# How to change Java version for project in sts tool?

Right click on project --> build path --> configure build path --> select libraries tab --> select JRE --> Edit --> add java version which is required.

Spring Overview:

**Spring framework** creates and manages object inside IOC container.

**Spring** has **IOC container** which reads the config file i,e beans.xml and creates object.

**Spring** Bean is nothing but the object which is created inside IOC container.

We can use IOC container in 2 ways

1. BeanFactoy (I)
2. ApplicationContext (I)

But ApplicationContext has more methods then BeanFactoy so it is best to use. ApplicationContext is called as FactoryDesignPattern.

ClassPathXmlApplicationContext implements ApplicationContext .

Dependency Injection is defined as injecting dependencies required for class.

2 Types we can inject values:

* 1. Constructor Injection
  2. Setter Injection

b) Setter Injection (we must define setters inside specific class)

<bean id = "vijay" class = "com.express.di.Student">

<property name="studentName" value="Vijay K" />

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

</bean>

Note: studentName and studentId are primitive types

a) Constructor Injection (we must define constructor inside specific class)

<bean id = "vijay" class = "com.express.di.Student">

<constructor-arg name="studentName" value="Vijay K"> </constructor-arg>

<constructor-arg name="studentId" value="1"></constructor-arg>

</bean>

Dependency Injection for object type:

When we want to inject Object Type

a)

<bean id = "studentObject" class = "com.express.di.Student">

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

<property name="mathCheat" >

<bean class="com.express.di.MathCheat"></bean>

</property>

</bean>

If we have same dependency object for 2 or more classes above approach not holds good. For example:

Student and AnotherStudent class has dependency on MathCheat Object

<bean id = "mathCheat" class="com.express.di.MathCheat"></bean>

<bean id = "studentObject" class = "com.express.di.Student">

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

<property name="mathCheat" ref="mathCheat" >

</property>

</bean>

<bean id = "anotherStudentObject" class = "com.express.di.AnotherStudent">

<property name="matCheat" ref="mathCheat">

</property>

</bean>

Autowire can be done in below ways

# autowire byName:

<bean id="heart" class="com.express.autowire.Heart"></bean>

<bean id="human" class="com.express.autowire.Human" autowire="byName"></bean>

Autowiring byName is possible when we have same reference name in java side and in

.XML file bean id name should be same.

# autowire byType:

<bean id="heartObject" class="com.express.autowire.Heart"></bean>

<bean id="human" class="com.express.autowire.Human" autowire="byType"></bean>

# autowire By using Constructor:

<bean id="heartObject" class="com.express.autowire.Heart"></bean>

<bean id="human" class="com.express.autowire.Human" autowire="constructor"></bean>

The above approach holds good when we have less dependency in class i,e when we have more dependency for a class, then we have to use

@Autowired annotation by using only java approach instead of defining in .XML file.

# @autowire usage by using constructor

1. **we should define default constructor.**

public Human() {}

Spring will be using default constructor for creating object instead of using --> Human(Heart heart).

If we don't specify default constructor we will be getting below exception:

# Instantiation failed we no default constructor found.

After specifying default constructor also we won't be able to get the output since it won't inject dependency. i,e DependencyInjection is off we need to make on.

# In order to activate @autowired in java side we have to specify

<beans xmlns=["http://www.springframework.org/schema/beans"](http://www.springframework.org/schema/beans) xmlns:x[si="h](http://www.w3.org/2001/XMLSchema-instance)ttp:[//www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance)

xmlns:context=["http://www.springframework.org/schema/context](http://www.springframework.org/schema/context)" xsi:schemaLocation="<http://www.springframework.org/schema/beans>

https:/[/www.springframework.org/schema/beans/spring-beans.xsd](http://www.springframework.org/schema/beans/spring-beans.xsd) <http://www.springframework.org/schema/context>

https:/[/www.springframework.org/schema/context/spring-context.xsd">](http://www.springframework.org/schema/context/spring-context.xsd)

<context:annotation-config></context:annotation-config> tag in .XML file

@Autowired

public Human(Heart heart) {

System.out.println("Constructor called"); this.heart = heart;

}

# @autowire usage by using setter methods

@Autowired

public void setHeart(Heart heart) { this.heart = heart;

}

# @Autowired works in following ways

Autowiring is not possible for primitive and string types it is possible only for object Types. First it looks to resolve byType if it doesn't find then it looks for byName

# The below example doesn't work since both are not matching

<bean id="heartObject" class="com.express.autowire.Heart"></bean>

<bean id="octopusHeart" class="com.express.autowire.Heart"></bean>

<bean id="human" class="com.express.autowire.Human"></bean>

# The below example will work since it resolves byName

<bean id="heart" class="com.express.autowire.Heart"></bean>

<bean id="octopusHeart" class="com.express.autowire.Heart"></bean>

<bean id="human" class="com.express.autowire.Human"></bean>

# When we have multiple objects and we want one object to use the specify @Qualifier

@Autowired

@Qualifier("humanHeart")

public void setHeart(Heart heart) {

System.out.println("setter method called"); this.heart = heart;

}

# In .XML file

<bean id="humanHeart" class="com.express.autowire.Heart">

<property name="nameOfAnimal" value="Human"></property>

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

</bean>

<bean id="octopusHeart" class="com.express.autowire.Heart">

<property name="nameOfAnimal" value="Octopus"></property>

<property name="noOfHeart" value="3"></property>

</bean>

<bean id="human" class="com.express.autowire.Human"></bean>

If we use @Autowired and @Qualifier before dependency we don't need to provide setters and constructor for class i,e

@Autowired

@Qualifier("humanHeart") private Heart heart;

# Note : \* @Autowire basically does only injecting the objects.

**SpringCore Annotations:**

For example in **.properties file** we have

student.name = vijay

student.hobby = cricket

If we want we can provide default values for properties then use below code:

Example:

@value("testing")

private String name;

# @PropertySource:

It is basically used for reading properties file i,e by using -->

@PropertySource("classpath:test.properties") specify on top of class.

Example:

@value("${student.name}") private String name;

# @required:

It is basically used for restricting user for not providing values for object.

Example:

@required

private String name;

# @Autowired:

\* Note : @Component basically creates the object thats it where as @Autowired injects the objects without using setter.

# @Component:

It is used for creating object in IOC container

For example: below code creates object of exam and stores in IOC container and Object referenceName will be exam.

It does same job as this-----> <bean id="exam" class="com.express.java"></bean>

Example:

@Component("exam") Public class Exam{

}

# In XML file write below code:

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns=["htt](http://www.springframework.org/schema/beans)p[://www.springframework.org/schema/beans"](http://www.springframework.org/schema/beans) xmlns:x[si="h](http://www.w3.org/2001/XMLSchema-instance)ttp:[//www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance)

xmlns:context=["http://www.springframework.org/schema/context](http://www.springframework.org/schema/context)" xsi:schemaLocation="<http://www.springframework.org/schema/beans>

https:/[/www.springframework.org/schema/beans/spring-beans.xsd](http://www.springframework.org/schema/beans/spring-beans.xsd) <http://www.springframework.org/schema/context>

https:/[/www.springframework.org/schema/context/spring-context.xsd">](http://www.springframework.org/schema/context/spring-context.xsd)

<context:component-scan base-package="com.express.college"></context:component-scan>

</beans>

i,e---> <context:component-scan base-package="com.express.college"></context:component-scan>

-----> will scan the package and find @component annotation

Hey spring we are not defining in XML file instead we are doing in java so scan the package specified.

# If we delete XML file how can we achieve?

@Configuration--> specify it is a configuration class

First define configuration class i,e

@Component

public class College{

}

@Configuration

@ComponentScan(basePackages = "com.express.college") Public config{

}

# inside main method write below code

ApplicationContext applicationContext = new AnnotationConfigApplicationContext(Config.class); College college = applicationContext.getBean("college", College.class);

// if we don't specify any name for component then by default name will be className System.out.println("college bean is created" +college);

college.test();

# 2nd way to create Bean

@Bean--> It is also used for creating object

public class College{

@Bean

public College collegeBean() { // collegeBean is beanId

College college = new College(); return college;

}

}

@Configuration

@ComponentScan(basePackages = "com.express.college") Public config{

}

# inside main method write below code 1st Approach:

ApplicationContext applicationContext = new AnnotationConfigApplicationContext(Config.class); College college = applicationContext.getBean("collegeBean", College.class);

System.out.println("college bean is created" +college); college.test();

((AnnotationConfigApplicationContext)applicationContext).close();

# 2nd Approach:

AnnotationConfigApplicationContext applicationContext = new AnnotationConfigApplicationContext(Config.class);

College college = applicationContext.getBean("collegeBean", College.class); System.out.println("college bean is created" +college);

college.test();

applicationContext.close();

If we are using @Bean annotation we don't need to specify @ComponentScan annotation.

**How does Proxy works in Spring:**

By Default all **beans** are **Singleton**.

**JDK Proxy:**

EmployeeSalaryInvocationHandler empHandler = **new** EmployeeSalaryInvocationHandler(employee);

Below line of code creates a JDK Proxy

IEmployee employeeProxyCreated = (IEmployee)Proxy.*newProxyInstance*(

ITEmployee.**class**.getClassLoader(),

**new** Class[] {IEmployee.**class**},

empHandler);

employeeProxyCreated.giveHike(5000);

Disadvantage of JDK Proxy:

------------------------------------------

JDK proxy will work only with interface it won't work with classes.

self-Invocation i,e when a method calls another method inside JDK proxy doesn't support it.

CGLib Proxy:

-------------------------------

To overcome disadvantage of JDK Proxy.

//creating product instance

Product product = new Product();

//creating proxy factory

ProxyFactoryBean proxyFactoryBean = new ProxyFactoryBean();

//giving the product to factory

proxyFactoryBean.setTarget(product);

//getting object from factory

// whenever we use class It returns a CGLIB proxy

Product productProxy = (Product)proxyFactoryBean.getObject();

productProxy.showProductInfo();

//But whenever we use interface it return JDKProxy

//IProduct productProxy = (IProduct)proxyFactoryBean.getObject();

Proof:

----------

Define config class (TestConfig)

@Configuration

public class TestConfig {

@Bean

public Test test() {

return new Test();

}

}

Inside main method() write below code:

-----------------------------------------------------

ApplicationContext context = new AnnotationConfigApplicationContext(TestConfig.class);

whenever we put debugger inside TestConfig class inside Test() it internally uses CGLIB Proxy How ?

What Spring does, It creates a new class called

public class TestConfigFiles$CGLib extends TestConfig {

@overide

public Test test() {

// it does certain checks

sup.test();

}

}

Internally all this will be done by Spring Framework.

2nd way of defining CGLIB:

public class RAM {

private String size = "64Gb";

private int price = 10000;

public String getSize() {

return size;

}

public void setSize(String size) {

this.size = size;

}

public int getPrice() {

return price;

}

public void setPrice(int price) {

this.price = price;

}

}

public static void main(String[] args) {

final RAM targetRam = new RAM();

Enhancer enhancer = new Enhancer();

enhancer.setSuperclass(targetRam.getClass());

//anonymous class

enhancer.setCallback(new MethodInterceptor() {

public Object intercept (Object obj, Method method, Object[] args, MethodProxy proxy) throws Throwable {

System.out.println("before invoking method");

Object ret = proxy.invokeSuper(obj, args); // which inturn calls actual method of RAM Class.

System.out.println(ret);

System.out.println("after invoking method");

return ret;

}

});

//creating CGLib Proxy

RAM ramProxy = (RAM) enhancer.create();

ramProxy.getSize(); // which will call anonymous method called --> intercept ()

ramProxy.getPrice();

//Internally spring is extending ur class and calling the method.

// Just make ur class as final it won't be working because then it won't able to extend class.

}

}

What Spring Framework does internally is:

----------------------------------------------------------------------------

//

//class RAM{

//

// void abc() {}

//}

//

//Class RAMProxy extends RAM {

//

// @overqride

// void abc() {

//

// sup.abc();

//

// }

//}

Imagine u have a class annotate with Singleton and that class has dependency of another class which has scope of Prototype what will happen?

----------------------------------------------------------------------------------------------------------------------------------------

Example:

---------

Computer.java:

-----------------

@Component

public class Computer {

@Autowired

private RAM ram; //LAZY

public Computer() {

System.out.println("Initializing computer");

}

public RAM getRam() {

return ram;

}

public void setRam(RAM ram) {

this.ram = ram;

}

}

RAM.java:

---------------

@Component

@Scope(value = "prototype")

public class RAM {

private int price = 100;

private String size = "64GB";

public RAM() {

System.out.println("Intializing RAM");

}

public int getPrice() {

return price;

}

public void setPrice(int price) {

this.price = price;

}

public String getSize() {

return size;

}

public void setSize(String size) {

this.size = size;

}

}

Since there is a need of RAM object so it creates, even though it is prototype scope.

Spring Framework is creating normal RAM Object.

//below line of code will initialize both Computer and RAM Classes.

ApplicationContext context = new AnnotationConfigApplicationContext(TestConfig.class);

But I don't want to create normal instead want to create CGLIB proxy object:

---------------------------------------------------------------------------------------------------------------

@Scope(value = "prototype",proxyMode = ScopedProxyMode.TARGET\_CLASS)

proxyMode = ScopedProxyMode.TARGET\_CLASS ==> which activates CGLib proxy.

whenever singleton bean is created a CGLIB proxy is getting assigned.

// line line of code will only intialize Computer Class

ApplicationContext context = new AnnotationConfigApplicationContext(TestConfig.class);

System.out.println("container loaded");

// this line of code will be having computer class object which inturn has RAM CGLIB proxy object.

Computer computer = context.getBean(Computer.class);

RAM ram1 = computer.getRam(); // CGLIB proxy will call getRam() of Original class and get us original object.

System.out.println(ram1); // when this line executes then only constructor will be called

RAM ram2 = computer.getRam(); // returns new object

System.out.println(ram2);

TO assign JDKProxy for above code:

----------------------------------------

Change RAM.java class to below code:

@Scope(value = "prototype",proxyMode = ScopedProxyMode.INTERFACES)

proxyMode = ScopedProxyMode.INTERFACES ==> It uses JDK Proxy.

Computer.java:

----------------

@Component

@Scope("singleton")

public class Computer {

@Autowired

private IRAM ram; // using interface reference since JDKProxy won't work with classes

public Computer() {

System.out.println("Initializing computer");

}

public IRAM getRam() {

return ram;

}

public void setRam(RAM ram) {

this.ram = ram;

}

}

RAM.java:

-------------

@Component

@Scope(value = "prototype",proxyMode = ScopedProxyMode.INTERFACES)

public class RAM implements IRAM {

private int price = 100;

private String size = "64GB";

public RAM() {

System.out.println("Intializing RAM");

}

public int getPrice() {

return price;

}

public void setPrice(int price) {

this.price = price;

}

public String getSize() {

return size;

}

public void setSize(String size) {

this.size = size;

}

}

Inside main() write below code:

-------------------------------------

AnnotationConfigApplicationContext container = new AnnotationConfigApplicationContext(ScopeConfig.class);

System.out.println("container loaded");

Computer computer = container.getBean(Computer.class);

IRAM ram1 = computer.getRam();

System.out.println(ram1);

IRAM ram2 = computer.getRam();

System.out.println(ram2);