуска задач — планировщик задач.

Джоб — конкретная задача, запускаемая по таймеру

Триггер — условие выполнения задачи — задает временные рамки запуска задач и их выполнения

pauseJob(String name, String Group) — остановить выполнение задачи шедулера в указанной группе джобов. Остановка происходит путем остановки соответствующего триггера (см. pauseTrigger)

resumeJob(String name, String Group) — возобновить выполнение задачи шедулера в указанной группе джобов. Восстановление происходит путем запуска соответствующего триггера (см. resumeTrigger)

pauseTrigger(String name, String Group) — останавливает триггер в соответствующей группе

resumeTrigger(String name, String Group) — возобновляет работу триггера в соответствующей группе

pauseAll — останавливает все задачи шедулера (pauseJobs(String group) — только у конкретной группы)

resumeAll — возобновляет запуск всех задач шедулера (см. также resumeJobs)

Можно запланировать удаление пользователей из БД, например в каждую среду. Так же может использоваться для логирования, пишем логи по расписанию.

Планировщик задач. Включается @EnableSheduller в конфигах.

# Внедрение в синглтон prototype

@lookup аннотация позволяет создавать прототип бины через синглтон

через application context

Через прокси”

# @Profile

Профили — основная особенность фреймворка, позволяющая нам отображать наши bean-компоненты на разные профили, например, dev, test, prod.

Рассмотрим базовый сценарий — у нас есть бин, который должен быть активен только во время разработки, но не развернут в продакшене. Мы аннотируем этот бин профилем ”dev", и он будет присутствовать только в контейнере во время разработки — в продакшене dev просто не будет активен.

# Транзакция с аннотацией @Transactional вызывает метод без аннотации

Обработано не будет из‑за прокси

# @Around

Запускает совет до и после выполнения метода.

# SPELL

Spring Expression Language (SpEL) — мощный язык выражений, который поддерживает запросы и манипулирование графом объектов во время выполнения. Он может использоваться с конфигурациями Spring на основе XML или аннотаций.

# Container — как устроен (Map)

# FileSystemApplicationContext

FileSystemXmlApplicationContext может получить доступ ко всей вашей файловой системе, например c:/config/applicationContext.xml.

# Bean Definition

BeanDefinition — 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.

# Статические методы в бинах

https://www.youtube.com/watch?v=nGfeSo52\_8A&t=1735s (начало)

# ContextLoaderListener

ContextLoaderListener — экземпляр, который загружает ваш WebApplicationContext, который по умолчанию использует класс XmlWebApplicationContext. А он в свою очередь разбирает настройки для корректной работы Spring.

# HandlerException

Обработка исключений до Spring 3.2

Вы можете добавить дополнительные (@ExceptionHandler) методы к любому контроллеру для специальной обработки исключений, вызванных методами обработки запросов (@RequestMapping) в том же контроллере. Такие методы могут:

∙ Обрабатывать исключения без @ResponseStatus аннотации (обычно предопределенные исключения, которые вы не написали)

∙ Перенаправить пользователя в специальный просмотр ошибок

∙ Создайте полностью индивидуальный ответ об ошибке

# @ControllerAdvice

Обработка исключений до Spring 3.2

Классы, помеченные как @ControllerAdvice могут быть явно объявлены как бины Spring или автоматически обнаружены посредством сканирования пути к классам. Все такие bean-компоненты сортируются на основе Ordered семантики или @Order/@Priority объявлений, причем Ordered семантика имеет приоритет над @Order/@Priority объявлениями. @ControllerAdvice бины затем применяются в этом порядке во время выполнения. Однако обратите внимание, @ControllerAdvice что реализующим PriorityOrdered компонентам не предоставляется приоритет над @ControllerAdvice реализуемыми компонентами Ordered. Кроме того, Ordered не учитывается для @ControllerAdvice bean — объектов с ограниченным диапазоном — например, если такой bean-компонент был сконфигурирован как bean-объект с областью запроса или сессионный объем. Для обработки исключений, @ExceptionHandler будет выбран по первому совету с подходящим методом обработчика исключений. Для атрибутов модели и инициализации привязки данных, @ModelAttribute а также @InitBinder методы будут следовать @ControllerAdvice порядку.

# @ModelAttribute над методом

# Реактивное программирование и 4 принципа

Реактивное программирование — 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 — 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. Т. е. функции, которые мы определили это наблюдатели (observers). А поток является субъектом который наблюдают.

Критерии реактивного приложения:

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 БД, вынимаем необходимый объем информации и отдаем ее пользователю. Здесь все хорошо, если наша система достаточно быстрая и БД 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.

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.

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.

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.

# WebSocket

WebSocket обеспечивает двустороннюю связь между клиентом и сервером, используя одно TCP соединение.

# @Lookup

Используется для внедрения prototype bean в singleton bean.

ПРИМЕР — 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. Т. е. 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).

# @Target и @Retention

@Retention — указываем, в какой момент жизни программного кода будет доступна аннотация

∙ SOURCE — аннотация доступна только в исходном коде и сбрасывается во время создания .class файла;

∙ CLASS — аннотация хранится в .class файле, но недоступна во время выполнения программы;

∙ RUNTIME — аннотация хранится в .class файле и доступна во время выполнения программы.

@Target — указывается, какой элемент программы будет использоваться аннотацией

∙ PACKAGE — назначением является целый пакет (package);

∙ TYPE — класс, интерфейс, enum или другая аннотация:

∙ METHOD — метод класса, но не конструктор (для конструкторов есть отдельный тип CONSTRUCTOR);

∙ PARAMETER – параметр метода;

∙ CONSTRUCTOR — конструктор;

∙ FIELD — поля-свойства класса;

∙ LOCAL\_VARIABLE — локальная переменная (обратите внимание, что аннотация не может быть прочитана во время выполнения программы, т. е., данный тип аннотации может использоваться только на уровне компиляции как, например, аннотация @SuppressWarnings);

∙ ANNOTATION\_TYPE — другая аннотация.

# Bean

Бин (bean) — не что иное, как самый обычный объект. Разница лишь в том, что бинами принято называть те объекты, которые управляются Spring-ом и живут внутри его DI-контейнера.

По умолчанию бин задается как синглтон в Spring. Т. о. все публичные переменные класса могут быть изменены одновременно из разных мест, а значит бин — не потокобезопасен. Однако поменяв область действия бина на request, prototype, session он станет потокобезопасным, но это скажется на производительности.

Конфигурационный файл спринг определяет все бины, которые будут инициализированы в Spring Context. При создании экземпляра Spring ApplicationContext будет прочитан конфигурационный xml файл и выполнены указанные в нем необходимые инициализации. Отдельно от базовой конфигурации, в файле могут содержаться описание перехватчиков (interceptors), view resolvers, настройки локализации и др.

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 - 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 — свойство определяет область видимости создаваемых объектов. singleton - Определяет один единственный бин для каждого контейнера Spring IoC (используется по умолчанию); prototype — контейнер Spring IoC создает новый экземпляр бина на каждый полученный запрос т. е. иметь любое количество экземпляров бина; request — Создается один экземпляр бина на каждый HTTP запрос. Касается исключительно ApplicationContext; session — создается один экземпляр бина на каждую HTTP сессию. Касается исключительно ApplicationContext; websocket — создается один экземпляр бина для определенного сокета. Application — создается один экземпляр бина для жизненного цикла бина. Похоже на синглтон, но когда бины ограничены областью приложения, значения, однажды установленное в applicationScopedBean, будет сохранено для всех последующих запросов, сеансов и даже для другого приложения сервлета, которое будет обращаться к этому бину, при условии, что оно выполняется в том же ServletContext. В то время как одноэлементные бины ограничены только одним контекстом приложения.

∙ 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 дает контейнеру IoC команду создавать экземпляр бина при первом запросе, а не при запуске приложения.

# Прокси

Spring AOP использует динамические прокси JDK или CGLIB для создания прокси для данного целевого объекта. (Динамические прокси JDK предпочтительны, когда у вас есть выбор).

Если целевой объект для прокси реализует по крайней мере один интерфейс, то будет использоваться динамический прокси JDK. Все интерфейсы, реализованные целевым типом, будут проксированы. Если целевой объект не реализует никаких интерфейсов, будет создан прокси-сервер CGLIB.

Если вы хотите принудительно использовать прокси-сервер CGLIB (например, прокси-сервер для каждого метода, определенного для целевого объекта, а не только для тех, которые реализованы его интерфейсами), вы можете сделать это. Тем не менее, есть несколько вопросов для рассмотрения:

∙ final нельзя рекомендовать методы, т. к. они не могут быть переопределены.

∙ Вам понадобятся двоичные файлы CGLIB 2 на вашем пути к классам, в то время как динамические прокси доступны с JDK. Spring автоматически предупредит вас, когда ему нужен CGLIB, а классы библиотеки CGLIB не найдены в пути к классам.

∙ Конструктор вашего прокси-объекта будет вызываться дважды. Это естественное следствие прокси-модели CGLIB, согласно которой для каждого прокси-объекта создается подкласс. Для каждого экземпляра с прокси создаются два объекта: фактический объект с прокси и экземпляр подкласса, который реализует рекомендацию. Такое поведение не проявляется при использовании прокси-серверов JDK. Обычно, вызов конструктора прокси-типа дважды не является проблемой, т. к. обычно выполняются только присваивания, и в конструкторе не реализована реальная логика.

# Spring MVC

Spring имеет собственную MVC‑платформу веб‑приложений, которая не была первоначально запланирована. Spring MVC является фреймворком, ориентированным на запросы. В нем определены стратегические интерфейсы для всех функций современной запросно-ориентированной системы. Цель каждого интерфейса — быть простым и ясным, чтобы пользователям было легко его заново имплементировать, если они того пожелают. MVC прокладывает путь к более чистому front-end-коду. Все интерфейсы тесно связаны с Servlet API. Эта связь рассматривается некоторыми как неспособность разработчиков Spring предложить для веб‑приложений абстракцию более высокого уровня. Однако, эта связь оставляет особенности Servlet API доступными для разработчиков, облегчая все же работу с ним. Наиболее важные интерфейсы, определенные Spring MVC, перечислены ниже:

HandlerMapping: выбор класса и его метода, которые должны обработать данный входящий запрос на основе любого внутреннего или внешнего для этого запроса атрибута или состояния.

HandlerAdapter: вызов и выполнение выбранного метода обработки входящего запроса.

Controller: включен между Моделью (Model) и Представлением (View). Управляет процессом преобразования входящих запросов в адекватные ответы. Действует как ворота, направляющие всю поступающую информацию. Переключает поток информации из модели в представление и обратно.

Класс DispatcherServlet является главным контроллером, которые 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, 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.

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. Настраивается в web.xml

Model: Этот блок инкапсулирует (объединяет) данные приложения. На практике это POJO-классы.

View: ответственно за возвращение ответа клиенту в виде текстов и изображений. Некоторые запросы могут идти прямо во View, не заходя в Model; другие проходят через все три слоя.

ViewResolver: выбор, какое именно View должно быть показано клиенту. Поддерживает распознавание представлений на основе логического имени, возвращаемого контроллером. Для поддержки различных механизмов распознавания представлений предусмотрено множество классов реализации. Например, класс UrlBasedViewResolver поддерживает прямое преобразование логических имен в URL.

Класс ContentNegotiatingViewResolver поддерживает динамическое распознавание представлений в зависимости от типа медиа, поддерживаемого клиентом (XML, PDF, JSON и т. д.). Существует также несколько реализаций для интеграции с различными технологиями представлений, такими как FreeMarker (FreeMarkerViewResolver), Velocity (VelocityViewResolver) и JasperReports (JasperReportsViewResolver).

HandlerInterceptor: перехват входящих запросов. Сопоставим, но не эквивалентен сервлет-фильтрам (использование не является обязательным и не контролируется DispatcherServlet-ом).

LocaleResolver: получение и, возможно, сохранение локальных настроек (язык, страна, часовой пояс) пользователя.

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.

Spring MVC предоставляет разработчику следующие возможности:

Ясное и прозрачное разделение между слоями в MVC и запросах.

Стратегия интерфейсов — каждый интерфейс делает только свою часть работы.

Интерфейс всегда может быть заменен альтернативной реализацией.

Интерфейсы тесно связаны с Servlet API.

Высокий уровень абстракции для веб‑приложений.

В веб‑приложениях можно использовать различные части Spring, а не только Spring MVC.

# Starter-pack Spring Boot — как создать?

https://www.youtube.com/watch?v=nGfeSo52\_8A&t=1735s (с 30мин)

# SOAP и REST

SOAP Simple Object Access Protocol (простой протокол доступа к объектам) — целое семейство протоколов и стандартов, для обмена структурированными сообщениями основанными на XML. Это более тяжеловесный и сложный вариант с точки зрения машинной обработки. Поэтому REST работает быстрее.

REST Representational State Transfer (передача состояния представления) — не протокол и не стандарт, а архитектурный стиль.

Специфика SOAP — формат обмена данными. С SOAP это всегда SOAP-XML, который представляет собой XML, включающий:  
∙ Envelope (конверт) — корневой элемент, который определяет сообщение и пространство имен, использованное в документе,  
∙ Header (заголовок) — содержит атрибуты сообщения, например: информация о безопасности или о сетевой маршрутизации,  
∙ Body (тело) — содержит сообщение, которым обмениваются приложения,  
∙ Fault — необязательный элемент, который предоставляет информацию об ошибках, которые произошли при обработке сообщений. И запрос, и ответ должны соответствовать структуре SOAP.  
∙ Специфика REST — использование HTTP в качестве транспортного протокола. Он подразумевает наилучшее использование функций, предоставляемых HTTP — методы запросов, заголовки запросов, ответы, заголовки ответов и т. д.

Особенности REST:  
∙ Наличие Клиентов и Серверов.  
∙ Отсутствие состояний. У клиентов и серверов нет необходимости отслеживать состояния друг друга.  
∙ Единообразие интерфейса. Это достигается через 4 ограничения: идентификацию ресурсов, манипуляцию ресурсами через представления, «самодостаточные» сообщения и гипермедиа.  
∙ Кэширование,   
∙ Система слоев  
∙ Код по требованию

# Spring Data — что под капотом

# Spring Security — что под капотом

# @Secured

@Secured используется для определения списка атрибутов конфигурации безопасности для бизнес — методов. Эта аннотация может использоваться как альтернатива Java 5.

# Как исключить класс, автоконфигурацию класса в Spring Boot

Отключить ненужные автоконфигурации можно при помощи свойств exclude и excludeName аннотаций @EnableAutoConfiguration, @ImportAutoConfiguration и @SpringBootApplication. Или в property задать SpringAutoconfiguration exclude и передать имена классов.

Можно отказаться от использования механизма автоконфигурации, вместо этого указывая необходимые автоконфигурации вручную. Для этого надо избавиться от аннотаций @SpringBootApplication и @EnableAutoConfiguration в коде вашего проекта, а для указания нужных конфигурационных классов использовать аннотации @SpringBootConfiguration и @ImportAutoConfiguration. Однако стоит помнить, что используемые автоконфигурации все еще могут содержать неиспользуемые компоненты.

# Как заинжектить примитив

@Value над полем — внедряем примитив.

# Как заинжектить коллекцию

https://www.logicbig.com/tutorials/spring-framework/spring-core/injecting-collections.html

# BeanPostProcessor и BeanFactoryPostProcessor

https://docs.spring.io/spring-framework/docs/3.0.0.M4/reference/html/ch03s08.html

# Часто используемые аннотации спринга

@Controller — класс фронт контроллера в проекте Spring MVC.

@RequestMapping — позволяет задать шаблон маппинга URI в методе обработчике контроллера.

@ResponseBody — позволяет отправлять Object в ответе. Обычно используется для отправки данных формата XML или JSON.

@PathVariable — задает динамический маппинг значений из URI внутри аргументов метода обработчика.

@Autowired — используется для автоматического связывания зависимостей в spring beans.

@Qualifier — используется совместно с @Autowired для уточнения данных связывания, когда возможны коллизии (например одинаковых имен\типов).

@Service — указывает что класс осуществляет сервисные функции.

@Scope — указывает scope у spring bean.

@Configuration, @ComponentScan и @Bean — для java based configurations.

AspectJ аннотации для настройки aspects и advices, @Aspect, @Before, @After, @Around, @Pointcut и др.