Contents

[0.1 Configuration 2](#_Toc40723164)

[**1.0 Spring Framework?** 2](#_Toc40723165)

[**1.1 Features of Spring Framework?** 3](#_Toc40723166)

[**1.2 Dependency Injection?** 3](#_Toc40723167)

[**1.3 Types of IOC containers?** 4](#_Toc40723168)

[1.4 **Difference between BeanFactory and ApplicationContext are following:** 4](#_Toc40723169)

[**1.5 Spring Beans?** 6](#_Toc40723170)

[1.5.1 Configuration metadata? 9](#_Toc40723171)

[1.5.2 Bean scopes? 10](#_Toc40723172)

[1.5.3 Explain Scoped Proxy in Spring 11](#_Toc40723173)

[1.5.4 Define Bean Wiring? When beans are combined within the Spring container, it’s called wiring or bean wiring. 12](#_Toc40723174)

[1.5.5 Auto wiring and name the different modes of it? 12](#_Toc40723175)

[**2.0 Annotation** 13](#_Toc40723176)

[**2.1 limitations with auto wiring?** 13](#_Toc40723177)

[**2.2 How annotation wiring can be turned on in Spring** 13](#_Toc40723178)

[**What’s the difference between @Component, @Controller, @Repository & @Service annotations in Spring?** 13](#_Toc40723179)

[**2.3 What do you understand by @Required annotation** 14](#_Toc40723180)

[**2.4 What do you understand by @Autowired annotation** 14](#_Toc40723181)

[**2.5 What do you understand by @Autowired annotation** 14](#_Toc40723182)

[**2.6 What do you understand by @RequestMapping annotation** 14](#_Toc40723183)

[3.0 Spring Dao 15](#_Toc40723184)

[**3.1 Describe Spring DAO support** 15](#_Toc40723185)

[**3**.2 Which classes are present in spring JDBC API? 15](#_Toc40723186)

[**3.3 What are the ways by which Hibernate can be accessed using Spring** 15](#_Toc40723187)

[**3.4 Name the types of transaction management that Spring supports** 15](#_Toc40723188)

[**3.5 What are the different ORM’s supported by Spring?** 15](#_Toc40723189)

[4.0 Describe AOP 15](#_Toc40723190)

[4.1 What is Aspect? 16](#_Toc40723191)

[4.2 What is Joint point and Point cut? 17](#_Toc40723192)

[4.3 What is Weaving? 17](#_Toc40723193)

[4.4 Concern and cross-cutting concern in Spring AOP? 17](#_Toc40723194)

[5.0 Spring MVC framework 18](#_Toc40723195)

[5.1 Describe DispatcherServlet 18](#_Toc40723196)

[5.2 Explain WebApplicationContext 18](#_Toc40723197)

[5.3 In Spring MVC framework, what is controller 18](#_Toc40723198)

[5.4 Web App-Servlet 18](#_Toc40723199)

[5.5 Spring MVC Web-App XML Configuration. 19](#_Toc40723200)

[5.6 Spring MVC Web-App Java Configuration? 21](#_Toc40723201)

[6 Spring Boot -> Web App Java Configuration 23](#_Toc40723202)

[**6.1 Introduction** 23](#_Toc40723203)

[6.1.1 Default Package Structure 23](#_Toc40723204)

[6.1.2 Application.properties 24](#_Toc40723205)

[6.2 Hal Browser and Actuator 24](#_Toc40723206)

[Spring Boot Rest Api 25](#_Toc40723207)

[6.3 Spring Boot MVC 25](#_Toc40723208)

[6.3.1 Testing 26](#_Toc40723209)

[6.3.2 Application-Start-Up 27](#_Toc40723210)

[**6.4 Exclude any package without using the basePackages filter?** 27](#_Toc40723211)

[**6.5 What is the Spring Initializer?** 27](#_Toc40723212)

[**6.6 What is a shutdown in the actuator?** 27](#_Toc40723213)

[**6.7 How can I enable auto reload of my application with Spring Boot?** 28](#_Toc40723214)

[**6.8 RequestMapping and GetMapping?** 28](#_Toc40723215)

[**6.9 What is the need for Profiles?** 28](#_Toc40723216)

[**6.10 What is “Transitively Dependency Resolution Management”** 29](#_Toc40723217)

# 0.1 Configuration

**Maven server : tomcat7:run Goal**

primary = "true"

<properties>

<failOnMissingWebXml>false</failOnMissingWebXml>

</properties>

# **1.0 Spring Framework?**

* Framework for Enterprise application.
* Framework of Framework like Hibernate, MVC etc.
* It is light-weighted and loosely coupled.

## **1.1 Features of Spring Framework?**

* **Lightweight:** Spring is lightweight when it comes to size and transparency.
* **Inversion of control (IOC):** The objects give their dependencies instead of creating or looking for dependent objects. This is called Inversion Of Control.
* **Aspect oriented Programming (AOP):**  cohesive development by separating application business logic from system services.
* **Container:** creates and manages the life cycle and configuration of the application objects.
* **MVC Framework:** Spring Framework’s MVC web application framework is highly configurable.
* **Transaction Management:** Generic abstraction layer for transaction management is provided by the Spring Framework. Spring’s transaction support can be also used in container less environments.
* **JDBC Exception Handling:** The JDBC abstraction layer of the Spring offers an exception hierarchy, which simplifies the error handling strategy.

## **1.2 Dependency Injection?**

* You not have to create your objects but must describe how they should be created.
* You don’t connect your components and services together in the code directly, but describe which services are needed by which components through configuration.
* The IoC container will wire them up together.
* **Constructor Injection.**
* **Setter or Variable Injection.**
* **Interface Injection.**
* **In Spring Framework, only constructor and setter injections are used.**
* Class which Require Dependency Injection. => Setter Based

@Autowired

**private Messages message;**

@Autowired

**public void setMessages(Messages msg) {**

**this.message = msg;**

**}**

* Class which Require Dependency Injection. => Constructor Based
* @Autowired

**private** Messages message;

// constructor

public Communication(Messages msg) {

this.message = msg;

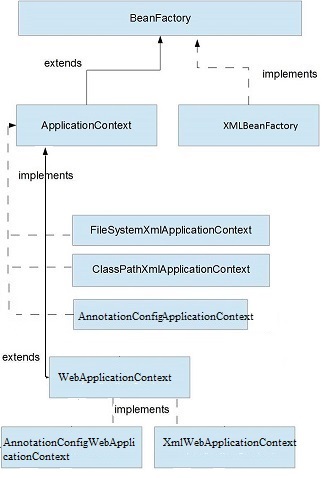
}

* Variable Based and Setter Injection are same and preferred.
* Constructor based Dependency is preferred when dependency is mandatory.

## **1.3 Types of IOC containers?**

* **BeanFactory**: BeanFactory is like a factory class that contains a collection of beans. It instantiates the bean whenever asked for by clients.
* **ApplicationContext**: The ApplicationContext interface is built on top of the BeanFactory interface. It provides some extra functionality on top BeanFactory.

## 1.4 **Difference between BeanFactory and ApplicationContext are following:**

[](https://i.stack.imgur.com/EweA3.jpg)

* **FileSystemXmlApplicationContext** Beans loaded through the full path.
* **ClassPathXmlApplicationContext** Beans loaded through the CLASSPATH
* **XMLWebApplicationContext** and **AnnotationConfigWebApplicationContext** beans loaded through the web application context.
* **AnnotationConfigApplicationContext** Loading Spring beans from Annotation based configuration.

example:

ApplicationContext applicationContext = new AnnotationConfigApplicationContext(BeansConfiguration.class);

* **ApplicationContext** is the container initialized by a **ContextLoaderListener** or **ContextLoaderServlet** defined in a web.xml and ContextLoaderPlugin defined in struts-config.xml.

**Note**: **XmlBeanFactory** is [deprecated](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/xml/XmlBeanFactory.html) as of Spring 3.1 in favor of **DefaultListableBeanFactory** and **XmlBeanDefinitionReader**.

1. **BeanFactory** uses lazy initialization **but** ApplicationContext uses eager initialization. In case of BeanFactory, bean is created when you call getBeans() method, but bean is created upfront in case of ApplicationContext when the ApplicationContext object is created.
2. BeanFactory explicitly provide a resource object using syntax **but** ApplicationContext creates and manages resource objects on its own.
3. BeanFactory doesn’t support internatiolization **but** ApplicationContext supports internationalization.
4. With BeanFactory annotation-based dependency injection is not supported **but** annotation based dependency injection is supported in ApplicationContext.

**Using BeanFactory:**

BeanFactory beanfactory = new XMLBeanFactory(new FileSystemResource("spring.xml"));

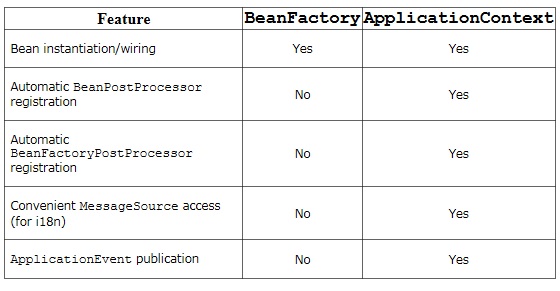
Triangle triangle =(Triangle)beanFactory.getBean("triangle");

**Using ApplicationContext:**

ApplicationContext context = new ClassPathXMLApplicationContext("spring.xml")

Triangle triangle =(Triangle)context.getBean("triangle");

* you are using auto wiring and using **BeanFactory** than you need to register **AutoWiredBeanPostProcessor** using API which you can configure in XML if you are using **ApplicationContext**.
* In summary **BeanFactory** is OK for testing and non-production use but **ApplicationContext** is more feature rich container implementation and should be favored over **BeanFactory**
* **BeanFactory** by default its support **Lazy** loading and **ApplicationContext** by default support **Aggressive** loading

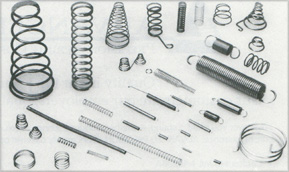


## **1.5 Spring Beans?**

Spring is a lightweight and flexible framework.

@Configuration: Tags the class as a source of bean definitions for the application context.

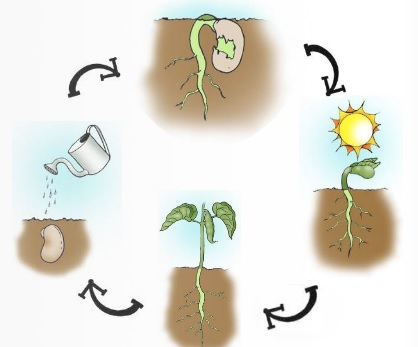
**Analogy:**

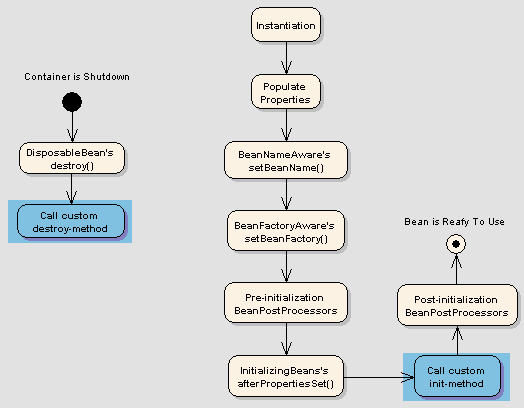


**Bean:** is an object, which is created, managed and destroyed in Spring Container. We can inject an object into the Spring Container through the metadata(either xml or annotation), which is called inversion of control.

**Analogy:** Let us assume farmer is having a farmland cultivating by seeds(or beans). Here, Farmer is Spring Framework, Farmland land is Spring Container, Beans are Spring Beans, Cultivating is Spring Processors.







**Like bean life-cycle, spring beans too having it's own life-cycle.**

[Following is sequence of a bean lifecycle in Spring:](https://intellipaat.com/interview-question/spring-interview-questions/)

* **Instantiate:** First the spring container finds the bean’s definition from the XML file or @Configuration notation and instantiates the bean.
* **Populate properties:** Using the dependency injection, spring populates all of the properties as specified in the bean definition. Dependency to any other class.
* **Set Bean Name**: If the bean implements **BeanNameAware** interface, spring passes the bean’s id to setBeanName() method.
* Set **Bean factory**: If Bean implements **BeanFactoryAware** interface, spring passes the beanfactory to setBeanFactory() method.
* **Pre-Initialization**: Also called post process of bean. If there are any bean **BeanPostProcessors** associated with the bean, Spring calls **postProcesserBeforeInitialization**() method.
* **Initialize beans**: If the bean implements **IntializingBean**,its **afterPropertySet**() method is called. If the bean has **init** method declaration, the specified **initialization** method is called.
* **Post-Initialization**: – If there are any **BeanPostProcessors** associated with the bean, their **postProcessAfterInitialization**() methods will be called.
* **Ready to use**: Now the bean is ready to use by the application
* **Destroy**: If the bean implements DisposableBean, it will call the **destroy**() method
* Spring beans are just object instances that are managed by the **Spring IOC container**.
* We can put the bean in to Spring by **Wiring** and **Auto Wiring**.
* **Wiring** mean we manually configure it into the **XML** file.
* **Auto Wiring** mean we put the annotations in the Java file then Spring automatically scan the root-context where **java configuration file**, make it and put into the bag of Spring.

@Configuration

**public** **class** BeanConfig {

@Bean(name = "sal" , initMethod="init", destroyMethod="destroy")

**public** Salary getSalaryBean() {

**return** **new** Salary();

}

}

**public** **class** Salary **implements** Company, BeanNameAware {

**private** String amount;

**@Autowired**

**private Employee employee;**

}

### ****Configuration metadata?****

* A Spring configuration file is an XML file.
* This file mainly contains the classes information.
* It describes how those classes are configured as well as introduced to each other. The XML configuration files, however, are verbose and cleaner.
* If it’s not planned and written correctly, it becomes very difficult to manage in big projects.

### ****Bean scopes?****

The Spring Framework supports five scopes. They are:

* **Singleton:**This provides scope for the bean definition to single instance per Spring IoC container.
* **Prototype:**This provides scope for a single bean definition to have any number of object instances.
* **Request:**This provides scope for a bean definition to an HTTP-request.
* **Session:**This provides scope for a bean definition to an HTTP-session.
* **Global-session:**This provides scope for a bean definition to an Global HTTP-session.

Annotation Based: -

#Singelton

@Component

@org.springframework.context.annotation.Scope(value = ConfigurableBeanFactory.***SCOPE\_PROTOTYPE***,

proxyMode = ScopedProxyMode.***TARGET\_CLASS***)

**public** **class** Text **implements** Messages {

@Autowired

**private** NetworkAuth con;

**public** **void** sendMessage() {

**if** (con.getConnection()) {

System.***out***.println("Text sent " + con);

}

}

}

**XML BASED:**

<bean id=*"email"*

class=*"com.rm.dev.springconcept.scope.beans.Email"* autowire=*"byName"*

primary=*"true"*>

<property name=*"con"* ref=*"con"*></property>

</bean>

<bean id=*"text"* class=*"com.rm.dev.springconcept.scope.beans.Text"* autowire=*"byName"*>

<property name=*"con"* ref=*"con"*></property>

</bean>

* proxyMode = ScopedProxyMode.TARGET\_CLASS Important if Prototype mode is used and class which wire dependency is Singleton.

@Component

@org.springframework.context.annotation.Scope(value = ConfigurableBeanFactory.SCOPE\_PROTOTYPE

,proxyMode = ScopedProxyMode.TARGET\_CLASS)

public class NetworkAuth {

int id = 0;

public NetworkAuth() {

}

public boolean getConnection() {

return this.id != 0 ? true : false;

}

}

**XML BASED:**

<bean id=*"con"* class=*"com.rm.dev.springconcept.scope.NetworkAuth"*

autowire=*"byType"* scope=*"prototype"*>

<aop:scoped-proxy proxy-target-class=*"true"* />

</bean>

### ****Explain Scoped Proxy in Spring****

**Spring** Scoped Proxy Beans – **An Alternative to Method Injection**. ... For those of you who are not aware of Method Injection, it allows you to inject methods instead of objects in your class. Method Injection is useful in scenarios where you need to inject a smaller scope bean in a larger scope bean.

* For example, you must inject a prototype bean inside a singleton bean, on each method invocation of Singleton bean.
* Just defining your bean prototype, does not create new instance each time a singleton bean is called because container creates a singleton bean only once, and thus only sets a prototype bean once.
* So, it is completely wrong to think that if you make your bean prototype you will get new instance each time prototype bean is called.

### Define Bean Wiring? When beans are combined within the Spring container, it’s called wiring or bean wiring.

### ****Auto wiring and name the different modes of it?****

The XML-configuration-based auto wiring functionality has five modes – no, byName, byType, constructor, and autodetect. The default mode is no.

Spring supports the following auto wiring modes:

* **no**: It’s the default auto wiring mode. It means no auto wiring.
* **byName**: The byName mode injects the object dependency according to name of the bean. In such a case, the property and bean name should be the same. It internally calls the setter method.
* **byType**: The byType mode injects the object dependency according to type. So it can have a different property and bean name. It internally calls the setter method.
* **constructor**: The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having many parameters.
* **autodetect**: In this mode, Spring first tries to auto wire by the constructor. If this fails, it tries to auto wire by using by Type.

**Question**: How container know what type of wiring ?

**Answer**: We define it as byType, byName , constructor.

**Question**: Is there are way we do not define type of autowiring ?

**Answer**: Yes, it's there by doing one annotation, @Autowired.

**Question**: But how system know, I need to pick this type of secondary data ?

**Answer**: You will provide that data in you spring.xml file or by using sterotype annotations to your class so that container can themselves create the objects for you.

# **2.0 Annotation**

## **2.1 limitations with auto wiring?**

* **Overriding possibility:**You can always specify dependencies using <constructor-arg> and <property> settings which will override auto wiring.
* **Primitive data type:**Simple properties such as primitives, Strings and Classes can’t be auto wired.
* **Confusing nature:**Always prefer using explicit wiring because auto wiring is less precise

## **2.2 How annotation wiring can be turned on in Spring**

<context:annotation-config />

## **What’s the difference between @Component, @Controller, @Repository & @Service annotations in Spring?**



**@Component:** This marks a java class as a bean. It is a generic stereotype for any Spring-managed component. The component-scanning mechanism of spring now can pick it up and pull it into the application context.

**@Controller:** This marks a class as a Spring Web MVC controller. Beans marked with it are automatically imported into the Dependency Injection container.

**@Service:** This annotation is a specialization of the component annotation. It doesn’t provide any additional behavior over the @Component annotation. You can use @Service over @Component in service-layer classes as it specifies intent in a better way.

**@Repository:** This annotation is a specialization of the @Component annotation with similar use and functionality. It provides additional benefits specifically for DAOs. It imports the DAOs into the DI container andmakes the unchecked exceptions eligible for translation into Spring DataAccessException.

## **2.3 What do you understand by @Required annotation**

## **2.4 What do you understand by @Autowired annotation**

@Resource can be used instead of @Autowired + @Qualifier when retrieving a bean of a specific name, and @ComponentScan has been added to scan for beans/configuration files due to them not existing as children of the package containing the @SpringBootApplication.

## **2.5 What do you understand by @Autowired annotation**

## **2.6 What do you understand by @RequestMapping annotation**

# 3.0 Spring Dao

## **3.1 Describe Spring DAO support**

## **3**.2 Which classes are present in spring JDBC API?

## **3.3 What are the ways by which Hibernate can be accessed using Spring**

## **3.4 Name the types of transaction management that Spring supports**

## **3.5 What are the different ORM’s supported by Spring?**

# **4.0 Describe AOP**

* Aspect Oriented Programming is sensibly new, and it is not a replacement for Object Oriented Programming.
* In fact, AOP is another way of organizing your Program Structure.

Some very common examples of these could be:

1. **Transaction Management**
2. **Logging**
3. **Exception Handling (especially when you may want to have detailed traces or have some plan of recovering from exceptions)**
4. **Security aspects**
5. **Instrumentation**

## 4.1 What is Aspect?

1. **Aspect = Point cut + Advice**

**@Pointcut("execution(\* com.rm.learn.springaop.controller.\*.\*(..))")**

**@Pointcut("within(com.rm.learn.springaop.service..\*)")**

**@Pointcut("execution(\* com.rm.learn.springaop.dao.\*.\*(..))**

**[")](mailto:\")@Pointcut(\"@annotation(com.rm.learn.springaop.aop.TimeTrack))**

**[@Pointcut("@annotation(com.rm.learn.springaop.aop.TimeTrack)](mailto:\")@Pointcut(\"@annotation(com.rm.learn.springaop.aop.TimeTrack))**

**[")](mailto:\")@Pointcut(\"@target(org.springframework.stereotype.Repository))**

**[@Pointcut("@target(org.springframework.stereotype.Repository)](mailto:\")@Pointcut(\"@target(org.springframework.stereotype.Repository))")**

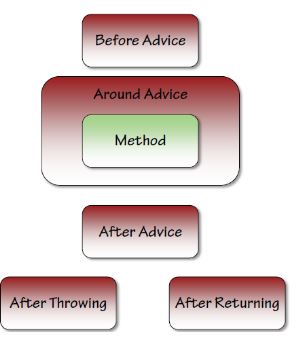
@After("com.rm.learn.springaop.aop.AopPointCutConfiguration.getTimeTrack()")

**public** **void** aroundTimeTrack(JoinPoint joinPoint) **throws** Throwable {

**long** value = System.*currentTimeMillis*();

*logger*.info(" aroundTimeTrack maxTime {} " , (System.*currentTimeMillis*() - value));

}



@Before("com.rm.learn.springaop.aop.AopPointCutConfiguration.getControllerPointCut()")

@After("com.rm.learn.springaop.aop.AopPointCutConfiguration.getServicePointCut()")

@Around("com.rm.learn.springaop.aop.AopPointCutConfiguration.getDaoPointCut()")

@AfterReturning(value = "com.rm.learn.springaop.aop.AopPointCutConfiguration.getControllerPointCut()",returning = "obj")

@AfterThrowing(value = "com.rm.learn.springaop.aop.AopPointCutConfiguration.getAllPointCut()",throwing="ex")

@After("com.rm.learn.springaop.aop.AopPointCutConfiguration.getTimeTrack()")

* **Advise** defines what needs to be apply.
* **Join point** is where an **Advice** is applied.
* **Pointcut** is a combination of different **Joint points**.
* **Aspect** is applying an **Advice** at **Pointcuts**.

## 4.2 What is Joint point and Point cut?

* Join point is a point of execution of the program, such as the execution of a method or the handling of an exception.
* In Spring AOP, a **join point always represents a method execution**.
* For example, all the methods defined inside your EmployeeManager interface can be considered joint points if you apply any cross-cutting concern of them.

**Pointcut is a predicate or expression that matches join points.**

* Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, expression “execution(\*EmployeeManager.getEmployeeById(..))” to match getEmployeeById() the method in EmployeeManager interface).
* The concept of join points as matched by pointcut expressions is central to AOP.
* Spring uses the AspectJ pointcut expression language by default.

## 4.3 What is Weaving?

* Applying **aspects to beans declared in the IoC container**

## 4.4 Concern and cross-cutting concern in Spring AOP?

**Concern is behavior which we want to have in a module of an application.** Concern may be defined as a functionality we want to implement to solve a specific business problem. E.g. in any eCommerce application different concerns (or modules) may be inventory management, shipping management, user management etc.

**Cross-cutting concern is a concern which is applicable throughout the application (or more than one module).** e.g. logging, security and data transfer are the concerns which are needed in almost every module of an application, hence they are termed as cross-cutting concerns.

# **5.0 Spring MVC framework**

## **5.1 Describe DispatcherServlet**

## **5.2 Explain WebApplicationContext**

## **5.3 In Spring MVC framework, what is controller**

## 5.4 Web App-Servlet

* Browser sends Http Request to Web Server
* Code in Web Server => Input: Http Request, Output: HTTP Response JEE with Servlets
* Web Server responds with Http Response
* Servlet is a Java programming language class used to extend the capabilities of servers that host applications accessed by means of a request-response programming model.

@**WebServlet**(urlPatterns = "/login.do")

**public** **class** LoginGetServlet **extends** HttpServlet {

@Override

**protected** **void** doGet(HttpServletRequest request, HttpServletResponse response)

**throws** IOException, ServletException {

request.getRequestDispatcher("/WEB-INF/view/Login.jsp").forward(request, response);

}

}

<!-- webapp/WEB-INF/web.xml -->

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns=*"http://java.sun.com/xml/ns/javaee"* xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"* version=*"3.0"*>

<display-name>To do List</display-name>

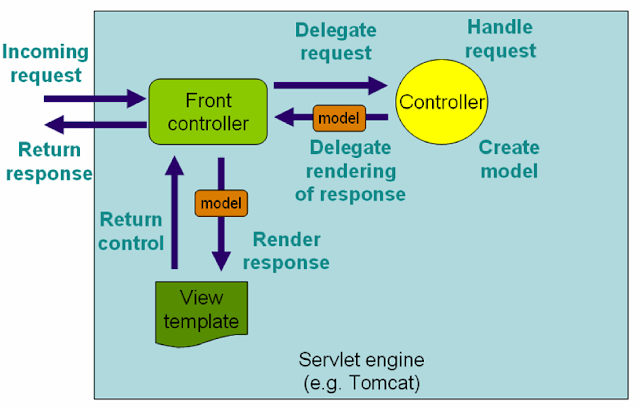
<welcome-file-list>

<welcome-file>login.do</welcome-file>

</welcome-file-list>

</web-app>

## 5.5 Spring MVC Web-App XML Configuration.



* The client sends an HTTP request to a specific URL
* DispatcherServlet of Spring MVC receives the request org.springframework.web.servlet.DispatcherServlet
* It passes the request to a specific controller depending on the URL requested using @Controller and @RequestMapping annotations.
* Spring MVC Controller then returns a logical view name and model to DispatcherServlet.
* DispatcherServlet consults view resolvers (*org.springframework.web.servlet.view.InternalResourceViewResolver)*  until actual View is determined to render the output
* DispatcherServlet contacts the chosen view (like Thymeleaf, Freemarker, JSP) with model data and it renders the output depending on the model data

The rendered output is returned to the client as a response  
  
org.springframework.web.servlet.DispatcherServlet - FrameworkServlet 'dispatcher': initialization started

org.springframework.web.context.support.XmlWebApplicationContext - Refreshing WebApplicationContext for namespace 'dispatcher-servlet': startup date [Sun May 10 08:36:55 IST 2020]; root of context hierarchy

org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loading XML bean definitions from ServletContext resource [/WEB-INF/todo-servlet.xml]

INFO 2020-05-10 08:36:57,186 [localhost-startStop-1]

org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor - JSR-330 'javax.inject.Inject' annotation found and supported for autowiring

org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping - Mapped "{[/login]}" onto private static java.lang.String com.controller.LoginServlet.getFirstPage()

org.springframework.web.servlet.mvc.method.annotation.**RequestMappingHandlerMapping** - Mapped "{[/welcome],methods=[POST]}" onto private java.lang.String com.controller.LoginServlet.getWelcomePage(java.lang.String,java.lang.String,org.springframework.ui.ModelMap)

org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter - Looking for @ControllerAdvice: WebApplicationContext for namespace 'dispatcher-servlet': startup date [Sun May 10 08:36:55 IST 2020]; root of context hierarchy

org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter - Looking for @ControllerAdvice: WebApplicationContext for namespace 'dispatcher-servlet': startup date [Sun May 10 08:36:55 IST 2020]; root of context hierarchy

2020-05-10 08:41:36,801 [http-bio-8089-exec-4] com.controller.LoginServlet - Inside getWelcomePage

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

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns=*"http://java.sun.com/xml/ns/javaee"* xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"* version=*"3.0"*>

<display-name>To do List</display-name>

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/todo-servlet.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/spring-mvc/\*</url-pattern>

</servlet-mapping>

</web-app>

<context:component-scan base-package=*"com"* />

<mvc:annotation-driven />

<bean

class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<property name=*"prefix"*>

<value>/WEB-INF/view/</value>

</property>

<property name=*"suffix"*>

<value>.jsp</value>

</property>

</bean>

## 5.6 Spring MVC Web-App Java Configuration?

|  |  |  |
| --- | --- | --- |
| XML Tag | Annotation | Description |
| <context:component-scan/> | @ComponentScan() | Scan starts from base package and registers all controllers, repositories, service, beans, etc. |
| <mvc:annotation-driven/> | @EnableWebMvc | Enable Spring MVC-specific annotations like @Controller |
| Spring config file | @Configuration | Treat as the configuration file for Spring MVC-enabled applications. |

**public** **class** AnnoationLoginServlet **implements** WebApplicationInitializer {

**public** **void** onStartup(**final** ServletContext sc) **throws** ServletException {

AnnotationConfigWebApplicationContext root = **new** AnnotationConfigWebApplicationContext();

root.scan("com");

sc.addListener(**new** ContextLoaderListener(root));

ServletRegistration.**Dynamic** appServlet = sc.addServlet("dispatcher",

**new** **DispatcherServlet**(**new** GenericWebApplicationContext()));

appServlet.setLoadOnStartup(1);

appServlet.addMapping("/");

}

@EnableWebMvc

@ComponentScan(basePackages = { "com" })

@Configuration

**public** **class** SpringConfig **extends** WebMvcConfigurerAdapter {

@Bean

**public** ViewResolver viewResolver() {

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

viewResolver.setViewClass(JstlView.**class**);

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

viewResolver.setSuffix(".jsp");

**return** viewResolver;

}

}

# 6 Spring Boot -> Web App Java Configuration

## **6.1 Introduction**

@SpringBootApplication annotation can be used to enable those three features, that is:

@EnableAutoConfiguration: enable [Spring Boot’s auto-configuration mechanism](https://docs.spring.io/spring-boot/docs/2.1.13.RELEASE/reference/html/using-boot-auto-configuration.html)

@ComponentScan: enable @Component scan on the package where the application is located (see [the best practices](https://docs.spring.io/spring-boot/docs/2.1.13.RELEASE/reference/html/using-boot-structuring-your-code.html))

@Configuration: allow to register extra beans in the context or import additional configuration classes

The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration, and @ComponentScan with their default attributes,

### 6.1.1 Default Package Structure

com

+- example

+- myapplication

+- Application.java

|

+- customer

| +- Customer.java

| +- CustomerController.java

| +- CustomerService.java

| +- CustomerRepository.java

|

+- order

+- Order.java

+- OrderController.java

+- OrderService.java

+- OrderRepository.java

@SpringBootApplication // same as @Configuration @EnableAutoConfiguration @ComponentScan

**public** **class** SpringBootWebMvcApplication **extends** SpringBootServletInitializer {

@Override

**protected** SpringApplicationBuilder configure(SpringApplicationBuilder application) {

**return** application.sources(SpringBootWebMvcApplication.**class**);

}

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

SpringApplication.*run*(SpringBootWebMvcApplication.**class**, args);

}

}

### 6.1.2 Application.properties

spring.mvc.view.prefix: /WEB-INF/view/

spring.mvc.view.suffix: .jsp

logging.level.org.springframework = INFO

server.port=8092

management.security.enabled=false

## 6.2 Hal Browser and Actuator

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-rest-hal-browser</artifactId>

</dependency>

* + **localhost:8080/actuator**
  + **localhost:8080/application**

Add in application.properties = management.endpoints.web.exposure.include=\*

* + Actuator through HAL Browser
  + <http://localhost:8098/browser/index.html#/actuator>
* It provides many features, i.e. what beans are created, the mapping in the controller, the CPU usage, etc.
* Automatically gathering and auditing health and metrics can then be applied to your application.

## Spring Boot Rest Api

@Autowired

**private** UserEnrollementService service;

@GetMapping(value= "/getUserDetails/{id}")

**public** List<UserVO> getUserDetails(@PathVariable("id") String id) {

UserVO vo = service.getUserDetails(id);

**return** Arrays.*asList*(vo);

}

@GetMapping(value= "/getUserDetails")

**public** List<UserVO> getUserDetailsFromID(@RequestParam String id) {

UserVO vo = service.getUserDetails(id);

**return** Arrays.*asList*(vo);

}

@GetMapping(value= "/allUser")

**public** Collection<UserVO> allUser() {

**return** service.allUser();

}

@PostMapping(value= "/addUser")

**public** UserVO addUser(@RequestParam String id,

@RequestParam String name,

@RequestParam String country) {

**return** service.addUser(id,name,country);

}

@PutMapping(value= "/updateUser")

**public** UserVO updateUser(@RequestParam String id,

@RequestParam String name,

@RequestParam String country) {

**return** service.updateUser(id,name,country);

}

@DeleteMapping(value= "/deleteUser")

**public** Object deleteUser(@RequestParam String id) {

**return** service.deleteUser(id);

}

* Get Request can be tested directly from browser.
* Post / Put / Delete should be tested from Postman.

## 6.3 Spring Boot MVC

* Put and Delete request can’t be sent from HTML pages as method type not exits.
* Get request to retrieve data.
* Post request used to create / update / delete data.

@RestController

**public** **class** UserEnrollementController {

@Autowired

**private** UserEnrollementService service;

@GetMapping(value= "/allUser")

**public** ModelAndView allUser() {

ModelAndView modelView = **new** ModelAndView("userDetails");

List<UserVO> userVo = (List<UserVO>) service.allUser();

modelView.addObject("addAllUser",userVo);

**return** modelView;

}

@PostMapping(value= "/addUser")

**public** ModelAndView addUser(@RequestParam String id,

@RequestParam String name,

@RequestParam String country) {

System.***out***.println("REQUEST PARAMS " + id);

ModelAndView modelView = **new** ModelAndView("userDetails");

modelView.addObject("addedUser",service.addUser(id,name,country));

**return** modelView;

}

@PostMapping(value= "/updateUser")

**public** ModelAndView updateUser(@RequestParam String id,

@RequestParam String name,

@RequestParam String country) {

System.***out***.println("REQUEST PARAMS " + id);

ModelAndView modelView = **new** ModelAndView("userDetails");

modelView.addObject("updatedUser",service.updateUser(id,name,country));

**return** modelView;

}

@PostMapping(value= "/deleteUser")

**public** ModelAndView deleteUser(@RequestParam String id) {

System.***out***.println("REQUEST PARAMS " + id);

ModelAndView modelView = **new** ModelAndView("userDetails");

modelView.addObject("deletedUser",service.deleteUser(id));

**return** modelView;

}

### 6.3.1 Testing

@SpringBootTest

@RunWith(SpringJUnit4ClassRunner.**class**)

**class** SpringBootWebMvcApplicationTests

### 6.3.2 Application-Start-Up

@SpringBootApplication // same as @Configuration @EnableAutoConfiguration @ComponentScan

**public** **class** SpringBootWebMvcApplication **extends** SpringBootServletInitializer {

@Override

**protected** SpringApplicationBuilder configure(SpringApplicationBuilder application) {

**return** application.sources(SpringBootWebMvcApplication.**class**);

}

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

SpringApplication.*run*(SpringBootWebMvcApplication.**class**, args);

}

}

## **6.4 Exclude any package without using the basePackages filter?**

You can use the exclude attribute while using the annotation  @SpringBootApplication.

@SpringBootApplication(exclude= {Employee.class})

public class FooAppConfiguration {}

//By using property file

spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration

## **6.5 What is the Spring Initializer?**

* The Spring Initializer is a web application that generates a Spring Boot project with everything you need to start it quickly.
* As always, we need a good skeleton of the project; it helps you to create a project structure/skeleton properly

## **6.6 What is a shutdown in the actuator?**

* S[hutdown](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html) is an endpoint that allows the application to be gracefully shutdown.
* This feature is not enabled by default.
* You can enable this by using management.endpoint.shutdown.enabled=true in your application.properties file

## **6.7 How can I enable auto reload of my application with Spring Boot?**

* Use Spring Boot Developer Tools.
* Adding Spring Boot Developer Tools to your project is very simple.
* Add this dependency to your Spring Boot Project pom.xml

<dependency>

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

<artifactId>spring-boot-devtools</artifactId>

<scope>runtime</scope>

</dependency>

* Restart the application

## **6.8 RequestMapping and GetMapping?**

* RequestMapping is generic - you can use with GET, POST, PUT or any of the other request methods using the method attribute on the annotation.
* GetMapping is specific to GET request method. It’s just an extension of RequestMapping to improve clarity.

## **6.9 What is the need for Profiles?**

Enterprise application development is complex. You have multiple environments

* Dev
* QA
* Stage
* Production

You want to have different application configuration in each of the environments.

*Profiles help to have different application configuration for different environments.*

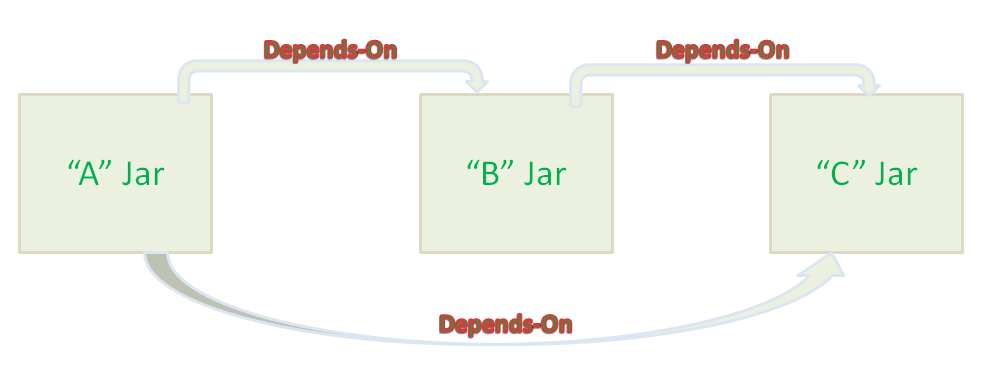
Spring and Spring Boot provide features where you can specify

* What is the configuration for various environments in different profiles?
* Set the active profile for a specific environment.

Spring Boot would pick up the application configuration based on the active profile that is set in a specific environment.

## **6.10 What is “Transitively Dependency Resolution Management”**

* “Transitively Dependency Resolution Management” means:
* If we define an “A” dependency in build scripts, “A” is dependent on “B” and “B” is dependent on “C”,
* That means “A” is also dependent on “C”.  
  Then Build Tools will download and add all Three Jar files “A”, “B” and “C” to our application class path.



**What is the difference between an embedded container and a WAR?**  
  
The main difference between an embedded container and a WAR file is that you can Spring Boot application as a JAR from the command prompt without setting up a web server.

But to run a WAR file, you need to first set up a [web server](http://www.java67.com/2016/06/3-difference-between-web-server-vs-application-server-vs-servlet-container.html) like Tomcat which has Servlet container and then you need to deploy WAR there.