**Java Annotations**

* Java Annotations sunt etichete speciale adaugate claselor.
* Ele furnizeaza meta-date despre clase.(diferite date legate de starea sau implementarea clasei)
* Unele sunt procesate la run-time, altele la compile-time
* Acestea ne ajuta sa configuram Spring beans fara a scrie ceva in XML
* ***Anotatiile(niciuna!!!) nu functioneaza la obiecte create cu new!Doar la beans create de Spring***

De ce le utilizam:

* XML configuration poate fi destul de mare la un moment dat
* Minimalizeaza configul din XML
* Sunt usor de folosit.

**Cum fuctioneaza anotatiile**

1. Spring scaneaza intai toate clasele unde se gasesc anotatii(daca component-scan e activat)
2. Odata ce gaseste o clasa cu o anotatie @Component, inregistreaza acest bean in Spring Container, **fara a mai folosi XML**.,sau daca gaseste o clasa fara @Component, dar cu anotatii, tot inregistreaza aceste anotatii, sa stie ca ele sunt acolo. Spring nu va scrie nimic in el deja.

**Procesul de Dezvoltare/utilizare a anotatiilor**

1. Activam Component Scanning in Spring Config file
2. Adaugam anotatia @Component(“ID”) la clasa, pentru ca beanul sa se creeze automat
3. Recuperam beanurile

Pentru a activa Component Scanning in Spring Config File, folosim tagul:

<context:component-scan base-package="com.example.udemy">

Unde base-package e package pe care il va scana complet, inclusiv si subpackage din el.

**Default Bean**

Spring suporta si Default Bean id. Default bean ID e generat de Spring cand noi nu oferim un ID in annotation.

Default bean id e numele clasei, doar ca prima litera e lower case.

Daca numele clasei incepe cu 2 liter upper case, Spring va lasa numele beanului identic cu al clasei, de ex:

MyClass devine myClass

MYclass ramane Myclass ca nume al beanului

Default name nu se aplica si in XML!

**AutoWiring(cablare automata)**

* AutoWiring inseamna interconectarea automata a obiectelor, adica Spring injecteaza un obiect in alt obiect, care depinde de el, prin cautarea unui obiect, a carui clasa are @Component,sau e creat in container prin XML, care sa corespunda tipului de obiect de care acesta e dependent.
* Cand vom cere ca Spring sa injecteze un obiect prin anotatii, Spring va cauta un obiect in container potrivit, dupa tip,adica va scana toate obiectele din container, a caror clasa logic ca au @Component,sau sunt beanuri inregisrate in xml, si odata ce il gaseste, il injecteaza. Acest obiect la fel e unic, dupa principiul singleton, ca si atunci cand facem acelasi lucru in XML. Adica adresa lui de memorie din interiorul obiectului va fi identica cu atunci cand il vom recupera si prin getBean();

**AutoWiring Injection Type**

* Constructor Injection
* Setter Injection
* Field Injection(nu e posibil cu XML)

**Constructor Injection prin AutoWiring**

1. Definim clasa/interfata de care depinde obiectul si folosim @Component
2. Cream clasa obiectului ce depinde de altul si construim un contructor pentru injectare si definim clasa tot ca @Component
3. Folosim @Autowired la constructor**(Utilizarea lui e optionala, Spring oricum va face acelasi lucru si fara el)**

@Component  
class TrackCoach implements Coach {  
 protected FortuneService fortuneService;  
  
 @Autowired  
 public TrackCoach(FortuneService fortuneService){  
 this.fortuneService = fortuneService;  
 }  
  
 @Override  
 public String getDailyWork() {  
 return "Go to the gym every 2 days";  
 }  
  
 @Override  
 public String getDailyFortune() {  
 return fortuneService.getFortune();  
 }  
}  
interface FortuneService{  
 String getFortune();  
}  
@Component  
class HappyFortuneService implements FortuneService{  
  
 private String[] fortune={"You're so lucky","This is your the best day","Have a nice day!"};  
  
 @Override  
 public String getFortune(){  
 return fortune[(((new Random()).nextInt(fortune.length)))];  
 }  
  
}

Spring va crea un bean pentru clasa de baza si un bean de care acesta depinde.

Totusi, daca de ex in constructor se va da un parametru de tip interfata, care logic va avea nevoie de un obiect a carui clasa implementeaza interfata, apare o problema cand exista mai multe clase care implementeaza inerfata data si sunt @Component sau in XML. Solutia va fi aratata mai jos.

Din Spring 4, utilizarea lui @Autowired e deja optionala! Spring mereu va cauta un obiect potrivit pentru orice constructor de acest tip a unui bean. Totusi, daca avem mai multi constructori, cel putin unul trebuie sa fie notat cu @Autowired ca Spring sa stie ca trebuie il foloseasca anume pe acesta. Daca e unul default, acela va fi folosit mereu.

**Setter Injection prin AutoWiring**

* **Se face la fel ca la Constructor Injection,** doar ca punem @Autowired la metoda setter. Punerea anotatiei @Autowired e obligatorie la metoda setter! Spring nu va cauta automat aceste metode, ci se va uita doar la cele cu @Autowired.
* Spre deosebire de XML, metoda setter poate avea orice nume, nu neaparat sa inceapa cu set si apoi cu un upper case. Important e doar sa aiba anotatia @Autowired
* Putem avea orice setter methods vrem.

**Field Injection prin AutoWiring**

Acesta metoda foloseste Java Reflection, si prevede ca punem anotatia @Autowired la membrul din clasa, si Spring va injecta dependenta pentru toti membrii cu anotatia data, chiar si cei privati.

class TrackCoach implements Coach {  
 @Autowired  
 protected FortuneService fortuneService;

....

}

**Multiples beans for Autowiring**

* Daca se intampla ca Spring gaseste mai multe obiecte in Container ce pot fi injectate, **el va lua beanul ce are numele identic cu al clasei, dar prima litera e cu lower-case**,chiar daca el tot este vreo subclasa,nu conteaza, important e numele, dar daca asa bean nu e, el nu va sti pe care sa il injecteze, si va da o eroare, spunand ca sunt mai multe beanuri disponibile si va afisa care anume.
* Pentru a rezolva problema data, folosim anotatia

@Qualifier(“ID”)

Daca beanurile noastre au nume default, atunci pur si simplu le punem.

Numele default sunt cele identice cu clasa, doar ca cu lower case prima

litera.

* Qualifier poate fi folosit pentru toate cele 3 tipuri de injectii

1. Field Injection

@Autowired  
@Qualifier("randomFortuneService")  
protected FortuneService fortuneService;

1. Setter Injection – putem pune anotatia fie deasupra metodei, fie inaintea parametrului formal. Acest lucru e din cauza ca in mod normal, metoda seteaza un obiect doar, si deci e logic sa poata fi pus doar pentru acel obiect. Insa, e posibil ca o metoda sa seteze mai multe obiecte, si nu putem repeta anotarea de mai multe ori mai sus de metoda, insa putem sa o folosim pentru fiecare parametru.

@Autowired  
@Qualifier("randomFortuneService")   
 public void setFortuneService(FortuneService fortuneService){  
 this.fortuneService = fortuneService;  
 }

@Autowired  
 public void setFortuneService(@Qualifier("randomFortuneService") FortuneService fortuneService){  
 this.fortuneService = fortuneService;  
 }

1. Constructor – putem pune anotatia doar inaintea parametrului! Constructorul, adesea injecteaza mai multe obiecte, de aceea nu accepta o singura anotatie deasupra sa.

@Autowired  
public TrackCoach(@Qualifier("randomFortuneService") FortuneService fortuneService){  
 this.fortuneService = fortuneService;  
}

**Putem folosi @AutoWired si orice anoatatie si fara sa punem @Component, asa incat putem scrie <bean> in xml, dar nu trebuie sa setam metodele sau constructorii in xml ce sa seteze fielduri, acest lucru fiind facut de anotatii, dar trebuie activat neaparat component-scan in XML**

**Injectarea prin valoare cu anotatii**

Putem injecta prin valoare de la un properties file folosind anotatiile.

1. Cream properties file
2. Il adaugam in contextApplication cu <context:property-placeholder location=””/>
3. Adaugam anotatia @Value(“${foo.ceva}”);

@Value("${foo.message}")  
protected String message;

Putem folosi value si in parametri:

@Autowired  
public Course(@Value("${course.name}") String title, Teacher teacher) {  
 this.title = title;  
 this.teacher = teacher;  
}

**Bean Scope**

Scopul il definim cu anotatia **@Scope(“tip”);**

**Annotation pentru init/destroy-method**

Pentru a rula o metoda la crearea obiectului, folosim anotatia **@PostConstruct**

Pentru a rula metoda la distrugerea lui, **@PreDestroy**

Iarasi, metoda trebuie sa fie fara argument!

Spring nu se ocupa de distrugerea unui obiect cu scope prototype,el il creaza, face injectarile, il livreaza si atat, mai departe clientul face ce vrea cu el, de aceea nici nu exista metoda de destroy. Odata livrat la client, Spring nu mai are nici-o urma a acelui obiect, renunta la el. De asta folosim acelasi ID ca sa obtinem un bean cu scope prototype, caci acel ID nu face referinta la nici-un obiect, daca el e prototype, ci va crea unul nou de fiecare data si il va abandona.