**Spring Framework and Fundamentals**

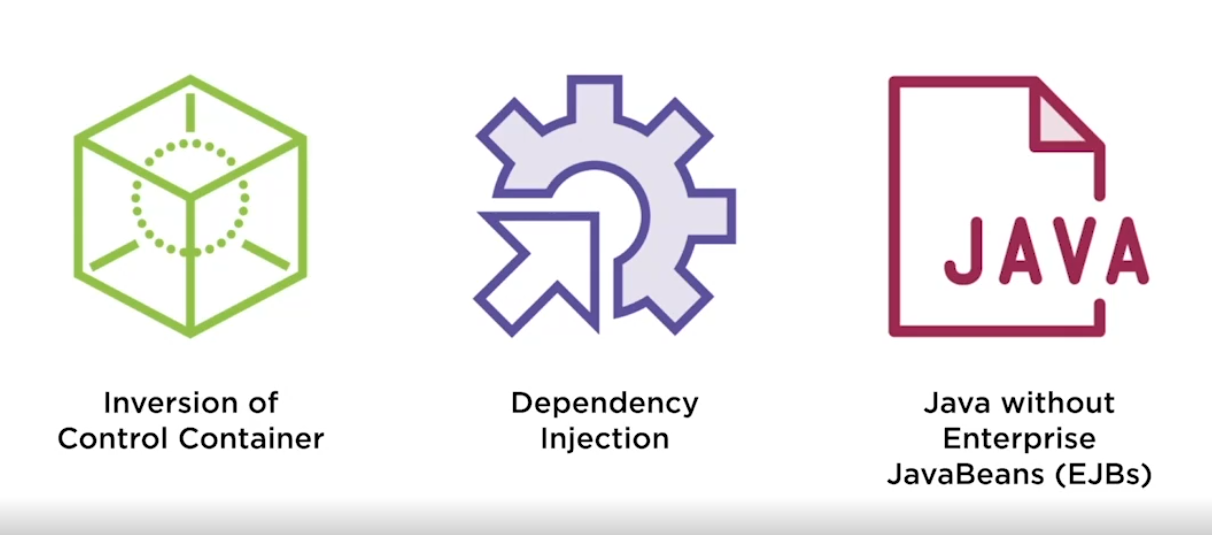
**Explain the process of injecting bean of Student having different impl inside a StudentService using field injection.**

**Look at last.**

What is Spring?

Spring framework is started out as an **inversion of control container** Often refer as **Dependency injection**.

It was used to replace some of the complex configuration of earlier Java enterprise edition. How we can do same things without using EJBs.



* It was started to reduce complexity in enterprise java development and later also providing enterprise development without EJBs.
* Spring can be used with or without java beans primarily now is used without EJBs.
* Spring allowed us to do enterprise development without **application server**. Tomcat that we use is isn’t an application server rather it is web server.
* It is complex to do development with application server like WebSphere and with tomcat as it is light weight can be used easily.

So finally, what is spring?



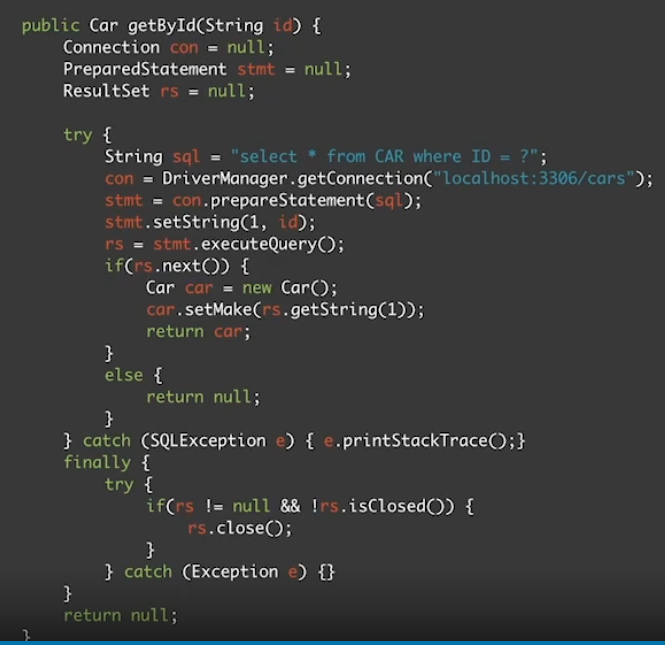
* JEE
* **POJO** – plain old java object
* **Unobtrusive**- As spring was developed to reduce complexity of enterprise development so it will not come in your way of development.
* **AOP/Proxies** – Spring also use AOP/Proxies to apply things like transactions to code to get those cross-cutting concerns out of our application so our code is smaller and more light weight.
* **Best Practices** – Spring is built around best practices

**Advantages of Spring:**



* Increases testability and maintainability. Helps in scaling our code.
* It decouples things and helps in add caching layer
* Reduce code complexity
* Focus on business, business doesn’t care about what framework we use it cares about things done and spring helps us do complex things faster.

**PROBLEM**



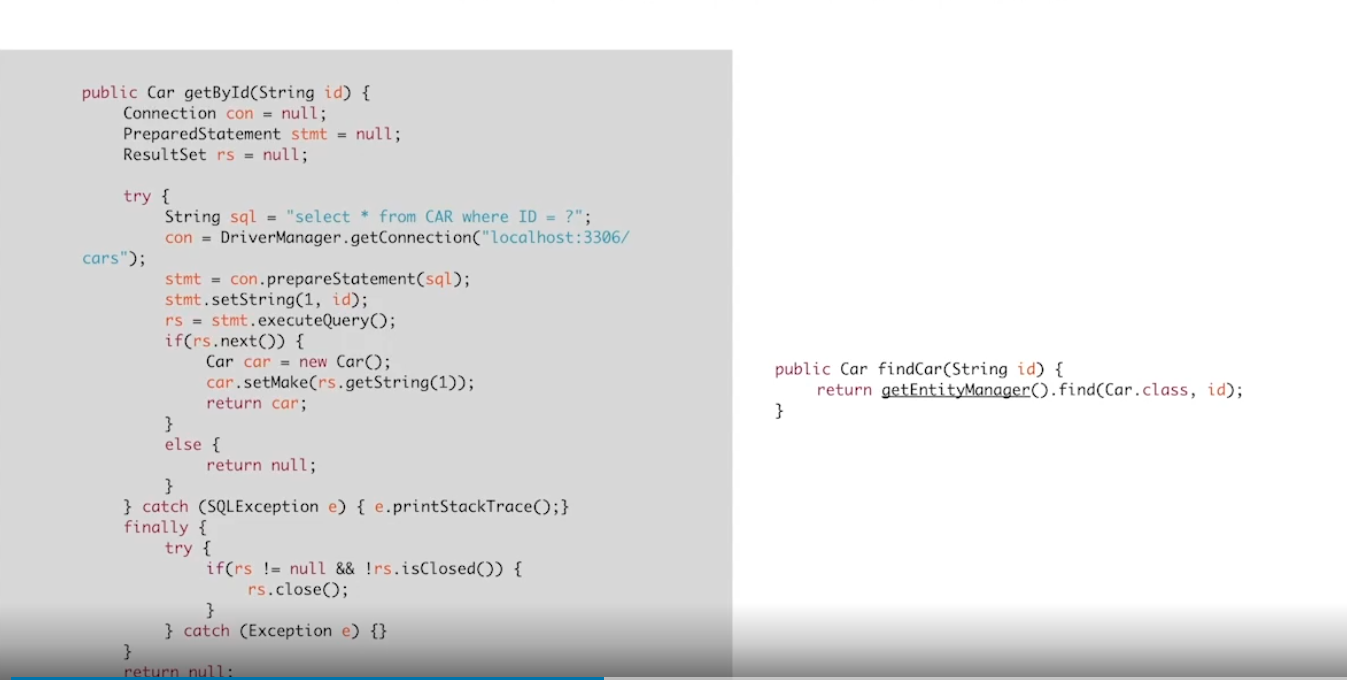
The only thing that business care about in the above code is selecting the car object and setting it. All these initialization, try/catch , big finally block etc etc just makes our code complex. Lets see how Spring resolves it.

THE SOLUTION

The solution we get from spring or using spring framework is:

* We can remove **Configuration** code or lookup code
* Developers can **Focus** upon business needs. Business doesn’t care about try/catch , exceptions etc
* **Increase TESTING**
* **Annotation/XML** based development – Rather than having big configuration code we can have annotation-based development.
* To achieve all of this spring encourage us for **interface-based development.**

**BUSINESS FOCUS REVISTED**



This library used on right is basically Spring-JDBC template code. It removes all configuration, we don’t need to care about all these. By using **TEMPLATE-Method pattern** we have achieved it.

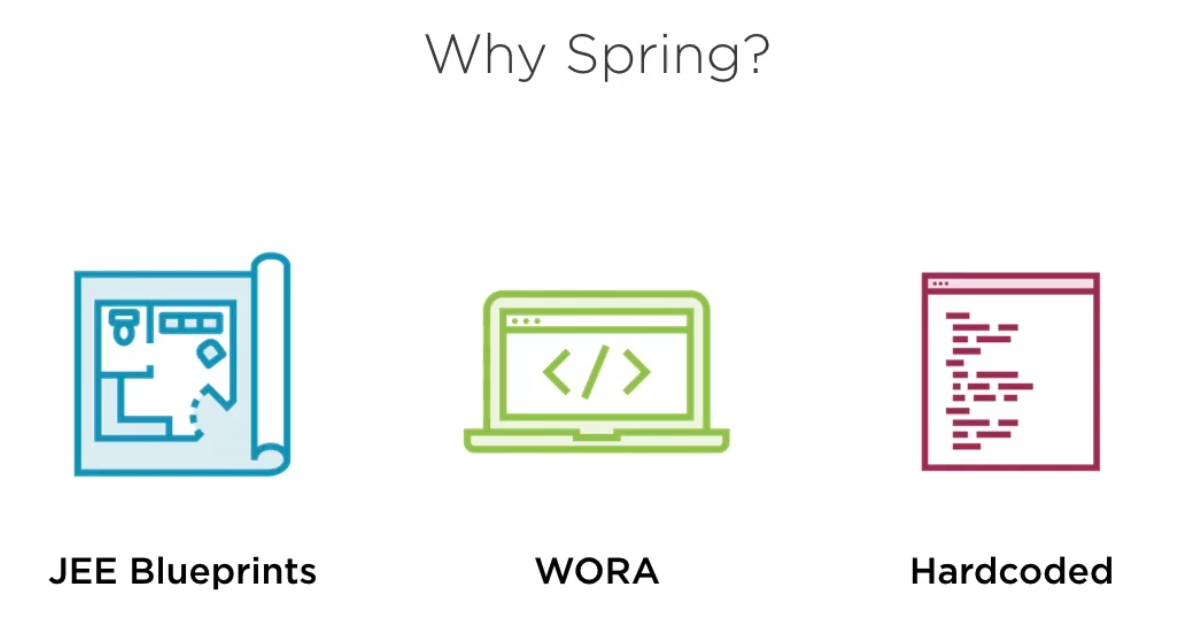
**So it basically helps us to achieve Business goals faster.**

**HOW SPRING HELPS US IN BUSINESS FOCUS?**

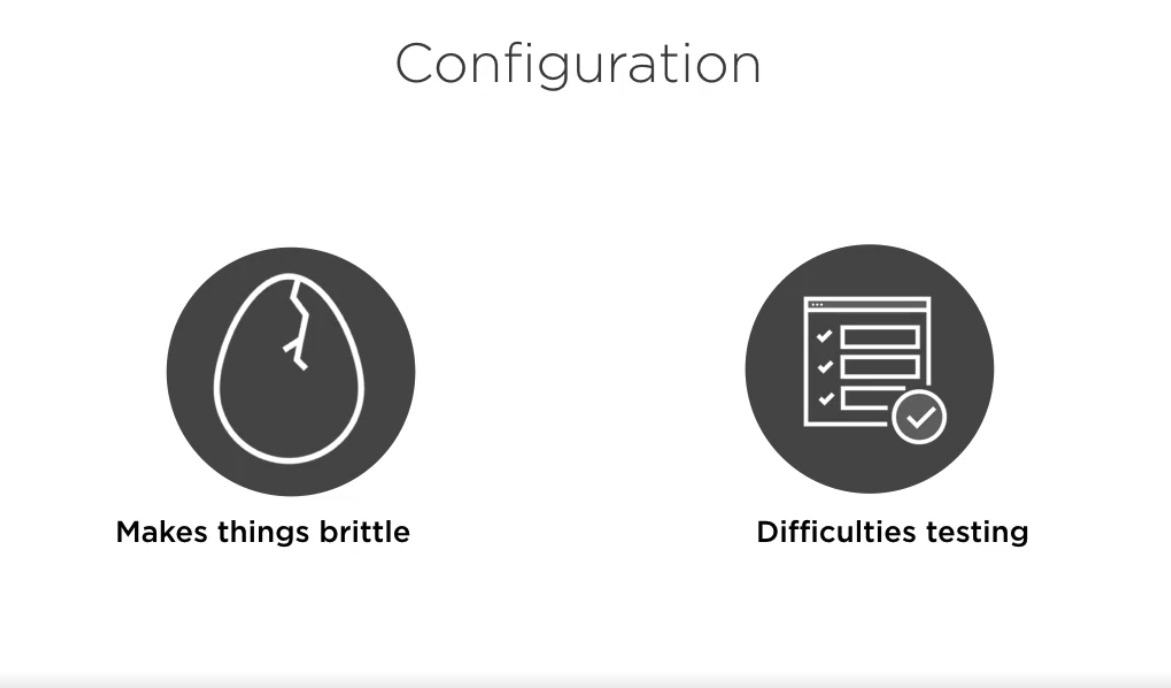
* **POJOs**
* **Hash Map**- Spring is thought of as a glorified hash map, it calls the Application context. Application context is spring container with all of the dependency wired up.
* **Registry**

**WHY SPRING WAS DEVELOPED?**

* Spring was developed to make existing task easier.
* Before spring we used some design patterns from **JEE blueprints** to help establish better code and repeatable process. This was helpful but often **made code brittle and untestable.**
* **WORA(Write once run anywhere)-** Another problem was if we have to recompile code into other environment or need to change URL etc.
* Actual implementation should not be **hardcoded inside application.**







We should always get rid of configuration code like

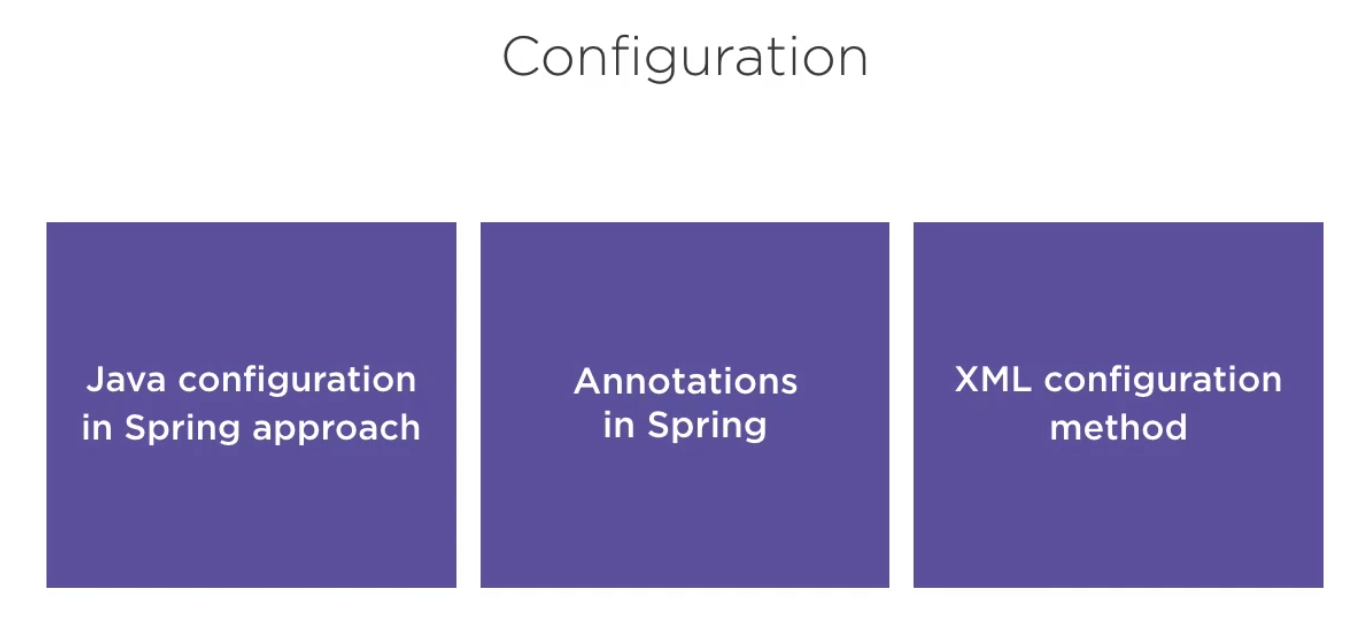
**public class** SpeakerServiceImpl **implements** SpeakerService {  
  
 *//This is the painpoint where we have to manually create a object of repo class. We should reduce Configuration code from our application  
 // cause configuration code is brittle (hard to move to diff env)* **private** SpeakerRepository **speakerRepository** = **new** HibernateSpeakerRepositoryImpl();  
  
 **public** List<Speaker> findAll(){  
 **return speakerRepository**.findAll();  
 }  
}

In the above code we have manually create the instance with new and we have hardcoded the reference of HibernateSpeakerRepositoryImpl object , if we make any changes we need to rebuild the entire application and such type of configuration we don’t want because.

* These are brittle, hard to move to different server env.
* Hard to test.

**CONFIGURATIONAL code is something that we don’t require as a business logic. Hence, we should minimize that.**

**In Spring we can deal with configuration using below:**

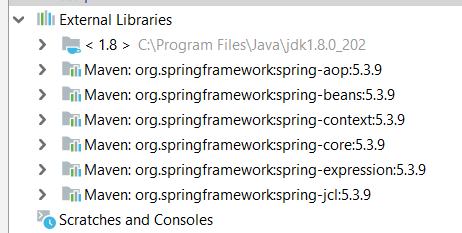


**In order to use Spring in our project we need to download Spring libraries from Maven using POM file.**

**Once we do this**

<**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-context</**artifactId**>  
 <**version**>5.3.9</**version**>  
 </**dependency**>  
</**dependencies**>

**Then it will download all external libraries**

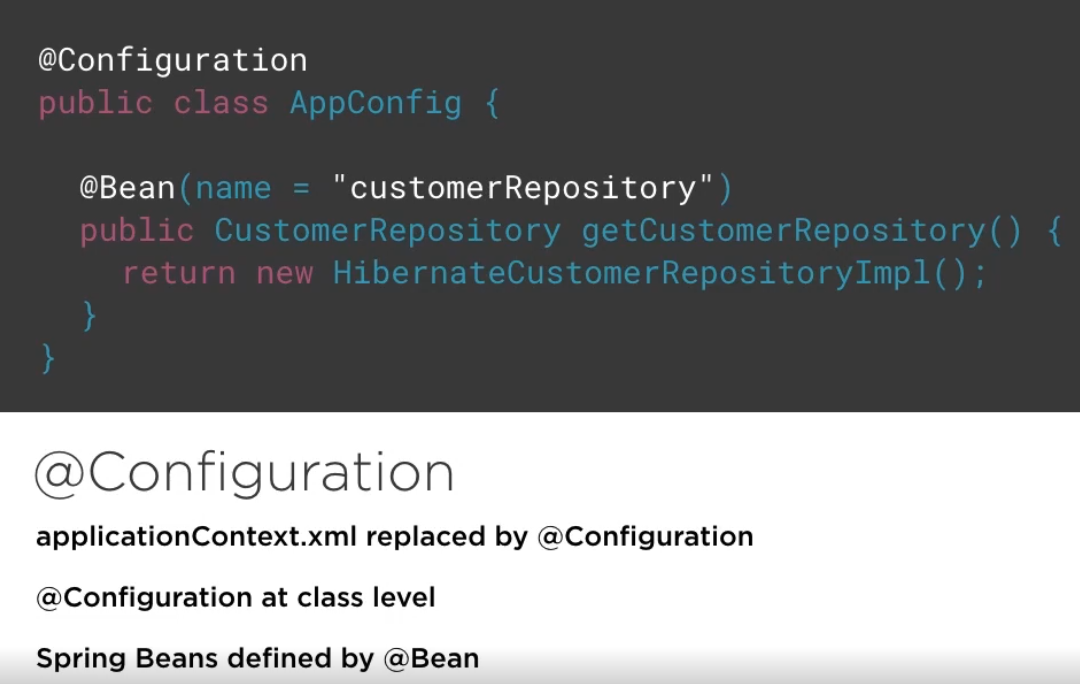


**SPRING CONFIGURATION USING JAVA**

* No ApplicationContext.xml **, No XML** we have context but not configured in xml.
* Earlier spring had a lot of xml but **now every configuration is done in pure java**.

**AppConfig**

1. First we need to add AppConfig, this is the place where we start bootstrapping[self starting] our application.
2. This is where we start configuring the context of our application.



We have two annotations here:

1. **@Configuration** – A class level annotation. Annotating a class with the @Configuration indicates that the class can be used by the Spring IoC container as a **source of bean definitions**.

This class provides one or more **@Bean** methods and may be processed by the Spring container to generate bean definitions

1. **@Bean –** A method level annotation**.** The @Bean annotation tells Spring that a method annotated with @Bean **will return an object** that should be **registered as a bean in the Spring application context.**

**SETTER INJECTION**

Simple as a method call

“Mystery” of injection goes away

**Setter injection is simply calling a setter.**

**Bean is always a singleton only one object will be created no matter we call it 100 times.**

**WITHOUT SPRING:**

**public class** SpeakerServiceImpl **implements** com.pluralsight.service.SpeakerService {  
  
 *//This is the painpoint where we have to manually create a object of repo class and we have hardcoded the reference of HibernateSpeakerRepositoryImpl  
 // object, if we make any changes we need to rebuild the entire application .  
 // We should reduce Configuration code from our application  
 // cause configuration code is brittle (hard to move to diff env)* **private** SpeakerRepository **speakerRepository** = **new** HibernateSpeakerRepositoryImpl();  
  
  
  
 **public** List<Speaker> findAll(){  
 **return speakerRepository**.findAll();  
 }  
}

**In the above code we are creating a hard-coded instance of repository class inside the service and then call the repo method.**

**WITH SPRING:**

Instead of hardcoding the instance we create a setter of repo which will wire up the configuration. Instances are injected rather than hardcoding.

**public class** SpeakerServiceImpl **implements** com.pluralsight.service.SpeakerService {  
  
**private** SpeakerRepository **speakerRepository**;  
  
 *//We create this setter so that when we create bean of service class at that time only we can inject the bean of repo class.* **public void** setSpeakerRepository(SpeakerRepository speakerRepository) {  
 **this**.**speakerRepository** = speakerRepository;  
 }  
  
 **public** List<Speaker> findAll(){  
 **return speakerRepository**.findAll();  
 }

**CONSTRUCTOR INJECTION:**

*//CONSTRUCTOR INJECTION* SpeakerServiceImpl speakerServiceImpl = **new** SpeakerServiceImpl(getSpeakerRepository());  
 **return** speakerServiceImpl;  
}

**AppConfig changes**

@Bean(name=**"speakerService"**)  
 **public** SpeakerService getSpeakerService(){  
 *// SETTER INJECTION  
// SpeakerServiceImpl speakerServiceImpl = new SpeakerServiceImpl();  
// speakerServiceImpl.setSpeakerRepository(getSpeakerRepository());  
  
 //CONSTRUCTOR INJECTION* SpeakerServiceImpl speakerServiceImpl = **new** SpeakerServiceImpl(getSpeakerRepository());  
 **return** speakerServiceImpl;  
 }

**SPRING SCOPES AND AUTOWIRING**

* Scopes of Bean inside our application and important part of bean life cycle.
* Scopes are associated with Patterns but are not the same thing. Patterns and Scopes goes hand in hand **but are not equal**.
* Spring implements a lot of patterns for us and we don’t have to implement manually.
* To configure a Bean inside our application there are **5 scopes available**.

Valid in any configuration:

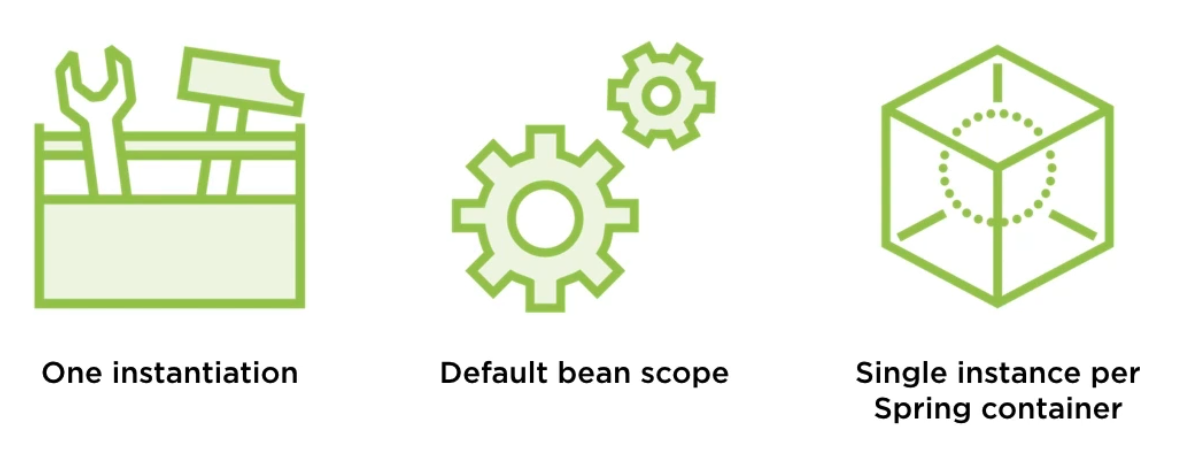
* Singleton - Actually by default
* Prototype – Actually new Bean per request
* We do want Singleton over prototype

Valid only for web-aware spring project

* Request
* Session
* Global

**SINGLETON PATTERN IN SPRING**

* Singleton pattern restricts creation of object to only one instance.
* It is a default Bean scope inside spring.
* There is **one instance per spring container or application context**. That means one instance per JVM (this is true when we have only one spring container if we have more than one container than instances can be more).



How to use scope in our spring project



@Scope is a transitive dependency in maven so it is there by default. No need for AOP jar.

**INSIDE APPLICATION CLASS**



Calling the Bean two times and printing the object.

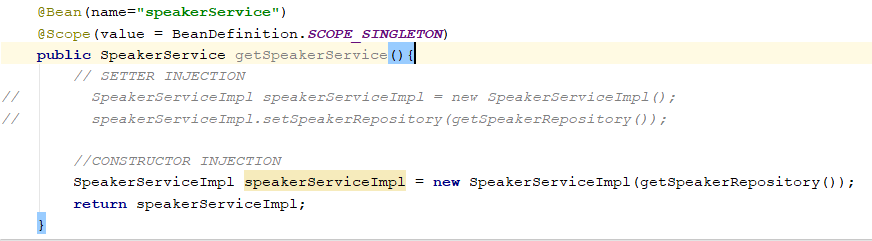
**OUTPUT:**



We have not made any change to the scope of the Bean and by default it is singleton.

**HOW TO MANUALLY WRITE THE SCOPE**

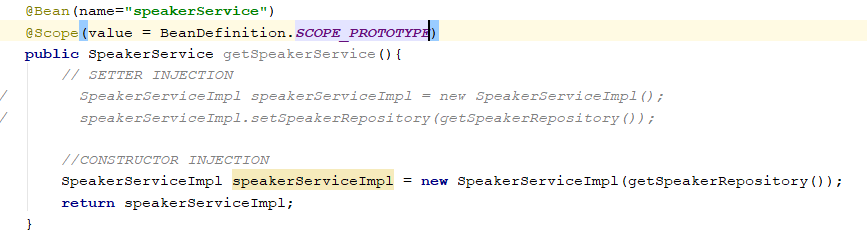
The place we register the Bean write

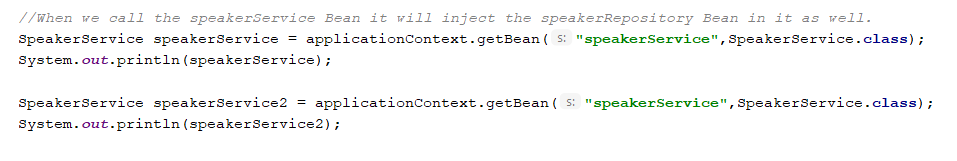


**PROTOTYPE**

* Prototype design pattern guarantees unique instance per request.
* Hence when we define BEAN with prototype scope than each time we request Bean from a container we get unique instance.
* It is opposite of a singleton.









**WEBSCOPES**

1. **Request – new** Bean per http request sounds like prototype but it is for the life cycle of Bean request.
2. **Session** – Returns single Bean per http session that will live till user is active.
3. **GlobalSession** – Return a single Bean per application. It will be available till we don’t reboot the application.

**AUTOWIRING**

Autowiring is a technique to reduce the wiring up and configuration of code.

To autowire our application using java configuration we just need to do add a componentScan to our configuration file.



We give our package as a parameter which tells this is the place we should look for auto wire annotations.

To autowire our Bean we should mark our bean as @autowire.

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

**By declaring all the bean dependencies in a Spring configuration file, Spring container can autowire relationships between collaborating beans**. This is called *Spring bean autowiring*.

**In simple language:**

**Autowiring is a technique of reducing the configuration and wiring up code. When we don’t use Autowiring then we manually have to create beans in our Appconfig file and wire up /inject one bean instance into another using setter or constructor injection. But once we start using Autowiring then these injection or wiring up is happening automatically.**

**By using stereotype annotations we are not defining @Beans in Appconfig rather using ComponentScan({“package”}) to let app context know where to look for Beans.**

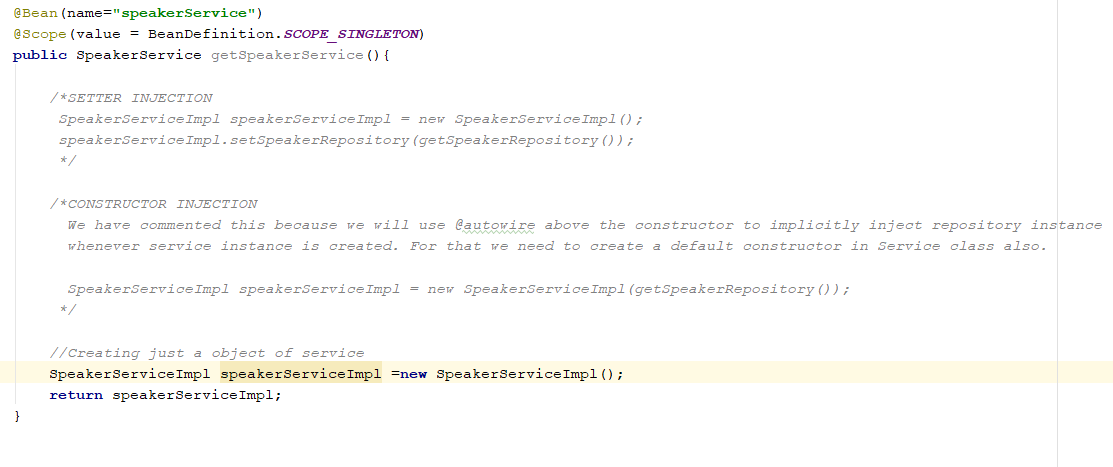
**And we use @Component (like @Service @Repository) to let app context register the Beans and look for any Autowiring inside the class.**

Autowiring means instead of we manually injecting/wiring one bean instance into another using setter or constructor. IF we define @Autowire in variable/setter/constructor then in AppConfig we don’t need to do it manually.

To convert code from manual injection to Autowiring:

Steps

1. Go to AppConfig.java and comment the manual way of injecting repo instance and just create object of service class.



1. Now go to SpeakerServiceImpl and put @Autowire above the setter.

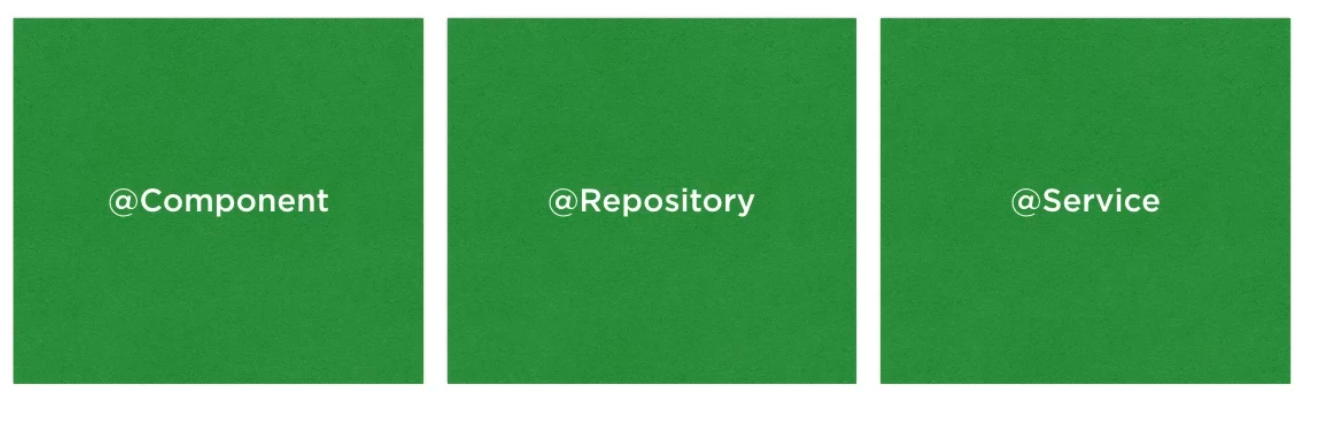


1. Once ApplicationMain try to get Bean of service from container then , first no args constructor of ServiceImpl is called and then it sees the @autowire and it will try to create instance of Repo and call no args constructor of repo and then code inside setter is called. Then finally injecting the instance of repo in service .

**In the above method the whole BEAN is not fully Autowired rather than just a method , hence we didn’t use componenetScan in AppConfig.**

**STEREOTYPE ANNOTATIONS**

* To fully autowire Beans we need stereotype annotations.
* By using this we don’t need to write configuration for Beans in the AppConfig file.



@Component is same as @Bean.

@Repository – Use to class used as Repo object.

@Service – Use for class such as business logic.

@Controller -

Technically we can use @Component for repo or service also.

**HOW TO USE**

1. Use @ComponentScan({“com.pluralsight”}) – The parameter tells that this is the package spring will start looking for Bean autowire annotations.
2. Get rid of all the Bean definition and code inside AppConfig.

**NOTE : @BEAN annotation can only be used at method level and not above any class level. For class level we have @Component.**

1. Use @Service at top of service class

**WHY STEREOTYPE ANNOTATION WAY IS GOOD BECAUSE WHEN WE HAVE 100-200 BEANS THEN WE NEED TO MANUALLY DO THINGS IN APPCONFIG. So we get ride of hundred lines of code**

**NOTE: IMPORTANT**

**When we execute**

ApplicationContext applicationContext=**new** AnnotationConfigApplicationContext(AppConfig.**class**);

Then it will load spring and load our congif file in the app context. At this point no matter we do partial Autowiring or full it will register our bean **(call the constructor of Bean).** As well as call the setters and inject the dependency of one Bean into another.

When we execute

SpeakerService speakerService = applicationContext.getBean(**"speakerService"**,SpeakerService.**class**);

At this point it will just take our instance from spring container. It will not create d new instance as by default everything is singleton.

**Stateless beans**: beans that are singleton and are initialized only once. The only state they have is a shared state. These beans are created while the ApplicationContext is being initialized. The SAME bean instance will be returned/injected during the lifetime of this ApplicationContext .

**SCENARIO 1** : **When we do partial Autowiring(injection using setter) and both are singleton.**

**OUTPUT**

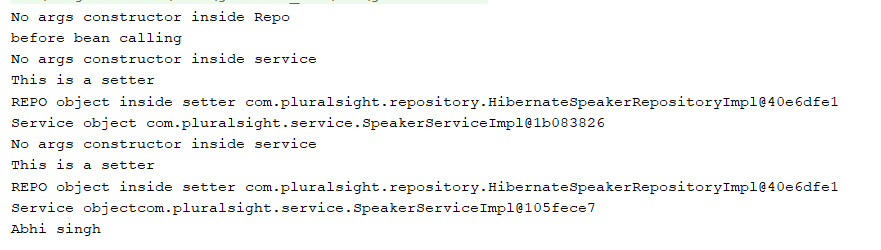
As soon as applicationcontext is created and config file is loaded then

* First call default constructor of service. [Register our service]
* Then calls default constructor of repository. [Register our repo]
* Then do injection goes inside setter method and execute. [ Inject repo into service]

After all of the above we do getBean for service then it will just return bean of service with repo injected**. NO NEW INSTANCE IS CREATED OR CONSTRUCTOR IS CALLED.**

**SCENARIO 2 : When we do partial Autowiring(injection using setter) and only service is prototype and repo is singleton.**

**OUTPUT**



As soon as we execute applicationcontext then

* Only constructor of repo is called as it is singleton[Registry of REPO]
* Then it will execute statement below app context.

As soon as we call getBean of service then

* No args constructor of service is called [Registry of Service]
* Constructor of repo will not be called as it is singleton and already registered.
* Setter is called [ Injection happen]

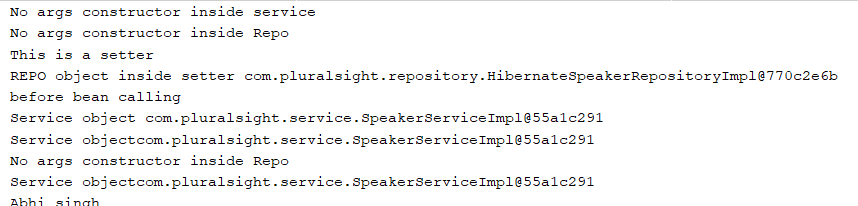
As soon as we call getBean of service again

* No args constructor of service is called [Registry of Service]
* Constructor of repo will not be called as it is singleton and already registered.
* Setter is called [ Injection happen]

So when service class is prototype then registry of service Bean will happen again and injection will also happen again.

**Service object changes but Repo object is always same.**

**SCENARIO 3 : When we do partial Autowiring(injection using setter) and service is singleton but repo is prototype**



As soon as we execute app context then

* Service constructor is called [Register service]
* Repo constructor is called because of autowire setter [ Repo is registered]

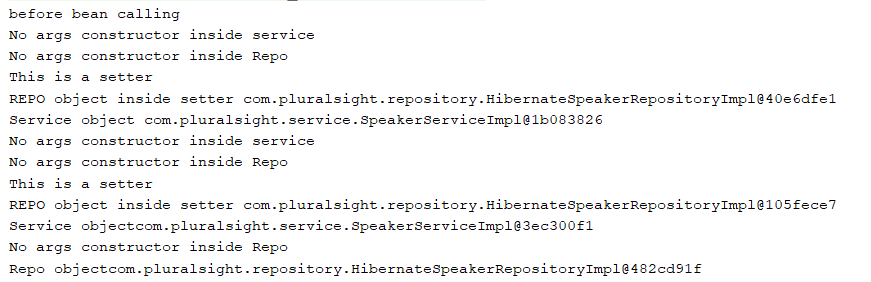
As soon as service bean is called then we will get the bean of service injected with repo

As soon as repo bean is called explicitly

* Repo constructor is called [Repo registered again]

**Service object is same but Repo object is always new when getBean of repo is called.**

**SCENARIO 4 : When we do partial Autowiring(injection using setter) and service and repo both are prototype**



When we execute app context then

* Nothing is registered at first because both are prototype

When we call getBean for service

* Service registry
* Repo registry
* Injection

When we call getBean again for service

* Service registry [ new instance]
* Repo registry [new instance]
* Again injection will happen

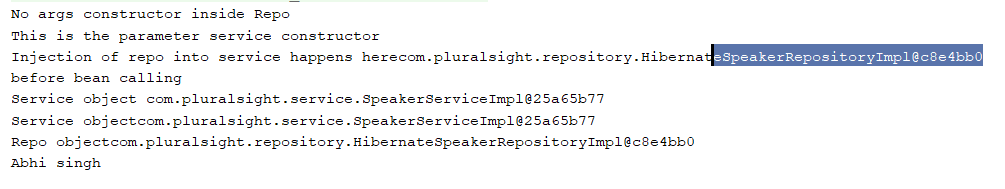
When we call getBean for Repo explicitly

* New repo registry

**Everytime we call getBean new service and repo objects are created and injected.**

**FULL AUTOWIRING [If we want to use constructor Autowiring]**

**Scenario 1 : Full Autowiring and both are singleton**



**NOTE: We don’t need default constructor of service when we are doing constructor Autowiring. Object is created using param constructor of service only.**

When we execute app context then

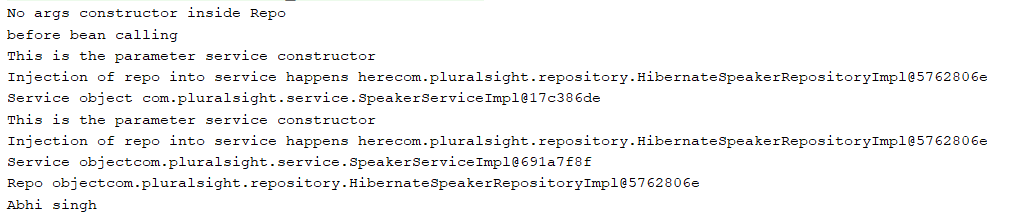
* First REPO registry [ cause it will first go to param constructor to create object of service but it sees @Autowire hence it will got to repo default constructor and register its bean]
* Then it register SERVICE BEAN
* Then injection happens

When we call getBean of service then always we get object single instance which is injected with repo.

When we explicitly call repo bean then also same instance we will get.

Service and Repo both object is same.

**SCENARIO 2 : Service prototype and Repo default**



As we execute appp context then

* ONLY REPO Registered

As soon as we call getBean for service

* Service register using param constructor but repo bean will not register
* Injection happen

As soon as we call getBean for service again

* Service register again using param constructor but repo bean will not register
* Injection happen again

As soon as we call getBean for repo

* Nothing is registered will get same repo object

**So REPO instance is same throughout but serive instance changes when we call getBean and injection happen again.**

**Scenario 3 : Service default and Repo prototype**



When we execute app context then

* It will first try to register the service by calling param constructor hence
* Repo register first
* Service register
* Injection

When we call getbean for service then just we get Bean of service injected with repo

When we call getbean again for service then just we get Bean of service injected with repo

When we call getbean for Repo then

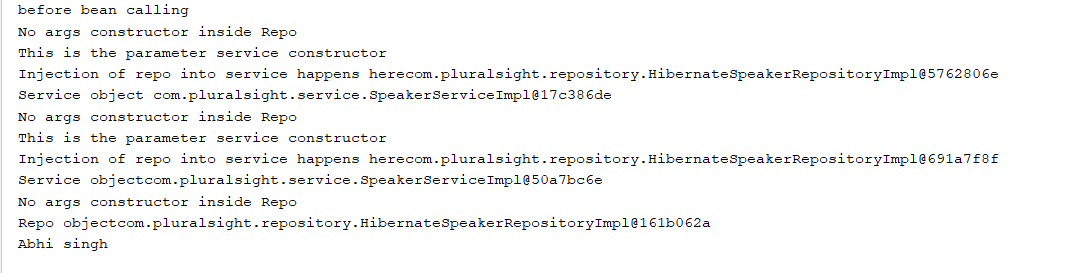
* Repo bean register again

**Service instance same, repo instance will change when we call getBean again.**

**DIFFERENCE BETWEEN scenario 1 and 3 is**

**OUTPUT looks the same but internally REPO registered in first place because we have service to register but in scenario 1 it has to register repo.**

**Scenario 4: when both are prototype**



**When we execute app context then nothing is registered cause both are prototype**

**When we call getBean for service then**

* **REPO registered**
* **Service registered**
* **Injection**

**When we call getBean for service again then**

* **Repo registered again**
* **Service registered again**
* **Injection happen again**

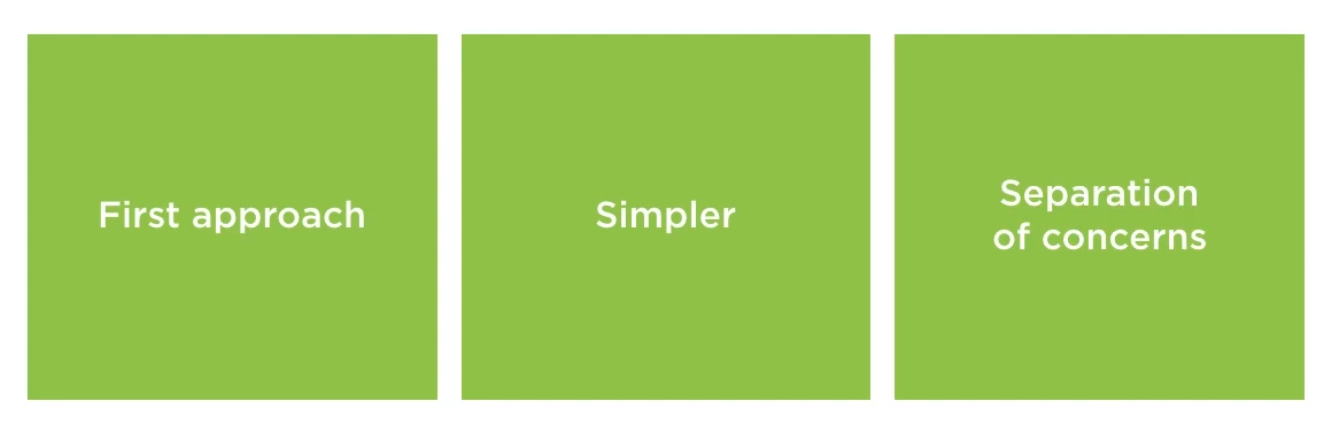
**When we call getbean for repo**

* **Repo registered again**

**EACH TIME SERVICE AND REPO IS REGISTERED AGAIN AND NEW INSTANCE CREATED.**

**SPRING FRAMEWORK USING XML**

**WHY XML?**



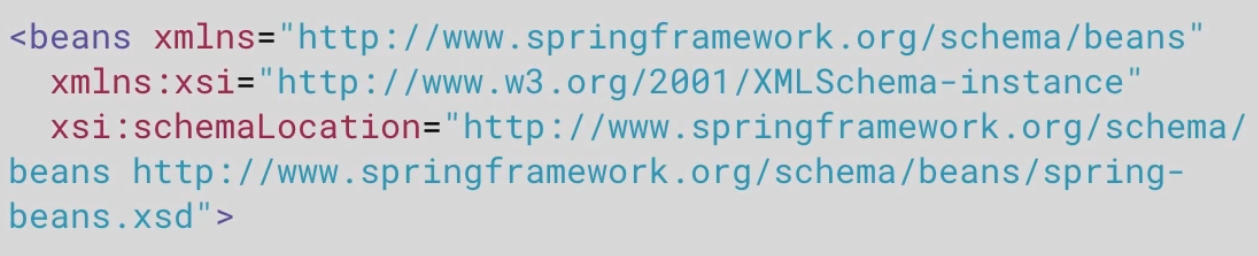
* First approach in spring
* Things are simpler
* Separations of concern – It happens naturally and organically when configuration code are written in separate file.

**APPLICATIONCONTEXT.xml**

This is a root of application configured with spring. Just like **AppConfig.java** in case of java configuration.

* Name doesn’t matter we can write app-context.xml any name.
* A simple view of spring is it is a hashmap of objects. We define that hashmap in applicationcontext.xml or appconfig.java. Objects here are just name-value pairs.
* Can be used as simple registry and we can be used to lookup our bean.
* XML configuration begins with this file.
* Namespace aid in configuration.

**NAMESPACES**

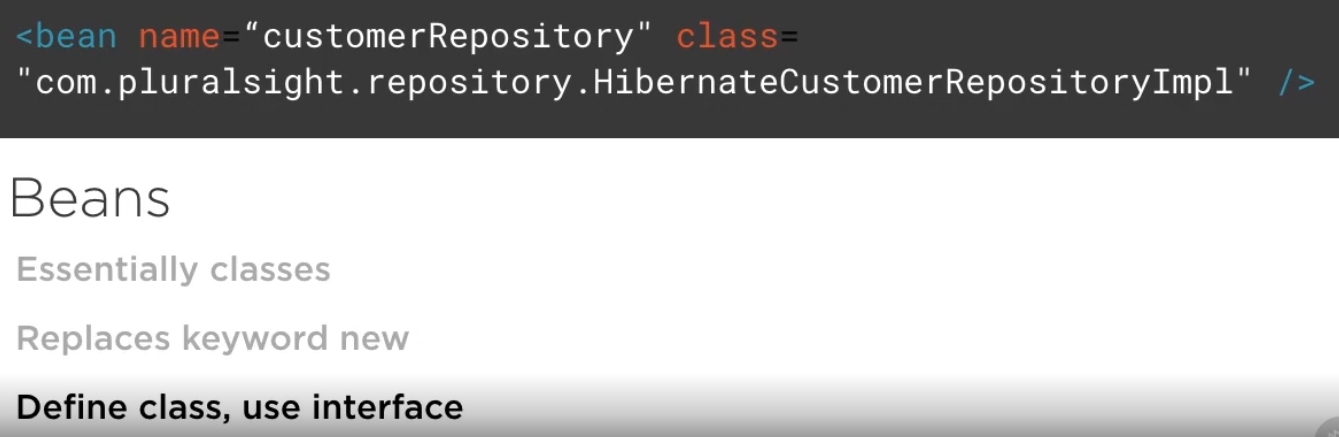


**Namespace is used as a dictionary for the properties that we use to create a Bean inside our application.**

Root namespace of xml is <beans>



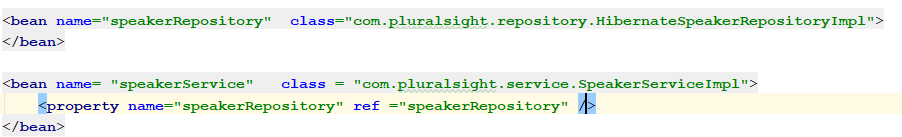
XML Declaration is use to define a Bean in XML.



**We always create a Bean with implementation class but we use in our application using Interface.**

**SETTER INJECTION**

* Create a applicationContext.xml in the resources directory so that when Maven compiles it will be created in target-classes and from where application.class will load from classpath.
* Inside applicationContext.xml we create beans with the help of <bean> namespace



* As we are injecting repo bean inside service bean using setter hence while writing bean for service we define property [ this is same as setter we define (case sensitive)]

And then ref means which bean we want to inject. If we don’t give correct name as of property then below error will come.

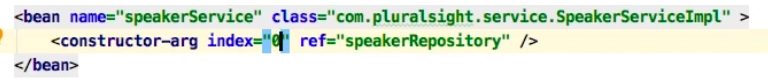


**CONSTRUCTOR INJECTION**

* We can use constructor and setter injection together in our code.
* But setter injection is better for existing code.
* Constructor injection guarantees a contract. We have our code and contract defined while creating an object because of constructor.
* Negative is we have to define our constructor.
* Constructor injection is index based and not named based.

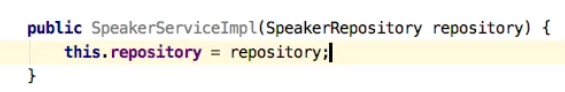
STEPS TO DO CONSTRUCTOR INJECTION:

1. As it is index based, inside applicationContext.xml we have to put the index



Index=0 means we are creating object using a constructor having 1 argument.

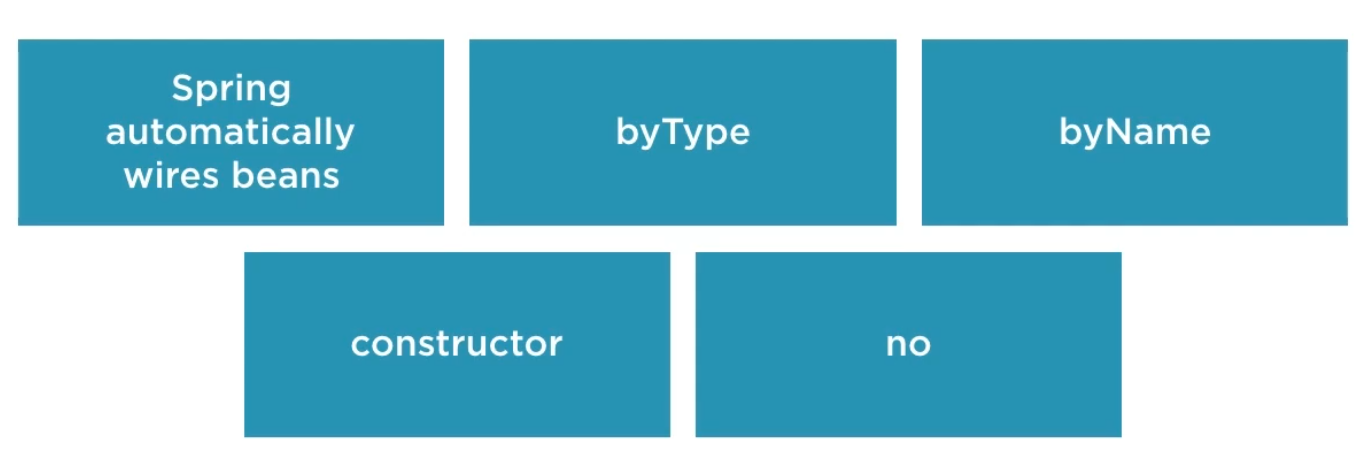
1. Inside our service class we have to add the constructor with 1 argument.



**AUTOWIRE**

* **Earlier it was a tedious task to wire up all the beans manually, if we have 100 beans and some are injected in others than doing all these manually is tedious. Hence to do injection implicitly Autowiring is introduced.**

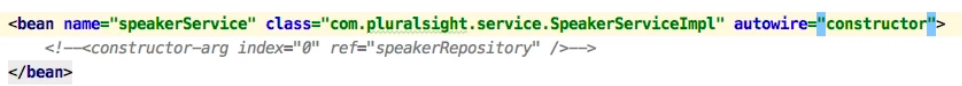
**Basically, we have 4 types of Autowiring possible:**



**Switching application from normal to autowire in XML**

**Using constructor**

* **Just go to the appContext.xml and comment the manual injection code.**
* **Go to Bean definition of service and autowire by constructor.**



* **We don’t have to worry about byName or byType when we are Autowiring by constructor because of the location.**

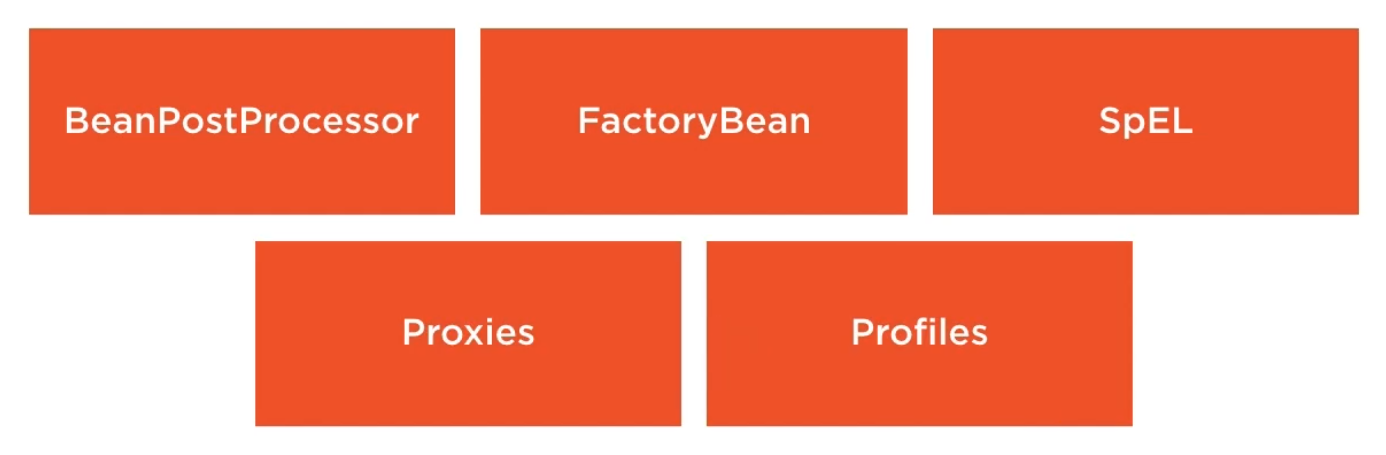
**Using Setter**

* **By Type**
* **When we do byType the name of the setter doesn’t matter.**
* **When we do byType it will just see if we have a setter in service to inject the repo bean and do we have a repo bean in appContext.xml**

**By Name**

* **When we do byName then the name of the setter should be same as what we have for bean of that class.**

**ADVANCED BEAN CONFIGURATION**

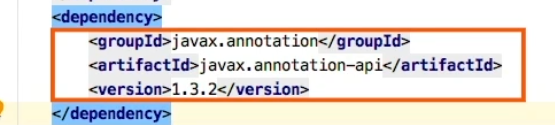




**initMethod() –** It is nothing but a **postconstruct call**, after all beans are created and constructor is called if we want to put some logs or configuration code we can create init() and put it here.

We should not use postconstruct to open or close connection to the database etc.

**Add a dependency**



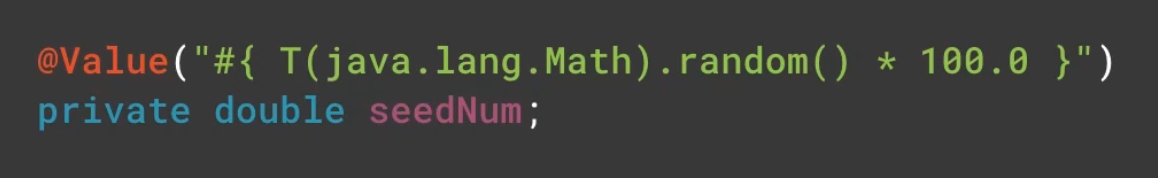
**FACTORY BEAN**

**Will see later**

**SpEL (Spring Expression language)**

It is a powerful expression language from spring framework, it is use to seed/salt values like password.

* Mostly used in libraries.
* Can be used to manipulate objects.
* Evaluate at runtime
* Evaluate or manipulate the configuration.
* Powerful tool to grab a random number.



**AOP proxies**

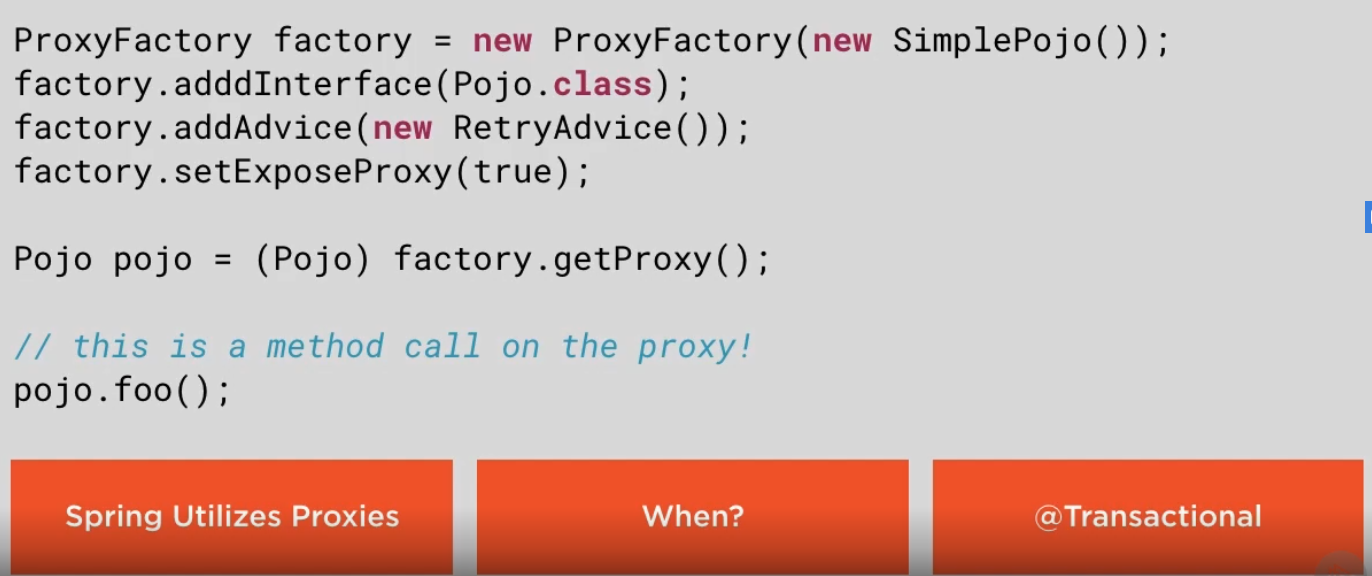
Aspect oriented programming- deals with cross-cutting concerns(not core business logic but part of many code modules like logging, security ,error handling, transactions)

With AOP, you keep your core business logic separate from concerns like logging, making your code cleaner, more modular, and easier to maintain.

**Aspect,advice and weaving**

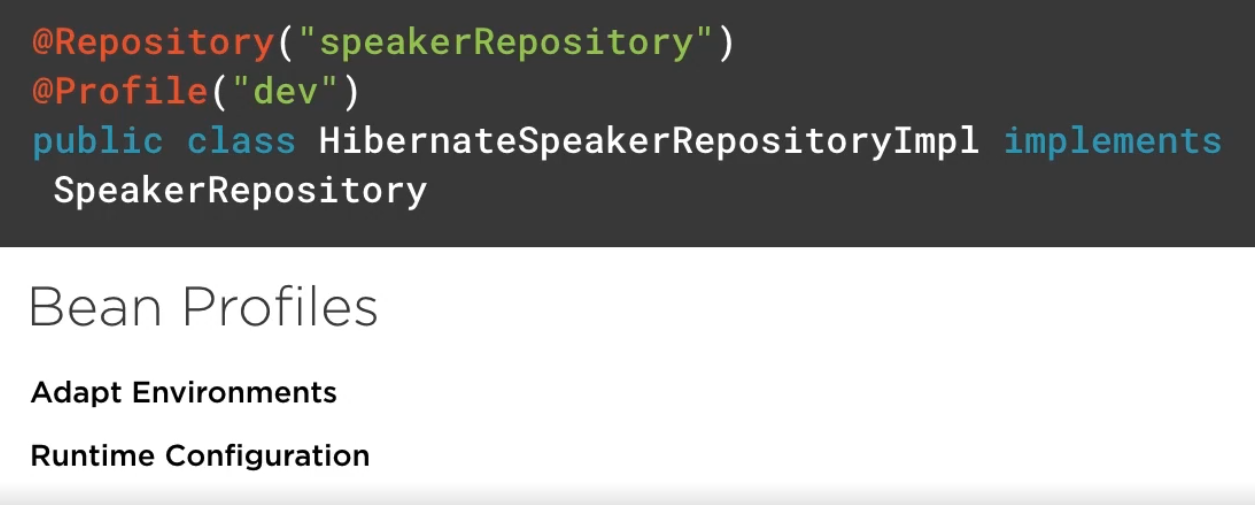
**Weaving:**

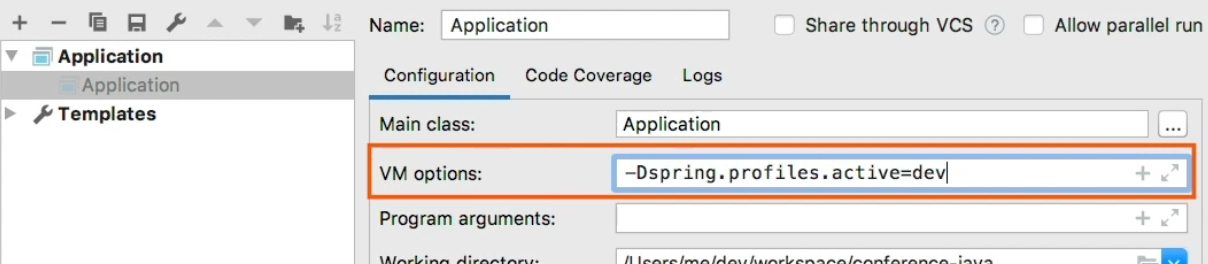
* Weaving is the process of integrating the aspect code with the main business logic. This can happen at different times, such as compile-time, load-time, or runtime.
* **Spring utilizes AOP proxies.**
* **Proxies are great way of injecting behavior into our code without modifying existing one.**



**BEAN PROFILES**

* Help us to adapt with different environment while working.
* Allow us to setup specific code that run in specific environment.
* We can swap out configuration at runtime.
* If we are dealing with sensitive data or sample data and we just want to run for dev env and not prod we can use profiles to do that.





**Need to change the run configuration like above. Don’t put in env variable it should be in VM option.**

**NEXT SPRING MVC – Bryan Hansen**

**Springboot fundamental – Dank bunker**

**HOW SPRING WILL KNOW WHICH BEAN TO IMPLEMENT IF THERE ARE MULTIPLE BEAN implementing same interface:**

The **@Qualifier** annotation in Spring is used to specify which bean should be autowired when multiple beans of the same type are present.

consider a scenario where you have multiple beans that implement the same interface. When you autowire the interface type into a component, Spring may not know which bean to inject by default because there are multiple candidates. In such cases, you can use **@Qualifier** along with **@Autowired** to specify the name or qualifier of the bean to inject.

