**Questions:**

**1. What else are the other annotations we need to keep in mind while working on core spring:**

Ex :

**@Bean**

*Answer:*

To declare a bean, simply annotate a method with the @Bean annotation. When JavaConfig encounters such a method, it will execute that method and register the return value as a bean within a BeanFactory. By default, the bean name will be the same as the method name . The following is a simple example of a @Bean method declaration:

Ex:

@Configuration  
public class AppConfig {  
 @Bean  
 public TransferService transferService() {  
 return new TransferServiceImpl();  
 }  
}

For comparison sake, the configuration above is exactly equivalent to the following Spring XML:

<beans>  
 <bean name="transferService" class="com.mytectra.TransferServiceImpl"/>  
</beans>

Both will result in a bean named transferService being available in the BeanFactory / ApplicationContext, bound to an object instance of type TransferServiceImpl:

transferService -> com.mytectra.TransferServiceImpl

**@Required**

*Answer:*

The @Required annotation applies to bean property setter methods and it indicates that the affected bean property must be populated in XML configuration file at configuration time. Otherwise, the container throws a BeanInitializationException exception. Following is an example to show the use of @Required annotation.

import org.springframework.beans.factory.annotation.Required;  
  
public class Student {  
 private Integer age;  
 private String name;  
  
 @Required  
 public void setAge(Integer age) {  
 this.age = age;  
 }  
 public Integer getAge() {  
 return age;  
 }  
   
 @Required  
 public void setName(String name) {  
 this.name = name;  
 }  
 public String getName() {  
 return name;  
 }  
}

**@Value**

This annotation can be used for injecting values into fields in Spring-managed beans and it can be applied at the field or constructor/method parameter level.

To describe different kinds of usage for this annotation, we need to configure a simple Spring application configuration class.

And naturally, we’ll need a properties file to define the values we want to inject with the @Value annotation. And so, we’ll first need to define a @PropertySource in our configuration class – with the properties file name.

Let’s define the properties file:

*value.from.file=Value got from the file*

*priority=Properties file*

*listOfValues=A,B,C*

Ex:

As a basic and mostly useless usage example we can only inject “string value” from the annotation to the field:

*@Value("string value")*

*private String stringValue;*

Using the @PropertySource annotation allows us to work with values from properties files with the @Value annotation. In the following example we get “Value got from the file” assigned to the field:

*@Value("${value.from.file}")*

*private String valueFromFile;*

We can also set the value from system properties with the same syntax. Let’s assume that we have defined a system property named systemValue and look at the following sample:

*@Value("${systemValue}")*

*private String systemValue;*

Default values can be provided for properties that might not be defined. In this example the value “some default” will be injected:

*@Value("${unknown.param:some default}")*

*private String someDefault;*

If the same property is defined as a system property and in the properties file, then the system property would be applied.

Suppose we had a property priority defined as a system property with the value “System property” and defined as something else in the properties file. In the following code the value would be “System property”:

*@Value("${priority}")*

*private String prioritySystemProperty;*

Sometimes we need to inject a bunch of values. It would be convenient to define them as comma-separated values for the single property in the properties file or as a system property and to inject into an array. In the first section, we defined comma-separated values in the listOfValues of the properties file, so in the following example the array values would be [“A”, “B”, “C”]:

*@Value("${listOfValues}")*

*private String[] valuesArray;*

**@DependsOn**

The @DependsOn annotation can force Spring IoC container to initialize one or more beans before the bean which is annotated by @DependsOn annotation.

The @DependsOn annotation may be used on any class directly or indirectly annotated with @Component or on methods annotated with @Bean.

The following example shows how to use @DependsOn annotation in spring application.

Ex:

Consider the following beans BeanOne, BeanTwo and BeanThree.

**import** org.springframework.beans.factory.annotation.Autowired;  
  
**public** **class** **BeanOne** {  
 @Autowired  
 **private** BeanTwo beanTwo;  
  
 @Autowired  
 **private** BeanThree beanThree;  
  
 **public** **BeanOne**() {  
 System.out.println("BeanOne Initialized");  
 }  
  
 **public** **void** **doSomthing**() {  
 System.out.println("Inside doSomthing() method of BeanOne");  
 beanTwo.doSomthing();  
 beanThree.doSomthing();  
 }  
}

**public** **class** **BeanTwo** {  
  
 **public** **BeanTwo**() {  
 System.out.println("BeanTwo Initialized");  
 }  
  
 **public** **void** **doSomthing**() {  
 System.out.println("Inside doSomthing() method of BeanTwo");  
 }  
  
}

**public** **class** **BeanThree** {  
  
 **public** **BeanThree**() {  
 System.out.println("BeanThree Initialized");  
 }  
  
 **public** **void** **doSomthing**() {  
 System.out.println("Inside doSomthing() method of BeanThree");  
 }  
}

**import** org.springframework.context.annotation.Bean;  
**import** org.springframework.context.annotation.Configuration;  
**import** org.springframework.context.annotation.DependsOn;  
  
@Configuration  
**public** **class** **AppConfig** {  
  
 @Bean("beanOne")  
 @DependsOn(value = { "beanTwo", "beanThree" })  
 **public** BeanOne **getBeanOne**() {  
 **return** **new** BeanOne();  
 }  
  
 @Bean("beanTwo")  
 **public** BeanTwo **getBeanTwo**() {  
 **return** **new** BeanTwo();  
 }  
  
 @Bean("beanThree")  
 **public** BeanThree **getBeanThree**() {  
 **return** **new** BeanThree();  
 }  
}

**import** org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
**public** **class** **MainApp** {  
 **public** **static** **void** **main**(String[] args) {  
 AnnotationConfigApplicationContext context = **new** AnnotationConfigApplicationContext(AppConfig.class);  
 BeanOne bean=context.getBean(BeanOne.class);  
 bean.doSomthing();  
 context.close();  
 }  
}

**@Lazy**

**By default, Spring creates all singleton beans eagerly at the startup/bootstrapping of the application context**. The reason behind this is simple: to avoid and detect all possible errors immediately rather than at runtime.

However, there are cases when we need to create a bean, not at the application context startup, but when we request it.

**Lazy Initialization**

The @Lazy annotation has been present since Spring version 3.0. There’re several ways to tell the IoC container to initialize a bean lazily.

**When we put @Lazy annotation over the @Configuration class, it indicates that all the methods with @Bean annotation should be loaded lazily.**

This is the equivalent for the XML based configuration’s default-lazy-init=“true“ attribute.

Let’s have a look here:

@Lazy

@Configuration

@ComponentScan(basePackages = "com.mytectra.lazy")

public class AppConfig {

@Bean

public Region getRegion(){

return new Region();

}

@Bean

public Country getCountry(){

return new Country();

}

}

Let’s now test the functionality:

@Test

public void givenLazyAnnotation\_whenConfigClass\_thenLazyAll() {

AnnotationConfigApplicationContext ctx

= new AnnotationConfigApplicationContext();

ctx.register(AppConfig.class);

ctx.refresh();

ctx.getBean(Region.class);

ctx.getBean(Country.class);

}

As we see, all beans are created only when we request them for the first time:

Bean factory for ...AnnotationConfigApplicationContext:

...DefaultListableBeanFactory: [...];

// application context started

Region bean initialized

Country bean initialized

To apply this to only a specific bean, let’s remove the @Lazy from a class.

Then we add it to the config of the desired bean:

@Bean

@Lazy(true)

public Region getRegion(){

return new Region();

}

**With @Autowired**

Before going ahead, check out these guides for @Autowired and @Component annotations.

Here, in order to initialize a lazy bean, we reference it from another one.

The bean that we want to load lazily:

@Lazy

@Component

public class City {

public City() {

System.out.println("City bean initialized");

}

}

And it’s reference:

public class Region {

@Lazy

@Autowired

private City city;

public Region() {

System.out.println("Region bean initialized");

}

public City getCityInstance() {

return city;

}

}

**Note, that the @Lazy is mandatory in both places.**

With the @Component annotation on the City class and while referencing it with @Autowired:

@Test

public void givenLazyAnnotation\_whenAutowire\_thenLazyBean() {

// load up ctx application context

Region region = ctx.getBean(Region.class);

region.getCityInstance();

}

Here, the City bean is initialized only when we call the getCityInstance() method.

**@Lookup**

Spring lookup method injection is the process of dynamically overriding a registered bean method. The bean method should be annotated with @Lookup. Spring returns the lookup result matched by the method's return type.

@Component

public class MySingletonBean {

public void showMessage(){

MyPrototypeBean bean = getPrototypeBean();

//each time getPrototypeBean() call

//will return new instance

}

@Lookup

public MyPrototypeBean getPrototypeBean(){

//spring will override this method

return null;

}

}

In above example the method getPrototypeBean is returning null. That doesn't matter, because this method will actually be overridden by spring dynamically. Spring uses CGLIB library to do so.

The dynamically generated code will look for the target bean in the application context. Something like this:

...

public MyPrototypeBean getPrototypeBean(){

return applicationContext.getBean(MyPrototypeBean.class);

}

...

For dynamic code generation to work, we have to follow these conditions on the bean class :

The bean class cannot be final.

The method annotated with @Lookup, cannot be private , static or final

The factory approach of JavaConfig doesn't work i.e. a factory method annotated with @Bean and returning a manually created instance of the bean doesn't work. Since the container is not in charge of creating the instance, therefore it cannot create a runtime-generated subclass on the fly.

**@Primary**

This Indicates that a particular bean should be given preference when multiple beans are candidates to be autowired to a single-valued dependency. If exactly one 'primary' bean exists among the candidates, it will be the autowired value.

**Difference between @Primary vs @Autowired with @Qualifier annotations**

If a bean has @Autowired without any @Qualifier, and multiple beans of the type exist, the candidate bean marked @Primary will be chosen, i.e. it is the default selection when no other information is available, i.e. when @Qualifier is missing.

A good use case is that initially you only had one bean of the type, so none of the code used @Qualifier. When you then add another bean, you then also add @Qualifier to both the old and the new bean, so any @Autowired can choose which one it wants. By also adding @Primary to the old original bean, you don't have to add @Qualifier to all the existing @Autowired. They are "grandfathered" in, so to speak.

@Primary is also good if e.g. 95% of @Autowired wants a particular bean. That way, only the @Autowired that wants the other bean(s) need to specify @Qualifier. That way, you have primary beans that all autowired wants, and @Qualifier is only used to request an "alternate" bean.

The following examples demonstrate the use of the @Primary annotation.

Consider the following User interface.

**public** **interface** **User** {  
 **public** **void** **doSomething**();  
}

**public** **class** **AdminUser** **implements** **User** {  
 @Override  
 **public** **void** **doSomething**() {  
 System.out.println("Inside doSomething() method of AdminUser");  
 }  
}

**public** **class** **GuestUser** **implements** **User** {  
 @Override  
 **public** **void** **doSomething**() {  
 System.out.println("Inside doSomething() method of GuestUser");  
 }  
}

**import** org.springframework.context.annotation.Bean;  
**import** org.springframework.context.annotation.Configuration;  
**import** org.springframework.context.annotation.Primary;  
  
**import** com.boraji.tutorial.spring.AdminUser;  
**import** com.boraji.tutorial.spring.GuestUser;  
**import** com.boraji.tutorial.spring.User;  
  
@Configuration  
**public** **class** **AppConfig** {  
 @Bean  
 @Primary  
 **public** User **getAdminUser**() {  
 **return** **new** AdminUser();  
 }  
  
 @Bean  
 **public** User **getGuestUser**() {  
 **return** **new** GuestUser();  
 }  
}

**import** org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
**public** **class** **MainApp** {  
 **public** **static** **void** **main**(String[] args) {  
 AnnotationConfigApplicationContext context =   
 **new** AnnotationConfigApplicationContext(AppConfig.class);  
   
 User user=context.getBean(User.class);  
 user.doSomething();  
   
 context.close();  
 }  
}

It is clear from the above output that the getAdminUser() method, annotated with @Primary, is autowired first.

**@Profile**

Most projects will have different environments like DEV, QA, PREPROD and PRODUCTION. Most of the projects have different databases for each Environment. First developer starts developing projects on DEV environment which uses DEV database. Once development is done, they will move code to QA environment which uses different database. Once QA is done successfully, they will move to PREPRODUCTION environment, which uses PREPRO database to do End-to-End and performance testing. Once everything is done and happy to go live, it will be deployed on LIVE or PRODUCTION Environment which uses PROD database.Then creating DataSource object for each environment requires different database details.

If we change Database details, then we need to rebuild and deploy application. We cannot use same application WAR or EAR file for all environments.To solve this kind of environment related setup dependencies, Spring 3.1 has introduced a new annotation. That is @Profile annotation. It can be used to develop an “If-Then-Else” conditional checking to configure. We cannot implement this scenario by using SpEL Ternary Operator.

To work with Profiles, Spring 3.1 Framework has provided the following two properties

spring.profiles.default

Spring.profiles.active

spring.profiles.active represents active profile.

spring.profiles.default represents default profile.

If we don’t specify active profile, then Spring IOC Container will look for default profile. We need to provide values to one of these properties as JVM Parameters. In Eclipse or Spring STS IDEs, we need to pass these values as shown below,

-Dspring.profiles.active=dev

Then Spring IOC Container uses this profile value and creates only those beans to run the application. To activate the profiles in JUnits, Spring Framework has provided another annotation: @ActiveProfiles. Instead of activating a profile using JVM Parameters, we can use this annotation to active a profile.

Ex:

@Configuration  
@Profile("dev") //@Profile("prod")  
public abstract class DevEmployeeConfig{   
 @Bean  
 public DataSource dataSource() {  
 return new DevDatabaseUtil();  
 }   
}

Ref: <https://javapapers.com/spring/profile-annotation-improvements-in-spring-4/>

**@Import**

Spring @Import annotation provides functionality similar to <import/> element in spring XML. Using @Import annotation you can import one or more @Configuration classes. It can also import classes containing at least one @Bean method.

**Declared Beans using @Configuration and @Bean**

AppConfig definitions is a @Configuration annotated class containing bean definitions defined using @Bean.

AppConfig:

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class AppConfig {

@Bean

public BeanA beanA() {

return new BeanA();

}

@Bean

public InnerBean innerBean() {

return new InnerBean();

}

static class InnerBean {

}

}

Here is a class that is not annotated with @Configuration but still can be imported as it contains @Bean methods.

BeanHolder:

import org.springframework.context.annotation.Bean;

public class BeanHolder {

@Bean

public BeanB beanB() {

return new BeanB();

}

@Bean

public BeanC beanC() {

return new BeanC();

}

}

BeanA:

public class BeanA {

}

BeanB:

public class BeanB {

}

BeanC:

public class BeanC {

}

**Import Beans using @Import**

We use @Import annotation and pass in the classes to be imported, AppConfig and BeanHolder. It not only import the classes passed in but also any beans declared using @Bean methods.

Bean definitions can be accessed using the ApplicationContext or by auto-wiring using @Autowired annotation.

ImportAnnotationExample:

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Import;

@Configuration

@Import({AppConfig.class, BeanHolder.class})

public class ImportAnnotationExample {

@Autowired

private AppConfig appConfig;

@Autowired

private BeanB beanB;

public static void main(String[] args) {

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(

ImportAnnotationExample.class);

printBean(context, "com.javarticles.spring.AppConfig");

printBean(context, "beanA");

printBean(context, "beanB");

printBean(context, "beanC");

printBean(context, "innerBean");

ImportAnnotationExample importAnnotationExample = (ImportAnnotationExample)

context.getBean("importAnnotationExample");

System.out.println("AppConfig member: " + importAnnotationExample.getAppConfig());

System.out.println("BeanB member: " + importAnnotationExample.getBeanB());

}

public static void printBean(ApplicationContext context, String beanId) {

System.out.println(beanId + ": " + context.getBean(beanId));

}

public AppConfig getAppConfig() {

return appConfig;

}

public BeanB getBeanB() {

return beanB;

}

}

**@ImportResource**

Indicates one or more resources containing bean definitions to import.

Like @Import, this annotation provides functionality similar to the <import/> element in Spring XML. It is typically used when designing @Configuration classes to be bootstrapped by an AnnotationConfigApplicationContext, but where some XML functionality such as namespaces is still necessary.

By default, arguments to the value() attribute will be processed using a GroovyBeanDefinitionReader if ending in ".groovy"; otherwise, an XmlBeanDefinitionReader will be used to parse Spring <beans/> XML files. Optionally, the reader() attribute may be declared, allowing the user to choose a custom BeanDefinitionReader implementation.

**Ex:**

BeanA:

public class BeanA {

}

BeanB:

public class BeanB {

}

BeanC:

public class BeanC {

}

BeanA is defined in context1.xml

BeanB is defined in com/javarticles/spring

BeanC will be defined a properties file

context1.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/spring-beans.xsd">

<bean id="beanA" class="com.javarticles.spring.BeanA" />

</beans>

context2.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/spring-beans.xsd">

<bean id="beanB" class="com.javarticles.spring.Beanb" />

</beans>

We define BeanC in a properties file.

context.properties:

beanC.(class)=com.javarticles.spring.BeanC

In the below example, we import the context files using @ImportResource. We need to pass in the location path of the files. In order to access the beans, we auto-wire them to the class members.

@ImportResource({ "context1.xml", "com/javarticles/spring/context2.xml" })

We can also load beans defined in properties file using an explicit reader. For example, in the below configuration bean we pass in the classpath:context.properties and use reader set to

PropertiesBeanDefinitionReader.class.

@Configuration

@ImportResource(value="classpath:context.properties", reader=PropertiesBeanDefinitionReader.class)

static class Config {

}

Here is the complete example of @ImportResource.

ImportResourceAnnotationExample:

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.support.PropertiesBeanDefinitionReader;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.ImportResource;

@Configuration

@ImportResource({ "context1.xml", "com/javarticles/spring/context2.xml" })

public class ImportResourceAnnotationExample {

@Autowired

private BeanA beanA;

@Autowired

private BeanB beanB;

@Autowired

private BeanC beanC;

public static void main(String[] args) {

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(

ImportResourceAnnotationExample.class);

try {

ImportResourceAnnotationExample importResourceAnnotationExample = (ImportResourceAnnotationExample) context

.getBean("importResourceAnnotationExample");

System.out.println("BeanA member: "

+ importResourceAnnotationExample.getBeanA());

System.out.println("BeanB member: "

+ importResourceAnnotationExample.getBeanB());

System.out.println("BeanC member: "

+ importResourceAnnotationExample.getBeanC());

} finally {

context.close();

}

}

public BeanA getBeanA() {

return beanA;

}

public BeanB getBeanB() {

return beanB;

}

public BeanC getBeanC() {

return beanC;

}

@Configuration

@ImportResource(value="classpath:context.properties", reader=PropertiesBeanDefinitionReader.class)

static class Config {

}

}

**@PropertySource**

Spring @PropertySource annotation is used to provide properties file to Spring Environment. This annotation is used with @Configuration classes.

Spring PropertySource annotation is repeatable, means you can have multiple PropertySource on a Configuration class. This feature is available if you are using Java 8 or higher version.

**Ex:**

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.PropertySource;

import org.springframework.core.env.Environment;

@Configuration

@PropertySource("classpath:db.properties")

@PropertySource("classpath:root.properties")

public class DBConfiguration

{

@Autowired

Environment env;

@Bean

public DBConnection getDBConnection()

{

System.out.println("Getting DBConnection Bean for App: "+env.getProperty("APP\_NAME"));

DBConnection dbConnection = new DBConnection(env.getProperty("DB\_DRIVER\_CLASS"), env.getProperty("DB\_URL"),env.getProperty("DB\_USERNAME"), env.getProperty("DB\_PASSWORD").toCharArray());

return dbConnection;

}

}

db.properties

#MYSQL Database Configurations

DB\_DRIVER\_CLASS=com.mysql.jdbc.Driver

DB\_URL=jdbc:mysql://localhost:3306/Test

DB\_USERNAME=root

DB\_PASSWORD=password

root.properties

APP\_NAME=PropertySource Example

**@PropertySources**

If there are multiple PropertySources, you can use annotation @PropertySources which aggregates several PropertySource annotations. In each @PropertySource, you can specify one or more resource locations.

**Ex:**

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.PropertySource;

import org.springframework.context.annotation.PropertySources;

import org.springframework.core.env.Environment;

@Configuration

@PropertySources({

@PropertySource("classpath:com/javarticles/spring/annotations/x.properties"),

@PropertySource("classpath:com/javarticles/spring/annotations/y.properties")

})

public class SpringPropertySourcesAnnotationExample {

public static void main(String[] args) {

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();

try {

ctx.register(SpringPropertySourcesAnnotationExample.class);

ctx.refresh();

Environment env = ctx.getEnvironment();

System.out.println("Topic: " + env.getProperty("topic"));

System.out.println("Example: " + env.getProperty("example"));

} finally {

ctx.close();

}

}

}

You can specify ${...} placeholders in the resource location path. Spring will resolve the place holders using the already registered property sources.

For example, classpath:com/javarticles/spring/annotations/${more}.properties contains placeholder ${more}. Before spring gets to register ${more}.properties, it already loads x.properties. Property more is set to z so spring resolves ${more}.properties to z.properties.

**Ex:**

x.properties:

topic=spring annotations

more=z

z.properties:

topic=spring jpa

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.PropertySource;

import org.springframework.context.annotation.PropertySources;

import org.springframework.core.env.Environment;

@Configuration

@PropertySources({

@PropertySource("classpath:com/javarticles/spring/annotations/x.properties"),

@PropertySource("classpath:com/javarticles/spring/annotations/${more}.properties"),

})

public class SpringPropertySourcesAnnotationPropertyHolderExample {

public static void main(String[] args) {

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();

try {

ctx.register(SpringPropertySourcesAnnotationPropertyHolderExample.class);

ctx.refresh();

Environment env = ctx.getEnvironment();

System.out.println("topic: " + env.getProperty("topic"));

} finally {

ctx.close();

}

}

}

**2. How I can configure Spring Core without using Maven ?**

Ref : <https://www.mkyong.com/spring-mvc/gradle-spring-mvc-web-project-example/>

**3. Is it possible to have XML based and annotation based application in Spring?**

Who says you need to choose between XML and annotation based configurations. Why not use both! It's as easy as a little import annotation at the top of your config file.

If you are using a Java @Configuration file, you can simultaneously load in an XML configuration file simply by adding an @ImportResource annotation. The following annotation loads in a Spring XML configuration file named meanbeans.xml into the annotation based Spring ApplicationContext object:

**@Configuration**

**@ImportResource("classpath:meanbeans.xml")**

**class JConfig { }**

So, if you've got a class named FooBean and a class named BarBean, and you have one bean configured in the XML file, and the other configured in the Java @Configuration file, you can pull them both into your application from a single beanFactory.

**class FooBean { }**

**class BarBean { }**

So, this would be the goal:

**public class XmlAndAnnotations {**

**public static void main(String args[]) {**

**AnnotationConfigApplicationContext beanFactory =**

**new AnnotationConfigApplicationContext(JConfig.class);**

**//new ClassPathXmlApplicationContext("meanbeans.xml");**

**System.out.println(beanFactory.getBean("fooBean"));**

**System.out.println(beanFactory.getBean("barBean"));**

**}**

**}**

Notice how both the fooBean and barBean are pulled from a common beanFactory, despite the two beans being configured in different configuration resources? Here's the full Java based configuration file, which only specifically describes the creation of the fooBean:

**@Configuration**

**@ImportResource("classpath:meanbeans.xml")**

**class JConfig {**

**@Bean()**

**public FooBean fooBean() {return new FooBean();}**

**}**





And that's it! When the main method of the XmlAndAnnotations class runs, both the fooBean and barBean will be loaded from two separate Spring configuration resources!

**4. How the annotation based parameters from HTTP been bind in spring? What does that mean is, how does it look like in conventional way of doing web service internally?**

**5. How to send XML data using REST API? Using Postman**

In this section we will discuss how to read XML data from REST request using Spring Boot. As I told you in the previous articles, spring boot by default support reading and producing the JSON data. But for any XML support we have to include *jackson-dataformat-xml* dependency.

## **xml dependency**

<dependency>  
 <groupId>com.fasterxml.jackson.dataformat</groupId>  
 <artifactId>jackson-dataformat-xml</artifactId>  
</dependency>

## **Final pom.xml**

<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
 <groupId>com.java4s</groupId>  
 <artifactId>SpringBootRestfulPostXML</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>1.5.6.RELEASE</version>  
 </parent>  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 <!-- xml dependency-->  
 <dependency>  
 <groupId>com.fasterxml.jackson.dataformat</groupId>  
 <artifactId>jackson-dataformat-xml</artifactId>  
 </dependency>  
 <!-- xml dependency end-->  
 </dependencies>  
 <properties>  
 <java.version>1.8</java.version>  
 </properties>  
</project>

## **SpringBootRestfulPostXML.java**

import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
*@SpringBootApplication*  
public class SpringBootApp {  
 public static void main(String[] args) {  
 SpringApplication.run(SpringBootApp.class, args);  
 }  
}

## **Customer.java**

public class Customer {  
 private int custNo;  
 private String name;  
 private String country;  
 public Customer() {  
 }  
 public Customer(int custNumber, String name, String country) {  
 this.custNo = custNumber;  
 this.name = name;  
 this.country = country;  
 }  
 public int getCustNo() {  
 return custNo;  
 }  
 public void setCustNo(int custNo) {

this.custNo = custNo;  
 }

public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
 public String getCountry() {  
 return country;  
 }  
 public void setCountry(String country) {  
 this.country = country;  
 }  
}

## **SpringJava4sController.java**

import org.springframework.web.bind.annotation.PostMapping;  
import org.springframework.web.bind.annotation.RequestBody;  
import org.springframework.web.bind.annotation.RestController;  
import com.mytectra.model.Customer;  
*@RestController*  
public class SpringJava4sController {  
 *@PostMapping(path = "/save-cust-info")*  
 public String customerInformation(*@RequestBody Customer cust) {*  
 /\* You can call your DAO logic here.  
 \* For time being I am printing the customer data just to show the POST call is working.  
 \*/  
 return "Customer information saved successfully ::." + cust.getCustNo() + " " + cust.getName() + " " + cust.getCountry();  
 }  
}

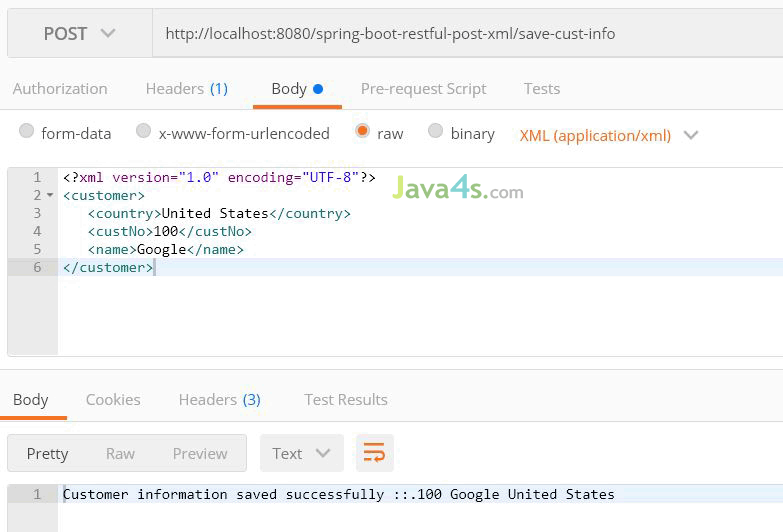
## **application.properties**

Run the application and open *http://localhost:8080/spring-boot-restful-post-xml/save-cust-info* in Postman.

Request:

<?xml version="1.0" encoding="UTF-8"?>  
<customer>  
 <country>United States</country>  
 <custNo>100</custNo>  
 <name>Google</name>  
</customer>

**Response**:



**6. When exactly we can use REST API in our application?**

The particular reason we can use the REST API, when we have different services to interact with each other and don’t care about the state of the protocol. There are several advantages and features to explain REST API to use. Some of them are as follows:

**Features**

1. *Stateless client/server protocol*: Each HTTP contains all the necessary information to run it, which means that neither the client nor the server need to remember any previous state to satisfy it. Be that as it may, some HTTP applications incorporate a cache memory. This configures what is known as the stateless client-cache-server protocol: it is possible to define some of the responses to specific HTTP requests as cachable, so the client can run the same response for identical requests in the future. However, the fact that the option exists doesn't mean it is the most recommended.
2. There are four very important data transactions in any REST system and HTTP specification: POST (create), GET (read and consult), PUT (edit) and DELETE.
3. *Objects in REST are always manipulated from the URI*. It is the URI and no other element that is the sole identifier of each resource in this REST system. The URI allows us to access the information in order to change or delete it, or for example to share its exact location with third parties.
4. *Uniform interface*: to transfer data, the REST system applies specific actions (POST, GET, PUT and DELETE) on the resources, provided they are identified with a URI. This makes it easier to obtain a uniform interface that systematizes the process with the information.
5. *Layer system*: hierarchical architecture between the components. Each layer has a functionality within the REST system.
6. *Use of hypermedia*: hypermedia is a term coined by Ted Nelson in 1965 and is an extension of the concept of hypertext. This concept, taken to web page development, is what allows the user to browse the set of objects through HTML links. In the case of a REST API, the concept of hypermedia explains the capacity of an app development interface to provide the client and the user with the adequate links to run specific actions on the data.

**Advantages**

1. *Separation between the client and the server*: the REST protocol totally separates the user interface from the server and the data storage. This has some advantages when making developments. For example, it improves the portability of the interface to other types of platforms, it increases the scalability of the projects, and allows the different components of the developments to be evolved independently.
2. *Visibility, reliability and scalability*. The separation between client and server has one evident advantage, and that is that each development team can scale the product without too much problem. They can migrate to other servers or make all kinds of changes in the database, provided the data from each request is sent correctly. The separation makes it easier to have the front and the back on different servers, and this makes the apps more flexible to work with.
3. *The REST API is always independent of the type of platform or languages*: the REST API always adapts to the type of syntax or platforms being used, which gives considerable freedom when changing or testing new environments within the development. With a REST API you can have PHP, Java, Python or Node.js servers. The only thing is that it is indispensable that the responses to the requests should always take place in the language used for the information exchange, normally XML or JSON.

**7. Is it possible to work on spring without using annotation?**

Yes and No. Yes, because, instead of using annotation, we can adhere XML based configuration. No. because, Spring provides many features which reduces boilerplate coding standard and it reduces common coding standard. That’s the reason, we should use Annotation.

**8. How can I configure a bean definition for an inner class?**

Ex:

package x.y.z;

public class A {

public class B { }

}

You cannot access your public inner class with the dot (.) notation, instead, use the currency ($). An example:

<bean class="x.y.z.A$B" name="innerBean" />

**Good to have an Idea:**

**Spring Inner Bean Injection**

A bean defined within the context of another bean is called a spring inner bean. It is similar to the concept of inner classes in java. e.g. In an ATM (Automated Teller Machine) system, it is possible for the Printer bean to be defined as an inner bean of ATM class.

## **Characteristics of Spring inner bean**

The following are the typical characteristics of Spring inner beans:

* When a bean does not need to be shared with other beans, then it can optionally be declared as an inner bean. Inner bean cannot be injected into any other bean except for the enclosing bean.
* Inner bean is defined within the context of its enclosing bean
* Inner bean typically does not have an *id* associated with it although it is perfectly legal to provide an *id*. This is because, by its very definition, inner bean will not be shared outside of its enclosing bean. The value of the *id* attribute of an inner bean  
  is ignored by Spring.
* Defining an inner bean does not imply that it should be defined as an inner class e.g. Printer bean is defined as inner bean but Printer class is not an inner class.
* The *scope* of an inner bean is always *prototype*. Spring will ignore the value of the *scope* attribute of an inner bean

## **Limitations of Spring inner bean**

The following are some of the limitations of using spring inner beans:

* Spring Inner beans cannot be reused/shared with other beans
* In practice, defining inner beans affects the readability of Spring configuration XML

Trail Tip: Use Spring inner beans minimally in your application.

The following sample program provides an overview of usage of Spring inner bean

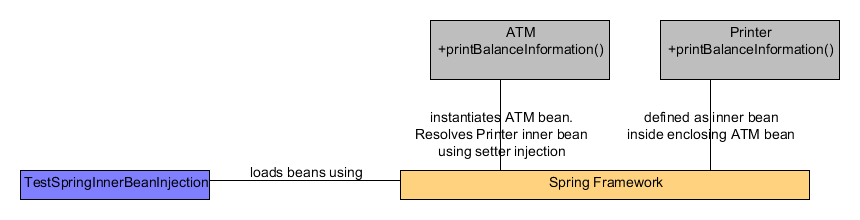
# Spring Inner Bean example

In the sample program, we will create an ATM class and Printer class. As the Printer class is not referenced anywhere outside of the ATM class we will declare Printer bean as an inner bean (inside the enclosing ATM bean). The ATM.printBalanceInformation() method delegates the call to Printer.printBalanceInformation() method.

Also, the Printer bean will be declared as an inner bean and will be injected into ATM bean using setter injection.

Note: Although in this sample program we are using setter injection, it is also possible to declare inner beans using constructor injection.

Finally, we will test our setup using TestSpringInnerBeanInjection class which will load Spring context and call the ATM.printBalanceInformation() method to verify that setter injection using inner bean declarations has happened successfully.



Source Code for Spring Inner Bean example

Create the ATM class with a dependency on Printer class. The ATM class delegates the call to print the balance information to the Printer class (see line 15 below).

public class ATM {

private Printer printer;

public Printer getPrinter() {

return printer;

}

public void setPrinter(Printer printer) {

this.printer = printer;

}

public void printBalanceInformation(String accountNumber) {

getPrinter().printBalanceInformation(accountNumber);

}

}

Create a Printer class with a method printBalanceInformation() (see line 5 below)

public class Printer {

public void printBalanceInformation(String accountNumber) {

System.out.println("The printer has printed the balance information for the account number " + accountNumber);

}

}

Declare the ATM class in spring-config.xml (see line 10 below).

Declare the printer property of ATM class for setter injection (see line 11 below)

Also declare the Printer bean as an inner bean (see line 12 below). As Printer bean has been declared inside the enclosing ATM bean, it is an inner bean. Note that no id attribute has been provided for the Printer bean as it cannot be used outside the context of ATM bean.

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

<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www.springframework.org/schema/aop" xsi:schemalocation=" http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

<bean id="atm" class="com.studytrails.tutorials.springinnerbeaninjection.ATM">

<property name="printer">

<bean class="com.studytrails.tutorials.springinnerbeaninjection.Printer">

</bean></property>

</bean>

</beans>

Finally, we need a java program to test our setup. This is done by TestSpringInnerBeanInjection.java (see source code below).

We need to tell Spring framework to use the ‘spring-config.xml’ to load our beans(see line 11 below).

We get the reference to ATM class through Spring using the bean name ‘atm’ (see line 12 below). In this step, Spring will resolve the dependency on Printer class (declared as an inner bean) by injecting it using setter injection.

We call the printBalanceInformation() on ATM class (see line 14 below) with some accountNumber (see line 13 below). We will verify that the call is delegated to Printer.printBalanceInformation() method successfully implying the inner bean injection has happened successfully.

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpringInnerBeanInjection {

public static void main(String[] args) {

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\* BEGINNING PROGRAM \*\*\*\*\*\*\*\*\*\*\*\*\*\*");

ApplicationContext context = new ClassPathXmlApplicationContext("spring-config.xml");

ATM atm = (ATM)context.getBean("atm");

String accountNumber = "AC5645786";

atm.printBalanceInformation(accountNumber);

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\* ENDING PROGRAM \*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

}

**9. What is HTTP content negotiation**

HTTP provides a smart mechanism to manage formats of exchanged payloads. This mechanism is responsible for selecting the appropriate representation when servicing an HTTP request. This feature is called content negotiation or CONNEG. In this post, we will describe how to use it, as well as the different headers involved and the possible status codes for corresponding errors.

As it’s used by browsers under the hood when interacting with a HTTP server, you can just as well leverage this mechanism when interacting with a RESTful service. This allows you to get data formats you expect and you can handle in your application. In addition, when implementing such a service, you can provide support for several formats and allow the client to choose the one he wants to use.

Before actually dealing with content negotiation, let’s define what a content type is and how to use it within HTTP requests and responses.

**Defining content format**

When you put data in the payload of a HTTP message within a request or a response, you must specify the corresponding format using a Content-Type header. Usable values correspond to media types (initially called MIME types).

It is mandatory to allow the client or the server to know the format and make it possible to handle data. Here is a sample use of the header within a HTTP request:

POST /contacts HTTP/1.1  
Content-Type: application/json  
  
{  
 "lastName": "Fielding",  
 "firstName": "Roy"  
}  
  
HTTP/1.1 201 OK  
Location: http://myrestservice/contacts/1

# **Media type structure**

More generally media types follow the format top-level type name / [ tree. ] subtype name [ +suffix ] [ ; parameters ]. This is defined in the RFC 4288 (https://tools.ietf.org/html/rfc4288). So the value of this header contains a value with a lop-level type like text or application and a sub type like xml or json. This value can also include some optional parameters like the character encoding with the charset one. Notice that these parameters can differ depending on the header used.

Here is a sample with the Content-Type header:

Content-Type: application/json; charset=UTF-8

This header is used by the server or the client to select the right approach to handle (and eventually parse) the data.

# **Additional content headers**

Some other headers can also be used to give more details about the content:

* The Content-Encoding header indicates which additional content encodings have been applied to the payload, and thus what decoding mechanisms must be applied in order to obtain the media type referenced by the Content-Type header field. The Content-Encoding header is primarily used to allow a document to be compressed without losing the identity of its underlying media type.
* The Content-Language header describes the natural language(s) of the intended audience for the enclosed entity.

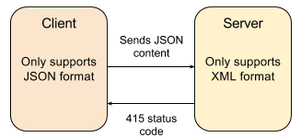
Here is a use sample of these two additional headers:

GET /contacts HTTP/1.1  
(...)  
  
HTTP/1.1 201 OK  
Content-Type: application/json  
Content-Language: en  
Content-Encoding: deflate

To get the content, the client needs to uncompress what was received in the payload.

# **Unsupported media type**

There is no obligation for the server to support the format sent. In this case, it will send back a 415 status code (Unsupported media type) to notify the client that made the request. “The request entity has a media type that the server or resource does not support”, according to the specification.



Now you understand what a media type is and how it identifies the payload format, let’s dive into the heart of the topic: how to tell the server to return the expected content.

# **Telling the server what format you expect**

When sending a request, you can specify the data format(s) you expect within the response payload. Defining several values is possible since clients are able to handle several formats.

The Accept header should be used for this purpose. Here is a simple sample to get a list of contacts with the JSON format:

GET /contacts HTTP/1.1  
Accept: application/json  
  
HTTP/1.1 200 OK  
Content-Type: application/json  
  
[  
 {  
 "id": 1,  
 "lastName": "Fielding",  
 "firstName": "Roy"  
 }  
]

# **Defining a set of media types**

As discussed in the first section, media type values are structured. This is particularly important in the context of the content negotiation since you can use the asterisk \* character to group media types in ranges. For example, the \*/\* value indicates all media types and type/\* all subtypes of that type.

If you can handle all text formats, simply use the following value in the Accept header:

Accept: text/\*

When no Accept header is defined, it corresponds to the \*/\* value i.e. any format can be returned and you let the server choose the format.

If the server cannot support a particular media type, you can define alternatives since the Accept header lets you define a set of media types.

# **Defining preferences**

When defining several values in the Accept header, it’s important to specify your preference order, i.e. which is your first preference according to what the server supports. The q parameter gives you the ability to define this order. Let’s take a sample:

Accept: application/json,application/xml;q=0.9,\*/\*;q=0.8

Let’s explain what this means. This allows you to ask the server a JSON format. If it can’t, perhaps it could return XML format (the second level). If it’s still not possible, let it return what it can. The preference order is defined through the q parameter with values from 0 to 1. When nothing is specified, the implicit value is 1.

Note that an implicit order is supported as shown in this simple sample:

Accept: text/\*, application/json, \*/\*

In this case, the preference order will be application/json, text/\* and finally \*/\*. Moreover with two values like application/json and application/xml, the order in the header value will be used for the preference order.

# **Additional headers**

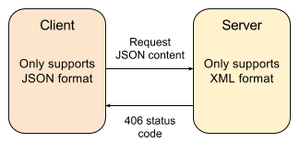
Other headers can also apply within content negotiation to build more advanced expectations using charsets, encodings and languages:

* Accept-Charset defines the character sets that are acceptable.
* Accept-Encoding contains the list of acceptable encodings. Possible values are gzip and deflate.
* Accept-Language contains the list of acceptable human languages for response.

When you specify a set of Accept-like headers with possible preferences, the server implementation needs to take everything into account in order to return the correct content. This can be tricky and the [RFC 2296](https://tools.ietf.org/html/rfc2296)specification “HTTP Remote Variant Selection Algorithm” describes the algorithm to implement based on scores.

# **Non supported requested format**

If the server doesn’t support the requested format, it will send back a 406 status code (Not Acceptable) to notify the client that made the request. “The requested resource is only capable of generating content not acceptable according to the Accept headers sent in the request”, according to the specification.



# **Conclusion**

As described throughout this post, HTTP natively supports a mechanism based on headers to tell the server about the content you expect and you’re able to handle. Based on these hints, the server is responsible for returning the corresponding content in the correct format.

Such mechanism is typically used by browsers under the hood when trying to interact with a website. RESTful services can also implement such a mechanism to adapt the exchanged contents according to the formats supported by clients.

The Restlet framework and the [Restlet Cloud](https://restlet.com/modules/cloud/) platforms both natively support such mechanisms.

**10. What is REST content negotiation**

REST resources can have multiple presentations (e.g JSON or XML) as different clients can request different representation.The mechanism for selecting a correct representation is known as **content negotiation**. It helps understanding type of data received by the system and expected response data format by the client.

### **1. Server Vs Client Driven Content Negotiation**

Content negotiation can happen at server or client side.Below are the high-level details of both approaches.

* In the server-driven negotiation, a server process is responsible for selecting best representation for the response.
* For client-driven content negotiation, the client made the selection for the resource representation.

*REST API implementations work based on client-driven content negotiation*

### **2. Content Negotiation using HTTP Headers**

HTTP headers provide following 2 type of information for the negotiation

* Content-Type: This header tells about the media type of the body of the request.
* Accept header: Determine what type of representation required on the client side.

#### 

#### **2.1 Content-Type Header**

***Content-Type header*** helps in determining the type of the incoming request (e.g XML, JSON).This header helps server or the client to select the correct approach to handle and eventually parse the data by selecting right converter.Some of the known *Content-Type headers* are

* application/xml
* text/plain
* application/json

Content-Type: application/json  
POST /customers HTTP/1.1  
Content-Type: application/json  
  
{  
 "lastName": "Fielding",  
 "firstName": "Roy"  
}

There are other content headers that can be used with *Content-Type header*

* Content-Encoding: The header indicates what other content encodings applied to the payload.
* Content-Language: The header describes the natural language.

*Server or client can send 415 status code(Unsupported media type) if they do not support the format sent in the request.*

**2.2 The Accept Header**

Client or calling API can specify data format they accept in the response by using *Accept* header.

Accept: application/json  
  
//complete example  
GET /customers HTTP/1.1  
Accept: application/json  
  
HTTP/1.1 200 OK  
Content-Type: application/json  
  
[  
 {  
 "id": 1,  
 "firstName": "Umesh",  
 "lastName": "Awasthi"  
 }  
]

Server API use default content mechanism if no Accept header is present in the request.There are other content headers that can be used with Accept header

* Accept-Language: List of acceptable languages.
* Accept-Encoding: List of acceptable encodings.
* Accept-Charset: Acceptable charset list.

*If the server cannot send data in a format requested in the Accept header, the server sends the 406 Not Acceptable error.*

**3. Content Negotiation using URL Extension**

URL extension or suffix is another *content negotiation strategy* used by multiple systems.Client API can use the extension or suffix in the resource URI to inform expected content type. Let’s take an example to understand this strategy.

https://www.javadevjournal.com/v1/customers/1234.xml  
https://www.javadevjournal.com/v1/customers/1234.json

The first request will return XML response while the second request is for the JSON data.

Similar to the Accept header strategy, server API use default content mechanism when the extension is not provided in the URL.

**11. What is Singleton inside a Request Scope? Or Injecting a prototype/Session bean into a singleton bean**

In Spring, most of the beans we work with are Singletons. If a singleton bean is wired with yet another singleton bean, there is absolutely no problem. But if it is wired with a bean which is of different scope, say prototype, how does it work? Here is the example:

public class RequestProcessor {

private RequestValidator validator;

public void handleRequest(String requestId){

validator.validate(requestId);

// Process the request and update

}

public RequestValidator getValidator() {

return validator;

}

public void setValidator(RequestValidator validator) {

this.validator= validator;

}

}

public class RequestValidator {

private List<String> errorMessages = new ArrayList<String>();

public RequestValidator() {

System.out.println("Validator instance created!");

}

// Validates the request and populates error messages

public void validate(String requestId){ }

public List<String> getErrorMessages() {

return errorMessages;

}

}

And here is the spring configuration:

<bean id="requestProcessor" class="com.pramati.spring.RequestProcessor">

<property name="validator" ref="validator"/>

</bean>

<bean id="validator" scope="prototype"class="com.pramati.spring.RequestValidator"/>

With this configuration, it is expected that when ever I fetch requestProcessor from application context, it will be wired with a new validator as we declared the validator bean is of prototype scope. But this does not happen.

When the application context gets initialized, it sees that requestProcessor is a singleton bean and initializes it to the context after wiring it with all the dependencies set. So from then onwards when we request context for requestProcessor, it return the same bean every time. To solve this issue, we have 2 approaches:

**1. Lookup Method injection:** For this, we have to declare the beans as follows:

<bean id="requestProcessor" class="com.pramati.spring.RequestProcessor">

<lookup-method name="getValidator" bean="validator"/>

</bean>

<bean id="validator" scope="prototype"class="com.pramati.spring.RequestValidator"/>

The Spring Framework implements method injection by using CGLIB library to generate dynamically a subclass that overrides the method. So for the method to be overridden, we have to define that method in the class and either provide a dummy implementation for it or make it abstract. Making a method abstract implies that class also has to be made abstract which will make it difficult to unit test. So providing a dummy implementation is a better choice.

Whenever we define a bean with lookup methods, Spring creates a subclass of the bean and overrides those methods which are marked as lookup-methods. And this subclassed bean gets registered into the context. The subclass delegates all the non-lookup methods to the original class. For the lookup methods, it overrides the implementation. So in our example, when getValidator() is called, it returns a new validator instance.

We can roughly imagine our new subclass(registered in container) like this:

requestProcessor = new RequestProcessor(){

public RequestValidator getValidator(){

return context.getBean("validator");

}

};

We could have directly fetched the bean from application context in RequestProcessor itself. But this would mean that the class is directly coupled to Spring framework. To do this in a cleaner way, we can use lookup injection. This puts all the spring related stuff at one place.

**2. Scoped Proxies**: This can be implemented as:

<bean id="requestProcessor" class="com.pramati.spring.RequestProcessor">

<property name="validator" ref="validator"/>

</bean>

<bean id="validator" scope="prototype"class="com.pramati.spring.RequestValidator">

<!-- This instructs the container to proxy the current bean-->

<aop:scoped-proxy/>

</bean>

Remember, in case of lookup method injection, proxy is created for singleton bean. But in case of scoped proxies, proxy is created for prototype bean and wired into the singleton bean during the process of registering the singleton bean in the context. The proxy thus created understands the scope and returns instances based on the requirements of the scope. So in our case, requestProcessor holds a reference to proxy in place of validator.

And in case of lookup method injection, when requestProcessor gets loaded into the context, validator will not be initialized at all. And when we call the lookup method, it returns the prototype bean. But instead of calling the method, if you try to directly access the prototype bean(assuming it is accessible), it gives a NullPointerException as it didn’t get initialized(We are not wiring it using property tag of bean)

In case of this, we can also configure how a proxy can be created. It can be done in 2 ways

1. CGLIB library which directly subclasses the object. This is the default option of Spring. For this, we must have CGLIB library our class path.
2. Java Dynamic Proxies. For this to be activated, we have to call:

<aop:scoped-proxy proxy-target-class="false"/>

Here in this case, we don’t need any additional libraries in our class path. But the scoped bean must implement at least one interface and it has to be referred through the same interface at all places in order to get itself wired.

**Few points to note:**

1. Both method injection and scoped proxies work not only for prototype beans. This works more generic. Whenever a bean of different scope is injected into a singleton bean, we can use any of these techniques to ensure that we get a corresponding scope object.

2. Note that in the proxy, the method returning the prototype bean is overridden to return a new instance for every single call.

Suppose we want to display the error messages that we have got after validation:

requestProcessor.getValidator().validate();

for(String message: requestProcessor.getValidator().getErrorMessages()){

logger.log(LogLevel.ERROR, message);

}

This code seems to print the error messages we have got after validation process. But this will never print any error messages even if there are many validation failures. This happens because requestProcessor.getValidator() returns a new validator instance every time it is called. So for this to work, the code has to be modified as:

RequestValidator validator = requestProcessor.getValidator();

validator.validate();

for(String message: validator.getErrorMessages()){

logger.log(LogLevel.ERROR, message);

}

This happens only in case of prototype beans but works perfectly in case of other non-singleton scopes(request, session, global-session).

**The singleton scope**

When a bean is a singleton, only one shared instance of the bean will be managed, and all requests for beans with an id or ids matching that bean definition will result in that one specific bean instance being returned by the Spring container.

To put it another way, when you define a bean definition and it is scoped as a singleton, then the Spring IoC container will create exactly one instance of the object defined by that bean definition. This single instance will be stored in a cache of such singleton beans, and all subsequent requests and references for that named bean will result in the cached object being returned.

**The session scope**

With the above bean definition in place, the Spring container will create a brand new instance of the bean , for the lifetime of a single HTTP Session.

According to Spring framework reference, a different approach needs to be followed in cases where a class which "lives longer"(singleton bean in this case) needs to be injected with another class having a comparatively shorter life-span(session-scoped bean). The approach is different for prototype & singleton scope though.

In your XML, what we want is that the singletonBean instance should be instantiated only once, and it should be injected with sessionBean. But since sessionBean is session-scoped(which means it should be re-instantiated for every session), the configuration is ambiguous(as the dependencies are set at instantiation time and the session scoped value can change later also).

So instead of injecting with that class, its injected with a proxy that exposes the exact same public interface as sessionBean. The container injects this proxy object into the singletonBean bean, which is unaware that this sessionBean reference is a proxy. Its specified by writing this tag in the sessionBean:

<aop:scoped-proxy/>

**XML Configuration:**

<bean name="singletonBean" class="somepkg.SingletonBean">  
<property name="someProperty" ref="sessionBean"/>  
</bean>  
  
<bean name="sessionBean" class="somepkg.SessionBean" scope="session">  
<aop:scoped-proxy/>  
</bean>

When a singletonBean instance invokes a method on the dependency-injected sessionBean object, it actually is invoking a method on the proxy. The proxy then fetches the real sessionBean object from (in this case) the HTTP Session, and delegates the method invocation onto the retrieved real sessionBean object.

**Singleton beans with prototype-bean dependencies**

**Lookup Method Injection**

When you use singleton-scoped beans with dependencies on prototype beans, be aware that dependencies are resolved at instantiation time. Thus if you dependency-inject a prototype-scoped bean into a singleton-scoped bean, a new prototype bean is instantiated and then dependency-injected into the singleton bean. The prototype instance is the sole instance that is ever supplied to the singleton-scoped bean.

However, suppose you want the singleton-scoped bean to acquire a new instance of the prototype-scoped bean repeatedly at runtime. You cannot dependency-inject a prototype-scoped bean into your singleton bean, because that injection occurs only once, when the Spring container is instantiating the singleton bean and resolving and injecting its dependencies.

<!-- a stateful bean deployed as a prototype (non-singleton) -->  
<bean id="command" class="fiona.apple.AsyncCommand" scope="prototype">  
 <!-- inject dependencies here as required -->  
</bean>  
  
<!-- commandProcessor uses statefulCommandHelper -->  
<bean id="commandManager" class="fiona.apple.CommandManager">  
 <lookup-method name="createCommand" bean="command"/>  
</bean>

Lookup method injection is the ability of the container to override methods on container managed beans, to return the lookup result for another named bean in the container. The lookup typically involves a prototype bean as in the scenario described in the preceding section. The Spring Framework implements this method injection by using bytecode generation from the CGLIB library to generate dynamically a subclass that overrides the method

**12. What is the difference between @RequestMapping defining in class level and in method level?**

**13. How View Resolver in Spring MVC deals with other formats Ex: PDF, Excel worksheets, XML, Velocity templates, XSLT, JSON, Atom and RSS feeds, JasperReports, etc.**

**14. Are we covering ViewControllerRegistry, ViewResolverRegistry, ResourceHandlerRegistry ?**