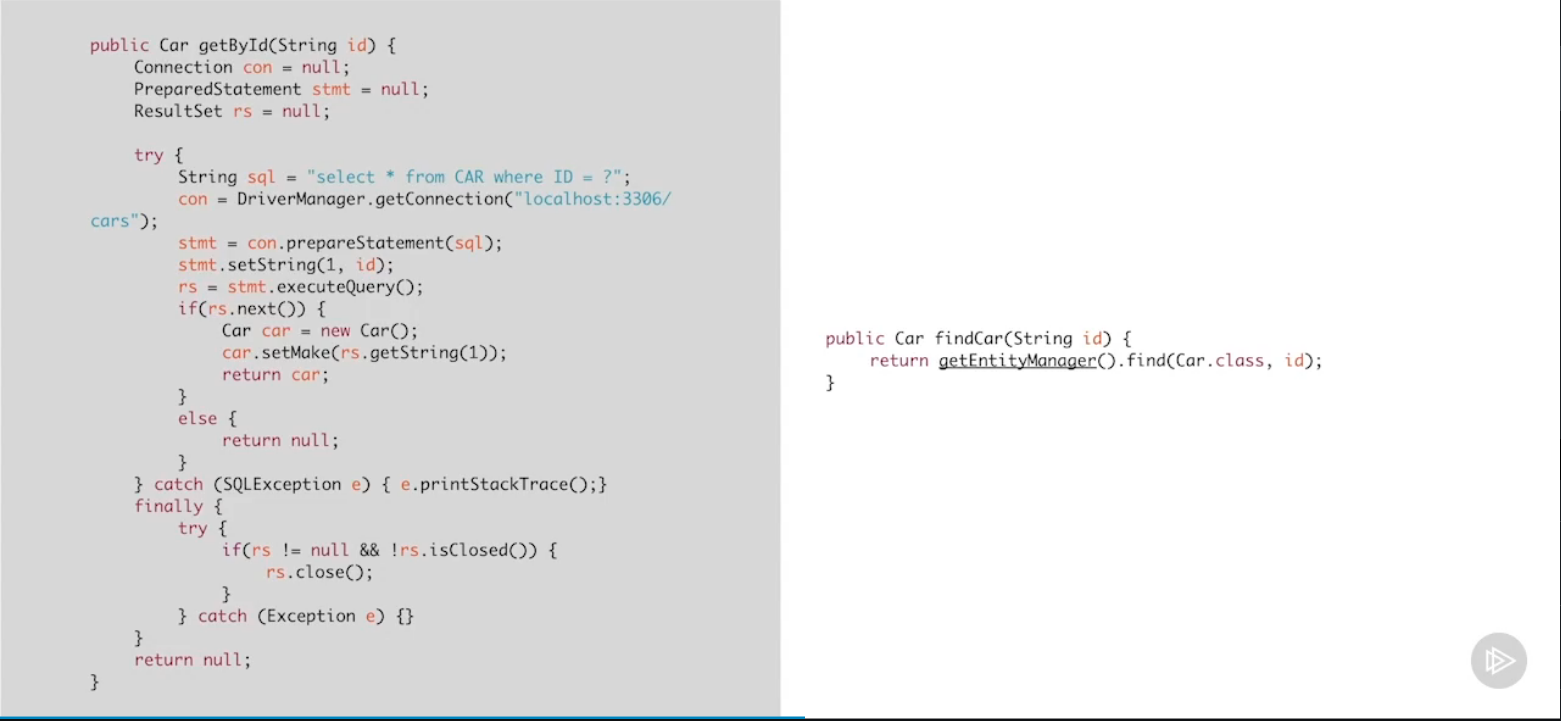
Cod normal care scoate info dintr-o baza de date:



Cod simplificat cu Spring: Spring is using a template method pattern in the background:

How it works:

**Everything in spring is a POJO: Plain old Java Object**

**Spring** in sine poate fi descris drept un Hashmap glorificat si este denumit Application Context

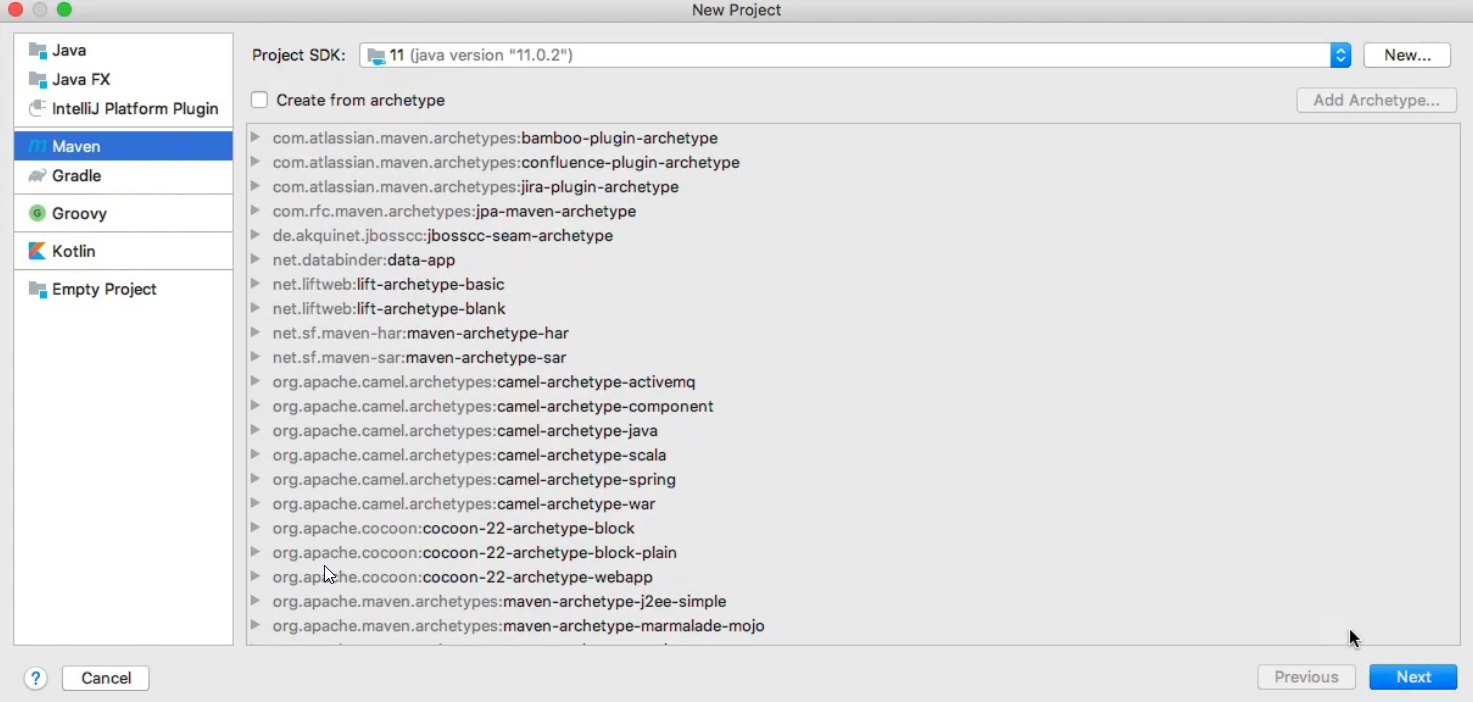
**Application Context** este container-ul Spring care a fost configurat, adica are toate dependentele configurate in el.

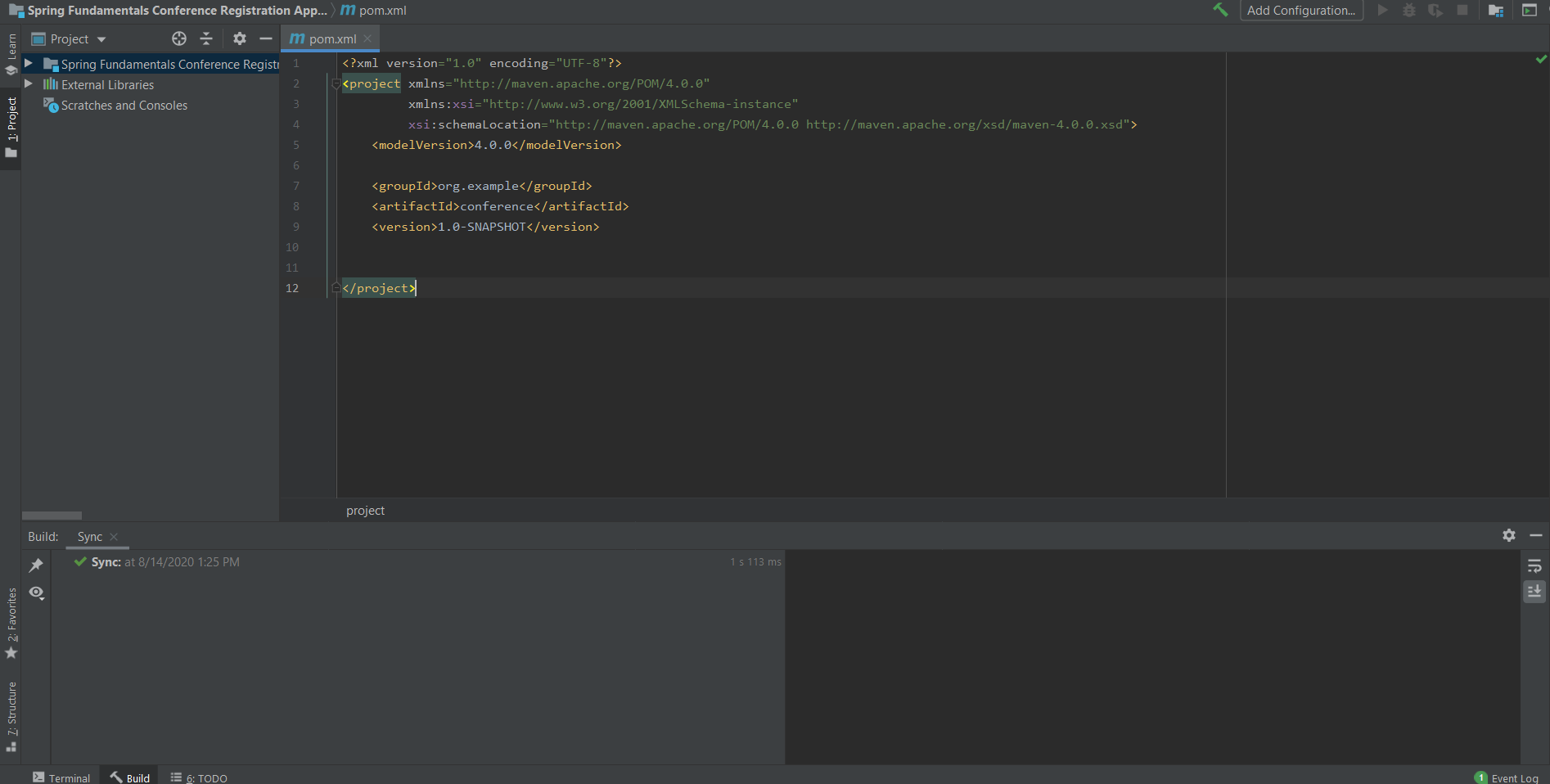
Face putin mai mult decat sa creeze un hashmap care baga in el obiecte.

Prerequissitess: JAVA + MAVEN 🡪 Java Conferencing registration app.

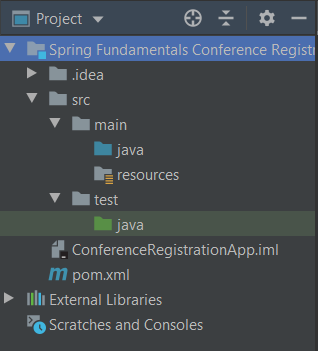
Sample app set-ul:

Cream un proiect nou cu intelij, selectam maven la create project

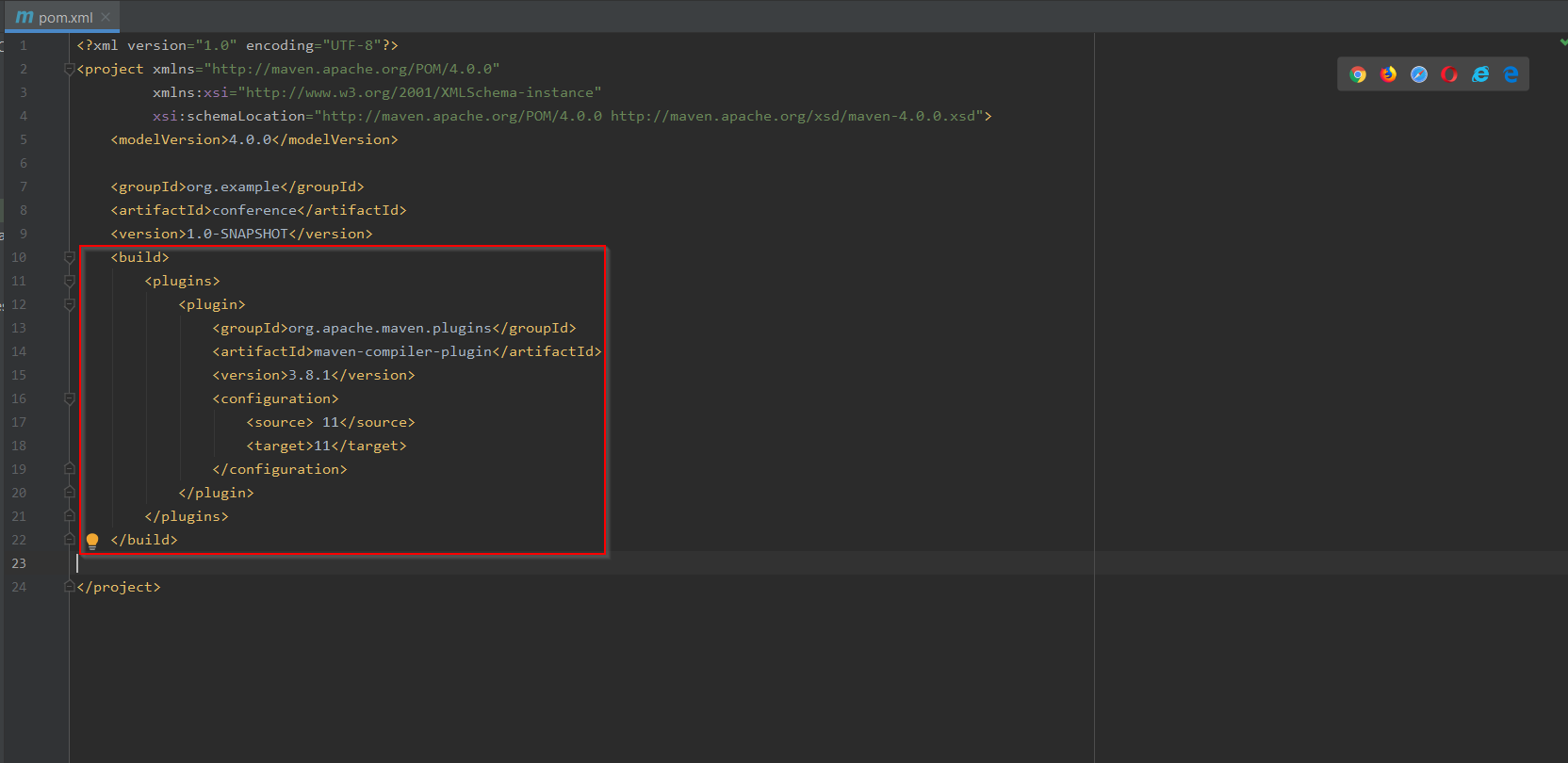


Inauntrul proiectului prima chestie pe care o vedem este basic pom file:

Si putem vedea ca ne-a creat si structura proiectului:



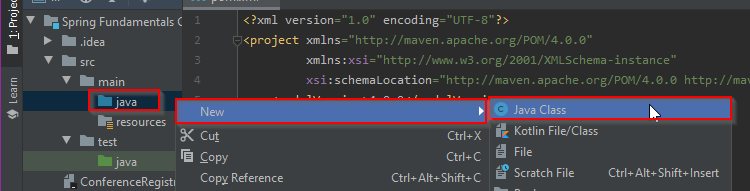
We need to fix smth I nthe Pom 🡪 we need to set a specific build version for our app:



Added this line of code to tell the idee to use the maven compiler and not the idee one.

Addig Java code:

src 🡪 main🡪 java



Putem adauga si un pachet cat timp folosim conventia corecta de nume:



Cream variabile si le cream getter is setter:

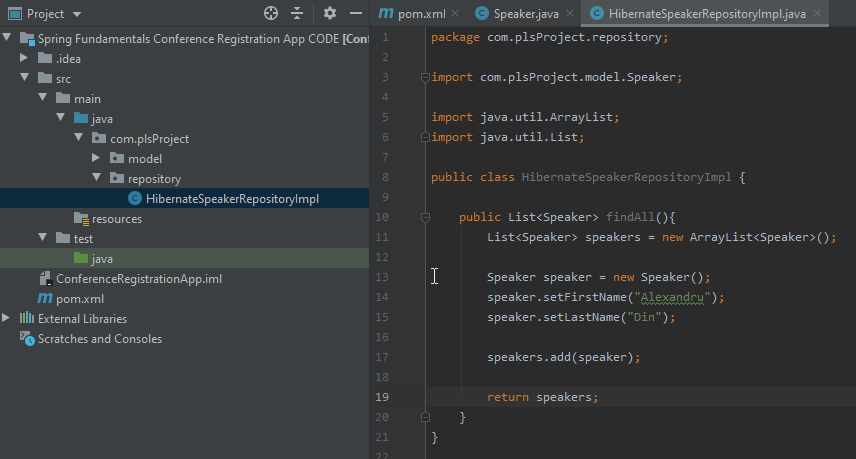
package com.plsProject.model;  
  
public class Speaker {  
 private String firstName;  
 private String lastName;  
  
 public String getFirstName() {  
 return firstName;  
 }  
  
 public String getLastName() {  
 return lastName;  
 }  
  
 public void setFirstName(String firstName) {  
 this.firstName = firstName;  
 }  
  
 public void setLastName(String lastName) {  
 this.lastName = lastName;  
 }  
}

Acum vom crea **obiectul Repository** catre structura de clase (Cea mai complexa.)

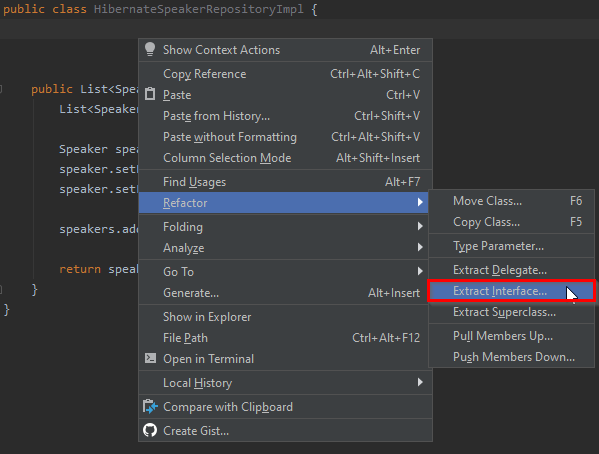
Vom crea un pachet nou care va functiona ca un repository:

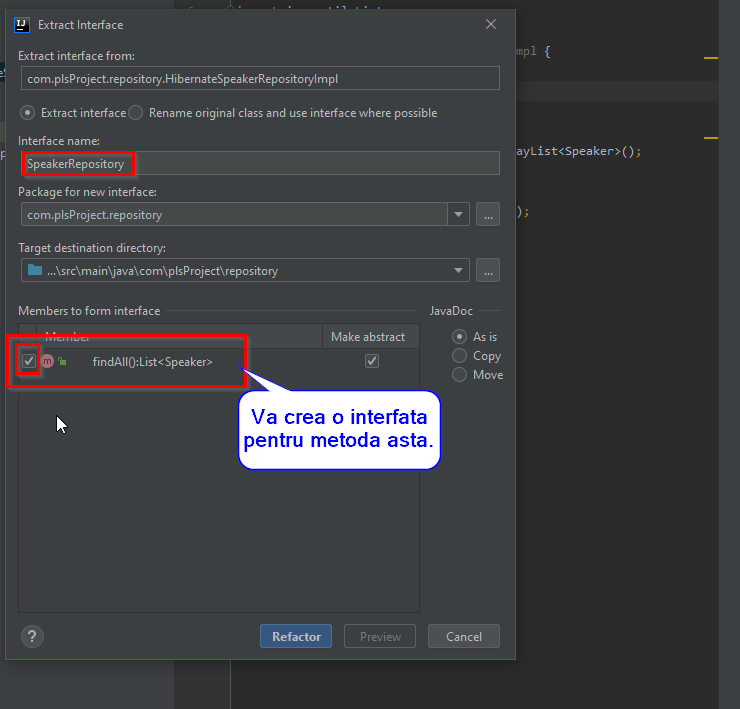
****

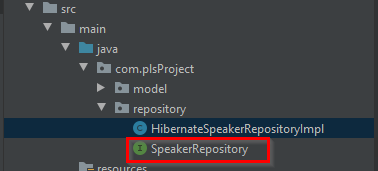
Vom avea o interfata pentru implementare mai tarziu (de aceea nu este o problema ca avem Impl in nume)

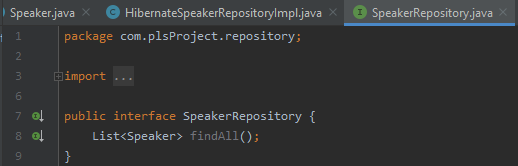


Vrem sa programam catre o interfata:



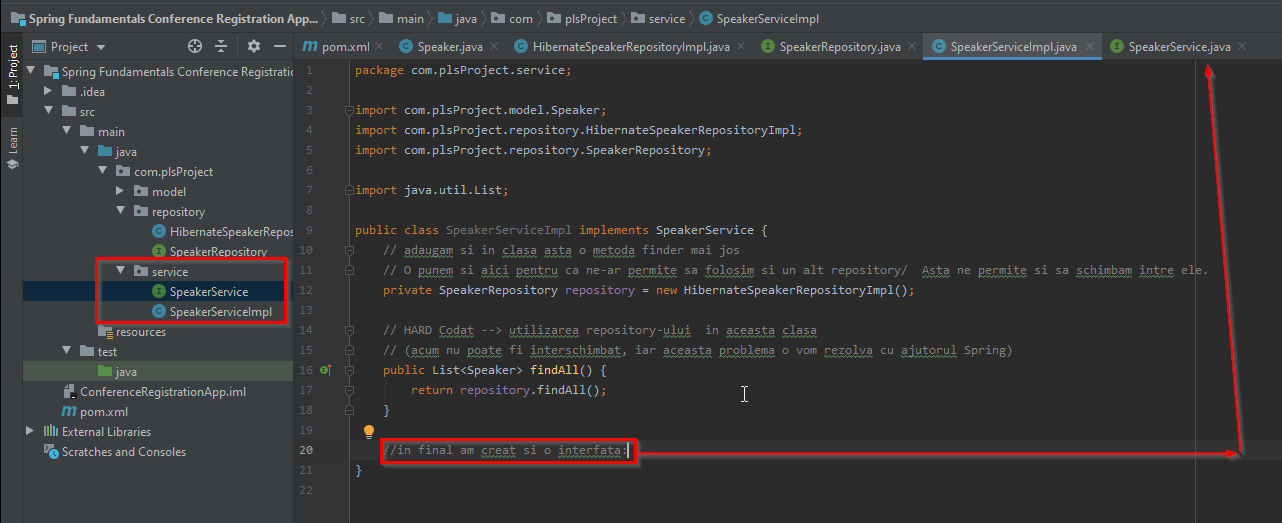






IMP. De oarece avem o interfata acum putem referi utilizarile metodei prin interfata si nu prin clasa **HibernateSpeakerRepositoryImpl** care este urata.

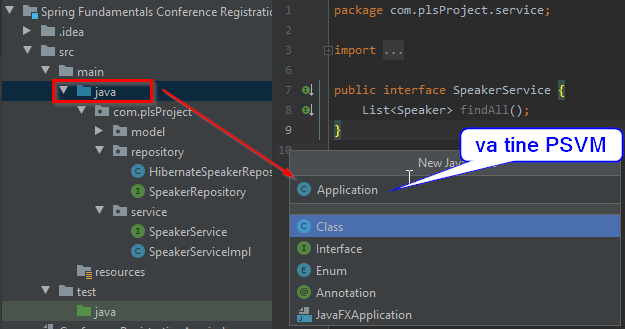
Implementam acum Service Tier ( the business logic):



public class SpeakerServiceImpl implements SpeakerService {  
 // adaugam si in clasa asta o metoda finder mai jos  
 // O punem si aici pentru ca ne-ar permite sa folosim si un alt repository/ Asta ne permite si sa schimbam intre ele.  
 private SpeakerRepository repository = new HibernateSpeakerRepositoryImpl();  
  
 // HARD Codat --> utilizarea repository-ului in aceasta clasa  
 // (acum nu poate fi interschimbat, iar aceasta problema o vom rezolva cu ajutorul Spring)  
 public List<Speaker> findAll() {  
 return repository.findAll();  
 }  
  
 //in final am creat si o interfata:  
}

public interface SpeakerService {  
 List<Speaker> findAll();  
}

Cream O metoda Main simpla care sa testeze bucatile aplicatiei:



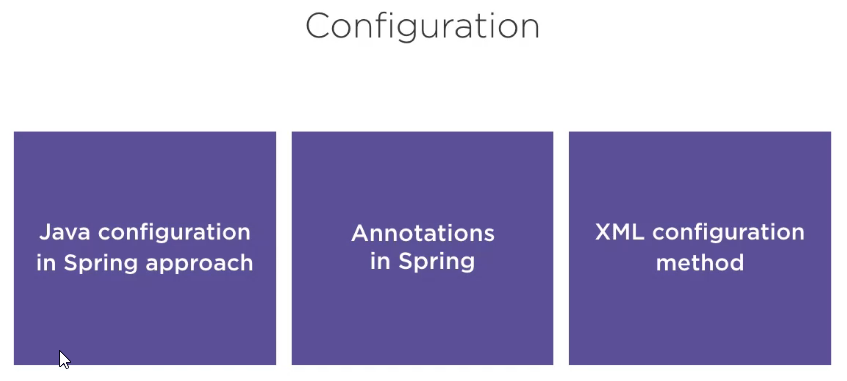
public class Application {  
  
 public static void main(String[] args) {  
  
 SpeakerService service = new SpeakerServiceImpl();  
 System.*out*.println(service.findAll().get(0).getFirstName());  
 }  
}

Cum functioneaza:

Am creat o instanta de **Speaker Service** cu numele **service** care se duce 🡪 catre clasa **SpeakerServiceImpl**

**SpeakerServiceImpl** va face load la instanta noastra de **SpeakerRepository repository** (fake hibernate instance) care va accesa la randul ei clasa **HibernateSpeakerRepositoryImpl** unde am simulat accesarea unei baze de date si returnarea unei liste de nume.

Configurarea aceste Aplicatii utilizand Spring:



Getting spring:

Vom adauga dependente:



Descarca mai multe pentru ca la randul lui spring este dependent de alte lucruri (Transient dependency).

Spring configuration using Java: (Some XML Stuff too)

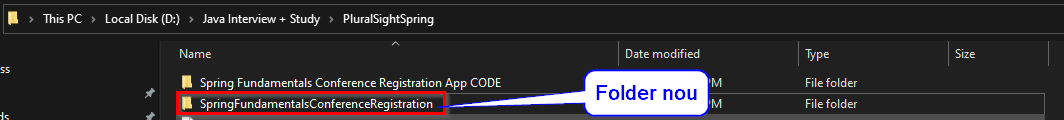
* Java annotations
* Java configuration loader

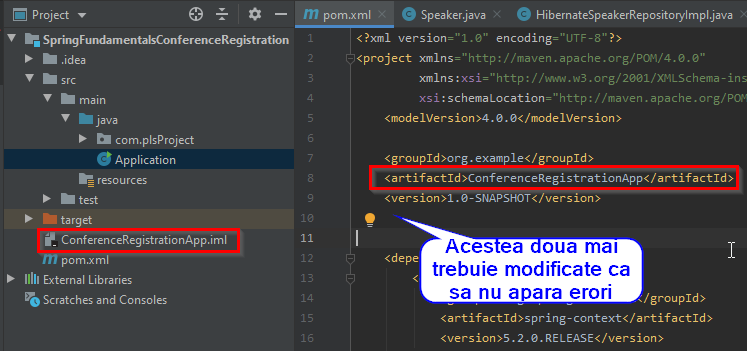
Cand avem Spring nu utilizam XML, No applicationContext.xml.

Vom copia proiectul original si paste in acelais folder

Am copiat folder-ul si l-am redenumit ca sa fie un proiect nou, sa vedem de unde am plecat si unde am ajuns:

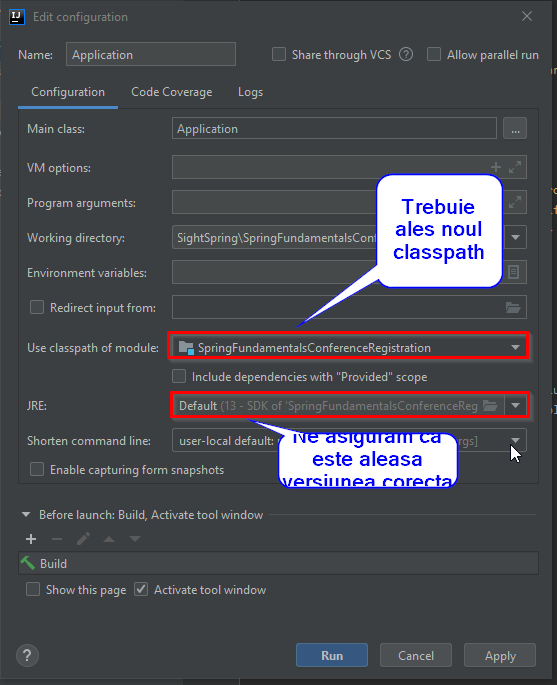
Acum am 2 foldere si eu voi continua sa lucrez in el nou:





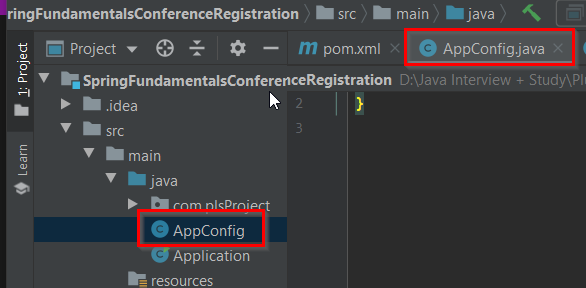
Ruland observam ca acum merge:





Pentru a incepe configurarea ne trebuie un fissier in care sa face bootstrap la toate.

Cream o clasa noua java:



Asta este tot ceea ce avem nevoie pentru a incepe configurarea (bootstraping-ul aplicatiei).

In Clasa AppConfig incepem configurarea Contextului Aplicatiei. (Aplication Context config)

Vom adauga adnotari aici:

**Adnotari**

Pentru a incepe configurarea vom incepe cu **@Configuration** careSe pune deasupra clasei

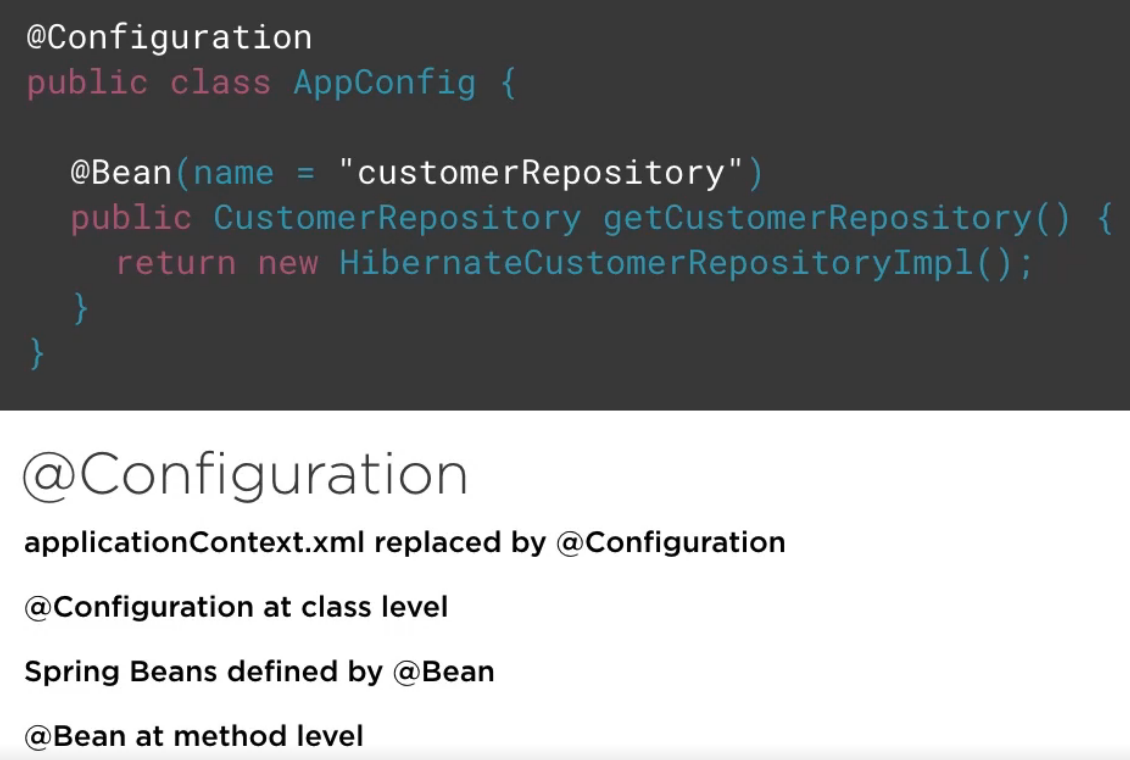
Fisierele java care contin **@Configuration** inlocuiesc orice fisier de tip xml pe care le-am fi putut folosi in trecut.

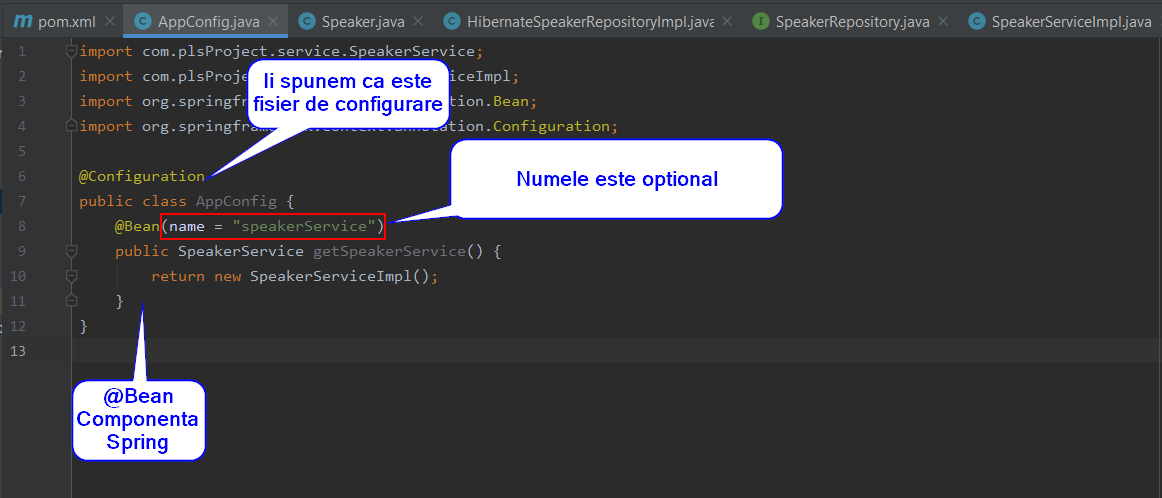
**!!! @Bean fucntioneaza doar la nivelul metodei nu si al clasei. LA nivelul clase se poate folosi @Component, care va crea un bean din instaantierea Clasei.**

Metodele utilizate impreuna cu adnotarea **@Bean** sunt utilizate pt a obtine/returna instante de **Spring Beans**

Aceste Spring Beans sunt acum inregistrate In Spring si valabile pentru a fi utilizate in interiorul aplicatiei.

Aceste metode nu trebuie sa aiba nume pentru a functiona , Tot ceea ce conteaza este sa aiba adnotarea @Beans si sa se afle intr-un fisier de configurate (AppConfig cu adnotarea @Configuration deasupra).





Ce am facut ?

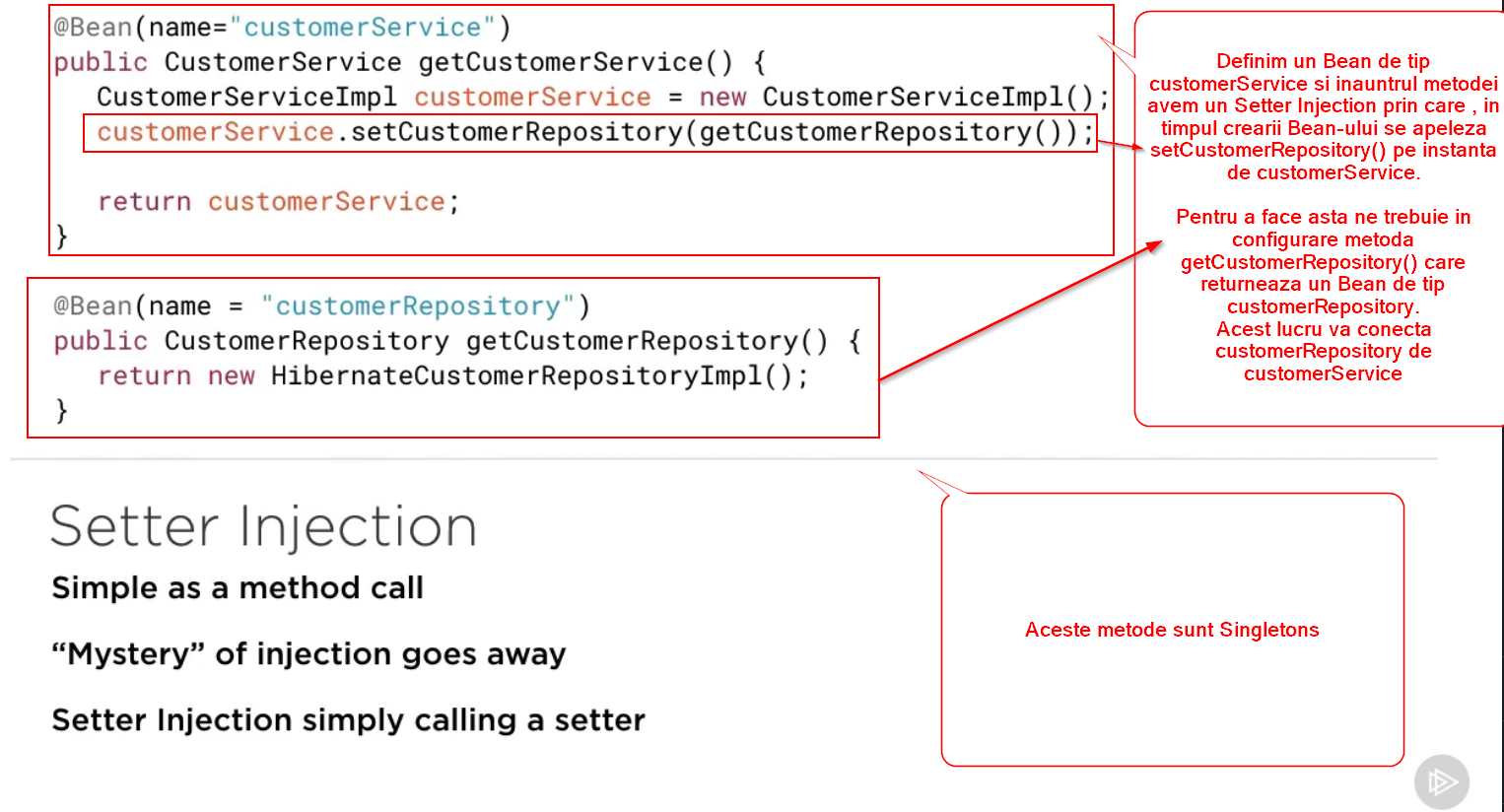
🡪We Spring Enabled out App

🡪 We told it to configure it as a Spring App

🡪 We created a Bean (speakerService)

**Setter Injection** (Is like a Simple Method call)

Este o cale de a chema un Setter pe un Bean.



Pentru a realiza asta trebuie sa facem cateva modificari in cod-ul normal.

Basically in setter-ul de la **speakerService** nu as putea sa apelez direct getCustomerRepository fara un **new** (o instanta a CustomerRepository) in aceeasi clasa pt CustomerRepository.

Facand-ule Beans in acelasi AppConfig, aplicatia mea stie ca la apelarea setCustomerRepository(); ca are nevoie de o instanta de new HibernateCustomerRepositoryImpl pe care sa apeleze getCustomerRepository();



Scapam de Instantele hard codate (**new**)

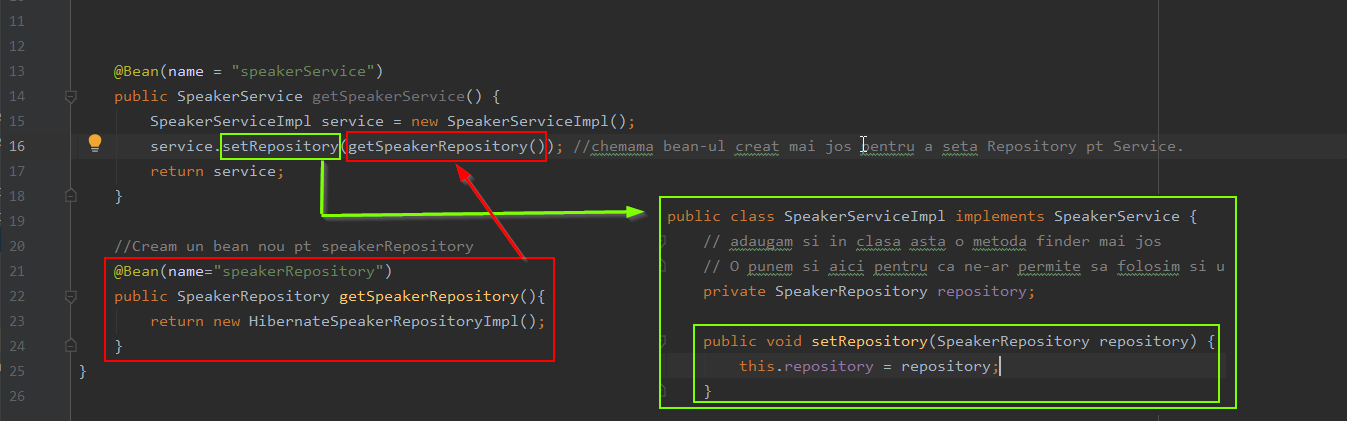
Cream un Setter pentru ca acesta ne va crea instanta cand avem nevoie de ea.

Acum ne intoarcem in App Config pentru a conecta metodele:

Cream un Bean nou pt speakerRepository

@Bean(name="speakerRepository")  
public SpeakerRepository getSpeakerRepository(){  
 return new HibernateSpeakerRepositoryImpl();  
}

Acum ca il avem creat, il putem folosit in celalata bean pentru crearea de Speaker Service:

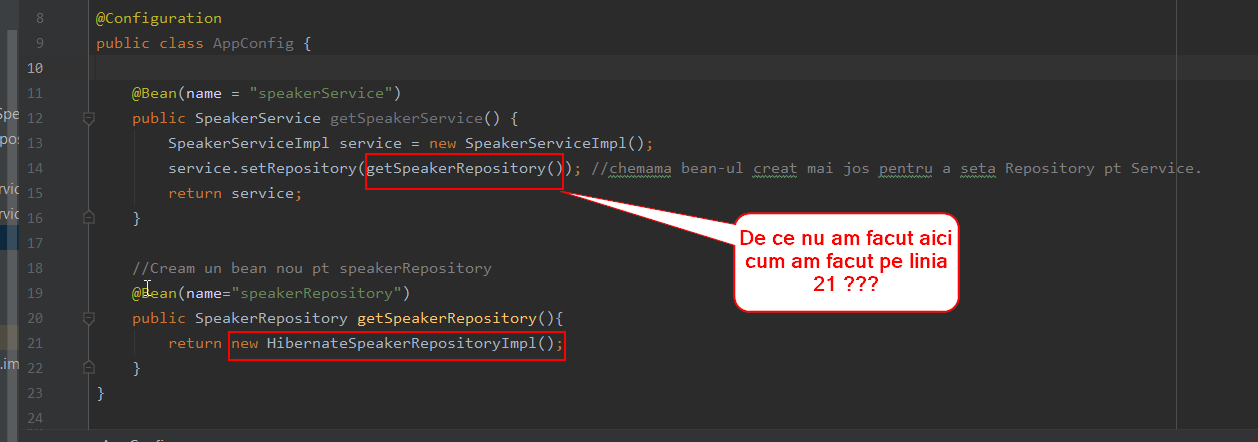


Metoda getSpeakerRepository necesita o instanta de **HibernateSpeakerRepositoryImpl()**

De aceea am creat Bean, **setRepository** este doar o metoda a **SpeakerServiceImpl**, deci nu este necesar un Bean

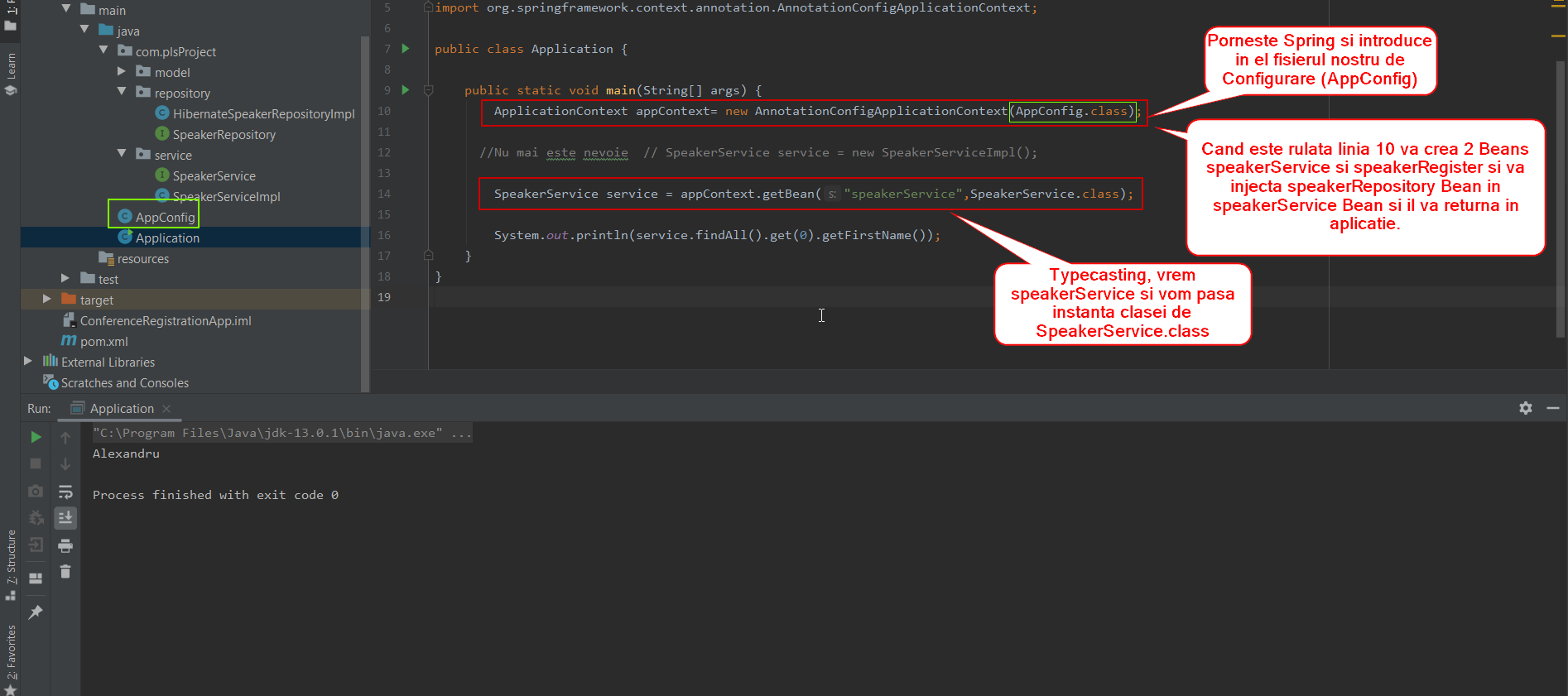
Astfel reusim sa injectam SpeakerRepository in Speaker Service utilizand Setter Injection.

**Intrebare? De ce nu am creat o instanta noua de HibernateSpeakerRepositoryImpl() cum am facut pe linia 21, pe linia 14 ?**



Deoarece speakerRepository este acum creat drept un **Bean,** Bean-ul este un Singleton si doar unul va fi creat, asta este un lucru bun.

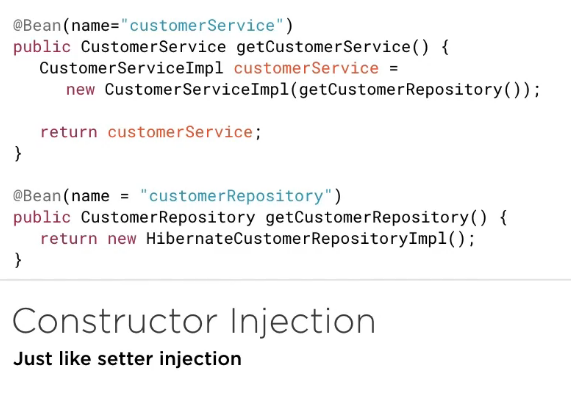
Trebuie sa merge in Application acum ca sa facem aplicatia sa foloseasca Spring si fisierul nostru de configurare.



🡪 Porneste Spring si introduce in el fisierul nostru de Configurare (AppConfig)

🡪 Cand este rulata linia 10 va crea 2 Beans speakerService si speakerRegister si va injecta speakerRepository Bean in speakerService Bean si il va returna in aplicatie.

**Constructor Injection (voi pune cod-ul original aici pentru a putea vedea diferentele)**



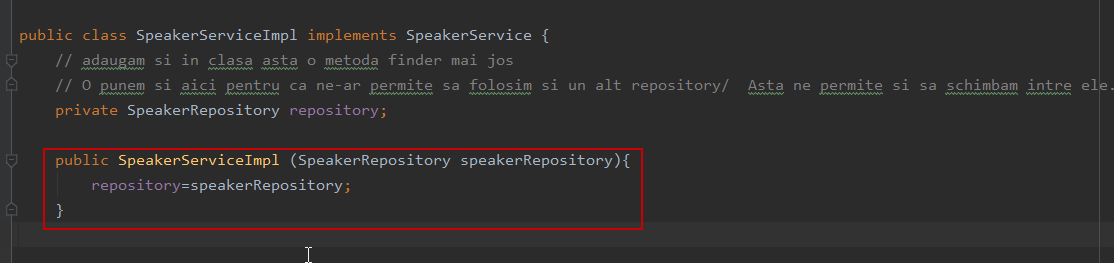
Cod Original:

public class SpeakerServiceImpl implements SpeakerService {  
  
 private SpeakerRepository repository;  
  
 public void setRepository(SpeakerRepository repository) {  
 this.repository = repository;  
 }  
  
 public List<Speaker> findAll() {  
 return repository.findAll();  
 }  
}

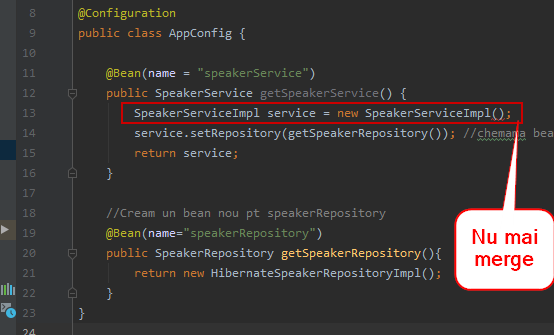
public class AppConfig {  
  
 @Bean(name = "speakerService")  
 public SpeakerService getSpeakerService() {  
 SpeakerServiceImpl service = new SpeakerServiceImpl();  
 service.setRepository(getSpeakerRepository());

return service;  
 }  
  
 @Bean(name="speakerRepository")  
 public SpeakerRepository getSpeakerRepository(){  
 return new HibernateSpeakerRepositoryImpl();  
 }  
}

1. Cream un Constructor: pt **SpeakerServiceImpl**



* Obs ca in fisierul de config nu mai functioneaza linia 13:



Pentru a repara cod-ul nu mai folosim:

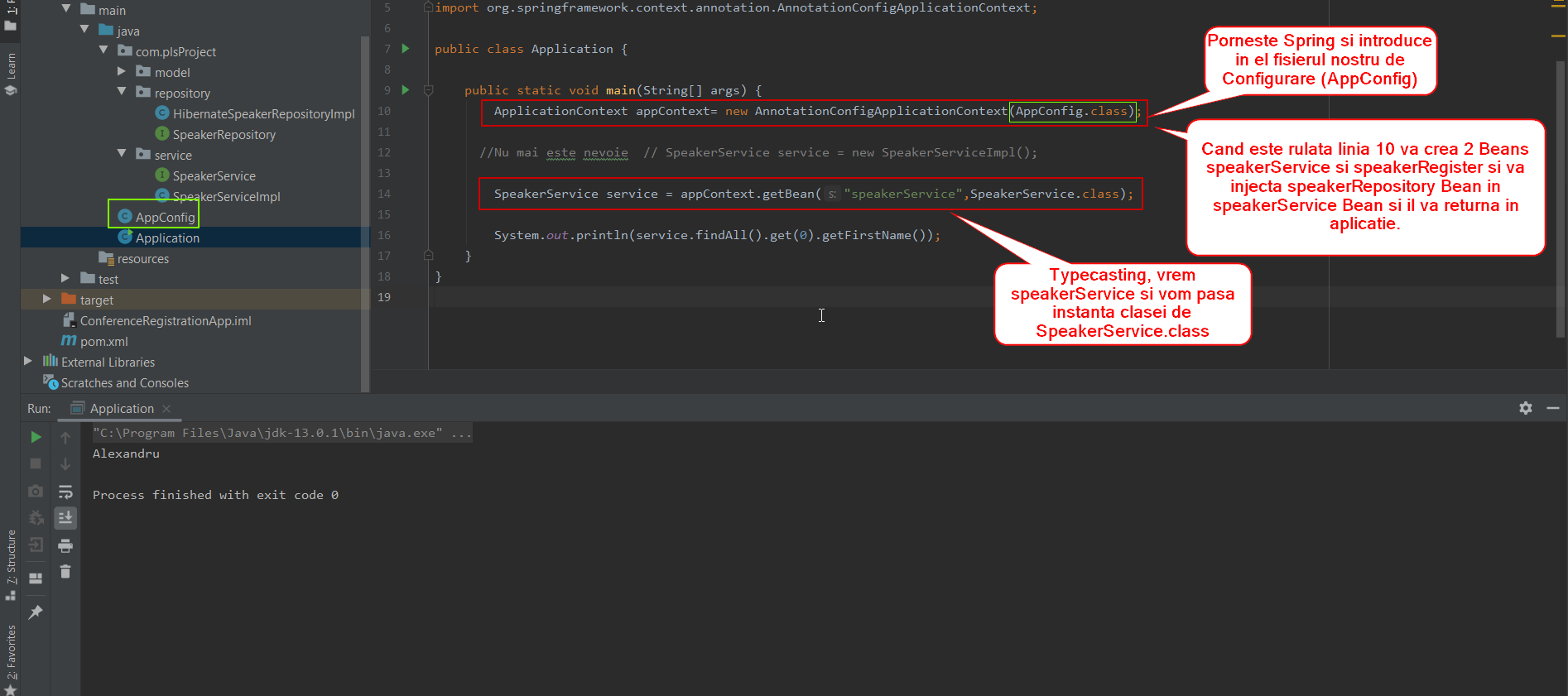
service.setRepository(getSpeakerRepository());

Ci vom folosi getSpeakerRepository()); direct in noul constructor:

SpeakerServiceImpl service = new SpeakerServiceImpl(getSpeakerRepository());



Este acelasi lucru ca la Setter Injection insa in loc sa folosim getSpeakerRepository() in Setter, il folosim in Constructor.



RECAP:

In loc de appContext putem folosi AppConfig file si il putem denota cu ajutorul @Config

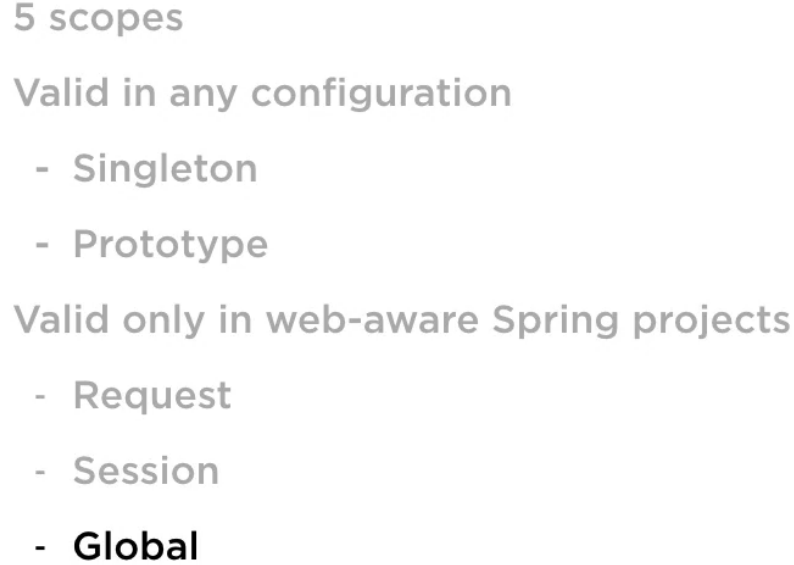
Pt a defini un Bean folosim @Bean

Setter and Constructor Injection.

**Autowiring + Component Scanning in Spring**

**(Should watch the design patterns course ( by Bryan Hansen))**

**Scopes**

****

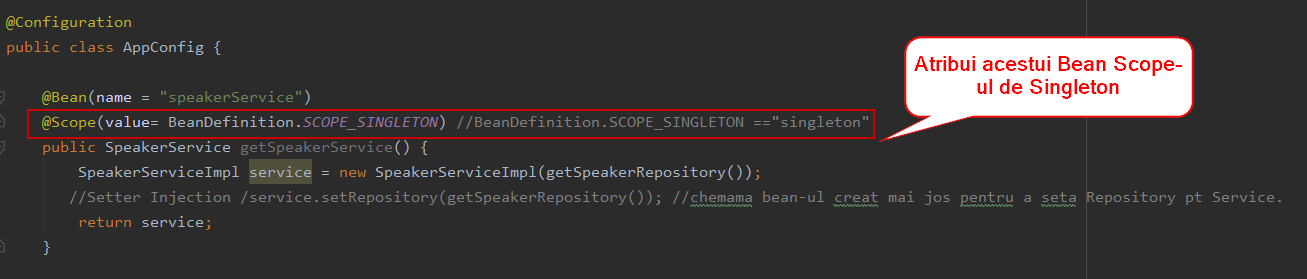
**Singleton**

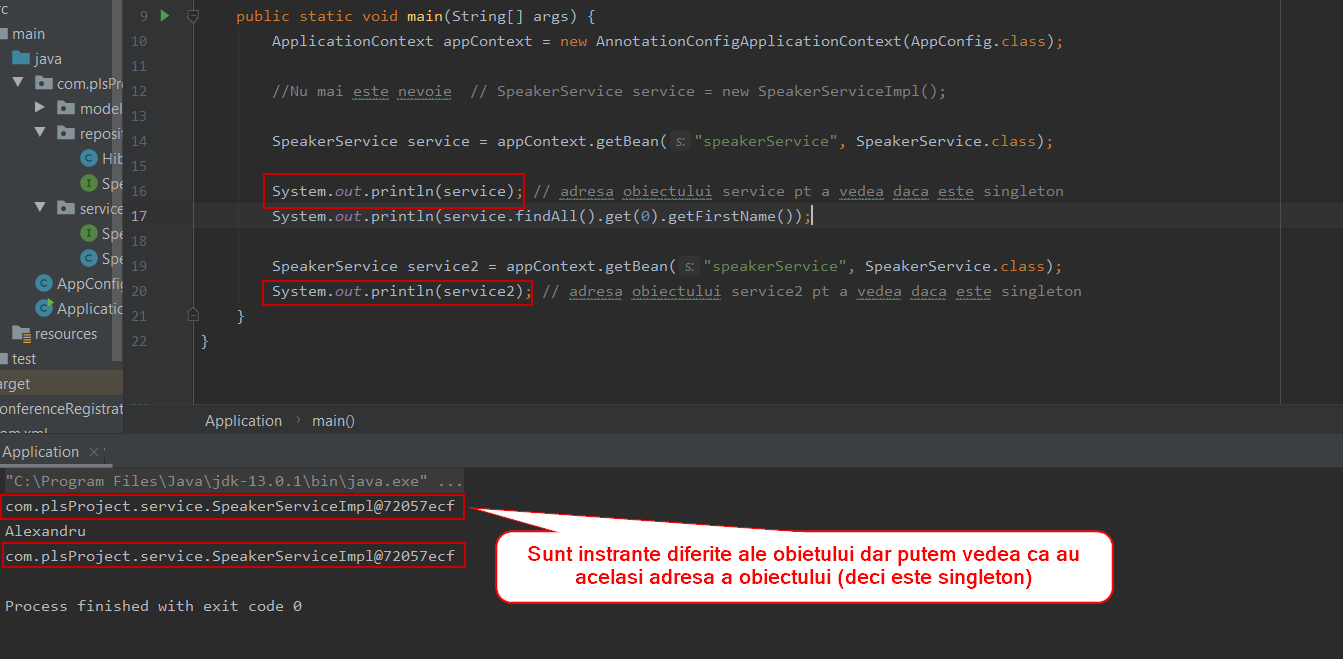
Singleton-ul restrange instaantierea unei clase la un singurobiect. Acesta este Default Scope in Spring (daca nu ii dau eu un scope acesta va fi setat default ca singleton)

Va fi o singura instanta pe Spring container / app Context.



**Pentru adaugarea unui Scope:**

****

****

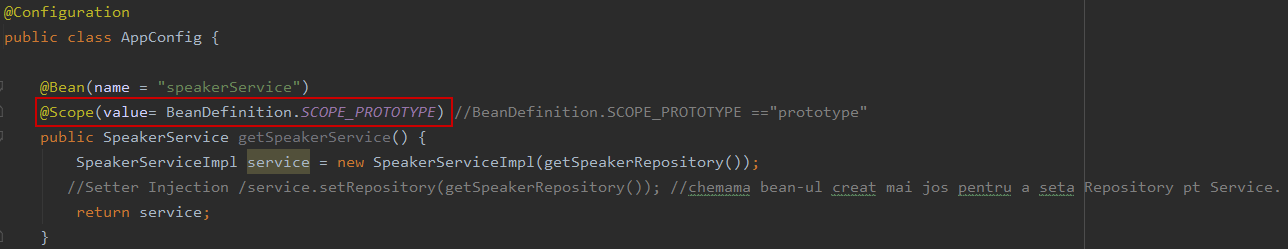
**Prototype Scope**

Prototype Design pattern gareanteaza o instanta unica pe Request asa ca Scope-ul Prototype in interiorul unui spring container va emula acest comportament. De fiecare data cand cerem un Bean din container va fi returnata o instanta unica. In esenta opusul unui Singleton.

Configurare:

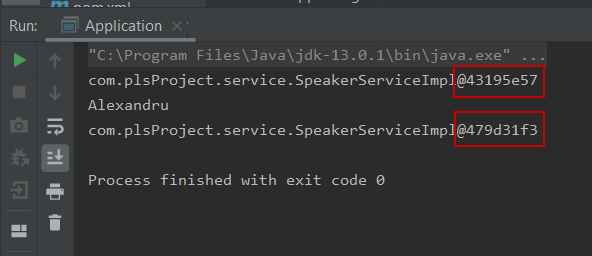


Vom folosi Bean-ul in aplicatia noastra:



Cand rulam aplicatia si printam obiectele: observam ca vom primi instante unice de fiecare data cand cerem un Bean din container (Object Adresses are different)

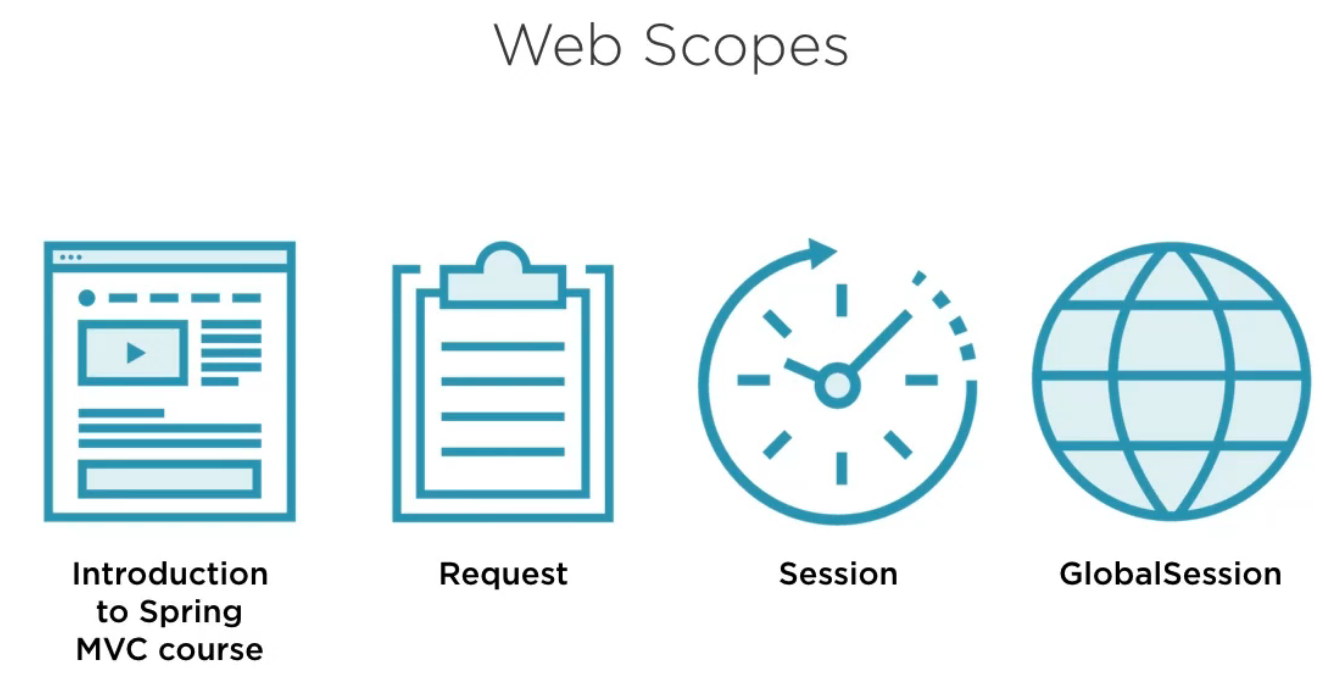
**“We get a unique Bean per Request” 🡪 de fiecare data cand cerem un Bean Nou din Container / App Context primim o instanta noua unica.**



* In proiect am modificat inapoi in **Singleton**

**Web Scopes**

Covered in (Introduction to Spring MVC course) 🡪 not really covered here:



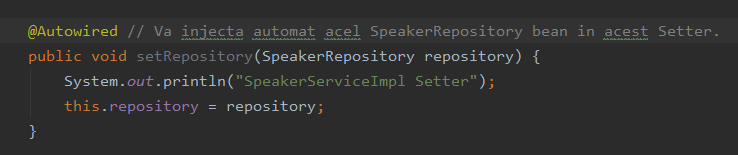
**Autowired**

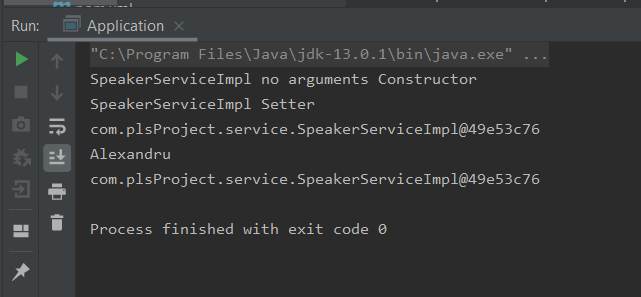
* Este o tehnica de a reduce legarea si configurarea codului.
* Pentru a folosi Autowire trebuie sa adaugam o **@ComponentScan({“”})**
*  Text-ul “com.pluralsight” mentioneaza unde ar trebui sa incep sa caut dupa adnotari @Autowire.
* Pentru a folosi autowire desemnam orice Bean vrem ca fiind Autowired si acest lucru poate fi facut dupa nume (@Bean) sau tip (Instance Type)
* Misto este ca pot avea un Bean drept autowired cu un bean inautrul ei si are mai mult sens.

**Utilizare Autowire:**



Pentru a utiliza Setter injection mergem in SpeakerServiceImpl si adnotam Setter-ul ca autowired:





Cand defineste un bean in app context el vede fiecare camp(obiect) din clasa mea care nu e instantiat, in momentul ala se uita la ce bean-uri are, din ficare bean care exista sanse sa poate instantia cu ele campuri din clasa mea (sa foloseasca Beans-urile in metode sau constructori). Cand vede Autowired el e gen “AHH pot sa folosesc asta sa instantiez Campul respectiv daca ia pasez unul dintre Beans-urile mele”.

Okay Deci ce se intampla, mai jos este cod-ul care interactioneaza:

@Configuration  
public class AppConfig {  
  
 @Bean(name = "speakerService")  
 @Scope(value= BeanDefinition.*SCOPE\_SINGLETON*) //BeanDefinition.SCOPE\_SINGLETON =="singleton"  
 public SpeakerService getSpeakerService() {  
 SpeakerServiceImpl service = new SpeakerServiceImpl(); //foloseste Constructor Default  
 return service;  
 }  
 //Cream un bean nou pt speakerRepository  
 @Bean(name="speakerRepository")  
 public SpeakerRepository getSpeakerRepository(){  
 return new HibernateSpeakerRepositoryImpl();  
 }  
}

public class SpeakerServiceImpl implements SpeakerService {  
 // adaugam si in clasa asta o metoda finder mai jos  
 // O punem si aici pentru ca ne-ar permite sa folosim si un alt repository/ Asta ne permite si sa schimbam intre ele.  
   
 private SpeakerRepository repository;  
   
 // Cream default Constructor  
 public SpeakerServiceImpl() {  
 System.*out*.println("SpeakerServiceImpl no arguments Constructor");  
 }  
  
 public SpeakerServiceImpl(SpeakerRepository speakerRepository) {  
 System.*out*.println("SpeakerServiceImpl repository Constructor");  
 repository = speakerRepository;  
 }  
  
 @Autowired // Va injecta automat acel SpeakerRepository bean in acest Setter.  
 public void setRepository(SpeakerRepository repository) {  
 System.*out*.println("SpeakerServiceImpl Setter");  
 this.repository = repository;  
 }  
  
 public List<Speaker> findAll() {  
 return repository.findAll();  
 }

public static void main(String[] args) {  
  
 ApplicationContext appContext = new AnnotationConfigApplicationContext(AppConfig.class);  
  
 //Nu mai este nevoie // SpeakerService service = new SpeakerServiceImpl();  
  
 SpeakerService service = appContext.getBean("speakerService", SpeakerService.class);  
  
 System.*out*.println(service); // adresa obiectului service pt a vedea daca este singleton  
 System.*out*.println(service.findAll().get(0).getFirstName());

}  
}

Output

SpeakerServiceImpl no arguments Constructor

SpeakerServiceImpl Setter

com.plsProject.service.SpeakerServiceImpl@49e53c76

Alexandru

**IMPORTANT:**

Cod-ul din appConfig Creaza un Bean din SpeakerServiceImpl **insa pentru a face asta are nevoie de o instanta de** private SpeakerRepository repository; Deoarece Setter-ul din clasa are adnotarea @Autowired el stie ca poate il poate instantia utilizand Setter-ul (Va injecta automat acel @Bean(name="speakerRepository")) in acest Setter putand astfel instantia campul repository.

Pana in momentul de fata am facut autowire la metode (Setter-ul). In continuare vom face Beans-urile Fully Autowired.

* Vom Discuta intai despre **Stereotype Annotations**:

**Stereotype Annotations**

**Adnotarile acestea nu modifica nimic este utilizat doar pentru a categorisi Beans-urile.**

@Component: Same this as @Bean.

@Repository: It’s used to denote a class that is used as a Repository Object (Se leaga la baza de date astea)

@Service (Nu inseamna Serviciu sau WebService) 🡪 Se refera la locul unde punem Business Logic-ul

@ Controler (out of the scope of this course aici cream WebServices).

1. In app config Adaugam ComponentScanner-ul: in AppConfig

@Configuration  
@ComponentScan ({"com.plsProject"})

In pranteze vom pasa un arraylist cu denumirile pachetelor unde trebuie sa se uite dupa componente (Beans / Component / Service / Repository (They are basically different names for the same thing)) care vor fi Autowired.

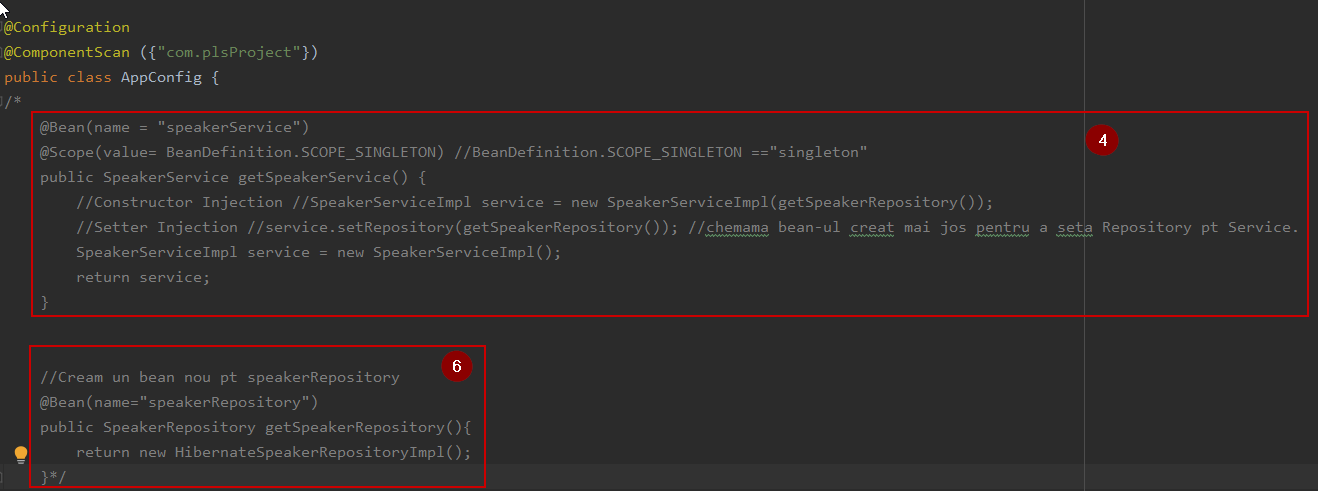
**!!! @Bean fucntioneaza doar la nivelul metodei nu si al clasei. La nivelul clase se poate folosi @Component, care va crea un bean din instaantierea Clasei.**

1. Definim SpeakerServiceImpl ca @Service( Ne va permite sa eliminiam @Bean-ul “speakerService” din AppConfig)
2. @Service("speakerService")  
   public class SpeakerServiceImpl implements SpeakerService {  
    // adaugam si in clasa asta o metoda finder mai jos  
    // O punem si aici pentru ca ne-ar permite sa folosim si un alt repository/ Asta ne permite si sa schimbam intre ele…

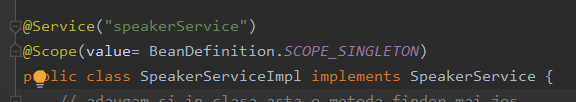
…

1. Eliminam Bean-ul “speakerService” din AppConfig:
2. Definim **HibernateSpeakerRepositoryImpl** ca @Repository( Ne va permite sa eliminiam @Bean-ul “speakerRepository” din AppConfig):

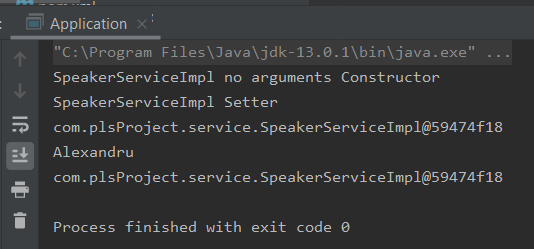
@Repository("speakerRepository") // acum acest bean va fi setat ca Repository utilizand Stereotype Annotation  
public class HibernateSpeakerRepositoryImpl implements SpeakerRepository { …

1. Eliminam Bean-ul “speakerRepository” din AppConfig:

OBS: Scope-ul poate fi definit in continuare la nivelul clasei:



Output:



Cateva lucruri care au loc in cod:

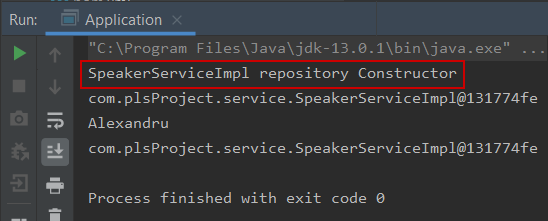
In momentul de fata cod-ul este Autowired pentru Setter Injection in continuare:

Il pot face sa foloseasca Constructor Injection doar prin mutarea adnotarii la Constructor.

@Service("speakerService")  
@Scope(value= BeanDefinition.*SCOPE\_SINGLETON*)  
public class SpeakerServiceImpl implements SpeakerService {  
 // adaugam si in clasa asta o metoda finder mai jos  
 // O punem si aici pentru ca ne-ar permite sa folosim si un alt repository/ Asta ne permite si sa schimbam intre ele.  
  
 private SpeakerRepository repository;  
  
 // Cream default Constructor  
 public SpeakerServiceImpl() {  
 System.*out*.println("SpeakerServiceImpl no arguments Constructor");  
 }

**@Autowired**  
 public SpeakerServiceImpl(SpeakerRepository speakerRepository) {  
 System.*out*.println("SpeakerServiceImpl repository Constructor");  
 repository = speakerRepository;  
 }  
  
 //@Autowired // Va injecta automat acel SpeakerRepository bean in acest Setter.  
 public void setRepository(SpeakerRepository repository) {  
 System.*out*.println("SpeakerServiceImpl Setter");  
 this.repository = repository;  
 }  
  
 public List<Speaker> findAll() {  
 return repository.findAll();  
 }  
  
 //in final am creat si o interfata:  
}

OUTPUT:



**Spring Configuration using XML**

ApplicationContext.xml is the root of the application.

By default, Spring will look for this one.

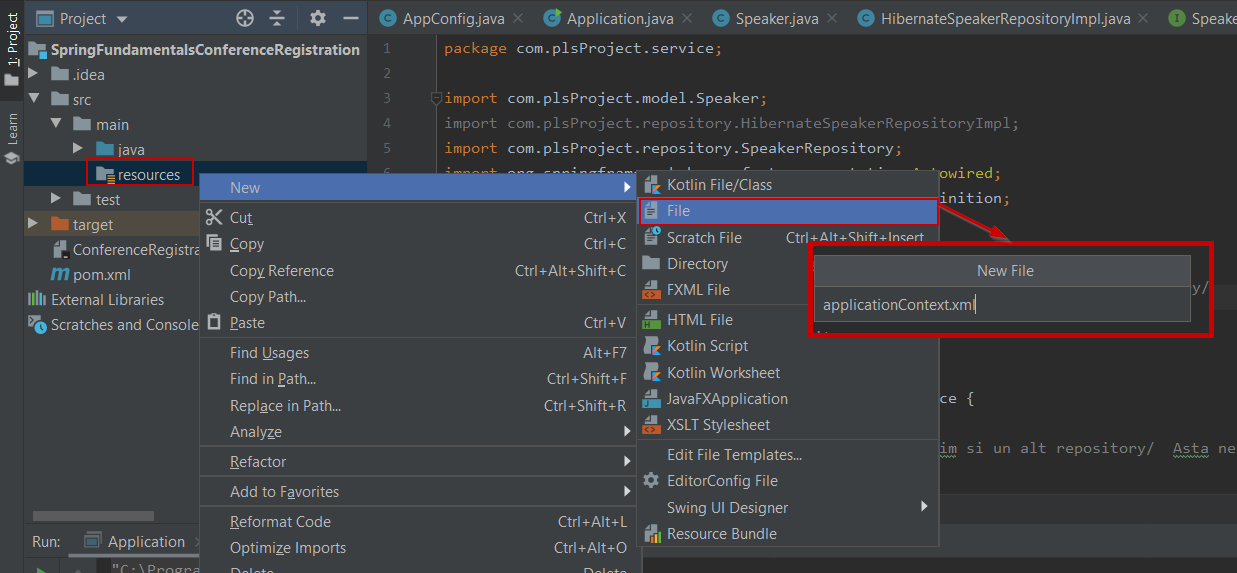
Spring is basically a glorified HashMap for objects that we define in the ApplicationContext.xml

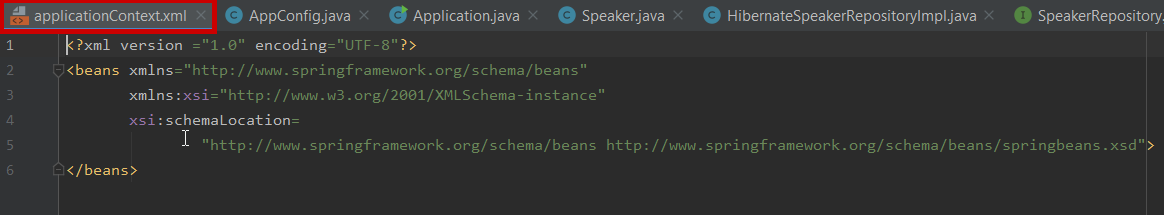
Or AppConfig.java

We have Namespaces that aid in configuration / validation.

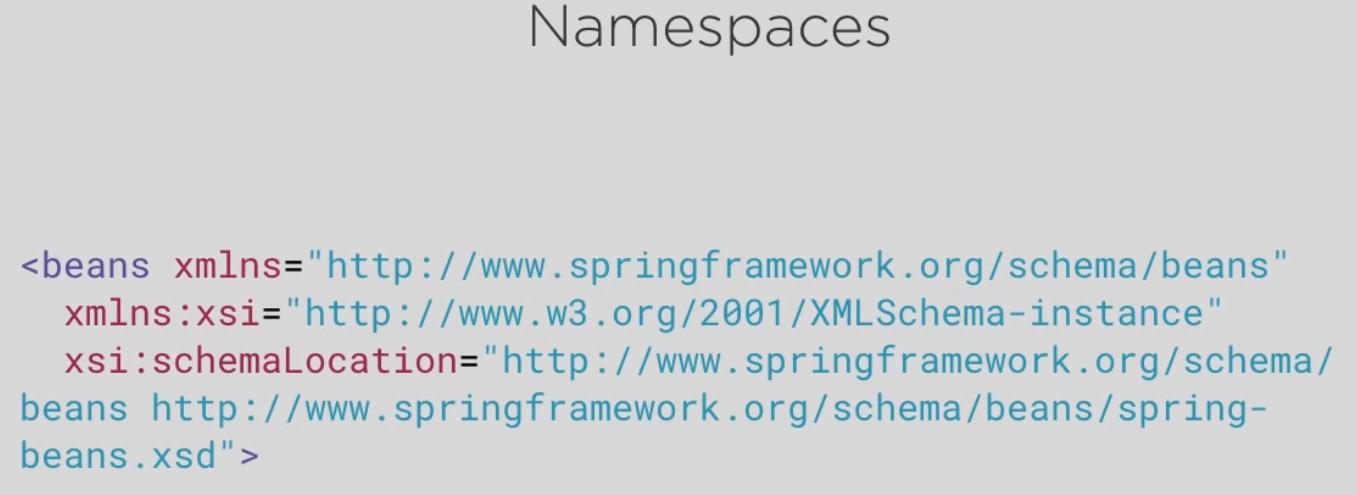
Copiem proiectul original fara SpringWiring si il modificam pe acela:

**Adding the applicationConfig.XML:**

****



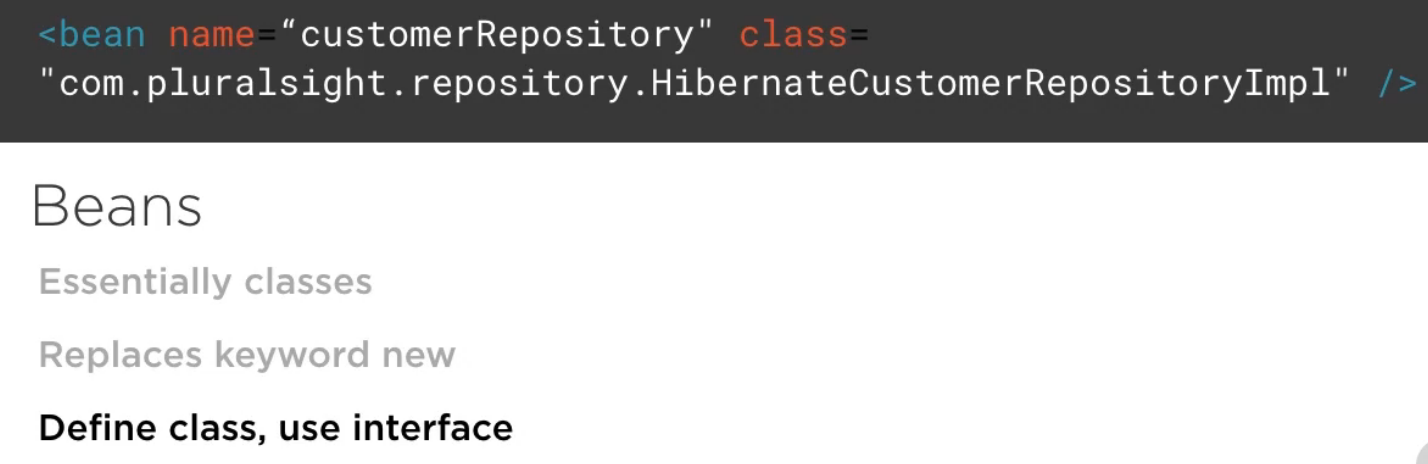
<?xml version ="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation=  
 "http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/springbeans.xsd">  
</beans>



Namespace-ul reprezinta un dictionar pt proprietatile cu care putem crea un Bean in interiorul aplicatiei.



XML Declaration ne permite sa definim un Bean in XML. Acest Bean se numeste : CustomerServiceImpl Bean si reprezinta locatia unde vrem sa punem Buisness Logic-ul din aplicatie.



Beans-urile sunt in esenta doar Clase, configuratia XML este compusa din Beans si sunt doar Pojos care sunt folosite in appContext.

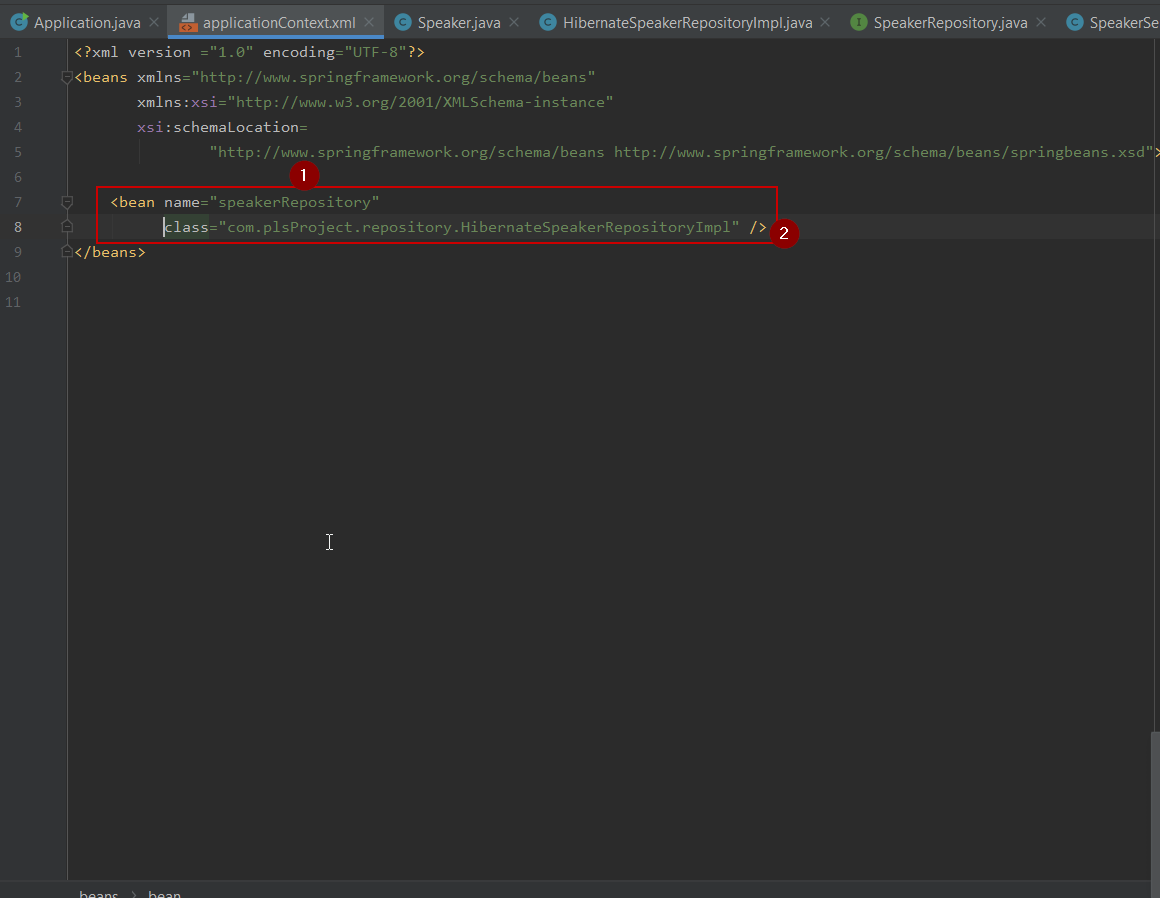
Definirea De Beans poate fi gandita ca inlocuirea de cuvant Cheie “new”

Ca regula generala: Oriunde folosim cuvantul cheie “new” acel lucru ar trebui sa fie pus in Java/xml config file.

Ca regula generala: Vrem sa definim clasa dar sa utilizam interfata.

Adaugarea unui Bean in interiorul AppContext.xml

**Introducerea unui Bean in applicationContext.xml:**



<bean name="speakerRepository"  
 class="com.plsProject.repository.HibernateSpeakerRepositoryImpl" />

Creez bean-ul speakerRepository ca fiind clasa com.plsProject.repository.HibernateSpeakerRepositoryImpl

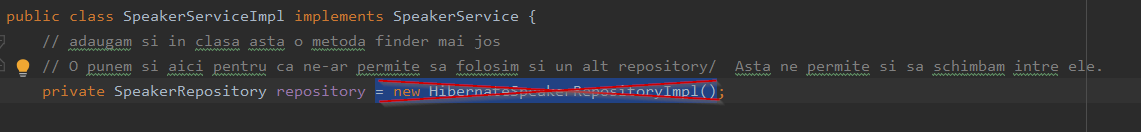
1. Adaugam numele bean-ului **speakerRepository**
2. Vrem sa alegem locatia clasei pentru care vrem instanta **com.plsProject.repository.**

In plus vrem o instanta a **HibernateSpeakerRepositoryImpl,** noi in cod o vom referi prin instanta insa bean-ul il vom crea cu implementarea.

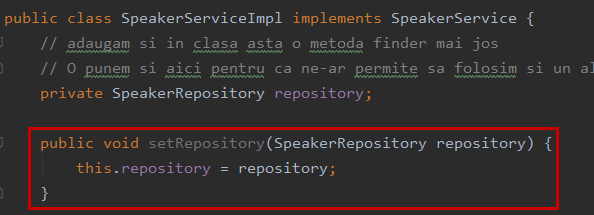
Nu injectam nimic in acest bean pentru ca nu are alte dependente. Esta doar o metoda care returneaza nume dintr-o baza de date creata la misto.

Pentru a seta app pt a putea face setter injection va trebui sa definim un nou Bean.

1. Vom sterge din **SpeakerServiceImp** instanta creata de **HibernateSpeakerRepositoryImpl**

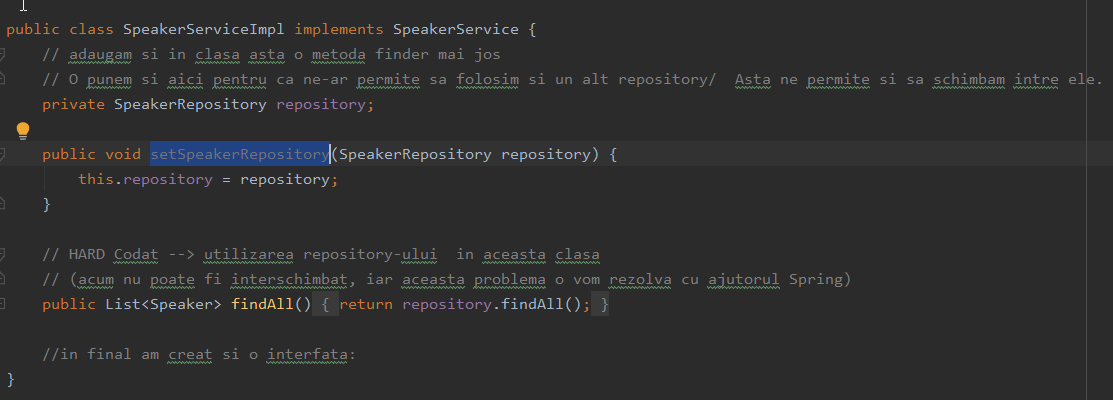


1. Vom genera un Setter: (pt setter injection)



1. pt a seta Setter Injection-ul:

3.1 modificarile facute la 2 plus schimbarea numelui de la setter pe care o vom utiliza mai tarziu la autowire:



3.2 Cream Bean pentru “speakerService”:

<bean name="speakerService" class="com.plsProject.service.SpeakerServiceImpl" >  
 <property name="speakerRepository" ref="speakerRepository"/>  
</bean>

**Bean-ul** speakerService are nevoie de bean-ul speakerRepository

**“<property../>”** = campurile din clasa **com.plsProject.service.SpeakerServiceImpl** (din care fac Bean) care necesita a fi ocupate de alte bean-uri carora le pun nume si ofer o referinta catre bean-ul care le va ocupa.

**“Name=”** numele pe care le voi da bean-urilor care vor ocupa acele campuri (obiecte)

“**Ref=”** referinta catre bean-ul de care am nevoie pentru a ocupa acele campuri.

Spre exemplu daca am am mai multe campurin in **class="com.plsProject.service.SpeakerServiceImpl" > care necesita a instantierea a mai multor alte clase. (adica sunt ocupat de alte bean-uri).**

**In plus, el isi da seama dupa nume** speakerRepository (set…-,,-) ca va putea folosi settar-ul pt a instantia SpeakerRepository

public class SpeakerServiceImpl implements SpeakerService {  
 // adaugam si in clasa asta o metoda finder mai jos  
 // O punem si aici pentru ca ne-ar permite sa folosim si un alt repository/ Asta ne permite si sa schimbam intre ele.  
 private SpeakerRepository repository;  
 private SpeakerRepository repository1;  
 private SpeakerRepository repository2;  
 private SpeakerRepository repository3;  
  
 public void setSpeakerRepository(SpeakerRepository repository) {  
 this.repository = repository;  
 }

<bean name="speakerRepository"  
 class="com.plsProject.repository.HibernateSpeakerRepositoryImpl" />  
  
<bean name="speakerService" class="com.plsProject.service.SpeakerServiceImpl" >  
 <property name="speakerRepository" ref="speakerRepository"/>  
 <property name="speakerRepository1" ref="speakerRepository"/>  
 <property name="speakerRepository2" ref="speakerRepository"/>  
 <property name="speakerRepository3" ref="speakerRepository"/>  
</bean>

**La crearea ban-ului** speakerService va crea alte 4 bean-uri cu numele **speakerRepository 1 2 3**, prin ajutorul referintei **ref="speakerRepository”** care idica faptul ca ocuparea lor se face prin Bean-ul **speakerRepository**

* **Cream o proprietate a Bean-ului:**
* **Numele:** speakerRepository
* **Referinta: (referim catre alt Bean) catre** speakerRepository

**Acum bean-urile noi sunt definite:**

**Ref luam bean-ul** speakerRepository si il injectam in bean-ul speakerService

Aceasta referinta va chema setter-ul din SpeakerServiceImpl (de aceea am si modificat numele Setter-ului), daca settar-ul ar fi ramas “setRepository” atunci numele proprietatii ar fi trbuit sa fie “repository”.

Modificari necesare pentru app java:

1. Trebuie creata o instanta de Application Context:

ApplicationContext appContext = new ClassPathXmlApplicationContext("applicationContext.xml");

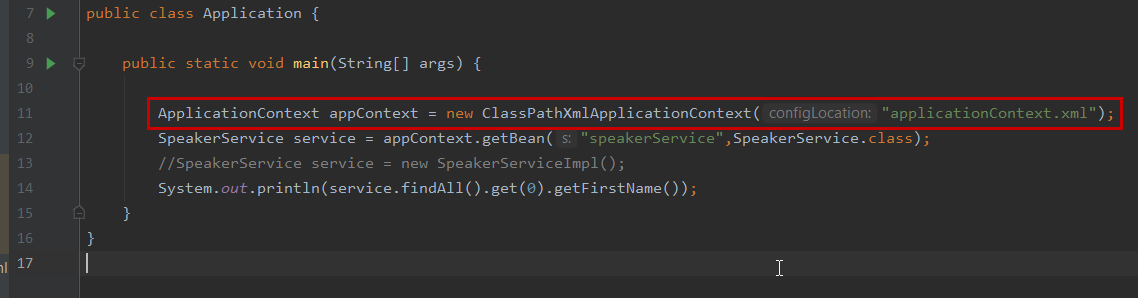
SpeakerService service = appContext.getBean("speakerService",SpeakerService.class);

OBS:

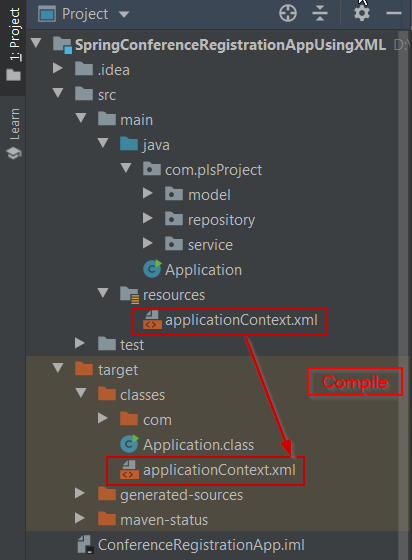
appContext.getBean("speakerService",SpeakerService.class); poate fi folosit si in alte clase in cazul in care vreau sa folosesc bean-uri acolo.

Ce se intampla dupa rulare:

Am facut Bootstrap la aplicatie pe linia 11 facand load la fisierul de configurare.xml



Deoarece applicationContext.xml este creat in interiorul folderului src🡪 resources:

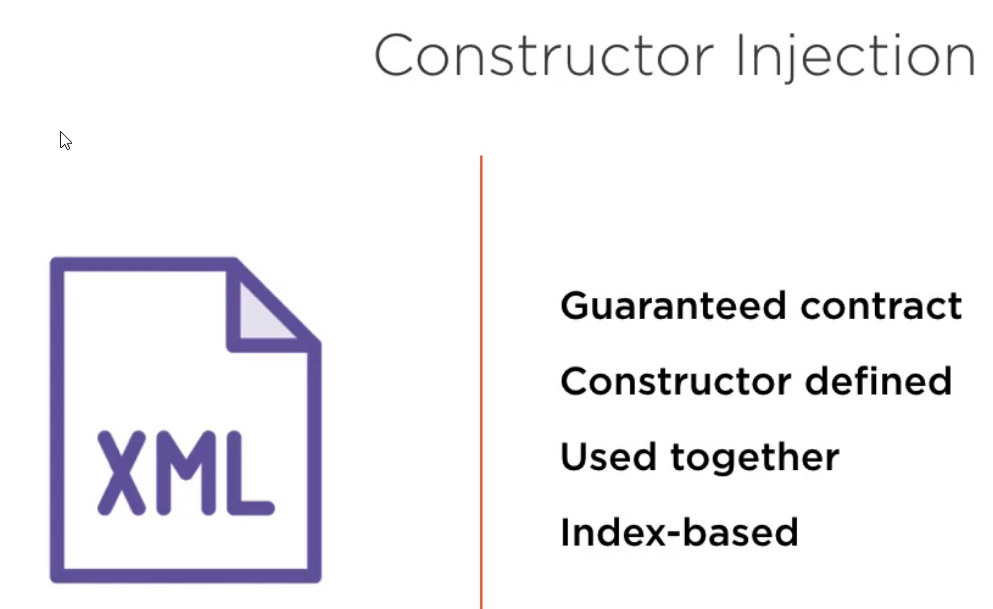


Maven la compilare, il va compila catre directory-ul claselor: asa ca la rulare il va copia catre target🡪 classes directory, loc de unde ruleaza class path-ul.

Sunt create bean-urile iar in bean-ul speakerService am dat o referinta catre speakerRepository care realizeaza setter injection.

De asemenea pot fi utilizate si setter injection si constructor injection.

**Constructor Injection:**

****

Using the defined constructor.

De asemenea pot fi utilizate si setter injection si constructor injection.

Obs pt cod deja enxistent este mai bun setter injection.

**Constructor Injection**

🡪 garanteaza un contract:

* Avem cod-ul si contractul definit cand cream un obiect datorita constructorului.

Un pozitiv si negativ este faptul ca trebuie sa am definit cate un constructor pentru fiecare scenariu pe care vreau sa il garantez.

🡪 Poate fi folosit impreuna cu setter injection

🡪 Constructor injection is index based and not name based.

**Modificarea aplicatiei curente pentru a utiliza constructor injection:**

Cod vechi:

<bean name="speakerRepository"  
 class="com.plsProject.repository.HibernateSpeakerRepositoryImpl" />  
  
<bean name="speakerService" class="com.plsProject.service.SpeakerServiceImpl" >  
 <property name="speakerRepository" ref="speakerRepository"/>  
  
</bean>

Cod nou:

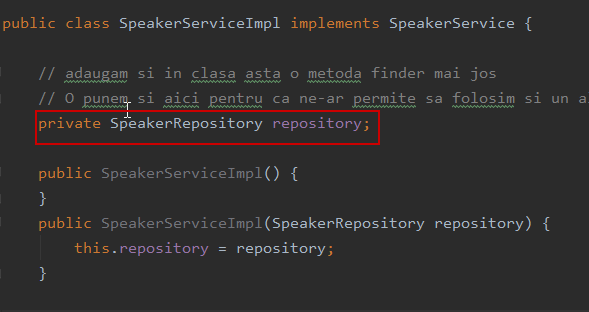
<bean name="speakerRepository"  
 class="com.plsProject.repository.HibernateSpeakerRepositoryImpl" />  
  
<bean name="speakerService" class="com.plsProject.service.SpeakerServiceImpl" >  
 <constructor-arg index="0" ref="speakerRepository"/>

1. **Modific appContext.xml**

In loc de **property** 🡪 **constructor-arg**

Costructor arguments sunt bazati pe index (ordinelor) deci din **name**🡪 **index**

**Index reprezinta numarul de argumente necesar clasei pentru a fi instantiata in cazul de fata argumentul reprezinta instantierea de speakerRepository: deci doar 0, un element**

****

1. **Creez constructori in** SpeakerServiceImpl

public SpeakerServiceImpl() {  
  
}  
  
public SpeakerServiceImpl(SpeakerRepository repository) {  
 this.repository = repository;  
}

**Autowire: definitii / info / explicatii:**

* La inceput spring avea o reputatie proasta datorata faptului ca era necesar foarte multa configurare XML
* Pentru a reolva asta au introdus un mecanism numit Autowire pentru a putea conecta beans-uri impreuna mai usor.

Exista 4 tipuri de Autowire:

* **byType**: permite unei proprietati sa fie autowired automat daca exact 1 bean al acelei tip de proprietate exista in container. Sa spunem ca am un obiect masina pe care il injectam, daca exista un singur obiect masina de acel tip, de acea clasa il putem injecta. Daca am doua obiecte masina cu nume difrite voi primi un fatal Exception pentru ca nu poate alege dintre cele doua pentru ca se uita la tip nu dupa nume.
* **byName**: rezolva problema de la byType, ne permite realizarea conexiunii dupa nume specific.
* **By constructor**: idesntic ca byType doar ca se aplica pantru argumente de tip constructor , are totusi putina aroma de byName pentru ca argumentele pot fi numite si indexate si introduse in obiectul nostru astfel. Trebuei sa ai un constructor potrivit pentru argumentlee pe care vrei sa le introduci.
* **No**: nu poate fi autoWired.

**Modificarea aplicatiei pentru a fi autoWired by constructor:**

**Cod original:**

<bean name="speakerService" class="com.plsProject.service.SpeakerServiceImpl" >  
 <constructor-arg index="0" ref="speakerRepository"/> 🡪 il vom comentaa  
  
</bean>

**Autowired for constructor: just specify to use constructor:**

<bean name="speakerService" class="com.plsProject.service.SpeakerServiceImpl" **autowire="constructor"** >  
 <!--<constructor-arg index="0" ref="speakerRepository"/>-->  
  
</bean>

Pentru a utiliza setter:

<bean name="speakerService" class="com.plsProject.service.SpeakerServiceImpl" autowire="byType" >  
 <!--<constructor-arg index="0" ref="speakerRepository"/>-->  
  
</bean>

Pentru a utiliza nume(tot setter dar o face dupa nume):

<bean name="speakerService" class="com.plsProject.service.SpeakerServiceImpl" autowire="byName" >  
 <!--<constructor-arg index="0" ref="speakerRepository"/>-->  
  
</bean>

Merge pentru ca setter-ul se numeste: **setSpeakerRepository** si bean-ul de care are nevoie se numeste **speakerRepository** si spring isi da seama ca setter-ul acela este utilizat pt a seta acest Bean.

Advanced Bean configuration. Corner Cases:

BeanPostProcessor (Init Methods)

Factory Bean (create a Bean that has Static methods in it )

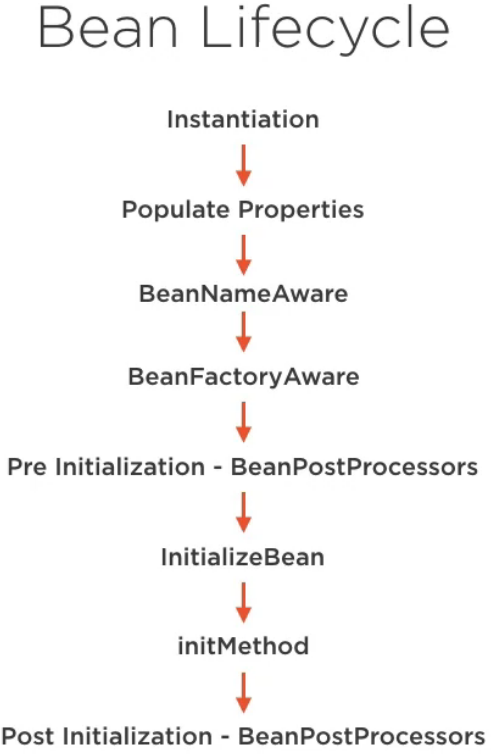
SpEL Spring Expresion Language

Proxies

Profiles



Bean LifeCycle



Incepe cu **Instantierea** apoi se realizeaza **Popularea Proprietatilor** aceste propriatati sunt citite din fisiere cu proprietati sau sunt injectate din alte resurse. Dupa aceasta Frameework-ul seteaza BeanName-ul si il face vizibil catre alte resurse (**BeanNameAware**) il putem seta in acest stagiu intr-un context **BeanFactoryAware.** Dupa realizam Pre-initializarea utilizand **BeanPostProcessors** (**Pre Initialization - BeanPostProcessors**). De aici putem initializa Bean-ul utilizand proprietati care tocmai au fost setate **InitializeBean.** Acum putem chema o metoda init **initMethod**. Iar apoi se incheie intreg-ul proces de initiaalizare cu inca un set de

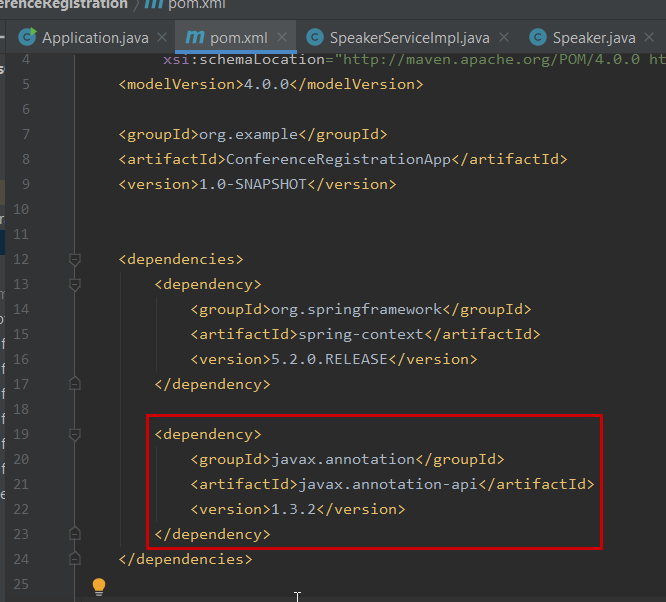
**Post Initialization - BeanPostProcessors**

**Configurarea unei metode Init pe un BEAN.**

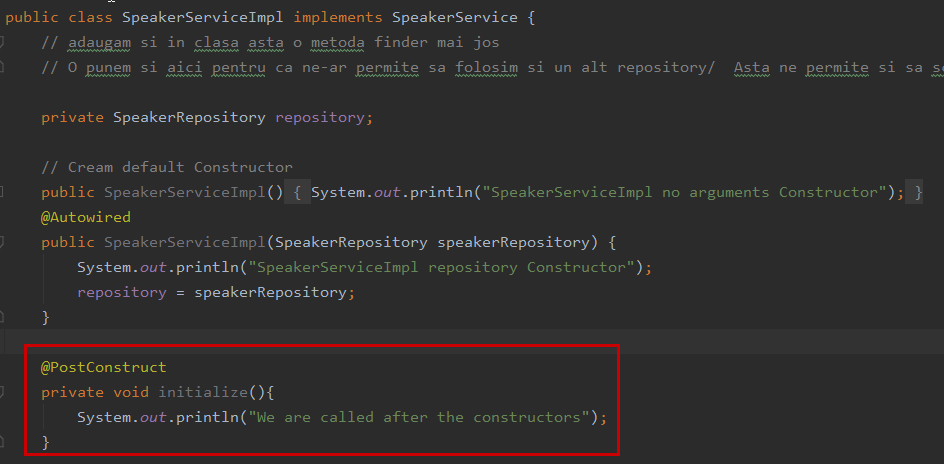
**Adaugarea unei metode init:**

In fisierul Pom.xml se adauga codul:

<dependency>  
 <groupId>javax.annotation</groupId>  
 <artifactId>javax.annotation-api</artifactId>  
 <version>1.3.2</version>  
</dependency>

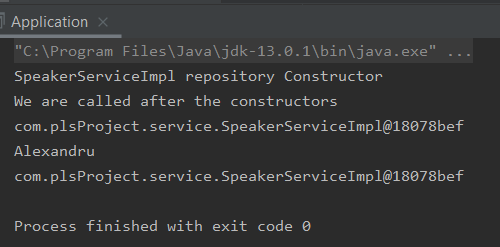


Vom naviga catre **SpeakerServiceImpl (POT realiza asta in orice Bean configurat de Spring)**

* Sub Constructori: (nu conteaza ce constructor este utilizat pentru ca metoda va fi utilizata dupa chemarea constructorilor):
* @PostConstruct  
  private void initialize(){  
   System.*out*.println("We are called after the constructors");  
  }
* 

Deci am creat o metoda init si am stampilat-o cu adnotarea @PostConstruct.

Odata ce rulam aplicatia observam ca a chemat Constructorul si apoi a chemat metoda init inainte sa ruleze orice altceva.

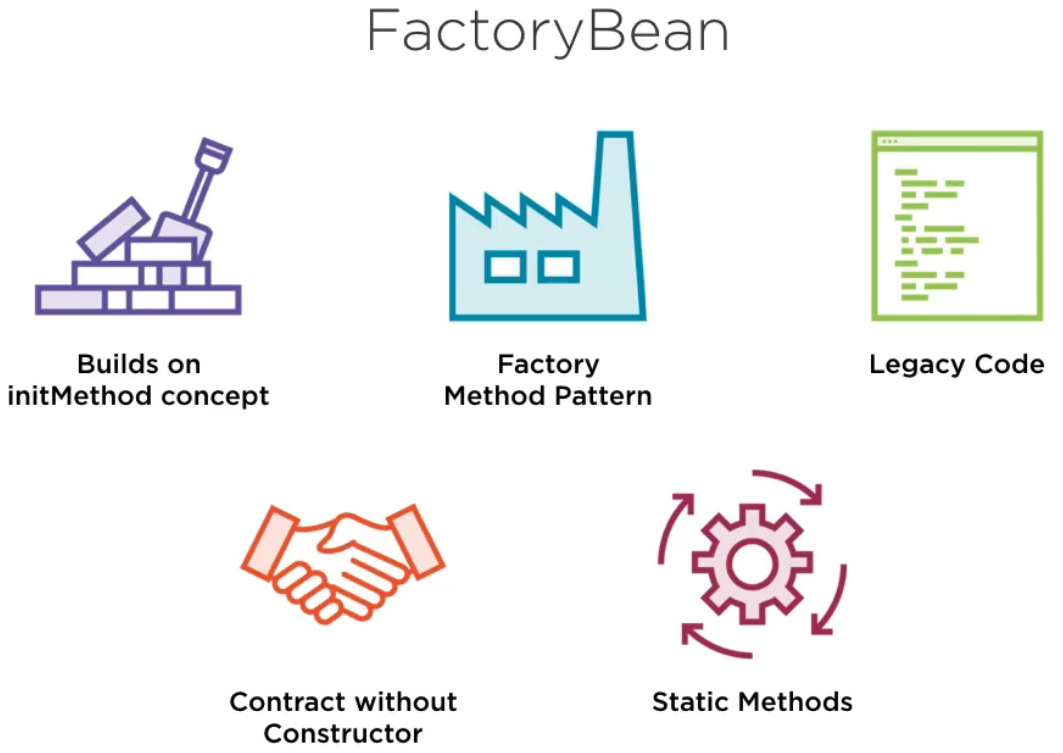


Este misto pt ca putem avea o metoda care ruleaza mereu dupa constructor, fie pt logging purposes, su configurari.

**OBS IMP:** NU este Bine sa introducem obtinerea de conexiuni la baza de date in metode de tip INIT, (Inchiderea si deschiderea de conexiuni la baza de date ar trebui sa fie facut de Spring si nu prin programare).

Init Methods are trebui sa ocupe de configurare suplimentara dupa deschiderea constructorilor.

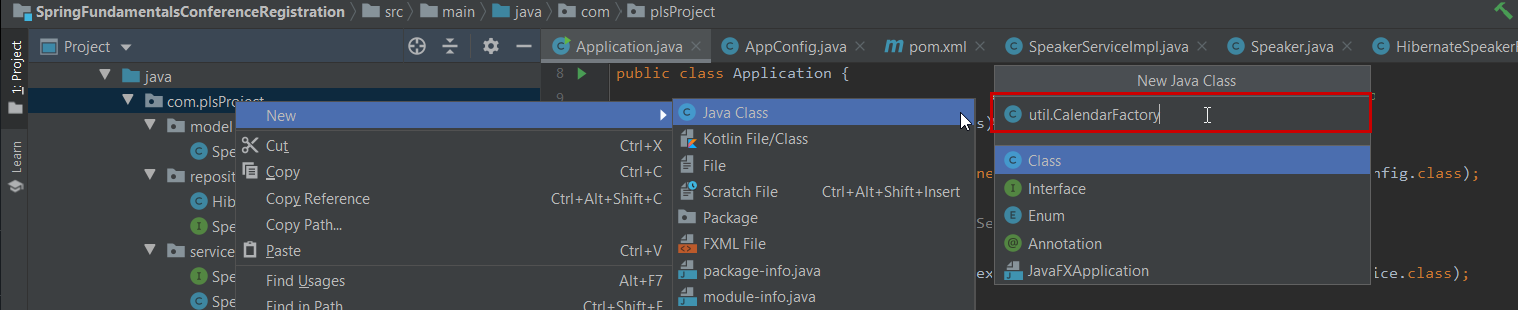
**Bean Factory 🡪 Configurare asemenatoare cu InitMethod**



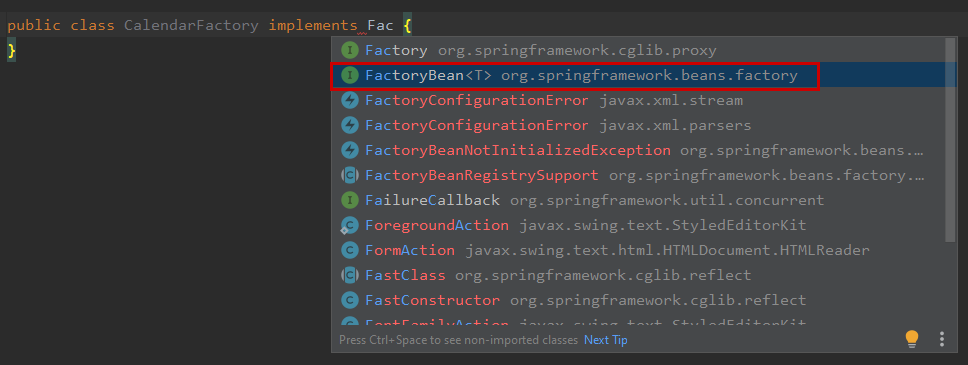
* Creaza peste Factory Method Design Pattern, si functioneaza corespunzator Design-ului.
* Cod-ul nu trebuie sa fie scris in asa fel incat sa respecte Factory Design Pattern pentru a fi utilizat in spring drept un Factory.
* Este o metoda buna de a implementa Legacy Code 🡪 Poti crea un contract despre cum este configurat codul fara a construi un constructor nou pentru cod. Tipic **Creational contracts** sunt implementate in construcotr insa in legacy code nu avem abilitatea de a ii modifica din cazua comitment-urilor deja existente. Utilizand acest Pattern ne permite sa cream un contract fara a modifica cod-ul.
* Permite utilizarea de metode statice intr-o clasa.

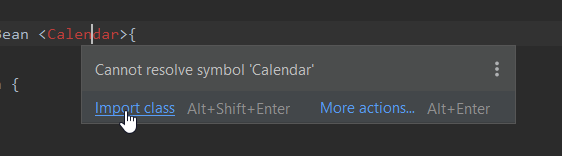
Pentru Demo-ul nostru vom face Wrap undei instante de calendar intr-un Factory pentru a putea face niste smacherii.

Cream un pachet cu o clasa nou in **com.plsProject 🡪 util.CalendarFactory**

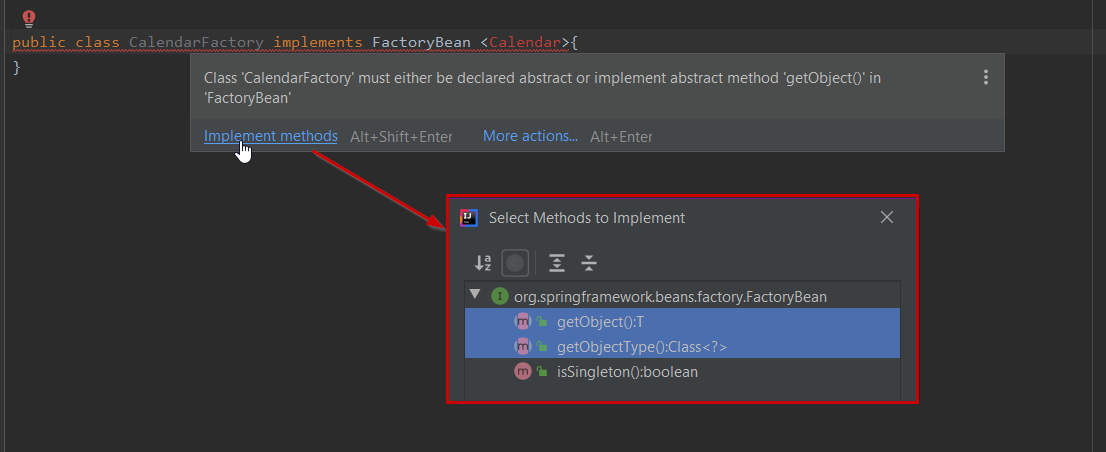


Implementam **FactoryBean** din spring: care are un tip, si vrem sa ii pasam un obiect de tip **Calendar**



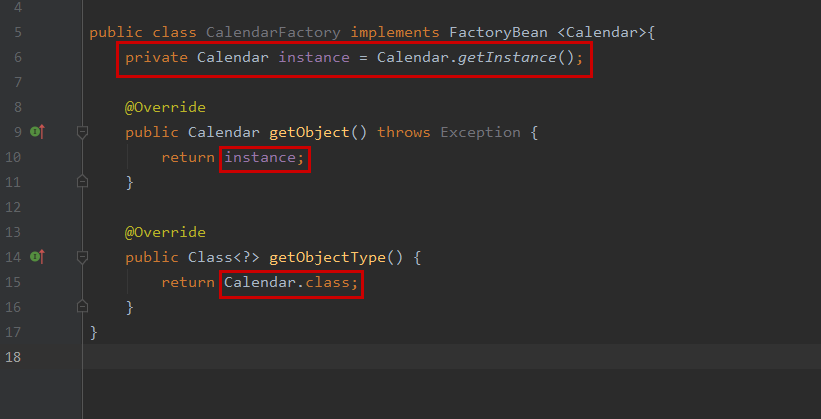


* Observam ca are nevoie de implemtarea metodelor din Interfata **FactoryBean:**



package com.plsProject.util;  
import org.springframework.beans.factory.FactoryBean;  
import java.util.Calendar;  
  
public class CalendarFactory implements FactoryBean <Calendar>{  
 @Override  
 public Calendar getObject() throws Exception {  
 return null;  
 }  
  
 @Override  
 public Class<?> getObjectType() {  
 return null;  
 }  
}

* Inainte de a edita metodele default cream o instanta de calendar inautru.



* Avem contractul pt Factory setat acum.
* Ne trebuie o cale de a da valori Bean-ului incapsulat de Bean Factory 🡪 adaugam o metoda public void: in **CalendarFactory**
* public void addDays (int num) {  
   this.instance.add(Calendar.*DAY\_OF\_YEAR*,num); // va adauga un numar de zile instantei pasate in metoda  
  }
* In momentul de fata avem Factory-ul creat Corespunzator.
* Urmeaza Configurarea:
* Deschidem AppConfig si cream doua Bean-uri:

Unul pentru **CalendarFactory**

Al doilea pentru **instance of that Calendar**

@Bean(name = "cal")  
public CalendarFactory calFactory() {  
 CalendarFactory factory = new CalendarFactory();  
 factory.addDays(2);  
 return factory;  
}  
  
@Bean  
public Calendar cal() throws Exception {  
 return calFactory().getObject(); // De aici obtinem instanta din Factory-ul nostru  
}

Utilizam instanta de Calendar in interior-ul **HibernateSpeakerRepositoryImpl:**

Si facem Autowire la instanta pentru ca Spring sa creeze Bean-ul



@Repository("speakerRepository") // acum acest bean va fi setat ca Repository utilizand Stereotype Annotation  
public class HibernateSpeakerRepositoryImpl implements SpeakerRepository {  
  
 @Autowired  
 private Calendar cal;  
  
 public List<Speaker> findAll() {  
 List<Speaker> speakers = new ArrayList<Speaker>();  
  
 Speaker speaker = new Speaker();  
 speaker.setFirstName("Alexandru");  
 speaker.setLastName("Din");  
 System.*out*.println("cal: " + cal.getTime());  
 speakers.add(speaker);  
  
 return speakers;  
 }  
}

Ruland aplicatia:



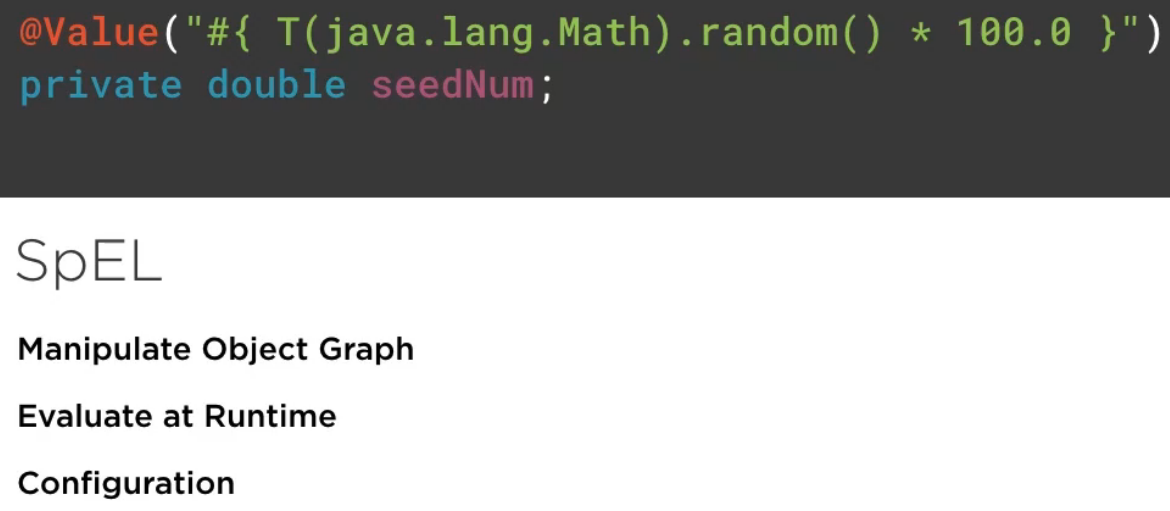
OBS:

cal: Wed Sep 02 14:19:54 EEST 2020 🡪 Vine din SpeakerRepository Instance;

Este o metoda de a manevra date dintr-un obiect utilizand o metoda

**Spring Expression language SpEL**

* Poate fi utilizat in libraries dar si sa faci modificari in cod.
* Poti manipula Object Graph: Odata ce obiectul a fost creat poti utiliza expression language pentru a il manipula
* Putem evalua valori la Runtime si sa modificam comportamentul codului in concordanta
* Putem utiliza pentru a evalua si manipula configurarea.

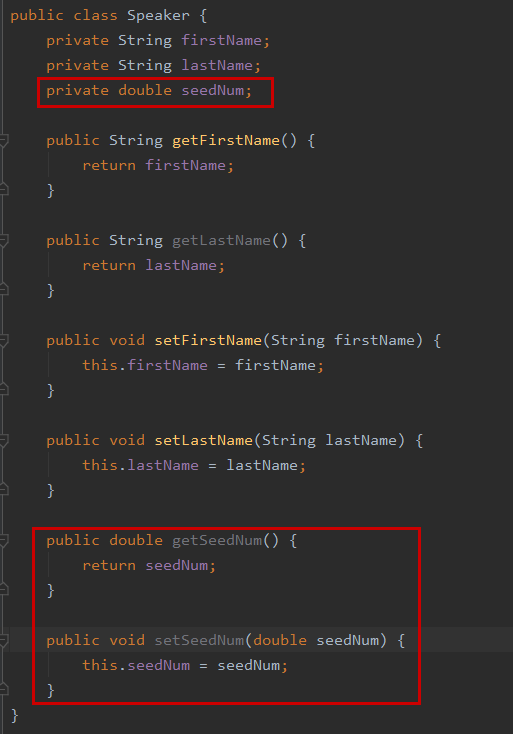


Vom adauga un Seed Value obiectului Speaker:

Si cream un Getter si Setter:

public class Speaker {  
 private String firstName;  
 private String lastName;  
 private double seedNum;

public double getSeedNum() {  
 return seedNum;  
}  
public void setSeedNum(double seedNum) {  
 this.seedNum = seedNum;  
}}



Mergem in **HibernateSpeakerRepositoryImpl**

Unde am creat instanta de Calendar: sub **@Autowired**

@Value("#{T(java.lang.Math).random() \* 100}")

Vom folosi Spring Expresion language pentru a obtine o valoare random din libraria **java.lang.Math**

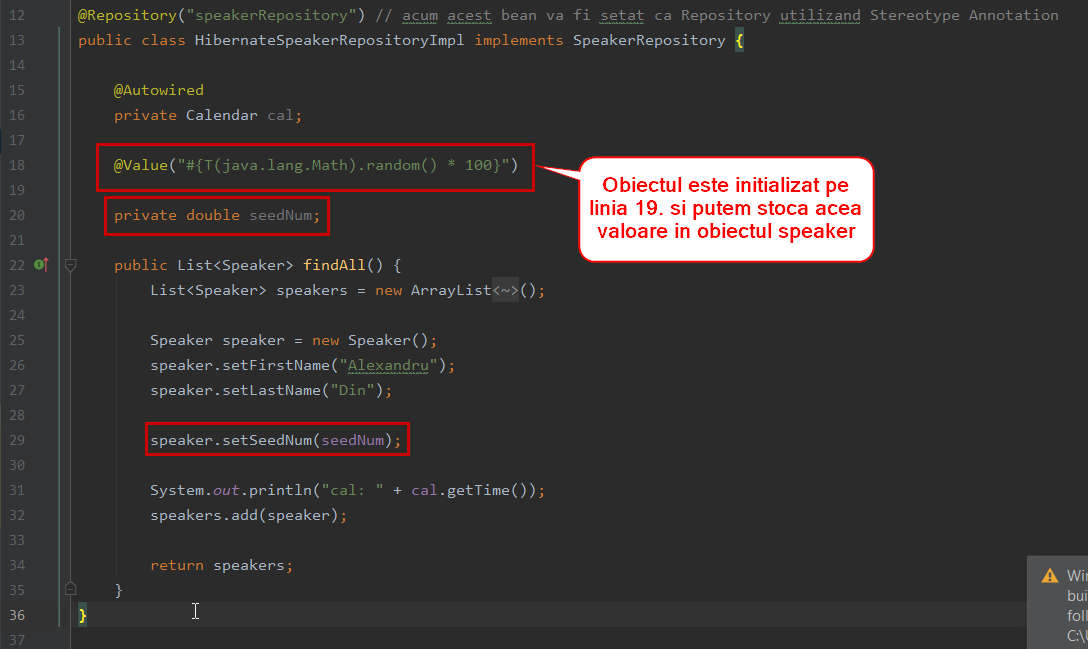
Pe care o vom stoca in

private double seedNum;

Acum in momentul compilarii putem lua acea valoare generata la runtime si sa o injectam in cod.

(Setter injection pentru valoarea seedNum)

speaker.setSeedNum(seedNum);



cod complet:

@Repository("speakerRepository") // acum acest bean va fi setat ca Repository utilizand Stereotype Annotation  
public class HibernateSpeakerRepositoryImpl implements SpeakerRepository {  
  
 @Autowired  
 private Calendar cal;  
  
 @Value("#{T(java.lang.Math).random() \* 100}")  
  
 private double seedNum;  
  
 public List<Speaker> findAll() {  
 List<Speaker> speakers = new ArrayList<Speaker>();  
  
 Speaker speaker = new Speaker();  
 speaker.setFirstName("Alexandru");  
 speaker.setLastName("Din");  
  
 speaker.setSeedNum(seedNum);  
  
 System.*out*.println("cal: " + cal.getTime());  
 speakers.add(speaker);  
  
 return speakers;  
 }  
}

Aceeasi metodologie se foloseste si la generarea de parole sau alte valori random.

Psvm:

ApplicationContext appContext = new AnnotationConfigApplicationContext(AppConfig.class);  
  
 SpeakerService service = appContext.getBean("speakerService", SpeakerService.class);  
  
 System.*out*.println(service); // adresa obiectului service pt a vedea daca este singleton  
 System.*out*.println(service.findAll().get(0).getFirstName());  
 System.*out*.println(service.findAll().get(0).getSeedNum());

Output:

SpeakerServiceImpl repository Constructor

We are called after the constructors

com.plsProject.service.SpeakerServiceImpl@156b88f5

cal: Mon Sep 07 20:52:46 EEST 2020

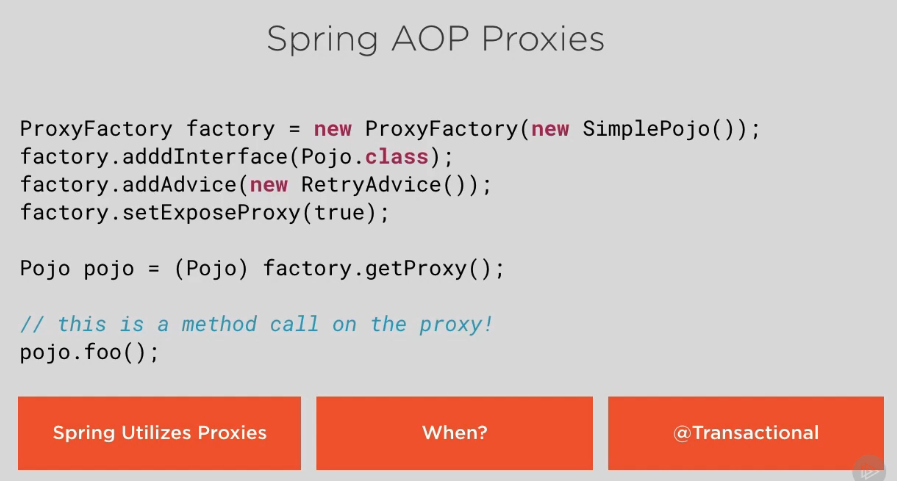
Alexandru

cal: Mon Sep 07 20:52:46 EEST 2020 🡪 Instanta calendar

83.45464133933731 🡪 Valoarea SeedNum pe care am pasat-o in app cu SpEl

**Spring AOP Proxies**

* Modalitate buna de a injecta comportament nou in cod-ul de baza fara a il modifica.
* In poza: **SimplePojo** este clasa catre care vrem sa proxy (sa o accesam prin proxy).
* **Pojo** este interfata cu care vom face chemari prin intermediul proxy-ului, care va permite interceptarea chemarilor.
* Se foloseste cand lucram pe baza de framework-uri sua librari dar cel mai des se utilizeaza la interactiunea cu baze de date, utilizand Adnotarea: **@Transactional**
  + **@Transactional** il pui la inceputul cod-ului de baza astfel sa fie inconjurat de o tranzactie (sa se comporte ca o tranzactie)



**Bean Profiles**

* Implementate pentru a putea adaptarea cod-ului la diferite medii.
* Permite setarea unui cod specific care va fi rulat doar intr-un mediu specific.
  + Astfel in cat sa putem modifica configurari la Runtime

Pentru a adauga Bean profiles la noi in aplicatie:

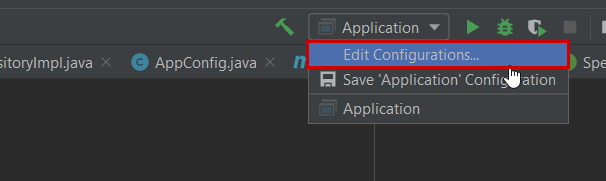
**Adaugam adnotarea @Profile(“DenumireProfil”)** atatin **HibernateSpeakerRepositoryImpl** cat si in **SpeakerServiceImpl:**

@Repository("speakerRepository") // acum acest bean va fi setat ca Repository utilizand Stereotype Annotation  
**@Profile("dev")**  
public class HibernateSpeakerRepositoryImpl implements SpeakerRepository {. . . . .

@Service("speakerService")  
**@Profile("dev")**  
@Scope(value= BeanDefinition.*SCOPE\_SINGLETON*)  
public class SpeakerServiceImpl implements SpeakerService {

“dev” = cuvant cheie ales de noi.

Mergem sus in Aplicatie: 🡪 Edit Configurations...



**VM options:** Asa ii spunem sa aleaga / schimbe profilul pe care il vom rula

**-Dspring.profiles.active=dev 🡪 ii spunem sa aleaga acest profil**



Si acum cand rulam aplicatia va rula cu un anumit profil.

SpeakerServiceImpl repository Constructor

We are called after the constructors

com.plsProject.service.SpeakerServiceImpl@74751b3

cal: Tue Sep 08 01:12:19 EEST 2020

Alexandru

cal: Tue Sep 08 01:12:19 EEST 2020

89.39838993328549

**OBS** Daca nu am fi trecut in VM options (**-Dspring.profiles.active=dev**) nu ar fi avut un profil ceea ce inseamna ca Bean-urile adnotate cu **@Profile("dev")** nu ar fi fost valabile:

Ex: daca modificam in **@Profile("costel") dar NU modificam si -Dspring.profiles.active=dev**.

Vom primi un NullPointer Exception deoarece nu poate gasi acel Bean adnotat cu **@Profile("costel")**. ,el vede doar ce este adnotat cu **@Profile("dev")**

Deci acele Bean-uri vor rula doar in anumit cod in anumite conditii de mediu.

**HibernateSpeakerRepositoryImpl** este un exemplu bunde locatie unde am vrea sa avem acest profil de **“dev”** deoarece noi simulam aici o baza de date hard codata. Si astfel acest Bean va rula doar in acest mediu, stim astfel ca trebuie reparata si conectata la o baza de date reala. Tinand profilul de dev nu va rula in productie si pot utiliza acest profil drept un reminder ca aici mai trebuie modificat cod-ul

Summary:

Bean Lifecycle 🡪 init method

FactoryBean 🡪 bun pt legacy code

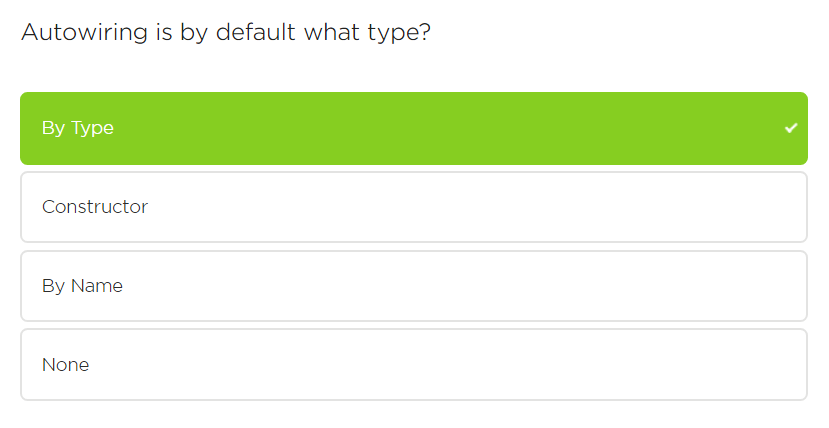
SpEL 🡪 manipulate the object Graph at runtime

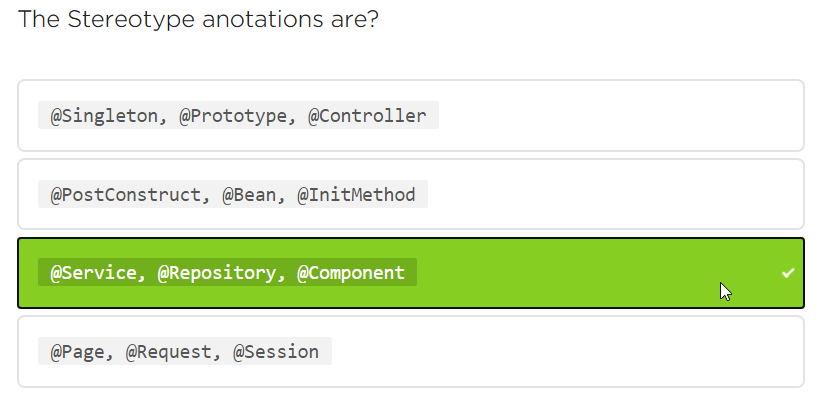
Proxies 🡪 just a summary

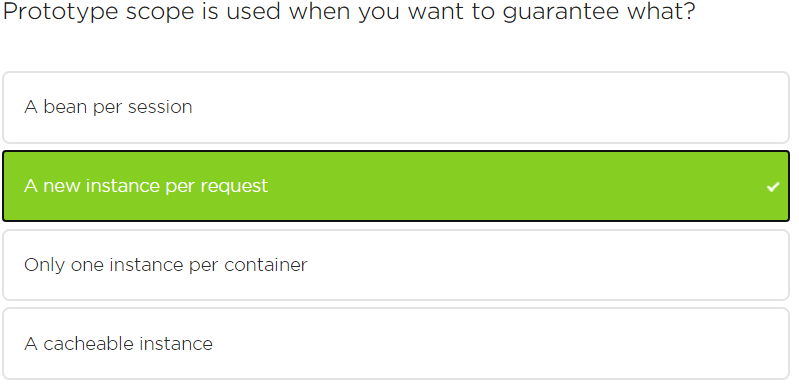
Profiles 🡪 To make sure Code runs in certain environments.

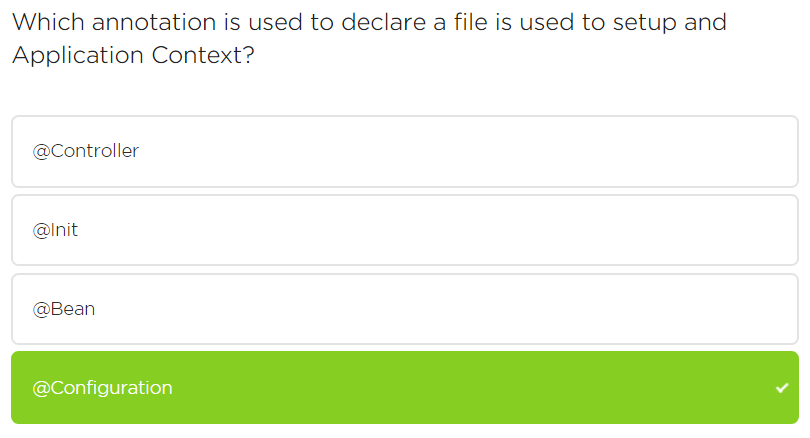
**alexdin95/Spring\_ConferenceApp\_Practice**

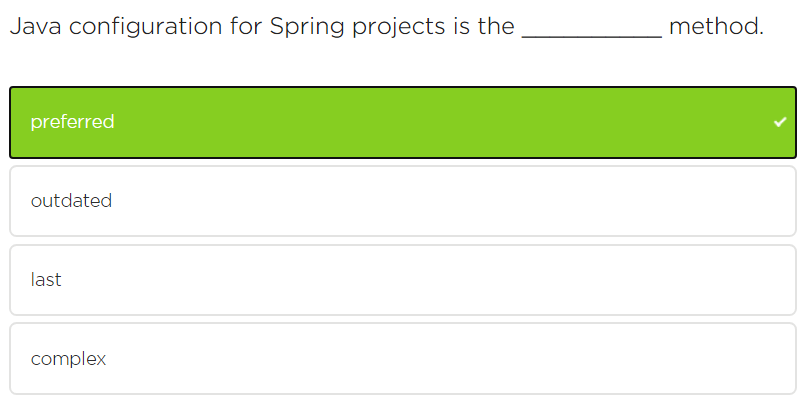
Quiz:

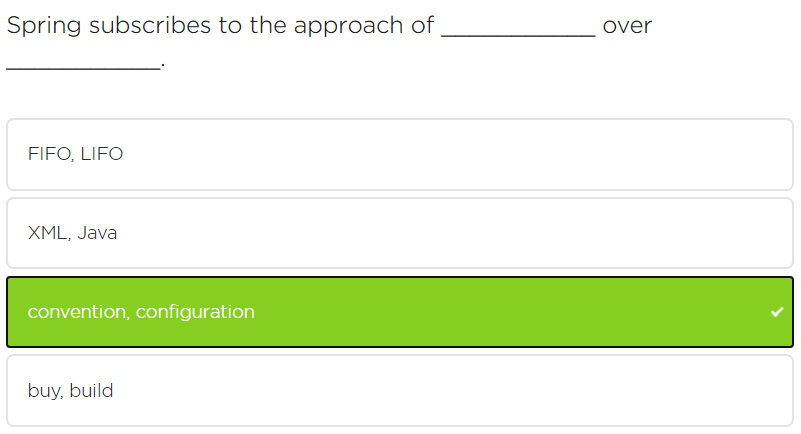




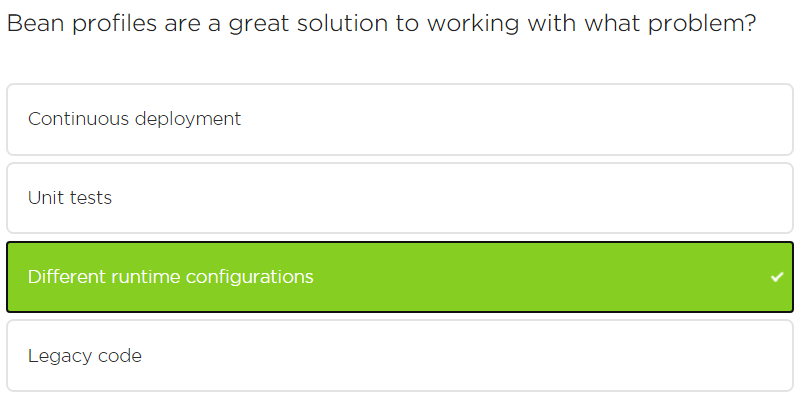


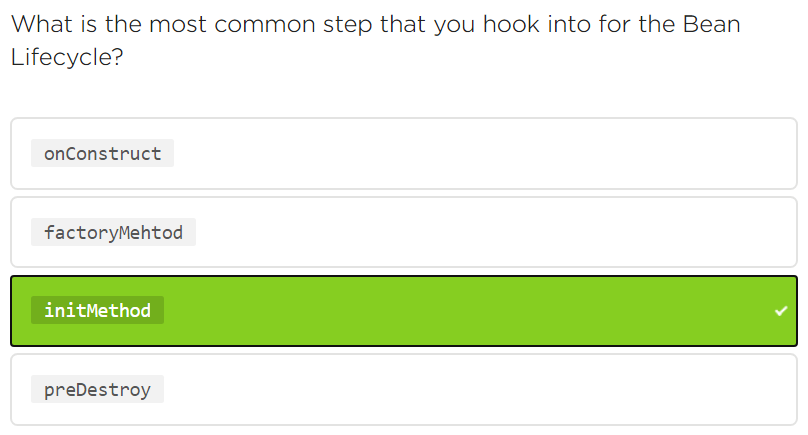


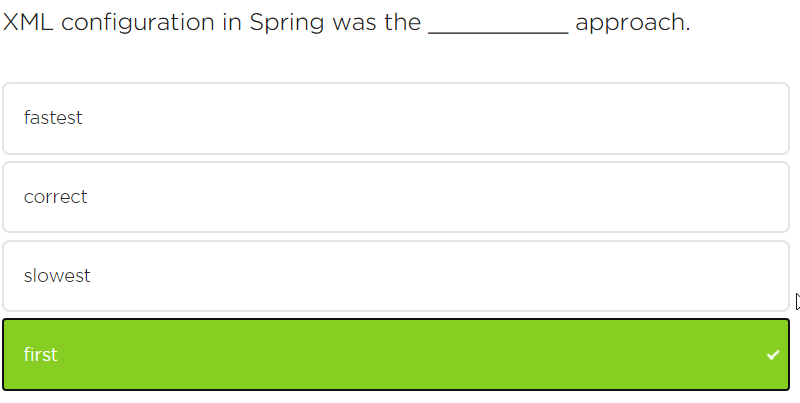












Next Courses to do:

Spring MVC Intro Course

**Autowiring + Component Scanning in Spring**

**(Should watch the design patterns course ( by Bryan Hansen))**

**Web Scopes**

Covered in (Introduction to Spring MVC course) 🡪 not really covered here: