**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

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

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

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