**Chapter 2 – Spring frame work**

­­­**Benefits of Spring Framework**

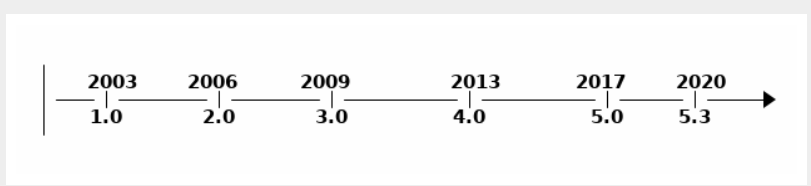
* easy to develop
* easy to test
* de-coupled
* High reusability

**Dependency Injection:** The relationships between components are established during the design phase, and linking dependents with their dependencies is called dependency injection.

This software design pattern implies that dependent components delegate the dependency resolution to an external service that will take care of injecting the dependencies. Inversion of control is a common characteristic of frameworks that facilitate injection of dependencies. And the basic idea of the dependency injection pattern is to have a separate object that injects dependencies with the required behaviour, based on an interface contract.

Spring framework uses following design pattern: **Factory**, **Abstract Factory**, **Singleton**, **Builder**, **Decorator**, **Proxy**, **Service Locator**, and **Reflection**

Prior to Spring version 2.5 XML configuration was used, since Spring 3.0 onwards it started to support JAVA configuration, Only with Spring 5.0 onwards – JAVA configuration has completely replaced the XML Configuration.



**Convention Over Configuration**

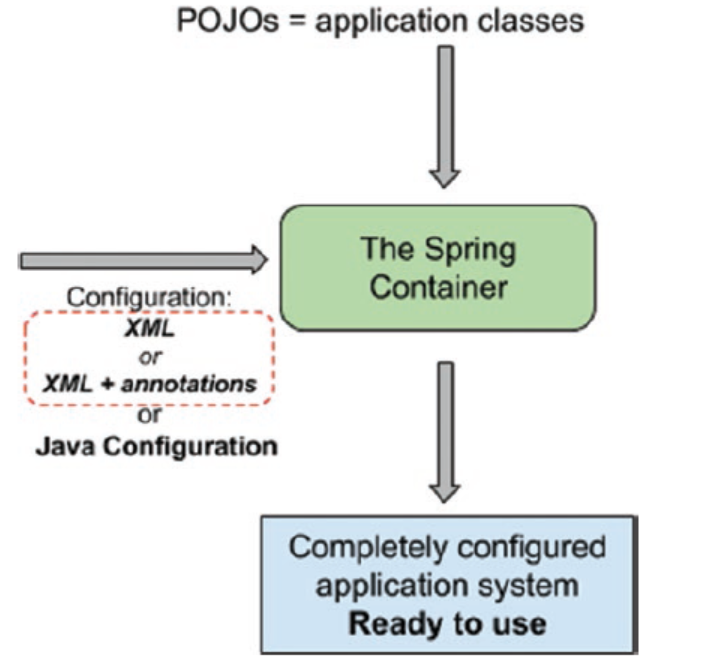
When configuring Spring applications, there are typical groups of infrastructure beans that have to be configured in a certain way, depending on the application we are building. After years of Spring applications being built, a pattern of configuration has emerged. When the same configuration is used in 90% of the applications written, this makes a good case for favouring convention over configuration. Spring Boot is the epitome of convention over configuration.

Java is an object-oriented programming language – it supports inheritance, use of generic helps it to achieve high level of inheritance.

The service layer is the bridge between the DAO layer and the Application layer – the dependency of the service layer can be injected using DI (dependency injection) so that they can be easy tested using stubbing the dependency. Spring makes the swapping of the dependencies easy based on the environment using spring profile.

**Spring IoC Container and Dependency Injection**

Spring IOC container is the core of the Spring Framework.



Spring responsibility to connect all the beans to make a working application – SpringFramework reads the beans dependency from the configuration created by the developer it can be an XML configuration file or JAVA configuration. Based on the configuration read, SpringFramework which is an external authority provide (inject) the dependency to the dependent bean, this happens during the runtime when SpringFramework put together the application after been complied – this gives more flexibility as the functionality of the application can be change by the external authority (SpringFramework) during the runtime without recompiling the code. Due to this low coupling – the application is easy to navigate and easy to maintain.

Spring framework helps replacing the dependent object with Stub making it easy for testing the core logic.

How does Hibernate propagates save functions from parent to dependent classes?

Hibernate, an ORM (Object-Relational Mapping) framework for Java, provides mechanisms to propagate save operations from parent entities to dependent entities through cascading operations. Cascade types can be defined for associations between entities, specifying how certain operations on one entity should be cascaded to associated entities.

When you save or update an entity, Hibernate checks the cascade settings defined for its associations and performs corresponding actions. These cascade types include:

* CASCADE: This setting propagates all save, update, and delete operations from the parent entity to its associated entities. If an entity has a CASCADE setting for a specific association, when you save the parent entity, Hibernate will automatically save the associated entities as well.
* ALL: Similar to CASCADE, but also includes evict and refresh operations.
* PERSIST: This setting propagates only save operations from the parent entity to its associated entities. It doesn't cascade updates or deletions.
* MERGE: This setting propagates only update operations. When the parent entity is merged, associated entities are also merged.
* REMOVE: This setting propagates only delete operations. When the parent entity is deleted, associated entities are also deleted.
* REFRESH: This setting propagates only refresh operations.
* DETACH: This setting propagates only detach operations.
* REPLICATE: This setting is specific to Hibernate's replication features and is used in distributed environments.

You can define these cascade types either using annotations (e.g., @OneToMany(cascade = CascadeType.ALL)) or through XML configuration.

For example, consider a parent entity Author and a dependent entity Book with a one-to-many association. If the cascade type for this association is set to CascadeType.ALL, when you save an Author object with its associated Book objects, hibernate will save both the Author and Book objects automatically.

Here's an example using JPA annotations:

@Entity

public class Author {

@Id

@GeneratedValue

private Long id;

**@OneToMany(mappedBy = "author", cascade = CascadeType.ALL)**

private List<Book> books = new ArrayList<>();

// other properties and methods

}

@Entity

public class Book {

@Id

@GeneratedValue

private Long id;

@ManyToOne

private Author author;

// other properties and methods

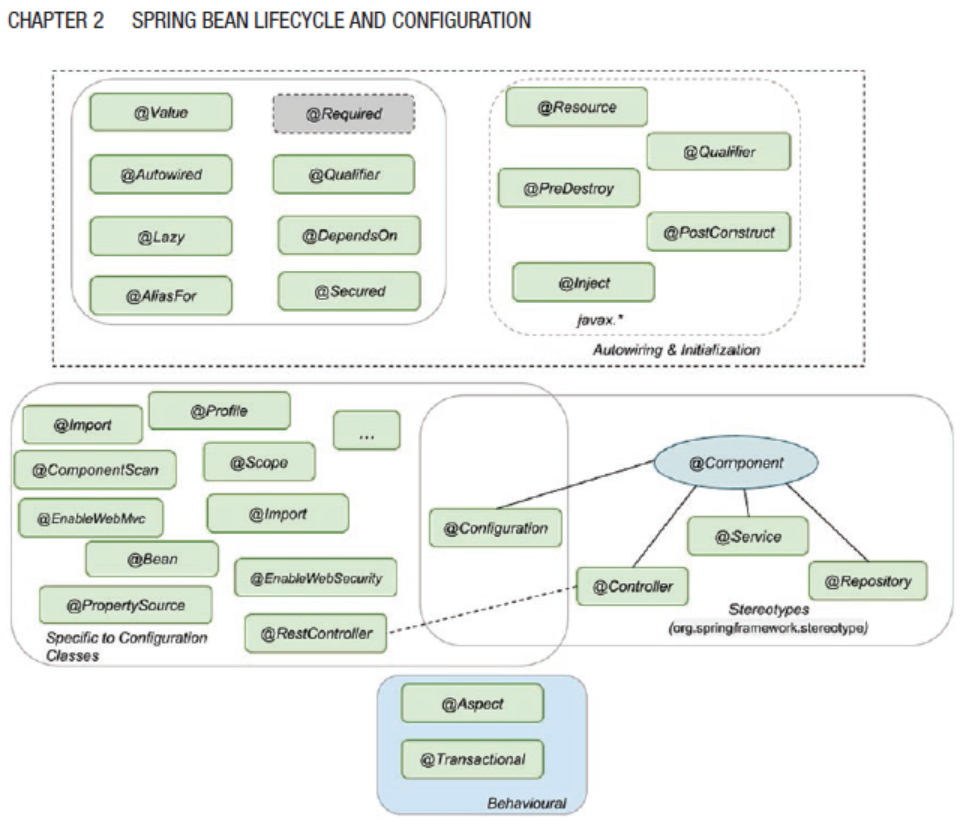
}

**NOTE:** Best Practice not to use schema version on the namespaces. Classpath is a common prefix used to refer xml configuration file(s), with classpath as prefix the spring container will look for the configuration file under src/main/resource directory.

Spring provided bean are also called as Infrastructure beans. These beans are need for spring framework to work; these dependencies need to be added for all spring projects.

| **spring modules** | **Descriptions** |
| --- | --- |
| spring-core | The fundamental parts of the Spring Framework, basic utility classes, interfaces, and enums that all other Spring libraries  depend on. |
| spring-beans | Together with spring-core provide the core components of the framework, including the Spring IoC container and dependency Injection features. |
| spring-context | Expands the functionality of the previous two, and it contains components that help build and use an application context. The ApplicationContext interface is part of this module, being the interface that every application context class implements. |
| spring-context-support | Provides support for integration with third-party libraries; for example, Quartz, FreeMarker, and a few more. |
| spring-expressions | Provides a powerful expression language (Spring Expression Language, also known as SpEL) used for querying and manipulating objects at runtime; for example, properties can be read from external sources decided at runtime and used to initialize beans. But this language is quite powerful, since it also  supports logical and mathematical operations, accessing arrays, and  manipulating collections. |

**Spring Bean Lifecyle and configuration**



| **Annotation** | **Descriptions** |
| --- | --- |
| @Component | Template for any Spring-managed component(bean). This annotation is used for those classes which has @Bean annotation. This is a part of the org.springframework.stereotype package and are the core annotations for creating beans. |
| @Repoisitory | Template for a component used to provide data access, specialization of the @Component annotation for the DAO layer. *This is a part of the org.springframework.stereotype package and are the core annotations for creating beans.* |
| @Service | Template for a component that provides service execution, specialization of the @Component annotation for the Service layer. *This is a part of the org.springframework.stereotype package and are the core annotations for creating beans.* |
| @Controller | Template for a web component, specialization of the @Component annotation for the web layer. *This is a part of the org.springframework.stereotype package and are the core annotations for creating beans.* |
| @RestContorller | This is specialized template created for REST webservices. |
| @Configuration | According to the official documentation @Configuration is not a stereotype annotation. Any class marked as @Configuration is the configuration class containing the definition of the bean.  This can be further annotate with @Profile or @Scope |
| @Autowire | Autowiring and initialization annotations are used to define which  dependency is injected.  @Autowired  annotation and can be used on fields, constructors, setters, and even methods. |
| @Inject |  |
| @Resource |  |
| @Required | Spring annotation that marks a dependency as mandatory. It can be used on setter methods, but *since Spring 5.1 was deprecated* as of in favour of using constructor injection for required settings. |
| @Lazy | When this annotation is used, dependency will be injected the first time it is used.  Although this annotation exists, avoid using it if possible. When a  Spring application is started ApplicationContext implementations eagerly create and configure all singleton beans as part of the initialization process, this is useful because configuration errors in the configuration or supporting environment (e.g., database) can be spotted fast. When @Lazy is being used spotting these errors might be delayed.  This annotation is useful when one like to avoid pre-creating a large object in memory from starting of the application. |
| @ComponentScan | This can be used to filtering and reducing the scope of the scan for searching @Component annotated classes. |
| @AliasFor | @AliasFor annotation is used to declare alias semantics for annotation attributes.  @Component  @Retention(RetentionPolicy.RUNTIME)  @Target(ElementType.TYPE)  public @interface MyService {  @AliasFor(annotation = Component.class) // Declaring alias  String value() default "";  @AliasFor(annotation = Component.class, attribute = "value") // Using the alias  String name() default ""; // This attribute serves as an alias for the 'value' attribute in @Component  }  In this example, MyService is a custom annotation that serves as a stereotype for Spring beans, similar to @Component. The name attribute in MyService is declared as an alias for the value attribute in @Component using @AliasFor. This means that when you use MyService, you can specify a value for the name attribute, and it will be automatically mapped to the value attribute in @Component. |
| @Profile | @Profile annotation is used to indicate that a component, configuration class, or bean definition should be active only when a specific set of profiles is active.  Profiles can be activated in various ways, such as through environment properties, system properties, or programmatically in your application. When a profile is activated, Spring evaluates the @Profile annotations to determine which components should be instantiated and which configurations should be applied. |
| @PropertySource | @PropertySource annotation that is commonly used in conjunction with the @Configuration annotation to specify the location of properties files to be loaded into the Spring application context.  @Configuration  @PropertySource("classpath:application.properties")  public class AppConfig {  } |

**Application Context**

The org.springframework.context.ApplicationContext is the interface implemented by classes that provide the configuration for an application. This interface is an extension of the interface org.springframework.beans.factory.BeanFactory, which is the root interface for accessing a Spring Bean container.

The interfaces BeanFactory and ApplicationContext represent the Spring IoC container. Here, BeanFactory is the root interface for accessing the Spring container. It provides basic functionalities for managing beans. On the other hand, the ApplicationContext is a sub-interface of the BeanFactory. Therefore, it offers all the functionalities of BeanFactory.

There are different ways to instantiate application context

ApplicationContext ctx = new AnnotationConfigApplicationContext(SimpleConfig.class);

ApplicationContext context = new FileSystemXmlApplicationContext(C:/users/ABCUSer/simpleConfig.xml);

ApplicationContext context = new ClassPathXmlApplicationContext("applicationcontext/account-bean-config.xml");

AnnotationConfigWebApplicationContext is a web variant of AnnotationConfigApplicationContext, this is configured Spring's ContextLoaderListener servlet listener or a Spring MVC DispatcherServlet in a web.xml file.

public class MyWebApplicationInitializer implements WebApplicationInitializer {

public void onStartup(ServletContext container) throws ServletException

{

AnnotationConfigWebApplicationContext context = new AnnotationConfigWebApplicationContext();

context.register(AccountConfig.class);

context.setServletContext(container);

// servlet configuration

}

}

XmlWebApplicationContext is used for the XML based configuration in a web application

public class MyXmlWebApplicationInitializer implements WebApplicationInitializer {

public void onStartup(ServletContext container) throws ServletException

{

XmlWebApplicationContext context = new XmlWebApplicationContext();

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

context.setServletContext(container);

// Servlet configuration

}

}

Spring context use prefix to decide where to look for file base configuration.

|  |  |
| --- | --- |
| **Prefix** | **Where configuration file will be looked** |
| No prefix | If there is no prefix mentioned, then spring will look for the configuration file in the root directory where the class creating the context is executed.  In springboot project it will be typically look under main or test folder depending upon where ApplicationContext is used. |
| classpath : | If classpath: prefix is used, then the configuration will be read from the classpath location.  In springboot project it will be typically looked under resource folder. |
| file: | If file: prefix is used then the configuration file will be loaded from the absolute path. |
| http: | If http: prefix is used the it will be loaded from the httpURL and the resource will be of type UrlResource. If the resource  is used to create an application context, the  WebApplicationContext class is suitable. |

Depending upon the context class, the resource loaded class will change accordingly.

* ClassPathXmlApplicationContext is used then resource class will be of type ClassPathResource .
* FileSystemXmlApplicationContext is used then resource class will be of type FileSystemResource.
* WebApplicationContext is used then resource class will be of type ServletContextResource.

An ApplicationContext implementation provides the following.

* Access to beans using bean factory methods
* The ability to load file resources using relative or absolute paths or URLs
* The ability to publish events to registered listeners
* The ability to resolve messages and support internationalization (most used in international web applications)

|  |  |
| --- | --- |
| **Application Context** | **Bean Factory** |
| Here we can have more than one config files possible | Only one config file or .xml file can be used |
| Application contexts can publish events to beans that are registered as listeners. | Doesn’t support this feature. |
| Support internationalization (I18N) messages | Doesn’t support this feature. |
| Support application life-cycle events, and validation. | Doesn’t support this feature. |
| Supports many enterprise services such as JNDI access, EJB integration, remoting | Doesn’t support this feature. |

Page # 52 – 28.10.2021 -12:51

@Configuration class has the bean define within itself, however @Component class the object of the class is the bean, there is no method with @Bean annotation. By default, the bean created is singleton in nature – unless explicitly configure with prototype scope. When there is a call for the object of an already created bean object – Spring IOC container will not instantiate another bean, instead provide the bean object of already created bean. This is achieved through a mechanism called as proxying.

Properties can be originated from:

|  |  |
| --- | --- |
| Property Files | Description |
| Property Files |  |
| JVM System Properties |  |
| System environment properties |  |
| JNDI |  |
| Property Instance |  |
| Map Instance |  |

|  |  |
| --- | --- |
| **Dependency Injection Types** | **Description** |
| Constructor injection | The Spring IoC container injects the dependency by providing it as an argument for the constructor.  @Component  public class ComposedBeanImpl implements ComposedBean {  private SimpleBean simpleBean;  @Autowired  public ComposedBeanImpl(SimpleBean simpleBean)  {  this.simpleBean = simpleBean;  }  }  @Autowired is not necessary if, there is a single constructor within a bean component.  If there are more than one constructor annotated with @Autowired, spring container will throw an error as it causes confusion which constructor to be called for bean instantiation.  In spring , its preferred to used construction injection as its immutable in nature and also ensure that the bean is created and initialized properly before been used. |
| Setter injection | The Spring IoC container injects the dependency as an argument for a setter.  @Autowired  public void setBeanTwo(BeanTwo beanTwo) {  this.beanTwo = beanTwo;  }  In case of the setter the bean instantiation and initialization happen in two different steps  There are three main reasons for using setter injection.   * It allows reconfiguration of the dependent bean, as the setter can be called explicitly later in the code, and a new bean can be provided as a dependency (this obviously means that a bean created using setter injection is not immutable) * Preferably, it is used for bean dependencies that can be set with default values inside the bean class * Third-party code only supports setter injections. |
| Filed Injection | The Spring IoC container injects the dependency  directly as a value for the field (via reflection and this requires the open directive in the module-info.java file)  Using field level injection, it hides the dependencies which makes hard for the other developer read dependencies AND also it makes writing of the test difficult, especially integration test.  Recommend using field injection only in the following contexts.   * @Configuration classes: Bean A is declared in configuration class A1, and bean B declared in configuration class B1, depends on bean A. * @Configuration classes: To inject infrastructure beans that are created by the Spring IoC container and need to be customized. (Be careful when doing this because it ties your implementation to Spring.) * Test classes: The tested bean should be injected using field injection as it keeps things readable. |

NOTE: Constructor and Setter Injection *used proxying* to inject dependencies – however field Injection *uses reflection* to autowired the dependencies as there is no other means available to access a private filed, which has performance impact.

NOTE: Bean Name – spring downcast the first letter of the ClassName for example – SpringBean Class, bean name will be springBean. In case one needs to change the Bean name then it can be achived by setting a string argument to @Component annotation , i.e. @Component (“SPRINGBEAN”).

Currently Alias name is not supported for the stereotype. However its possible to have multiple names of a single Bean, where one will be unique identifier and others will be just the alias. Here is how one can assign multiple names to a single bean.

Spring tries to inject based on @Qualifier, if no match found then it will try to load using beanName. If nothing is found then spring will throw NoSuchBeanDefinitionException.

If there are more than one bean with the same type then spring container NoUniqueBeanDefinitionException.

@Bean(name= {"beanOne", "beanTwo"})

SimpleBean simpleBean()

{

return new SimpleBeanImpl();

}

Page # 71 29.10.2021 – 04:39 PM

Public class ComponentBeanImpl

{

Private String stringValue;

Private Boolean BooleanValue;

Private SimpleBean simpleBean;

@Autowired

componnentBeanImpl(@Value(“some-string-value”)String stringValue,@Value (true) Boolean BooleanValue, SimpleBean simpleBean)

{

This.stringVlaue=stringValue;

This.booleanValue=BooleanValue;

This.simpleBean=simpleBean;

}

}

The above code snippet is equal to

SimpleBean simpleBean=new simpleBean;

ComponentBeanImpl composedBean = new ComponentBeanImpl ("some-string-value", true, simpleBean);

When using constructor-based injection – one need not to explicitly mention the required attributes that needs to be created during bean instantiation. If @Autowired field level injections is used then one can used required attribute which can be set to true or false to make the value mandatory or optional.

@Atowired(requried=false)

SpringBean springBean;

The above code snippet XML configuration equivalent would be

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

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

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

<bean name="simpleBeanImpl"

class="com.apress.cems.beans.ci.SimpleBeanImpl"/>

<bean name="componnentBeanImpl "

class="com.apress.cems.beans.ci.componnentBeanImpl ">

<constructor-arg index="0" ref="simpleBeanImpl"/>

<constructor-arg index="1" value="some-string-value" />

<constructor-arg index="2" value="true" />

</bean>

C namespace and P namespace are shorter tags to make it more readable.

**Bean Scope**

| **Bean Scope** | **Annotation** | **Description** |
| --- | --- | --- |
| Singleton | None scope annotation (default)  @Scope(“singleton”)  @Scope(ConfigurableBean  Factory.SCOPE\_SINGLETON) | Default scope, Spring IOC container will maintain a single instance of the bean till application is shutdown OR application context is not closed. |
| Prototype | @Scope("prototype"),  @Scope(ConfigurableBean  Factory.SCOPE\_PROTOTYPE) | Every time a request is made for this specific  bean, the Spring IoC creates a new instance. |
| request | @Scope("request"),  @RequestScope,  @Scope(WebApplication  Context.SCOPE\_REQUEST) | The Spring IoC creates a bean instance for each HTTP request. Only valid in the context of a web-aware Spring ApplicationContext. |
| Session | @Scope("session"),  @SessionScope,  @Scope(WebApplication  Context.SCOPE\_SESSION) | The Spring IoC creates a bean instance for each HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |
| application | @Scope("application"),  @ApplicationScope,  @Scope(WebApplication  Context.SCOPE\_APPLICATION) | The Spring IoC creates a bean instance  for the global application context. Only  valid in the context of a web-aware Spring ApplicationContext. |
| websocket | @Scope("websocket") | The Spring IoC creates a bean instance  for the scope of a WebSocket. Only valid  in the context of a web-aware Spring  ApplicationContext. |
| thread | @Scope("thread") | Introduced in Spring 3.0, *it is available, but not registered by default*, so the developer must explicitly register it in the same way as if a custom scope would be defined. |

**What is the difference between application scope and singleton scope?**

In application scope, container creates one instance per web application runtime. It is almost similar to singleton scope, with only differences is application scoped bean is singleton per ServletContext , whereas singleton scoped bean is singleton per ApplicationContext . There can be multiple application context for a single application.

**Proxy pattern**

*For example a TheamManager bean needs a userSetting bean – however this userSetting bean should be different for each login user , creating userSetting bean for each login user can be achieved by setting bean scope of SESSION for the user setting bean but how do we ensure that the TheamManager calls setter method everytime there is a new userSetting bean available. TO resolve this issue proxy pattern can be helpful ~ A proxy is an implementation that wrap around the actual implantation and provide the exact same behaviour in a transparent manner.*

*If the bean is implementing an interface, then the proxy will also implement the same interface; if the bean is extending a class, then the proxy will also extend the same class. Using the proxy, Spring can provide the behaviour of the refreshing state based on each HTTP session.*

@Component

@Scope(value = WebApplicationContext.SCOPE\_SESSION,

proxyMode = ScopedProxyMode.INTERFACES)

*Class UserSetting implements BasicUserSettings*

*{*

*}*

@Component

@Scope(value = WebApplicationContext.SCOPE\_SESSION,

proxyMode = ScopedProxyMode.TARGET\_CLASS)

*Class UserSetting extends BasicUserSettings*

*{*

*}*

@Component

@Scope(value = WebApplicationContext.SCOPE\_SESSION,

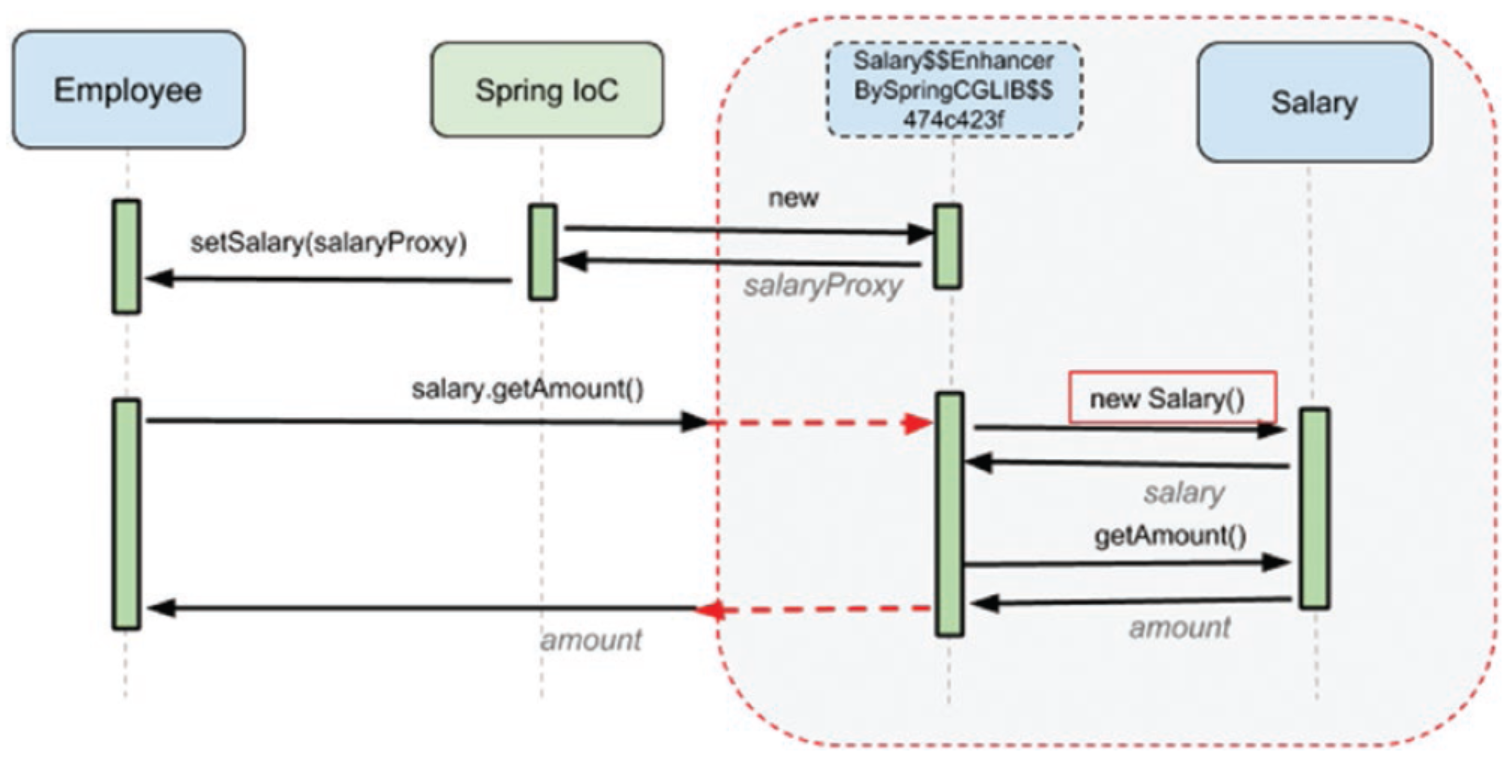
proxyMode = ScopedProxyMode.DEFAULT)

//proxy mode is set to default – spring container will play safe and //create a CGLIB based class proxy by default

*Class UserSetting extends BasicUserSettings*

*{*

*}*



The problem with proxyMode = ScopedProxyMode.TARGET\_CLASS that only public & final method can be proxied.

**30-10-2021 10:34 AM Page # 99**

**@Scope annotation can also be applicable to @Bean**

@Bean

@Scope(value = "prototype",

proxyMode = ScopedProxyMode.TARGET\_CLASS)

Salary salary()

{

return new Salary();

}

**Bean Lifecylce**

Spring bean lifecycle has three phases – **initialization**, **uses** and **destruction**.

|  |  |
| --- | --- |
| * **Initialization** is the phase where spring IOC container reads the bean definition, create the bean instance, inject dependencies and make the bean read for uses. * **Uses** is the phase where bean is ready and ready to server. * **Destruction** is the phase where application is shutdown, application context is closed and bean is handed over to garbage collection. |  |

Lifecycle of a spring bean

1. Bean Creation
   1. The **bean is instantiated**, the bean factory calls the constructor of each bean, if the bean is created using dependency injection, then the bean will create the dependencies first and then instantiated.
   2. The **bean dependency injected,** if the bean dependency is injected using field injection or setter method injection then the required bean dependency is injected.
   3. Bean is **fully initialized**.
   4. The post process bean is invoked – if the bean has any method annotated with @**PostConstruct** then then that method will be called. NOTE: its good practice to make the @PostConstruct method as private – as these are the method that needs to be called by the spring container once in the bean lifetime its not intended to have any one else call this method. Spring container uses reflection to find and call it. @**PostConstruct method don’t take any arguments and returns void. ONLY one method can be annotated with @PostConstruct.**

There are three possible ways to initialized a bean

1. Using @PostConstruct
2. By implementing & overriding InitializingBean. afterPropertiesSet() method.
3. By using property intiMethod of @Bean annotation.

**NOTE: Spring suggest not to used InitializingBean class afterPropertiesSet() method class as it creates a unnecessary coupling between the application and spring framework, thus only#1 and #3 are prescribed methods.**

**Difference between initMethod and @PostContruct method?**

@ PostConstructs method is called before the intailization of the bean, where @Bean initMethod is called post initialization of the bean.

If your bean implements InitializingBean and overrides afterPropertiesSet , first @PostConstruct is called, then the afterPropertiesSet and then init-method.

**There are three different ways to notified before destruction**

1. **By overriding the destroy method of** **org.springframework.beans.factory.DisposableBean interface.**
2. **Another way to annotate a method @PreDestory method**
3. **Configure a bean attribute destroyMethod (@Bean(destoryMethod=”NameOfTheBeanMethod”)**

Note in order to gracefully destroy a bean, we have to call close() method of applicationContext or registerShutdownHook() method.

ConfigurableApplicationContext ctx =

new AnnotationConfigApplicationContext(SimpleConfig.class);

ctx.close();

ctx.registerShutdownHook();

**Bean Definition**

public class FunBean implements InitializingBean, DisposableBean {  
 private Logger logger = LoggerFactory.*getLogger*(FunBean.class);  
  
 private DepBean depBean;  
  
 public FunBean() {  
 logger.info("Stage 1: Calling the constructor");  
 }  
  
 @Autowired  
 public void setDepBean(DepBean depBean) {  
 logger.info("Stage 2: Calling the setter");  
 this.depBean = depBean;  
 }  
  
 @PostConstruct  
 void initMethod() {  
 logger.info("Stage 3: Calling the initMethod.");  
 }  
  
 @Override  
 public void afterPropertiesSet() throws Exception {  
 logger.info("Stage 4: Calling the afterPropertiesSet.");  
 }  
  
 void beanInitMethod(){  
 logger.info("Stage 5: Calling the beanInitMethod.");  
 }  
  
  
 @PreDestroy  
 void destroyMethod() {  
 logger.info("Stage 6: Calling the destroyMethod.");  
 }  
  
 @Override  
 public void destroy() throws Exception {  
 logger.info("Stage 7: Calling the destroy.");  
 }  
  
 void beanDestroyMethod(){  
 logger.info("Stage 8: Calling the beanDestroyMethod.");  
 }  
  
}

**Bean Configuration**

@Configuration  
public class FunBeanConfig {  
  
 @Bean(initMethod = "beanInitMethod", destroyMethod = "beanDestroyMethod")  
 FunBean funBean(){  
 return new FunBean();  
 }  
}

**Bean Invocation**

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(FunBeanConfig.class);

ctx.registerShutdownHook(); // THIS OR ctx.close()

FunBean funBean = ctx.getBean(FunBean.class);

Ctx.close(); // THIS OR ctx.registerShutdownHook()

TRACE o.s.c.a.ConfigurationClassBeanDefinitionReader - Registering bean definition for @Bean method com.apress.cems.fun.FunBeanConfig.funBean()

TRACE o.s.c.a.ConfigurationClassEnhancer - Successfully enhanced com.apress.cems.fun.FunBeanConfig; enhanced class name is: com.apress.cems.fun.FunBeanConfig$$EnhancerBySpringCGLIB$$d706912d

TRACE o.s.c.a.ConfigurationClassPostProcessor - Replacing bean definition 'funBeanConfig' existing class 'com.apress.cems.fun.FunBeanConfig' with enhanced class 'com.apress.cems.fun.FunBeanConfig$$EnhancerBySpringCGLIB$$d706912d'

INFO c.a.c.f.FunBean - Stage 1: Calling the constructor

TRACE o.s.c.a.CommonAnnotationBeanPostProcessor - Found init method on class [com.apress.cems.fun.FunBean]: void com.apress.cems.fun.FunBean.initMethod()

TRACE o.s.c.a.CommonAnnotationBeanPostProcessor - Found destroy method on class [com.apress.cems.fun.FunBean]: void com.apress.cems.fun.FunBean.destroyMethod()

TRACE o.s.c.a.CommonAnnotationBeanPostProcessor - Registered init method on class [com.apress.cems.fun.FunBean]: [org.springframework.beans.factory.annotation.InitDestroyAnnotationBeanPostProcessor$LifecycleElement@4601611](mailto:org.springframework.beans.factory.annotation.InitDestroyAnnotationBeanPostProcessor$LifecycleElement@4601611)

TRACE o.s.c.a.CommonAnnotationBeanPostProcessor - Registered destroy method on class [com.apress.cems.fun.FunBean]: [org.springframework.beans.factory.annotation.InitDestroyAnnotationBeanPostProcessor$LifecycleElement@6360f97b](mailto:org.springframework.beans.factory.annotation.InitDestroyAnnotationBeanPostProcessor$LifecycleElement@6360f97b)

TRACE o.s.b.CachedIntrospectionResults - Getting BeanInfo for class [com.apress.cems.fun.FunBean]

TRACE o.s.b.CachedIntrospectionResults - Caching PropertyDescriptors for class [com.apress.cems.fun.FunBean]

TRACE o.s.b.CachedIntrospectionResults - Found bean property 'class' of type [java.lang.Class]

TRACE o.s.b.f.a.InjectionMetadata - Registered injected element on class [com.apress.cems.fun.FunBean]: AutowiredMethodElement for public void com.apress.cems.fun.FunBean.setDepBean(com.apress.cems.fun.DepBean)

INFO c.a.c.f.FunBean - Stage 2: Calling the setter

INFO c.a.c.f.FunBean - Stage 3: Calling the initMethod.

INFO c.a.c.f.FunBean - Stage 4: Calling the afterPropertiesSet.

INFO c.a.c.f.FunBean - Stage 5: Calling the beanInitMethod.

INFO c.a.c.f.FunBean - Stage 6: Calling the destroyMethod.

INFO c.a.c.f.FunBean - Stage 7: Calling the destroy.

INFO c.a.c.f.FunBean - Stage 8: Calling the beanDestroyMethod.

**Bean inheritance**

Another advantage of using XML is that abstract classes can provide a template for

beans of types that extend the abstract class to be created.

**Bean property value conversion**

For primitive dateType spring provides out of the box conversion mechanism, however to provide any custom conversion one need to write a implementation class implementing org.springframework.core.convert.converter.Converter<S,T> and override its ONLY method T convert(S source). AND add new converter method to its list of supported converter methods of ConversionService.

For example, converting string value "1977-10-16" to LocalDate one need to write a custom converter class implementing Convert interface & then declare a bean of type org.springframework.core.convert.ConversionService, named it as conversionService AND add StringToLocalDate converter to its list of supported converters.

If one does not add their own custom conversion following error will be thrown

Failed to convert value of type 'java.lang.String' to required type 'java.time.LocalDate'; nested exception is java.lang.IllegalStateException: Cannot convert value of type 'java.lang.String' to required type 'java.time.LocalDate': no matching editors or conversion strategy found

@Autowired

public PersonBean(@Value("1977-10-16") LocalDate birthDate,

@Value("John Mayer") String name)

{

this.birthDate = birthDate;

this.name = name;

}

@Component

public class StringToLocalDate **implements Converter<String,**

**LocalDate>**

{

private DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");

//Override convert method and provide required conversion.

**@Override**

**public LocalDate convert(String source)**

**{**

**return LocalDate.parse(source, formatter);**

**}**

}

@Configuration

public class ConfigurationClass {

@Autowired

**StringToLocalDate** stringToLocalDateConverter;

@Bean

ConversionServiceFactoryBean conversionServiceFactoryBean()

{

ConversionServiceFactoryBean factory = new ConversionServiceFactoryBean();

factory.setConverters(Set.of(**stringToLocalDateConverter**,stringToDate));

return factory;

}

@Bean

ConversionService conversionService(ConversionServiceFactoryBean factory)

{

return factory.getObject();

}

}

**SpEL language**: supports getting and setting property values, method invocation, and usage of arithmetic and logic operators and bean retrieval from the Spring IoC container.

SpEL language can be used to copy value from one bean to another bean, for example dataSource bean is getting data populated from dbProps bean.

@Bean

public Properties dbProps()

{

Properties p = new Properties();

p.setProperty("driverClassName", "org.h2.Driver");

p.setProperty("url", "jdbc:h2:~/sample");

p.setProperty("username", "sample");

p.setProperty("password", "sample");

return p;

}

@Bean

public DataSource dataSource(

@Value("#{dbProps.driverClassName}")String driverClassName,

@Value("#{dbProps.url}")String url,

@Value("#{dbProps.username}")String username,

@Value("#{dbProps.password}")String password) throws

SQLException

{

DriverManagerDataSource ds = new DriverManagerDataSource();

ds.setDriverClassName(driverClassName);

ds.setUrl(url);

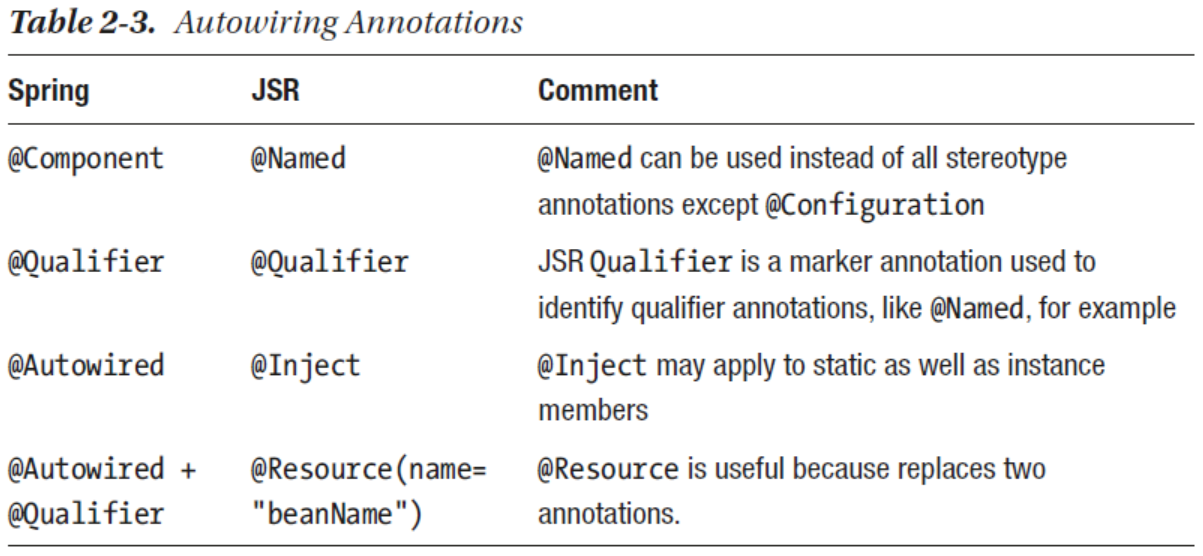
ds.setUsername(username);

ds.setPassword(password);

return ds;

}

**TIPS : Ctrl + Alt +B provided the name of all the inherited classes.**



Spring reads the property files from in below sequential order, the properties define in the upper position overrides the property of the below location.

1. If the application is running on the development mode (i.e. it has devtools libraries included) then properties from ~/.spring-boot-devtools.properties from home folder will be read first.
2. If @TestPropertySource annotation is found on the test class then then properties will be read from there. [highest priority]
3. Properties attribute on test classes on class annotation with @SpringBootTest then the properties will be read from those classes.
4. Properties and values provided using the command line argument will be considered.
5. Properties provided as value of SPRING\_APPLICATION\_JSON will be consider.
6. ServletConfig init parameter.
7. ServletContext init parameter.
8. JNDI attribute from java:comp/env
9. Java system properties accessible by calling System.getProperties(“property-name”).
10. Operating system environment variable
11. **A RandomValuePropertySource that has properties only in random.\*.**
12. Profile specific application.property file available outside package jar
13. Profile specific application.property file available within the package jar
14. Application.property file available outside the package jar
15. Application.property file available within the package jar
16. @PropertySource annotations on @Configuration classes.
17. Default properties (specified by calling SpringApplication.

setDefaultProperties). [Least priority]

**Chapter 3 – Testing Spring Application**

­­­

Different Types of testing

1. **Test Driven Development.**
2. **Unit Testing and Integration Testing**: Unit testing – testing smallest possible testable unit, while in integration testing – testing end-to-end scenario.
3. **Testing with Stubs**: There are few disadvantages of using stubs
   1. Can only be written for simpler/smaller application.
   2. Any changes made to the interface – all the implementing classes of the stub also needs to be changed. (JAVA 8 – interface default methods can come handy to overcome this limitation).
   3. Any changes made to the base level interface – all stubs needs to refactored for that change.

Junit5 as three main modules

* + 1. Junit 5 platform: the junit 5 testing platform for launching Junit5 platform on the JVM
    2. Junit Jupiter Library – New library written for Junit 5
    3. Junit Vintage – Junit library providing engines that support execution of Junit 3 and Junit 4 test.

Difference between Junit 4 and Junit 5.

1. Testing with Mocks: Mocks helps to mock the behavior of the dependent unit of code, and help testing the unit under test with ease. There are different mocking frameworks available for Java some are
   1. EasyMock: Spring was using it extensively until Spring-3 , where it embrace Mockito for its mocking needs. When Easy mock is called following things happens
      1. Declare a mock
      2. Create a mock
      3. Inject a mock
      4. Record what mock is supposed to do
      5. Tell mock actual testing that is been done
      6. Test
      7. Make sure the methods are called on the mock
      8. Validate the execution
   2. jMock : Nice and stable API for testing complex stateful logic.
      1. Declare a mock
      2. Declare and define the context of object under test , an instance of the org.jmock.Mockery class.
      3. Create a mock
      4. Inject a mock
      5. Define the expectation from the mock
      6. Test
      7. Check that mock was used
      8. Validate the execution.
   3. Mockito: Currently Spring prefer this mocking framework – its powerful & provides partial mock capabilities , even when real methods are invoked it can be used for verify and subbed.

Simplest of the all –

* + 1. Declare and create the mock
    2. Inject mock
    3. Define the behavior of the mock
    4. Test
    5. Validate the execution
  1. PowerMock : Its an extension of the easy mock, it usages a custom classloader and bytecode manipulator to test – private & static methods including constructors , final methods / Classes making it possible to test un-testable code.

**Testing with Spring – SQL annotation**

**@Sql**

@Sql annotation is used on method along with @Test annotation to specify the sql script that needs to be executed before running the test method. @Sql provides two attributes Sql.ExecutionPhase.

BEFORE\_TEST\_METHOD [default] this specifies the script to be executed before executing the test methods. Sql.ExecutionPhase.AFTER\_TEST\_METHOD this specifies the script to be executed after the execution of the test method.

@Sql annotation can also be applied to class level

@Sql({"classpath:db/test-data-two.sql"})

@Sql(statements = {"INSERT INTO PERSON(ID, USERNAME, FIRSTNAME, LASTNAME, PASSWORD, HIRINGDATE, VERSION, CREATED\_AT, MODIFIED\_AT) VALUES (2, 'irene.adler', 'Irene', 'Adler', 'id123ds', '1990-08-18', 1, '1990-07-18', '1998-01-18');"})

@Sql(statements = "DELETE FROM PERSON", executionPhase = Sql.ExecutionPhase.AFTER\_TEST\_METHOD)

@Sql(scripts = {"classpath:db/person-schema.sql", "classpath:db/test-data.sql"},executionPhase = Sql.ExecutionPhase.BEFORE\_TEST\_METHOD )

@SpringJUnitConfig(classes = RepositoryTest5.TestCtxConfig.class)

public class JunitTestCassName {

//….

//….

//….

}

**@SqlConfig**

@SqlConfig annotation provides information about the syntax used in SQL scripts provided as argument to an @Sql annotation.

Example

@Sql(scripts = "classpath:db/test-data-one.sql",

config = @SqlConfig(commentPrefix = "`", separator = "@@"))

**@SqlGroup**

**@SqlGroup** annotation is used for grouping multiple Sql annotations, which use different scripts/statements. You can use it on test classes or methods.

**Spring – SQL testing also supports @BeforeTransation @AfterTransation, @Rollback and @Commit.**

**NOTE : when using @ActiveProfiles, any bean that are marked with @Profile with active profile name are loaded , and the beans that are NOT marked with @Profile are also loaded regardless if there is a active profile been mention for them or not.**

**Chapter 4 – Aspect Oriented Programming (AOP) with Spring**

­­­

Aspect oriented programming is a way of programming that aims to increase reusability & mobility of cross cutting functionality which are usually tangled with business logic, which otherwise is difficult to segregate – cross cutting functionalities are auditing, transaction, security, logging etc.

*This is achieved by, adding an additional behavior to the existing code while the code is complied, and directly affect byte code. In codding this is done by, adding* ***advice*** *containing code that will be executed in a location name* ***join-points*** *specified as point-cuts.*

The following are the feature that can be implemented using AOP

* Logging
* Data Validation
* Caching
* Internationalization
* Error detection and correction
* Memory management
* Performance Monitoring
* Synchronization

Spring AOP

Spring AOP – cannot manage object those are NOT managed by the spring container. Spring AOP uses dynamic proxies for weaving aspect – once the target code is compiled the bytecode is not change, instead the target code is effectively wrapped by a runtime object that either implements the same interface OR extend the class of that of target object.

Spring AOP framework comprises of following libraries

* **Spring-AOP:** provides AOP alliance API compliant components that can define method interceptors and pointcuts so that code with different responsibilities can be cleanly decoupled.
* **Spring-aspects: Provides integration with AspectJ**
* **spring-instrument**: provides class instrumentation support and classloader implementations that can be used on application servers.

Spring AOP Terminology

1. **Aspect:** A class containing code specific to a cross-cutting concern. A class declaration is recognized in Spring as an aspect if it is annotated with the @Aspect annotation.
2. **Weaving:** A synonym for this word is interlacing, but in software the synonym is linking and it refers to aspects being combined with other types of objects to create an advised object.
3. **Join point:** A point during the execution of a program. In Spring AOP, a join point is always a method execution. Basically, the join point marks the execution point where aspect behavior and target behavior join.
4. **Target object:** An object to which the aspect applies.
5. **Target method:** the advised method.
6. **Advice:** The action taken by an aspect at a join point. In Spring AOP, there are multiple advice types.

* **Before advice:** Methods annotated with **@Before** that will execute before the join point. These methods do not prevent the execution of the target method unless they throw an exception.
* **After returning advice**: Methods annotated with **@AfterReturning** that will execute after a join point completes normally, meaning that the target method returns normally without throwing an exception.
* **After throwing advice:** Methods annotated with **@AfterThrowing** that will execute after a join point execution ends by throwing an exception.
* **After (finally) advice:** Methods annotated with **@After that** will execute after a join point execution, no matter how the execution ended.
* **Around advice:** Methods annotated with **@Around** intercept the target method and surround the join point. This is the most powerful type of advice since can perform custom behavior before and after the invocation. It has the responsibility of choosing to perform the invocation or return its own value, and it provides the option of stopping the propagation of an exception.

1. **Pointcut**: A predicate used to identify join points. Advice definitions are associated with a pointcut expression and the *advice will execute on any join point matching the pointcut expression*. Pointcut expressions are defined using AspectJ Pointcut Expression Language Pointcut expressions can be defined as arguments for Advice annotations or as arguments for the **@Pointcut** annotation.
2. **Introduction:** Declaring additional methods, fields, interfaces being implemented, and annotations on behalf of another type. Spring AOP allows this using a suite of AspectJ @Declare\* annotations that are part of the aspectjrt library.
3. **AOP proxy:** The object created by AOP to implement the aspect contracts. In Spring proxy objects can be **JDK dynamic proxies** or **CGLIB proxies**. By default, the proxy objects are JDK dynamic proxies, and the object being proxied must implement an interface that is also

implemented by the proxy object. But a library like CGLIB can create proxies by subclassing, so an interface is not needed.

# Page # 285 – Date 08-11-2021

By using @EnableAspectJAutoProxy one can enable aspect support , which helps spring to create Proxies for the beans, default spring ONLY creates JDK dynamic proxying which can be alerted to create CGLIB proxies specifying proxyTargetClass attributes which are also know as subclass proxies example - @EnableAspectJAutoProxy(proxyTargetClass = true).

Difference between JDK dynamic proxy and GCLIB proxies.

|  |  |
| --- | --- |
| **JDK Dynamic Proxy** | **GCLIB Proxy** |
| This is suited for the scenarios where the target class implements one or more interfaces. The proxy class will also implement the same interfaces to inject the desired behavior. | This is more suited for the scenarios where the target class doesn’t implement any interface, in that case spring generates a class at the runtime and extend it from the target class to wrapped it with desired behavior. |
| Default when enable @EnableAspectJAutoProxy | One need to explicitly add property proxyTarget = true  @EnableAspectJAutoProxy (proxyTargetClass = true). |

High level steps that need to be performed in order to implements apects in spring

1. Aspect related dependencies needs to be included
2. Configuration calls containing advice needs to be annotated with @Aspect and also declare it as a bean using (@Compoent OR @Bean)
3. Advice method needs to be annotated with one of the many advice annotations.
4. Finally, one needs to add @EnableAspectJAutoProxy to configuration class for enabling aspectJ.

Aspect is available prior to spring 3.0 version, where one needs to enable to aspectJ configuration using XML annotation. AspectJ has its own namespace which has to be used to enabling the annotation.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns=”<http://www.springframework.org/schema/beans>” xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance **xmlns:aop="http://www.springframework.org/schema/aop** xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/aop  http://www.springframework.org/schema/aop/spring-aop.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context.xsd">  <!-- Configuration for the aspects that apply to the application -->  <bean id="personRepoMonitor" class="com.apress.cems.aop.PersonMonitor"/>  <context:component-scan base-package="com.apress.cems.aop" />  **<!-- Configure Aspect support -->**  **<aop:aspectj-autoproxy>**  **<aop:include name="personRepoMonitor"/>**  **</aop:aspectj-autoproxy>**  **<!-- Configure advice -->**  **<aop:config>**  **<aop:aspect ref="personRepoMonitor">**  **<aop:before pointcut="execution(public \* com.apress.cems.repos.\*.JdbcPersonRepo+.findById(..))" method="beforeFindById"/>**  **</aop:aspect>**  **</aop:config>**  </beans> |

**PointCuts**

Pointcuts expression – Template

|  |
| --- |
| **execution([Modifiers] [ReturnType] [FullClassName].[MethodName]([Arguments]) throws [ExceptionType])** |

*The expression can contain wildcards like + and \* and can be made of multiple expressions concatenated by boolean operators such as &&, ||, and so forth. The \* wildcard replaces any group of characters when used to match pieces of package names, classes, and methods, and a single character when used to match method parameters. The + wildcard specifies that the method to advise can also be found in subclasses identified by [FullClassName] criteria. The + wildcard works in a similar way when the criteria used is an interface and the pointcut expression matches the methods in all*

*implementations.*

* *The* ***[ReturnType] is mandatory****. If the return type is not a criterion, just use \*. If it is missing the application crashes at boot time throwing an java.lang.IllegalArgumentException with a message explaining that the pointcut is not well-formed.*
* *The* ***[Modifers] is not mandatory*** *and if not specified defaults to public.*
* *The [****MethodName] is not mandatory****, meaning no exception will be thrown at boot time. But if unspecified, the join point where to execute the advice won’t be identified. It’s safe to say that if you want to define a technically useful pointcut expression you need to specify it.*
* *The* ***[Arguments] is mandatory.*** *If it is missing the application crashes at boot time throwing a java.lang.IllegalArgumentException with a message explaining that the pointcut is not well formed. If the arguments are not a criterion, just use (..) which matches a method with 0 or many arguments. If you want the match to be done on a method with no arguments, use (). If you want the match to be done on a method with a single argument, use (\*).*

***For the ease of maintainability and reusability, the point cut expression can be split into multiple smaller expression OR even can be externalized completely.***

***NamePointCuts – single complex pointcut expression can be split into two simple pointcut expression – name of the method then can be used to identify these pointcuts. That’s why they are called a namePointCuts.***

***Example:***

***//NAME POINT-CUT repoFind()***

@Pointcut ("execution(\* com.apress.cems.\*.\*PersonRepo+.findBy\*(..))")

public void repoFind() { }

***//NAME POINT-CUT serviceFind ()***

@Pointcut ("execution (\* com.apress.cems.aop.service.\*Service+.findBy\*(..)))")

public void serviceFind() { }

***//Composite pointcut that refer serviceFind() and repoFind() named pointCuts.***

@Before("repoFind() || serviceFind()")

public void beforeFind(JoinPoint joinPoint) {

..

}

*NOTE : the name pointCuts can be move to different package to make it completely decoupled from the composite pointCuts.*

*Spring PointCut AOP Designator*

*The args() designator is used to identify methods with a parameter configuration defined by it.*

*Limitation of Spring AOP*

* *ONLY public Join Point can be advised.*
* *Aspect can only be applied to springBean*
* *Even if the spring AOP is not set to use CGLIB , if the target class is not implementing any interface , spring will try to use CGLIB proxy instead of JDK dynamic proxy.*
* *If there are two methods one calling another , pointcut expression matching both the method then the advice will be called only for the first method NOT for the second method.*

*WHO write writes the method for repository custom interfaces?*

*Spring defined an aspect that have a pointcut which executes every time when findBy\* method is called, using spring AOP spring provide the necessary method body for those repository methods.*

**Chapter 4 – Spring Data Access** ­­­

Basic way to access a data base is using JDBC connection, but using this approach developer needs to managed their own connection and handle JDBC objects. JDBC is redundant usages, prone to error, poor exception handling and cumbersome to use.

Memory leak – object still utilizing memory long after its needed or accessible.

**Spring JDBC template**

Spring JDBC framework consist of four major packages, and a small config package that house the infrastructure bean & utility classes for embedded databases.

* **Core**: Spring JDBCTeamplate class of the core package hides major boilerplate code and unburden developers from managing connections.
* **Data source**: The Datasource package contains the utility classes for datasource management
* **Object**: Object package contains the RDBMS quries , update , store procedures as thread-safe , reusable objects.
* **Support**: provides feature for SQLException translation functionality and some utility classes.

JDBC template is designed over template design pattern which is a behavior design pattern category.

Every time a JBDC template method is called, a new connection is automatically opened and close one the execution is complete. JDBC template is perfect for small project where one needs to get rid of connection management using JDBC connectors.

JDBCTemplate method queryForObject was change to specific native dataTypes like queryForInt, queryForLong,QueryForString

JdbcTemplate method queryForMap method return pairs of columns [column\_name(string), column\_value(object)]

The RowMapper object is stateless. RowMapper is a functional interface, one can implement a lambda function instead, there is no harm in defining one row-mapper per interface. RowMapper is not required if ORM products are used .

|  |
| --- |
| RowMapper<Person> personRowMapper = (rs,rowInt) -> {  return  Person.Builder  .setPersonName(rs.getString(“PERSON-NAME”))  .setPersonAge(rs.getInt(“AGE”)).build();  } |

**Things to know about JdbcTemplate**

* JdbcTemplate works with queries that specify parameters using the '?' placeholder.
* Use queryForObject when it is expected that execution of the query will return a single result.
* Use RowMapper<T> when each row of the ResultSet maps to a domain object.
* Use RowCallbackHandler when no value should be returned.
* Use ResultSetExtractor<T> when multiple rows, or multiple records from different tables returned in a ResultSet map to a single object.
* DDL – Data Definition Language are database operations that manipulate database objects, such as tables, views, cursors, and so forth.
* DML – Data Manipulation Language, the SELECT, INSERT, UPDATE, and DELETE commands are database statements to create, update, or delete data from existing tables.

JdbcTemplate can execute both DML and DDL statements. However, its not advisable to use jdbcTemplate for running DDL as there is a risk of sqlInjection.

jdbcTemplate.execute(“CREATE TABLE DUMMY\_TABLE”).

NOTE here table name cannot be provided a parameter as , its expected that parameter should be part of an expression whereas the table name is not part of any expression – if any one attempt to do so, it will throw JdbcSQLSyntaxErrorException

jdbcTemplate.execute(“select \* from DUMMY\_TABLE”,Integer.class);

**Exception handling in accessing data.**

NOTE: the checked exception is the once that needs to handle within the code or throws up to call method hierarchy, otherwise it will throw a compile time error. Whereas, the unchecked exception is once which are not checked by the compiler – even if they are NOT handle will not throw any error.

In pure JDBC same type of checked excetion java.sql.SQLException is thrown, which needs to be handle or thrown throw out the calling method hierarchy making in tightly coupled. In case of spring , all database exception are unchecked exception and are extend from RuntimeException class. Developer need not to handle the exception making it more loosely coupled.

Spring data base transaction are of three major types

* Exception those are consider as **non-transient** exception – these exceptions are not recoverable unless originating cause is fixed. The are extend from org.springframework.dao.

NonTransientDataAccessException abstract class.

* **Recoverable exception** – these exception can be recovered if certain recovery steps are performed extend from org.springframework.dao.RecoverableDataAccessException
* The third type is **transient** exception – this are those exception which can be recovered jut by retrying. These exceptions are extended from org.springframework.dao.TransientDataAccessException abstract class.

Apart from these three major exception categories there is another category which is cause due to wrong initialization of database or failure of initialization scripts.

**Data base transactions.**

For managing a database transaction, a transactions environment is required, spring usages an infrastructure bean called as transaction manager to manage the transactions – when moving from one environment to another only these bean needs to be change. There are basically four flavor of transaction manager environment available.

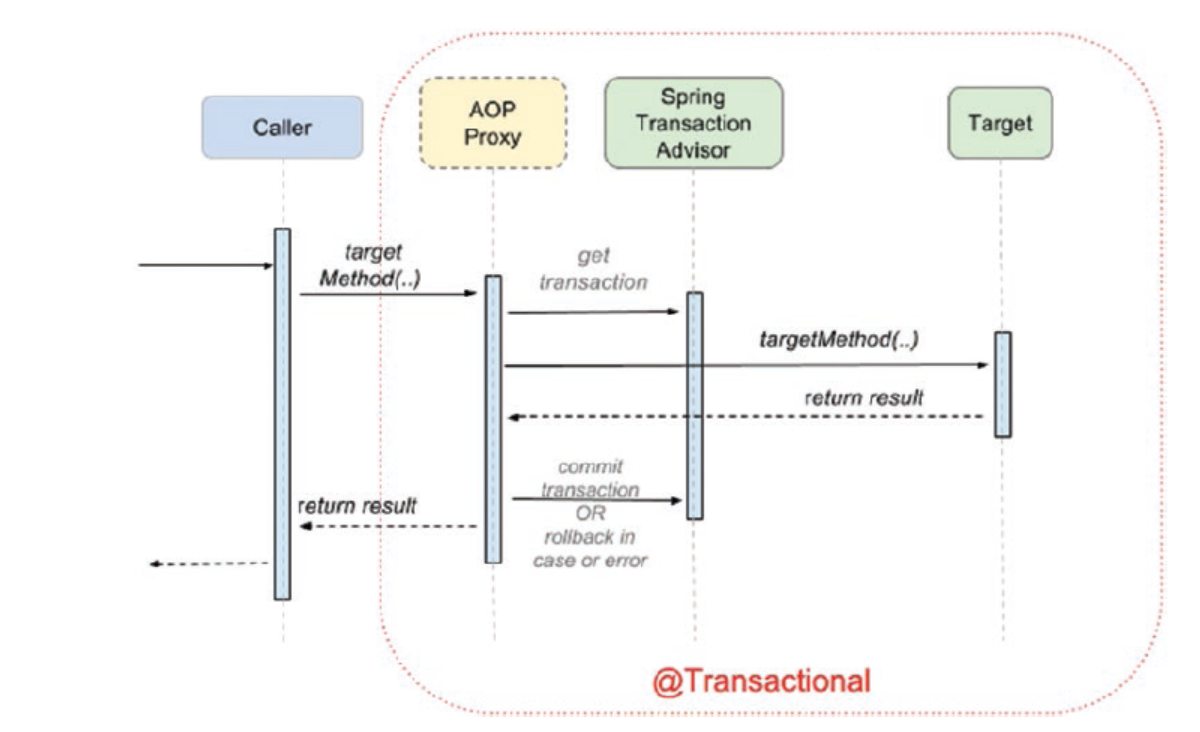
* **JDBC Spring environment:** Type org.springframework.jdbc.datasource.

DataSourceTransactionManage.

* **Hibernate Spring environment:** Type org.springframework.orm. hibernate5. HibernateTransactionManager, a Spring-specific implementation that uses a hibernate session object created by an infrastructure bean of type org.springframework.orm.hibernate5.LocalSessionFactoryBean to manage entities in a transactional context.
* **JPA environment**: a bean of type org.springframework.orm.jpa.JpaTransactionManager, a Spring-specific implementation that uses an entity manager object created by an infrastructure bean of type org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean to manage entities in a transactional context.
* **Enterprise JTA environment**: setup requires an application server that will configure and provide a datasource bean using JNDI. JNDI works with other technologies on the Java platform, such as the Enterprise Edition (Java EE), to organize and locate components in a distributed computing environment. Spring loads a bean of type extending org.springframework.transaction. jta.JtaTransactionManager specific to the application server used. This transaction manager is appropriate for handling distributed transactions, which are transactions that span multiple resources, and for controlling transactions on application server resources.

In spring the transaction is handled through AOP as it’s a cross cutting concerns, AOP framework provide an around advice wrapping the original bean methods which executed in a transaction, retrieving or opening a transaction before execution and committing

it afterward is necessary.



The AOP proxies use two infrastructure beans for this: an org.springframework.transaction.interceptor.TransactionInterceptor in conjunction with an implementation of org.springframework.transaction.PlatformTransactionManager. Spring provides a flexible and powerful abstraction layer for transaction management support.

In Spring any method that is annotated with @Transactions is executed as single unit of transaction. For AOP annotation to work, the method should be public otherwise the transaction cannot be managed.

The simplest way to configure a transaction manager is to create a bean of type PlatformTransactionManager and add a class level annotation @EnableTransactionManagement.

The @EnableTransactionManagement is more flexible; it looks for a bean of any type

that implements the org.springframework.transaction.PlatformTransactionManager,

so, the name is not important. In case the default transaction manager bean must

be established without a doubt, this can be done by making the configuration class.

In case of bigger application where it needs more than one datasource there needs to have multiple transaction manger, if in the @Transaction annotation the transaction manger to be used is not defined then the spring application would start successfully but fails when the transactional method is called, as spring wouldn’t check transaction manager beans until transactional method is called. When it checks it would find two bean of Type PlatformTransactionManager, and spring cannot decide on its own which one to us. This problem can be overcome by specifying transaction manager to be use in @Transactional(transactionManager = "simpleManager", readOnly = true). Alternatively one can also used @Primary to ensure spring chose the right transaction manager OR make sure the configuration class implements TransactionManagementConfigurer interface which has a single method annotationDrivenTransactionManager() that specifies default transaction.

The flowing are the transaction propagation behaviors, one of these behaviors can be selected while executing an existing transaction OR creating a new transaction.

| **Propagation** | **Description** |
| --- | --- |
| REQUIRED: | An existing transaction will be used or a new  one will be created to execute the method annotated with @  Transactional(propagation = Propagation.REQUIRED). |
| REQUIRES\_NEW | A new transaction is created to execute the  method annotated with @Transactional(propagation =  Propagation.REQUIRES\_NEW). If a current transaction exists, it will be suspended. |
| NESTED: | An existing nested transaction executes the method  annotated with @Transactional(propagation = Propagation. NESTED). If no such transaction exists, it will be created. This approach is quite similar to REQUIRED, so if the datasource supports I think it should be mandatory to use it because it reuses existent resources. |
| MANDATORY: | An existing transaction must be used to execute  the method annotated with @Transactional(propagation =  MANDATORY). If there is no transaction to be used, an exception will be thrown. |
| NEVER: | Methods annotated with @Transactional(propagation =  Propagation.NEVER must not be executed within a transaction. If  a transaction exists, an exception will be thrown. |
| NOT\_SUPPORTED | No transaction executes the method annotated  with @Transactional(propagation = Propagation.NOT\_  SUPPORTED). If a transaction exists, it will be suspended. |
| SUPPORTS: | An existing transaction executes the method annotated  with @Transactional(propagation = Propagation.SUPPORTS).  If no transaction exists, the method will be executed anyway,without a transactional context. |

In general, the transactions should be isolated, one transaction should NOT access data from other transactions, however in DBMS there are four level of isolation provided, spring supports five different level of isolations.

|  |  |
| --- | --- |
| **Isolation** | **Description** |
| DEFAULT | The default isolation level of the DBMS. |
| READ\_UNCOMMITED | Data changed by a transaction can be read by a different transaction while the first one is not yet committed, also known as *dirty reads*. Dirty reads are possible at this isolation level. |
| READ\_COMMITTED | Dirty reads are not possible when a transaction is used with this isolation level. This is the default strategy for most databases. But a different phenomenon could happen here: *non-repeatable read*: when the same query is executed multiple times, different results might be obtained. (For example, a person is extracted repeatedly within the same transaction. In parallel, a different transaction edits the person and commits. If the first transaction has this isolation level, it will return the person with the new properties after the second transaction is committed.) |
| REPEATABLE\_READ | This level of isolation does not allow  dirty reads, and repeatedly querying a table row in the same transaction will always return the same result, even if a different transaction has changed the data and committed while the reading occurs. The process of reading the same row multiple times in the context of a transaction and always getting the same  result is called *repeatable read*. But, at this level, *phantom reads* are still possible. A *phantom read* happens when in the course of a transaction, the execution of identical queries leads to different result sets returned. |
| SERIALIZABLE | This is the most restrictive isolation level, since transaction are executed in a serialized way. So no dirty reads, no repeatable reads, and no phantom reads are possible. |

**Timeout** – this is one of the attributes of the @Transation annotation, default transaction time out is set by the value set within the transaction Manager object. This value can be override by setting a timeout value as attribute to the @Transaction annotation. Default value for the timeout attribute is set to 1 which means timeout not supported, other integer value can be set equivalent to the timeout in millisecond.

**rollbackFor attributes** – this @Transaction annotation attribute defines one or more exception or Throwable subclass , by default transaction is ONLY rollback when RuntimeException is thrown.

**noRollbackFor attribute** – this @Tansaction annotation attribute defines one or more exception for Throwable subclasses, with this one can also define runtime Exception as non rollback exception.

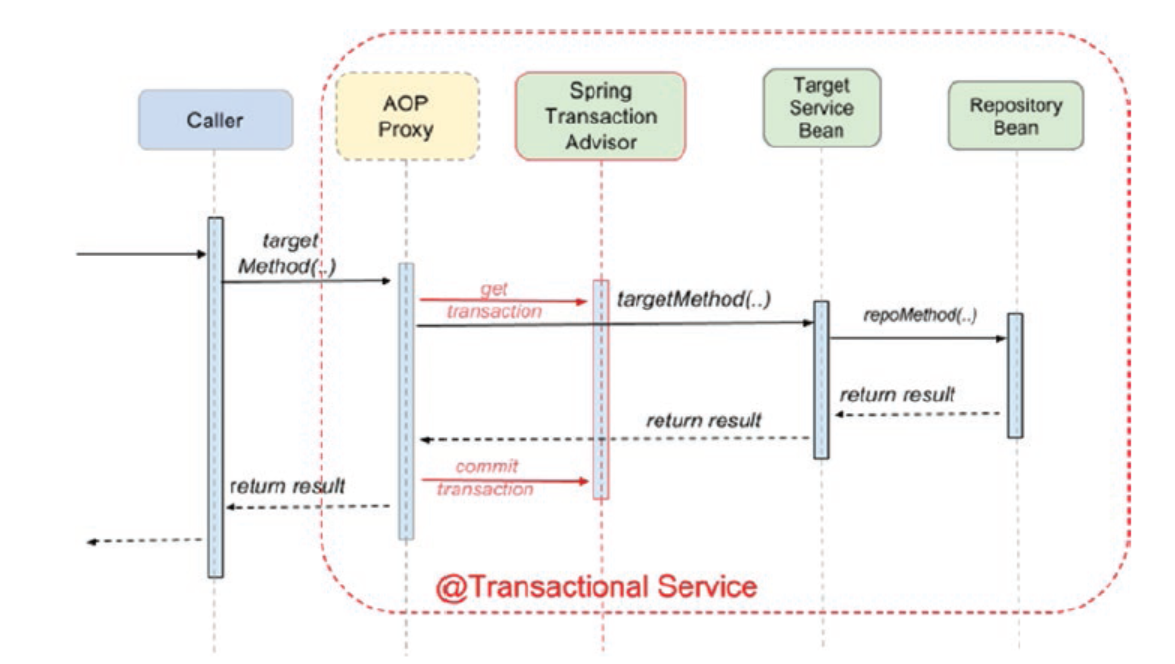
**NOTE: @Transcation can be used at the class level too, in that case all the methods defined in the class will inherited the same transactional behavior.**

Transaction is an cross cutting concerns and its been implemented in spring using AOP, the default behavior is to create an interface based proxy this behavior can be change by setting property proxyTargetClass = true of @EnableTransactionManager annotation.

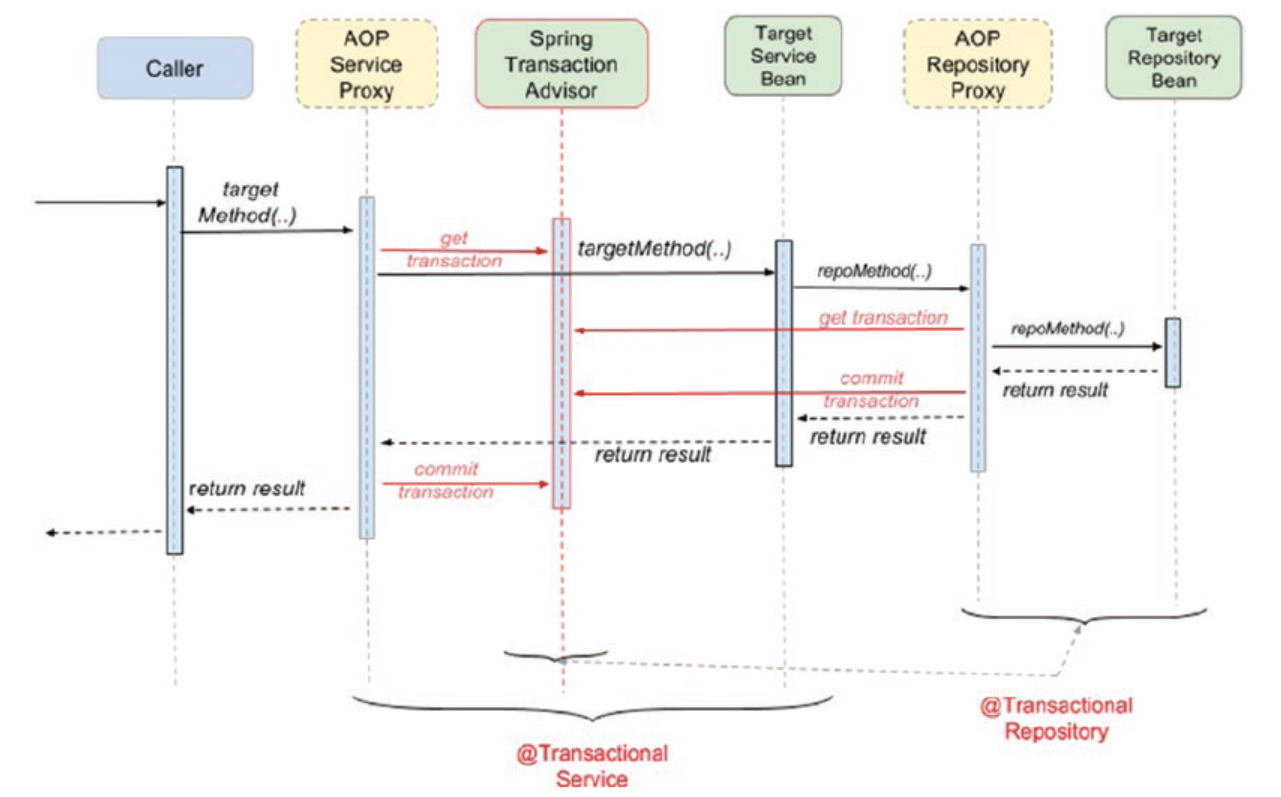
Its important that @Transaction annotation is to be applied at the public method level. (AOP cannot.

If @Transaction needs to be put for Service classes OR for the repository classes?

Spring usages decorative design pattern to wrap a target Bean – when a Service method is annotated with the @Transaction it creates a wrapper around the service target class object where service class method is executed with the accrued transaction context. If there are multiple repository methods are called within the same service class target method then all the methods are called within the same transaction context. Setting the @Transaction annotation only to the repository alone will not be suffices if the service method calls methods from different repository classes within some method, then they can’t be executed within a same transaction context.



Setting @Transaction annotation to both service and repository classes is redundant as the APO would be creating two wrapper one for service and other for repository classes and its needs to propagate the transaction from service to repository.



NOTE Use @Transactional in the service layer or the DAO/repository layer, but not both. The service layer is the usual choice, because service methods call multiple repository methods that need to be executed in the same transaction. The only reason to make your repositories transactional is if you do not need a service layer at all, which is usually the case for small educational applications.

Spring also provide way to handle the transaction proamativally where developer can manage their transactions – for that spring does provide TransactionTemplate , but the developer handle the transactions on their own.

|  |
| --- |
| public class ProgramaticDetectiveService implements DetectiveService  {  private DetectiveRepo detectiveRepo;  private TransactionTemplate txTemplate;  public ProgramaticDetectiveService(DetectiveRepo detectiveRepo,  PlatformTransactionManager transactionManager) {  this.detectiveRepo = detectiveRepo;  this.txTemplate = new TransactionTemplate(transactionManager);  }  @Override  public Optional<Detective> findById(Long id) {  return txTemplate.execute(status -> {  Optional<Detective> opt = null;  try {  opt = detectiveRepo.findById(id);  } catch (Exception e) {  status.setRollbackOnly();  }  return opt;  });  }  } |

State.SetRollbackOnly() method needs to be called when the transactions need to be rollbacked.

NOTE: Distributed transactions: The distributed transaction are those transactions that span across multiple environments – to manage this type of transactions one needs to use JTA java transaction API and specific XA drivers.

**Hibernate ORM (Object Relational Mapping) framework**

There are number of ORM projects available like Apache OpenJPA , Hibernate , EclipesLink etc. Hibernate ORM is an object relational mapping framework that provides support for mapping an object-oriented domain model to a relational database.

**Hibernate Properties**

* Hibernate.Dialect, this shou be matching with the data base which is currently been used.
* Hibernate.hbm2ddl, this represents what application should do when it starts up the following are the option available
  + Create-only
  + Drop
  + Create
  + Create-drop
  + Validate
  + Update
* hibernate.format\_sql Boolean value, if set to true generated sql statements will be make human readable on the console.
* Hibernate.show\_sql Boolean value, if this values to set as true then the generated sql will be displayed on the console.
* Hibernate.use\_sql\_comments Boolean value, if this value is set to true than hibernate will insert comments to generated SQL for developer to understand better.

Benefit of using Hibernate ORM

Spring Data JPA

Spring picks up the type interface and implements them at the runtime to create a bean.

There are various repositories available,

NOTE : if spring-boot-starter-data-jpa module is on the classpath, there is no need to add @EnableJpaRepositories or @EnableTransactionManagement on a configuration class.

**Chapter 5 – Spring Web**

­­­

* Typically, a 3-tier application consist of DAO layer, Service layer and Presentation layer.
* Spring Webflow and Spring MVC can be used in the presentation layer.
* DispatcherServlet is the main component of the Spring MVC
* The DispatcherServlet delegates to a ViewResolver to map logical view names to view implementations.
* The core view resolver provided by Spring is **InternalResourceViewResolver** and is the default view resolver.
* **How Spring MVC works?**

1. The **DispatcherServlet** identified the handler method for the persons/list GET request and requests the PersonController method to handle the request.
2. The handler method is executed, and a logical view named persons/list and a model object containing the collection named persons are returned to the **DispatcherServlet**. The

model={persons} is a notation that means the *persons* object is contained by the *model* object.

1. The **DispatcherServlet** asks the **InternalResourceViewResolver** to which view template the persons/list logical view name corresponds.
2. The **InternalResourceViewResolver** that is configured as follows, it takes the logical view name, applies the prefix (/WEB-INF/) and the suffix(.jsp) and sends the result (/WEB-INF/persons/list.jsp), which is the physical path of the view template to the DispatcherServlet, which will further use the view template and the model object received from the controller to create a view.

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/"/>

<property name="suffix" value=".jsp"/>

</bean>

1. The View interface from the org.springframework.web.servlet package is the core MVC component for web interactions. Implementations of this interface are responsible for rendering data and exposing model contents. There are various implementations available and the most renown are the ones that are JSP based because of the tag libraries that can make the views dynamic based on the model data. Views are beans instantiated by the ViewResolver. The InternalResourceViewResolver configured previously is a view resolver for JSP pages and creates beans of type InternalResourceView to render JSP pages. If the JSTL tag library is found in the classpath it creates beans of type JstlView that expose JSTL-specific request attributes specifying locale. Both view implementations are part of the org.springframework.web.servlet.view package.
2. The DispatcherServlet will further use the view template provided by the InternalResourceViewResolver and the model object received from the PersonController to create a view.

There can be multiple viewResolver, where each view resolvers are chained using their priority. This technique is also known as resolver chaining. This option does not work, if there is no single view associated with each resource – based on the request if client would like to request for different viewResolver for a same resource then the chain resolver doesn’t work. In that case, content negotiation works best. This was made available with Spring 3.0 version, where ContentNegotiatingViewResolver bean is added that would delegate the task to view resolver implementation define in the application configuration, selecting the view type matching the content type header value in the client request. There are three specific strategies for client to request for a view from a server

* **Using distinct URL** like , <http://localhost:8080/viewresolver/person.xls> for excel view, <http://localhost:8080/viewresolver/person.pdf> for pdf view.
* **Using same URL with request header with desired value for requested view.**
* **Parameterized URL**, with a query parameter <http://localhost:8080/viewresolver/person?view=excel> for excel view and <http://localhost:8080/viewresolver/person?view=pdf> for pdf view.

Starting Spring3.0 onwards web.xml file is not needed, it can be replace by a class implementing springframework.web.SpringServletContainerInitializer OR any extends any spring class that is implementing SpringServletContainerInitializer.

The recent version of the embedded tomcat is 9.0.22 which is written using JDK 8 , using spring with JDK11 along with current embedded version is unpredictable – best solution is to deploy spring war to standalone tomcat server.

Exception handling in Spring: in Spring following type of exception resolver are already added

* ExceptionHandlerExceptionResolver resolves exceptions by invoking methods annotated with @ExceptionHandler found within a controller or a class annotated with @ControllerAdvice.
* ResponseStatusExceptioResolver: ResponseStatusExceptionResolver resolves methods annotated with @ResponseStatus and maps them to the status code configured using this annotation.
* DefaultHandlerExceptionResolver resolves exceptions raised by Spring MVC and maps them to HTTP status codes

The MockMvc class is the core component needed to test Spring MVC applications. The MockMvcBuilders is a utility class used to instantiate the mockMvc object, and provides two methods to create thinned web application contexts used for test execution. In the previous example, the one that receives a WebApplicationContext as argument is used. If we intend to write an integration test where no mocks are needed, the @AutoConfigureMockMvc can be used with @SpringBootTest to bootstrap the full application context. When this annotation is used, the mockMvc object is created and configured for you automatically. You just need to autowire it in the test class to send requests to the application.

When running the spring test in parallel, using a single port can be difficult as the tests would be conflicting, for that reason one can have SpringBootTest.WebEnvironment.RANDOM\_PORT assigned a random port assigned and the value of that port would be added to @LocalServerPort annotation, which can then be used for testing by setting port() method

**Chapter 6 – Spring Security**

­­­

Web application security can be grouped under

* Authentication
* Authorization web requests
* Authorizing method calls
* Authorizing access to individual domain object

When securing an application following concepts are very important

1. Principal: This is the user, system or a device that that can perform an action within a system.
2. Credentials: these are the identification key that a principal user to confirm its identity.
3. Authentication: is the process of verifying the validity of a principal’s credentials.
4. Authorization: is a process of verifying if a principal is authorized to perform request action on the application.
5. Secure Item is the term used to describe a application resources that is secured.

Spring is popular because it helps, its modules are completely decoupled. One can change the modules are swap then with other modules to meet specific requirements. Spring implements security as cross cutting concerns, it implemented using AOP (aspect-oriented programming).

In-order to implement security in spring one need to perform the following

1. Declare a security filter
2. Define a spring security context
3. Configure Authentication and Authorization

In spring the security is implemented by chain of security beans , delegatingFilterProxy bean delegates the calls to a list of chained security filter beans and acts as an interceptor for secured requests. These chained proxy filters have the following responsibilities , these filter are replaceable at runtime.

1. Driving authentication
2. Enforcing authorization
3. Managing logouts
4. Maintaining security context in httpSession

**What is the effect of annotating a method with @Secured("ROLE\_ADMIN")?**

in a Spring secured application, where @EnableGlobalMethodSecurity(securedEnabled = true) is used on a configuration class, it causes the class containing the method to be wrapped in a secure proxy to restrict access only to users with the ROLE\_ADMIN

**Chapter 7 – Spring REST** ­­­

Spring Remote procedure call duplicated, as its difficult to operate through firewall.

Spring webservice – supports both REST and SOAP style webservice.

Spring platform is most suited for building stateless REST webservice – where REST service allows access and manipulation of textual representation of web resources using a uniform and predefined set of stateless operation.

|  |  |  |
| --- | --- | --- |
| **HTTP Method Name** | **Purpose** | **Description** |
| GET | Read | Reads a resource, does not change it. It is considered safe. Reading  the same resource always returns the same result. It is considered  idempotent. This does not apply when the resource is not cached and  modified by a parallel operation on the server that it is being retrieved  from. |
| POST | Create | Used to create a new resource. Neither safe nor idempotent. Two  identical POST requests will result in two identical resources being  created or errors at the application level. |
| PUT | Update | Most often used for update capabilities. It is not safe, as it modifies the  state on the server, but is idempotent (unless subsequent calls of the  same PUT request increments a counter within the resource for example). |
| DELETE | Delete | Used to delete resources. Not safe, but can be considered idempotent. Because requests to delete a resource that no longer exists will always  return a 404 (not found). |

When working with REST – the session need not to be stored as REST services are stateless in nature.

@Controller is not equivalent to use @RestController

Based on the existing message convertor on the classpath , spring would facilitate the conversion of incoming message to desired format.

Validation , @Validation annotation can be used to validate the data that is sent by the client based on the validation rule added to the Domain Object.

SO if we would actually intend to create a new Person instance and set the password for it, using this annotation and configuring its access attribute to Access. WRITE\_ONLY we can specify that this property is to be set only when de-serializing an object. defaults to Access. AUTO, which means visibility rules automatically determine read and/or write-access of this property, and since in the previous example there are no visibility rules defined, the file will be available at serialization and deserialization.

The scope for a controller advice can be limited to exceptions thrown by methods of controllers declared in a specific set of packages (or subpackages) by setting the basePackages attribute.

Error Handling

|  |  |  |
| --- | --- | --- |
| **Method** | **Scenario** | **HTTP Status Code** |
| GET |  |  |
| POST |  |  |
| PUT |  |  |
| DELETE | Deleted Item does not exist | HTTP 404 |
|  | Deletion couldn’t execute successfully | HTTP 500 |