**1․ IoC կոնտեյներ**

**Ինչ է Spring Framework – ը**

Framework – ը իրենից ներկայացնում է կարկաս, պլատֆոր որը սահմանում է app – ի կառուցվածքը և հեշտացնում է app – ի կոմպոնենտների մշակումը և դրանց ինտեգրումը։ Framework - ը ավելի լայն հասկացություն է քան սովորական Java գրադարանը, գրադարանը դա կոդի կտոր է որը իմպորտ է արվում և օգտագործվում իսկ framework – ը սահմանում է կառուցվածքը, տրամադրում պատտերներ և այլն։ Ներդրվել է 2003 – ին, ընթացիկ վերսիան 6 – ն է (11.2022)։

Spring Framework – ը կոմպոնենտների հավաքածու է, որը հեշտացնում է Java – ով աշխատանքը օր՝ application context, dependency injection, db – ի հետ աշխատանքի հեշտացված մեխանիզմներ, spring mvc (spring framework – ի մաս) կոմպոնենտ web app – ներ ստեղծելու համար։ spring mvc – ն Java EE սերվլետի համամատ ավելի արդիական է և հարմար օգտագործման համար։

Spring framework – ի այլ կոմպոնենտներ են՝

Security – անվտանգության համար

Boot – հեշտացնում է spring app – ների ստեղծումը, ձերբազատում է հավելյալ config – ներից

Հասարակ app ստեղծելու համար պետք են հետևյալ dependency – ները՝

spring-core

spring-beans

spring-context

**Inversion of Control**

Java կլասսների մեջ կարող են լինել կախվածություններ այլ կլասսներից որոնք բարդացնում են աշխատանքը։ Ավելի ճկուն լինելու համար օգտագործվում են ինտերֆեյսները և աբստրակտ կլասսները։ Սրանից բացի խնդրահարույց է նաև օբյեկտների ստեղծումը, կարիք է առաջանում անընդհատ ստեղծել նոր օբյեկտ, կլասսը ինքն է ստեղծում իր կախվածությունների էկզեմպլյարները։ Սրա լուծման համար օգտագործվում են bean – երի հայտարարումը։

Java bean – ը սովորական օբյեկտ է, bean կոչվում է երբ այն ստեղծվում է spring – ի միջոցով։

IoC – ի սկզբունքի համաձայն կլասսի կախվածությունները պետք է ներդրվեն արտաքինից այլ ոչ թե հայտարարվեն կլասսի մեջ։ IoC – ը դա ծրագրային միջավայրում կառուցվածքային լուծում է որի ժամանակ կլասսը ոչ թե ինքն է ստեղծում իր կախվածությունները այլ դրանք ներդվում են (inject) դրսից։

Ioc կամ (dependency injection) մեխանիզմ է որի ժամանակ օբյեկտները սահմանում են իրենց կախվածությունները այլ օբյեկտներից միայն կոնստրուկտորի, կամ factory մեթոդի արգումենտների միջոցով։ Factory մեթոդը պատտերն է որի միջոցով օբյեկտը ստեղծվում է անուղակի ձևով մեթոդի կանչի միջոցով։ Դրանից հետո կոնտեյները bean - երի ստեղծման ժամանակ ներդնում է այդ կախվածությունները։ Ioc – ի հիմքը org.springframework.beans և org.springframework.context package – ներն են։ Spring – ում օբյեկտները որոնք կազմում են app – ի հիմքը և կառավարվում են Spring IoC կոնտեյների կողմից կոչվում են bean կոմպոնենտներ։

**Կոնտեյների ընդհանուր նկարագիրը**

Կոնտեյները տրամադրում է ApplicationContext – ը։ Կոնտեյները ստանում է instruction – ներ XML ֆայլի, անոտացիաների կամ Java կոդի միջոցով և պատասխանատու է էկզեմպլյարների ստեղծման, դրանց միջև կապերի սահմանման և այլ գործառույթների համար։



Կոնֆիգուրացիոն մետատվյալները սահմանում են թե ինչպես է կոնտեյները ստեղծելու էկզեմպլյարները, կարգավորելու և հավաքելու app – ի օբյեկտները։

Կոնտեյները ինքնին կախված չէ թե ինչ տարբերակով են հայտարարվում config – ները։ XML – ը ավելի հին տարբերակ է։ Xml – ի դեպքում կոմպոնենտների կարգավորումը կատարվում է <bean><bean/> էլեմենտի միջոցով։

<bean id="..." class="...">

<!-- collaborators and configuration for this bean go here -->

</bean>

Id – ի միջոցով նույնականացվում է տվյալ bean – ն

Class – ը ցույց է տալիս bean – ի տիպը, նշվում է կլասսի ամբողջկան անունը ներառյալ գտնվելու package – ը։

**Կոնտեյների ստեղծումը**

ApplicationContext – ի կոնստրուկտորի միջոցով նշվում այն xml – ը որում նշված config – ները պետք է կոնտեյները ընդունի որպես instruction – ներ։

ApplicationContext context = new ClassPathXmlApplicationContext("services.xml", "daos.xml");

Context – ից bean – ը ստանալու համար ApplicationContext - ում գոյություն ունի T getBean(String beanName, Class<T> classType) – մեթոդ

context.getBean("myBeanId", MyBeanClass.class);

<bean/> էլեմենտի միջոցով bean – ներ ստեղծելիս դրանք կոնտեյնարի մեջ ունեն BeanDefinition տիպի օբյեկտների տեսքերով որոնք ունեն մի շարք մետատվյալներ՝ կլասսի անուն package – ի նշումով, կոնֆիգուրացիոն էլէմենտներ թե ինչպես պետք է bean – ը իրեն պահի կոնտեյների մեջ օր՝ տեսանելիության սկոպ, հղումներ դեպի այլ bean – երի և այլն։ Այս մետատվյալները արտահայտվում են հետևայալ property – ներում՝

|  |  |
| --- | --- |
| Class | [Instantiating Beans](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-class) |
| Name | [Naming Beans](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-beanname) |
| Scope | [Bean Scopes](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-scopes) |
| Constructor arguments | [Dependency Injection](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-collaborators) |
| Properties | [Dependency Injection](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-collaborators) |
| Autowiring mode | [Autowiring Collaborators](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-autowire) |
| Lazy initialization mode | [Lazy-initialized Beans](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-lazy-init) |
| Initialization method | [Initialization Callbacks](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-lifecycle-initializingbean) |
| Destruction method | [Destruction Callbacks](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-lifecycle-disposablebean) |

**Bean naming**

Bean – ունի id որը կոնտեյների շրջանակում պետք է լինի ունիկալ։ id – ից բացի կարող են լինել նաև այլ կեղծանուններ։ Դրանք սահմանվում են համապատասխանաբար id և name արտիբուտների միջոցով։ id – ի ունիկալությունը ապահովում/երաշխավորում է կոնտեյները։ Եթե id – ն ակնհայտ չի նշվում ապա այն կգեներացվի կոնտեյների կողմից (կլասսի անունով)։ Անվանավորման համար գործում են նույն convention – ները։

Bean – ի կեղծանուններ

Որպես bean – ի անուն կարող ենք օգտագործել մեկ id և անսահմանափակ name

**Instantiating Beans**

Xml – ով աշխատելիս նշվում է bean – ի class -ը

**Bean – ի ստեղծում ստատիկ factory մեթոդով**

Այս մեթոդներում կոնստրուկտորը նշվում է private: factory-method ատրիբուտով Spring - ին նշվում է որ տվյալ bean – ի օբյեկտները պետք է ստողծվեն ոչ թե new – ով (private կոնստրուկտորի պատճառով) այլ factoryմեթոդի միջոցով։ Անկախ այն բանից որ factory մեթոդը իր մեջ ամեն անգամ new -ով վերադարձնում է նոր օբյեկտ այնուամենայնիվ singleton սկոպի դեպքում կունենանք bean – ի միայն մեկ էկզեմպլյար քանի անգամ էլ որ bean – ը get արվի, քանի որ spring – ը միայն մեկ անգամ է կանչում factory մեթոդը։

Կարող է նաև լինել հետևյալ կերպ՝

<bean id="clientService"

class="examples.ClientService"

factory-method="createInstance"/>

որտեղ createInstance – ը ClientService – ի ստատիկ մեթոդ է որը վերադարձնում է ClientService տիպի օբյեկտ։

**Bean – ի ստեղծում instance factory մեթոդով**

<bean id="serviceLocator" class="examples.DefaultServiceLocator">

<!-- inject any dependencies required by this locator bean -->

</bean>

<!-- the bean to be created via the factory bean -->

<bean id="clientService"

factory-bean="serviceLocator"

factory-method="createClientServiceInstance"/>

որտեղ createClientServiceInstance – ը DefaultServiceLocator – ի instance մեթոդ է որը վերադարձնում է ClientService տիպի օբյեկտ։

**Dependencies**

App – ում տարբեր օբյեկտները որոնք աշխատում են իրար հետ գտնվում են միմյանցից կախվածության մեջ։

DI – ն մեխանիզմ է որի ժամանակ օբյեկտները սահմանում են իրենց կախվածությունները կոնստրուկտորի, factory մեթոդի արգումենտների միջոցով։ Կոնտեյները ներդնում է կախվածությունները դրանց bean – ները ստեղծելուց հետո։ Սրա շնորհիվ app – ն դառնում է ավելի ճկուն։

DI – ի հիմնական կիրառման տարբերակներն են կոնստրուկտորի և setter – ի միջոցով։

Կոնստրուկտորի միջոցով՝

public ExampleBean(DependencyBean bean) {

this. bean = bean;

}

Xml – ում bean – ի կոնստրուկտորին արգումենտներ փոխանցելու համար՝

<beans>

<bean id="beanOne" class="x.y.ThingOne">

<constructor-arg ref="beanTwo"/>

<constructor-arg ref="beanThree"/>

</bean>

<bean id="beanTwo" class="x.y.ThingTwo"/>

<bean id="beanThree" class="x.y.ThingThree"/>

</beans>

Պրիմիտիվների դեպքում՝

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg type="int" value="7500000"/>

<constructor-arg type="java.lang.String" value="42"/>

</bean>

Type – ի փոխարեն կարող է լինի index որով նշվում է կոնստրուկտորի արգումենտի ինդեքսը, կամ պարրամետրի անունը։ Օգտագործվում է նույն տիպերի դեպքում սխալներից խուսափելու համար։ Անունները օգտագործելու դեպքում պետք է  [@ConstructorProperties](https://download.oracle.com/javase/8/docs/api/java/beans/ConstructorProperties.html) – ով նշվի պարրամետրերի անունները։

@ConstructorProperties({"years", "ultimateAnswer"})

public ExampleBean(int years, String ultimateAnswer) {

this.years = years;

this.ultimateAnswer = ultimateAnswer;

}

Setter – ի միջոցով՝

public void setBean(DependencyBean bean) {

this. bean = bean;

}

Այն կախվածությունները որոնք պարտադիր պետք են ցանկալի է նշել կոնստրուկտորում իսկ կախվածությունները որոնք սկզբնական աշխատանքի համար պարտադիր չեն կարող են նշվել setter – ում։

Կոնստրուկտորի միջոցով DI – ն կազմակերպելու դեպքում հնարավոր են Ցիկլիկ կախվածություններ։ Այս դեպքում դեպքում կոնտեյները կտա BeanCurrentlyInCreationException։ Որպես լուծում կարելի է օգտագործել DI setter – ների այլ ոչ թե կոնստրուկտորի միջոցով։ Ցիկլիկ կախվածության դեպքում փաստացի մենք փորձում ենք bean – երից մեկը մինչև դրա ամբողջական ինիցիալիզացիան ներդնել մեկ այլ bean – ի մեջ։

Beanc – ի property կարելի է ավելացնել նաև հետևյալ տարբերակով՝

public class SomeClass {

private Map<String, Float> accounts;

public void setAccounts(Map<String, Float> accounts) {

this.accounts = accounts;

}

}

<beans>

<bean id="something" class="x.y.SomeClass">

<property name="accounts">

<map>

<entry key="one" value="9.99"/>

<entry key="two" value="2.75"/>

<entry key="six" value="3.99"/>

</map>

</property>

</bean>

</beans>

Կամ

<bean class="ExampleBean">

<property name="email" value=""/>

</bean>

Նույնն է ինչ որ exampleBean.setEmail("");

Bean- երը կարող են նաև ունենալ lazy-init="" ատրիբուտը որը ընդունեում է բուլյան արժեքներ և օգտագործվում է եթե կոմպոնենտը պետք է ստեղծվի դրան առաջին անգամ դիմելու այլ ոչ թե start – ի ժամանակ։

<bean id="lazy" class="com.something.ExpensiveToCreateBean" lazy-init="true"/>

<bean name="not.lazy" class="com.something.AnotherBean"/>

**Bean Scopes**

կ scope – ը սահմանում է թե ինչպես են ստեղծվելու bean – երը։ կարող են ստեղծվել bean - երի ինչպես մեկ այնպես էլ մեկից ավելի էկզեմպլյարներ։ Կնոֆիգուրացիայի միջոցով կարող են սահմանվել bena – ի տեսանելիության տիրույթը (scope): Spring framework – ում կան 6 scope – եր որոնցից 4 – ը հասանելի են միայն web համադրելի ApplicationContext – ների համար։

singleton – default scope որը սահմանում է կոմպոնենտի մեկ էկզեմպլյարի նկարագրություն յուրաքանչյուր կոնտեյների համար։ Այսինքն ամեն inject – ի ժամանակ ներդրվում է օբյեկտի նույն, միակ էկզեմպլյարը։ Յուրաքանչյուր անգամ երբ bean – ը կանչվում է իր իդենտիֆիկատորով, կոնտեյները վերադարձնում է օբյեկտի էզակի էկզեմպլյարը։ Էկզեմպլյարը կոնտեյները պահում է քեշում և ամեն անգամ օբյեկտը վերադարձվում է քեշից։ Չնայած որ singleton – ը default scope – է այն կարելի է նաև ակնհայտ նշել scope – ով։ singletone – ի դեպքում bean – ը ստեղծվում է մինչև getBean() – ի կանչելը։ Օգտագործվում է stateless օբյեկտների դեպքում։

<bean id="accountService" class="com.something.DefaultAccountService" scope="singleton"/>

prototype – նույն կոմպոնենտի ցանկացած քանակությամբ էկզեմպլյարների համար։singleton – ի հակառակ պատկերը, ամեն անամ երբ հղում է կատարվում կոմպոնենտին/bean – ին ստեղծվում է դրա նոր էկզեմպլյար։ Օգտագործվում է stateful օբյեկտների դեպքում։

<bean id="accountService" class="com.something.DefaultAccountService" scope="prototype"/>

request – կոմպոնոնտի մեկ սահմանում HTTP request – ի կյանքի ցիկլի համար։ Այսինքն յուրաքանչյուր request ունի իր սեփական էկզեմպլյարը ստեղծված նույն կոմպոնենտի հիման վրա, աշխատում է միայն web համադրելի applicationContext – ների համար։ scope="request" կամ @RequestScope

session – նույնը session – ի կյանքի ցիկլի համար համար: scope=" session " կամ @SessionScope

application - նույնը ServletContext – ի կյանքի ցիկլի համար համար: scope=" session " կամ @ApplicationScope

websocket - նույնը websocket – ի կյանքի ցիկլի համար համար

**Customizing the Nature of a Bean (կյանքի ցիկլի կառավարում)**

**Bean - ի կյանքի ցիկլը հետևյալն է՝**

* Spring api – ի start
* Կոնտեյների start
* Bean – ի օբյեկտի ստեղծում
* Benad – ի ներդրում dependency injection
* Նշված int մեթոդի կանչ
* Bean – ի պատրաստ է օգտագործման
* Նշված destroy մեթոդի կանչ

Կարգավորումների համար կարող են օգտագործվել հետևյալ մեխանիզմնեը՝

* [Lifecycle Callbacks](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-lifecycle)
* [ApplicationContextAware and BeanNameAware](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-aware)
* [Other Aware Interfaces](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#aware-list)

Lifecycle callbacks – ինտերֆեյսները իմպլեմենտացնելով ստանում ենք մեթոդներ bean – ի init – ի ուն destroy – ի ժամանակ գործողություններ կատարելու համար։ Ավելի լավ պրակտիկա է համարվում [@PostConstruct և @PreDestroy](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-postconstruct-and-predestroy-annotations) անոտացիաների օգտագործումը։

org.springframework.beans.factory.InitializingBean ինտերֆեյսը տրամադրում է միակ՝ void afterPropertiesSet() throws Exception; մեթոդը, սրա փոխարեն խորհուրդ է տրվում օգտագործել @PostConstruct անոտացիան, իսկ xml – ի դեպքում init-method="init" ատրիբուտը, որի դեպքում ինտերֆեյսը իմպլեմենտացնել պետք չէ։

org.springframework.beans.factory.DisposableBean ինտերֆեյսը տրամադրում է միակ՝ void destroy() throws Exception; մեթոդը, սրա փոխարեն խորհուրդ է տրվում օգտագործել @PreDestroy անոտացիան, իսկ xml – ի դեպքում destroy-method="cleanup" ատրիբուտը, որի դեպքում ինտերֆեյսը իմպլեմենտացնել պետք չէ։

Եթե օգտագործվում են ոչ թե ինտերֆեյսների մեթոդները այլ custom մեթոդներ ապա ցանկալի է որ դրանք ունենան conventional անուններ՝ init(), initialize(), dispose() և այլն:

Custom մեթոդները կարող են ունենալ ցանկացած հասանելիության մոդիֆիկատորներ։ Վերադարձվող տիպը կարող է լինել ցանկացած բայց օգտագործվում է void քանի որ չենք կարող ստանալ վերադարձրած արժեքը։ Մեթոդները արգումենտներ չպետք է ստանան։ Նաև այլ նրբություն prototype scope – ով bean – երի համար չի կանչվում destroy մեթոդը։ Prototype – ի դեպքում Spring – ը միայն ստեղծում է bean – ը։

Ամենավերևի <beans> էլեմենտում կարելի է օգտագործել <beans default-init-method="init"> ատրիբուտը և կոնտեյները նշված մեթոդը կփնտրի բոլոր bean -երում: Նույնը վերաբերում է default-destroy-method – ին:

Այսպիսով bean – կյանքի ցիկլի իրադարձություններին հտևելու համար համար կան հետևյալ մեխանիզմները՝

* [InitializingBean](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-lifecycle-initializingbean) և [DisposableBean](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-lifecycle-disposablebean) ինտերֆեյսներ
* Custom init() և destroy() մեթոդներ
* @PostConstruc @PreDestroy անոտացիաներ:

Եթե օգտագործվեն միանգամից մեկից ավելի մեխանիզմենրը մեթոդների տարբեր անուններով ավա դրանք կաշխատեն բոլորը ըստ հերթականության, իսկ եթե բոլոր մեթոդները ունենան նույն անունները կաշխատեն միայն մեկ անգամ:

Կյանքի ցիկլի հետ աշխատելու համար կա նաև Lifecycle ինտերֆեյսը:

**Կոնտեյների աշխատանքի դադարեցում ոչ web application – ների համար:**

Web app – ներում ApplicationContext – ունի ռեալիզացիա կոնտեյների աշխատանքի դադարեցման համար: որ web app -ներում պետք է օգտագործել

ConfigurableApplicationContext – ի registerShutdownHook() մեթոդը:

ConfigurableApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");

ctx.registerShutdownHook();

**ApplicationContextAware և BeanNameAware**

Նշված ինտերֆեյսները տրմադրում են համապատասխանաբար՝

void setApplicationContext(ApplicationContext applicationContext) throws BeansException;

void setBeanName(String name) throws BeansException;

մեթոդները, որոնք օգտագործվում են bean – ին ApplicationContext տալու և անունը փոխելու համար:

Սրանցից բացի կան նաև այլ Aware ինտերֆեյսներ որոնք ունենիրենց անվանը համապատասխան ֆունկցիոնալություն bean – ի հետ կատարվող տարբեր իրադարձությունների հետևելու համար:

Նշված ինտերֆեյսների օգտագործումը հակասում է IOC սկզբունքին և կարծր կապեր է ստեղծում Spring API – ի հետ: Խորհուրդ է տրվում սրանք օգտագործել այն կոմպոնենտների համար որոնց պետք է հասանելիություն կոնտեյներին:

**Bean – երի ժառանգումը**

Ժառանգման համար կարող է օգտագործվել parent ատրիբուտը՝

<bean id="inheritedTestBean" abstract="true"

class="org.springframework.beans.TestBean">

<property name="name" value="parent"/>

<property name="age" value="1"/>

</bean>

<bean id="inheritsWithDifferentClass"

class="org.springframework.beans.DerivedTestBean"

parent="inheritedTestBean" init-method="initialize">

<property name="name" value="override"/>

<!-- the age property value of 1 will be inherited from parent -->

</bean>

abstract ատրիբուտով true/false կարող է սահմանվել

Ժառանգ bean – ը ժառանգում է նաև տեսանելիության scope – ը:

**Պոստպրոցեսսորներ(լրացուցիչ նայել)**

**Կոնտեյների կոնֆիգուրացիա անոտացիաների միջոցով**

Անոտացիաները կոդը լրացնող մետատվյալներ են։ Անոտացիաների օգտագործումը ավելի լակոնիկ է և ընթեռնելի, իսկ xml լավ է այն տեսանկյունից որ պետք չէ փոփոխել կոմպենենտի Source կոդը: Կոնֆիգուրացիայի անոտատիվ և xml տեսակները կարող են աշխատել միասին:

@Autowired – օգտագործվում է bean - երի ավտոմատ ներդրման DI համար, նշված անոտացիան հանդիպելուց Spring – ը փնտրում է դրան համապատասխանող bean – ը inject – ի համար։ Inject – ի բոլոր տարբերակները ֆունկցիոնալության տեսանկյունից նույնն են, առավելություններ մեկը մյուսի նկատմամբ չունեն։ Ընթեռնելիության տեսանկյունից կարևոր է պրոյեկտի շրջանակներում օգտագործել jinect – ի նույն ոճը։

Կարող է օգտագործվել՝

* կոնստրուկտորի համար, բայց կարիք չկա եթե կոմպոնենտում սահմավում է միայն մեկ կոնստրուկտոր։ Եթե կոնստրուկտորները 1 – ից ավելի են և չկա հիմնականը ապա դրանցից մեկը (միայն մեկը չհաշված required=false ատրիբուտը) պետք է նշվի @Autowired – ով որպեսզի կոնտեյները իմանա թե որը պետք է օգտագործի։ Որի վրա եղավ անոտացիան դրա մեջ նշված կախվածությունները inject կլինեն։
* մեթոդների համար, եթե գրվի setter – ի կամ ցանկացած այլ մեթոդի վրա որի արգումենտում կա կախվածությունը, այն inject կլինի
* property – ների համար, կախվածությունը inject կլինի նույնիսկ private – ների համար, անգամ եթե չկա կոնստրուկտոր, setter: Այդ թվում նաև bean – երի մասիվ ստանալու համար՝

@Autowired

private MovieCatalog[] movieCatalogs;

@Autowired – ը կարող է ունենալ (required=false) պարամետր, որոի դեպքում inject – ը տեղի չի ունենա։

Կախվածության ոչ պարտադիր լինելը կարելի է նշել նաև Optional – ի կամ @Nullable անոտացիայի միջոցով, բայց այս դեպքում @Autowired 1 – ից ավելի կոնստրուկտորների վրա չի թույլատրվում։

Պոլիմորֆիզմի հետևանքով inject արվող կոմպոնենտը կարող է ունենալ 1 – ից ավելի ռեալիզացիա, օր եթե inject է արվում ինտերֆեյս բայց կան այն իմպլեմենտացնող 2 bean – եր։ Որպես լուծում կարելի է նշել bean – ի նախապատվությունը @Primary անոտացիայի միջոցով՝

@Bean

@Primary

public MovieCatalog firstMovieCatalog() { ... }

@Bean

public MovieCatalog secondMovieCatalog() { ... }

xml – ում կա համապատասխան primary="true" ատրիբուտը։

Այս տարբերակը էֆֆեկտիվ է երբ bean – ի հիմնական թեկնածուն որոշված է։

Նույն գործողությունը կարելի է կատարել @Qualifier – ի միջոցով երբ 1 – ից ավելի bean – երից պետք է ընտրել որևէ մեկը։

@Autowired

@Qualifier("main")

private MovieCatalog movieCatalog;

կամ

@Autowired

public void prepare(@Qualifier("main") MovieCatalog movieCatalog,

CustomerPreferenceDao customerPreferenceDao) {

this.movieCatalog = movieCatalog;

this.customerPreferenceDao = customerPreferenceDao;

}

որպես արժեք կարող է նշվել bean – ի id – ն։ xml – ում կարող է օգտագործվել նաև <qualifier value="main"/>։ Ի տարբերություն id – ի qualifier – ները կարող են կրկնվել և օգտագրծվել օրինակ նույն qualifier-ներով bean – երը զանգվածում կամ լիստում autowiring անելու համար։

**@Resource։** Injection կարելի է կատարել նաև jakarta.annotation.Resource @Resource – ի միջոցով։

@Resource(name="myMovieFinder")

public void setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

Ընդունում է name ատրիբուտ որը default – ով հասկանում է որպես կոմպոնենտի անուն։ Եթե name – ը չի նշված այն վերցվում է մեթոդի կամ property – ի անունից որի վրա որ դրված է։

**@Value – ի միջոցով։**

Օգտագործվում է արտաքին property – ներ ներդնելու համար` spring expression language – ի միջոցով։

public MovieRecommender(@Value("${catalog.name}") String catalog) {

this.catalog = catalog;

}

Կոնֆիգուրացիոն ֆայլում @PropertySource("classpath:application.properties") – ի միջոցով նշվում է ռեսուրսի / ֆայլի տեղը և դրանում առկա catalog.name փոփոխականով արժեքը։ Տվյալ դեպքում String catalog – ը կստանա catalog.name – ի արժեքը։ Արգումենտում string – երի մասիվով կարելի է նշել 1 – ից ավելի ֆայլեր, իսկ դրացում նույն անունով փոփոխականների առկայության դեպքում փոփոխականը կստանա մասիվում վերջին ինդեքով նշված ֆայլում վերագրված արժեքը։ classpath – ով նշվում է քանի որ կոմպիլյացիայից հետո ֆայլերը հայտնվում են classes folder – ում։

Նշված ձևով արժեքներ ստանալու պրոցեսի նկատմամբ խիստ հսկողություն սահմանելու համար կարելի է հայտարարել հետևյալ bean – ը

@Bean

public static PropertySourcesPlaceholderConfigurer propertyPlaceholderConfigurer() {

return new PropertySourcesPlaceholderConfigurer();

}

Այս bean – ի հայտարարումը երշխավորում է որ եթե ${ } սինտաքսով նշված արժեքները չգտնվեն ապա կունենանք exception, եթե bean – ը չկա ապա որպես արժեք սկտանանք ${ } գոյություն չունեցող property – ի անունով։

Spring – ը ավտոմատ աշխատեցնում է ConversionService – ը որը կատարում է casting @Value – ով կանչվող տվյալների համար։

@Value("#{{'Thriller': 100, 'Comedy': 300}}") տեսքով կարելի է փոխանցել նաև ավելի բարդ տվյալներ օր․՝ map:

**@PostConstruct և @PreDestroy**

Օգտագործվում են կյանքի համապատասխան ցիկլերին հետևելու համար

@PostConstruct

public void populateMovieCache() {

// populates the movie cache upon initialization...

}

@PreDestroy

public void clearMovieCache() {

// clears the movie cache upon destruction...

}

Որպեսզի PreDestroy – ը աշխատի պետք է context.close();

Նշված անոտացիաները գտնվում են jakarta.anotation-api – ում։

**Classpath Scanning and Managed Components**

Եթե bean -երը xml – ում չեն հայտարարվում այլ ստեղծվում են անոտացիաների միջոցով ապա պետք է սահմանել սքանավորման տիրույթը՝

<context:component-scan base-package="org.example"/>

Կամ կոնֆիգուրացիոն կլասսում @ComponentScan(basePackage="org.example"): Անոտացիան պետք է @Configuration – ի հետ միասին լինի main կլասսի վրա, իսկ bean – երը get կարվեն արդեն AnnotationConfigurationAllicationContext – ից։

Կոմպոնենտ ստեղծվում է @Component անոտացիայով, սրա մասնավոր դեպքեր են՝ @Repository, @Service, @Controller, @Configuration ավելի նեղ սահմանումների համար։

**Մետաանոտացիաներ**

Այն անոտացիաներն են որոնք կարող են կիրառվել այլ անոտացիաների համար։ Օր @Service անոտացիան գրադարանում իր վրա ունի @Component, @Document անոտացիա։

Կոմպոնենտ կարելի է հայտարարել նաև մեկ այլ կոմպոնենտի մեջ օգտագործելով @Bean անոտացիան՝

@Component

public class FactoryMethodComponent {

@Bean

@Qualifier("public")

public TestBean publicInstance() {

return new TestBean("publicInstance");

}

}

Անոտացիաներով ստեղծվող Կոմպոնենտների համար կիրառելի է @Scope անոտացիան`

@Scope("prototype")

@Repository

public class MovieFinderImpl implements MovieFinder {

// ...

}

**@Qualifier** – bean – ի համար անուն սահմանելու համար։

@Component

@Qualifier("Action")

public class ActionMovieCatalog implements MovieCatalog {

// ...

}

**@Inject** – ը ունի նույն ֆունկցիան ինչ որ @Autowired – ը և կիրառվում է ճիշտ դրա պես։ bean – ը կարող է նշվել @Named անոտացիայով։

@Inject

public void setMovieFinder(@Named("main") MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

@**Named և @ManagedBean** անոտացիաները կարող են օգտագործվել @Component – ի փոխարեն։

@Named("movieListener") // @ManagedBean("movieListener") could be used as well

public class SimpleMovieLister {

private MovieFinder movieFinder;

@Inject

public void setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

// ...

}

**Անոտացիաները Spring – ում և jakarta – ում և դրանց տարբերությունները։**

|  |  |  |
| --- | --- | --- |
| **Spring** | **Jakarta** |  |
| @Autowired | @Inject | Inject – ը չունի required ատրիբուտ |
| @Component | @Named @ManagedBean |  |
| @Scope("singleton") | @Singleton |  |
| @Qualifier | @Qualifier / @Named |  |
| @Value |  |  |
| @Lazy |  |  |

**Կոնֆիգուրացիա Java կոդով**

Կոնֆիգուրացիան կարելի է կազմակերպել նաև Java կոդի միջոցով։

**@Bean և @Configuration**

Java կոդի միջոցով կոնֆիգուրացիայի առանցքը դա @Configuartion անոտացիայով դեկորացված կլասսն ու դրանում @Bean անոտացիայով դեկորացված մեթոդներն են։

**@Bean – ը** կիրառվում է մեթոդների հետ և ցույց է տալիս որ մեթոդը ստեղծում և կարգավորում է նոր օբյեկտ որիը կառավարվելու է Spring IoC կոնտեյների կողմից՝ այսինքն ստեղծում է bean: ունի նույն նշանակություն ինչ որ <beans/> էլեմենտը xml – ում։ @Bean անոտացիայով մեթոդները կարող են լինել ցանկացած կոմպոնենտի մեջ բայց սովորաբար օգտագործվում են @Configuration կլասսներում։ @Bean – ը կորղ է հայտարարվել նաև ինտերֆեյսի մեջ՝

public interface BaseConfig {

@Bean

default TransferServiceImpl transferService() {

return new TransferServiceImpl();

}

}

@Configuration

public class AppConfig implements BaseConfig {

}

Կարող է մեթոդի վերադարձվող տիպը կարող է լինել նաև ինտերֆեյսի տիպը։ bean – երը ստեղծվում են ըստ դրանց հայտարարման հերթականության։

Կախվածությունները այլ bean – երից սահմանվում են inject – ով։

Այս ձևով ստեղծված bean – երը ունեն նույն կյանքի ցիկլը։ Կլասսներում հայտարարված init ու destroy մեթոդները կարող են նշվել @Bean – ի արգումնետներում։

@Bean(initMethod = "init")

@Bean(destroyMethod = "cleanup")

Եթե պետք չէ որ destroy մեթոդը աշխատի ապա կարելի է դրան տալ դատարկ string`

@Bean(destroyMethod = "")

Bean – ի հետ կարելի է օգտագործե նաև @Scope անոտացիան

@Bean

@Scope("prototype")

public Encryptor encryptor() {

// ...

}

Որպես default id օգտագործվում է @Bean մեթոդի անունը, բայց կարելի է նաև id – ն նշել առանձին անոտացիայի արգումենտի մեջ՝ @Bean("myThing"); Կեղծանուններ տալու համար կարելի է փոխանցել անունների մասիվ՝ @Bean({"dataSource", "subsystemA-dataSource"}),

@Description անոտացիան՝ նկարագրություն ավելացնելու համար @Description("Provides a basic example of a bean");

**@Configuration – ը** կլասսի մակարդակի անոտացիա է և ցույց է տալիս որ կլասսի նախատեսված է որպես bean – երի հայտարարման աղբյուր։ նույն կլասսում հայտարարելու շնորհիվ կարող են նաև հեշտությամբ սահմանվել կախվածություններ իրարից։

@Configuration

public class AppConfig {

@Bean

public MyService myService() {

return new MyServiceImpl();

}

}

Կարևոր է որ կախվածություններ սահմանելիս bean մեթոդի մեջ չկանչվի մեկ այլ bean մեթոդ այլ անհրաժեշտ կախվածությունները inject լինեն։

@Import – անոտացիան օգտագործվում է config – ները մուդուլավորել, այսինքն կոնֆիգուրացիոն կլասսում իմպորտ անել մեկ այլ կոնֆիգուրացիոն կլասսներ։ @Import(ConfigA.class)

Իմպորտ եղած կոնֆիգւրացիոն կլասսում հայտարարված bean – երը կարող են inject լինել այն կլասսում որտեղ իմպորտ են եղել։ Եթե նույն bean – ը ստեղծվում է տարբեր կոնֆիգուրացիոն կլասսներում inject – ի դեպքում inject կլինի տվյալ կլասսում հայտարարված bean – ը։

Տարբեր կոնֆիգուրացիոն կլասսներում հայտարարված bean – երը կարող են @Autowired լինել մեկ այլ կոնֆիգուրացիոն կլասսում։

**AnnotationConfigApplicationContext**

Անոտավորված config – ների դեպքում կոնտեքստին (կոնտեյներին) հասանելիության համար օգտագործվում է AnnotationConfigApplicationContext կլասսը որը որպես արգումենտ ընդունում է @Configuration անոտացիայով դեկորացված կլասսները՝

ApplicationContext ctx = new AnnotationConfigApplicationContext(MyServiceImpl.class, Dependency1.class);

Իմպորտենրի դեպքում կարելի է նշել միայն այն կլասսը որը իր մջ իմպորտ է անում անացած կոնֆիգուրացիաները։

Կլասսի էկզեմպլյար կարելի է ստեղծել նաև դատարկ կոնստրուկտորով և register(MyServiceImpl.class) մեթոդով ավելացնել կասսները։

@Configuration – կլասսի հետ կարելի է օգտագործել նաև @ComponentScan(basePackages = "com.acme") – ը <context:component-scan base-package="com.acme"/> - ի փոխարեն։

Կոնֆիգուրացիան xml – ի ու java կոդի միջոցով կարող է լինել XML կենտրոնացված և @Configuration կենտրոնացված։

**Աբստրակտ միջավայր**

Environment ինտերֆեյսը դա կոնտեյներում ինտեգրված աբստրակցիա է որը մոդելավորում է application - ի միջավայրի 2 առանցքային ասպեկտ՝ պրոֆիլներ և հատկություններ։

Պրոֆիլը դա bean կոմպոնենտների անվանավորված խումբ է որը պետք է ստեղծվի կոնտեյներում միայն այն դեպքում երբ տվյալ պրոֆիլը ակտիվ է։ կոմպոնենտների պատկանելիությունը պրոֆիլին կարող է սահմանվել @Profile անոտացիայով կամ xml – ի միջոցով։ Evironment – ի դերը պրոֆիլի դեպքում կայանում է նարանում որ որոշում է թե որ պրոֆիլներն են ակտիվ և թե որոնք պետք է լինեն default ակտիվ։

Հատկությւոնները կարևոր են բոլոր application – ների համար։ Environment – ի դերը հատկությունների դեպքում դա գործիքների տրամադրումն է այդ հատկությունների օգտագործման համար։

Պրոֆիլների միջոցով կարելի է կոմպոնենտները գրանցել տարբեր միջավայրերում և դրանք հասանելի կլինեն միայն այդ միջավայրում։

**@Profile** – ը օգտագործվում է կոնֆիգուրացիոն կլասսի, bean ստեղծող մեթոդների մակարդակում և ցույց է տալիս որ կոմպոնենտը ենթակա է ստեղծման եթե ակտիվ է դրա արգումենտում նշված պրոֆիլը։ Եթե պրոֆիլը ակտիվ չէ ապա այդ կլասսի և իմպորտ եղած կլասսների բոլոր bean – երը անտեսվում են։

@Configuration

@Profile("development")

public class JndiDataConfig {

}

@Configuration

@Profile("production")

public class JndiDataConfig {

}

Xml – ի դեպքում <beans> էլեմենտը ունի profile ատրիբուտ որը աշխատում է նույն սկզբունքով։

Պրոֆիլը / պրոֆիլները ակտիվացնելու համար՝

annotationConfigApplicationContext.getEnvironment().setActiveProfiles("profile1", "profile2");

Եթե պրոֆիլը ունի “default” արժեքը ապա այն կակտիվանա ավտոմատ։

Evironment – ից կարելի է ստանալ նաև property – ներ առանց EL օգտագործելու՝

Environment.getProperty(“propertyName”)

ApplicationContext – ում event – ներին հետևելը կատարվում է ApplicationEvent կլասսի և ApplicationListener ինտերֆեյսի միջոցով։

Ստանդարտ event – ները հետևայալ կլասսներում են և հետևում են իրենց անվանը համապատասխան իրադարձություններին՝

ContextRefreshedEvent

ContextStartedEvent

ContextStoppedEvent

ContextClosedEvent

RequestHandledEvent

ServletRequestHandledEvent

Պետք է ժառանգել նշված կլասսները։

Custom event սահմանող կլասսներ ստեղծելու համար պետք է կլասսը ժառանգել ApplicationEvent – ից, իսկ դրա listener ունենալու համար:

Custom Event – ների մասը առանձին նայել

**BeanFactoryAPI \_**

Ապահովում է Spring IoC – ի ֆունկցիոնալության բազային հիմքը։ Այսինքն իրենից ներկայացնում է կոնտեյների մակարդակ։ Կարելի է ասել որ ApplicationContext – ը իր մեջ ներառում է BeanFctory – ի բոլոր հնարավորությունները և օգտագործման տեսանկյունից ավելի նախընտրելի է։

**2․ Ռեսուրսներ**

Ցածր մակարդակի ռեսուրսների հետ աշխատելու համար սահմանված էResource ինտերֆեյսը որը ժառանգում է InputStreamSource – ից։

Resource ինտերֆեյսը ունի հետևյալ կլասս ռեալիզացիաները՝

* [UrlResource](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#resources-implementations-urlresource)
* [ClassPathResource](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#resources-implementations-classpathresource)
* [FileSystemResource](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#resources-implementations-filesystemresource)
* [PathResource](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#resources-implementations-pathresource)
* [ServletContextResource](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#resources-implementations-servletcontextresource)
* [InputStreamResource](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#resources-implementations-inputstreamresource)
* [ByteArrayResource](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#resources-implementations-bytearrayresource)

UrlResource – URL կլասսի wrapper – ն է։ Կարող է օգտգործվել այն օբյեկտներին հասանելիության համար որոնք սովորաբար հասանելի են URL – ի միջոցով։ Օր՝ ֆայլերին, HTTPS հղուներ և այլն։ URL – ները ունեն ստանդարտավորված String նախաբաններ օր file: , https: ևայլն։

[ClassPathResource](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#resources-implementations-classpathresource) ռեալիզացիա է File – ի համար։

**3. Validation, Data Binding, and Type Conversion**

Validator ինտերֆեյսը նախատեսված է օբյեկտների ստուգման համար։ Validator – ը աշխատում է Errors օբյեկտի օգտագործմամբ, որի նախատեսված է վալիդացիայի ընթացքում ի հայտ եկած error – ները իր մեջ պահելու համար։

Ինտերֆեյսը ունի 2 մեթոդ՝x

boolean supports(Class clazz) – կարող է արդյոք վալիդատորը ստուգել տվյալ օբյեկտը

void validate(Object obj, Errors e) – ստուգում է փոխանցված օբյեկտը

որպես օժանկակող կլասս գոյություն ունի ValidationUtils – ը։

Կարելի է ունենալ նաև ունիվերսալ վալիդատոր կլասս բոլոր ստուգումների համար բայց ավելի նպատակահարմար է վալիդացիայի լոգինակն ինկապսուլացնել և յուրաքանչյուր օբյեկտի համար ունենալ առանձին վալիդատոր կլասս։

error message ավելացնելու համար պետք է օգտագործվի errors.rejectValue(fieldName, errorMessage);

Errors օբյեկտ ստեղծելու համար կարելի է օգտագարծել MapBindingResult օբյեկտից՝

Map<String,String> map = new HashMap<String,String>();

MapBindingResult err = new MapBindingResult(map, User.class.getName());

BindingResult – ը bean է որը կլասսում կարող է inject լինել։

**BeanWrapper**

Օգտագործվում է օբյետների դաշտերի հետ գործողություններ կատարելու համար։

BeanWrapper company = new BeanWrapperImpl(new Company());

// setting the company name..

company.setPropertyValue("name", "Some Company Inc.");

կարող է փոխանցվել նաև PropertyValue տիպի օբյեկտ։

PropertyValue value = new PropertyValue("name", "Some Company Inc.");

company.setPropertyValue(value);

**Տարբեր տիպերի կոնվերտացումը կարող է կատարվել մի քանի եղանակներով**

Converter<S, T> ինտերֆեյսը իմպլեմենտացնելով և դրա T convert(S source); մեթոդը ռեալիզացնելու միջոցով։

Վերադարձվող տիպը ժառանգական կապով սահմանափակելու համար կարող է օգտագործվել ConverterFactory – ն՝

public interface ConverterFactory<S, R> {

<T extends R> Converter<S, T> getConverter(Class<T> targetType);

}

Տվյալների ֆորմատավորումը կարող է կատարվել անոտացիաների միջոցով՝

//ISO (гггг-мм-дд):

@DateTimeFormat(iso=ISO.DATE)

private Date date;

**Java bean validation**

օբյեկտների դաշտերի վալիդացիաներ կարելի է սահմանել անոտացիաների միջոցով։

@NotNull

@Size(max=64)

private String name;

@Min(0)

private int age;

**4․ Spring Expression Language (SpEL)**

Թույլ է տալիս runtime դինամիկ ներդնել օբյեկտներ։

SpEL – ը գրվում է ' ' – ի մեջ և parse է արվում ExpressionParser տիպի օբյեկտի միջոցով։

ExpressionParser parser = new SpelExpressionParser();

Expression exp = parser.parseExpression("'Hello World'.concat('!')");

String message = (String) exp.getValue();

Կարելի է օգտագործել նաև այլ ֆունկցիաներ օր՝ toUpperCase()

SpEL – կոնտեքստում կարևոր է EvaluationContext ինտերֆեյսը և դրա SimpleEvaluationContext և StandardEvaluationContext ռելիզացիաները։ Կոնտեքստը օգտագործվում է արտահայտությունների մշակման ժամանակ՝ լրացուցիչ դաշտերի, մեթոդների ընդլայնման նաև parsing – ի համար։ Օր՝

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

Օբյեկտների դաշտերի հետ աշխատելիս SpEL – ը տակից օգտագործում է դրանց get մեթոդները։

SpEL – ի confg – ները կարելի կարգավորել SpelParserConfiguration – ի միջոցով։

Spring – ը իր մեջ ներառում է expression - ների կոմպիլյատոր, որը default – ով միացված չէ։ Կարող է աշխատել 3 ռեժիմներով՝ OFF, IMMEDIATE, MIXED։

SpEL – ը xml – ում ունի հետևյալ տեսքը՝

#{ <expression string> } օր՝ <property name="randomNumber" value="#{ T(java.lang.Math).random() \* 100.0 }"/>

Անոտացիայի միջոցով կարելի է կատարել հետևյալը՝

@Value("#{ systemProperties['user.region'] }")

private String defaultLocale;

նույնը կիրառելի է նաև set մեթոդների և կոնստրուկտորների մեջ։

SpEL – ում կարող են լինել հետևյալ տիպի արտահայտություններ՝

**Տառային** որոնք գրվում են մեկ չակերտների մեջ ' ', կարող են օգտագործվել նաև թվային լիթերալներ ամբողջ, կոտորակային, 16 – ական և այլն տեսքով։ Թույլատրվում են նաև թվաբանական գործողություններ։

String helloWorld = (String) parser.parseExpression("'Hello World'").getValue();

double avogadrosNumber = (Double) parser.parseExpression("6.0221415E+23").getValue();

// evals to 2147483647

int maxValue = (Integer) parser.parseExpression("0x7FFFFFFF").getValue();

boolean trueValue = (Boolean) parser.parseExpression("true").getValue();

Object nullValue = parser.parseExpression("null").getValue();

**Զանգվածներ, լիստեր և այլն**

**Ներդրված լիստեր**

List numbers = (List) parser.parseExpression("{1,2,3,4}").getValue(context);

**Ներդրված map**

Map inventorInfo = (Map) parser.parseExpression("{name:'Nikola',dob:'10-July-1856'}").getValue(context);

Map mapOfMaps = (Map) parser.parseExpression("{name:{first:'Nikola',last:'Tesla'},dob:{day:10,month:'July',year:1856}}").getValue(context);

**Ստեղծվող մասիվ**

int[] numbers2 = (int[]) parser.parseExpression("new int[]{1,2,3}").getValue(context);

**Մեթոդներ**

Մեթոդները կարող են կանչվել java – ի սինտաքսով։

**Օպերատորներ**

Կարող են օգտագործվել համեմատության, տրամաբանական, մաթեմատիկական, վերագրման օպերատորներ։ Կիրառելի է նաև instanceof արտահայտությունը և matches – ը որին հաջորդում է regexp: Համեմատության օպերատորները կարող են փոխարինվել նաև համարժեք տեքստային արտահայտությամբ օր lt - <, gt - >։

Տրամաբանական օպերատորներից՝ and( && ), or( || ), not( ! )։

Մաթեմատիկական օպերատորների դեպքում կիրառվում է նախապատվության ստանդարտ սխեման։

SpEL – ում կարող է օգտագործվել ts – ի ? օպերատորը nullPointer չունենալու համար՝

String city = parser.parseExpression("placeOfBirth?.city").getValue(context, tesla, String.class);

**5. Aspect Oriented Programming**

Maven dependency Spring AOP: AOP – լրացնում է OOP – ն առաջարկելով այլ մոտեցում։ Oop – ում առաջցքային միավորը դա կլասսն է իսկ AOP – ում ասպեկտը։ Spring – ի կոնտեյները կախվածությւոն չունի AOP – ից, այն կիրառելի է ըստ ցանկության և միայն լրացնում է կոնտեյները տրամադրելով խնդրիների միջանկյալ լուծումներ։

AOP – ն դա ծրագարավորման մեթոդիկա է որը հիմնված է հիմնական և միջանկյալ ֆունկցիոնալի իրարից անջատելու գաղափարի վրա։ Միջանկյալ ֆունկցիոնալը գրվում է Aspect կլասսներում։ AOP – ն հիմնված է ասպեկտի վրա՝ կոդի բլոկ որը ինկապսուլացնում է միջանկյալ գործողությունները:

AOP – ի կոնտեքստում ցանկացած ֆունկցիոնալություն ուղղակիորեն չի առնչվում բիզնես լոգիկաին համարվում է միջանկյալ կամ cross-cutting logic: Օր՝ լոգավորում, security check, տրանզակցիաների մշակում, exception – ների մշակում, քեշավորում և այլն։

AOP – ի աշխատանքը կազմակերպվում է AOP proxy – ի միջնորդի միջոցով որը ավտոմատ աշխատացնում է Spring – ը։

AOP կիրառելիս կրկնվող միջանկյալ ֆունկցոինալը կենտրոնանում է մեկ կամ մի քանի կլասսներում որը ավելի ճկուն է դարձնում դրա փոփոխումը կամ ավելացումը։ Բիզնես կոդը դառնում է ավելի կոմպակտ և ընթեռնելի։

AOP – ի բացասական կողմը այն է որ ասպեկտների աշխատանքը ազդում է ծրագրի արագության վրա։

**Aspect**: AOP – ն ռեալիզացվում է սովորական կլասսների սխեմայի կամ @Aspect – ի հիման վրա։ Իր մեջ միավորում է Join point – ը և Advice – ը:

**Join point:** Կետ ծրագրի իրականացման ընթացքում որտեղ կարող է ներդրվել advice – ը: Որպես join point այս պահին support է արվում միայն մեթոդի կանչը։

**Advice**: Գործողություն ասպեկտի կողմից որոշակի join point – ում։ Այն լոգիկան որը պետք է կանչվի join point – ում: Կարող է լինել մեթոդից մինչև, հետո, հաջող ավարտից հետո, exception – ով ավարտից հետո, շուրջը այսինքն մինչև և հետո: Before advice, After returning advice, After throwing advice, After (finally) advice, Around advice

**Pointcut:** join point – ների բազմություն: Սրա միջոցով որոշվում է թե join point - ը համապատասխանում է արդյոք ասպեկտին:

**Introduction:** Ասպեկտի ներդրում որոշակի կոդի բլոկում:

**Target object:** Օբյեկտ որին advice են անում ասպեկտները։

**AOP proxy:** Օբյեկտ ստեղծված AOP պլատֆորմի կողմից ասպեկտների ռեալիզացիայի համար։

**Weaving:** ասպեկտի կիրառում target objet – ի վրա:

AOP – ն կարող է կիրառվել Sprin AOP – ի կամ AspectJ – ի միջոցով, իսկ անոտացիոն հատված ըկարող է լինել @AspectJ – ի կամ xml – ի միջոցով:

**@AspectJ style:** Ասպեկտները հայտարարվում են որպես անոտավորված Java կլասսներ: Միացվում է maven AspectJ Weaver -ով: Որից հետո Configuration կլասսի վրա ավելացվում է @EnableAspectJAutoProxy անոտացիան: Xml -ում՝ <aop:aspectj-autoproxy/>:

Ասպեկտի հայտարարումը: Ցանկացակ bean որը ունի @Aspect անոտացիան Spring – ը օգտագործում է AOP-ի կարգավորումներում: Միայն @Aspect անոտացիայով կոմպոնենտ չի ստեղծվի դրա հետ միասին պետք է օգտագործվի նաև bean հայտարարման մեխանիզմներից որևէ մեկը։

**Pointcut – ի հայտարարում։** Սրանց միջոցով նշվում են մեզ հետաքրքրող կետերը / միացման կետերը և թույլ են տալիս կառավարել ասպեկտում հայտարարված մեթոդների կանչերը։ Որպես pointcut support են արվում միայն մեթոդները։ Կազմված է 2 մասից մեթոդը ևpointcut expression – ը որը ցույց է տալիս թե մեթոդի իչպիսի իրականացումն է մեզ հետաքրքրում։ Կատարվում է **@Pointcut** անոտացիայի միջոցով։ Օր՝

@Pointcut("execution(\* transfer(..))") // the pointcut expression

private void anyOldTransfer() {} // the pointcut signature

նշանակում է հայտարարել anyOldTransfer pointcut որը համապատասխանում է ցանկացած transfer անունով մեթոդի իրականացմանը։

Pointcut expression – ում support են արվում հետևյալ ցուցիչները՝

execution։ մեթոդների իրականացման համար, ամենահաճախ օգտագործվողն է։

within, this, target, args, @target, @args, @within, @annotation սահմանափակումների համար են։

proxy օբյեկտը կապված է this – ին իսկ նպատակայինը կապված է target -ին։

Օր՝

@Pointcut("execution(public \* \*(..))")

private void anyPublicOperation() {}

anyPublicOperation() – ը համապատասխանում է ցանկացած public մեթոդի։

@Pointcut("within(com.xyz.myapp.trading..\*)")

private void inTrading() {}

inTrading համապատասխանում է իրականացվելիք մեթոդը գտնվում է նշված մոդուլում / package -ում։

@Pointcut("anyPublicOperation() && inTrading()")

private void tradingOperation() {}

pointCut – ների դեպքում կիրառելի են նաև տրամաբանական օպերատորները։

**Advice – ի հայտարարումը**

Advice – ը կապված է pointcut – ի հետ և կարող է աշխատել համապատասխան pointcut մեթոդից առաջ, հետո կամ շուրջը (և առաջ և հետո)։ Դրա համար կիրառվում են համապատասխան անոտացիաները՝ @Before(), @AfterReturning(), @AfterThrowing(), @After(), @Around()

@After() – աշխատում է try-catch օպերատորի finaly – ի նման, այսինքն միշտ աշխատում է։

Արգումենտում կարող է նշվել կամ մեթոդի անունը կամ pointCut արտահայտություն։

@Before("com.xyz.myapp.CommonPointcuts.dataAccessOperation()")

public void doAccessCheck() {

// ...

}

@Before("execution(\* com.xyz.myapp.dao.\*.\*(..))")

public void doAccessCheck() {

// ...

}

args(param) – ի միջոցով կարելի է փոխանցել արգումենտներ մեթոդին, պարամետրի անունը պետք է համընկնի։

Եթե նույն pointcut – ին միացած են 1 – ից ավելի advice – ներ ապա դրան կաշխատեն ըստ նախապատվության, որը սահմանվում է @Order անոտացիայով կամ Orderer ինտերֆեյսով։ Ավելի փոքր արժեք վերադարձնող Advice – ի նախապատվությունը ավելի բարձր։

Նույն գործողությունները կարելի է կատարել նաև Xml սինտաքսի միջոցով։ Դրա համար պետք է spring-aop dependency – ն։ Ասպեկտներին առնչվող բոլոր էլէմենտները պետք է գրվեն <aop:config> էլեմենտի մեջ որը կարող է լինել 1 – ից ավելի և իր մեջ պարունակել pointcut, advice և այլ էլեմենտներ։

Ասպեկտի հայտարարում՝

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

...

</aop:aspect>

</aop:config>

<bean id="aBean" class="...">

...

</bean>

Pointcut - ի հայտարարում՝

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.\*.\*(..))"/>

Advice - ի հայտարարում (կարող են լինել նույն 5 տեսակների)՝

<aop:aspect id="beforeExample" ref="aBean">

<aop:before

pointcut-ref="dataAccessOperation"

method="doAccessCheck"/>

...

</aop:aspect>

Անոտացիաները և Xml – ը կարող են կիրառվել միաժամանակ։

**7. Null-safety**

Null – երից ապահով աշխատելու համար կան հետևյալ անոտացիաները, որոնք կարող են կիրառվել պարամետրի, դաշտի կամ վերադարձվող արժեքի հետ

[@Nullable](https://docs.spring.io/spring-framework/docs/6.0.5/javadoc-api/org/springframework/lang/Nullable.html) –կարող է լինել Null

@NonNull – չի կարող լինել null, կոդի ռեդակտորները տեսելով անոտացիան ցույց կտան error եթե լինի null

[@NonNullApi](https://docs.spring.io/spring-framework/docs/6.0.5/javadoc-api/org/springframework/lang/NonNullApi.html) – package level անոտացիա նշանակում է որ նշված package - ում բոլոր պարամետրերը և վերադարձվող արժեքները չեն կարող լինել null`

@NonNullApi **package** org.baeldung.nullibility;

[@NonNull](https://docs.spring.io/spring-framework/docs/6.0.5/javadoc-api/org/springframework/lang/NonNullApi.html)Fields - package level անոտացիա նշանակում է որ նշված package - ում բոլոր բոլոր դաշտերը չեն կարող լինել null`

@NonNullFields **package** org.baeldung.nullibility;