**SPRING CORE**

**What is ApplicationContext? what are common implementations of ApplicationContext Interface?**

Spring’s ApplicationContext object represents an instance of Spring container. It is basically a central interface to provide configuration for an application. This is read-only while the application is running but may be reloaded if the implementation supports this.

***An ApplicationContext provides-***

Bean factory methods for accessing application components. Inherited from ListableBeanFactory.

The ability to load file resources in a generic fashion. Inherited from the ResourceLoader interface.

The ability to publish events to registered listeners. Inherited from the ApplicationEventPublisher interface.

The ability to resolve messages, supporting internationalization. Inherited from the MessageSource interface.

Inheritance from a parent context. Definitions in a descendant context will always take priority. This means, for example, that a single parent context can be used by an entire web application, while each servlet has its own child context that is independent of that of any other servlet.

***Known Implementing Classes-***

1) AnnotationConfigApplicationContext - Standalone application context, accepting component classes as input — in particular @Configuration-annotated classes, but also plain @Component types and JSR-330 compliant classes using javax.inject annotations. Allows for registering classes one by one using register(Class...) as well as for classpath scanning using scan(String...).

In case of multiple @Configuration classes, @Bean methods defined in later classes will override those defined in earlier classes. This can be leveraged to deliberately override certain bean definitions via an extra @Configuration class.

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();

ctx.register(AppConfig.class);

2) ClassPathXmlApplicationContext - ClassPathXmlApplicationContext can load an XML configuration from a classpath and manage its beans. The config location defaults can be overridden via AbstractRefreshableConfigApplicationContext.getConfigLocations(), Config locations can either denote concrete files like "/myfiles/context.xml" or Ant-style patterns like "/myfiles/\*-context.xml".

It is a standalone XML application context, taking the context definition files from the class path, interpreting plain paths as class path resource names that include the package path (e.g. "mypackage/myresource.txt"). Useful for test harnesses as well as for application contexts embedded within JARs.

***ApplicationContext context = new ClassPathXmlApplicationContext("xmlapplicationcontext-example.xml");***

***Student student = (Student) context.getBean("student");***

3) FileSystemXmlApplicationContext - FileSystemXmlApplicationContext is XML ApplicationContext. It works standalone. It can pick the XML file from absolute path by appending keyword "file:" and also can fetch from classpath by appending keyword "classpath:".

Plain paths will always be interpreted as relative to the current VM working directory, even if they start with a slash. Use an explicit "file:" prefix to enforce an absolute file path. Config locations can either denote concrete files like "/myfiles/context.xml" or Ant-style patterns like "/myfiles/\*-context.xml".

In case of multiple config locations, later bean definitions will override ones defined in earlier loaded files. This can be leveraged to deliberately override certain bean definitions via an extra XML file.

***ApplicationContext context = new FileSystemXmlApplicationContext("file:c:/beans.xml");***

***SayHello hello = (SayHello) context.getBean("hello");***

4) XmlWebApplicationContext - XmlWebApplicationContext takes its configuration from XML documents. By default, the configuration will be taken from "/WEB-INF/applicationContext.xml" for the root context, and "/WEB-INF/test-servlet.xml" for a context with the namespace "test-servlet" (like for a DispatcherServlet instance with the servlet-name "test"). Config locations can either denote concrete files like "/WEB-INF/context.xml" or Ant-style patterns like "/WEB-INF/\*-context.xml". In case of multiple config locations, later bean definitions will override ones defined in earlier loaded files. This can be leveraged to deliberately override certain bean definitions via an extra XML file.

***XmlWebApplicationContext context = new XmlWebApplicationContext();***

***context.setConfigLocation("/WEB-INF/beans.xml");***

***context.setServletContext(request.getServletContext());***

***context.refresh();***

***SayHello hello = (SayHello) context.getBean("hello");***

**What is difference between BeanFactory and ApplicationContext?**

Spring beans are Java objects that are managed by the Spring container.

In the XML-based configuration, 'beans.xml' supplies the metadata for the Spring container to manage the bean.

The Spring container is responsible for instantiating, configuring, and assembling the Spring beans.

Once bean is managed by the Spring container. The only question is: how we can access it?

***The BeanFactory Interface-***

This is the root interface for accessing the Spring container. To access the Spring container, we will be using Spring's dependency injection functionality using this BeanFactory interface and its sub-interfaces. Usually, the implementations use lazy loading, which means that beans are only instantiating when we directly calling them through the getBean() method.

The most used API that implements the BeanFactory is the XmlBeanFactory.

***XmlBeanFactory factory = new XmlBeanFactory (new ClassPathResource("beans.xml"));***

***HelloWorld obj = (HelloWorld) factory.getBean("helloWorld");***

The ApplicationContext Interface

The ApplicationContext is the central interface within a Spring application that is used for providing configuration information to the application.

It extends the BeanFactory interface. Hence, the ApplicationContext includes all functionality of the BeanFactory and much more!

It uses eager loading, so every bean instantiate after the ApplicationContext is started up.

Here is an example of the ApplicationContext usage:

**ApplicationContext context = new ClassPathXmlApplicationContext("xmlapplicationcontext-example.xml");**

**Student student = (Student) context.getBean("student");**

BeanFactory is also called basic IOC and ApplicationContext is called Advanced IOC. Although BeanFactory and ApplicationContext both are used to get the beans from IOC container by using method getBean(String beanName). But they have some significant differences in their implementation which are described as below :

1. BeanFactory uses lazy initialization approach whereas ApplicationContext uses eager initialization approach. i.e BeanFactory creates a singleton bean only when it is requested from it but ApplicationContext creates all singleton beans at the time of its own initialization.

2. ApplicationContext creates and manages resources objects on its own whereas BeanFactory used to be explicitly provided a resource object using the syntax :

***ClassPathResource resource = new ClassPathResource("beans.xml");***

***XmlBeanFactory factory = new XmlBeanFactory(resource); // Here resource object is provided explicitly***

3. ApplicationContext supports internationalization but BeanFactory do not.

4. Annotation based dependency Injection is not supported by BeanFactory whereas ApplicationContext supports using annotation @PreDestroy, @Autowired.

***BeanFactory Features:***

* Bean instantiation/wiring

***ApplicationContext Features:***

* Bean instantiation/wiring
* Automatic BeanPostProcessor registration
* Automatic BeanFactoryPostProcessor registration
* Convenient MessageSource access (for i18n)
* ApplicationEvent publication

The ApplicationContext includes all the functionality of the BeanFactory. It is generally recommended to use the former. There are some limited situations, such as in mobile applications, where memory consumption might be critical. In those scenarios, it would be justifiable to use the more lightweight BeanFactory. However, in most enterprise applications, the ApplicationContext is what you will want to use.

***Difference between @Configuration and @Component -***

@Configuration Indicates that a class declares one or more @Bean methods and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime.

@Component Indicates that an annotated class is a "component". Such classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning.

@Configuration is meta-annotated with @Component, therefore @Configuration classes are candidates for component scanning

***@Bean Lite Mode -***

@Bean methods may also be declared within classes that are not annotated with @Configuration. For example, bean methods may be declared in a @Component class or even in a plain old class. In such cases, a @Bean method will get processed in a so-called 'lite' mode.

Bean methods in lite mode will be treated as plain factory methods by the container (similar to factory-method declarations in XML), with scoping and lifecycle callbacks properly applied. The containing class remains unmodified in this case, and there are no unusual constraints for the containing class or the factory methods.

In contrast to the semantics for bean methods in @Configuration classes, 'inter-bean references' are not supported in lite mode. Instead, when one @Bean-method invokes another @Bean-method in lite mode, the invocation is a standard Java method invocation; Spring does not intercept the invocation via a CGLIB proxy.

***Differences between @Component and @Bean -***

1)@Component is a class level annotation where as @Bean is a method level annotation and name of the method serves as the bean name.

@Component auto detects and configures the beans using classpath scanning (@ComponentScan), whereas @Bean explicitly declares a single bean, rather than letting Spring do it automatically.

2)@Bean

First its a method level annotation . Second you generally use to configure beans in a java code (if you are not using xml configuration) and then call it from a class using ApplicationContext's getBean method like

***@Configuration***

***class MyConfiguration{***

***@Bean***

***public User getUser(){***

***return new User();***

***}***

***}***

***class User{***

***}***

***//Getting Bean***

***User user = applicationContext.getBean("getUser");***

***@Component***

It is general way to annotate a bean and not a specialized bean. A class level annotation and is used to avoid all that configuration stuff through java or xml configuration.

***@Component***

***class User {***

***}***

***//to get Bean***

***@Autowired***

***User user;***

3)You can use @Bean to make an existing third-party class available to your Spring framework application context.

***@Bean***

***public ViewResolver viewResolver() {***

***InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();***

***viewResolver.setPrefix("/WEB-INF/view/");***

***viewResolver.setSuffix(".jsp");***

***return viewResolver;***

***}***

By using the @Bean annotation, you can wrap a third-party class (it may not have @Component and it may not use Spring), as a Spring bean. And then once it is wrapped using @Bean, it is as a singleton object and available in your Spring framework application context. You can now easily share/reuse this bean in your app using dependency injection and @Autowired.

***@Bean methods***

@Bean methods are declared within @Configuration classes.

BeanFactoryPostProcessor-returning @Bean methods

Special consideration must be taken for @Bean methods that return Spring BeanFactoryPostProcessor (BFPP) types. Because BFPP objects must be instantiated very early in the container lifecycle, they can interfere with processing of annotations such as @Autowired, @Value, and @PostConstruct within @Configuration classes. To avoid these lifecycle issues, mark BFPP-returning @Bean methods as static. For example:

***@Bean***

***public static PropertySourcesPlaceholderConfigurer pspc() {***

***// instantiate, configure and return pspc ...***

***}***

By marking this method as static, it can be invoked without causing instantiation of its declaring @Configuration class, thus avoiding the above-mentioned lifecycle conflicts.

***@Bean***

***public Coach swimCoach() {***

***SwimCoach mySwimCoach = new SwimCoach();***

***return mySwimCoach;***

***}***

It is important to note that this method has the @Bean annotation. The annotation will intercept ALL calls to the method "swimCoach()". Since no scope is specified the @Bean annotation uses singleton scope. Behind the scenes, during the @Bean interception, it will check in memory of the Spring container (applicationContext) and see if this given bean has already been created.

If this is the first time the bean has been created then it will execute the method as normal. It will also register the bean in the application context. So that is knows that the bean has already been created before. Effectively setting a flag.

The next time this method is called, the @Bean annotation will check in memory of the Spring container (applicationContext) and see if this given bean has already been created. Since the bean has already been created (previous paragraph) then it will immediately return the instance from memory. It will not execute the code inside of the method. Hence this is a singleton bean.

In example below, we are creating a SwimCoach and injecting the sadFortuneService().

// define bean for our sad fortune service

***@Bean***

***public FortuneService sadFortuneService() {***

***return new SadFortuneService();***

***}***

// define bean for our swim coach AND inject dependency

***@Bean***

***public Coach swimCoach() {***

***SwimCoach mySwimCoach = new SwimCoach(sadFortuneService());***

***return mySwimCoach;***

***}***

This code creates an instance of SwimCoach. Note the call to the method sadFortuneService(). We are calling the annotated method above. The @Bean will intercept and return a singleton instance of sadFortuneService. The sadFortuneService is then injected into the swim coach instance.

This is effectively dependency injection. It is accomplished using all Java configuration (no xml).

While a name() attribute is available, the default strategy for determining the name of a bean is to use the name of the @Bean method. This is convenient and intuitive, but if explicit naming is desired, the name attribute (or its alias value) may be used. Also note that name accepts an array of Strings, allowing for multiple names (i.e. a primary bean name plus one or more aliases) for a single bean.\*/

@Bean({"b1", "b2"}) // bean available as 'b1' and 'b2', but not 'myBean'. But if attributes are absent then name of bean will be 'myBean'

***public MyBean myBean() {***

***// instantiate and configure MyBean obj***

***return obj;***

***}***

***Annotations typically used with @Bean annotation -***

Note that the @Bean annotation does not provide attributes for profile, scope, lazy, depends-on or primary. Rather, it should be used in conjunction with @Scope, @Lazy, @DependsOn and @Primary annotations to declare those semantics.

1) @Profile allows for selective inclusion of certain beans.

2) @Scope changes the bean's scope from singleton to the specified scope.

3) @Lazy only has an actual effect in case of the default singleton scope. 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’re cases when we need to create a bean, not at the application context startup, but when we request it.

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

***Example A -***

***@Bean***

***@Lazy(true)***

***public Area getArea(){***

***return new Area();***

***}***

With @Autowired

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

The bean that we want to load lazily:

***Example B -***

***@Lazy***

***@Component***

***public class City {***

***public City() {***

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

***}***

***}***

***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.

4) @DependsOn enforces the creation of specific other beans before this bean will be created, in addition to any dependencies that the bean expressed through direct references, which is typically helpful for singleton startup.

5) @Primary is a mechanism to resolve ambiguity at the injection point level if a single target component needs to be injected but several beans match by type.

6) Additionally, @Bean methods may also declare qualifier annotations and @Order values.

Qualifiers narrow the set of candidates after the initial type match; order values determine the order of resolved elements in case of collection injection points (with several target beans matching by type and qualifier).

***@Import annotation-***

// Indicates one or more @Configuration classes to import. If XML or other non-@Configuration bean definition resources need to be imported, use the @ImportResource annotation instead.

//@Import({WebSecurityConfig.class})

If component scanning is enabled, you can split bean definitions in multi @Configuration classes without using @Import. You don't need to provide all of them to the application context constructor. I think the main purpose for @Import is to provide you a way to simplify multi configuration registration if you'd like to avoid component scanning.

@Bean definitions declared in imported @Configuration classes should be accessed by using @Autowired injection. Either the bean itself can be autowired, or the configuration class instance declaring the bean can be autowired.

@Import annotation you can import one or more @Configuration classes.

It can also import classes containing at least one @Bean method but no @Configuration annotation (Just BeanHolder class)

@Configuration classes may be composed using the @Import annotation, similar to the way that <import> works in Spring XML. Because @Configuration objects are managed as Spring beans within the container, imported configurations may be injected — for example, via constructor injection:

***@Configuration***

***public class DatabaseConfig {***

***@Bean***

***public DataSource dataSource() {***

***// instantiate, configure and return DataSource***

***}***

***}***

***@Configuration***

***@Import(DatabaseConfig.class)***

***public class AppConfig {***

***private final DatabaseConfig dataConfig; // @Autowired here implicit??***

***public AppConfig(DatabaseConfig dataConfig) {***

***this.dataConfig = dataConfig;***

***}***

***@Bean***

***public MyBean myBean() {***

***// reference the dataSource() bean method***

***return new MyBean(dataConfig.dataSource());***

***}***

***}***

Now both AppConfig and the imported DatabaseConfig can be bootstrapped by registering only AppConfig against the Spring context:

new AnnotationConfigApplicationContext(AppConfig.class);

To customize the imported configuration from WebMvcConfigurationSupport, implement the interface WebMvcConfigurer and override individual methods as shown in below example:

***@Configuration***

***@EnableWebMvc***

***@ComponentScan(basePackageClasses = MyConfiguration.class)***

***public class MyConfiguration implements WebMvcConfigurer {***

***@Override***

***public void addFormatters(FormatterRegistry formatterRegistry) {***

***formatterRegistry.addConverter(new MyConverter());***

***}***

***@Override***

***public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {***

***converters.add(new MyHttpMessageConverter());***

***}***

***}***

// Only one @Configuration class may have the @EnableWebMvc annotation to import the Spring Web MVC configuration. There can however be multiple @Configuration classes implementing WebMvcConfigurer in order to customize the provided configuration.

If WebMvcConfigurer does not expose some more advanced settings that needs to be configured, consider removing the @EnableWebMvc annotation and extending directly from WebMvcConfigurationSupport or DelegatingWebMvcConfiguration.

***@Configuration***

***@ComponentScan(basePackageClasses = { MyConfiguration.class })***

***public class MyConfiguration extends WebMvcConfigurationSupport {***

***@Override***

***public void addFormatters(FormatterRegistry formatterRegistry) {***

***formatterRegistry.addConverter(new MyConverter());***

***}***

***@Bean***

***public RequestMappingHandlerAdapter requestMappingHandlerAdapter() {***

***// Create or delegate to "super" to create and***

***// customize properties of RequestMappingHandlerAdapter***

***}***

***}***

***@Configuration classes bootstrapping-***

@Configuration classes are typically bootstrapped using either AnnotationConfigApplicationContext or its web-capable variant, AnnotationConfigWebApplicationContext. A simple example with the former follows:

***AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();***

***ctx.register(AppConfig.class);***

***ctx.refresh();***

***MyBean myBean = ctx.getBean(MyBean.class);***

***// use myBean ...***

@Configuration classes can also be bootstrapped Via component scanning. @Configuration classes may not only be bootstrapped using component scanning, but may also themselves configure component scanning using the @ComponentScan annotation:

***@Configuration***

***@ComponentScan("com.acme.app.services")***

***public class AppConfig {***

***// various @Bean definitions ...***

***}***

**How @Bean annotation works behind the scenes?**

This is an advanced concept. But I'll walk through the code line-by-line.

For this code:

***@Bean***

***public Coach swimCoach() {***

***SwimCoach mySwimCoach = new SwimCoach();***

***return mySwimCoach;***

***}***

At a high-level, Spring creates a bean component manually. By default the scope is singleton. So any request for a "swimCoach" bean, will get the same instance of the bean since singleton is the default scope.

However, let's break it down line-by-line

@Bean

The @Bean annotation tells Spring that we are creating a bean component manually. We didn't specify a scope so the default scope is singleton.

public Coach swimCoach(){

This specifies that the bean will bean id of "swimCoach". The method name determines the bean id. The return type is the Coach interface. This is useful for dependency injection. This can help Spring find any dependencies that implement the Coach interface.

The @Bean annotation will intercept any requests for "swimCoach" bean. Since we didn't specify a scope, the bean scope is singleton. As a result, it will give the same instance of the bean for any requests.

SwimCoach mySwimCoach = new SwimCoach();

This code will create a new instance of the SwimCoach.

return mySwimCoach;

This code returns an instance of the swimCoach.

Now let's step back and look at the method in it's entirety.

***@Bean***

***public Coach swimCoach() {***

***SwimCoach mySwimCoach = new SwimCoach();***

***return mySwimCoach;***

***}***

It is important to note that this method has the @Bean annotation. The annotation will intercept ALL calls to the method "swimCoach()". Since no scope is specified the @Bean annotation uses singleton scope. Behind the scenes, during the @Bean interception, it will check in memory of the Spring container (applicationContext) and see if this given bean has already been created.

If this is the first time the bean has been created then it will execute the method as normal. It will also register the bean in the application context. So that is knows that the bean has already been created before. Effectively setting a flag.

The next time this method is called, the @Bean annotation will check in memory of the Spring container (applicationContext) and see if this given bean has already been created. Since the bean has already been created (previous paragraph) then it will immediately return the instance from memory. It will not execute the code inside of the method. Hence this is a singleton bean.

The code for

***SwimCoach mySwimCoach = new SwimCoach();***

***return mySwimCoach;***

is not executed for subsequent requests to the method public Coach swimCoach() . This code is only executed once during the initial bean creation since it is singleton scope.

That explains how @Bean annotation works for the swimCoach example.

Now let's take it one step further.

Here's your other question

>> Please explain in detail whats happening behind the scene for this statement.

return new SwimCoach(sadFortuneService())

The code for this question is slightly different. It is injecting a dependency.

In this example, we are creating a SwimCoach and injecting the sadFortuneService().

***// define bean for our sad fortune service***

***@Bean***

***public FortuneService sadFortuneService() {***

***return new SadFortuneService();***

***}***

***// define bean for our swim coach AND inject dependency***

***@Bean***

***public Coach swimCoach() {***

***SwimCoach mySwimCoach = new SwimCoach(sadFortuneService());***

***return mySwimCoach;***

***}***

Using the same information presented earlier

The code

***// define bean for our sad fortune service***

***@Bean***

***public FortuneService sadFortuneService() {***

***return new SadFortuneService();***

***}***

In the code above, we define a bean for the sad fortune service. Since the bean scope is not specified, it defaults to singleton.

Any calls for sadFortuneService, the @Bean annotation intercepts the call and checks to see if an instance has been created. First time through, no instance is created so the code executes as desired. For subsequent calls, the singleton has been created so @Bean will immediately return with the singleton instance.

Now to the main code based on your question.

return new SwimCoach(sadFortuneService())

This code creates an instance of SwimCoach. Note the call to the method sadFortuneService(). We are calling the annotated method above. The @Bean will intercept and return a singleton instance of sadFortuneService. The sadFortuneService is then injected into the swim coach instance.

This is effectively dependency injection. It is accomplished using all Java configuration (no xml).

**What is dependency injection and what are the advantages?**

Dependency injection (DI) is a process whereby objects define their dependencies only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean

***DI Benefits -***

1) Reduced glue boilerplate code, so code is cleaner.

2) Decoupling is more effective (IOC containers support eager instantiation and lazy loading of services)

3) Easier to test (no singletons or JNDI lookup mechanisms are required in unit tests)

4) Better applications design with DI

5) Increased module re-usability.

6) Increased maintainability.

7) Standardizes parts of application development.

A Java application consists of objects that interact with each other to provide application behavior. The objects with which an object interacts are referred to as its dependencies. For instance, if an object X interacts with objects Y and Z, then Y and Z are dependencies of object X. DI (short for 'Dependency Injection') is a design pattern in which the dependencies of an object are typically specified as arguments to its constructor and setter methods. And, these dependencies are injected into the object when it’s created.

In a Spring application, Spring IoC container (also referred to as 'Spring container') is responsible for creating application objects and injecting their dependencies. The application objects that the Spring container creates and manages are referred as beans. As the Spring container is responsible for putting together application objects, you don’t need to implement design patterns, like Factory, Service Locater, and so on, to compose your application. DI is also referred to as Inversion of Control (IoC) because the responsibility of creating and injecting dependencies is not with the application object, but with the Spring container.

**What is an interface and what are the advantages of making use of them in Java? Why are they recommended for Spring beans?**

Interfaces cannot be instantiated and it’s a way of implementing multiple inheritance (polymorphism).

***Advantages -***

* Providing different implementations at runtime
* The ability to inject dependencies
* Polymorphism

**Why Interfaces are recommended for Spring beans?**

Spring’s DI implementation is based on two core Java concepts: JavaBeans and interfaces.

1) JavaBeans (POJOs): Any Spring-managed resource is referred to as a bean.

2) Using interfaces you can get the most out of DI because your beans can utilize any interface implementation to satisfy their dependency. The use of interfaces also allows Spring to utilize JDK dynamic proxies to provide powerful concepts such as AOP for crosscutting concerns.

* Increased testability, by mocking or stubbing
* JDK dynamic proxying
* Easy dependency injection

**What is meant by “application-context"?**

Spring’s ApplicationContext object represents an instance of Spring container. Spring provides a few built-in implementations of ApplicationContext interface like AnnotationConfigApplicationContext, ClassPathXmlApplicationContext, FileSystemXmlApplicationContext, XmlWebApplicationContext, and so on.

The choice of the ApplicationContext implementation depends on how you have defined the configuration metadata (using XML, or annotations ), and the type of your application (standalone or web). For instance, ClassPathXmlApplicationContext and FileSystemXmlApplicationContext classes are suitable for standalone applications in which configuration metadata is supplied in XML format, XmlWebApplicationContext is suitable for web applications in which the configuration metadata is supplied in XML format, AnnotationConfigWebApplicationContext is suitable for web applications in which configuration metadata is supplied programmatically through Java code, and so on.

ApplicationContext interface represents the Spring's IoC container and is responsible for instantiating, configuring, and assembling the beans.

In Spring-based applications, information about application objects and their dependencies is specified using configuration metadata. Spring IoC container reads application’s configuration metadata to instantiate application objects and inject their dependencies.

You provide the configuration metadata for an application in XML format by creating an application context XML file or you can also supply the configuration metadata to the Spring container through Java classes annotated with Spring’s @Configuration annotation.

BeanFactory and ApplicationContext both are Java interfaces and ApplicationContext extends BeanFactory. In short BeanFactory provides basic Inversion of control (IoC) and Dependency Injection (DI) features while ApplicationContext provides advanced features.

The application objects defined via <bean> elements are created and managed by the Spring container. You can access instances of these application objects by calling one of the getBean methods of the ApplicationContext interface.

**How are you going to create a new instance of an ApplicationContext?**

The choice of the ApplicationContext implementation depends on how you have defined the configuration metadata (using XML, or annotations ), and the type of your application (standalone or web). For instance, ClassPathXmlApplicationContext and

FileSystemXmlApplicationContext classes are suitable for standalone applications in which configuration metadata is supplied in XML format, XmlWebApplicationContext is suitable for web applications in which the configuration metadata is supplied in XML format, AnnotationConfigWebApplicationContext is suitable for web applications in which configuration metadata is supplied programmatically through Java code, and so on.

1) ***AnnotationConfigApplicationContext*** - Standalone application context, accepting component classes as input — in particular @Configuration-annotated classes, but also plain @Component types and JSR-330 compliant classes using javax.inject annotations.

Allows for registering classes one by one using register(Class...) as well as for classpath scanning using scan(String...).

In case of multiple @Configuration classes, @Bean methods defined in later classes will override those defined in earlier classes. This can be leveraged to deliberately override certain bean definitions via an extra @Configuration class.

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();

ctx.register(AppConfig.class);

2) ***ClassPathXmlApplicationContext*** - ClassPathXmlApplicationContext can load an XML configuration from a classpath and manage its beans. The config location defaults can be overridden via AbstractRefreshableConfigApplicationContext.getConfigLocations(), Config locations can either denote concrete files like "/myfiles/context.xml" or Ant-style patterns like "/myfiles/\*-context.xml".

It is a standalone XML application context, taking the context definition files from the class path, interpreting plain paths as class path resource names that include the package path (e.g. "mypackage/myresource.txt"). Useful for test harnesses as well as for application contexts embedded within JARs.

ApplicationContext context = new ClassPathXmlApplicationContext("xmlapplicationcontext-example.xml");

Student student = (Student) context.getBean("student");

3) ***FileSystemXmlApplicationContext*** - FileSystemXmlApplicationContext is XML ApplicationContext. It works standalone. It can pick the XML file from absolute path by appending keyword "file:" and also can fetch from classpath by appending keyword "classpath:".

Plain paths will always be interpreted as relative to the current VM working directory, even if they start with a slash. Use an explicit "file:" prefix to enforce an absolute file path. Config locations can either denote concrete files like "/myfiles/context.xml" or Ant-style patterns like "/myfiles/\*-context.xml".

In case of multiple config locations, later bean definitions will override ones defined in earlier loaded files. This can be leveraged to deliberately override certain bean definitions via an extra XML file.

ApplicationContext context = new FileSystemXmlApplicationContext("file:c:/beans.xml");

SayHello hello = (SayHello) context.getBean("hello");

4) ***XmlWebApplicationContext*** - XmlWebApplicationContext takes its configuration from XML documents. By default, the configuration will be taken from "/WEB-INF/applicationContext.xml" for the root context, and "/WEB-INF/test-servlet.xml" for a context with the namespace "test-servlet" (like for a DispatcherServlet instance with the servlet-name "test"). Config locations can either denote concrete files like "/WEB-INF/context.xml" or Ant-style patterns like "/WEB-INF/\*-context.xml". In case of multiple config locations, later bean definitions will override ones defined in earlier loaded files. This can be leveraged to deliberately override certain bean definitions via an extra XML file.

XmlWebApplicationContext context = new XmlWebApplicationContext();

context.setConfigLocation("/WEB-INF/beans.xml");

context.setServletContext(request.getServletContext());

context.refresh();

SayHello hello = (SayHello) context.getBean("hello");

**Can you describe the lifecycle of a Spring Bean in an ApplicationContext?**

It’s important to understand the lifecycle of a Spring bean, because you may want to take advantage of some of the opportunities that Spring offers to customize how a bean is created.

***Orders -***

Spring bean configuration is read and metadata in the form of a BeanDefinition object is created for each bean.

All instances of BeanFactoryPostProcessor are invoked in sequence and are allowed an opportunity to alter the bean metadata.

For each bean in the container, starts creation phase

The beans are instantiated: the bean factory is calling the constructor of each bean. If the bean is created using constructor dependency injection, the dependency bean is created first and then injected where needed .

Dependencies are injected via setter.

From this point. Instantiation is completed. Now, bean post process beans are invoked before initialization. The preinitialization BeanPostProcessor infrastructure beans are consulted to see whether they want to call anything from this bean. These are Spring-specific infrastructure beans that perform bean modifications after they are created.

1)The @PostConstruct annotation method is called, which is registered by CommonAnnotationBeanPostProcessor.

2)The InitializingBean’s afterPropertiesSet() method is invoked by a BeanFactory after it has set all the bean properties supplied and has satisfied BeanFactoryAware and ApplicationContextAware.

3)The init-method attribute is executed last because this is the actual initialization method of the bean.

Beans are being used

Spring application context is to shut down, the beans in it will receive destruction callbacks in this order:

1) @PreDestroy

2) destroy() as defined by the DisposableBean callback interface

3) A custom configured destroy() method

***Combining lifecycle mechanisms-***

***@PostConstruct and @PreDestroy JSR-250 annotations. Best practice!***

***public class Bar {***

***@PostConstruct***

***public void init() throws Exception {***

***System.out.println("init method is called");***

***}***

***@PreDestroy***

***public void destroy() throws RuntimeException {***

***System.out.println("destroy method is called");***

***}***

***}***

The InitializingBean and DisposableBean. Beans can define callback methods, which can be invoked by the container BeanPostProcessor. Do not use!

In InitializingBean, container calls afterproperteisSet().

In DisposableBean, container callsdestroy().

***public class Baz implements InitializingBean, DisposableBean {***

***@Override***

***public void afterPropertiesSet() throws Exception {***

***System.out.println("init method invoked");***

***}***

***@Override***

***public void destroy() throws Exception {***

***System.out.println("destroy method invoked");***

***}***

***}***

Custom init() and destroy() methods from Bean. Not bad.

***public class Foo {***

***public void init() {***

***System.out.println("init method is called");***

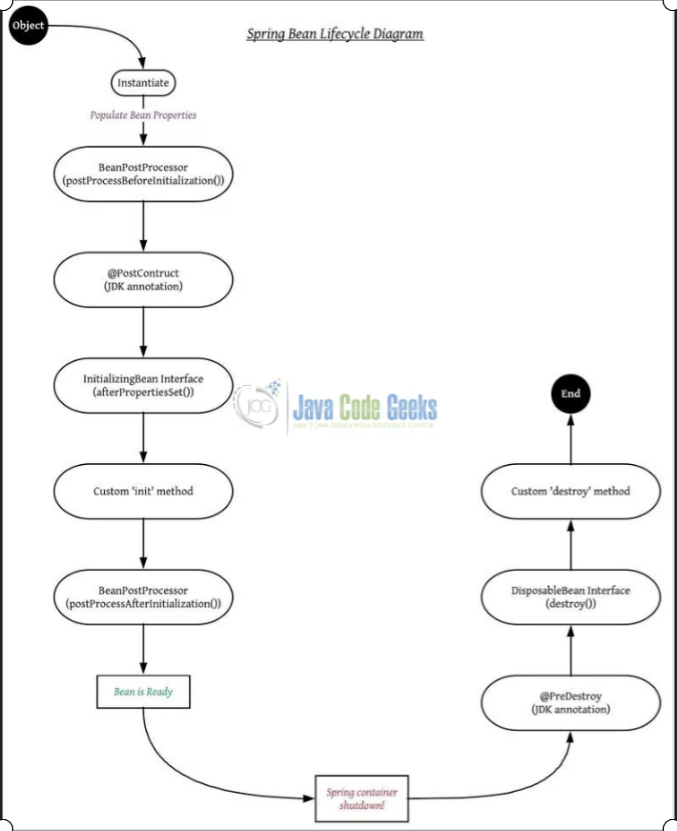
***}***

***public void destroy() {***

***System.out.println("destroy method is called");***

***}***

***}***



**How are you going to create an ApplicationContext in an integration test?**

ContextLoader is a strategy interface for loading an ApplicationContext for an integration test managed by the Spring TestContext Framework.

Its sub-interface SmartContextLoader to provide support for annotated classes, active bean definition profiles, test property sources, context hierarchies, and WebApplicationContext support.

***Implementations of SmartContextLoader:***

***AnnotationConfigContextLoader:*** Loads a standard ApplicationContext from annotated classes.

***AnnotationConfigWebContextLoader:*** Loads a WebApplicationContext from annotated classes.

Annotations @WebAppConfiguration is a class-level annotation that you can use to declare that the ApplicationContext loaded for an integration test should be a WebApplicationContext.

@ContextConfiguration defines class-level metadata that is used to determine how to load and configure an ApplicationContext for integration tests.

In the following example that uses path-based resource locations, the ApplicationContext for ExtendedTest will be loaded from "base-context.xml" and "extended-context.xml", in that order. Beans defined in "extended-context.xml" may therefore override those defined in "base-context.xml".

***@ContextConfiguration("base-context.xml")***

***public class BaseTest {***

***// ...***

***}***

***@ContextConfiguration("extended-context.xml")***

***public class ExtendedTest extends BaseTest {***

***// ...***

***}***

In the following example that uses component classes, the ApplicationContext for ExtendedTest will be loaded from the BaseConfig and ExtendedConfig configuration classes, in that order. Beans defined in ExtendedConfig may therefore override those defined in BaseConfig.

***@ContextConfiguration(classes=BaseConfig.class)***

***public class BaseTest {***

***// ...***

***}***

***@ContextConfiguration(classes=ExtendedConfig.class)***

***public class ExtendedTest extends BaseTest {***

***// ...***

***}***

@DirtiesContext is a test annotation which indicates that the ApplicationContext associated with a test is dirty and should therefore be closed and removed from the context cache. Use this annotation if a test has modified the context — for example, by modifying the state of a singleton bean, modifying the state of an embedded database, etc. Subsequent tests that request the same context will be supplied a new context. @DirtiesContext may be used as a class-level and method-level annotation within the same class or class hierarchy.

Note that @WebAppConfiguration must be used in conjunction with @ContextConfiguration, either within a single test class or within a test class hierarchy.

***@RunWith(SpringRunner.class)***

***@WebAppConfiguration***

***@ContextConfiguration (classes=AppConfig.class)***

***public class MyWebAppTest {***

***@Autowired***

***private WebApplicationContext wac;***

***}***

**What is the preferred way to close an application context? Does Spring Boot do this for you?**

A Spring application has a lifecycle composed of three phases:

***Initialization:*** In this phase, bean definitions are read, beans are created, dependencies are injected, and resources are allocated, also known as the bootstrap phase. After this phase is complete, the application can be used.

***Running:*** In this phase, the application is up and running. It is used by clients, and beans are retrieved and used to provide responses for their requests. This is the main phase of the lifecycle and covers 99% of it.

***Destruction:*** The context is being shut down, resources are released, and beans are handed over to the garbage collector. But some beans work with resources that might refuse to release them if they are not notified before destruction.

When Spring application context is to shut down, the beans receive destruction callbacks in this order:

* @PreDestroy
* destroy() as defined by the DisposableBean callback interface
* A custom configured destroy() method.

However, Spring doesn’t fire above destruction callbacks automatically.

For web application runs as a servlet, you can simply call destroy() in the servlet’s destroy() method.

For stand-alone application, things are not simple, especially if you have multiple exit points out of your application. In stand-alone/desktop application we need to call registerShutdownHook to shutdown IoC container gracefully.

***Solution:*** use AbstractApplicationContext’s registerShutdownHook() method. The method automatically instructs Spring to register a shutdown hook of the underlying JVM runtime. After it is added, calls to ctx.destroy() or close() will be removed.

public class DestructiveBeanWithHook {

public static void main(String... args) {

GenericApplicationContext ctx = new AnnotationConfigApplicationContext( DestructiveBeanConfig.class);

ctx.getBean(DestructiveBeanWithJSR250.class);

ctx.registerShutdownHook();// no need to call ctx.destroy() or close()

}

}

***Spring Boot Application -***

Spring boot Application registers a shutdown hook with the JVM to make sure the application exits appropriately.

Beans may implement the ExitCodeGenerator interface to return a specific error code:

***ConfigurableApplicationContext ctx = new SpringApplicationBuilder(Application.class)***

***.web(WebApplicationType.NONE).run();***

***int exitCode = SpringApplication.exit(ctx, new ExitCodeGenerator() {***

***@Override***

***public int getExitCode() {***

***// return the error code***

***return 0;***

***}***

***});***

***System.exit(exitCode);***

The same code with the application of Java 8 lambdas:

***SpringApplication.exit(ctx, () -> 0);***

After calling the System.exit(exitCode), the program terminates with a 0 return code:

Process finished with exit code 0

**What are different ways to shut down a Spring Boot Application?**

There are other ways to shut down a Spring Boot Application as below -

1) Actuator Shutdown Endpoint

By default, all the endpoints are enabled in Spring Boot Application except /shutdown; this is, naturally, part of the Actuator endpoints.

Here's the Maven dependency to set up these up:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

And, if we want to also set up security support, we need:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

Lastly, we enable the shutdown endpoint in application.properties file:

management.endpoints.web.exposure.include=\*

management.endpoint.shutdown.enabled=true

endpoints.shutdown.enabled=true

Note that we also have to expose any actuator endpoints that we want to use. In the example above, we've exposed all the actuator endpoints which will include the /shutdown endpoint.

To shut down the Spring Boot application, we simply call a POST method like this:

curl -X POST localhost:port/actuator/shutdown

In this call, the port represents the actuator port.

2) Close Application Context directly

We can also call the close() method directly using the application context.

Let's start with an example of creating a context and closing it:

***ConfigurableApplicationContext ctx = new***

***SpringApplicationBuilder(Application.class).web(WebApplicationType.NONE).run();***

***System.out.println("Spring Boot application started");***

***ctx.getBean(TerminateBean.class);***

***ctx.close();***

This destroys all the beans, releases the locks, then closes the bean factory. The important thing here to keep in mind: while closing the application context, the parent context isn't affected due to separate lifecycles.

3) Exit SpringApplication - Spring boot Application registers a shutdown hook as above

4) Kill the App Process -

Finally, we can also shut down a Spring Boot Application from outside the application by using a bash script. Our first step for this option is to have the application context write it's PID into a file:

***SpringApplicationBuilder app = new SpringApplicationBuilder(Application.class)***

***.web(WebApplicationType.NONE);***

***app.build().addListeners(new ApplicationPidFileWriter("./bin/shutdown.pid"));***

***app.run();***

Next, create a shutdown.bat file with the following content:

kill $(cat ./bin/shutdown.pid)

The execution of shutdown.bat extracts the Process ID from the shutdown.pid file and uses the kill command to terminate the Boot application.

While it's up to the developer to choose an appropriate a method; all of these methods should be used by design and on purpose.

.exit() is preferred when we need to pass an error code to another environment, say JVM for further actions. Using Application PID gives more flexibility, as we can also start or restart the application with the use of bash script.

Finally, /shutdown is here to make it possible to terminate the applications externally via HTTP. For all the other cases .close() will work perfectly.

**Describe Dependency injection using Java configuration?**

The programmatic approach to configuring beans and the Spring container is also referred to as ‘Java-based container configuration.

@Configuration and @Bean annotations are central to Java-based configuration.

If you annotate a class with @Configuration annotation, it indicates that the class contains one or more @Bean annotated methods that create and return bean instances. The bean instances returned by @Bean annotated methods are managed by the Spring container.

***@Configuration***

***public class BankAppConfiguration {***

***.....***

***@Bean(name = "fixedDepositService")***

***public FixedDepositService fixedDepositService() {***

***return new FixedDepositServiceImpl();***

***}***

***.....***

***}***

In the above example listing, the fixedDepositService method creates and returns an instance of FixedDepositServiceImpl bean that is registered with the Spring container as a bean named fixedDepositService. If you don’t specify the @Bean’s name attribute, the name of the method is considered as the bean name.

You can also define @Bean methods in bean classes annotated with @Component or JSR 330’s @Named annotation.

***@Service***

***public class TransactionServiceImpl implements TransactionService {***

***@Autowired***

***private TransactionDao transactionDao;***

***@Override***

***public void getTransactions(String customerId) {***

***transactionDao.getTransactions(customerId);***

***}***

***@Bean***

***public TransactionDao transactionDao() {***

***return new TransactionDaoImpl();***

***}***

***}***

In the above example listing, TransactionServiceImpl bean class defines an @Bean annotated transactionDao method that returns an instance of TransactionDaoImpl (an implementation of TransactionDao interface). TransactionServiceImpl uses the @Autowired annotation to autowire the TransactionDaoImpl instance.

An @Configuration annotated class can use @ComponentScan annotation to search and register @Component classes.

The following code shows an @Configuration annotated class that uses @ComponentScan annotation:

***@Configuration***

***@ComponentScan(basePackages = "com.sample")***

***public class ABean {***

***.....***

***}***

The basePackages attribute specifies the package(s) that should be searched for @Component annotated classes. If @Component annotated classes are found, they are registered with the Spring container.

@Configuration annotation is meta-annotated with @Component. This is the reason why @Configuration and @Component classes have so much in common. For instance, you can define @Bean annotated methods in them, both can use autowiring, Spring container creates and registers instances of both @Configuration and @Component classes as beans, and so on.

***Injecting bean dependencies using Java configuration -***

When using Java-based configuration approach, @Bean methods are used to create beans. To satisfy dependencies of a bean created by an @Bean method, you have the following options:

1) Obtain the dependencies by explicitly calling the @Bean methods that create and return the dependencies.

***@Configuration***

***public class BankAppConfiguration {***

***@Bean(name = "accountStatementService")***

***public AccountStatementService accountStatementService() {***

***AccountStatementServiceImpl accountStatementServiceImpl =***

***new AccountStatementServiceImpl();***

***accountStatementServiceImpl.setAccountStatementDao(accountStatementDao());***

***return accountStatementServiceImpl;***

***}***

***@Bean(name = "accountStatementDao")***

***public AccountStatementDao accountStatementDao() {***

***return new AccountStatementDaoImpl();***

***}***

***.....***

***}***

2) Specify bean dependencies as arguments to the @Bean method. The Spring container takes care of calling the @Bean methods corresponding to the dependencies and supplying the dependencies as method arguments.

***@Configuration***

***public class BankAppConfiguration {***

***@Bean(name = "accountStatementService")***

***public AccountStatementService accountStatementService(***

***AccountStatementDao accountStatementDao) {***

***AccountStatementServiceImpl accountStatementServiceImpl =***

***new AccountStatementServiceImpl();***

***accountStatementServiceImpl.setAccountStatementDao(accountStatementDao);***

***return accountStatementServiceImpl;***

***}***

***@Bean(name = "accountStatementDao")***

***public AccountStatementDao accountStatementDao() {***

***return new AccountStatementDaoImpl();***

***}***

***.....***

***}***

3) Autowire dependencies by using @Autowired, @Inject and @Resource annotations in the bean class.

***@Configuration***

***public class BankAppConfiguration {***

***.....***

***@Bean(name = "fixedDepositService")***

***public FixedDepositService fixedDepositService(FixedDepositDao fixedDepositDao) {***

***return new FixedDepositServiceImpl();***

***}***

***@Bean***

***public FixedDepositDao fixedDepositDao() {***

***return new FixedDepositDaoImpl();***

***}***

***.....***

***}***

***public class FixedDepositServiceImpl implements FixedDepositService {***

***@Autowired***

***private FixedDepositDao fixedDepositDao;***

***@Override***

***public void createFixedDeposit(FixedDepositDetails fdd) throws Exception {***

***fixedDepositDao.createFixedDeposit(fdd);***

***}***

***}***

In the above example listing, @Autowired annotation autowires the FixedDepositDao bean created by fixedDepositDao method of BankAppConfiguration class.

**Describe Dependency injection using annotations (@Autowired/@Qualifier)?**

The bean definitions contained in the application context XML file are used as a blueprint by the Spring container to create bean instances. A bean definition specifies information about bean dependencies, initialization and destruction methods of a bean, lazy or eager initialization strategy for the bean instance, bean scope, and so on. When using annotations, we can specify the same information in the bean class itself, thereby saving the effort to explicitly configure a bean in the application context XML file.

As we don’t define annotated bean classes in the application context XML file, we don’t have the option to use <property> or <constructor-arg> element to specify their dependencies. For this reason, annotated bean classes make use of annotations like @Autowired, @Inject, and so on, to specify their dependencies.

Spring’s AutowiredAnnotationBeanPostProcessor performs autowiring of fields, methods and constructors that are annotated with Spring’s @Autowired or JSR 330’s @Inject annotation.

When using the @Autowired annotation, exception is thrown if a bean matching the required type is not found. @Autowired’s "required" attribute specifies whether the dependency is mandatory or optional. If you set @Autowired’s "required" attribute value to false, dependency is considered optional. This means that if the "required" attribute’s value is set to 'false', exception is not thrown if no bean matching the required type is found in the Spring container. By default, value of "required" attribute is 'true'; dependencies must be satisfied by the Spring container.

You can express the non-required nature of a particular dependency through Java 8’s java.util.Optional.

@Autowired(required = false)

public void setMovieFinder(MovieFinder movieFinder) { this.movieFinder = movieFinder; }

OR

@Autowired public void setMovieFinder(Optional movieFinder) { }

@Autowired - autowiring dependencies by type --

@Autowired annotation can be used at constructor-level, method-level and field-level.

***1) Field-level -***

***@Service(value="accountStatementService")***

***public class AccountStatementServiceImpl implements AccountStatementService {***

***@Autowired***

***private AccountStatementDao accountStatementDao;***

***@Override***

***public AccountStatement getAccountStatement(Date from, Date to) {***

***return accountStatementDao.getAccountStatement(from, to);***

***}***

***}***

In the above example listing, the accountStatementDao field (of type AccountStatementDao) is annotated with @Autowired annotation. When an instance of AccountStatementServiceImpl is created, Spring’s AutowiredAnnotationBeanPostProcessor (a BeanPostProcessor implementation) is responsible for autowiring accountStatementDao field. The AutowiredAnnotationBeanPostProcessor retrieves reference to an AccountStatementDao type bean from the Spring container and assigns it to the accountStatementDao field.

It is important to note that the field annotated with @Autowired annotation need not be public or have a corresponding public setter method.

***2) Method-level -***

***@Service("customerRegistrationService")***

***public class CustomerRegistrationServiceImpl implements CustomerRegistrationService {***

***private CustomerRegistrationDetails customerRegistrationDetails;***

***.....***

***@Autowired***

***public void obtainCustomerRegistrationDetails(***

***CustomerRegistrationDetails customerRegistrationDetails) {***

***this.customerRegistrationDetails = customerRegistrationDetails;***

***}***

***.....***

***@Override***

***public void setAccountNumber(String accountNumber) {***

***customerRegistrationDetails.setAccountNumber(accountNumber);***

***}***

***.....***

***}***

In the above example listing, obtainCustomerRegistrationDetails method is annotated with @Autowired annotation. If a method is annotated with @Autowired annotation, the arguments of the method are autowired. As obtainCustomerRegistrationDetails method is annotated with @Autowired annotation, its CustomerRegistrationDetails argument is autowired by type.

It is important to note that an @Autowired annotated method need not be public.

A method annotated with @Autowired annotation is automatically invoked after the bean instance is created and the fields annotated with @Autowired annotation are injected with matching bean instances.

***3) Constructor-level --***

***@Service(value="customerRequestService")***

***public class CustomerRequestServiceImpl implements CustomerRequestService {***

***private CustomerRequestDetails customerRequestDetails;***

***private CustomerRequestDao customerRequestDao;***

***@Autowired***

***public CustomerRequestServiceImpl(CustomerRequestDetails customerRequestDetails,***

***CustomerRequestDao customerRequestDao) {***

***this.customerRequestDetails = customerRequestDetails;***

***this.customerRequestDao = customerRequestDao;***

***}***

***.....***

***}***

In the above example listing, the CustomerRequestServiceImpl’s constructor is annotated with @Autowired annotation. If a constructor is annotated with @Autowired annotation, the arguments of the constructor are autowired. As CustomerRequestServiceImpl’s constructor is annotated with @Autowired annotation, its CustomerRequestDetails and CustomerRequestDao arguments are autowired by type. It is important to note that an @Autowired annotated constructor need not be public.

Since Spring 4.3, if the bean class defines only one constructor, you don’t need to annotate the constructor with @Autowired annotation; the Spring container performs autowiring of the constructor arguments by default.

If a bean class defines an @Autowired annotated constructor with required attribute’s value set to true, it can’t have another @Autowired annotated constructor. In such a case it results in an exception thrown by Spring.

***@Service(value="customerRequestService")***

***public class CustomerRequestServiceImpl implements CustomerRequestService {***

***.....***

***@Autowired(required=false)***

***public CustomerRequestServiceImpl(CustomerRequestDetails customerRequestDetails) {***

***.....***

***}***

***@Autowired***

***public CustomerRequestServiceImpl(CustomerRequestDetails customerRequestDetails,***

***CustomerRequestDao customerRequestDao) { ..... }***

***}***

A bean class can define multiple @Autowired annotated constructors with required attribute’s value set to false. In such a case, one of the constructors will be invoked by Spring to create an instance of the bean class. The following example listing shows a bean class that defines two constructors annotated with @Autowired (required = false), and a default constructor:

***@Service(value="customerRequestService")***

***public class CustomerRequestServiceImpl implements CustomerRequestService {***

***public CustomerRequestServiceImpl() { ..... }***

***@Autowired(required=false)***

***public CustomerRequestServiceImpl(CustomerRequestDetails customerRequestDetails) {***

***.....***

***}***

***@Autowired(required=false)***

***public CustomerRequestServiceImpl(CustomerRequestDetails customerRequestDetails,***

***CustomerRequestDao customerRequestDao) {***

***.....***

***}***

***}***

In the above example listing, both the @Autowired annotated constructors are candidates for autowiring by Spring to create an instance of the CustomerRequestServiceImpl class. The constructor with the largest number of satisfied dependencies is chosen. In the case of CustomerRequestServiceImpl class, if beans of types CustomerRequestDetails and CustomerRequestDao are registered with the Spring container, Spring invokes CustomerRequestServiceImpl(CustomerRequestDetails, CustomerRequestDao) constructor. If a bean of type CustomerRequestDetails is registered with container but no bean of type CustomerRequestDao is registered, CustomerRequestServiceImpl(CustomerRequestDetails) constructor is invoked. In case none of the dependencies are found, the default constructor of CustomerRequestServiceImpl class is invoked.

***@Qualifier – autowiring dependencies by name -***

You can use Spring’s @Qualifier annotation along with @Autowired annotation to autowire dependencies by name. The @Qualifier annotation can be used at field-level, method-parameter-level and constructor-argument-level for autowiring dependencies by name.

***@Service(value="fixedDepositService")***

***.....***

***public class FixedDepositServiceImpl implements FixedDepositService {***

***@Autowired***

***@Qualifier(value="myFixedDepositDao")***

***private FixedDepositDao myFixedDepositDao;***

***.....***

***}***

Spring first finds autowiring candidates ‘by type’ for the fields, constructor arguments and method arguments that are annotated with @Autowired annotation. Then, Spring uses the bean name specified by @Qualifier annotation to locate a unique bean from the list of autowiring candidates. In above code, Spring first finds beans of type FixedDepositDao for myFixedDepositDao field, and then locates the bean named myFixedDepositDao from the list of autowiring candidates. If a bean named myFixedDepositDao is found, Spring assigns it to the myFixedDepositDao field. Basically, @Qualifier annotation is used to avoid conflicts when there are multiple autowiring candidates which satisfy autowiring by type (like multiple implementations of same interface).

***@Qualifier usage at method-parameter-level and constructor-argument-level***

***public class Sample {***

***@Autowired***

***public Sample(@Qualifier("aBean") ABean bean) { .... }***

***@Autowired***

***public void doSomething(@Qualifier("bBean") BBean bean, CBean cBean) { ..... }***

***}***

JSR 330 (Dependency Injection for Java) standardizes dependency injection annotations for the Java platform. JSR 330 defines @Inject and @Named annotations that are similar to Spring’s @Autowired and @Qualifier annotations, respectively. Spring provides support for @Inject and @Named annotations.

**Describe Component scanning, Stereotypes?**

***Component scanning —*** Spring automatically discovers beans to be created in the application context. Bean classes annotated with @Component, @Controller, @Service or @Repository annotations are automatically registered with the Spring container.

Filtering configuration can be added to the @ComponentScan annotation as to include or exclude certain classes.

* basePackages
* basePackageClasses
* includeFilters
* includeFilters

***@ComponentScan(***

***basePackages = "...",***

***basePackageClasses = XxxService.class,***

***excludeFilters = @ComponentScan.Filter(type = FilterType.REGEX, pattern = ".\*Repository"),***

***includeFilters = @ComponentScan.Filter(type = FilterType.ANNOTATION, classes = MyService.class))***

***public class FooApplication(){}***

Stereotype annotations (something conforming to a fixed or general pattern) are annotations that are applied classes that contains information of which role Spring beans implemented by the class fulfills.

* @Component
* @Controller
* @RestController
* @Service
* @Repository

Spring’s @Component annotation is a type-level annotation, which indicates that a class represents a Spring bean (also referred to as a Spring component). It is recommended that you use the specialized forms of @Component annotation to annotate controllers, services and data access objects (DAOs) of your application. For instance, annotate controllers with @Controller, services with @Service, and DAOs with @Repository annotation.

You should note that @Service, @Controller and @Repository annotations are meta-annotated with @Component annotation, that is, they are themselves annotated with @Component annotation. For example, the following definition of @Service annotation shows that it is meta-annotated with @Component annotation:

***@Target({ElementType.TYPE})***

***@Retention(RetentionPolicy.RUNTIME)***

***@Documented***

***@Component***

***public @interface Service {***

***String value() default "";***

***}***

@Configuration classes are meta-annotated with @Component, so they are candidates for component-scanning. But, it’s not a sterotype.

**Describe Scopes for Spring beans? What is the default scope?**

1) Singleton scope: per container, default bean scope, stateless

2) Prototype: each time a bean is request, stateful

3) Request: Scopes a single bean definition to the lifecycle of an HTTP request, web-aware contexts only

4) Session: Scopes a single bean definition to the lifecycle of an HTTP Session, web-aware contexts only

5) Application: Scopes a single bean definition to the lifecycle of a ServletContext, web-aware contexts only

6) Websocket: Scopes a single bean definition to the lifecycle of a WebSocket, web-aware contexts only

**Are beans lazily or eagerly instantiated by default? How do you alter this behavior?**

A singleton-scoped bean is pre-instantiated by default, which means an instance of a singleton-scoped bean is created when you create an instance of the Spring container. That means, Singleton beans are eagerly instantiated by default.

A prototype-scoped bean is always lazily-initialized.

This behavior can be altered by using @Lazy annotation.

@Lazy: postpone the creation of a bean until it is first accessed. By default is true. So @Lazy(value=true) equals to @Lazy. To alter behavior, set @Lazy to true or false

1) @Lazy is used on injection points (as @Autowired)

2) @Lazy is used with @Component

3) @Lazy is used with @Bean along with @Configuration

***@Repository***

***public class JdbcPetRepo extends JdbcAbstractRepo<Pet> implements PetRepo {***

***// 1) @Lazy is used on injection points (as @Autowired)***

***@Lazy***

***@Autowired(required=false)***

***public void setDataSource(DataSource dataSource) {***

***this.dataSource = dataSource;***

***}***

***}***

***//2)@Lazy is used with @Component***

***@Service("accountService")***

***@Lazy(true)***

***public class AccountServiceImpl implements AccountService { }***

***@Configuration***

***public class Ch2BeanConfiguration {***

***// 3)@Lazy is used with @Bean along with @Configuration***

***@Bean***

***@Lazy(true)***

***public AccountService accountService() {***

***AccountServiceImpl bean = new AccountServiceImpl();***

***return bean;***

***}***

***}***

***Advantage of @Lazy:*** It speeds container bootstrap time and has a smaller memory footprint. useful when the dependency is a huge object.

***Disadvantage of @Lazy:*** Error remains unnoticed until using of it. In most application scenarios, it is beneficial to pre-instantiate singleton beans to discover configuration issues at the time of creation of the Spring container.

**What is a property source? How would you use @PropertySource?**

A property source is basically a file containing key-value pairs of properties. These properties can be DataSource configuration properties or general application specific properties.

***@PropertySource -***

Annotation providing a convenient and declarative mechanism for adding a PropertySource to Spring's Environment. To be used in conjunction with @Configuration classes.

Given a file app.properties containing the key/value pair testbean.name=myTestBean, the following @Configuration class uses @PropertySource to contribute app.properties to the Environment's set of PropertySources.

***@Configuration***

***@PropertySource("classpath:/com/myco/app.properties")***

***public class AppConfig {***

***@Autowired***

***Environment env;***

***@Bean***

***public TestBean testBean() {***

***TestBean testBean = new TestBean();***

***testBean.setName(env.getProperty("testbean.name"));***

***return testBean;***

***}***

***}***

Notice that the Environment object is @Autowired into the configuration class and then used when populating the TestBean object. Given the configuration above, a call to testBean.getName() will return "myTestBean".

**Resolving ${...} placeholders in <bean> and @Value annotations -**

In order to resolve ${...} placeholders in <bean> definitions or @Value annotations using properties from a PropertySource, you must ensure that an appropriate embedded value resolver is registered in the BeanFactory used by the ApplicationContext. This happens automatically when using <context:property-placeholder> in XML. When using @Configuration classes this can be achieved by explicitly registering a PropertySourcesPlaceholderConfigurer via a static @Bean method.

***@Configuration***

***@PropertySource("classpath:/META-INF/devDB.properties")***

***public class DevDBConfiguration {***

***private static Logger logger = LogManager.getLogger(DevDBConfiguration.class);***

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

***private String driverClass;***

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

***private String url;***

***.....***

***@Bean***

***public DataSource dataSource() {***

***return new DataSource(driverClass, url, username, password);***

***}***

***@Bean***

***public static PropertySourcesPlaceholderConfigurer***

***propertySourcesPlaceholderConfigurer() {***

***return new PropertySourcesPlaceholderConfigurer();***

***}***

***}***

In the above example listing, @PropertySource annotation reads database configuration from the devDB.properties file and adds them to Spring’s Environment object. For instance, the driverClassName, url, username and password properties defined in devDB.properties file are added to Environment object. The propertySourcesPlaceholderConfigurer method configures PropertySourcesPlaceholderConfigurer (a BeanFactoryPostProcessor) that resolves ${.....} placeholders specified by @Value annotations against the Environment object.

***Resolving ${...} placeholders within @PropertySource resource locations -***

Any ${...} placeholders present in a @PropertySource resource location will be resolved against the set of property sources already registered against the environment.

***@Configuration***

***@PropertySource("classpath:/com/${my.placeholder:default/path}/app.properties")***

***public class AppConfig {***

***@Autowired***

***Environment env;***

***@Bean***

***public TestBean testBean() {***

***TestBean testBean = new TestBean();***

***testBean.setName(env.getProperty("testbean.name"));***

***return testBean;***

***}***

***}***

Assuming that "my.placeholder" is present in one of the property sources already registered, e.g. system properties or environment variables, the placeholder will be resolved to the corresponding value. If not, then "default/path" will be used as a default. Expressing a default value (delimited by colon ":") is optional. If no default is specified and a property cannot be resolved, an IllegalArgumentException will be thrown.

***@TestPropertySource -***

Class-level annotation annotation for integration test. Have higher precedence than -

* Property sources loaded from operating system’s environment
* Java system properties
* Property sources added by application via @PropertySource

NB: inline properties have higher precedence than properties loaded from resource locations.

***@ContextConfiguration***

***@TestPropertySource(properties = { "timezone = GMT", "port: 4242" }) // inline properties***

***public class MyIntegrationTests { }***

**What does component-scanning do?**

Component, or classpath, scanning is the process using which the Spring container searches the classpath for classes annotated with stereotype annotations (@Component, @Service, @Repository, @Controller) and registers bean definitions in the Spring container for such classes.

An @Configuration annotated class can use @ComponentScan annotation to search and register @Component classes.

The following code shows an @Configuration annotated class that uses @ComponentScan annotation:

***@Configuration***

***@ComponentScan(basePackages = "com.sample")***

***public class ABean {***

***.....***

***}***

The basePackages attribute specifies the package(s) that should be searched for @Component annotated classes. If @Component annotated classes are found, they are registered with the Spring container.

@Configuration annotation is meta-annotated with @Component. This is the reason why @Configuration and @Component classes have so much in common. For instance, you can define @Bean annotated methods in them, both can use autowiring, Spring container creates and registers instances of both @Configuration and @Component classes as beans, and so on.

**• What is an initialization method and how is it declared on a Spring bean?**

**• What is a destroy method, how is it declared and when is it called?**

**• Consider how you enable JSR-250 annotations like @PostConstruct and @PreDestroy? When/how will they get called?**

**• How else can you define an initialization or destruction method for a Spring bean?**

***A) Custom initialization and destruction Methods using init-method and destroy-method attributes -***

1) The Spring container is responsible for creating a bean instance and injecting its dependencies.

2) After creating a bean instance by invoking the constructor of the bean class, the Spring container sets bean properties by invoking bean’s setter methods.

3) If you want to execute custom initialization logic (like opening a file, creating a database connection, and so on) after the bean properties are set but before the bean is completely initialized by the Spring container, specify the name of the initialization method as the value of init-method attribute of the <bean> element.

4) If you want to execute custom cleanup logic before the Spring container containing the bean instance is destroyed, you can specify the name of the cleanup method as the value of destroy-method attribute of <bean> element.

5) It is important to note that the initialization and destruction methods specified by init-method and destroy-method attributes of <bean> element must not accept any arguments, but can be defined to throw exceptions.

***XML Configuration for init-method and destroy-method attributes -***

<beans .....>

<bean id="fixedDepositService"

class="sample.spring.chapter05.bankapp.service.FixedDepositServiceImpl">

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

</bean>

<bean id="myFixedDepositDao"

class="sample.spring.chapter05.bankapp.dao.FixedDepositDaoImpl"

init-method="initializeDbConnection" destroy-method="releaseDbConnection" />

</beans>

The above example listing shows that the <bean> element corresponding to the FixedDepositDaoImpl class specifies initializeDbConnection and releaseDbConnection as the values of init-method and destroy-method attributes, respectively. The FixedDepositDaoImpl instance is a dependency of FixedDepositServiceImpl.

6) The Spring container injects a dependency (the FixedDepositDaoImpl instance) into the dependent bean (the FixedDepositServiceImpl instance) after the initialization method of the dependency is invoked by the Spring container.

7) The web version of ApplicationContext implementation is represented by Spring’s WebApplicationContext object. WebApplicationContext implementation has the necessary logic to invoke the cleanup method (specified by the destroy-method attribute) of singleton bean instances before the web application is shutdown.

8) However, for standalone application, we need to make Spring gracefully destroy singleton bean instances by calling the cleanup method specified by the <bean> element’s destroy-method attribute.

9) Spring’s ConfigurableApplicationContext (a sub-interface of ApplicationContext) defines a registerShutdownHook method that registers a shutdown hook with the JVM. The shutdown hook is responsible for closing the ApplicationContext when the JVM is shutdown. When the standalone application's main method exists, the shutdown hook destroys all cached singleton bean instances and closes the ApplicationContext instance.

10) In the case of prototype-scoped beans, destroy-method attribute is ignored by the Spring container. The destroymethod attribute is ignored because the Spring container expects that the object that fetches the prototype bean instance from the ApplicationContext is responsible for explicitly calling the cleanup method on the prototype bean instance.

***B) JSR 250’s @PostConstruct and @PreDestroy annotations -***

1) JSR 250 (Common Annotations for the Java Platform) defines standard annotations that are used across different Java technologies. JSR 250’s @PostConstruct and @PreDestroy annotations identify initialization and destruction methods of an object. A bean class in Spring can set a method as an initialization method by annotating it with @PostConstruct, and set a method as a destruction method by annotating it with @PreDestroy annotation.

2)To use @PostConstruct and @PreDestroy annotations in your Spring application, you need to configure Spring’s CommonAnnotationBeanPostProcessor class in the application context XML file. CommonAnnotationBeanPostProcessor implements Spring’s BeanPostProcessor interface and is responsible for processing JSR 250 annotations.

***C) InitializingBean and DisposableBean lifecycle interfaces -***

1) A bean that implements lifecycle interfaces, like InitializingBean and DisposableBean, receives callbacks from the Spring container. These callbacks give an opportunity to the bean instance to perform some action like bean initialization or bean cleanup respectively.

2) InitializingBean interface defines an afterPropertiesSet method that is invoked by the Spring container after the bean properties are set. Beans perform initialization work in the afterPropertiesSet method, like obtaining connection to a database, opening a flat file for reading, and so on. DisposableBean interface defines a destroy method that is invoked by the Spring container when the bean instance is destroyed.

3) Beans should avoid implementing InitializingBean and DisposableBean interfaces because it couples application code with Spring.

***Initialization sequence of calls -***

1) The @PostConstruct annotation method is called, which is registered by CommonAnnotationBeanPostProcessor.

2) The InitializingBean’s afterPropertiesSet() method is invoked by a BeanFactory after it has set all the bean properties supplied and has satisfied BeanFactoryAware and ApplicationContextAware.

3) The init-method attribute is executed last because this is the actual initialization method of the bean.

***Destruction sequence of calls -***

1) @PreDestroy

2) destroy() as defined by the DisposableBean callback interface

3) A custom configured destroy() method

**• What is a BeanFactoryPostProcessor and what is it used for? When is it invoked?**

**• Why would you define a static @Bean method?**

**• What is a PropertySourcesPlaceholderConfigurer used for?**

Spring’s BeanFactoryPostProcessor interface is implemented by classes that want to make modifications to bean definitions. A BeanFactoryPostProcessor is executed after bean definitions are loaded by the Spring container, but before any bean instance is created. A BeanFactoryPostProcessor is created before any other bean defined in the application context XML file, giving the BeanFactoryPostProcessor an opportunity to make modifications to bean definitions of other beans. You configure a BeanFactoryPostProcessor implementation in the application context XML file like any other Spring bean.

BeanFactoryPostProcessor interface has postProcessBeanFactory method

a) void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException - Modify the application context's internal bean factory after its standard initialization. This method is is executed after bean definitions are loaded by the Spring container, but before any bean instance is created.

It is important to note that a ConfigurableListableBeanFactory provides access to the Spring container just like the ApplicationContext object. ConfigurableListableBeanFactory additionally allows you to configure the Spring container, iterate over beans, and modify bean definitions.

***Static Bean method -***

Static @Bean methods are called without creating their containing configuration class as an instance, since such beans will get initialized early in the container lifecycle and should avoid triggering other parts of the configuration at that point.

Calls to static @Bean methods never get intercepted by the container, because CGLIB subclassing can override only non-static methods.

This makes particular sense when defining postprocessor beans of type BeanFactoryPostProcessor or BeanPostProcessor. For e.g. PropertySourcesPlaceholderConfigurer can be defined as a a static @Bean method.

***PropertySourcesPlaceholderConfigurer -***

PropertySourcesPlaceholderConfigurer (a BeanFactoryPostProcessor) lets you specify the actual string value of bean properties and constructor arguments in a properties file. In the bean definition, you only specify property placeholders (of the form ${<property\_name\_in\_properties\_file>}) as the value of <property> or <constructor-arg> element’s value attribute. When bean definitions are loaded by the Spring container, the PropertySourcesPlaceholderConfigurer pulls the actual values from the properties file and replaces the property placeholders in the bean definitions with actual values.

In order to resolve ${...} placeholders in <bean> definitions or @Value annotations using properties from a PropertySource, you must ensure that an appropriate embedded value resolver is registered in the BeanFactory used by the ApplicationContext. This happens automatically when using <context:property-placeholder> in XML. When using @Configuration classes this can be achieved by explicitly registering a PropertySourcesPlaceholderConfigurer via a static @Bean method.

***applicationContext.xml - Bean definitions that use property placeholders***

<bean id="datasource" class="sample.spring.chapter05.domain.DataSource">

<property name="url" value="${database.url}" />

<property name="username" value="${database.username}" />

<property name="password" value="${database.password}" />

<property name="driverClass" value="${database.driverClass}" />

</bean>

<bean id="webServiceConfiguration" class="sample.spring.chapter05.domain.WebServiceConfiguration">

<property name="webServiceUrl" value="${webservice.url}" />

</bean>

applicationContext.xml - PropertySourcesPlaceholderConfigurer bean definition

<bean class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">

<property name="locations">

<list>

<value>classpath:database.properties</value>

<value>classpath:webservice.properties</value>

</list>

</property>

<property name="ignoreUnresolvablePlaceholders" value="false" />

</bean>

The ignoreUnresolvablePlaceholders property specifies whether PropertySourcesPlaceholderConfigurer silently ignores or throws an exception in case a property placeholder value is not found in any of the properties files specified by the locations property.

**What is a BeanPostProcessor and how is it different to a BeanFactoryPostProcessor?**

1) A BeanPostProcessor is used to interact with newly created bean instances before and/or after their initialization method is invoked by the Spring container. You can also use a BeanPostProcessor to execute custom logic before and/or after a bean’s initialization method is invoked by the Spring container.

2) A bean that implements Spring’s BeanPostProcessor interface is a special bean type; the Spring container automatically detects and executes a BeanPostProcessor bean.

3) BeanPostProcessor interface defines the following methods:

a) Object postProcessBeforeInitialization(Object bean, String beanName) – this method is invoked before the initialization method of a bean instance is invoked.

b) Object postProcessAfterInitialization(Object bean, String beanName) – this method is invoked after the initialization method of a bean instance is invoked.

BeanPostProcessor’s methods accept the newly created bean instance and its name as arguments, and they may return the same or modified bean instance or an object that wraps the original bean instance.

4) You configure a BeanPostProcessor implementation in the application context XML file like any other Spring bean. Spring container automatically detects beans that implement the BeanPostProcessor interface, and creates their instance before creating instance of any other bean defined in the application context XML file. Once the BeanPostProcessor beans are created, the Spring container invokes BeanPostProcessor’s postProcessBeforeInitialization and postProcessAfterInitialization methods for each bean instance created by the Spring container.

5) It’s only after the invocation of postProcessAfterInitialization method that a bean instance is considered completely initialized by the Spring container. For this reason, if a ABean bean is dependent on BBean, container will inject BBean instance into ABean only after MyBeanPostProcessor’s postProcessAfterInitialization method has been invoked for BBean instance.

***BeanPostProcessor example – Validating bean instances -***

Let’s see how we can use a BeanPostProcessor implementation to verify that a bean instance is configured correctly before it is injected into dependent beans or accessed by other objects in the application.

InstanceValidator interface that must be implemented by beans whose configurations we want to validate using a BeanPostProcessor implementation

***public interface InstanceValidator {***

***void validateInstance();***

***}***

InstanceValidator interface defines a validateInstance method that verifies whether the bean instance was correctly initialized or not.

The FixedDepositDaoImpl class that implements the InstanceValidator interface and validateInstance method.

***public class FixedDepositDaoImpl implements FixedDepositDao, InstanceValidator {***

***private static Logger logger = LogManager.getLogger(FixedDepositDaoImpl.class);***

***private DatabaseConnection connection;***

***public FixedDepositDaoImpl() {***

***logger.info("FixedDepositDaoImpl's constructor invoked");***

***}***

***public void initializeDbConnection() {***

***logger.info("FixedDepositDaoImpl’s initializeDbConnection method invoked");***

***connection = DatabaseConnection.getInstance();***

***}***

***@Override***

***public void validateInstance() {***

***logger.info("Validating FixedDepositDaoImpl instance");***

***if(connection == null) {***

***logger.error("Failed to obtain DatabaseConnection instance");***

***}***

***}***

***}***

If connection attribute is null, the validateInstance method logs an error message indicating that the FixedDepositDaoImpl instance is not correctly initialized. In a real world application development scenario, if a bean instance is not configured correctly, the validateInstance method may take some corrective action or throw a runtime exception to stop the application from starting up.

The InstanceValidationBeanPostProcessor class that implements Spring’s BeanPostProcessor interface, and is responsible for invoking validateInstance method of newly created beans.

***public class InstanceValidationBeanPostProcessor implements BeanPostProcessor, Ordered {***

***private static Logger logger =***

***LogManager.getLogger(InstanceValidationBeanPostProcessor.class);***

***private int order;***

***public InstanceValidationBeanPostProcessor() {***

***logger.info("Created InstanceValidationBeanPostProcessor instance");***

***}***

***@Override***

***public Object postProcessBeforeInitialization(Object bean, String beanName)***

***throws BeansException {***

***logger.info("postProcessBeforeInitialization method invoked");***

***return bean;***

***}***

***@Override***

***public Object postProcessAfterInitialization(Object bean, String beanName)***

***throws BeansException {***

***logger.info("postProcessAfterInitialization method invoked");***

***if (bean instanceof InstanceValidator) {***

***((InstanceValidator) bean).validateInstance();***

***}***

***return bean;***

***}***

***public void setOrder(int order) {***

***this.order = order;***

***}***

***@Override***

***public int getOrder() {***

***return order;***

***}***

***}***

The above example listing shows that the InstanceValidationBeanPostProcessor class implements Spring’s BeanPostProcessor and Ordered interfaces. The postProcessBeforeInitialization method simply returns the bean instance passed to the method. In the postProcessAfterInitialization method, if the bean instance is found to be of type InstanceValidator, the bean instance’s validateInstance method is invoked. This means that if a bean implements InstanceValidator interface, InstanceValidationBeanPostProcessor calls the validateInstance method of the bean instance after the initialization method of the bean instance is invoked by the Spring container.

The Ordered interface defines a getOrder method which returns an integer value. The integer value returned by the getOrder method determines the priority of a BeanPostProcessor implementation with respect to other BeanPostProcessor implementations configured in the application context XML file. A BeanPostProcessor with higher order value is considered at a lower priority, and is executed after the BeanPostProcessor implementations with lower order values are executed. As we want the integer value returned by the getOrder method to be configured as a bean property, a setOrder method and an order instance variable are defined in the InstanceValidationBeanPostProcessor class.

***Bean definition for InstanceValidationBeanPostProcessor class-***

<bean class="...bankapp.postprocessor.InstanceValidationBeanPostProcessor">

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

</bean>

***Similarities between BeanPostProcessor and BeanFactoryPostProcessor -***

1) A bean that implements Spring’s BeanPostProcessor interface is a special bean type; the Spring container automatically detects and executes a BeanPostProcessor bean. A bean that implements Spring’s BeanFactoryPostProcessor interface is a special bean type; the Spring container automatically detects and executes a BeanFactoryPostProcessor bean.

2) You can configure multiple BeanFactoryPostProcessors in the application context XML file. To control the order in which BeanFactoryPostProcessors are executed by the Spring container, implement Spring’s Ordered interface.

3) Even if you specify that a BeanFactoryPostProcessor implementation is lazily initialized by the Spring container,

BeanFactoryPostProcessors are created when the Spring container instance is created.

***Differences between BeanPostProcessor and BeanFactoryPostProcessor -***

1) A BeanPostProcessor is used to modify or interact with newly created bean instances before and/or after their initialization method is invoked by the Spring container. whereas, Spring’s BeanFactoryPostProcessor interface is implemented by classes that want to make modifications to bean definitions. A BeanFactoryPostProcessor is executed after bean definitions are loaded by the Spring container, but before any bean instance is created.

2) The beans that implement the BeanFactoryPostProcessor interface are created before the beans that implement the BeanPostProcessor interface. A BeanFactoryPostProcessor is created before any other bean defined in the application context XML file, giving the BeanFactoryPostProcessor an opportunity to make modifications to bean definitions of other beans.

3) BeanPostProcessor interface defines the following methods:

a) Object postProcessBeforeInitialization(Object bean, String beanName) – this method is invoked before the initialization method of a bean instance is invoked.

b) Object postProcessAfterInitialization(Object bean, String beanName) – this method is invoked after the initialization method of a bean instance is invoked.

***BeanFactoryPostProcessor interface has postProcessBeanFactory method -***

a) void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException - Modify the application context's internal bean factory after its standard initialization. This method is is executed after bean definitions are loaded by the Spring container, but before any bean instance is created.

4) It is important to note that the beans that implement the BeanFactoryPostProcessor interface are processed before beans that implement the BeanPostProcessor interface. For this reason, you can’t use a BeanPostProcessor to make modifications to a BeanFactoryPostProcessor instance.

5) ***Examples of BeanPostProcessor -*** Spring’s AutowiredAnnotationBeanPostProcessor performs autowiring of fields, methods and constructors that are annotated with Spring’s @Autowired or JSR 330’s @Inject annotation.

@Required annotation, which is backed by the built-in Spring post-processor RequiredAnnotationBeanPostProcessor which checks whether all the bean properties with the @Required annotation have been set.

***Example of postProcessBeanFactory*** - Spring’s PropertySourcesPlaceholderConfigurer lets you specify the actual string value of bean properties and constructor arguments in a properties file.

**What is the behavior of the annotation @Autowired with regards to field injection, constructor injection and method injection?**

@Autowired tries to find a matching bean by type and inject it at the place on annotation - that may be a constructor, a method (not only setter, but usually setter) and field. i.e. @Autowired is basically used for autowiring dependencies by type.

1) Container examines the type of field

2) Container searches for a bean that matches the type

3) If multiple matching, @Primary bean is injected

4) If multiple matching, @Qualifier bean might be used

5) If multiple matching, try to match bean name and field name

6) Exception throws if no unique matching

7) @Autowired cannot be used to autowire primitive values, or Strings. @Value specializes in this exactly.

***Example -***

***public class MovieRecommender {***

***private final CustomerPreferenceDao customerPreferenceDao;***

***@Autowired***

***@Qualifier("main")***

***private MovieCatalog movieCatalog; //@Autowired to fields***

***@Autowired // Constructor level***

***public MovieRecommender(@Qualifier("second")CustomerPreferenceDao customerPreferenceDao) {***

***this.customerPreferenceDao = customerPreferenceDao;***

***}***

***@Autowired // Method- level***

***public void obtainCustomerRegistrationDetails(***

***CustomerRegistrationDetails customerRegistrationDetails) {***

***this.customerRegistrationDetails = customerRegistrationDetails;***

***}***

***}***

Note: For details Refer to question - Describe Dependency injection using annotations (@Autowired/@Qualifier)

**How does the @Qualifier annotation complement the use of @Autowired?**

You can use Spring’s @Qualifier annotation along with @Autowired annotation to autowire dependencies by name. The @Qualifier annotation can be used at field-level, method-parameter-level and constructor-argument-level for autowiring dependencies by name.

@Autowired + @Qualifier = @Resource(name="beanName")

\*\*@Qualifier used at 3 locations: \*\*

Inject Points. The most basic use of the @Qualifier annotation is to specify the name of the Spring bean to be selected the bean to be dependency-injected.

On a field,method or constructor

***@Autowired***

***@Qualifier("iceCream")***

***public void setDessert(Dessert dessert) {***

***this.dessert = dessert;***

***}***

On a method/constructor argument (before parameter type)

***public class MovieRecommender {***

***private MovieCatalog movieCatalog;***

***@Autowired***

***public void prepare(@Qualifier("main") MovieCatalog movieCatalog) {***

***this.movieCatalog = movieCatalog;***

***}***

***}***

Bean Definitions. This will assign a qualifier to the bean and the same qualifier can later be used at an injection point to inject the bean in question.

***Declare it -***

***@Component***

***@Qualifier("cold")***

***public class IceCream implements Dessert {***

***.......***

***}***

***Use it -***

***@Autowired***

***@Qualifier("cold")***

***public void setDessert(Dessert dessert) {***

***this.dessert = dessert;***

***}***

***Annotation Definition -***

@Qualifier can also be used alongside the @Bean annotation when explicitly defining beans with Java configuration

***@Bean***

***@Qualifier("cold")***

***public Dessert iceCream() {***

***return new IceCream();***

***}***

Note: For details Refer to question - Describe Dependency injection using annotations (@Autowired/@Qualifier)

**What do you have to do, if you would like to inject something into a private field? How does this impact testing?**

Using @Autowired or @Inject to inject something into a private field is not a good choice. "Why do I need to start a DI container to run some basic unit tests?". The answer was "because I have no way to set a value to the private field". All I wanted was to inject a mock component to the bean under tests. I couldn’t, because I had no way to set the mock component instance to the private field. I was coupled to the DI container because only the container could change that field. I could have used Reflection, but, my test code would fail after each refactoring session, because in reflection I would be referring to the field name as a string and that name could change while refactoring. On the other hand, using reflection is a hack. I was looking for a more natural and elegant solution.

When you want to be able to define the value of a private field, you have two options: The first one is to create a constructor that receives the value to be set and the second one is to create a setter for such private field.

The difference between these two is that creating a constructor allows you to preserve the immutability, while creating the setter doesn’t, meaning anything gets the chance to set a different value.

Anyway, I wouldn’t need to start a DI container within my tests, because I would be able to set the value myself. I would be decoupled from the DI container. This is the big win of Constructor and Setter based dependency injection.

By adopting Constructor and Setter based dependency injection, it actually felt more natural to set the mocks myself.

@Value annotation can also be used to inject something into a private field. For testing purpose, @TestPropertySource allows using either a test-specific property file or customizing individual property values. Spring framework also provides ReflectionTestUtils.

**• What is a proxy object and what are the two different types of proxies Spring can create?**

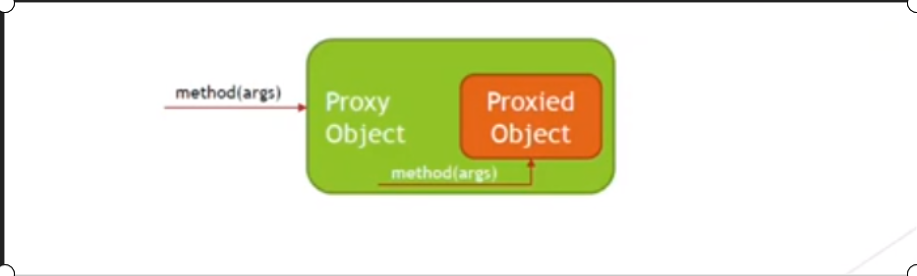
**• What are the limitations of these proxies (per type)?**

**• What is the power of a proxy object and where are the disadvantages?**

A proxy object is an object that adds additional logic (logging,security,auditing etc.) on top of object that is being proxied without having to modify code of proxied object.

Proxy object has the same public methods as proxied object and it should be as much as possible indistinguishable from proxied object.

When any method is invoked on the proxy object, additional code from proxy object (usually before and after sections) are invoked and then code from proxied object is invoked.



Using Proxy pattern this way improves code readability and maintainability through clear separation of concerns.

There are multiple ways to implement proxy pattern in Spring. We can do it manually but it will need proxy object to implement original interface implemented by proxied object and duplicate additional logic in each of the methods of proxied object.

To avoid code duplication problem of manual implementation of proxy pattern, Spring supports two kind of proxies -

***JDK Dynamic Proxy*** - Used by default if target object (proxied object) implements an interface.

***CGLIB Proxy*** - Used when target object (proxied object) doesn’t implement any interface. CGLIB Proxy dynamically generates the bytecode for a new class on runtime for each proxy, reusing already generated classes wherever possible.

***Limitations of JDK Dynamic Proxy -***

1) Requires target object (proxied object) to implement at least one interface.

2) Only interface methods will be proxied.

3) Only public methods found in the implemented interface(s) will be available in the proxy object.

4) Proxy objects must be referenced using an interface type and cannot be referenced using a type of a superclass of the proxied object type.

5) No support for self-invocation.

***Limitations of CGLIB Proxy -***

1) Under the hood it creates extension of the original class (Proxied object). It uses inheritance internally.

2) Does not work for final classes

3) Does not work for final methods

4) No support for self-invocation.

5) Cannot proxy private methods. (public, protected and package-visible are ok)

6) Proxies can only work from the outside

7) Proxied objects must be instantiated by the Spring container(not new keyword)

8) Proxies are not serializable

***Power (Advantages) of Proxy Object -***

1) Ability to change behavior of existing beans without changing original code.

2) Separation of concerns (logging,transactions,security, etc.)

***Disadvantages) of Proxy Object -***

1) May create code which is hard to debug

2) Needs to use unchecked exception for exceptions not declared in original method.

3) May cause performance issues if before/after section in proxy code is using I/O.

4) May cause unexpected Equals operator (==) results since Proxy object and proxied object are two different objects.

5) Configuration classes cannot be final. Configuration classes are subclassed by the Spring container using CGLIB and final classes cannot be subclassed.

***Practical use of Proxies in Spring***

1) For singleton scoped bean method invocation, Spring use CGLIB proxy. Use of proxy ensures that bean instance is created only first time and then same instance is refereed by proxy object during subsequent calls instead of creating a new instance every time from proxied bean method.

2) Spring AOP framework creates a CGLIB- or JavaSE-based proxy. If the target object doesn’t implement any interface, Spring AOP creates a CGLIB-based proxy. If the target object implements one or more interfaces, Spring AOP creates a JavaSEbased proxy. If the value of <aspectj-autoproxy> element’s proxy-target-class attribute is set to false, it instructs Spring AOP to create a JavaSE-based proxy if the target object implements one or more interface. If you set proxy-target-class attribute’s value to true, it instructs Spring AOP to create CGLIB-based proxies even if a target object implements one or more interfaces.

In Java-based configuration approach, the @EnableAspectJAutoProxy annotation serves the same purpose as the <aspectj-autoproxy> element.

**What does the @Bean annotation do?**

The @Bean annotation tells Spring that this method will return an object that should be registered as a bean in the Spring application context. The @Bean annotation together with the method are treated as a bean definition, and the method name becomes the bean id. The body of the method contains logic that ultimately results in the creation of the bean.

**What is the default bean id if you only use @Bean? How can you override this?**

The @Bean annotation together with the method are treated as a bean definition, and the method name becomes the bean id. Method name is default bean id or bean name. Override it by using name or value attributes on @Bean.

***@Bean(name="overrideName")***

***public MyBean myBeanDefaultId() {***

***return new MyBean();***

***}***

**Why are you not allowed to annotate a final class with @Configuration?**

JavaConfig requires CGLIB subclassing of each configuration class at runtime, so that @Configuration classes and their factory methods must not be marked as final or private.

**How do @Configuration annotated classes support singleton beans?**

Each bean method in @configuration class is a singleton scope by default.

***@Bean***

***public Coach swimCoach() {***

***SwimCoach mySwimCoach = new SwimCoach();***

***return mySwimCoach;***

***}***

This method has the @Bean annotation. The annotation will intercept ALL calls to the method "swimCoach()". Since no scope is specified the @Bean annotation uses singleton scope. Behind the scenes, during the @Bean interception, it will check in memory of the Spring container (applicationContext) and see if this given bean has already been created. This interception is done by CGLIB proxy object of each configuration class at runtime.

If this is the first time the bean has been created then it will execute the method as normal. It will also register the bean in the application context. So that is knows that the bean has already been created before. Effectively setting a flag.

The next time this method is called, the @Bean annotation (CGLIB proxy object) will check in memory of the Spring container (applicationContext) and see if this given bean has already been created. Since the bean has already been created then it will immediately return the instance from memory. It will not execute the code inside of the method. Hence this is a singleton bean.

**Why can’t @Bean methods be final either?**

For singleton scoped bean method invocation, Spring use CGLIB proxy. Use of proxy ensures that bean instance is created only first time and then same instance is refereed by proxy object during subsequent calls instead of creating a new instance every time from proxied bean method. CGlib proxying cannot proxy a final method. JavaConfig requires CGLIB subclassing of each configuration class at runtime, so that @Configuration classes and their factory methods must not be marked as final or private.

**How do you configure profiles?**

***1) JVM argument***

One easy way is to specify them as the -Dspring.profiles.active JVM argument value.

***2) Environment API***

Environment API is the most straightforward way to configure profiles. The Environment and PropertySource abstraction features in Spring assist developers in accessing various configuration information from the running platform. Environment interface includes: - all system properties, - environment variables, and - application properties.

***// Environment API***

***AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();***

***ctx.getEnvironment().setActiveProfiles("development");***

***// ctx.getEnvironment().setActiveProfiles("profile1", "profile2"); //works with multi profiles***

***ctx.register(SomeConfig.class, StandaloneDataConfig.class, JndiDataConfig.class);***

***ctx.refresh();***

***// System properties***

***public class BankAppWithProfile {***

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

***System.setProperty("spring.profiles.active", "mybatis, production");***

***ConfigurableApplicationContext context = new ClassPathXmlApplicationContext(***

***"classpath:META-INF/spring/applicationContext.xml");***

***.....***

***}***

***}***

***3) PropertySource***

For the PropertySource abstraction, Spring will access the properties in the following default order --

a) System properties for the running JVM

b) Environment variables

c) Application-defined properties

In real life, you seldom need to interact directly with the Environment interface but will use a property placeholder in the form of ${} and inject the resolved value into Spring beans.

***@Configuration***

***@PropertySource({"classpath:application.properties", "classpath:application-${spring.profiles.active}.properties"})***

***static class NonDefaultProperties {***

***}***

***4) @Profile annotation***

Indicates that a component is eligible for registration when one or more specified profiles are active.

A profile is a named logical grouping that may be activated programmatically via ConfigurableEnvironment.setActiveProfiles(java.lang.String...) or declaratively by setting the spring.profiles.active property as a JVM system property, as an environment variable, or as a Servlet context parameter in web.xml for web applications. Profiles may also be activated declaratively in integration tests via the @ActiveProfiles annotation.

The @Profile annotation may be used in any of the following ways:

1) As a type-level annotation on any class directly or indirectly annotated with @Component, including @Configuration classes

2) As a meta-annotation, for the purpose of composing custom stereotype annotations

3) As a method-level annotation on any @Bean method

If a @Configuration class is marked with @Profile, all of the @Bean methods and @Import annotations associated with that class will be bypassed unless one or more of the specified profiles are active. A profile string may contain a simple profile name (for example "p1") or a profile expression. A profile expression allows for more complicated profile logic to be expressed, for example "p1 & p2". See Profiles.of(String...) for more details about supported formats.

This is analogous to the behavior in Spring XML: if the profile attribute of the beans element is supplied e.g., <beans profile="p1,p2">, the beans element will not be parsed unless at least profile 'p1' or 'p2' has been activated. Likewise, if a @Component or @Configuration class is marked with @Profile({"p1", "p2"}), that class will not be registered or processed unless at least profile 'p1' or 'p2' has been activated.

If a given profile is prefixed with the NOT operator (!), the annotated component will be registered if the profile is not active — for example, given @Profile({"p1", "!p2"}), registration will occur if profile 'p1' is active or if profile 'p2' is not active.

If the @Profile annotation is omitted, registration will occur regardless of which (if any) profiles are active.

**What are possible use cases where they might be useful?**

1) Different databases in development and production environments depending on profile.

2) Supporting both Hibernate and MyBatis ORM frameworks for database interaction depending on profile.

**Can you use @Bean together with @Profile?**

1) On Class, along with @Configuration, inner beans only create when @profile condition is met

2) On Class, along with @Component, inner beans create when condition met

3) On method, along with @Bean, bean creates when condition met

4) As meta-annotation, to create custom annotations.

**Can you use @Component together with @Profile?**

Yes we can use @Component (@Repository) together with @Profile as shown in below scenario.

The application should support both Hibernate and MyBatis ORM frameworks for database interaction. To support both Hibernate and MyBatis for database interaction, separate DAOs (FixedDepositHibernateDao and FixedDepositMyBatisDao) need to be created for Hibernate and MyBatis.

***@Profile({ "hibernate", "default" })***

***@Repository***

***public class FixedDepositHibernateDao implements FixedDepositDao {***

***private DataSource dataSource;***

***.....***

***@Autowired***

***public FixedDepositHibernateDao(DataSource dataSource) {***

***this.dataSource = dataSource;***

***}***

***.....***

***}***

In the above example listing, the @Profile annotation specifies that FixedDepositHibernateDao bean is registered with the Spring container only if the active profile is hibernate or default. This means, if the active profile is not hibernate or default, the Spring container will not create an instance of FixedDepositHibernateDao bean. Notice that the FixedDepositHibernateDao’s constructor accepts DataSource as argument that is used by Hibernate for connecting with the database and executing SQLs.

***@Profile("mybatis")***

***@Repository***

***public class FixedDepositMyBatisDao implements FixedDepositDao {***

***private DataSource dataSource;***

***.....***

***@Autowired***

***public FixedDepositMyBatisDao(DataSource dataSource) {***

***this.dataSource = dataSource;***

***}***

***.....***

***}***

In the above example listing, the @Profile annotation specifies that the FixedDepositMyBatisDao is registered with the Spring container only if the active profile is mybatis. FixedDepositMyBatisDao’s constructor accepts DataSource as argument which is used by MyBatis for connecting with the database and executing SQLs.

**How many profiles can you have?**

setActiveProfiles() accepts String… varargs, which is a String[]. In Java, arrays internally use integers for index, the max size is Integer.MAX\_VALUE. So theoretically it is 2^31-1 = 2147483647.

**How do you inject scalar/literal values into Spring beans?**

@Autowired cannot be used to autowire primitive values, or Strings, @Value specializes in this exactly.

It can be used to insert scalar values or can be used together with placeholders and SpEL in order to provide flexibility in configuring a bean.

***@Value(#{systemProperties.myProp}) - SPEL***

***@Value("${fooName}") - placeholder***

**What is @Value used for?**

Annotation at the field or method/constructor parameter level that indicates a default value expression for the affected argument.

Typically used for expression-driven dependency injection. Also supported for dynamic resolution of handler method parameters.

The @Autowired, @Inject, @Value, and @Resource annotations are handled by Spring BeanPostProcessor implementations. This means that you cannot apply these annotations within your own BeanPostProcessor or BeanFactoryPostProcessor types (if any). These types must be ‘wired up’ explicitly by using XML or a Spring @Bean method.

1) Setting (default) values of bean fields, method parameters and constructor parameters using #{systemProperties.myProp} style expressions.

2) Injecting environment variable values into bean fields, method parameters and constructor parameters.

3) Evaluate expressions and inject the result.

***@Autowired***

***public void splitName(@Value("#{configuration.splitName(‘FirstName LastName')}")***

***String[] splitName) {***

***this.splitName = splitName;***

***}***

***@Autowired***

***@Value("#{configuration.getCity()}")***

***public void city(String city) {***

***this.city = city;***

***}***

The @Value annotation can be used at method-level and method-parameter-level only if the method is annotated with @Autowired or @Resource or @Inject annotation. SpEL expression #{configuration.splitName('FirstName LastName')} results in invocation of

Configuration class's (@Component class) splitName method with ‘FirstName LastName’ as argument. This shows that SpEL expressions can be used to invoke methods that accept arguments.

**What is the difference between $ and # in @Value expressions?**

1) With $, reference a property name in the application’s environment.

Expressions are evaluated by the PropertySourcesPlaceholderConfigurer Spring bean prior to bean creation and can only be used in @Value annotations.

2) With #, SpEL

**What is Spring Expression Language (SpEL for short)?**

SpEL is a Spring expression language that you can use to query and manipulate objects (object graph) at runtime.

***@Component(value="sample")***

***public class Sample {***

***@Value("#{configuration.environment}") // static environment variable in Configuration class***

***private String environment;***

***@Value("#{configuration.getCountry()}") // getCountry() method in Configuration class***

***private String country;***

***@Value("#{configuration.state}") // getState() method in Configuration class***

***private String state;***

***}***

The above example listing shows that the @Value annotation specifies a value that has the syntax #{<spel-expression>}. The SpEL expression specified by @Value annotation is processed by AutowiredAnnotationBeanPostProcessor.

***@Component("configuration")***

***public class Configuration {***

***public static String environment = "DEV";***

***public String getCountry() {***

***return "Some country";***

***}***

***public String getState() {***

***return "Some state";***

***}***

***public String[] splitName(String name) {***

***return name.split(" ");***

***}***

***public String getCity() {***

***return "Some city";***

***}***

***}***

Using mathematical, relational and logical operators in SpEL --

***@Value("#{101 > 100}")***

***private boolean isGreaterThan;***

***@Value("#{3 > 2 && 4 > 3}")***

***private boolean isConditionTrue;***

***@Value("#{100 + 200 - 300\*1 + 4/2}")***

***private int totalAmount;***

isGreaterThan --> true

isConditionTrue --> true

totalAmount --> 2

Regular expressions are also supported by SpEL via matches operator --

***@Value("#{('abcd@xyz.com' matches '^[A-Za-z0-9+\_.-]+@(.+)$') == true ? true : false}")***

***private boolean isEmailId;***

Working with maps and lists in SpEL --

***@Value("#{listType[0]}")***

***private String listItem;***

***@Value("#{mapType['map key 1']}")***

***private String mapItem;***

In the above example listing, #{listType[0]} expression retrieves the first element from the listType, and #{mapType['map key 1']} expression retrieves the value corresponding to ‘map key 1’ key.

**What can you reference using SpEL?**

1) Static methods and static properties/fields

2) You can obtain reference to a bean by simply specifying the bean name in the @Value annotation.

***@Value("#{configuration}")***

***private Configuration myConfiguration;***

3) Properties and methods in Spring beans: @mySuperComponent.injectedValue

4) Properties and methods in Java objects: #javaObject.firstName

5) (JVM) System properties: @systemProperties['os.name']

6) System environment properties: @systemEnvironment['KOTLIN\_HOME']

7) Spring application environment: @environment['defaultProfiles'][0]

**What is the Environment abstraction in Spring?**

The Environment is an abstraction integrated in the container that models two key aspects of the application environment: profiles and properties.

***profiles -*** A profile is a named, logical group of bean definitions to be registered with the container only if the given profile is active.

***properties -*** Properties may originate from following main sources:

1) JVM system properties

2) Operating system environment variables

3) Command-line arguments

4) Application property configuration files - properties or yml

The Spring ApplicationContext interface extends the EnvironmentCapable interface, which contain one single method namely the getEnvironment(), which returns an object implementing the Environment interface. Thus a Spring application context has a relation to one single Environment object.

**ApplicationContext ctx = new GenericApplicationContext();**

**Environment env = ctx.getEnvironment();**

**boolean containsFoo = env.containsProperty("foo");**

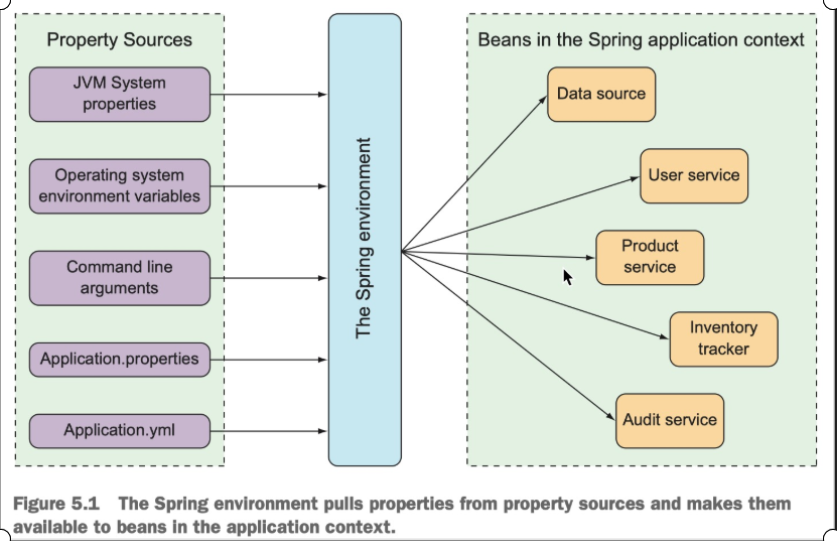
**System.out.println("Does my environment contain the 'foo' property? " + containsFoo);**

**// obtain Environment instance, and set the active profile**

**ConfigurableEnvironment environment = applicationContext.getEnvironment();**

**environment.setActiveProfiles("dev");**

Where can properties in the environment come from – there are many sources for properties – check the documentation if not sure. Spring Boot adds even more.



1) Application properties files,

2) JVM system properties,

3) Operating system environment variables,

4) JNDI,

5) servlet context parameters,

6) ad-hoc Properties objects,

7) Maps, and so on.