`

Chapter 2 – Notes

SpringFramework benefit

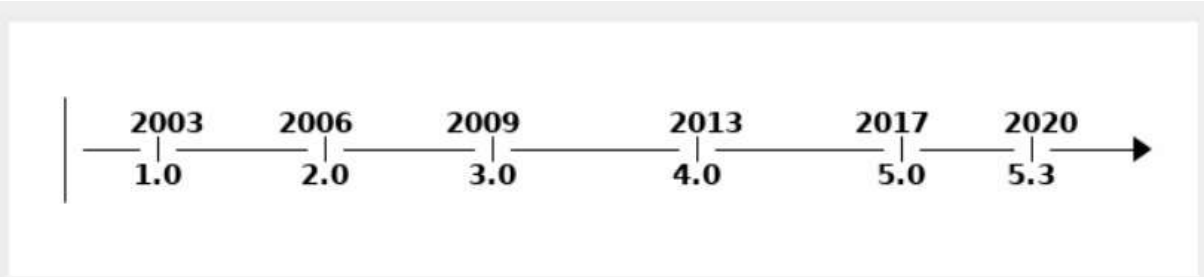
- easy to develop
- easy to test
- de-coupled

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.

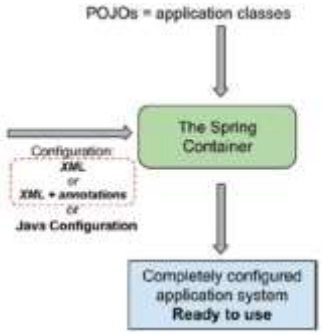
Java is an object-oriented programming language – it supports inheritance, use of generic helps it to achieve high level of inheritance.

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 – spring framework 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 Spring Framework put together the application after been complied – this gives more flexibility as the functionality of the application can be change by the external authority (spring Framework) 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?

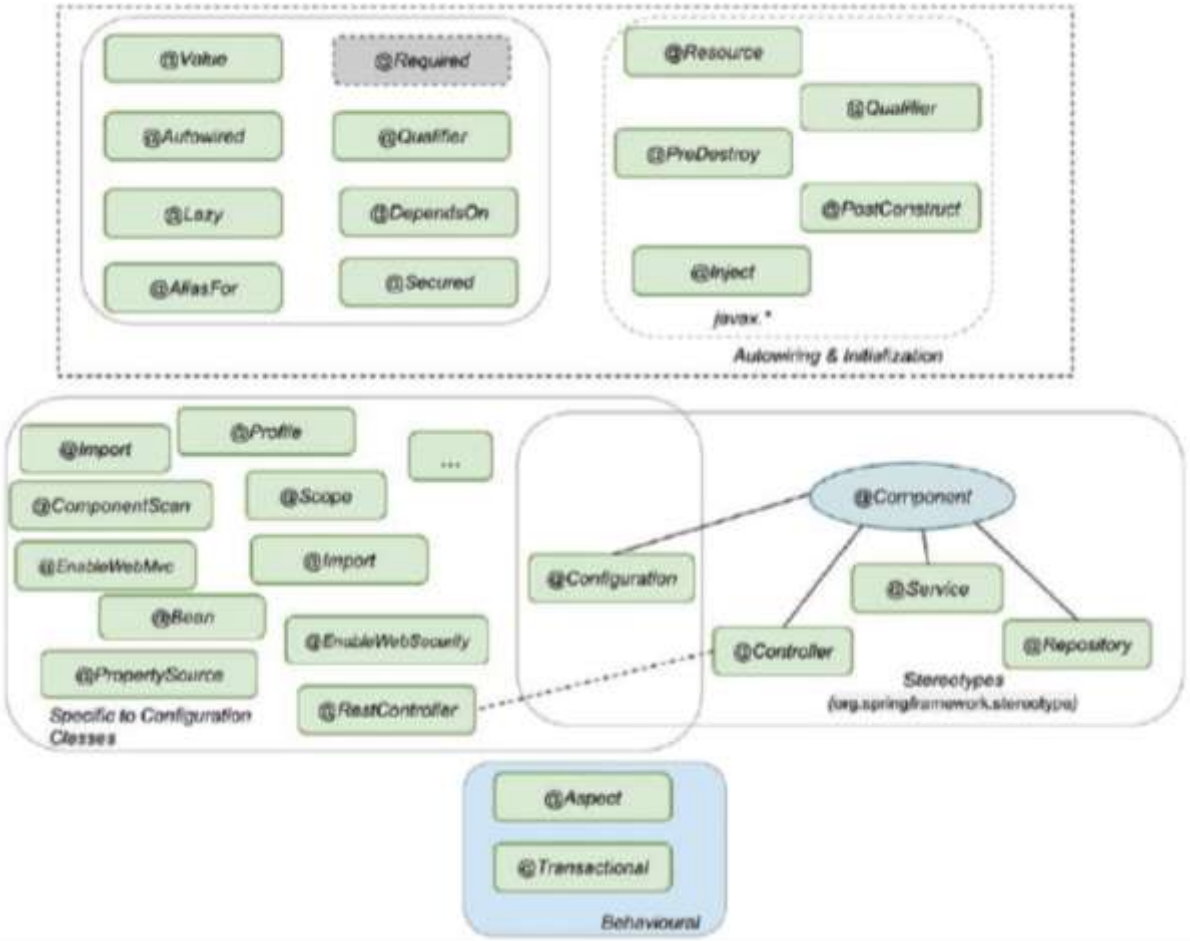
Best Practice not to use schema version on the namespaces.
Classpath is a common prefix used to refer xml configuration file(s), with classpath as prefix the spring container will look for the configuration file under src/main/resource directory.

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.*

`

Annotation	Descriptions
@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	?? Need to do more research on this.
@Profile	
@Properties	

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 configure 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.

@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();
}
```

`


```
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.*

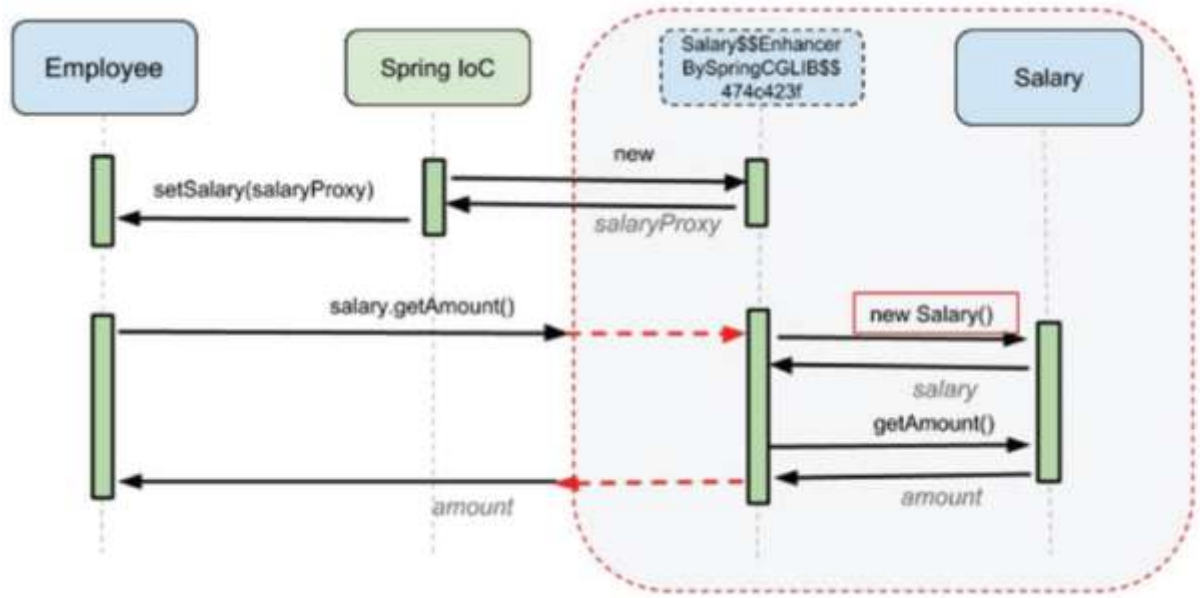
Proxy pattern

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.

`

@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
    a. 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.
    b. The **bean dependency injected,** if the bean dependency is injected using field injection or setter method injection then the required bean dependency is injected.
    c. Bean is **fully initialized**.
    d. 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


TRACE o.s.c.a.CommonAnnotationBeanPostProcessor - Registered destroy method on class [com.apress.cems.fun.FunBean]:
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();
}
}
```

`

Page # 137 03-11-2021

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.

Table 2-3. *Autowiring Annotations*

Spring	JSR	Comment
`@Component`	`@Named`	`@Named` can be used instead of all stereotype annotations except `@Configuration`
`@Qualifier`	`@Qualifier`	JSR `Qualifier` is a marker annotation used to identify qualifier annotations, like `@Named`, for example
`@Autowired`	`@Inject`	`@Inject` may apply to static as well as instance members
`@Autowired` + `@Qualifier`	`@Resource(name= "beanName")`	`@Resource` is useful because replaces two annotations.

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]