Spring:

Spring offered a simpler approach to enterprise Java development, utilizing **dependency injection** and **aspect-oriented programming** to achieve the capabilities of EJB with plain old Java objects (POJOs).

But while Spring was lightweight in terms of component code, it was heavyweight in terms of configuration. Initially, Spring was configured with XML (and lots of it). Spring 2.5 introduced annotation-based component-scanning, which eliminated a great deal of explicit XML configuration for an application’s own components. And Spring 3.0 introduced a Java-based configuration as a type-safe and refactorable option to XML.

Component-scanning reduced configuration and Java configuration made it less awkward, but Spring still required a lot of configuration.

Spring Boot brings a great deal of magic to Spring application development. But there are four core tricks that it performs:

■ Automatic configuration—Spring Boot can automatically provide configuration for application functionality common to many Spring applications.

■ Starter dependencies—You tell Spring Boot what kind of functionality you need, and it will ensure that the libraries needed are added to the build.

■ The command-line interface—This optional feature of Spring Boot lets you write complete applications with just application code, but no need for a traditional project build.

■ The Actuator—Gives you insight into what’s going on inside of a running Spring Boot application.

The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The Spring container uses DI to manage the components that make up an application. These objects are called Spring Beans.

Spring Beans scope is singleton (default, one instance for spring IoC container), prototype, request, session, global-session, this should be given along with id, class and then scope=”singleton” in beans.xml.

**Xml based configuration:**

Beans init and destroy method can be invoked as follow:

<bean id = "exampleBean" class = "examples.ExampleBean" init-method = "init"/>

<bean id = "exampleBean" class = "examples.ExampleBean" destroy-method = "destroy"/>

Bean definition inheritance:

<bean id ="helloIndia" class = "com.tutorialspoint.HelloIndia" parent = "helloWorld">

<property name = "message1" value = "Hello India!"/>

<property name = "message3" value = "Namaste India!"/>

Bean definition template:

<bean id = "beanTeamplate" abstract = "true">

<property name = "message1" value = "Hello World!"/>

<property name = "message2" value = "Hello Second World!"/>

<property name = "message3" value = "Namaste India!"/>

</bean>

Constructor based DI:

<beans>

<bean id = "exampleBean" class = "examples.ExampleBean">

<constructor-arg type = "int" value = "2001"/>

<constructor-arg type = "java.lang.String" value = "Zara"/>

</bean>

</beans>

<!-- Definition for textEditor bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

<constructor-arg ref = "spellChecker"/>

</bean>

<bean id = "exampleBean" class = "examples.ExampleBean">

<constructor-arg index = "0" value = "2001"/>

<constructor-arg index = "1" value = "Zara"/>

</bean>

Setter based DI:

<!-- Definition for textEditor bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

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

</bean>

<bean id = "john-classic" class = "com.example.Person">

<property name = "name" value = "John Doe"/>

<property name = "spouse" ref = "jane"/>

</bean>

Very similar to inner class, we have inner bean defined as below:

A <bean/> element inside the <property/> or <constructor-arg/> elements is called inner bean…

<bean id = "outerBean" class = "...">

<property name = "target">

<bean id = "innerBean" class = "..."/>

</property>

</bean>

<!-- Definition for textEditor bean using inner bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

<property name = "spellChecker">

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker"/>

</property>

</bean>

Property values need not be a primitive datatype, it can be any collection (list, map, set, property) as below:

<!-- Definition for javaCollection -->

<bean id = "javaCollection" class = "com.tutorialspoint.JavaCollection">

<!-- results in a setAddressList(java.util.List) call -->

<property name = "addressList">

<list>

<value>INDIA</value>

<value>Pakistan</value>

<value>USA</value>

<value>USA</value>

</list>

</property>

<!-- results in a setAddressMap(java.util.Map) call -->

<property name = "addressMap">

<map>

<entry key = "1" value = "INDIA"/>

<entry key = "2" value = "Pakistan"/>

<entry key = "3" value = "USA"/>

<entry key = "4" value = "USA"/>

</map>

</property>

<!-- results in a setAddressProp(java.util.Properties) call -->

<property name = "addressProp">

<props>

<prop key = "one">INDIA</prop>

<prop key = "one">INDIA</prop>

<prop key = "two">Pakistan</prop>

<prop key = "three">USA</prop>

<prop key = "four">USA</prop>

</props>

</property>

**Annotation based configuration:**

1 @Required

The @Required annotation applies to bean property setter methods.

2 @Autowired

The @Autowired annotation can apply to bean property setter methods, non-setter methods, constructor and properties.

3 @Qualifier

The @Qualifier annotation along with @Autowired can be used to remove the confusion by specifiying which exact bean will be wired.

4 JSR-250 Annotations

Spring supports JSR-250 based annotations which include @Resource, @PostConstruct and @PreDestroy annotations.

**Java based Configuration**, we can have any number of Java configuration classes

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();

ctx.register(AppConfig.class, OtherConfig.class);

ctx.register(AdditionalConfig.class);

ctx.refresh();

@Configuration

**@Import(ConfigA.class)**

public class ConfigB {

@Bean

public B b() {

return new B();

}

}

@Bean(initMethod = "init", destroyMethod = "cleanup" )

public Foo foo() {

return new Foo();

}

@Bean

@Scope("prototype")

public Foo foo() {

return new Foo();

}

**Spring xml configuration details:**

**Element :** alias

Defines an alias for a bean (which can reside in a different definition resource).

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

In the Spring framework, the IoC (inversion of control) container is represented by the interface ApplicationContext.

The Spring container is responsible for instantiating, configuring and assembling objects known as beans, as well as managing their lifecycle.

**A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.**

**Element :** bean

Defines a single (usually named) bean. A bean definition may contain nested tags for constructor arguments, property values, lookup methods, and replaced methods. Mixing constructor injection and setter injection on the same bean is explicitly supported.

**Content Model :** ((description?, (meta | constructor-arg | property | qualifier | lookup-method | replaced-method | namespace:uri="##other")\*))

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

**Element :** beans

Container for <bean> and other elements, typically the root element in the document. Allows the definition of default values for all nested bean definitions. May itself be nested for the purpose of defining a subset of beans with certain default values or to be registered only when certain profile(s) are active. Any such nested <beans> element must be declared as the last element in the document.

**Content Model :** (description?, (import | alias | bean | namespace:uri="##other")\*, beans\*)

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

**Element :** annotation-config

Activates various annotations to be detected in bean classes: Spring's @Required and @Autowired, as well as JSR 250's @PostConstruct, @PreDestroy and @Resource (if available), JAX-WS's @WebServiceRef (if available), EJB 3's @EJB (if available), and JPA's @PersistenceContext and @PersistenceUnit (if available). Alternatively, you may choose to activate the individual BeanPostProcessors for those annotations. Note: This tag does not activate processing of Spring's @Transactional or EJB 3's @TransactionAttribute annotation. Consider the use of the <tx:annotation-driven> tag for that purpose. See javadoc for org.springframework.context.annotation.AnnotationConfigApplicationContext for information on code-based alternatives to bootstrapping annotation-driven support.

**Content Model :** (namespace:uri="##any")

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

**Element :** component-scan

Scans the classpath for annotated components that will be auto- registered as Spring beans. By default, the Spring-provided @Component, @Repository, @Service, @Controller, @RestController, @ControllerAdvice, and @Configuration stereotypes will be detected. Note: This tag implies the effects of the 'annotation-config' tag, activating @Required, @Autowired, @PostConstruct, @PreDestroy, @Resource, @PersistenceContext and @PersistenceUnit annotations in the component classes, which is usually desired for autodetected components (without external configuration). Turn off the 'annotation-config' attribute to deactivate this default behavior, for example in order to use custom BeanPostProcessor definitions for handling those annotations. Note: You may use placeholders in package paths, but only resolved against system properties (analogous to resource paths). A component scan results in new bean definitions being registered; Spring's PropertySourcesPlaceholderConfigurer will apply to those bean definitions just like to regular bean definitions, but it won't apply to the component scan settings themselves. See javadoc for org.springframework.context.annotation.ComponentScan for information on code-based alternatives to bootstrapping component-scanning.

**Content Model :** (include-filter\*, exclude-filter\*)

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

**Element :** load-time-weaver

Activates a Spring LoadTimeWeaver for this application context, available as a bean with the name "loadTimeWeaver". Any bean that implements the LoadTimeWeaverAware interface will then receive the LoadTimeWeaver reference automatically; for example, Spring's JPA bootstrap support. The default weaver is determined automatically: see DefaultContextLoadTimeWeaver's javadoc for details. The activation of AspectJ load-time weaving is specified via a simple flag (the 'aspectj-weaving' attribute), with the AspectJ class transformer registered through Spring's LoadTimeWeaver. AspectJ weaving will be activated by default if a "META-INF/aop.xml" resource is present in the classpath. This also activates the current application context for applying dependency injection to non- managed classes that are instantiated outside of the Spring bean factory (typically classes annotated with the @Configurable annotation). This will only happen if the AnnotationBeanConfigurerAspect is on the classpath (i.e. spring- aspects.jar), effectively activating "spring-configured" by default. See javadoc for org.springframework.context.annotation.EnableLoadTimeWeaving for information on code-based alternatives to bootstrapping load time weaving support.

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

**Element :** mbean-export

Activates default exporting of MBeans by detecting standard MBeans in the Spring context as well as @ManagedResource annotations on Spring-defined beans. The resulting MBeanExporter bean is defined under the name "mbeanExporter". Alternatively, consider defining a custom AnnotationMBeanExporter bean explicitly.

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

**Element :** mbean-server

Exposes a default MBeanServer for the current platform. Autodetects WebLogic, WebSphere and the JVM's platform MBeanServer. The default bean name for the exposed MBeanServer is "mbeanServer". This may be customized through specifying the "id" attribute.

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

**Element :** property-override

Activates pushing of override values into bean properties, based on configuration lines of the following format: beanName.property=value

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

**Element :** property-placeholder

Activates replacement of ${...} placeholders by registering a PropertySourcesPlaceholderConfigurer within the application context. Properties will be resolved against the specified properties file or Properties object -- so called "local properties", if any, and against the Spring Environment's current set of PropertySources.

Note that as of Spring 3.1 the system-properties-mode attribute has been removed in favor of the more flexible PropertySources mechanism. However, applications may continue to use the 3.0 (and older) versions of the spring-context schema in order to preserve system-properties-mode behavior. In this case, the traditional PropertyPlaceholderConfigurer component will be registered instead of the newer PropertySourcesPlaceholderConfigurer.

See ConfigurableEnvironment javadoc for more information on usage.

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

**Element :** spring-configured

Signals the current application context to apply dependency injection to non-managed classes that are instantiated outside of the Spring bean factory (typically classes annotated with the @Configurable annotation).

**Data Type :** string

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

**Element :** description

Contains informative text describing the purpose of the enclosing element. Used primarily for user documentation of XML bean definition documents.

**Content Model :** ()

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

**Element :** import

Specifies an XML bean definition resource to import.

<import resource="../../bean1.xml"/>

============================================

**Spring 4.0 version used instead of old spring 3.0**

Injection -> we are trying to inject some activities (like creating of class, loading the class, initiating class, setting property default values and etc..) into the xml instead of hardcoding in the java class. With this approach we don’t have to recompile the java class in case of any changes required, we only update the values in xml and re-run it without re-compiling the whole code base.

Xml needs a definition for each tag, so for xmlns, xmlns:xsi,xmlns:context and xsi:schemaLocation get from google…

Dependency Injection => For a class (car) we need its dependency classes (engine, tyre etc) to be injected into car class, this is achieved by using BeanFactory or applicationContext.

Dependency injection is a technique that allows the client code to be independent from the services it is relying on. The client does not control how objects of the services are created - it works with an implementation of the service through interface. This is somewhat in inverse to trivial programming so dependency injection is also called **inversion of control**.

And the Spring framework is based on dependency injection, as well as other frameworks like Google Guice and Play.

Eg:

1. Dependency Injection (Method/setter based, field based, cons
2. tructor based) with the help of spring framework:

Factory design pattern (object creation using abstraction via ApplicationContext) achieved using the

1. bean tag in xml

(**<bean id=”car” class=”com.sri.test.Car”></bean>**) or using

1. Annotation based configuration:

**@Component** on the class and then defining the xml to look for component by mentioning **<context:component-scan> tag (<context:component-scan base-package=”com.sri.test”> </context:component-scan>**)

1. Purely Annotation based configuration (no xml at all)

Create a new Class named AppConf.java as below:

**@Configuration**

public class AppConfig {

**@Bean**

public Samsung getPhone() {

return new Samsung();

} }

And…

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

Samsung s7 = factory.getBean(Samsung.class);

1. Setter Injection:

Initializing the properties of the class: by using property tag inside a bean (setter method is used).

<bean id=”tyre” class=”com.sri.test.Tyre”>

<property name=”brand” value=”MRF”></property>

</bean>

1. Constructor Injection:

By using constructor tag inside bean, it calls the default constructor available in the class definition.

<bean id=”tyre” class=”com.sri.test.Tyre”>

<constructor-arg index="0" value="MRFNEWPrint"></constructor-arg>

<constructor-arg index="1" type="int" value="45"></constructor-arg>

</bean>

1. Autowired annotation:

Inside a class, if we wanted to create instance of any other class and automatically load and initialize it, then we use @Autowired just above the field declaration as:

**@Autowired**

**private Tyre tyre;**

With this, the sprint IoC will try to look for Tyre class and load it, initialize it if configured in xml.

1. Configuration Bean:

For annotation based configuration, we need to write a Configurator class that will be used to inject the actual implementation bean to the component property.

Spring framework with the help of annotations:

@Configuration and @Bean inside AppConfig.java file and then use

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class)

IVehicleAnno obj = (IVehicleAnno) context.getBean(CarAnno.class)

@Configuration annotation is used to let Spring know that it’s a Configuration class.

@ComponentScan annotation is used with @Configuration annotation to specify the packages to look for Component classes.

@Bean annotation is used to let Spring framework know that this method should be used to get the bean implementation to inject in Component classes.

1. If incase we wanted an object to be created by just declaring the reference:
2. Create a bean method in AppConfig.java as below:

@Bean

public Tyre get(){

return new Tyre();

}

1. In the class, where you declared the reference, we have to add Autowired annotation as below:

@Autowired

private Tyre tyre;

tyre.print();

//Then we don’t get the NullPointerException, since the required Tyre objects gets created and loading by spring IoC container with the help of AppConfig Configuration class

1. Annotation Component – Autowired primary qualifier

Or create Beans without defining the @Bean tag in AppConfig.java file

=> Spring Core Annotations:

@Configuration -> Annotation configuration file,

@ComponentScan -> will search all the components and load objects

“@ComponentScan(basePackages="com.sri.springframework.annotation,com.sri.springframework")”

Where we are defining/configuring in spring as bean to be loaded:

@Component annotation is added to the class, so that when Spring framework will scan for the components, this class will be treated as component.

@Bean -> creates the object and make it readily available to be used … and

@Primary -> makes this as primary for the object selection by spring IoC container

Where we are directly using/consuming the beans:

@Component -> particular class will be loading and an object is created for ready use.

@Autowired annotation is used to let Spring know that auto wiring is required. This can be applied to field, constructor and methods. This annotation allows us to implement constructor-based, field-based or method-based dependency injection in our components.

@Qualifier(“car”) -> to match the name

Note: Either @Autowired and @Qualifier (inside where we are consuming) will make spring IoC container to look for unique object, or @Component and @Primary (inside where we are declaring) makes an object unique. Both works.