**Spring**

**Few basics things to know:**

**1. JAR (Java archive) File:** Basically, ek project ya classes ka zipped/compressed form hota hai Jar file.

Jb .**java** files are compiled into bytecode which are created with **.class** extension. Jb apne project ko compress krte hai to ye .class compress hote hai and Jar file form ho jaata hai. Jar files ko online download krke ya khudke se create krke uske classes and all chizo ka use kr sakte hai in our project.

**2. Maven Project:** Project Management tool hai jo ki bahut saare jar files internet se hame laake de deta hai jiske jarurat hame project development me lagega. Ye dependency management me bahut kaam aata hai. Better create a Maven project than normal java project. Many Enterprise level companies are built using it.

**3. Configuration file/xml file:** Here we create beans using which we initialize the fields of a class (aage samjhega)

**Dependency Injection**

It’s a design pattern which is used to develop an application.

Suppose Class B requires class A so what we usually do is create an object inside class B using new Keyword.

It’s not a good practice since our application becomes Tightly coupled. So what Spring does is it takes the control of creating an object in it’s own hand and frees the coder from the job of creating an object. This idea is known as a **Inversion of Control** and iske Implementation of bolte hai Dependency Injection. Ek class B, class A pe depend kr raha tha and Spring ne Inject kr diya class A ko class B me.

**Dependency Injection** can be done using **1. Setter Injection/Property Injection** (uses Setter functions of a class) **2. Constructor Injection** (uses Constructor of a class)**.**

Aage hai mentioned kaise krte hai

**Spring IoC Container**

It is a predefined program which we get with the Spring Framework.

It is responsibe for 3 things

1. Object Creation
2. Holding them in memory
3. Injecting the object in another object as required.

Basically complete lifecyle of a object from creation to destruction is managed by IoC container.

IoC container requires 2 things 1. Beans (java class) 2. XML Configuration

This containers takes help of XML config jo batata h konsa bean kis bean pe dependent hai,etc.

Now our Application Code can use these object apne jarurat ke hisab se.

**ApplicationContext** is a Interface which represents IoC Containter

Obviously ye ek Interface hai to jo class isko implement krta hai uska help lete hai to use IoC containter.

3 classes isse Implement krti hai.

1. ClasspathXmlApplicationContext (mostly used)
2. FileSystemXMLApplicationContext
3. AnnotationConfigApplicationContext

IoC Container can initiaize/handle the following Datatypes (Depdendencies)

**1. Primitive Data type**

**2. Collections**

**3. Reference type(user defined data type).**

**Some basic things to do while creating java project.**

1. FIrst create a maven project.

2. After creating a maven project pom.xml me spring core and spring context ke depedencies add krde inside <dependencies> </dependencies> tag

Is se basically hm dependency injection available karate hai hum in our project from. Basically ye dependencies add karte hi download ho jaate hai saare Dependencies neccessary to perform Dependency injection.

**Itna karlo and your ready to use Spring.**

**Adding Spring Dependencies and Creating configuration file**

1. Create a class (bean). Make sure that class has setter functions (neccessary for setter Injection) and Construtor (necessary for Constructor Injection).

2. Create a xml file (can name it anything and better keep it in resources).

3. From Spring documentation copy the basic structure of XML based configuration metadata and paste in your configuration/xml file.

Niche hai utha le.

<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xmlns:p="http://www.springframework.org/schema/p"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd  
 http://www.springframework.org/schema/context  
 http://www.springframework.org/schema/context/spring-context.xsd">  
 <bean class = "" name = "">  
<!--code-->  
 </bean>  
   
 <bean class = "" name = "">  
 <!--code-->  
 </bean>  
   
</beans>

Note:

This must be present for p schema (aage pata chalega)

xmlns:p="http://www.springframework.org/schema/p"

This for c schema

xmlns:c="http://www.springframework.org/schema/c"

**Setter Injection**(for primitive types)

<!-- class tag: complete path of our class-->  
<!-- name tag: name of the bean-->  
 <bean class="com.springcore.Student" name="student1">  
<!-- property tag se Setter Injection hota hai -->  
<!--name tag me we specify variable/field ka naam of that class. value tag me value which we wanna assign it.-->  
 <property name="studentId">  
 <value>1</value>  
 </property>  
 <property name="studentName">  
 <value>Pano meri</value>  
 </property>  
 <property name="studentAddress">  
 <value>Mumbai</value>  
 </property>  
 </bean>

<!--Another simplier way(also known as Value as Attribute)-->  
 <bean class="com.springcore.Student" name="student2">  
 <property name="studentId" value="2"/>  
 <property name="studentName" value="Bano Meri"/>  
 <property name="studentAddress" value="Kolkata"/>  
 </bean>

<!--More simpiler way using p schema-->  
 <bean class="com.springcore" name="student3" p:studentId="3" p:studentName="Tano meri" p:studentAddress="Ranchi"/>

Now our bean and configuration file ready. Now we can create IoC container wherever need to create an object of that bean in required java file.

// Creating Spring IoC Container.  
 ApplicationContext context = new ClassPathXmlApplicationContext("config.xml"); // para -> location of the xml file resources folder me stored tha to sirf xml file ka naam diya hai.  
 // We can pass single xml file as string(like above) ya phir if we have multiple xml file we can pass the array.

// Asking the container for the object.  
 Student student1 = (Student) context.getBean("student1");

// or we can do  
// Student s1 = con2.getBean("student1",Student..class);

// context.getBean("student1"); // student1 is name of the bean. With this code we the get the object from the container and we typecast it to class (here our class name was Student). Ab object bn gaya hai using Spring IoC container kr jo krna hai

**Setter Injection** (for collection types)

<bean class="com.springcore.ioccontainer.Emp" name="emp">  
 <property name="name" value="Gano"/>  
  
<!-- For List (note Set ka bhi same bss list hata ke write set-->  
 <property name="phoneNo">  
 <list>  
 <value>9819762521</value>  
 <value>8819762521</value>  
<!--We can also store null as mentioned below-->  
 <null/>  
 </list>  
 </property>  
  
   
<!-- For Map-->  
 <property name="courses">  
 <map>  
 <entry key="java" value="2 months"/>  
 <entry key = "C++" value="3 months"/>  
 </map>  
 </property>  
 </bean>

**Setter Injection** (for reference types)

Consider class A is dependent on class B. A has reference of B.

We need to first create bean of B and then Bean of A.

<bean name="bref" class="com.springcore.setterinjection.B">  
 <property name="y" value="5"/>  
 </bean>  
  
 <bean name="aref" class="com.springcore.setterinjection.A">  
 <property name="x" value="12"/>  
 <property name="ob">  
 <ref bean="bref"/>  
 </property>

<!-- Or Instead we could have used value as attribute -->  
 <property name="ob" ref = "bref"/>  
 </bean>  
  
<!--using p schema -->  
 <bean name="aref" class="com.springcore.setterinjection.A" p:x="134" p:ob-ref="bref"/>

**Constructor Injection**

<bean name="certificateRef" class="com.springcore.constructorinjection.Certificate">  
**<!-- First constructor-arg tag will initialize first para of the constructor, second waala second para ko and so on -->**  
 <constructor-arg>  
 <value>5 Gold</value>  
 </constructor-arg>  
  
<!-- Or we simply could have used value as attribute -->  
 <constructor-arg value="5 gold"/>  
 </bean>  
  
<!-- Or we could have used c schema-->  
 <bean name="certificateRef" class="com.springcore.constructorinjection.Certificate" c:a="5 gold"/>  
  
 <bean class="com.springcore.constructorinjection.Person" name="pers">  
<!-- for primitive type -->  
 <constructor-arg value="Zano"/>  
 <constructor-arg value="178"/>  
  
<!-- for collectoion -->  
 <constructor-arg>  
 <list>  
 <value>5</value>  
 <value>34</value>  
 <value>25</value>  
 </list>  
 </constructor-arg>  
  
<!-- for reference type -->  
 <constructor-arg ref="certificateRef"/>  
 </bean>

**Ambiguity problem and it’s solution**

Consider a bean jisme 2 para passed hai to the constructor.

<bean class="com.springcore.constructorinjection.AmbiRes" name="ambi">  
  
 <constructor-arg value="13"/>  
 <constructor-arg value="178"/>  
 </bean>

What spring does is, it first considers the value passed as String and search krta hai us Constructor ka jisme 2 String parameter ho.

Mila to value assign kr deta hai and nhi mila to phir class me Upar se niche aate hue search karega us Constructor ka jisme 2 para ho and khud hi typecast kr deta hai.

public class AmbiguityResolution {  
 private int a,b;  
  
 // Agar String waale Constructor na present ho to ye call kar dega.Upar se niche aata hai Spring and ye upar hai  
 public AmbiguityResolution (double x, double y){  
 a = (int)x;  
 b = (int)y;  
 }  
  
 // Ye present h to ye Spring sb ignore karke ye Constructor call kr dega  
 public AmbiguityResolution (String x, String y){  
 a = Integer.*parseInt*(x);  
 b = Integer.*parseInt*(y);  
 }  
  
  
// Isko call hone ke liye type tag me int mention karna hoga  
 public AmbiguityResolution (int x, int y){  
 a = x;  
 b = y;  
 }  
}

3rd constructor ko call krna hua to hame **type tag me int mention** krna hoga.Niche bagh

<bean class="com.springcore.constructorinjection.AmbiguityResolution" name="ambi">  
 <constructor-arg value="34" type="int"/>  
 <constructor-arg value="178" type="int"/>  
</bean>

**Note**: We can explicitly decide konsa constructor-arg tag kis para ko initialize karega by using **index tag**. Niche bagh

<constructor-arg value="34" type="int" index="1"/>  
 <constructor-arg value="178" type="int" index="0"/>

Indexing 0 based hai (obviously). 178 para 1 ko milega and 34 to the 2nd .

**Life Cycle methods of Bean**

Spring provides 2 important methods to every bean.

**1. public void init()**

**2. public void destroy()**

**init()** function me we can write initialization code,loading config, connecting-db, webservices, etc.

**destroy()** function me we can write the clean up code.

**Life cycle of a bean**

1. IoC container initializes the object.
2. Container assign value to the the fields/properties
3. Container calls init method
4. User uses the object created.
5. Container calls destroy method
6. Bean destroyed.

**Configuration (khuch likhna) of init and destroy function.**

3 Techniques using

**1. Xml 2. Spring Interface 3. Annotations**

**1.** using **Xml**

Inside Samosa.java file

package com.springcore.beanlifecycle;  
  
public class Samosa {  
 private int price;  
  
 public int getPrice() {  
 return price;  
 }  
  
 public void setPrice(int price) {  
 this.price = price;  
 System.*out*.println("Assigning value of the field");  
 }  
 public void init(){  
 System.*out*.println("Inside Init method");  
 }  
 public void destroy(){  
 System.*out*.println("Inside Init method");  
 }  
}

Inside Xml file.

<bean class="com.springcore.beanlifecycle.Samosa" name="samo" **init-method="init" destroy-method="destroy"** p:price="12" />

<!-- We obviously could have used the other 2 ways to initialize the value -->

We can give name anything to the init and destroy method. Bss xml file me us function ka name init-method and destroy-method tag me mention kr dena.

Inside java file with main class

public class Test {  
 public static void main(String[] args) {

// init method is invoked with ApplicationContext interface but for destroy method we create Object of one of the parent class of ApplicationContext named AbstractApplicationContext

AbstractApplicationContext context = new ClassPathXmlApplicationContext("ambiRes.xml");  
 Samosa sam = (Samosa)context.getBean("sam");  
 System.*out*.println(sam.getPrice());  
  
 // This Class has a registerShutdownHook() method which calls our destroy method.  
 context.registerShutdownHook();  
 }  
}

**2. Spring Interface**

We need to implement our class with **Initializingbean** interface which has **afterPropertiesSet** abstract method which does the work of init function.

And Implement the class with **DisposableBean** Interface which has **destroy** abstract method which does the work of destroy function.

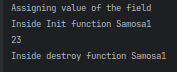
public class Samosa1 implements **InitializingBean, DisposableBean** {  
 private int price;  
  
 public int getPrice() {  
 return price;  
 }  
  
 public void setPrice(int price) {  
 this.price = price;

System.*out*.println("Assigning value of the field");  
 }  
 **@Override  
 public void afterPropertiesSet() throws Exception {  
 System.*out*.println("Inside Init function Samosa1");  
 }  
  
 @Override  
 public void destroy() throws Exception {  
 System.*out*.println("Inside destroy function Samosa1");  
 }  
}**

Configuration file

<bean class="com.springcore.beanlifecycle.Samosa1" name="samo1" p:price="23" />

main function waale class me vahi 1. xml tecnique ki tarah object create kar.

Output bean life cylce yaha follow ho rha h

**3. Using Annotation**

We use Annotation **@PostConstruct** and **@PreDestroy** for init and destroy function.

Note: both annotation are part of a JavaEE which is now deprecated. So we need to put it’s dependencies in pom.xml. Niche pada hai utha le.

<dependency>

<groupId>javax.annotation</groupId>

<artifactId>javax.annotation-api</artifactId>

<version>1.3.2</version>

</dependency>

**Note:** By default all the annotations are disabled so we need to enable all the annotations in the xml file.

Inside xml file.

<!-- This way we enable all annotations -->  
 **<context:annotation-config/>**

<!-- Below bean info -->  
 <bean class="com.springcore.beanlifecycle.Samosa2" name="samo2" p:price="34" />

Inside java file

package com.springcore.beanlifecycle;  
  
import javax.annotation.PostConstruct;  
import javax.annotation.PreDestroy;  
  
public class Samosa2 {  
 private int price;  
  
 public int getPrice() {  
 return price;  
 }  
  
 public void setPrice(int price) {  
 this.price = price;  
 System.*out*.println("Assigning value of the field");  
 }  
  
 **@PostConstruct  
 public void start(){  
 System.*out*.println("Inside Init method");  
 }  
  
 @PreDestroy  
 public void end(){  
 System.*out*.println("Inside Init method");  
 }**  
}

And now bss same shit like type 1 and 2 in main function waala class and done

**Autowiring**

It’s a feature of Spring Framework in which spring container inject the dependencies / objects automatically.

Though it **works only and only for reference types.**

Creating object of one class in another class is known as wiring.

Above using bean tag we used do that manually hence termed Manualwiring.

But if spring container automatically creates a object then we call it **Autowiring.**

**Autowiring** can be done in 2 ways

1. **using XML** (types :- byName, byType, constructor)

2. **Annotations**

Advantages of Autowiring

It is automatic and requires less coding.

Disadavntages

Can’t be used for types other than reference types,

**1. Autowiring using XML**

Consider kr **Employee** class jisme we have declared reference of Address and Education class named as **address** and **education** respectively (Teeno class me setter, getter toString sb present hai obviously).

See below XML file

<!-- address bean ka value initialization -->  
 <bean class="com.springcore.auto.wire.Address" name="address">  
 <property name="street" value="SN Dubey"/>  
 </bean>  
   
 <!-- address bean ka value initialization -->  
 <bean class="com.springcore.auto.wire.Education" name="education">  
 <property name="edu" value="SE"/>  
 </bean>  
  
 **<bean class="com.springcore.auto.wire.Employee" name="emp" autowire="byName"/>**

**Autowiring byName** :

Last line dekh, we are saying in class named Employee jiske bean ka naam emp rakha hai.

By **autowire="byName"** we are saying Employee class me agar object declared hai of any of the bean specified in this xml file then initialize that object automatically.

Eg, above Employee class me Address and Education ke objects named address and education decleared hai and xml file me in dono name se bean bhi hai to inke objects ko initalize kardo automatically.

Pahle upar dekh pg 4 isme apn ref tag se kr rahe the and yaha automatically.

**Autowiring byType**:

**autowire="byName"**ye hata ke **autowire="byType"** likh du this means ab hm check karenge Employee class me agar kisi class ka Object declared hai and us class ka bean info xml file me present hai to object initialize karde.

Basically yaha bss class dekh rahe hai. Bean name khuch bhi usse koi issue nhi.

**byType avoid kar.**

Reason: Consider kr 1 class ka 2 bean banata and byType krta to Spring to bss class ka naam dekhta and confuse ho jaata object ko pahle bean se initialize karu ki dusre se and Exception throw kr deta.

<bean class="com.springcore.auto.wire.Education" name="education">  
 <property name="edu" value="SE"/>  
</bean>  
<bean class="com.springcore.auto.wire.Education" name="education1">  
 <property name="edu" value="SE"/>  
</bean>  
  
<bean class="com.springcore.auto.wire.Employee" name="emp" autowire="byType"/>

Ye exception throw kar deta.

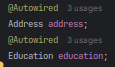
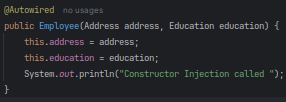
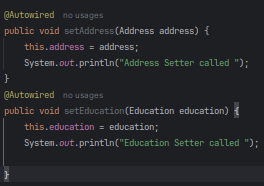
**Note: Autowiring byName and byType uses Setter Injection.**

**Autowiring using constructor:**

**autowire="constructor"** --> Autowiring using byName hi karega bss setter injection nhi constructor Injection karega

**2. Autowiring using Annotation**

It Simplest and mostly used. We just need to mention @Autowired annotation on any of these

**1. field 2. Setter functions** (for setter injection) **3. Constructor** (for constructor injection)

Bss make sure

1. Information of the bean is provided in the xml file.

2. All the annotations are enabled

Inside xml file.

<context:annotation-config/>  
   
 <bean class="com.springcore.autowiring.usingannotation.Address" name="address">  
 <property name="street" value="SN Dubey"/>  
 </bean>  
  
 <bean class="com.springcore.autowiring.usingannotation.Education" name="education">  
 <property name="edu" value="SE"/>  
 </bean>  
  
 <bean class="com.springcore.autowiring.usingannotation.Employee" name="emp"/>

Dekh emp bean me reference type ke liye khuch alg se nhi likha (Employee class me Address and Education ka object is declared)

**Note: Annotation byType intialize krta hai reference types ko.**

byType basically class tag ke andar mentioned class ko dekhta hai, bean name se fharak nhi isko.

Bean name and reference type ka name khuch aur hota to fhark nhi pdta.

**Note:** byType ke related issue to pata hi hai.

**To avoid that after @Autowired annotation write @Qualifier(’beanName’) annotation.**

Assuming multiple beans hai of a single class in our xml file, ye to exception de dega.

To avoid this error after @Autowired annotation we specify which bean should be used to initialze the object using @Qualifier annotation by passing the name of the bean as a parameter to it

**Note: If using @Qualifier then make sure either no constructor or default and parameterized constructor is present aisa nhi kiya exception throw krta h.**

**Standalone Collections**

kisi bhi collections (like list,set,etc) ke values alg se xml me configure krke rakhna and jb jarurat ho to usko refer krna.

Maan le ek class **A** jisme LinkedList ka variable present hai.

xml file me class **A** ke mutliple beans declared hai and saare me ya khuch me linkedList ke saare values sb hai. Ab baar baar same code hr bean likhne se acha ek bean alg se hota jisko refer kr lete bss and baar baar same likhne se bach jaate. Yahi hai Standalone collection

Eg code, consider class PersonalInfo which has List and Map variable declared. See the xml file for it,

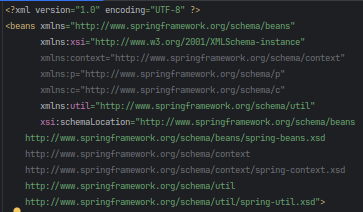
<!-- Creating Standalone Collection -->  
<!-- Make sure to specify the exact collection type in list-class tag -->  
<!-- id tag: name given to the standalone collection -->  
 <util:list list-class="java.util.LinkedList" id="friendsName">  
 <value>Aano</value>  
 <value>Bano</value>  
 <value>Cano</value>  
 </util:list>  
<!-- Now we can use this list anywhere in the list anywhere in this xml file.-->  
  
<!-- this for map (samjh gaya hoga kaise work krta hai util tag) -->  
 <util:map map-class="java.util.TreeMap" id="friendsAge">  
 <entry key="Aano" value="20"/>  
 <entry key="Bano" value="19"/>  
 </util:map>  
  
 <bean class="com.springcore.standalonecollections.PersonalInfo" name="person1">  
<!-- Now we will refer to the standalone collections instead of writing inside this bean -->  
 <property name="friends" ref="friendsName"/>  
 <property name="age" ref="friendsAge"/>  
<!-- reference type jaise inject hota hai vaise hi standalone collection bhi inject krte hai -->  
 </bean>

xmlns:util="http://www.springframework.org/schema/util"

Make sure ye depedendency present ho (khuch nhi **p schema** waale dependency me jaha jaha p tha usko hata ke **util** likh diya)

http://www.springframework.org/schema/util  
<http://www.springframework.org/schema/util/spring-util.xsd>”

And ye bhi present ho http ke sections me



(See pg 4th last and 3rd last me se **context** hata ke **util** likh diya).

**Stereotype Annotations**

Ab tk we have been creating beans in the xml file to create an object.

Using Stereotype Annotations we can avoid creating beans and instead use Annotations to create an object.

Basically, instead of creating bean we’ll use Annotations for object instantiaion.

Simply write **@Component** over a class.Spring container will know ki haa bhai mujhe is class ka object khud create krna hoga.

Here how it’s done

Inside xml file,

<!-- Allowing Sterotype Annotions to function -->  
<context:component-scan base-package="com.springcore.stereotype"/>

<!-- base-package tag tells where we want to allow StereoType Annotations to function.

Here we are allowing it in stereotype package and all it's subpackages -->ate krna hoga.

Inside Class with @Component Annotation

package com.springcore.stereotype;  
  
**@Component**  
public class Student {  
 **@Value("Tano") /**/ Initializing name to Tano whenever the object is created.  
 String name;

**@Value("22")** // Initializing age  
 int age;  
}

Note: We don’t unneccesarily need to create setter, getter and constructor here.

Inside Class with main function/where are creating the object.

package com.springcore.stereotype;  
  
public class Test {  
 public static void main(String[] args) {  
 ApplicationContext con = new ClassPathXmlApplicationContext("stereoConfig.xml");  
 Student stu = con.getBean("student",Student.class); /**/ Note: Name of the class is Student then default bean name is student.**

System.*out*.println(stu.name);  
 }  
}

If want to name the bean other than student write **@Component(“”)**. Instead of just writing @Component over a class. **Inside “” pass the name of the bean.**

**Initializing Collections with @Value Annotation**

For collections create Standalone collection in the same xml file.

and pass the **id** to @Value. Eg

inside xml file

<!-- To initialize collections need to create StandAlone collections -->  
 <util:list list-class="java.util.ArrayList" id="**nameofStudents**">  
 <value>Eano</value>  
 <value>Fano</value>  
 </util:list>

Inside Student Class

**@Value("#{nameofStudents}")** // Passed id name of the StandAlone collection  
ArrayList<String> studentNames;

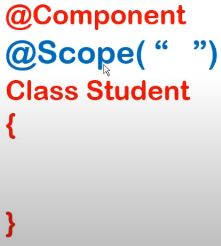
**Bean Scope**

Types of Bean Scope

**1**. **Singleton**: This is the default scope. Spring container ek hi object create krta h and maan le 10 object banaya jaye us class ka to Spring sabko same object dega.

**2. Prototype**: Each time you request a bean with this scope, Spring creates a new instance. So, every time you ask for the bean, you get a fresh copy. Bss object naya milega values jo initialized honge (either using xml or annotation) vo sabke same rahenge.

**3.Request 4. Session 5. Global Session:**  Ye deeno web development kaam aate hai.



We can either specify the scope in the configuration file



or

specifiy inside **@Scope(“”)** annotation. Pass the name of the scope inside ” ”. See =>

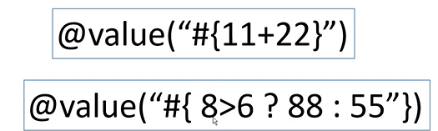
**Spring Expression Lanaguage (SpEL)**

It allows parsing and execution of expressions with the help of @Value annotation.

Basically jis tarah **@Value(“#{}”)** pg 13 collection pass kiya tha ussi tarah symbols (numbers,alphabets,etc), ternary operator, classes, variables, methods, constructors and objects khuch bhi pass kr sakte hai.

Jo bhi hm pass kar rahe hai we call that Expression here.

Syntax: **@Value(“#{expression}”)**



Eg, passing arithmetic operation and ternary operation

**Invoking Static method and variable**

****

**Syntax for method: Eg,**

**Syntax for Variable: Eg,**

**Object creation**

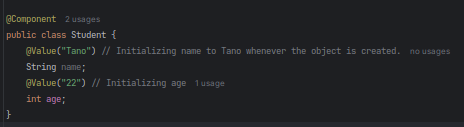
****

** Passing Boolean value:** By default to false hi store hota but to store True, Bss pass an expression which is true. Eg,

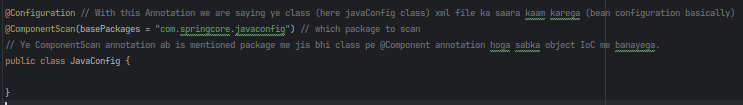
**Annotation Based Configuration**

Above we studied XML based configuration (annotation based bhi padha h but still xml file ka jarurat pada tha usme bhi). We’ll see bina xml file kaise things can still work for us.

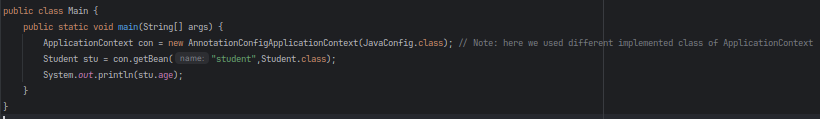
**Step 1:** Create class/classes with **@Component** annotation.



**Step 2:** Create a Java Class with Annotation **@Configuration** and **@ComponentScan(basePackages = “”)**. This class will do the work which our xml file was doing.



**Step 3**: Jis class me object ke jarurat h us class me create ApplicationContext Interface ka object.



Bss ho gaya

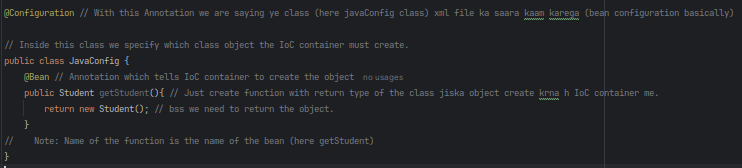
**This way of Configuration is known as Java Configuration**

We can do java configuration another way

**Step 1:**  Create your class (suppose we created Student class)

**Note:**We do not need to put **@Component** annotation over the class

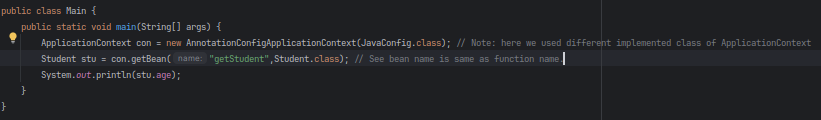
**Step 2:** Create a Java Class with Annotation **@Configuration.** Inside this class create functions which will only return object. Ye objects IoC container me create ho jaayega.



**Note:** We don’t need to put **@ComponentScan()** annotation over our Class

**Note:**  Instead of **@Bean** if we write **@Bean(name={“student”, “s1”, “xyz”}**. Ye saare we can use as name of of bean while passing to .getBean().

**Step 3:** Same as previous step



**Spring JDBC**

Spring JDBC internally uses JDBC. Everything is similar to JDBC but much than JDBC.

**Problem of normal JDBC**

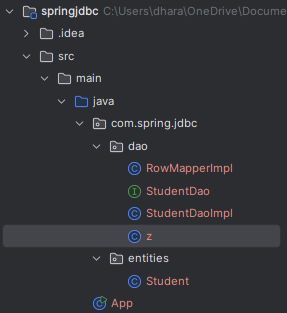
1. Need to write same code again and again (Connection, statement, exection and closing the connection is required everytime.

2. Need to write all JDBC code in try catch block since it throws SQL Exception.

Spring JDBC solves all these problems.

Spring JDBC provides JdbcTemplate class which has all the important methods required to work with the db.

**Note: Make sure to add the dependency of springjdbc and mysql connector in pom.xml** (bean and and context ka dependency to chahiye hi chahiye)**.**

**First Know the Design Pattern of of DAO in Real Time Application**

1. Inside your project package create a **dao** and **entities** (can name entities package anything) package.
2. Entities package me keep the pojos (classes with setter getter and toString methods).
3. Inside dao package first create Interface with all your required abstract function. For eg, here Student pojo ke liye StudentDao interface hai jaha necessary abstract functions declared h.
4. Now create Impementation class of that Interface and yaha saare codes likhenge of fetching and sending from db. Here StudentDaoImpl implements StudentDao.

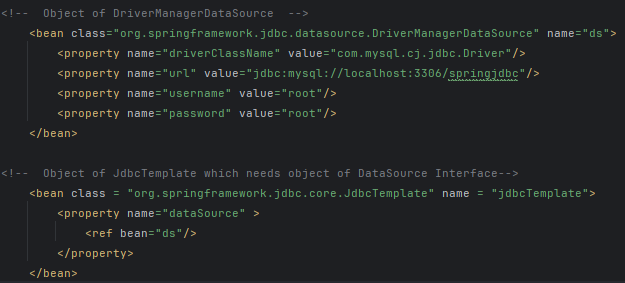
**Aage Sab samjhega.**

**JdbcTemplate Class:**  We create object of this class which need object of DataSource Interface.

**DataSource Interface:** It has fields which helps us to connect with the Database

DriverManagerDataSource class implements DataSource, so we’ll use DriverManagerDataSource

See How we did it using XML (Aage we’ll how it’s done using



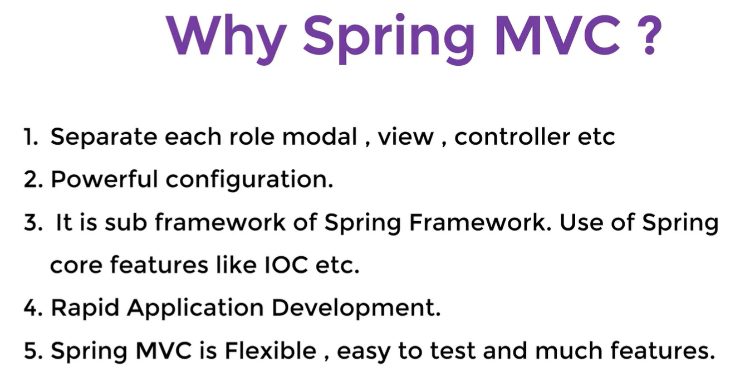
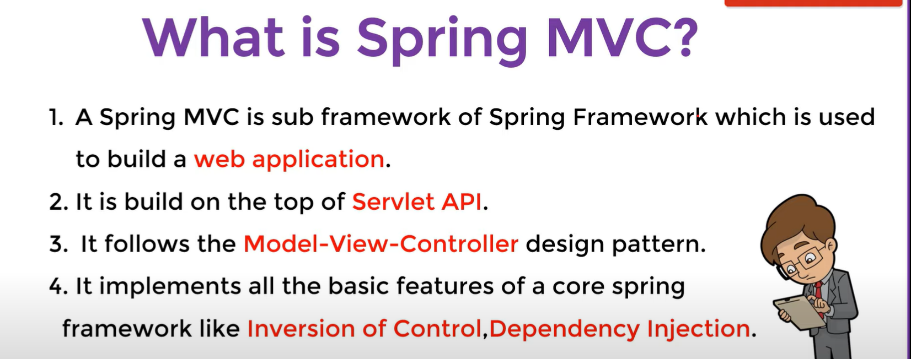
**Spring ORM**

When you use @Entity over a class, it tells the ORM framework (like Hibernate or JPA) that this class should be mapped to a table in the database, and its fields should be mapped to the columns of that table.

In short, @Entity makes the class ready for database operations through ORM.

**SPRING MVC** (Modal View Controller)

Used to create web application.

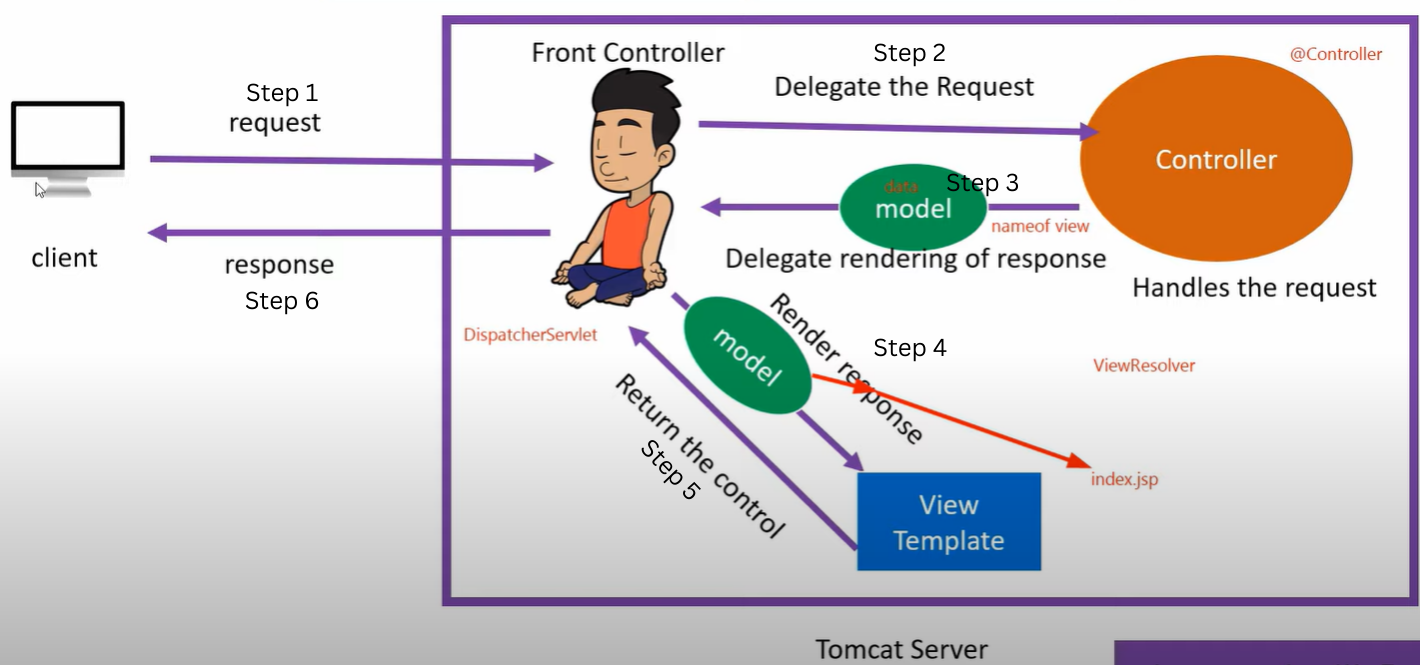


**MVC** is a design pattern (way to organize our application code).

**MVC stands for Model-View-Controller.**

* **Model:** Manages the data and business logic.
* **View:** Displays the data (user interface).
* **Controller:** Handles user requests and controls the flow between Model and View( mtlb data ko kaise kaha kis tarah use krna taki desired output display kar sake).

**Working of MVC**



MVC follows client-server architecture where client can be our browser and the server internally does the necessary task and responds to the client request and return something to the client.

1. **User Sends a Request**: The user interacts with the web application by clicking a link or submitting a form.
2. **DispatcherServlet Receives the Request**: The request is first handled by the **DispatcherServlet**, also known as the Front controller in Spring MVC.
3. **DispatcherServlet Routes the Request**: The DispatcherServlet determines which **Controller** should handle the request and forwards it to that Controller.
4. **Controller Processes the Request**: The **Controller** handles the request. It may interact with the **Model** to fetch or update data.
5. **Model Provides Data**: The Model returns the necessary data to the Controller.
6. **Controller Chooses the View Template**: The Controller decides which **View Template** (e.g., JSP, Thymeleaf) should be used to present the data.
7. **DispatcherServlet Passes Data to the View Template**: The DispatcherServlet sends the data to the selected View Template.
8. **View Template Renders the Data**: The View Template processes the data and generates the final HTML (or other format) that will be displayed to the user.
9. **User Sees the Result**: The user sees the final web page, which is the result of the View Template rendering the data.