Оглавление

IoC

Инверсия управления (IoC) — принцип разработки ПО, по которому управление объектами или частями программы передается в контейнер или среду.

Внедрение зависимости (DI) — шаблон для реализации IoC.

Контейнер IoC — общая характеристика сред, характеризующих IoC. В среде Spring контейнер IoC представлен интерейсом ApplicationContext. Этот контейнер отвечает за создание, настройку и сборку объектов (бинов), а также за управлением их жизненным циклом.

Объекты, создаваемые контейнером, называются ***бинами***, а конфигурирование контейнера осуществляется аннотациями или XML-файлом.

Внедрение зависимостей может быть реализовано через конструкторы, сеттеры или поля.

Бин

Бин (bean) — объекты, управляемые IoC-контейнером. Живут внутри DI-контейнера.

По умолчанию — синглтон.

Могут быть созданы при помощи аннотации `@Bean` или внутри XML-файла (`<bean></bean>`).

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

`@Bean`	`@Component`
	`@Service`
	`@Repository`

явное объявление бин-компонента с кастомной логикой	автоматические обнаружение и настройка бин-компонента
над методом	над классом

```
1.  @Bean
2.  Integer theNumber(){
3.      return new Integer(3456);
4.  }
```

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

@Component, @Service и @Repository

Обозначают класс как бин.

`@Component`	универсальный компонент

`@Service`	содержит бизнес логику и вызывает методы хранилища
`@Repository`	роль хранилища

`@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:

```
1.  PersistenceExceptionTranslationPostProcessor
```

В `@Repository` будет полный стектрайс со всеми методами откуда летит эксепшн (в `@Component` и `@Service` — один эксепшн).

```
1.  @Target({ElementType.TYPE})
2.  @Retention(RetentionPolicy.RUNTIME)
3.  @Documented
4.  @Indexed
5.  public @interface Component {
6.      String value() default "";
7.  }
```

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

```
1.  @Target({ElementType.TYPE})
2.  @Retention(RetentionPolicy.RUNTIME)
3.  @Documented
4.  @Component
5.  public @interface Repository {
6.      @AliasFor(
7.          annotation = Component.class
8.      )
```

```
    String value() default "";
}
```

@Autowired, @Recource, @Inject

Аннотации для внедрения зависимостей. Ставятся над объектом при создании.

`@Resource` (java) пытается получить зависимость: по имени, по типу, затем по описанию (имя из сеттера иля поля, либо параметра name).

```
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.Ob-
ject.class;

    enum AuthenticationType {
            CONTAINER,
            APPLICATION
    }

    AuthenticationType authenticationType() de-
fault AuthenticationType.CONTAINER;

    boolean shareable() default true;

    String mappedName() default "";

    String description() default "";
```

```
31. }
```

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

```
 1. package javax.inject;
 2.
 3. import java.lang.annotation.Documented;
 4. import java.lang.annotation.ElementType;
 5. import java.lang.annotation.Retention;
 6. import java.lang.annotation.RetentionPolicy;
 7. import java.lang.annotation.Target;
 8.
 9. @Target({
10.         ElementType.METHOD,
11.         ElementType.CONSTRUCTOR,
12.         ElementType.FIELD})
13. @Retention(RetentionPolicy.RUNTIME)
14. @Documented
15. public @interface Inject {
16. }
```

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

```
20. }
```

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

@Lookup

Прототип вместо синглтона (несколько объектов). Над методом.

```
1. @Lookup
2. public Passenger createPassenger() {
3.     return null;
4. };
```

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

Были какие-то варианты с сетером.

@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. @Target({ElementType.TYPE, ElementType.METHOD})
2. @Retention(RetentionPolicy.RUNTIME)
3. @Documented
4. public @interface Primary {
5. }
```

```
1. @Target({
2.         ElementType.FIELD,
3.         ElementType.METHOD,
4.         ElementType.PARAMETER,
5.         ElementType.TYPE,
6.         ElementType.ANNOTATION_TYPE})
7. @Retention(RetentionPolicy.RUNTIME)
8. @Inherited
```

```
@Documented
public @interface Qualifier {
    String value() default "";
}
```

Как заинжектить примитив

`@Value`.

Как заинжектить коллекцию

Либо `@Qualifier`, либо в поле возвращаем класс реализации коллекции.

Список

```
@Configuration
public class AnyObjectsConfig {
    @Bean
    public ArrayList<AnyObjects> action() {
        ArrayList<AnyObjects> result =
                new ArrayList<>();
        result.add(new Object1());
        result.add(new Object2());
        return result;
    }
}
```

```
@Service
@Getter
public class ActionHeroesService {
    @Autowired
    @Qualifier("action")
    List<Hero> actionHeroes;
}
```

Карта

```
@Configuration
public class HeroesConfig {

    @Bean
    public HashMap<String, Hero> mainCharacters-
ByFilmName() {
        HashMap<String, Hero> result = new
HashMap<>();
```

```
 9.
10.         result.put("rambo", new Rambo());
11.         result.put("terminator", new Termina-
12. tor());
13.         result.put("LOTR", new Gandalf());
14.
15.         return result;
16.     }
17. }
```

```
 1. @Service
 2. @Getter
 3. public class MainCharactersService {
 4.     @Autowired
 5.     @Qualifier("mainCharactersByFilmName")
 6.     Map<String, Hero> mainCharactersByFilmName;
 7. }
```

@Conditional

`@Conditional` накладывает определенные условия (при выполнения условия — бин создаться).

Готовые решения (условие выполняется, если ...):

· ConditionalOnBean — присутствует нужный бин в BeanFactory;

· ConditionalOnClass — нужный класс есть в classpath;

· ConditionalOnCloudPlatform — активна определенная платформа;

· ConditionalOnExpression — SpEL выражение вернуло положительное значение;

· ConditionalOnJava — приложение запущено с определенной версией JVM;

· ConditionalOnJndi — через JNDI доступен определенный ресурс;

· ConditionalOnMissingBean — нужный бин отсутствует в BeanFactory;

· ConditionalOnMissingClass — нужный класс отсутствует в classpath;

· ConditionalOnNotWebApplication — контекст приложения не является веб контекстом;

· ConditionalOnProperty — в файле настроек заданы нужные параметры;

· ConditionalOnResource — присутствует нужный ресурс в classpath;

· ConditionalOnSingleCandidate — bean-компонент указанного класса уже содержится в BeanFactory и он единственный;

· ConditionalOnWebApplication — контекст приложения является веб контекстом.

@Profile

Частный случай @Conditional. Позволяет отображать bean-компоненты на разные профили (dev (разработки), test (тестирование), prod (продакшн — готовая программа)). В ином случае он просто не активен.

```
1.  package org.springframework.context.annotation;
2.
3.  import java.lang.annotation.Documented;
4.  import java.lang.annotation.ElementType;
```

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional({ProfileCondition.class})
public @interface Profile {
    String[] value();
}
```

BenaFactory

BeanFactory — контейнер, который создает, настраивает и управляет бинами.

Зависимости между объектами отражены в данных конфигурации, используемых BeanFactory.

ApplicationContext

`ApplicationContext` — интерфейс Spring с информацией о конфигурации приложения.

BenaFactory + дополнительная функциональность.

```
public interface ApplicationContext
        extends EnvironmentCapable,
            ListableBeanFactory,
            HierarchicalBeanFactory,
            MessageSource,
            ApplicationEventPublisher,
            ResourcePatternResolver {
    @Nullable
    String getId();

    String getApplicationName();

    String getDisplayName();

```

```
15.     long getStartupDate();
16.
17.     @Nullable
18.     ApplicationContext getParent();
19.
20.     AutowireCapableBeanFactory
21.                 getAutowireCapableBeanFactory()
22.             throws IllegalStateException;
23. }
```

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.

BeanFactory и ApplicationContext

BeanFactory:

· при ограниченных ресурсах;

· только синглтон и прототип;

· ленивый контейнер — создает объекты только при вызовы `.getBean()` (`ApplicationContext` — нетерпеливый контейнер — при загрузке файла spring.xml, если не прототип).

Рекомендуется использовать ApplicationContext — нетерпеливый контейнер

сокращает время ответа на запрос пользователя.

Жизненный цикл бина в Spring

Жизненный цикл — время существования класса. Бины инициализируются при инициализации контейнера и происходит внедрение зависимостей. При уничтожении контейнера — уничтожается все его содержимое.

Для задании действия при инициализации или уничтожении бина — `init()` и `destroy()` (`@PostConstruct` и `@PreDestroy()`).

Жизненный цикл бинов:

· Загрузка описаний бинов и создание графа зависимостей (между бинами);

· Создание и запуск BeanFactoryPostProcessors;

· Создание бинов;

· Spring внедряет значения и зависимости в свойства бина;

· Если бин реализует методом интерфейса `NameBeanAware` (`setBeanName()`) — ID бина передается в метод;

· Если бин реализует интерфейс `BeanFactoryAware` или `ApplicationContextAware` — Spring устанавливает ссылку на BeanFactory или

ApplicationContext (setBeanFactory() или setApplicationContext()).

· Реализуя метод интерфейса BeanPostProcessor (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.

```
 1. package org.springframework.beans.factory.config;
 2.
 3. import org.springframework.beans.BeansException;
 4. import org.springframework.lang.Nullable;
 5.
 6. public interface BeanPostProcessor {
 7.     @Nullable
 8.     default Object
 9.             postProcessBeforeInitialization(
10.                 Object bean, String beanName)
11.             throws BeansException {
12.         return bean;
13.     }
14.
15.     @Nullable
16.     default Object
17.             postProcessAfterInitialization(
18.                 Object bean, String beanName)
19.             throws BeansException {
20.         return bean;
21.     }
22. }
```

Bean scopes

В Spring предусмотрены различные области времени действия бинов:

· singleton — один экземпляр, по умолчанию, не потокобезопасный;

· prototype — новый экземпляр при каждом запросе;

· request — новый экземпляр при каждом HTTP request;

· session — новый экземпляр при каждой HTTP сессии;

· global-session — глобальных бинов на уровне сессии (для Portlet приложений).

В Spring 5:

· singleton;
· prototype;
· request;
· session;
· application — один экземпляр на жизненный цикл ServletContext;
· websocket — один экземпляр на жизненный цикл WebSocket.

@ComponentScan

Есть классы с аннотациями `@Component`, `@Service` или `@Repository`. В отличие от аннотации @Bean .... `@ComponentScan` указывает пакеты и подпакеты.

Аннотация `@SpringBootApplication` включает в себя три аннотации:

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
```

```
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;
```

```
39.
40.     String resourcePattern() default "**/*.class";
41.
42.     boolean useDefaultFilters() default true;
43.
44.     ComponentScan.Filter[] includeFilters()
45.             default {};
46.
47.     ComponentScan.Filter[] excludeFilters()
48.             default {};
49.
50.     boolean lazyInit() default false;
51.
52.     @Retention(RetentionPolicy.RUNTIME)
53.     @Target({})
54.     public @interface Filter {
55.         FilterType type()
56.                 default FilterType.ANNOTATION;
57.
58.         @AliasFor("classes")
59.         Class<?>[] value() default {};
60.
61.         @AliasFor("value")
62.         Class<?>[] classes() default {};
63.
64.         String[] pattern() default {};
65.     }
66. }
```

```
 1. package org.springframework.boot.autoconfigure;
 2.
 3. import java.lang.annotation.Documented;
 4. import java.lang.annotation.ElementType;
 5. import java.lang.annotation.Inherited;
 6. import java.lang.annotation.Retention;
 7. import java.lang.annotation.RetentionPolicy;
 8. import java.lang.annotation.Target;
 9. import org.springframework.beans.factory.support
10.         .BeanNameGenerator;
11. import org.springframework.boot
12.         .SpringBootConfiguration;
13. import org.springframework.boot.context
14.         .TypeExcludeFilter;
15. import org.springframework.context.annotation
16.         .ComponentScan;
```

```
17. import org.springframework.context.annotation
18.         .Configuration;
19. import org.springframework.context.annotation
20.         .FilterType;
21. import org.springframework.context.annotation
22.         .ComponentScan.Filter;
23. import org.springframework.core.annotation
24.         .AliasFor;
25.
26. @Target({ElementType.TYPE})
27. @Retention(RetentionPolicy.RUNTIME)
28. @Documented
29. @Inherited
30. @SpringBootConfiguration
31. @EnableAutoConfiguration
32. @ComponentScan(
33.     excludeFilters = {@Filter(
34.     type = FilterType.CUSTOM,
35.     classes = {TypeExcludeFilter.class}
36. ), @Filter(
37.     type = FilterType.CUSTOM,
38.     classes =
39.             {AutoConfigurationExcludeFilter.class}
40. )}
41. )
42. public @interface SpringBootApplication {
43.     @AliasFor(
44.         annotation = EnableAutoConfiguration.class
45.     )
46.     Class<?>[] exclude() default {};
47.
48.     @AliasFor(
49.         annotation = EnableAutoConfiguration.class
50.     )
51.     String[] excludeName() default {};
52.
53.     @AliasFor(
54.         annotation = ComponentScan.class,
55.         attribute = "basePackages"
56.     )
57.     String[] scanBasePackages() default {};
58.
59.     @AliasFor(
60.         annotation = ComponentScan.class,
```

```
61.             attribute = "basePackageClasses"
62.         )
63.         Class<?>[] scanBasePackageClasses()
64.                 default {};
65.
66.         @AliasFor(
67.             annotation = ComponentScan.class,
68.             attribute = "nameGenerator"
69.         )
70.         Class<? extends BeanNameGenerator>
71.                     nameGenerator()
72.                 default BeanNameGenerator.class;
73.
74.         @AliasFor(
75.             annotation = Configuration.class
76.         )
77.         boolean proxyBeanMethods() default true;
78.     }
```

Транзакции в Spring. @Transactional

Типы управления транзакциями:

· при помощи программирования (гибкий, но его сложно поддерживать);

· при помощи `@Transactional` (или конфигурацией на основе XML для управления транзакциями).

Аннотация `@Transactional` 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.

```
<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:

```
@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. @Transactional (noRollbackFor =
2.         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.

@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.  @Target({ElementType.TYPE})
2.  @Retention(RetentionPolicy.RUNTIME)
3.  @Documented
4.  @Component
5.  public @interface Controller {
6.      @AliasFor(
7.          annotation = Component.class
8.      )
```

```
    String value() default "";
}
```

```
@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).

```
@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.

```
@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:

```
@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) {
        // ...
    }
}
```

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, 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.

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.

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

AOP

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.

!!! В чем разница между Filters, Listeners и Interceptors?

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

Да. Использовать массив.

Spring Security

Spring Security предоставляет возможности для защиты приложения: аутентификация, авторизация, роли и маппинга доступных страниц (ссылок и т. п.), а также защиту от различных вариантов атак.

Основные сущности (содержат):

· `class SecurityContextHolder` — информация о текущем контексте безопасности приложения (с подробной информации о пользователе (`Principal`), работающем в настоящее время с приложением). Стратегии хранения информации: `ThreadLocal` —по умолчанию;

· `interface SecurityContext` — объект `Authentication` и информация, связанная с запросом от пользователя;

```
1.  package org.springframework.security.core.context;
2.
3.  import java.io.Serializable;
4.  import org.springframework.security.core
5.          .Authentication;
6.
7.  public interface SecurityContext
8.          extends Serializable {
9.      Authentication getAuthentication();
```

```
10.
11.     void setAuthentication(Authentication var1);
12. }
```

· `interface Authentication` — пользователя (`Principal`);

```
 1. package org.springframework.security.core;
 2.
 3. import java.io.Serializable;
 4. import java.security.Principal;
 5. import java.util.Collection;
 6.
 7. public interface Authentication
 8.         extends Principal, Serializable {
 9.     Collection<? extends GrantedAuthority>
10.             getAuthorities();
11.
12.     Object getCredentials();
13.
14.     Object getDetails();
15.
16.     Object getPrincipal();
17.
18.     boolean isAuthenticated();
19.
20.     void setAuthenticated(boolean var1)
21.             throws IllegalArgumentException;
22. }
```

· `interface GrantedAuthority` — роли (разрешения);

```
 1. package org.springframework.security.core;
 2.
 3. import java.io.Serializable;
 4.
 5. public interface GrantedAuthority
 6.         extends Serializable {
 7.     String getAuthority();
 8. }
```

· `interface UserDetails` — информацию о пользователе, хранимая в таблице (имя, пароль, ...);

```
 1. package org.springframework.security.core
 2.         .userdetails;
 3.
 4. import java.io.Serializable;
 5. import java.util.Collection;
 6. import org.springframework.security.core
 7.         .GrantedAuthority;
 8.
 9. public interface UserDetails
10.         extends Serializable {
11.     Collection<? extends GrantedAuthority>
12.             getAuthorities();
13.
14.     String getPassword();
15.
16.     String getUsername();
17.
18.     boolean isAccountNonExpired();
19.
20.     boolean isAccountNonLocked();
21.
22.     boolean isCredentialsNonExpired();
23.
24.     boolean isEnabled();
25. }
```

· `interface UserDetailsService` — содержит единственный метод для создания `UserDetails` объекта.

```
1. package org.springframework.security.core
2.         .userdetails;
3.
4. public interface UserDetailsService {
5.     UserDetails loadUserByUsername(String var1)
6.             throws UsernameNotFoundException;
7. }
```

Spring Boot

Spring Boot — набор классов конфигурации, которые автоматически создают нужные бины в контексте.

Преимущества:

· автоконфигурация;

· разрешение проблем конфликтов разных версий компонентов (starter-пакеты);

· встроенный Tomcat/Jetty.

`@ConditionalOn*`.

Можно отключить ненужные автоконфигурации (`@EnableAutoConfiguration`, `@ImportAutoConfiguration` и `@SpringBootApplication`) или совсем отказаться от этого механизма (убрать `@SpringBootApplication` и `@EnableAutoConfiguration`, а для указания нужных конфигурационных классов — `@SpringBootConfiguration` и `@ImportAutoConfiguration`).

Нововведение Spring 5

· поддержка JDK 8+;

· совместимость с Java EE 8;

· поддержка Kotlin;

· реактивность (Web on Reactive Stack);

· null-safety аннотации (@Nullable);

- новая документация;
- поддержка Junit 5;

@ResponseBody и @ResponseEntity

`@ResponseEntity` необходим, только если мы хотим кастомизировать ответ, добавив к нему статус ответа (1xx — информационные, 2xx — успешные, 3xx — перенаправления, 4xx — ошибки клиента, 5xx — 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.

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. public interface CrudRepository<
2.             T,
3.             ID extends Serializable>
4.         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 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 коллаборатором 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 коллаборатоpы 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.

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.

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.

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”

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 сгенерированные 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.

@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.

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.

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.

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.