**SPRINGS**

**Advantages of Spring Framework**

1. **Lightweight:**Spring is lightweight when it comes to size and transparency. The basic version of spring framework is around 2MB.
2. **Inversion of control (IOC):** Loose coupling is achieved in Spring, with the [Inversion of Control technique](http://www.javacodegeeks.com/2011/08/what-is-dependency-inversion-is-it-ioc.html). The objects give their dependencies instead of creating or looking for dependent objects.
3. **Aspect oriented (AOP):** [Spring supports Aspect oriented programming](http://www.javacodegeeks.com/2011/01/aspect-oriented-programming-spring-aop.html) and separates application business logic from system services.
4. **Container:** Spring contains and manages the life cycle and configuration of application objects.
5. **MVC Framework:** Spring’s web framework is a well-designed [web MVC framework](http://www.javacodegeeks.com/2011/02/spring-mvc-development-tutorial.html), which provides a great alternative to web frameworks.
6. **Transaction Management:** Spring provides a consistent transaction management interface that can scale down to a local transaction and scale up to global transactions (JTA).
7. **Exception Handling:** Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO) into consistent, unchecked exceptions.

**ARCHITECTURE**

->spring framework contains lot of features, which are well organized with 6 modules as shown below.

->Starting from spring 2.5 it became possible to configure dependency using annotation.

**WEB**

SPRING WE MVC,TRAMEWORK INTEGRATION,STRUTS,WEBWORK,TAPESTRY,JSF,RICH VIEW SUPPORT,JSP,VELOCITY,FREEMARKER,PDF,JASPER REPORTS,EXCEL,SPRING PORTAL MVC

**JEE**

JMX,JMS,JCA,REMOTING,EJBS,EMAIL

**SPRING CORE CONTAINER**

THE IOC CONTAINER

**JDBC &DAO** SPRING JDBC TRANSACTION MANAGEMENT

**AOP**

SPRING AOP

ASPECTJ INTEGRATION

**ORM** HIBERNATE JPA TOPLINK JDO,OJB,IBATIS

There are two ways via which you can inject dependency in spring.

1. By configuring XML.
2. By using annotation.
3. Java Configuration

**Inject Dependency By configuring XML.**

**DEPENDENCY INJECTION(IoC)**

1)DI is the heart of spring framework.

2)it is a process of injecting(pushing) the dependencies into an object.

3)the basic concept of dependency injection is that,we don’t create objects but we describe how they should be created in a configuration file.

4)we don’t directly connect dependency object and main object together in code but we describe which dependencies are required by which objects in a configuration file.

5)the spring container(IoC container) is responsible for creating objects along with its dependencies.

Country.java Third party

Spring xml(application-context.xml)

String countryNm

Capital capitalObj

Capital instances

**BENEFITS OF DEPENDENCY INJECTION(IoC) IN SPRINGS**

1)loosely coupled architecture because configuration and code both are separate.

2)dependencies can be easily identified .no need to read the code to see that what dependencies a particular object has.

3)using configuration a different implementation can be supplied without changing the dependent code.

4)e.g.dev,test,stage,production database

**TYPES OF DEPENDENCY INJECTION(IoC)**

1)DI exists in two major variants.

->setter based DI

->constructor based DI

**Setter Injection**

->setter based dependencies injection is created by calling setter methods on main object or bean after invoking no argument constructor.

**Constructor Injection**

->constructor based dependency injection is created by invoking a constructor with arguments on main object or bean.

**WHEN TO USE CONSTRUCTOR AND SETTER BASED DEPENDENCY INJECTION**

1)spring team generally advocate the usage of setter injection because if a constructor has large no of arguments then it becomes cumbersome and also their might be a chance that some arguments might be optional. **setter method DI can be easily reconfigured at faster time**.

2)but the drawback of setter injection is that it doesn’t ensure DI.we can’t guarantee that certain dependencies is injected or not.which means we may have an object with incomplete dependency.

On the other hand constructor injection doesn’t allow to construct object until our dependencies are ready.

3)since we can mix both constructor & setter based DI.it’s better to use constructor injection for mandatory dependencies and setter injection for optional dependencies.

**Dependency Injection with Factory Method in Spring**

Spring framework provides facility to inject bean using factory method. To do so, we can use two attributes of bean element.

factory-method: represents the factory method that will be invoked to inject the bean.

factory-bean: represents the reference of the bean by which factory method will be invoked. It is used if factory method is non-static.

A method that returns instance of a class is called factory method.

## Factory Method Types

There can be three types of factory method:

1) A **static factory method** that returns instance of **its own** class. It is used in singleton design pattern.

<bean id="a" class="com.javatpoint.A" factory-method="getA"></bean>

2) A **static factory method** that returns instance of **another** class. It is used instance is not known and decided at runtime.

<bean id="b" class="com.javatpoint.A" factory-method="getB"></bean>

3) A **non-static factory** method that returns instance of **another** class. It is used instance is not known and decided at runtime.

<bean id="a" class="com.javatpoint.A"></bean>

<bean id="b" class="com.javatpoint.A" factory-method="getB" factory-bean="a"></bean>

**CIRCULAR DEPENDENCY**

->if we use predominantly constructor injection,it may create unresolvable circular dependency scenario.

->for example

1)class A requires an instance of class B through constructor injection and class B require an instance of class A through constructor injection again.

2)if we configure beans for class A and class B to be injected into each other.springIoC container detects this circular reference at runtime and throws a BeanCurrentlyInCreation Exception.

**SOLUTION TO CIRCULAR DEPENDENCY**

1)although it is not recommended.we can configure circular dependencies with setter injection unlike the typical case (with no circular dependencies).a circular dependencies between class A & class B forces one of the beans to be injected into other prior to being fully initialized itself.

**APPLICATION CONTEXT**

1)theorg.springframework.beans and org.springframework.context packages are the basis for spring framework IoC container.

2)the interface org.springframework.context.ApplicationContext represents the spring IoC container and is responsible for instantiating ,configuring and assembling the beans as mentioned in the configuration metadata.

3)the configuration metadata is represented in XML or java annotation or java code.

4)several implementation of the ApplicationContext interface are supplied out of the box with spring in stand alone application .it is common to create an instance of ClassPathXmlApplicationContext or FileSystemXmlApplicationContext. while xml has been the traditional format for defining configuration metadata.

5)ClassPathXmlApplicationContext equal to loads bean definition from an xml file located in the classpath.

6)FileSystemXmlApplicationContext equal to loads bean definition from an xml file located in the file system.

7)XmlWebApplicationContext equal to loads bean definition from an xml file contain within the web application.

8)following diagram is a high level view of how spring works.

BUSINESS OBJECTS(POJO’S)

FULLY CONFIGURED READY TO USE OBJECT

SPRING IOC CONTAINER

CONFIGURATION INFORMATION

**BEAN FACTORY**

1)the bean factory interface provides an advance configuration mechanism capable of managing any type of object.

2)ApplicationContext is a sub interface of BeanFactory .it adds easier integration with springs AOP features.

3)in short BeanFactory provides the configuration framework and basic functionality and application context adds more enterprise specific functionality.

4)theApplicationContext the complete superset of the BeanFactory.

**BeanFactory OR ApplicationContext**

1)use ApplicationContext unless we have a good reason for not doing so.because the ApplicationContext includes all functionality of the BeanFactory.it is generally recommended over the BeanFactory,except for a few situation such as in an applet,where memory consumption might be critical and few extra kb’s might make a difference.

**ApplicationContextAware INTERFACE**

1)when we want the ApplicationContext object in the bean class then we have implements ApplicationContextAware interface and override setApplicationContext() method.

2) Also we can set servletconext and servletconfig using ServletConextAware and ServletConfigAware interface and override setServletConext() and setServletConfig() method.

**BeanPostProcessor**

1. It is used to do some process after or before initializing all the bean.

**BeanFactoryPostProcessor**

1. Used to do some process before initialization of beanfactory.

**BEANS SCOPE**

1)in spring we can also provide scope for the object created from bean definition file(application-context.xml)

2)this approach is powerful and flexible so that we can define the scope of the objects in configurationfile,instead of taking care of the scope at java level.

3)out of the box spring framework supports five scopes.

**Singleton,prototype,request,session,globalSession**

**Singleton Scope**

1)if the bean have singleton scope then spring IoC container creates exactly one instance of the bean.

2)this single instance is stored in the cache and for all subsequent request IoC container return the bean instance from the cache.

**HOW TO IMPLEMENT SINGLETON DESIGN PATTERN IN SPRING**

**Prototype Scope**

1)its non-singleton in nature.

2)this scope results in creation of new bean instance everytime a request for that specific bean is made.

3)as a rule use prototype scope for all stateful beans and singleton scope for stateless bean.

**Request Scope**

1)in the bean has request scope then spring container create a new instance of the bean for each & every Http request.

2)when request completes then these beans is garbage collected.

3)operation is as same as request attribute in case of servlets.

**Session Scope**

1)if the bean has session scope then spring container creates new instance of a bean for every new Http session.when session ends then this bean is garbage collected.

2)operation is same as session attribute in case of servlets.

**GlobalSession Scope**

1)if the bean has globalSession scope then spring container creates a object or bean and is available for entire application and lives as long as application alive.

2)operation as same as context attribute in case of servlets.

Dfference between java singleton and spring singleton

|  |  |
| --- | --- |
| **Java singleton** | **spring singleton** |
| 1) Java singleton is scoped by the Java class loader | 1) Spring singleton is scoped by the spring container . |
| 2) Java singleton is a truly a singleton only within the context of the class loader which loaded it. Other class loaders should be capable of creating another instance of it. | 2) We can load spring singleton class in two different cointainer. |
| 3) Java singleton cannot create more than one object within a given class loader. | 3) whereas spring singleton cannot create more than one instance of a class within a given container |

**AUTOWIRING**

1)spring container is able to autowire relationship between beans.this is an automagical process in which bean references don’t need to be coded explicitly in the xml file but rather the spring container takes care of injecting dependencies.

2)there are five autowire nodes in springs.

->autowire “no”

->autowire “byName”

>autowire “byType”

>autowire “constructor”

>autowire “autodetect”(not present in spring 3.0 onwards)

**AUTOWIRING “no or default ”**

1)means no autoworing.this is the default mode.we need to wire dependent means via ref attribute (normal way to achieve dependency injection).

2)spring recommended this mode as it makes documentation more explicit.

**AUTOWIRING “byName”**

1)autowiring by property name.if id of a bean is same as the name of main bean property then container autowire.

2)there is no matching bean byname exist in configuration file then nothing special happen.

3)if there is more than one bean definition exist in configuration file with different bean id then container choose the proper bean with the property name and autowire.

**AUTOWIRING “byType”**

1)autowire a bean by property datatype.

2)if there is no bean definition exist then nothing special will happen.

3)if there is more than one bean definition exist in configuration file then exception is thrown and object creation fails.

**AUTOWIRING “constructor”**

1)autowire a bean by property datatype in constructor argument.

2)if there is no bean definition exist then again exception is thrown and object creation fails.

3)if there is more than one bean definition exist in configuration file then exception is thrown and object creation fails.

**AUTOWIRING “autodetect”**

1)autodetect internally makes use of autowire “constructor” or autowire “byType” through self analysis of the bean class.

2)if we found default constructor then it makes use of autowire=”byType”.

3)if non default constructor is found then it makes use of autowire=”constructor”.

**INITIALIZING COLLECTION IN SPRING**

->spring supports initializing the following collection via configuration file.

1)List(<list>)

2)Set(<set>)

3)Map(<map>)

4)Properties(<props>)

**LIFECYCLE CALL BACKS**

1)spring we can provide lifecycle method to the beans.this allows the bean to perform certain action upon initialization and destruction of that particular beans.

2)we can achieve this by following fashion.

->**by implementing InitializingBean and DisposableBean interfaces.**

<bean id="countryBean" class="org.arpit.javapostsforlearning.Country"/>

public class Country implements InitializingBean {

    public void afterPropertiesSet() {

        // do some initialization work

    }

}

<bean id="countryBean" class="org.arpit.javapostsforlearning.Country"/>

public class Country implements DisposableBean{

    public void destroy() {

        // do some destruction work(like releasing pooled connections)

 }

}

->**by defining the method names in configuration files.**

<bean id="countryBean" class="org.arpit.javapostsforlearning.Country" init-method="init"/>

public class Country{

    public void init() {

        // do some initialization work

    }

} 

<bean id="countryBean" class="org.arpit.javapostsforlearning.Country" init-method="destroy"/>

public class Country{

    public void destroy() {

        // do some destruction work(like releasing pooled connections)

    }

}

->by using @PostConstruct and @PreDestroy annotations.

**Collections in Spring**

<beanid=*"countryBean"*class=*"com.spring.collections.beans.Country"*>

<propertyname=*"countryname"*value=*"India"*/>

<propertyname=*"listOfStates"*>

<list>

<value>MP</value>

<value>UP</value>

<value>Bihar</value>

</list>

</property>

</bean>

**Spring properties dependency checking**

**->**In Spring,you can use dependency checking feature to make sure the required properties have been set or injected.

**Dependency checking modes**

There are four types of dependency checking modes.

1)none – No dependency checking.

2)simple – If any properties of primitive type (int, long,double…) and collection types (map, list..) have not been set, UnsatisfiedDependencyException will be thrown.

3)objects – If any properties of object type have not been set, UnsatisfiedDependencyException will be thrown.

4)all – If any properties of any type have not been set, an UnsatisfiedDependencyException  
will be thrown.

**Inject Dependency By using annotation**

->In all our previous posts,we have injected dependency by configuring XML file but instead of doing this,we can move the bean configuration into the component class itself by using annotations on the relevant class, method, or field declaration.

->You might think what if you have done both i.e.used annotations and XML both.In that case,XML configuration will override annotations because XML configuration will be injected after annotations.

->Now annotations based configuration is turned off by default so you have to turn it on by entering **<context:annotation-config/>** into spring XML file.

<?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:context="http://www.springframework.org/schema/context"

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

    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

    http://www.springframework.org/schema/context

    http://www.springframework.org/schema/context/spring-context-3.0.xsd">

**<context:annotation-config/>**

 <!-- beans declaration goes here -->

</beans> 

[**@Required:**](http://javapostsforlearning.blogspot.in/2012/09/required-annotation-in-spring.html)

-> [dependency](http://www.mkyong.com/spring/spring-properties-dependency-checking/) check is used to make sure all properties of a certain types (primitive, collection or object) have been set. But when we need to make sure a particular property has been set or not then we have to use @Required annotation on setter methods.

[**@Autowired:**](http://javapostsforlearning.blogspot.in/2012/09/autowired-annotation-in-spring.html)

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

[**@Qualifier:**](http://javapostsforlearning.blogspot.in/2012/09/qualifier-annotation-in-spring.html)

* The @Qualifier annotation along with @Autowired can be used to remove the confusion by specifiying which exact bean will be wired.
* you can have more than one bean of same type in your XML configuration but you want to autowire only one of them ,so @Qualifier removes confusion created by @Autowired by declaring exactly which bean is to autowired.

e.g-

<bean id="jedisConnector" class="com.legolas.jedis.JedisConnector" init-method="init" destroy-method="destroy">

<property name="host" value="${jedis.host}" />

<property name="port" value="${jedis.port}" />

</bean>

<bean id="jedisConnectorPOD" class="com.legolas.jedis.JedisConnector" init-method="init" destroy-method="destroy">

<property name="host" value="${jedis.pod.host}" />

<property name="port" value="${jedis.pod.port}" />

</bean>

@Autowired

@Qualifier("jedisConnector")

JedisConnector beanA;

@Autowired

@Qualifier("jedisConnectorPOD")

JedisConnector beanB;

Or

@Resource(name="jedisConnector")

JedisConnector beanA;

@Resource(name="jedisConnectorPOD")

JedisConnector beanB;

Or

@Resource

JedisConnectorjedisConnector;

@Resource

JedisConnectorjedisConnectorPOD;

[**@Repositry:**](http://javapostsforlearning.blogspot.in/2012/09/qualifier-annotation-in-spring.html)

@Repository is an annotation that marks the specific class as a Data Access Object, thus clarifying it's role. Other markers of the same category are @Service and @Controller

<repositoriesbase-package="com.acme.repositories" />

<repositoriesbase-package="com.acme.repository"repository-impl-postfix="FooBar" />

*@Configuration*

*@EnableWebMvc*

*@EnableSpringDataWebSupport*

classWebConfiguration { }

*@Controller*

*@RequestMapping("/users")*

[@Target](http://java.sun.com/javase/6/docs/api/java/lang/annotation/Target.html?is-external=true)([value](http://java.sun.com/javase/6/docs/api/java/lang/annotation/Target.html?is-external=true#value%28%29)=[TYPE](http://java.sun.com/javase/6/docs/api/java/lang/annotation/ElementType.html?is-external=true#TYPE))

[@Retention](http://java.sun.com/javase/6/docs/api/java/lang/annotation/Retention.html?is-external=true)([value](http://java.sun.com/javase/6/docs/api/java/lang/annotation/Retention.html?is-external=true#value%28%29)=[RUNTIME](http://java.sun.com/javase/6/docs/api/java/lang/annotation/RetentionPolicy.html?is-external=true#RUNTIME))

[@Documented](http://java.sun.com/javase/6/docs/api/java/lang/annotation/Documented.html?is-external=true)

[@Component](http://docs.spring.io/spring/docs/2.5.6/api/org/springframework/stereotype/Component.html)

*@Bean*

CrudRepository

[DataAccessException](http://docs.spring.io/spring/docs/2.5.6/api/org/springframework/dao/DataAccessException.html)

**Spring MVC**

The Spring web MVC framework provides model-view-controller architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

* The **Model** encapsulates the application data and in general they will consist of POJO.
* The **View** is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
* The **Controller** is responsible for processing user requests and building appropriate model and passes it to the view for rendering.

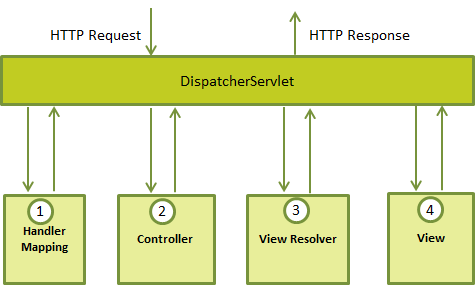
## The DispatcherServlet

The Spring Web model-view-controller (MVC) framework is designed around a *DispatcherServlet* that handles all the HTTP requests and responses. The request processing workflow of the Spring Web MVC

Following is the sequence of events corresponding to an incoming HTTP request to *DispatcherServlet*:

1. After receiving an HTTP request, *DispatcherServlet* consults the *HandlerMapping* to call the appropriate *Controller*.
2. The *Controller* takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the *DispatcherServlet*.
3. The *DispatcherServlet* will take help from *ViewResolver* to pickup the defined view for the request.
4. Once view is finalized, The *DispatcherServlet* passes the model data to the view which is finally rendered on the browser.

*DispatcherServlet* is illustrated in the following diagram:



**Important points on load-on-startup element**

1. By default servlet is loaded at first request then it consumes more time at first request. So better it should load at deployment time or server start if value is positive so it will take less time at first request. Hence we use <load-on-startup> in web.xml file.
2. Also known as pre initialization of servlet.
3. If you pass the positive value, the lower integer value servlet will be loaded before the higher integer value servlet.
4. If you pass the negative value, servlet will be loaded at request time, at first request.
5. @RestController
6. @RequestMapping("/app")
7. public class UserController {
8. @Autowired
9. private UserService userService;
10. @JsonView(Profile.PublicView.class)
11. @RequestMapping(value= "/publicprofile", produces = MediaType.APPLICATION\_JSON\_VALUE)
12. public List<User> getAllPublicProfile() {
13. List<User> users = userService.getAllUsers();
14. return users;
15. }
16. }

* Here @JsonView(Profile.PublicView.class) produce JSON response only with public view properties.
* **Spring REST architecture**is also based on Spring MVC, slightly making the difference on the View part. Traditional Spring MVC relies on the View technology to render the model data, the Spring REST architecture also does the same, except that the model object is set directly into the HTTP response, which the **@ResponseBody** converts into JSON/XML automatically.

**Spring AOP**

* Main aim of aop is separating service(logging , security Tx,mailing or jms,) layer from business. Using AOP combine both the layer.

In Spring AOP, 4 type of advices are supported :

1. **Before advice** – It will execute before the method execution. Create a class which implements MethodBeforeAdvice interface.
2. **After returning advice** – It will execute after the method is returned a result. Create a class which implements **AfterReturningAdvice** interface.
3. **After throwing advice** – It will execute after the method throws an exception. Create a class which implements ThrowsAdvice interface, and create a**afterThrowing** method to hijack the **IllegalArgumentException** exception.
4. **Around advice** – It combines all three advices above, and execute during method execution. Create a class which implements **MethodInterceptor** interface. You have to call the **“methodInvocation.proceed();**” to proceed on the original method execution, else the original method will not execute.

### Conclusion

Most of the Spring developers are just implements the ‘Around advice ‘, since it can apply all the advice type, but a better practice should choose the most suitable advice type to satisfy the requirements.

**Diff between JoinPoints and Pointcut**

1)When you go out to a restaurant, you look at a menu and see several options to choose from. You can order one or more of any of the items on the menu. But until you actually order them, they are just "opportunities to dine". Once you place the order and the waiter brings it to your table, it's a meal.

2)Join points are the options on the menu and pointcuts are the items you select. A joinpoint is an opportunity within code for you to apply an aspect...just an opportunity. Once you take that opportunity and select one or more joinpoints and apply an aspect to them, you've got a pointcut.

Jointpoints are the options where I can put the advice. It means it contains all the method informations.

3)So basically **PointCuts are the subset of JoinPoints**.

4)Two types of pointcut.

1)Static Method Mather pointcut

Used to filters method on the basis of name, type of arguments and class name but not by the values of the arguments.

2)Name Method Mathcher pointcut

Used to filters method on the basis of the values of the arguments. For example, you can apply the advice to ding(10); but not ding(20);

**Aspect**

* A service which I have to add before or after or after throwing exception of the business method.9

**Advice**

* Indicate the action to take either before or after the method execution.(service provider)
* Wildcard Expression : @Before(“execution(\* get\*(\*))”)

**Pointcut**

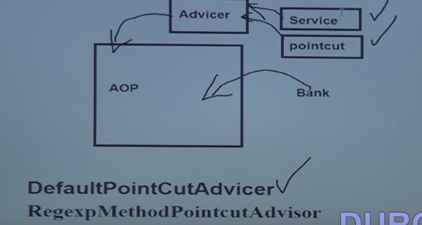
* A point or condition to execute aspect(service) for business method. Suppose we have multiple method and I have to apply a service for particular method then we have to use pointcut.
* Two pointcut.StaticMethodMatcherPointcut and NameMatchMethodPointcut.

**Advisor**

* Combination of ‘Advice’ and ‘Pointcut’ is called advisor.
* Two advisors. DefaultPointcutAdvisor and RegexpMethodPointcutAdvisor.

**Proxy**

* It will combine business code along with services code.



**Target**

* It is a business class.

**InceptorName**

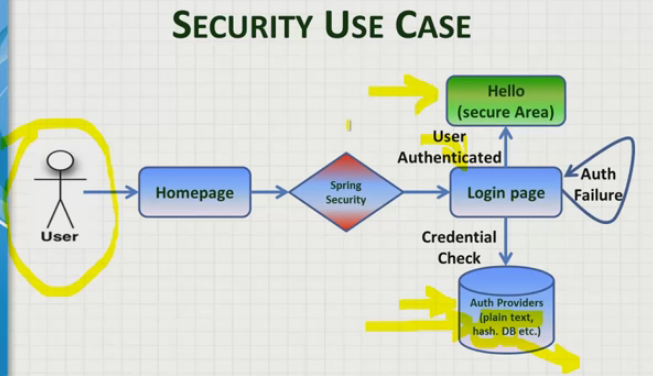
* It is a service name which has to apply with business.

**Spring Security**

1. Use to provide security services for J2EE-based enterprise software applications.
2. Major Operations

Authentication of username and password(Who are you)

Authorization of roles(Are you allowed for access)



3)Three types of security

->Windows security

->Default form

->Custom form

4)Hash password concept

->use to convert the password into fixed length of character and use into security.xml file. So that nobody can find the password.

5)Database password

Spring Framework 4.0

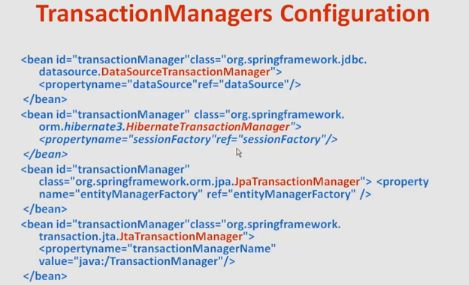
1. Spring 4 introduced @RestController which is combination of @Controller + @ResponseBody. So when using @RestController, you do not need to use @ResponseBody. It’s optional. Till Spring 3, we would have been using [@Controller](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/stereotype/Controller.html) annotation and in that case it was important to use [@ResponseBody](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/ResponseBody.html) annotation
2. The AsyncRestTemplate class has been added, [allowing non-blocking asynchronous support](https://docs.spring.io/spring/docs/4.1.5.RELEASE/spring-framework-reference/html/remoting.html#rest-async-resttemplate) when developing REST clients.
3. **Removed Deprecated Packages and Methods.**
4. **Spring Framework 4.0 provides support for several Java 8 features. You can make use of lambda expressions and method references with Spring’s callback interfaces.**
5. Spring now offers comprehensive timezone support when developing Spring MVC applications.e.g. on LocaleContext.

**Transaction**

* By default, if a RuntimeException is thrown from within the transactional code, the transaction will be rolled back. If a checked exception (i.e. not a RuntimeException) is thrown, then the transaction will not be rolled back.

**Declarative transaction management**

* allows you to manage the transaction with the help of configuration instead of hard coding in your source code.
* Steps
  1. We use <tx:advice /> tag, which creates a transaction-handling advice and same time we define a pointcut that matches all methods we wish to make transactional and reference the transactional advice.
  2. If a method name has been included in the transactional configuration then created advice will begin the transaction before calling the method.
  3. Target method will be executed in a *try / catch* block.
  4. If the method finishes normally, the AOP advice commits the transaction successfully otherwise it performs a rollback.





* **Propagation Level :**

1. REQUIRED

* Transaction support a current transaction, create a new one if none exists.
* outer and inner both transaction will rollback.

1. REQUIRES\_NEW

* Transaction create a new transaction and suspend the current transaction if one exists.
* outer and inner transaction both will independent, it means inner transaction will be rollback but outer transaction may commit.

1. NESTED

* Create the nested transaction from same physical transaction but sets some savepoint between nested method so inner transaction may also rollback independently of outer transaction.
* This behavior should only be used with spring jdbc managed transaction.

1. MANADATORY

* It states that an existing opened transaction must already exist. if opened transaction is not there then container will throw the exception.

1. NEVER

* It states that an existing opened transaction must not already exist. if opened transaction is exists then container will throw the exception.

1. NOT\_SUPPORTED

**->** It will execute outside of the scope of any transaction. If an opened transaction already exists it will be paused

7) SUPPORT

-> It will execute in the scope of a transaction if an opened transaction already exists.

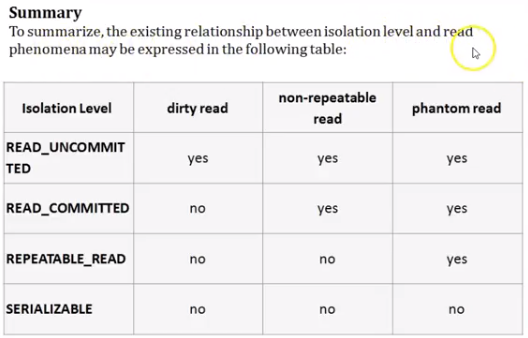
-> If an opened transaction is not exists the method will execute but in a non-transactional way.

* **Isolation Level**
* If multiple users modify the data simultaneously then data inconsistency problem will come so resolve this problem we can use isolation level.
* This controls database locking.
* There are multiple problems will occur with concurrency.

1. Lost Updates : If two users update the same row without locking then last update overwrites and first one is lost.
2. Dirty Read : If first transaction writes the data and at same time second transaction read the data that has not been committed and make the report. Now first transaction roll back the change then report of the second transaction has incorrect data.
3. Nonrepeatable Read : If first transaction writes the data and at same time second transaction read the data. Then first data rollback the change again second transaction read the data twice. Hence second transaction get two different results.
4. Phantom Reads : First transaction updating or deleting the rows on database and at same time second transaction selecting a set of rows. Now first transaction complete the operation and commit then second transaction get the different result set.

* There are multiple isolation levels are there to resolve above problems.

1. READ\_UNCOMMITTED : This is the lowest isolation level and here dirty read, non-repeatable read and phantom reads are possible.
2. COMMITTED : This is the default level and here non-repeatable read and phantom reads are possible.
3. REPEATABLE\_READ : here phantom reads are possible.
4. SERIALIZABLE : This isolation level resolves all above problems.



**Spring with Java Configuration**

1. **@Component**

Used on class level to create the object of the class by spring container automatically.

1. **@Bean**

Use inside configuration file at method level to create the bean of a class.

1. **@Configuration**

Use on configuration class and that class will use inside the constructor of AnnotationConfigApplicationContext.

1. **@ComponentScan**

For java configuration use on configuration class but for annotation based use tag into context.xml file.

1. **@Autowire**

Use to inject any object inside another object.

1. **@Primary or @Qualifier**

Suppose I have multiple bean of same type then used @Primary on class level to tell that which bean I have to used.

Otherwise use qualifier with autowire at variable level.

1. **@Import**

Use to import one configuration into another configuration class.

1. **@Scope**

Used with @component or @Bean to indicate that how many objects will create.

1. @**Lookup**

Used at getter method of singleton class to inject prototype bean under singleton bean.

1. **@Required**

Used on setter method. if we are not using this annotation then we can pass null value but after using mandatory we have to pass this value otherwise it will throw bean initialization exception.