

25/08/11

## Spring

framework is a special SW, that is developed based on core technologies having capability to develop the common logics of the applications development dynamically. This process makes application developer to just concentrate on application specific logic development. This increases productivity in application development.

Doing more work in less time with good accuracy is called good productivity.

### **Download more materials**

#### Web flow SW's :-

<http://ameerpetmaterials.blogspot.in/> To develop MVC based web applications.

<http://ameerpetmatbooks.blogspot.in/>

<http://satyajohnny.blogspot.in/>

These are given based on Servlet, JSP core technologies.

#### ORM flow SW's :-

Hibernate, EJB3, Toplinks, JDO, OJB .....

These use JDBC as underlying core technology.

ORM flow SW based persistence logic is db independent.

JDBC persistence logic is db dependent.

#### Java-JEE flow SW's :-

#### Spring

(Allows) provides environment to develop all Java, JEE

applications in frameworks style. It removes all the code

User multiple JSE, JEE module technologies as Core Technologies like JDBC, JNDI, servlet, Jsp, JavaMail, JMS etc.

Using Spring we can develop Standalone applications, desktop applications, two tier applications, web applications, distributed apps, enterprise apps, n-tier apps, etc

Plain JDBC application :-

- ① Register driver with DriverManager service.
  - ② Establish connection with db driver.
  - ③ Create statement object.
  - ④ Prepare sql query, send and execute sql query in db.
  - ⑤ Gather results and process results.
  - ⑥ Perform exception handling.
  - ⑦ Perform Transaction management.
  - ⑧ Close JDBC object (especially connection object)
- 
- The diagram illustrates the decomposition of a plain JDBC application. It shows a vertical list of steps from 1 to 8. Brackets on the right side group certain steps into 'Common logic' and others into 'app specific logic'. Step 1 (Register driver) and step 2 (Establish connection) are grouped under 'Common logic'. Step 8 (Close JDBC object) is also grouped under 'Common logic'. Steps 3 through 7 are grouped under 'app specific logic'.

Spring JDBC Application :-

\* Spring JDBC application internally uses plain JDBC

1. Get access to JDBCTemplate object.
2. Prepare sql query, send and execute sql query in db.
3. Gather results and process results.

Note : In spring JDBC application the JDBCTemplate will take care of all the common logic generation, and makes programmer to just concentrate on the app specific logic's development.

So this simplifies the process of application development.

flw slw's provide abstraction layer on core technologies and simplifies the process of application development.

The abstraction layer word indicates without having strong knowledge on core technologies the programmer can develop core technologies based application with the support of flw slw.

Every flw slw uses one or more core technologies as underlying technologies for application development and execution.

### Difference between Struts and Spring

#### Struts

1. It is a web flw slw to develop mvc 2 arch web apps.
2. we must need web/application server to execute struts apps
3. struts app resources are struts API dependent.
4. Allows only Jsp programs in view layer.
5. Doesn't provide built-in middleware services.
6. Runs on MVC2 principles.

#### Spring

1. It is a Java-Gee flw slw to develop any kind of apps including web.
2. Some spring apps can be executed without web/application server.
3. spring app resources are spring API independent.
4. In spring based web apps Velocity, freemarker and Jsp programs can be taken in the view layer.
5. provides built-in middleware services.
6. runs on dependency injection or inversion of control principles.

Middleware services are additional, optional services/ logics which can be applied on the applications to run the applications perfectly and accurately ~~in~~ all the situations.

Ex: 1. security service (protects the application)

2. Transaction Management Service (executes the app logic)

by applying do everything or nothing principle

(Ex: Transfer money task logic)

3. Logging Service. (keeps tracks of app flow of execution)

through Log messages or confirmation messages)

Etc (6+ middleware services are there) ...

## Spring:

Type : Java-Gee fwk slw to develop any Java, Gee applications

Version : 2.5 (compatable with Java 1.5/1.6), 3.x (compatable with

Vendor : Interface & Java 1.6 + ....)

Creator : Mr. Rod Johnson

It is open source slw, download slw as zip file  
from [www.springframework.org](http://www.springframework.org) website

for good online tutorial [www.springframework.org](http://www.springframework.org)

[www.roredika.net](http://www.roredika.net).

Reference Book : Spring live, Spring in action, ..

The seven modules of spring 1.1 :-

1. Spring core (base for all other)

2. Spring dao

4. Spring web

5. Spring web mvc

6. Spring AOP

## Spring six modules:

Spring core (base for all other modules)

Spring DAO

Spring ORM

Spring JEE (Same as Spring 1.x Context module)

Spring web (Spring 1.x web + Spring 1.x webmvc)

Spring AOP (Aspect oriented programming)

\* AOP is noway related with OOP (Object oriented programming)

→ OOP is the methodology of creating programming languages like C++, Java etc.

→ AOP is the methodology of applying middleware services like security service, Transaction Management service etc on spring application.

If resource/application is spending time to search and gather its dependent values then it is called dependency lookup. In dependency lookup the resource/application pulls the values from other resources.

Ex: Student gathers his course material from Institute by requesting for it.  
→ gathering explicitly.      ↳ dependent value

If underlying container Jw or fw Jw or server Jw is dynamically assigning values to resource application then it is called as dependency injection / inversion of control.

In dependency injection the underlying fwk or server fwk dynamically pushes the dependent values to resource / application.

Ex: If student is getting course material automatically the moment he registered for course

Container is a fwk or fwk application that can take care of the whole life cycle of given resource or component from object birth to object death.

Ex: Applet viewer (container takes care of applet life cycle)

Servlet container (takes care of servlet program life cycle)

Spring container (takes care of Spring Resource life cycle)  
(class)

In Spring we have two within containers,

Those are 1. Beanfactory (part of Spring core module)

2. ApplicationContext (part of Spring core/context module)

Spring containers are light weight containers.

Servlet Container, JSP Container and EJB Containers are fwk applications of web server, application server fwk so they are heavy weight containers.

Spring containers are given as predefined Java classes.

by just creating objects for these classes we can activate spring containers anywhere. This indicates Spring containers are light weight containers.

The two containers of spring are now alternate for  
servlet container, JSP container and EJB container.

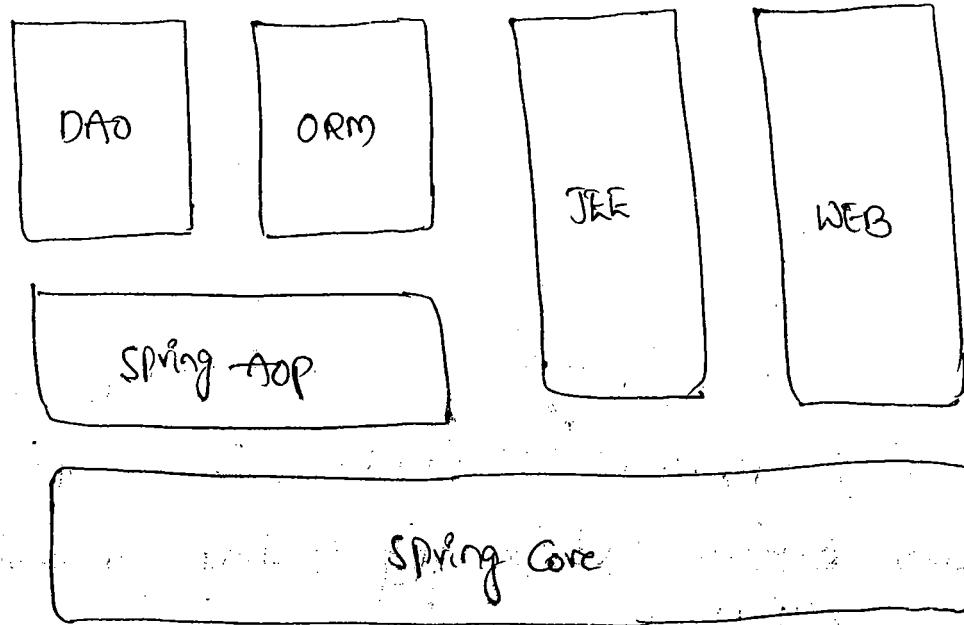
Resource means a program or file of a application.

Spring resources means the classes and interfaces of  
Spring application.

Spring I-x High level Architecture Diagram :-



Spring &-x High level Architecture (Overview) Diagram :-



Spring Core module is base module for other modules, it provides Beanfactory container supporting Dependency Injection or Inversion of Control.

Spring DAO module allows the programmer to develop framework style JDBC persistence logic by providing abstraction layer for the core JDBC technology.

Spring ODM module provides abstraction layer on ODM frameworks like Hibernate, JPA, TopLink, JDO etc. It provides environment to develop db SQL independent OR Mapping persistence logic.

Spring JEE module provides abstraction layer on JMS, EJB, RMI, JavaMail etc concepts of Java, JEE environment. This module gives environment to develop distributed applications, enterprise applications and other JEE applications in fw style.

Spring Web module provides,

- a) Environment to make spring applications communicable from the web fw like struts app, JSF application etc.
- b) provides spring's own web fw called <sup>fw</sup> SpringMVC | SpringWebMVC to develop MVC2 arch based web applications.

Spring AOP provides environment to develop new middleware services and to apply existing mw services on server applications.

29/08

The client server applications with location transference are called as distributed applications.

(Independency)

Location Transference means the change in sever application location will be recognized by clients dynamically.

A web app can be developed as distributed or non-distributed application.

The website with location Transference is called distributed application. (Ex: The real website like gmail.com etc)

The web app without location Transference are called client-server application (ex: class room level web app's)

The app that deals with complex, heavy weight, large scale business logic having the support of middleware services, is called as Enterprise Application.

Ex: Banking application, credit/debit card processing app, etc.

Spring can develop Standalone, desktop, two-tier, web, distributed, enterprise, n-tier applications.

### MVC 2 :-

Model layers → Business logic + persistence logic

View layers → presentation logic

Controller layer → Integration logic / Connectivity logic

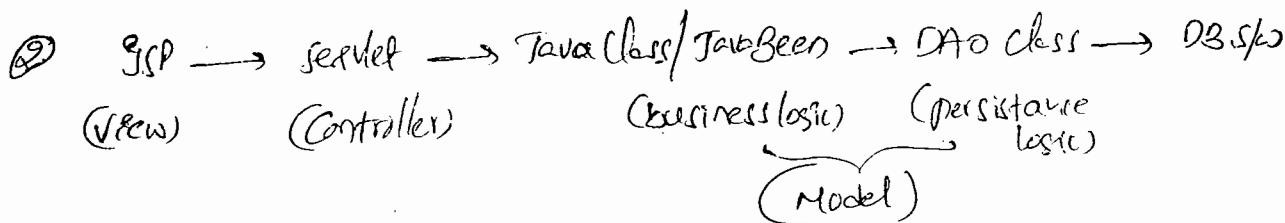
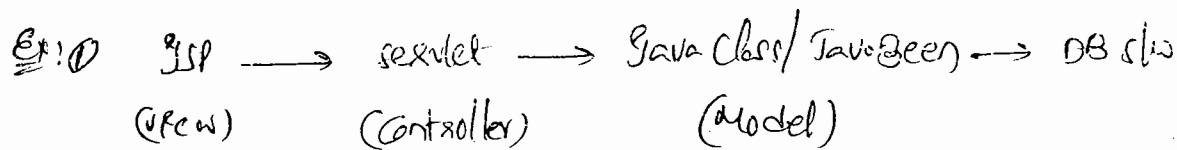
The logic that gives communication b/w view and model layers resources, that monitors and controls all the operations of application execution is called as Integration logic.

The logic that generates UserInterface to provide input values and to format results is called as presentation logic. (Ex HTML)

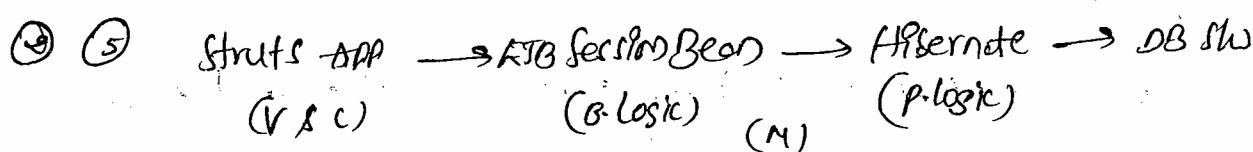
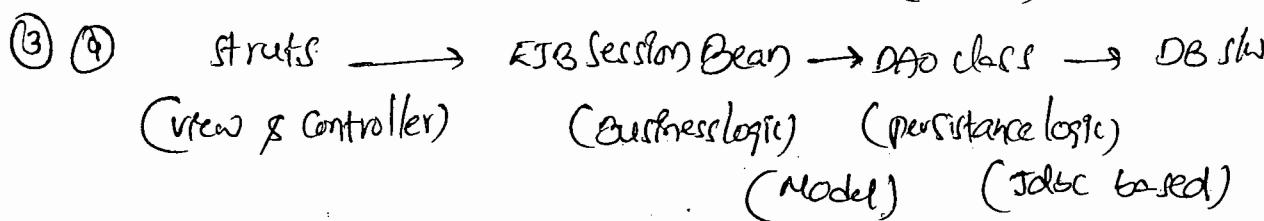
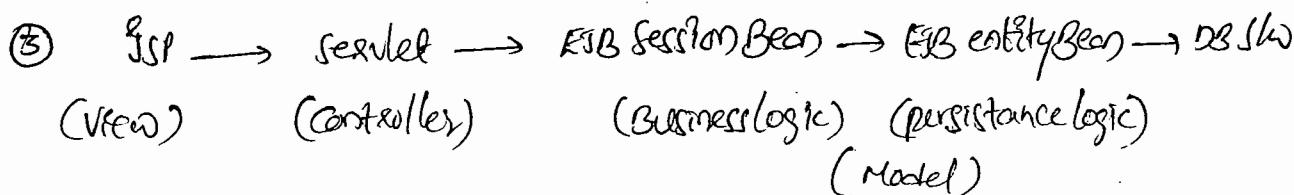
The main logic of the application that generates results is called as Business logic / service logic.

The logic that manipulates db data is called as persistency logic. ( JDBC code, Hibernate code etc)

Now a days all Companies are preferring to develop their projects as MVC architecture based projects.



DAO: The Java class that separates persistance logic from other logics of the application is called as DAO.



\*: RIB Components are heavy weight, because they need RIB Container and application server is the kind of heavy weight s/w's for execution.

Spring can be used in every layers of MVC architecture based projects to develop all the logics, as of now Endless, is using spring only in the model layer to develop business logics as alternate for its Session Bean.

## The possible technologies of view layers

JSP, HTML, velocity, freemarker, xslt, etc

## controller layers

## Servlet, Servlet filters

Web flow flow's to develop View & Controller layer logics.

struts, webwork, gsf, wicket, xwork, cocoon, Tansley  
spring web mvc etc.

Model layer to develop Business logic

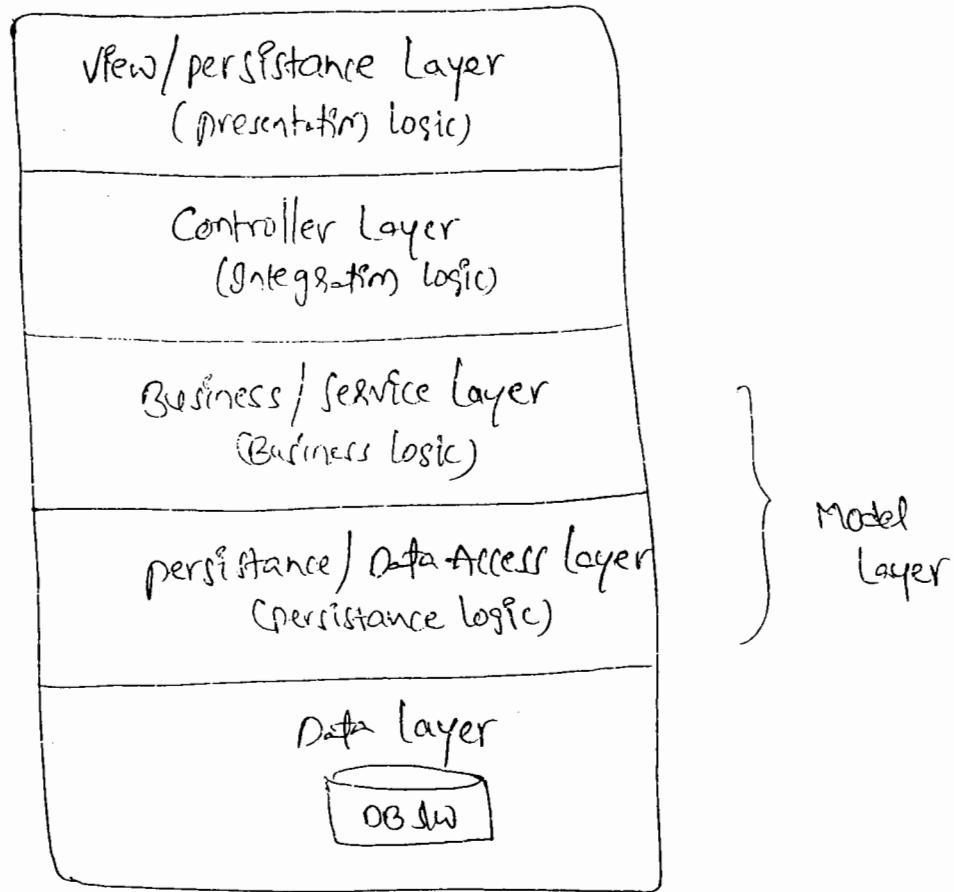
EJB Session Bean, spring (see module), Jack class, JavaBeans, RMI, CORBA, Web Services, HTTP Sniffer etc.

Model Layer to develop persistance logic

JDBC, Hibernate, PBatis, Spring DAO, Spring ORM, JDO, OJB etc

The above are some technologies which are used in various layers to develop respective logics.

## Project High Level Architecture :-



In struts-spring-hibernate combination based projects:

JSP → View/Presentation layer

Action servlet → Controller layer  
& resources

Spring Core API → Service/Business layer  
Spring Sec API

Spring ORM | → persistence layer  
Hibernate

Oracle/MySQL → Data layer

Spring :- Spring is an open source, light weight, loosely coupled, aspect oriented, dependency injection based Java-EE framework to develop all kinds of applications in Java.

~~3dof~~  
OpenSource

OpenSource SW are not only free SW but they also expose their source code to programmers when SW is installed.

To install Spring 2.5 SW extract "spring-framework-2.5.1-with-dependencies.jar" file to a folder.

<sup>lightweight</sup> Spring-home/src folder contains the source code of Spring SW.  
Spring SW and Spring applications are light-weight because of the following reasons.

1. To run Spring applications the heavy weight app server and web server based servlet container and EJB containers are not required. (for most of the Spring applications).
2. Spring supplies it own light weight containers as BeanFactory or ApplicationContext Containers which can be activated anywhere by just creating objects for some predefined classes.
3. Most of the resources (classes and interfaces) can be developed without using Spring API in Spring applications.

\* The Servlet, JSP, EJB, etc Components of JEE are heavy weight bcoz they need heavy weight Container/ servers (like web and app servers) for execution and the resources of those components are their respective API dependent.

- <sup>loosely coupled</sup>: If degree of dependency is very less b/w two components then they are called as loosely coupled components.
- If degree of dependency is high/more then they are called as tightly coupled components.

Ex: CPU and monitor are tightly coupled devices  
TV and remote are loosely coupled devices.

Spring SW is loosely coupled SW, becoz

1. we can use specific spring module or all spring modules in application development. (The degree of dependency is less b/w spring modules).
2. we can use spring to develop whole project or along with other technologies to develop the project.

Aspect Oriented: Aspect means middleware service like security, Transaction Management etc.

The AOP module of spring SW provides facilities for the programmer to work with built-in middleware services and third party middleware services. This module also allows the programmer to develop user-defined SW services.

These features of spring makes the Spring SW as Aspect Oriented.

Dependency Injection: The common principle that can be seen in every spring app. is dependency injection/IoC, that means the values required for a spring app resources will be assigned by spring SW, SW / container dynamically.

Developing Spring app is nothing but developing normal Java application where the resources of the app will be developed by taking the support of Spring API.

To know various versions, their releases, and their features of ~~various~~ Spring you refer "changelog.txt file."

springhome/dist/spring.jar file represents the whole Spring API for this jar file commons-logging.jar file is the dependent jar file. That means the classes of spring.jar file are using the classes and interfaces of commons-logging.jar.

The sample projects can be gathered from mock, samples folders of springhome directory.

The springhome/test folder contains multiple sample examples on individual concepts.

Annotations are Java statements and they are alternate for the XML file based resources configurations and metadata operations. The resources of Spring application can be configured either by using annotations or by using XML files.

for annotations based Spring app samples refer springhome/tiger folder.

The program or file of the application is called as resource, Making underlying fw or server to recognize resource and its details is called as resource configuration.

The way we specify servlet program details in web.xml file is nothing but servlet program configuration.

~~overly~~ Spring application resources are Spring API independent. That means the classes and interfaces of Spring application can be developed as POJO classes and POJO's.

An ordinary class is called as POJO class and an ordinary interface is called as POJI.

Spring applications are light weight, bcoz it supports POJO/POJI model programming.

EJB 3.x, Spring 2.x, Struts 2.x, Hibernet 3.x are designed supporting POJO/POJI model programming.

Every SW technology contains API, API is the base for programmer to develop that technology based SW applications.

In Java API comes in the form of classes, interfaces, annotations, enums etc, by residing in packages.

JDBC API → java.sql, javax.sql packages

AWT API → java.awt, java.awt-client packages.

POJO class:- If Java class that is taken as resource of certain SW technology based Java application and if that class is not extending, implementing predefined classes and interfaces of the technology specific API then that Java class is called as POJO class.

Ex:- If Java class is taken as resource of Spring application and if that class is not implementing and not extending the predefined classes and interfaces of spring API then that class is called as POJO class.

Ex:- public class Test extends HttpServlet { } class Test extends HttpServlet { }  
} = using Java with this we can do { } → Based on Spring

POTI: - If Java interface is taken as resource of certain technology specific SW application and if that interface is not extending from the predefined interfaces of that technology specific API then that interface is called as POTI.

If Java interface is taken as Spring application resource and if that interface is not extending from the predefined interfaces of Spring API then that interface is called as POJO.

If Java app uses third party API (other than PdK's API's)  
Then the third party API related manifest and dependent jar files  
must be added to classpath to make Java tools (javac, java etc.)  
to recognize and use third party API's

- \* when Spring application uses Spring API then Spring API related both main and dependent jar files must be added to classpath (Spring.jar, Commons-logging.jar)
  - main
  - dependent to main

03\09\11

## Spring features :-

Light weight flow is used to develop all kinds of Java, C++, and other applications in this style.

Supports  $\text{P}(\text{TO})/\text{P}(\text{TE})$  model programming

Supports both XML, Annotations based configurations.

Allows to Develop standalone, desktop, two tier, web distributed, enterprise app.

Gives built-in middleware services like Connection Pooling, Transaction Management etc.

Allows to work with 3rd party supplied or server managed middleware services.

Allows to develop user defined middleware services using Spring AOP module.

Provides abstraction layer on existing technologies like Jdbc, Grid, sockets, Jsp, Rmi, Bios, Javamail,... to simplify the development process.

Provides its own slw's to develop applications directly.

Ex: HttpWorker : spring's own distributed technology to develop distributed apps.

Spring webMVC : spring's own web fwk slw to develop mvc arch based projects.

Provides light weight containers which can be activated without using webserver or app server slw.

Can be used to develop whole project, or can be used along with other technologies in the project development.

Gives good support for dependency injections.

Easy to learn and easy to apply.

Improves productivity in the project development.

Provides abstraction on ORM slw to develop ORM persistence logic.

Spring app resources means classes and interfaces, these are also called as Spring Beans.

Spring Beans are many related with JavaBeans, But JavaBean can also become as SpringBean (Every SpringBean need not be JavaBean)

Spring Bean can be a pojo or non pojo class.

predefined or User-defined or Third party supplied Java classes

that can be instantiated by spring container is called as Spring Bean.

All Spring beans must be configured in Spring configuration file

Ex: Spring Bean class

1. Java. Util. Date.

2. Public class Test

{  
    || some properties

,  
    || some methods

Any file named.xml can act as Spring cfg file, while activating Spring Container we must specify this Spring cfg file name.

To activate Beanfactory container create obj for a Java class that implements "org.st. beans. factory. Beanfactory" (interface)

The regularly used implementation class is

"org. st. beans. factory. Xml Beanfactory".

Ex:

```
fileSystemResource res = new fileSystemResource("demo.xml");
```

```
Xml Beanfactory factory = new Xml Beanfactory(res);
```

Spring Core Module: -

→ Gives Beanfactory container

→ Allows to observe diff modes of dependency injection & life cycle <sup>bean</sup>

→ Allows to develop app that can have only local clients

Note:-

In Spring environment any reusable and ~~reusable~~ <sup>reusable</sup>

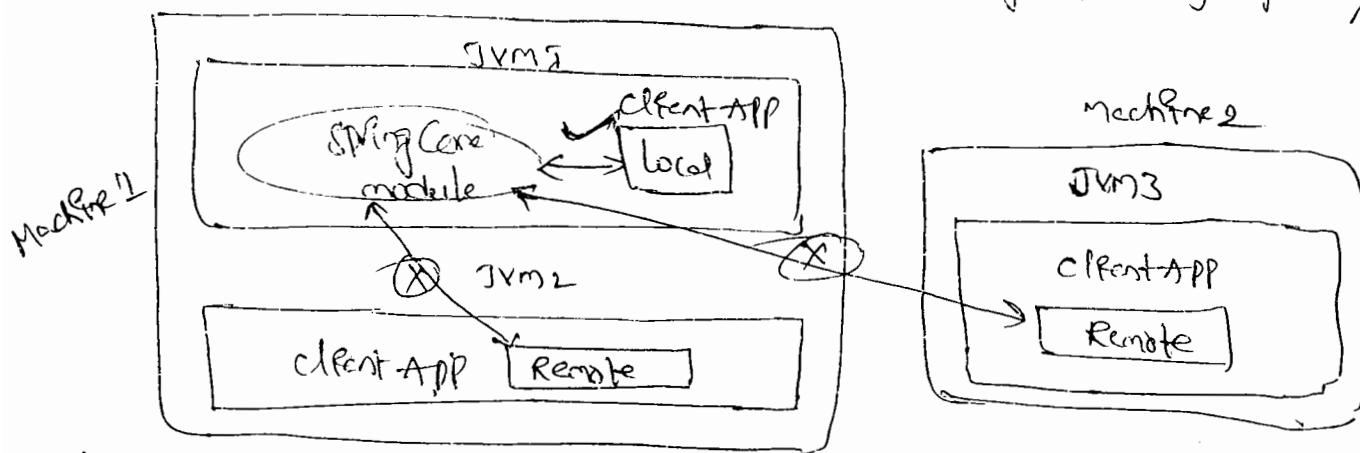
Java class is called as SpringBean

App and its client for same J2EE level, different → remote client

Distributed apps often work with local and remote clients

But Spring Core module cannot develop distributed apps.

To run multiple Java apps simultaneously from single computer we need multiple JVM (Java Virtual Machine) in that computer. i.e. In one computer multiple JVM's can be activated simultaneously or parallelly.



### Resources of Spring Core Module App :-

- Spring Core module app resources*
- 1) Spring Interface (Can be a POJO) (.java) → optional
  - 2) Spring Bean class (Can be a POJO class) (.java)
  - 3) Spring Bean Configuration file (Anyfilename.xml) (.xml)
  - 4) Client Application (Must be local client app) (.java)

### Spring Interface :-

Client to Core module application.

- It is POJO (It can be a POJO)
- Contains the declaration of business methods or utility methods.
- Acts as common understanding document b/w Spring bean developer and client application developer.

### Spring Bean :-

- Can be a POJO class.
- Can be a predefined or userdefined or third party supplied class.

\* → Implements spring interface and gives business methods;  
utility methods having business logic.

- optional  
in spring  
beans
- Contains helper methods supporting dependency injections.
  - Contains helper methods as life cycle methods.
  - Contains optional utility methods, supporting some Bean life cycle operations.
  - Contains Bean properties as instance variables.

Note: The instance variables in SpringBean class are nothing but Bean properties. These properties can hold the dependent values injected by Spring container.

### Spring Bean Configuration file :-

- Any `filename.xml` can be there as Spring Config file.
- No default file <sup>name</sup> exists for Spring Config file.
- Contains Spring Beans configuration.
- Contains dependent values configuration.
- Contains other miscellaneous configurations.

### Client Application :-

- Must be local client to the above application.
- Contains logic to activate Spring Container.
- Contains logic to get SpringBean class obj from Spring Container.
- Calls business methods using SpringBean class obj reference.

Note: In dependency injection the SpringBean needs to spend time to search and gather dependent values. In d.i.p process the Spring Container dynamically assigns dependent values to Spring Bean.

## Advantages of Dependency Lookup :-

- Resource/Spring Bean can search and gather only the needed dependent values.

### DisAdvantage :

- Resource/Spring Bean needs to spend time to search and gather dependent values.

## Advantage of Dependency injection :-

- Resource/Spring Bean need not spend time to search and gather dependent values.

### DisAdvantage :

- Both necessary and unnecessary values will be injected.

\* Spring supports three modes of Dependency Injection,

Those are 1. Setter Injection (Through setXXX() of Spring Bean)

2. Constructor Injection (Through parameterized constructor of Spring Bean class)

3. Interface Injection (By implementing special interfaces on Spring Bean class)

In Setter Injection Spring Container uses setXXX() of Spring Bean class to assign dependent values to Bean properties.

In Constructor Injection Spring Container uses parameterized constructor to create Spring Bean class obj and to assign dependent values to Bean properties.

Sample code to understand Setter Injection process:-

Spring Bean (DemoBean.java)

```
public class DemoBean
```

```
{
```

```
    // Bean property
```

```
    String msg;
```

```
// Setters supporting Setter Injection
```

```
public void setMsg(String msg)
```

```
{ this.msg = msg; }
```

```
// write business method
```

```
public void talk()
```

```
{ } } } } } → Business logic of business method  
} } } } } → The injected 'msg' property value  
can be used here.
```

Demo.xml (Spring config file)

```
<beans>
```

```
<bean id="ds" class="DemoBean"> → SpringBean  
<property name="msg">  
    <values> hello </values> } → Bean property Config  
<property> } → value to Inject for Setter Injection  
</bean>  
</beans>
```

When above code is given to Spring Container

- a) Spring container loads DemoBean class and create Bean class object having name "ds" with support of zero argument constructor.  
 $(\text{Id attribute value}) [\text{like } \text{DemoBean ds} = \text{new DemoBean();}]$

b) Spring container dynamically calls setMsg() with argument value "hello" on Bean class object ds and injects value hello to "msg" property. [ds.setMsg("hello");]

Note: Every Spring Bean class must be configured in spring config file then only Spring container recognizes that class.

every Spring Bean will be identified through its BeanId.

Sample code for Constructor Injection :-

### DemoBean.java

```
public class DemoBean
{
    String msg;
    public DemoBean (String msg) → 1-param constructor
    {
        this.msg = msg;           supporting Constructor Injecting
    }
    public void say()
    {
    }
}
```

### Demo.xml

```
<beans>
    <bean id="db" class="DemoBean">
        <constructor-args>
            <value>Hello</value>
        </constructor-args> </bean> </beans>
```

Note: If Bean property is Configured by using <property> tag then Spring container performs Setter Injection on that property.

If Bean property is Configured by using <constructor-args> tag then Spring container performs Constructor Injection on that bean property.

When above XML code is given to Spring Container the Spring Container creates "DemoBean" class obj having name "db" by using one-param constructor of DemoBean class having value "Hello". Due to this value "Hello" will be injected to "msg" property.

Like DemoBean db = new DemoBean("Hello");

Q6/09/14

Naming Conventions of Spring Core module App:

- \* → Spring Interface Name
- \*Bean → Spring Bean class Name
- \*cfg.xml → Spring Configuration file Name
- \*Client → Client Application

Note: Naming conventions are suggestions to follow to provide self-description to resources of the application.

Spring Core Module Application:

es) Apps (To demonstrate Setter Injection)

↳ spring  
    ↳ test

↳ Demo.java  
↳ DemoBean.java  
↳ DemoCfg.xml  
↳ DemoClient.java

Procedure to Create and execute the above Core Module App:

Step 1: Develop the above resources related source code.

Demo.java (POJO):

```
public interface Demo
{
    public String generateWithMsg(String uname);
    public long findFactorial(int val);
```

## DemoBean.java (POJO):

```
import java.util.*;
```

```
public class DemoBean implements Demo
```

```
{  
    private String msg;
```

|| setter method supporting setter injection on msg property

```
    public void setMsg( String msg )
```

```
{  
    System.out.println("DemoBean: setMsg(-)");
```

```
    this.msg = msg;
```

|| implements business methods having business logic

```
    public String generateWishMsg( String uname )
```

```
{  
    System.out.println("DemoBean: generateWishMsg(-)");
```

```
    Calendar cl = Calendar.getInstance();
```

```
    int h = cl.get(Calendar.HOUR_OF_DAY);
```

```
    if ( h <= 12 )
```

```
        return msg + "Good Morning!" + uname;
```

```
    else if ( h <= 16 )
```

```
        return msg + "Good Afternoon!" + uname;
```

```
    else if ( h >= 20 )
```

```
        return msg + "Good Evening!" + uname;
```

```
    else
```

```
        return msg + "Good Night!" + uname;
```

```
    public long findFactorial( int val ) {
```

```
        System.out.println("DemoBean: findFactorial(-)");
```

```
        long res = 1
```

```
        for ( int i = 1; i <= val; ++i )
```

```
            res = res * i;
```

## DemoCfg.xml

```

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
 "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

```

↓  
[Get it from  
Spring-home/dtd/  
schemas/  
resources/  
spring-beans-2.0.dtd  
file.]

```

<beans>
  <bean id="ds" class="DemoBean">
    <property name="msg">
      <value>Hello</value>
    </property>
  </beans>
</beans>

```

Spring Bean Configuration

for Setter Injection.

\* Spring Configuration file can be developed either based on dtd rules or schema (xsd) rules.

The above Spring Config file is developed based on spring-beans-2.0.dtd

## DemoClient.java

```

import org.springframework.core.io.FileSystemResource;
import org.springframework.beans.factory.xml.XmlBeanFactory;

public class DemoClient
{
  public (String args[]) throws Exception
  {
    // locate Spring Configuration file
    FileSystemResource res = new FileSystemResource("DemoCfg.xml");
    // Activate Spring Container
    XmlBeanFactory factory = new XmlBeanFactory(res);
    // get Spring Bean object from Spring Container
    DemoBean obj = (Demo)factory.getBean("ds");
  }
}

```

// call business methods on spring Bean class object

```
S.O.P(" Wish msg is :" + bob1.generateWishMsg("Nani"));  
S.O.P(" factorial value of :" + bob1.findFactorial(5));  
}  
}
```

Step @: Add the Spring API related Main and dependent

JAR files (spring.jar, commons-logging.jar) to classpath.

Note: The above two JAR files are available in `spring-home\dist` folder.

Step ③: Compile all the Java resources.

```
E:\Apps\spring\sec1>javac *.java,
```

Step ④: Execute the Client Application

```
E:\Apps\spring\sec1>java DemoClient
```

07/09/11

```
Demo db1 = (Demo) factory.getBean("db");
```

Here this `factory.getBean("db")` performs,

- Makes BeanFactory container to load DemoBean class based on the bean id "db".
- Makes Spring container to create DemoBean class object using zero argument constructor.
- Makes Spring container to call `setMsg("Hello")` on DemoBean class obj to perform setter injection on 'msg' property.
- Returns demoBean class obj to client application, The client app is referring that object through Demo interface ref variable.

prototype of getBean() method :-

public Object getBean(String beanId) throws BeansException

If Java method return type is Concrete class then that method can return either that concrete class object or one of its subclass object.

If Java method return type is java.lang.Object class then it can return any Java class object. Because java.lang.Object is common super class for any Java class.

factory.getBean() returns SpringBean class object based on the given SpringBean id.

In Interface reference variable can refer its implementation class object and can be used <sup>to call</sup> the implemented methods of Implementation class.

Super class reference variable can refer its one sub class obj and can be used to call the overridden or implemented methods of subclass with respect to super class methods.

When Java method returns an object that object can be referred by using object related class reference variable or super class reference variable or interface reference variable implemented by that object related class.

When factory.getBean("ds") returns DemoBean class obj with respect to our first app then it can be referred by using

- DemoBean class reference variable.

`DemoBean obj = (DemoBean) factory.getBean("ds")`

(Can be used in all business methods)

b) Using object class reference variable. (Not recommended)

Object obj = factory.getBean("ds");  
(cannot be used to call b methods)

c) By using Interface reference variable that is implemented by DemoBean class. (i.e Demo Interface reference variable)

Demo bob = (Demo)factory.getBean("ds");  
(can be used to call b methods)

Note: It is never recommended to expose Spring Bean class code/name to client application. But client application should call business methods of Spring Bean class, so option c) is recommended.

Problems and Solutions related to factory.getBean() call :-

problem:

Object obj = factory.getBean("ds");  
obj.findFactorial(5); }  
obj.generateWithMsg("Nani"); } Invalid

Solution 1:

DemoBean bob = (DemoBean)factory.getBean("ds");  
bob.findFactorial(5); }  
bob.generateWithMsg("Nani"); } Valid

Solution 2:

Demo bob1 = (Demo) bob;  
(or) Demo bob1 = (Demo)factory.getBean("ds");  
bob1.findFactorial(5); }  
bob1.generateWithMsg("Nani"); } Valid

Spring Container perform dependency injection only on those Spring Bean objects that are created by Spring Container. It can not perform that dependency injection operations on programmer supplied and explicitly created Spring Bean class objects.

Note: The Spring Container use SAXParser (xml parser) to read and process XML documents. (Spring config file).

When Spring Container BeanFactory is activated it first reads and verifies XML entries of Spring config file against syntax rules and DTD rules but it never creates Spring Bean class object in this process.

BeanFactory Container creates Spring Bean class object only when client app calls getBean(-) with BeanId.

Ques

In Spring applications placing DOCTYPE statements or schema related statement at the top of Spring Config file is mandatory.

If certain bean property of Spring Bean class is configured for both setter, constructor injections with two diff values. Can you tell me which value will be effected?

Since setter(-) executes after constructor execution, the value assigned through setter injection will be effected as final value.

If all Bean properties of Spring Bean are configured for only setter injection then Spring Container uses zero param constructor to create Bean class object.

If few or all Bean properties of Spring Bean are configured only for constructor injection then Spring Container uses parameterized constructor.....

If few or all Bean properties are configured for both Setter, Constructor Injections then spring container uses parameterized constructor to create SpringBean class object.

The way JRE/JVM dynamically Constructor and assigns Initial values to instance variables, <sup>when</sup> Java object is created is known as Dependency Injection.

The way ActionServlet assigns the received form data of form page to formBean class properties by creating / locating formBean class object comes under Dependency Injection.

To assign simple values to Bean properties use <value> tag, to configure other Spring Bean objects to Bean properties use <ref> tag in Spring Configuration file.

Application :- (On Setter Injection) :-

### Resources :

Demo.java → spring interface  
DemoBean.java      } → Spring Beans  
TestBean.java  
DemoClient.java → Client Application

Demo.java :-

```
public interface Demo {  
    public String sayHello();  
}
```

DemoBean.java :-

```
import java.util.*;
```

```
public class DemoBean implements Demo {
```

```
    private int age; → simple property
```

```
private Date d;  
private TestBean tb;
```

} reference type properties.

// write setxxx() for Setter Injection

```
public void setAge (int age) {  
    this.age = age;  
    s.o.p ("DemoBean: setAge()");  
}
```

```
public void setDate (Date d) {  
    this.d = d;  
    s.o.p ("DemoBean: setDate()");  
}
```

```
public void setTb (TestBean tb) {  
    this.tb = tb;  
    s.o.p ("DemoBean: setTb()");  
}
```

```
public String sayHello () {  
    return "Good Morning :" + " age = " + age + " d = " + d.toString()  
          + " tb = " + tb.toString();  
}
```

TestBean.java :-

```
public class TestBean {  
    private String msg; public TestBean () { s.o.p ("Hello org example"); }  
    public void setMsg (String msg) {  
        this.msg = msg; s.o.p ("TestBean: setMsg()");  
    }  
    public String toString () {  
        return "TestBean: msg = " + msg;  
    }  
}
```

## DemoCfg.xml:-

```
<!DOCTYPE beans>

<beans>
    <bean id="dt" class="java.util.Date">
        <property name="date"><value>5</value></property>
        <property name="year"><value>105</value></property>
        <property name="month"><value>10</value></property>
    </beans>
    <bean id="t1" class="TestBean">
        <property name="msg"><value>Hello</value></property>
    </bean>
    <bean id="db" class="DemoBean">
        <property name="age"><value>24</value></property>
        <property name="d"><ref bean="dt"/></property> *
        <property name="tb"><ref bean="t1"/></property> **
    </beans>
</beans>
```

\* Spring Container injects `java.util.Date` class object to "d" property of `DemoBean` class.

\*\* Spring Container injects `TestBean` class object (`t1`) to "tb" property of `DemoBean` class.

## DemoClient.java :-

```
import org.springframework.core.io.FileSystemResource;
import org.springframework.beans.factory.xml.XmlBeanFactory;
public class DemoClient {
    {
        P.S. VM, (String s[1]) throws Exception
```

```

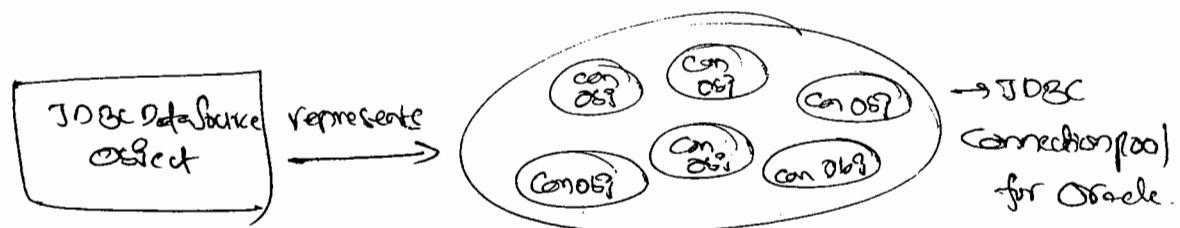
        fileSystemResource res = new fileSystemResource("DemoCfg.xml");
        XMLBeanFactory factory = new XMLBeanFactory(res);
        Demo bob = (Demo)factory.getBean("bob");
        System.out.println("Result is :" + bob.sayHello());
    }
}

```

Q19/IV

JDBC dataSource object represents JDBC Connection pool to access each JDBC connection object from JDBC Connection pool, we must depend upon JDBCDataSource object. JDBCDataSource object means it is the object of a Java class that implements `java.sql.DataSource` interface.

Client applications cannot access JDBC Connection objects of Connectionpool without using JDBC DataSource obj/reference of obj.



Spring supplies `org.springframework.jdbc.DataSource.DriverManagerDataSource` class which gives JDBC dataSource obj representing the built-in JDBC Connection pool of Spring Container environment.

→ Bean properties of DriverManagerDataSource class :-  
username, url, password, driverClassName

When Spring container instantiates the above DriverManagerDataSource class it collects the values from Bean properties and creates one JDBC Connectionpool representing this JDBC Connectionpool one JDBCDataSource object will be generated.

Example Application to inject JDBC DataSource object to Spring Bean through setter injection.

Resources of Application :

DBOperation.java (Spring Interface) → POJI

DBOperationBean.java (Spring Bean class) → POJO

SpringConfig.xml (Spring Configuration file)

DBClient.java (client application)

DBOperation.java :-

Public Interface DBOperation

```
{  
    public float fetchSalary (int eno);  
    }  
    public String fetchName (int eno);
```

DBOperationBean.java :-

```
import javax.sql.*;
```

```
import java.sql.*;
```

public class DBOperationBean implements DBOperation

{

DataSource ds; → To hold the injected JDBC DataSource obj

public void setDs (DataSource ds)

```
{  
    this.ds = ds;  
}
```

} → setXXXX for  
after Injection

|| Implementing business methods

public float fetchSalary (int eno).

```
{  
    float sal = 0.0f;
```

```
    try {
```

|| use JDBC DataSource obj to get JDBC connection from pool

Connection con = ds.getConnection();

// write JDBC persistence logic

```
Statement st = con.createStatement();
ResultSet rs = st.executeQuery("Select sal from emp
                                where empno = " + empno);
if (rs.next())
```

```
    sal = rs.getFloat(1);
```

// close JDBC objects

```
    rs.close();
```

```
    st.close();
```

```
    con.close();
```

```
}
```

```
Catch (Exception e)
{
```

```
    e.printStackTrace();
```

```
}
```

```
return sal;
```

```
}
```

```
public String fetchName (int empno)
```

```
{
```

```
    String name = null;
```

```
try {
```

```
    Connection con = ds.getConnection();
```

```
    Statement st = con.createStatement();
```

```
    ResultSet rs = st.executeQuery("Select ename from
```

```
        emp where empno = " + empno);
```

```
    if (rs.next())
```

```
        name = rs.getString(1);
```

```
    } // rs.close(); st.close(); con.close();
```

```
} // Catch (Exception e) { e.printStackTrace(); }
```

```
return name;
```

```
}
```

## Spring(fg.xml) :-

```
<!DOCTYPE . . . .>

<beans>
    <bean id="drds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName"><value>oracle.jdbc.driver.OracleDriver</value></property>
        <property name="url" value="oracle:thin:@localhost:1521:orcl"/>
        <property name="username" value="scott"/>
        <property name="password" value="tiger"/>
    </bean>
    <bean id="dsb" class="DBOperationBean">
        <property name="ds"><ref bean="drds"/></property>
        <!-- <property name="ds" ref="drds" / -->
    </bean>
</beans>
```

Predefined  
class supplied  
by Spring API  
(non POJO)  
(configured as  
Spring Bean)

Our Spring  
Bean class  
configuration  
config(pojo)

Our Spring Bean class property is  
injected with JDBCDataSource obj  
given by DriverManagerDataSource class

→ This application deals with the SpringContainer created and  
Managed JDBC Connection pool.

## DBClient.java :-

```
import org.springframework.core.io.*;
import org.springframework.beans.factory.xml.*;

public class DBClient
{
    public void main(String args[])
    throws Exception
```

|| Activate Container by loading spring cfg file

```
fileSystemResource res = new fileSystemResource ("spring.cfg.xml");  
XmlBeanFactory factory = new XmlBeanFactory (res);
```

|| get our spring bean class obj

```
DBOperation bob = (DBoperation) factory.getBean ("dbob");
```

|| Call business methods

```
s.o.p ("Employee salary is :" + bob.fetchSalary(4225));
```

```
s.o.p ("Employee name is :" + bob.fetchName("Vishal"));
```

```
}
```

Jar files:-

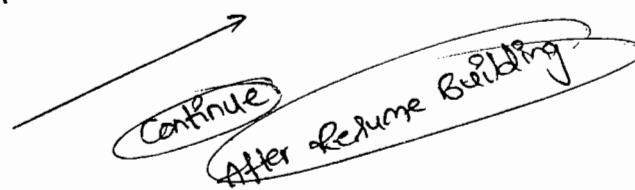
spring.jar

common-logging.jar

dbclass.jar

\* When and where your Beanfactory container is activated  
and deactivated in the above spring examples?

Ans:- In Main() of client application when object is created for  
XmlBeanfactory class the SpringContainer (Beanfactory) will be  
activated. Once control reaches to end of the Main() the Beanfactory  
Container will be stopped.



02/01/11

## Resume Building

Resume: (10 - Bold)

Font : Times New Roman, Arial, Verdana

Size : 9 or 10

Name : 11 - Bold

Phone : 9, Email : 9

documents (Expt)

Employee Id

Salary slips

Bank statements

Form 16

Releasing letter

Consultancy: (fresh)

→ magna IF - Macintosh

HR  
Question: (Expt)

- 1) Why you are leaving that company?
- 2) After how many do you join?
- 3) CTC → 3rd levels  
(After 1-2 months)
- 4) Hike → 40 to 50%.

# RESUME



Java  
certification  
logos



NARAYANA REDDY KANALA  
Phone: +91-90303 67416  
Email: kndro2@gmail.com

## Career Objective:

## Achievements:-

1. Academic ranks related.
2. Point on scip/other score
3. Point on paper presentation.
4. point on attending conferences (technical)

## Strengths:

self motivator

Good Team player

## Academic profile:

Stream	Institution / University / Board	Year	Performance
MCA	Osmania University	2008-11	70%.
B.Sc(che)	Alagappa University	2005-08	84%.
10+2	Board of Intermediate	2003-05	88%.
10th	Z.P. High School (C.S.C)	2002-03	75%.

## Technical profile:

programming Languages : C, C++, Java

Operating Systems : windows

Main Academic Subjects:

→ DBMS, SE, COSD

(End of page 1)

Project Profile:

Project #1 (As my final sem project) (Feb 2011 to June 2011)

Title: System to monitor Gini schemes

Client org: Product / Company name.

Environment: - same as xp -

Team size: 1

Role: Involved everywhere

Description:

System (Gini)

Personal profile:

Name: Nagayana Reddy Kanala

DOB: 1st May 1988

Alternate mail: nroo2@yahoo.com

Alternate Contact no: +91-97031 98472

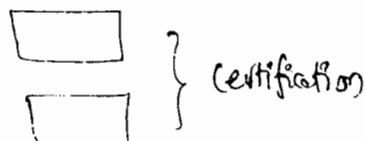
Passport no: 4252425

Language known: English, Telugu, Hindi

(End of page 2)

(End),

# CURRICULUM VITAE



NARAYANA REDDY KANALA  
Phone: +91-90803 67416  
Email: Enron2@gmail.com

Career Objective: (Optional)

Experience Summary:

- Having 4+ year experience on Java, JEE and frameworks
- hands on experience to work with struts, spring, hibernate ..
- Ability to implement design patterns
- Expertised to work with servers like weblogic, glassfish, ..
- Good Team player , self motivator

Achievements:

- Secured 85% in CCJP Exam,
- Awarded as Best Employee in the year 2010.
- Received appreciation letters from PL
- Point on attending conferences

Experience profile:

- Working as senior Tech. consultant for Wipro from Jan 2010 to Jul 2011 date
- Worked as programmer for Infostis from Aug 2009 to Dec 2010

Technical profile

Skill	Experienced	Awareness
Programming lang's	Java	c, c++
Operation systems	Windows	Unix
Database db's	Oracle	mysql
Web technologies	HTML, JavaScript	AJAX
JSE concepts	aut, swing, plaf, swingx	Swing
JEE concepts	servlets, JSP's	EJB

frameworks	struts, spring	jsp
ORM slw's	hibernate	-
servers	TOMCAT, weblogic	glass, glassfish
SOA	web services	-
EDE's.	MyEclipse, NetBeans	-
Logging tool	Log4j	-
Built tool	ANT	Maven
CVE tool	CVS NT	-
Unit testing	JUNIT	-
Report Generation	Jasper Reports	-

(End of the first page)

Project's profile :

Project #1

(for TCS)

(from Jan 2010 to Till date)

Title : openfx

Client org : Citi Banks / Intance / Product

Environment : Glass 1.5, Struts 1.3.8, JSP 2.0, servlets 4, Tomcat 5.5  
NetBeans 6.7,

Team size : 8

Duration : 1+ years

Role : programmer

Description :

===== } 6th 7 lines (keep less, speaks more)  
===== } 8th 9 lines (justify)

Responsibilities :

- Involved in user interface designing
- Participated in integration of project
- done coding for "PayRoll" module
- Implemented design patterns
- performed unit testing in using Junit tool

end lesson

12/09/11

Spring config file gives diff tags to cfg dependent values  
to diff type of Bean properties.

<u>Bean property type</u>	<u>tag name</u>
Simple data type ( int, float... )	<values>
java.lang.String	<values>
Bean class ( ref-type)	<refs>
java.util.List	<list>
array	<list>
java.util.Map	<maps>
java.util.set	<sets>
java.util.Properties	<props>
To set "null" value at Beanproperty	<null/>

each element of regular Map data structure like HashTable or  
or HashMap can take any object as key and any object as element.  
where as the element of java.util.Properties map data structure  
allows only string as key and string as value.

MyEclipse :-

Type : IDE for Java environment

Version : 8.x Compatible with JDK 1.6

Vendor : Eclipse.org

It is a commercial sw, allows to configure any external server  
with IDE sw. To download [www.myeclipseide.com](http://www.myeclipseide.com).

A plugin is a patch sw or sw application that can enhance  
functionalities of existing sw or sw applications.

MyEclipse, Eclipse Galileo, -eclipse are the advanced  
IDE's that are developed based on Eclipse IDE.

\* procedure to develop spring app by using myeclipse 8.x to demonstrate <sup>(setter)</sup> dependency injection on diff types of Bean properties:

Step①: Launch MyEclipse IDE by choosing a workspace folder.

Step②: Create Java project in MyEclipse IDE

file Menu → New → Java project → project Name: [MyProject] →  
Next → finish.

Step③: Add spring Capabilities to the project.

right click on project (MyProject) → MyEclipse → Add Spring Capabilities  
→  Spring 2.5 →  select spring as core libraries → Next →  
Bean Configuration file: [beans.xml] → finish

Step④: Add Spring Interface, Spring Bean class to project as shown below.

→ right click on project → New → Interface → Name: [Demo] → finish  
Demo.java :-

```
public interface Demo
{
    public String sayHello (String uname);
}
```

→ Right click on project → class → Name: [DemoBean] → Interfaces →  
add → choose Demo Interface → class → or → finish

DemoBean.java :-

(Ctrl + Shift + O → package import)

```
import java.util.*;
```

```
public class DemoBean implements Demo
```

```
{ int age;
```

```
    String name;
```

```
    Date bd; // birthday
```

```
    int marks[];
```

```
    List fruits;
```

```
    Set phones;
```

```
    Map m;
```

properties faculties;

// write setXXX() supporting SetterInjection

} select above bean properties → Right Click →  
Source code → generate getters and setters →  
select setters → OK (gives 8 setXXX())

public String sayHello(String uname)

```
{  
    return "Good Morning :" + uname +  
           " age = " + age +  
           " name = " + name +  
           " d = " + d.toString() +  
           " marks = " + marks[0] + " ... " + marks[1] +  
           " fruits = " + fruits.toString() +  
           " phones = " + phones.toString() +  
           " capitals = " + capitals.toString() +  
           " faculties = " + faculties.toString();  
}
```

Step ①: Add following content inside the democfg.xml (In beans)

→

```
<bean id="dt" class="java.util.Date"/>  
<bean id="ds" class="DemoBean">  
    <property name="age" value="25"/>  
    <property name="name" value="Nani"/>  
    <property name="d" ref="dt"/>  
    <property name="marks">  
        <list>  
            <value>80</value>  
            <value>90</value>  
        </list>  
</property>
```

```
<property name = "fruits">
  <list>
    <value>apple</value>
    <value>banana</value>
    <ref bean = "dt"/>
  </list>
</property>

<property name = "phones">
  <set>
    <value>9030367116</value>
    <value>00065538968</value>
    <ref bean = "dt"/>
  </set>
</property>

<property name = "capitals">
  <maps>
    <entry>
      <key><value>India</value></key>
      <value>New Delhi</value>
    </entry>
    <entry>
      <key><value>China</value></key>
      <value>Beijing</value>
    </entry>
    <entry>
      <key><ref bean = "dt"/></key>
      <ref bean = "dt"/>
    </entry>
  <maps>
</property>
```

```
<property name="faculties">  
    <props>  
        <prop key="mataraz">Java faculty</prop>  
        <prop key="prakash">.Net faculty</prop>  
        <prop key="Kamini Reddy">Java faculty</prop>  
    </props>  
    </property>  
</beans>  
</beans>
```

Step⑥: Add Client Application to the project.

Right click on project → New → class → Name: (DemoClient) →  
Select main() → finish

DemoClient.java:

[sysout + ctrl + space → sop]

```
public class DemoClient {  
    public static void main (String args) {  
        FileSystemResource res = new FileSystemResource (  
            ".\\src\\DemoCfg.xml");  
        XmlBeanFactory factory = new XmlBeanFactory (res);  
        Demo dsl = (Demo) factory.getBean ("ds");  
        System.out.println (dsl.sayHello ("Reddy"));  
    }  
}
```

Step⑦: Run the client application

Right click on the source code of DemoClient.java → Run as →  
Java Application.

13/09/14:

In predefined org. sf. Jdbc. dataSource. DriverManagerDataSource bean there is a java.util.Properties type Bean property whose name is connectionProperties.

To configure dependent values in this bean property we use <props> tag as shown below.

In spring.cfg.xml:

```
<bean id="drcls" class="org.springframework.jdbc.DataSource">
    <property name="driverClassName"><value>oracle.jdbc.driver.OracleDriver</value></property>
    <property name="url"><value>jdbc:oracle:thin:@</value></property>
    <property name="connectionProperties">
        <props>
            <prop key="user">scott</prop>
            <prop key="password">tiger</prop>
        </props>
    </property>
</bean>
```

\* The Bean properties of SpringBean class can be taken as static or non static member variables but the setter methods taken for Setter Injection must not be taken as static methods.

Q:- Can I place only parameterized constructors in SpringBean class when all Bean properties of that SpringBean class are configured only for Setter Injection.

A2:- Not possible, because in the above said situation the Spring Container uses zero-arg constructor to create SpringBean class object. Since that constructor is not available the Spring Container fails to do

Spring bean class must have either explicitly placed zero-arg constructor or Java compiler generated default zero-arg constructor in the above said situation.

→ If Java class is having explicitly placed constructors then Java ~~does~~ compiler will not generate the default zero-arg constructor.

## Understanding Constructor Injection :

In Constructor Injection, the Spring Container uses parameterized constructor to create Spring Bean class object and to assign / inject values to Bean properties.

Based on the no. of times that we have configured <constructor-args> under Bean tag in springConfiguration file an appropriate parameterized constructor will be picked up to perform constructor injection.

Ex: If <constructor-args> is written for 3 times under <beans> then  
Spring Container uses 3-param constructor to perform Ref ID Construction.  
Injection on Bean properties.

Ex: Demo. Prob: -

## Public Interface Demo

```
{ public String sayHello();  
}
```

## DemoBean.java :-

Import Java Util.\*;

public class DemoBean implements Demo

private Bring me;

private first age;

private.flat.org

### Role of the parameterized Contracts for contractor Selection.

```
public DemoBean( String msg, int age, float avg)
```

```
{
```

```
    System.out.println("3-param constructor");
```

```
    this.msg = msg;
```

```
    this.age = age;
```

```
}    this.avg = avg;
```

```
public DemoBean( int age, float avg)
```

```
{
```

```
    System.out.println("2-param constructor");
```

```
    this.age = age;
```

```
}    this.avg = avg;
```

```
public DemoBean( float avg)
```

```
{
```

```
    System.out.println("1-param constructor");
```

```
}    this.avg = avg;
```

```
public String sayHello()
```

```
{
```

```
    return "Good Morning" + "msg=" + msg +
```

```
}        "age=" + age + "avg=" + avg;
```

```
}
```

DemoCfg.xml:-

```
<beans>
```

```
    <bean id="obj" class="DemoBean">
```

```
        <constructor-arg><value>Hello</value></constructor-arg>
```

```
        <constructor-arg><value>24</value></constructor-arg>
```

```
        <constructor-arg><value>12.34</value></constructor-arg>
```

```
</beans>
```

```
</beans>
```

Note: Based on this code Spring Container uses 3-param Constructor while

## DemoClient.java :-

Same as first Application but call sayHello() business method instead of other business methods.

Note: Generally we configure dependent values to bean properties in constructor injection by specifying the values in the parameters order of parameterized constructor.

In spring config file while performing Constructor Injection configurations we can identify the parameters of constructors either based on their type or index.

Resolving parameters based on their type:

e.g.: Demo.java, DemoClient.java are same as previous application.  
DemoBean.java:-

```
public class DemoBean implements Demo
{
    private String msg;
    private int age;
    private float avg;

    public DemoBean( String msg, int age, float avg)
    {
        System.out.println("3-param constructor");
        this.msg = msg;
        this.age = age;
        this.avg = avg;
    }

    public String sayHello() { return "Goodit"; }
}
```

```
<beans>
    <bean id="ds" class="DemoBean">
        <constructor-arg type="int" value="30"/>
        <constructor-arg type="float" value="36.78"/>
        <constructor-arg type="java.lang.String" value="Hello"/>
    </bean>
```

If multiple parameters are having same datatype in constructor then they can be resolved through their index ('0' based index).

Example app to resolve/identify parameters based on their index:-

Demo.java, DemoClient.java are same as previous application.

DemoBean.java:-

```
public class DemoBean implements Demo
{
    private int a, b, c;

    public DemoBean(int a, int b, int c)
    {
        System.out.println("3-param constructor");
        this.a = a;
        this.b = b;
        this.c = c;
    }

    public String sayHello()
    {
        return "a=" + a + "b=" + b + "c=" + c;
    }
}
```

DemoCfg.xml:-

```
<beans>
    <bean id="db" class="DemoBean">
        <con-arg index="0"><v>30</v></con-arg>
        <con-arg index="1"><v>60</v></con-arg>
        <con-arg index="2"><v>90</v></con-arg>
    </beans>
</beans>
```

Note:-

If there is single Bean property in spring bean class then go for Constructor Injection coz constructor executes before setter method and it always bit faster when compared to Setter Injection.

If springBean class is having multiple Bean properties then go for SetterInjection bcoz it reduces burden on the programmer.

Note It is not mandatory that every bean property of SpringBean class must be configured for Dependency Injection.

Ex:- If there are 4 Bean properties in SpringBean class to satisfy SetterInjection in any angle (`setXXX()`) are enough. But we need to ~~16~~ (24) no. of constructors to satisfy Constructor Injection in all angles. So SetterInjection is preferred compared to Constructor Injection when SpringBean class is having more than one Bean property.

Why: Most of the predefined classes in Spring API are given supporting SetterInjection because they generally contain multiple Bean properties.

To add more Spring libraries to existing Spring project of MyEclipse IDE the procedure is :

Right click on the project  $\rightarrow$  BuildPath  $\rightarrow$  Add Libraries  $\rightarrow$  MyEclipse Libraries  $\rightarrow$  Next  $\rightarrow$  Select more Spring libraries  $\rightarrow$  Finish.

Example application on Constructor Injection to inject dependent values on diff types of Bean properties :-

Demo.java :- same as Sep 12<sup>th</sup> example

DemoBean.java :- same as Sep 12<sup>th</sup> example, but Replace 8 setter methods with one 8-param constructor as shown below.

```
public DemoBean( int age, String name, Date d, String[] marks,
    List fruits, Set phones, Map capitals, Properties faculties )
{ this.age = age }
```

Note: To generate the above parameterized constructors through my eclipse IDE select 8 bean properties of DemoBean class and Right click → Source → Generate Constructor Using fields

### DemoCfg.xml:-

Same as Sept 1<sup>st</sup> example but replace Complete <property> (including attribute) with <constructor-args> tag. Similarly replace </property> tag with </constructor-args> tag.

### DemoClient.java :-

Same as Sept 1<sup>st</sup> example.

We can locate Spring Config file for Beanfactory container either by using filesystem Resource class or Classpath Resource class.

The filesystemResource class locates given Spring Config file in the specified path of Computer filesystem. (In all files and directories) We can specify either Absolute path or Relative path while working with this class.

Ex: filesystemResource res = new filesystemResource ("demoCfg.xml");

- Locates demoCfg.xml from current directory. ↓  
relative path

filesystemResource res = new filesystemResource ("..\\demoCfg.xml"); ↑

- Locates demoCfg.xml from Parent directory →  
absolute path

filesystemResource res = new filesystemResource ("E:\\asc\\demoCfg.xml");

- Locates demoCfg.xml from E:\asc folder

The classpathResource class locates the given config file from the directories or jar files that are added to classpath environment variable.

directory based: If our demoCfg.xml is located in E:/APPS folder then add apps E:/APPS folder to environmental variable classpath.

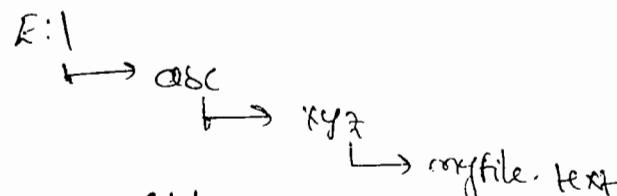
→ ClassPathResource res = new ClassPathResource("DemoCfg.xml");

jar file based: OR create a jar file in E:/APPS folder like  
E:/APPS > jar cf Test.jar • ↴

Then add this jar file E:/APPS/Test.jar to classpath.

What is the diff b/w Absolute path and Relative path?

Ans:- original resource file is like:



1) Current works folder is: e:\abc\xyz

The Absolute path of myfile.txt is : e:\abc\xyz\myfile.txt

The Relative path of myfile.txt is : ..\xyz\myfile.txt

2) Current works folder is: e:\abc\xyz\demo folder

The Absolute path of myfile.txt is : e:\abc\xyz\myfile.txt

The Relative path of myfile.txt is : ..\myfile.txt

If you refer location of certain file with respect to the location of current working folder then it is called relative path.

If you refer certain file (location completely) right from its root directory then it is called as absolute path.

Absolute path never change based on current directory only relative path changes.

15/09/14

A method in a Java class that is capable of constructing and returning its own Java class obj is called as factory method.

Ex: `Class c = Class.forName("Test");`

`Thread t = Thread.currentThread();`

`String s = String.valueOf(10);`

factory method is very useful to create the object of Java class outside of that class when that class is having only private constructors.

Ex: `java.lang.Class` is having only one private constructor, so we always use the factory method "forName()" to create object of `java.lang.Class` in our applications. (outside the `java.lang.Class` definition).

The factory method of Concrete class generally returns its own Java class object.

The factory method of Abstract class returns (creates) one of its subclass object.

In Java we can't create objects for Interfaces and Abstract classes by any angle means directly or indirectly.

When Spring Container cannot use / cannot access constructors of given Spring Bean class to create Bean class obj then we can make Spring Container creating that Bean class obj through factory method. for this we need to use `*factory-method*` attribute of ~~<bean>~~ tag in spring-configuration file.

Ex:-

DemoConfig.xml :-

```
<beans>
    <bean id="c1" class="java.lang.String"
          factory-method="valueOf">
        <constructor-arg> <value>java.lang.Integer</value></constructor-arg>
    </bean>
    <bean id="s1" class="java.lang.String"
          factory-method="getinstance">
        <constructor-arg> <value>10</value></constructor-arg>
    </bean>
    <bean id="cal" class="java.util.GregorianCalendar"
          factory-method="getinstance"/>
</beans>
```

Here `valueOf` returns `java.lang.String` object

`getinstance` returns `java.util.GregorianCalendar` class object

`getinstance` returns `java.util.GregorianCalendar` class object which  
is the subclass of `java.util.Calendar`.

`<constructor-arg>` is not for constructor injection, it supplies  
argument value to the factory methods.

DemoClient.java :-

```
public class DemoClient {
    public static void main(String args[]) throws Exception {
        ClasspathResource res = new ClasspathResource("DemoConfig");
        XMLBeanFactory factory = new XMLBeanFactory(res);
        Class<?> clazz = (Class<?>) factory.getBean("c1");
        System.out.println("Cost data is " + clazz.toString());
        System.out.println("Cost class name is " + clazz.getCanonicalName());
```

```

String sobj = (String) factory.getBean("s1");
System.out.println("sobj data is :" + sobj);
System.out.println("sobj class name is :" + sobj.getClass());
}

Calendar cal = (Calendar) factory.getBean("cal");
System.out.println("cal data is :" + cal.toString());
System.out.println("cal class name is :" + cal.getClass());
}
}

```

The Java class that allows to create only one object per JVM is called as singleton Java class. That means even though some server or container like JRE tries to create multiple objects for that class it will allow to create only one object.

for a normal Java class which actually allows to create multiple Java objects if container like server or JRE is creating only one object and not interested to create other multiple objects then that Java class will not be called as singleton Java class.

our servlet program class normal Java class (will not be designed as singleton Java class by programmer) but servlet container always creates and uses only one object of that class.

It doesn't mean programmers develop servlet program related classes as singleton Java classes.

Note: The classes which acts as servlet programs are not at all singleton Java classes.

we can make Spring Container keeping spring bean class which is our class.

- Those are
- |                        |   |  |
|------------------------|---|--|
| 1. singleton (default) | { | Applicable for standalone<br>and web environment of<br>Spring apps |
| 2. prototype           |   |  |
| 3. request             |   |  |
| 4. session             |   | }  |

Singleton: Spring container creates only one object for SpringBean class even though factory.getBean() is called for multiple times with same Bean Id.

We can use scope attribute in <beans> tag to specify the scope of SpringBean class object.

```
<bean id="d5" class="DemoBean" scope="singleton">
```

When SpringBean scope is taken as Singleton, the programmer or container never makes SpringBean class as Singleton Java class. It is just making Spring container to create only one object for SpringBean class per JVM.

Prototype: Spring container creates separate object for SpringBean class for each factory.getBean() call. That means factory.getBean() is called for 3 times, then Spring Container creates 3 no. of objects for SpringBean class objects.

```
<bean id="d6" class="DemoBean" scope="prototype">
```

Request

Request scope means SpringBean class object will be stored as Request attribute value in web application.

Session scope means the SpringBean class object will be stored as Session attribute value in web application.

## ApplicationContext Container:-

ApplicationContext Container is extension of Beanfactory Container. It gives support for the following Beanfactory Container and also gives the following additional features.

1. Ability to work with multiple Spring configuration files.
2. Instantiation of Spring Beans.
3. Support to read Bean property values from properties file.
4. Support for Internationalization.
5. Support for event handling on Spring Container through event listeners and etc..

Activating ApplicationContext Container is nothing but creating object for a Java class that implements org.springframework.context.ApplicationContext Interface (this interface is the sub interface of org.springframework.beans.factory.BeanFactory Interface)

ApplicationContext container also supports all the three modes dependency injections.

There are three popular implementation classes for this ApplicationContext Interface. So we can use one of these classes to activate ApplicationContext container.

1. org.springframework.context.support.FileSystemXmlApplicationContext
  - Activates ApplicationContext container by locating given configuration file from the specified path of file system.

FileSystemXmlApplicationContext ctx = new FileSystemXmlApplicationContext

## 2) org.springframework.context.support.ClassPathXmlApplicationContext

- Activates ApplicationContext Container by locating the given spring config file from the directories or jar files that are added to classpath environment variable.

## 3) org.springframework.web.context.support.XmlWebApplicationContext

- Useful to activate ApplicationContext Container in web application environment.

Executing first Spring app with ApplicationContext Container:-

Demo.java, DemoBean.java, DemoCfg.xml are some of 1st app.  
DemoClient.java :-

```
import org.springframework.context.*;  
  
public class DemoClient  
{  
    public void main(String args[]) throws Exception  
    {  
        // Activated ApplicationContext container  
        FileSystemXmlApplicationContext ctx = new  
            FileSystemXmlApplicationContext("DemoConfig.xml");  
        // get Access to spring bean class obj from Container  
        Demo bob = (Demo) ctx.getBean("db");  
        // call b. methods  
        System.out.println("Wish message is :" + bob.generateWishMsg("Nani"));  
        System.out.println("Factorial value is :" + bob.findFactorial(5));  
    }  
}
```

Output :--> the output is < "msg" > when i run it

~~Beanfactory~~ Container cannot take multiple SpringConfig files, whereas ApplicationContext container can work with multiple Spring Configuration files, as shown below.

ApplicationContext container is supplied in springContext/JEE module, yet it can be activated in any spring module based application.

working with multiple Spring Configuration files:-

If spring project is having multiple modules then we need to deal with multiple Spring Configuration files, In that situation Application Context container is very useful.

Demo.java :-

```
public interface Demo  
{  
    public String sayHello();  
}
```

DemoBean.java :-

```
import java.util.*;  
  
public class DemoBean implements Demo  
{  
    Date d;  
    Calendar cal;  
    // setter methods, business method write here  
}
```

DemoCfg.xml :-

```
<beans>  
    <bean id="dt" class="java.util.Date">  
        <property name="year"><value>100</value></property>  
</beans>
```

DemoCfg1.xml :-

```
<beans>
  <bean id="calen" class="java.util.GregorianCalendar"
        factory-method="getInstance"/>
</beans>
```

DemoCfg2.xml :-

```
<beans>
  <bean id="db" class="DemoBean">
    <property name="d" ref="dt"/>
    <property name="cal" ref="calen"/>
  </beans>
</beans>
```

DemoClient.java :-

```
import org.springframework.context.support.*;
public class DemoClient {
  public void main(String args[]) throws Exception {
    String cfg[] = {"DemoCfg1.xml", "DemoCfg2.xml",
                    "DemoCfg3.xml"};
    FileSystemXmlApplicationContext ctx =
      new FileSystemXmlApplicationContext(cfg);
    Demo db = (Demo) ctx.getBean("db");
    System.out.println("Result is: " + db.sayHello());
  }
}
```

13/03

Regarding Spring Container creating SpringBean class objects automatically when container is activated is called as Pre-Instantiation of Spring Beans.

- Application Context container can perform preinstantiation only on "singleton scope" Spring Beans.

Beanfactory container cannot perform preinstantiation on Spring Beans.

factory.getBean("db") → returns SpringBean class object by locating or creating that object, but doesn't create SpringBean class object through preinstantiation process.

ctx.getBean("ds") → Returns SpringBean class object by creating or locating object but creates SpringBean class object through preinstantiation process if SpringBean scope is singleton.

F&Q: There are ten beans in config file, so you tell me what can be done to make Spring container performing preinstantiation only on first five beans.

Ans:

→ Use Application Context Container.

→ Take Singleton scope for first five beans and prototype scope for remaining last five beans.

→ Make sure that the beans' properties of first five bean classes are not using last five SpringBean class objects.

F&Q →

```
<beans>
```

```
  <bean id="t1" class="TestBean" scope="prototype"/>
```

```
  <bean id="ds" class="DataSource" scope="singleton">
```

```
    <property name="tb"> <ref bean="t1"/> </property>
```

In the above scenario given in SpringConfig file can you tell me when the ApplicationContext container creates TestBean class object?

Ans:- Since TestBean class object ( Bean Id)(t1) is configured as dependent value to a Singleton scope DemoBean class property (t1) The ApplicationContext container also creates TestBean class object through (preInstantiation) process , but it does not change TestBean scope to singleton from prototype.

How to make ApplicationContext container Instantiating prototype scoped SpringBean through preInstantiation process?

Ans:- Configure ~~below~~ that Bean class Bean id as dependent value to Singleton scope SpringBean related Bean property as shown in the above question related <sup>Spring</sup> config file.

Properties file is a text file that maintains the entries in the form of key = value pairs.

Ex Myfile.properties

# test file

my.name = Nani

my.age = 24

Note! PreInstantiation of Beans is not that much useful in Standalone based Spring applications. But it is very useful in web environment based Spring applications.

Using this concept of preInstantiation we can make SpringContainer creating SpringBean class objects either during server startup or during deployment of web application.

The standard principle in the industry is don't hardcode any values to your applications that will be changed in future. Moreover pull them to application from outside the application by taking the support of properties files.

We generally make JDBC driver details, db user, username and password details as flexible details to modify by giving them to JDBC application from outside the app through properties files.

Beanfactory container doesn't support properties files whereas ApplicationContext container supports them.

For security reasons the username and passwords of db will be changed at regular intervals.

To supply properties file values as Bean property values the Spring config file should have place holders, as shown below.

```
<property name="age"><values>${my.age}</values></property>
```

↓  
place holder which says value to  
Bean property age will be pulled and assigned  
By collecting from myage key of properties file.

To make ApplicationContext recognizing place holders and to specify the name of properties file always configure the following predefined Spring Bean class.

org.springframework.beans.factory.config.PropertyPlaceholderConfigurer  
For this we use two properties of our class

location → An single property file

Location → An single property file

In spring Config file:-

```
<bean id="propConfig" class="org.sf.beans.factory.config.  
PropertyPlaceholderConfigurer",  
<property name="location"><values>myfile1.properties</values></property>  
</bean>
```

The above code is for single property file configuration.

```
<bean id="propConfig" class="org.sf....">  
<property name="locations">  
<list>  
<values>myfile1.properties </values>  
<values>myfile2.properties </values>  
</list>  
</property>  
</bean>
```

The above code is for multiple properties file configurations

1st Oct 11.  
Procedure to add properties file support for spring app that  
is given on 10<sup>th</sup> of this month. (dataSource example).

Resources:

- 1. DBOperation.java
  - 2. DBOperationBean.java
  - 3. Spring Cfg. xml
  - 4. DBClient.java
  - 5. DBDetails.properties.
- } → same as 10<sup>th</sup> date example.

DBDetails.properties!

my.driver = oracle.jdbc.driver.OracleDriver

my.url = jdbc:oracle:thin:@localhost:1521:orcl

my.username = scott

my.password = tiger

## Spring cfg.xml :-

```
<beans>
    <bean id="propConfig" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="location"><value>DBDetails.properties</value></property>
    </bean>

    <bean id="drds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName"><value>$\{my.driver}</value></property>
        <property name="url"><value>$\{my.url}\</value></property>
        <property name="connectionProperties">
            <props>
                <prop key="user">$\{my.username}\</prop>
                <prop key="password">$\{my.password}\</prop>
            </props>
        </property>
    </bean>

    <bean id="dbot" class="DBOperationBean">
        <property name="dc" ref="drds"/>
    </bean>
</beans>
```

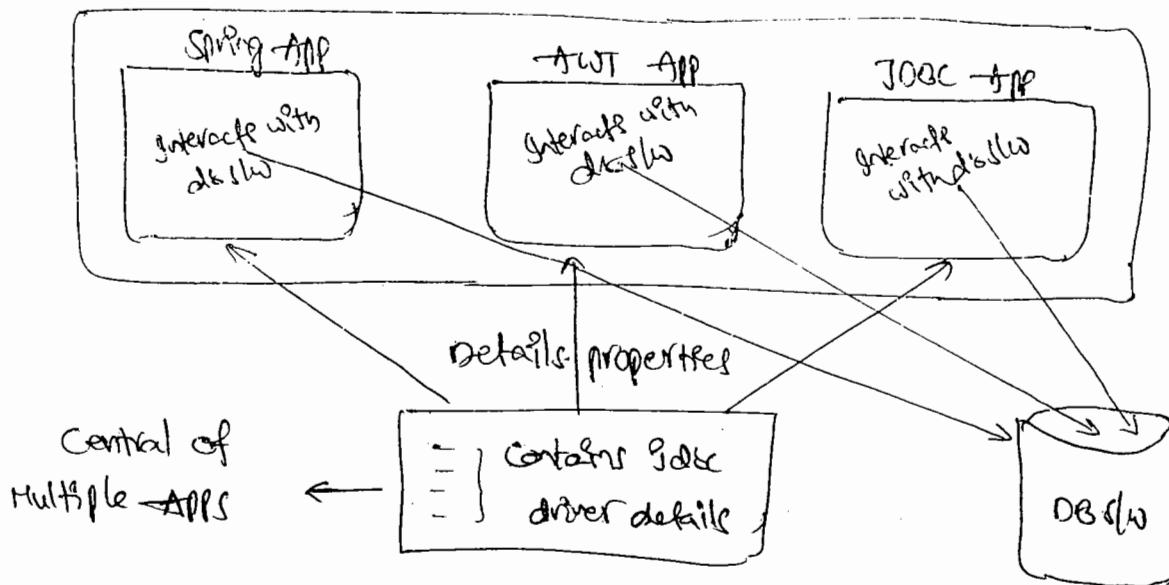
to make  
spring container  
to recognize  
and work  
with properties  
file and placeholders

## DBClient.java :-

same as sepiyah example, but ApplicationContext Container  
instead of beanfactory container

In spring applications properties file support is not just required to supply bean properties values, it is actually required to manage some common data (like JDBC details) as centralized data for multiple applications of projects/modules (for both spring and non spring app).

### Proj / Module of project



### Internationalization :-

Locale → Language + Country

ISO has given the following locale codes

en-US (English as it speaks in US)

en-UK

fr-FR (French as it speaks in France)

de-DE

Making our application working for multiple locales, multiple countries, multiple commercial based client organizations is nothing but enabling Internationalization (i18n) on the application.

IT Persian deal with only presentation (after) logic to render presentation content in diff languages and to show date, time, currency values in locale specific patterns.

Java is based Unicode characters having 65,535 characters. In this every language is having numbers range representing its alphabets and extensions. We can use Unicode editor (third party tool), Native ASCII (Built-in tool of JDK) to get Unicode numbers of any word or sentence belonging to any language.

Unicode editor tool can be downloaded from [www.hiragopi.com](http://www.hiragopi.com) website.

procedure to Getting Hindi words related Unicode range numbers By using Unicode editor, NativeASCII tools.

- Step① : Download Unicode Editor tool from [hiragopi.com](http://hiragopi.com) and install it.
- Step② : Launch Unicode editor tool (use index.html)
- Step③ : Type certain Hindi words in the editor area.

निकले	-	stop delete
संग्रहित	-	save
त्रैक्षणि	-	delete stop
संतुष्टि	-	cancel

- Step④ : Copy and paste words in a text file choosing encoding as Unicode. (like myword.txt)

Step⑤ : Use nativeascii to get unicode numbers for above Hindi words

E:/> nativeascii -encoding unicode myword.txt myfile.txt

Ex: myfile.txt

~~Object~~  
Internationalization  
Internationalization enabled on the application always deals with presentation logic of the application and it is noway responsible business logic and persistence logic of the application.

To enable Internationalization on Standalone Java applications use Locale Resource Bundle classes of java.util package.

Ex:-

Step①: Prepare multiple properties files including base file.

APP.properties :-

# Base file (for English)

str1 = delete

str2 = save

str3 = stop

str4 = cancel

APP-de-DE.properties :-

# for German

str1 = LÖSCHEN

str2 = DENN

str3 = ZUG

str4 = ABSCAGEN

APP-fr-CA.properties :-

# for french language

str1 = EFFACER

str2 = SAUVE

str3 = ARRÊT

str4 = ANNULER

APP-hi-IN.properties :-

# for Hindi language

str1 = लॉस्चेन्

str2 = देन्

str3 = जुग

str4 = अब्सागेन्

} collect from  
myfile.txt  
file

Note:- Keys must be same in all properties files.

We can use Google Translator or some language converters to get English equivalent words in other languages.

If no matching file is found the application automatically picks up the base properties file.

Step②:- Develop Application by enabling Internationalization as shown below

## I18nApp.java! (Save encoding style as Unicode)

```
import java.util.*;  
import javax.swing.*;  
import java.awt.*;  
  
public class I18nApp {  
    public (String args[]) {  
        // Create Locale object based on language, country code  
        Locale l = new Locale(args[0], args[1]);  
        // Pick up properties file based on the Locale object data  
        ResourceBundle r = ResourceBundle.getBundle("app", l);  
        // Create frame/window.  
        JFrame ff = new JFrame();  
        Container cf = ff.getContentPane();  
        // Create buttons by getting labels from properties file.  
        JButton b1 = new JButton(r.getString("str1"));  
        JButton b2 = new JButton(r.getString("str2"));  
        JButton b3 = new JButton(r.getString("str3"));  
        JButton b4 = new JButton(r.getString("str4"));  
        // Add button comps to frame window.  
        cf.setLayout(new flowLayout());  
        cf.add(b1);  
        cf.add(b2);  
        cf.add(b3);  
        cf.add(b4);  
        ff.pack();  
        ff.show();  
    }  
}
```

Note: Make sure that all the above properties and .java files are stored in a single folder.

Step ③: Compile and execute the application.

> javac -encoding unicode I18nApp.java

> java I18nApp de DE → for Germany

> java I18nApp x y → Base file will pickup, bcoz lang code, country code are wrong.

Note: ApplicationContext container support I18n, but to enable this we must configure one special predefined spring bean class called org.springframework.context.support.ResourceBundleMessageSource class in spring configuration file with multiple properties file names.

When I18n is enabled on Spring App we can use ctx.getMessage() method to read keys from the values of properties file based on the properties file that is activated.

Beanfactory container doesn't support I18n.

Example Application on I18n based Spring Environment :-

Step①:- prepare multiple locale specific properties files including base file.

Step②:- Develop the spring configuration file and client app as shown below.

beans.xml:-

fixed Bean id

```
<beans>
    <bean id="messageSource" class="org.springframework.context.support.RBMS">
        <property name="baseNames">
            <list>
                <value>APP</value>
                <value>APP-de-DE</value>
                <value>APP-fr-CA</value>
                <value>APP-hi-IN</value>
            </list>
        </property>
    </bean>
</beans>
```

## Demo Configuration :-

```
import org.springframework.context.support.*;  
import java.util.*;  
  
public class DemoClient {  
  
    public void sum(String args[]) throws Exception {  
        // Locale object  
  
        Locale l = new Locale(args[0], args[1]);  
  
        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext(  
            "demoConfig.xml");  
        // Activates the Spring Container.  
  
        // get msg from the activated properties file  
        String msg = ctx.getMessage("str3", null, "Default msg", l);  
        System.out.println("Message is :" + msg);  
    }  
}
```

\* ①:- Actually it is Java.lang.Object class Object() to supply  
in } argument values of messages that are kept in properties file.

\* ②:- If app fails to gather messages from properties file then  
the specified msg will be taken as default msg.

\* ③:- Java.util.Locale class object based on which an appropriate  
Locale specific properties file will be picked up.

Note :- The ApplicationContext container consumes few bytes of extra  
memory compare to Beanfactory. But bcoz of its features by  
all real time project this ApplicationContext container is used.

Moreover the ApplicationContext container gives better support to work  
with web module and services when compare to Beanfactory container.

Q Ans  
An action performed on the component or object is called as an event. Executing some logic when an event is raised is called as Event Handling. To handle events we used event listeners. These event listeners provide event handling methods for this operation.

In AWT, Swing level we perform event handling on GUI components like Button, Text boxes and etc.

In web environment we can perform event handling on Request, session, servletContext objects to notice their creation and destruction timings. In spring environment we can perform event handling on ApplicationContext container to notice its startup time and shutdown time.

To perform event handling we need the following details.

1. Source obj/Component
2. event
3. event listener
4. Event handling method.

AWT/Swing Example :-

Source Component → Button

Event → ActionEvent

Event Listener → ActionListener

Event Handling method → public void actionPerformed(ActionEvent e)

Web Environment Example :-

Source obj → ServletContext obj

Event → ServletContextEvent

Event Listener → ServletContextListener

Event Handling method → 1. public void contextInitialized(Event e) { } { }

(executes when servletContext obj is created)

2. public void contextDestroyed(Event e) { } { }

Spring Example :-

source obj → ApplicationContext container

Event → ApplicationEvent

EventListener → ApplicationListener

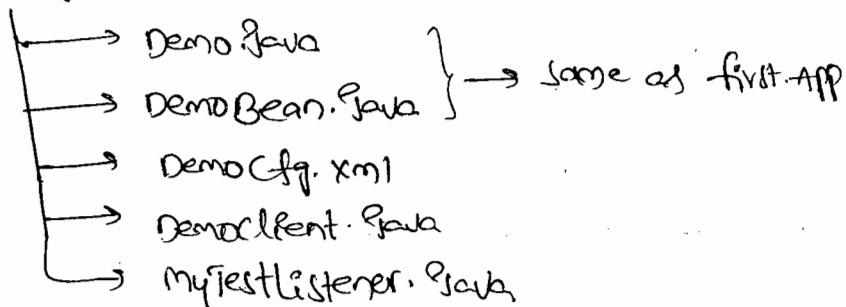
eventHandlingMethod → public void onApplicationEvent(ApplicationEvent ae)  
(executes when → App Context Container { ..... }  
is started or stopped.)

We cannot perform event handling on BeanFactory Container but  
possible with AppContext Container.

Without disturbing the existing Spring app we can perform  
event handling on AppContext Container by taking one separate  
Java class.

Example Application :-

E:\ Apps\Spring



DemoCfg.xml:-

```
<beans>
  <bean id="mt1" class="MyTestListener"/>
  <bean id="ds" class="DemoBean">
    <property name="msg"><values>Hello</values></property>
  </bean>
</beans>
```

MyTestListener.java →

```
import org.springframework.context.ApplicationListener;
public class MyTestListener implements ApplicationListener
```

```

    long stime, endtime;
    public void onApplicationEvent(ApplicationEvent ae)
    {
        System.out.println("onApplicationEvent (-)");
        if (ae.toString().indexOf("refreshed") == -1)
        {
            System.out.println("Hey... ApplicationContextContainer just started");
            stime = System.currentTimeMillis();
        }
        else if (ae.toString().indexOf("closed") != -1)
        {
            System.out.println("Oh... ApplicationContext Container just stopped");
            endtime = System.currentTimeMillis();
            System.out.println("Spring Container was there in running mode for"
                + (endtime - stime) + "ms");
        }
    }
}

```

### DemoClient.java:-

- same as first app but works with ApplicationContext container.
- stop/close ApplicationContext container explicitly at the end of the application by calling ctx.close().

→ Then compile and execute as usual.

→ filesystem XmlApplicationContext ctx = new FileSystemXmlApplicationContext("DemoCfg.xml");  
 → starts container

→ ctx.close() → stops container

Note: Event Handling on SpringContainer is very useful when ApplicationContext

Container is used in web environment.

we can't explicitly write BeanFactory Container.

## Injecting values to Bean properties through dependency

Injection is called as wiring. This can be done in two ways.

1. Explicit wiring → Configure dependent values to Bean properties explicitly.
2. Auto wiring → The Spring container dynamically detects and assigns dependent values to Bean properties without figuring them explicitly. Both Beanfactory, ApplicationContext Container support Autowiring.

Note :- The dependency injection configurations that we have performed so far comes under explicit wiring (Needs property, <constructor> utilizations explicitly).

In Auto wiring there is no need of working with <property>, <constructor> tags explicitly.

Autowiring is possible only on ref type bean properties (the Bean properties which can hold other bean class objects).

### Limitations with AutoWiring :-

- ① We cannot perform Auto wiring on primitive data type or string type bean properties which expect direct and simple string values.  
Ex: int age, String name, float avg etc.

- ② There is a possibility of getting Ambiguous state (ambiguous state) error when Spring Container finds multiple dependent objc for a single bean prop.

26/08/11

To know info about any XML tag or attributes of Spring config file refer its DTD file rules or schema file rules (xsd file)

In DTD file like spring-beans-2.0.dtd file <spring-beans>/list/references

<!ELEMENT <tagname> ... → for tag rules ex: <!ELEMENT bean

<ATT! T#T &tagname; n> <attribute names> ... → for attribute

Programmer can give instruction to Spring Container to perform auto wiring in diff modes, by using the autowire attribute of bean's tag.

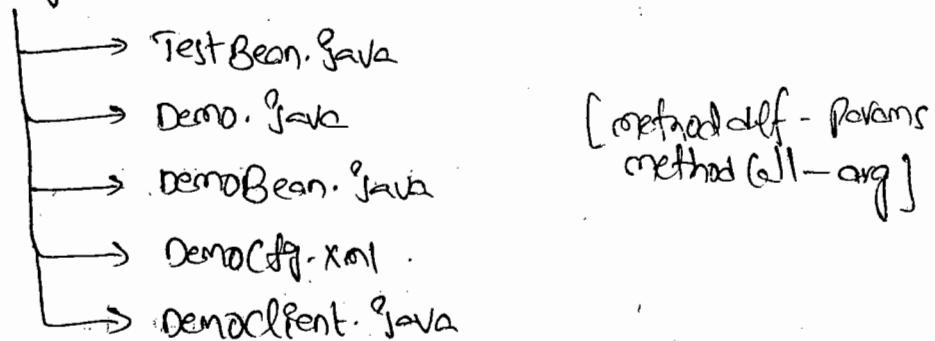
1. no → Indicates that this bean doesn't support autowiring.
2. byName → performs Setter Injection
3. byType → performs Setter Injection
4. constructor → performs Constructor Injection
5. autodetect → performs either byType or constructor based dependency injection.

② byName:- (autowire = "byName");-

Makes Spring Container to perform Setter Injection on bean properties. for this the bean property names and bean id names of dependent bean class obis must match.

Example:-

E:\apps\spring



TestBean.java:-

```
public class TestBean
```

```
{ private String msg;
```

```
public TestBean()
```

```
{ System.out.println("0-param Constructor of Test Bean"); }
```

```
public void setMsg( String msg )  
{  
    this.msg = msg;  
}  
  
public String toString()  
{  
    return "TestBean.msg=" + msg;  
}
```

Demo.java :-

```
public interface Demo  
{  
    public String sayHello();  
}
```

DemoBean.java :-

```
public class DemoBean implements Demo  
{  
    TestBean tb;  
  
    public DemoBean()  
    {  
        System.out.println("0-param Constructor of DemoBean");  
    }  
  
    public DemoBean( TestBean tb )  
    {  
        this.tb = tb;  
        System.out.println("1-param Constructor of DemoBean");  
    }  
  
    public void setTB( TestBean tb )  
    {  
        this.tb = tb;  
        System.out.println("DemoBean: setTB() method");  
    }  
  
    public String sayHello()  
    {  
        return "Good morning" + "tb=" + tb.toString();  
    }  
}
```

DemoCfg.xml :-

<beans>

<bean id="tb" class="TestBean" ></bean>  
(Based on this configuration the Spring Container  
Injects TestBean class obj (tb) to the "tb" property  
of DemoBean class through  
setter injection)

<property name="msg" ><value>Hello</value></property>  
</beans>

<bean id="db" class="DemoBean" autowire="ByName" />  
</beans>

DemoClient.java :-

(Here Bean property name tb and  
Bean id name tb are must match)

import org.springframework.context.\*;

public class DemoClient

{ public (String args[])

{ FileSystemXmlApplicationContext ctx = new

FileSystemXmlApplicationContext ("DemoCfg.xml");

(OR) → XmlBeanFactory factory = new XmlBeanFactory (new

FileInputStreamSource ("DemoCfg.xml");

Demo bobj = (Demo) ctx.getBean ("db");

System.out.println ("Result = " + bobj.sayHello());

}

③ byType :— Spring Container performs autowiring on bean property  
if there is exactly one Bean property class obj of the Bean property  
type in spring cfg file. If there is more than one Bean objects  
then Spring Container raises "fatal error" (Ambiguity problem)  
and you cannot perform byType mode of autowiring for that Bean.

This performs setterInjection on Bean properties.

Ex) DemoGfg.xml :-

```
<beans>
    <bean id="tb1" class="TestBean">
        <property name="msg"><values>Hello</values></property>
    </bean>
    <bean id="db" class="DemoBean" autowire="byType"/>
</beans>
```

Note: Here the property type of DemoBean class is matching with tb1 bean obj type (TestBean), so byType mode of autowiring is allowed.

The following code gives ambiguity error while dealing with byType mode of Autowiring. (Null pointer exception)

```
<beans>
    <bean id="tb1" class="TestBean">
        <p>name = "msg"><v>Hello </v></p>
    </bean>
    <bean id="tb2" class="TestBean">
        <p name="msg"><v>Hello </v></p>
    </bean>
    <bean id="db" class="DemoBean" autowire="byType"/>
</beans>
```

④ constructor :- Makes Spring Container to perform Constructor Injection through parameterized constructor of spring Bean class.

If more matching Bean objects are there as shown in the above XML code, there is a possibility of getting fatal error.

```
<beans>
    <bean id="db" class="DemoBean" autowire="constructor"/>
```

## ⑤ autodetect:-

If springBean class is having 0-param Constructor (Default) then springContainer performs "ByType" mode of Autowiring. otherwise spring container looks to perform Constructor Injection through parameterized Constructors by using "Constructor" mode of Autowiring.

Ex:- <beans>

```
<bean id="tb" class="TestBean">
    <p name="msg"><v>Hello</v></p>
</bean>
<bean id="ds" class="DemoBean" autowire="autodetect"/>
</beans>
```

Note:- Since Autowiring kills the readability of Spring Cfg file and gives other limitations in real time projects it is recommended to avoid autowiring and recommended to work with explicit wiring.

So the most suitable value for autowire attribute is "no".

(Should not allow autowiring)

Ex:-

```
<bean id="ds" class="DemoBean" autowire="no">
    <property name="tb"><ref bean="tb"/></property>
</bean>
```

We can configure both Explicit wiring and Autowiring on single Bean property but the wiring that perform Setter Injection will be effected.

If Both Explicit and Autowiring are there performing Setter Injection then Explicit wiring values will be effected.

Ex <beans> <bean id="tb" class="TestBean">
 <p name="msg"><v>Hello</v></p>
</bean> <bean id="t1" class="TestBean">
 <p name="msg"><v>Hello</v></p>
</bean>

```
<bean id="ds" class="demoBean" autowire="byName">  
    <property name="tb"> <ref bean="tb2"/> </property>  
</beans>  
</beans>
```

↓  
(Explicit wiring)

27/08/11

Every object contains some life cycle events in its life cycle when these events are raised life cycle methods will be executed automatically, so programmer keeps some Event Handling logic and life cycle operational related logic in these life cycle methods.

Programmer never calls life cycle methods explicitly. These methods will be called by underlying container automatically.

The methods that will be called by underlying container based on the events of life cycle that are fired on the OSI are called as life cycle methods or container callback methods.

Ex. 1. Applet life cycle methods are → init(), start(), paint(), stop() and destroy().

2. servlet life cycle methods are → init(-), public service(-,-); and destroy().

Spring Container allows the programmer to keep user-defined methods as life cycle methods, but they must specified/configured in spring.cfg file during spring Bean Configuration.

In SpringBean class the lifecycle method names are not fixed names, they are programer choice method names.

Applet Container (Applet Viewer or Browser Window) calls Ample lifecycle methods.

Servlet Container calls servlet program life cycle methods.

Spring Container calls the specified user defined methods as spring life cycle methods.

Spring Bean life cycle methods:-

### ① init-method (User defined)

- Spring Container calls this method immediately after Spring Bean instantiation and dependency injection.

- This method is useful to keep logic

i) That verifies whether Bean properties are injected with values or not.

ii) That verifies whether Bean properties are injected with appropriate values are not, like age must positive number

### ② destroy-method (User defined)

- Spring Container calls this method when it is about to destroy spring Bean class object.

- This method is useful to keep uninitialization logic like nullifying bean property values.

Example:-

Demo.java :- Same as previous App

DemoBean.java :-

```
import java.util.*;
```

```
public class DemoBean implements Demo
```

```
    int age;
```

```
    String name; // Name of your company
```

```
    Date dob; // Date of birth
```

```
    public DemoBean() {super("OrParam Constructor");}
```

```
public void setAge(int age)
{
    this.age = age;
    System.out.println("setAge() method");
}
```

```
public void setMsg(String msg)
{
    this.msg = msg;
    System.out.println("setMsg() method");
}
```

```
public void setDate(Date d)
{
    this.d = d;
    System.out.println("setDate() method");
}
```

// life cycle methods of Spring Bean

```
public void myInit() throws Exception
{
    System.out.println("myInit life cycle method");
    if(age <= 0 || msg == null || d == null)
        throw new Exception("Set values to Bean properties");
}
```

```
public void myDestroy()
{
    System.out.println("myDestroy life cycle method");
    age = 0;
    msg = null; // nullifying Bean properties
    d = null;
}
```

↳ public String sayHello() { return "Hello"; }

Democfg.xml:-

```
<beans> <bean id="ds" class="DemoBean" init-method="myInit"
    <property name="age" value="20" /> <!-- destroy-method="myDestroy" -->
    <property name="msg" value="Hello" />
    <property name="d" ref="dt" /> </bean>
<bean id="dt" class="java.util.Date" /> </beans>
```

DemoClient.java :-

```
public class DemoClient {
    public static void main(String args[]) throws Exception {
        XMLBeanFactory factory = new XMLBeanFactory(new FileInputStream("Hello.xml"));
        Demo bob = (Demo) factory.getBean("ds");
        System.out.println("Result" + bob.sayHello());
        factory.destroySingletons();
    }
}
```

In the above program, if the Container is ApplicationContext Container then it is same as previous example.

The only diff is we use (tx. close()) there.

→ Both ApplicationContext, Beanfactory Container support life cycle methods.  
we cannot design these life cycle methods with parameters.

Limitations of life cycle methods :-

1. If programmer forgets to configure life cycle method names in spring.cfg file their effect will not be there.
2. while working third party supplied Java classes, spring API supplied Java classes as SpringBeans, identifying life cycle method names from huge number of methods is complex or error-prone process.

To solve above two problems make your Spring Bean class implementing two special interfaces,

1. org.springframework.InitializingBean

2. org.springframework.DisposableBean, and get the effect of life cycle methods by implementing methods of these interfaces.

But here there is no need of configuring these methods named explicitly in Spring config file.

The org.springframework.InitializingBean interface contains afterPropertiesSet() → programmer implements this method having some logic of init life cycle method.

The org.springframework.DisposableBean interface contains

destroy() → programmer implements this method having some logic of destroy life cycle method.

When Spring Bean implements these two interfaces it becomes Non-POJO class.

Example :- DemoClient.java → same as previous

Demo.java → same as previous App.

DemoBean.java → same as previous but make DemoBean class implementing InitializingBean, DisposableBean Interfaces.

And replace myInit(), myDestroy() methods with afterPropertiesSet(), destroy() methods as shown below.

public void afterPropertiesSet() throws Exception

{  
if( age <= 0 || msg == null || d == null )

    throw new Exception("Set values to Bean properties");

public void destroy()

    age = 0;

    msg = null;

    d = null;

(init-method)

(destroy-method)

DemoConfig.java → same as previous, but no need to specify lifecycle methods.

Note: Most of the Spring API supplied predefined Bean classes are there implementing InitializingBean, DisposableBean interfaces instead of giving init(), destroy() lifecycle methods.

If you keep init() along with afterPropertiesSet() then the Spring Container executes init() lifecycle method after - afterPropertiesSet(). By destroy() lifecycle method executes after destroy() of DisposableBean interface.

\* \* \* Q: How did you use Abstract class and Interface & Real time?

Ans:- PL or TL designs project specification providing set of Rules and guidelines to Team members for implementation of project.

Interface methods, Abstract class - Abstract methods represents Rules, whereas abstract class concrete methods and Concrete class concrete methods are called as Guidelines.

If PL or TL wants to give only rules then he chooses Interfaces, if PL or TL wants to give both rules and guidelines then he chooses Abstract classes.

Every SW specification gives certain API (classes, & interfaces) having Rules and Guidelines to develop SW's.

Ex:- JDBC specification contains set of Rules and Guidelines to develop JDBC driver SW's.

Servlet specification contains set of Rules and guidelines to develop Servlet Container SW.

- The interfaces of servlet API (javax.servlet, javax.servlet.http.HttpServletRequest) represents rules of servlet specification.

## Interface Injection :-

Since life cycle methods, Initializing Bean, Disposable Bean Interface related methods are not capable of Injecting any data to bean properties either by collecting data from spring cfg file or from spring container, so we can't say these methods are performing dependency injection, In any container life cycle methods cannot perform dependency injection bcoz they cannot gather any external data.

If spring bean class implements special xxx-Aware Interfaces then spring container can inject special values to spring bean properties and this process is called as "Interface Injection".

These special values are like current Bean Id, underlying Beanfactory, ApplicationContext container and etc..

The xxx-Aware interfaces are :-

① org. st. beans. factory. BeanNameAware

- useful to inject / get current bean class BeanId Being from that bean class.

method → public void setBeanName(<sup>repeat BeanId</sup> string name).

② org. st. bean. factory. BeanFactoryAware

- useful to inject / get underlying Beanfactory container as dependent value to Bean class.

method → public void setBeanFactory(Beanfactory factory)

throws BeansException.

### ③ org. st. ~~beansfactory~~<sup>Context</sup>.ApplicationContextAware

- Useful to inject/get underlying Application Context Container as dependent value to Bean class.

method → public void setApplicationContext(ApplicationContext ctx)  
throws BeansException

Note! :-

Spring Beans use the injected underlying Spring Containers to know about other Spring Beans managed by that Container and to communicate with them by getting access to their objs.

eg/

Example app on Interface Injection to inject beanId and underlying AppContext Container:-

Demo.java :- same as previous App

DemoBean.java :-

```
import org.st. context.*;
import org.st. beans. factory.*;
```

public class DemoBean implements Demo, BeanNameAware,  
ApplicationContextAware

{ String bname;

ApplicationContext ctx;

II. Implement methods of BeanNameAware, ApplicationContextAware (Not setxxx)

public void setBeanName(String bname)

```
{ this.bname = bname;
}
```

public void setApplicationContext(ApplicationContext ctx)

```
{ this.ctx = ctx;
}
```

```

public String sayHello()
{
    System.out.println("Current Bean class BeanId is : " + beanName);
    System.out.println("Current container is managing " + ctx.getBeanDefinitionCount() + " Beans");
    System.out.println("Current container is managing the following Beans");
    String[] names = (ctx.getBeanDefinitionNames());
    for (int i = 0; i < names.length; ++i)
        System.out.println(names[i]); → gives id, dt, dt1
    System.out.println("Current Bean scope is Singleton ? " + ctx.isSingleton(beanName));
    System.out.println("Current Bean scope is prototype ? " + ctx.isPrototype(beanName));
    return "Good Morning " + beanName..."; false
}
}

```

(Here Both dt, dt1 are just demonstration purpose, not related with this)

DemoCfg.xml:-

```

<beans>
    <bean id="dt" class="DemoBean"/>
    <bean id="dt" class="java.util.Date"/>
    <bean id="dt1" class="java.util.Date"/>
</beans>

```

DemoClient.java:-

Same as previous, but works with ApplicationContext container

- \* If programmer wants to modify Bean property values after Injection and before using them in Business method then we need to work with `org.springframework.beans.factory.ConfigurableBeanPostProcessor` interface.

This allows to perform this activity for multiple Spring Beans managed by container. This interface is having two methods.

-----Demo.java-----

```
1 public interface Demo {  
2     public String sayHello();  
3 }  
4  
5 -----DemoBean.java-----  
6 import org.springframework.beans.factory.DisposableBean;  
7 import org.springframework.beans.factory.InitializingBean;  
8  
9 public class DemoBean implements Demo, InitializingBean, DisposableBean {  
10     String msg;  
11  
12     public void setMsg(String msg) {  
13         System.out.println("DemoBean:setMsg(-)");  
14         this.msg = msg;  
15     }  
16  
17     @Override  
18     public void afterPropertiesSet() throws Exception {  
19         System.out.println("DemoBean:afterPropertiesSet()");  
20         if(msg==null)  
21             throw new Exception("set value to msg ");  
22     }  
23     public void myInit()throws Exception  
24     {  
25         System.out.println("DemoBean:myInit()");  
26         if(msg==null)  
27             throw new Exception("set value to msg ");  
28     }  
29  
30     public void myDestroy()throws Exception  
31     {  
32         System.out.println("DemoBean:mydestroy()");  
33         msg=null;  
34     }  
35  
36     @Override  
37     public void destroy() throws Exception {  
38         System.out.println("DemoBean:destroy()");  
39         msg=null;  
40     }  
41  
42     @Override  
43     public String sayHello() {  
44         return "good morning"+msg+msg;  
45     }  
46  
47     @Override  
48     public String sayHello() {  
49         return "good morning"+msg+msg;  
50     }  
51  
52     }  
53  
54  
55  
56  
57  
58 }  
59  
60  
61 -----MyPostProcessor.java-----  
62 import org.springframework.beans.BeansException;  
63 import org.springframework.beans.factory.config.BeanPostProcessor;  
64  
65  
66 public class MyPostProcessor implements BeanPostProcessor {  
67  
68     @Override
```

```

69     public Object postProcessBeforeInitialization(Object bean, String beanName)
70         throws BeansException {
71     System.out.println("DemoBean:postProcessBeforeInitialization(-,-)");
72
73     // TODO Auto-generated method stub
74     return bean;
75 }
76
77
78     @Override
79     public Object postProcessAfterInitialization(Object bean, String beanName)
80         throws BeansException {
81     System.out.println("DemoBean:postProcessAfterInitialization(-,-)");
82     DemoBean obj=(DemoBean)bean;
83     if(obj.msg.length()<=3)
84         obj.msg=obj.msg+"satya";
85
86     return obj;
87 }
88
89
90 }
91
92
93 -----DemoCfg.xml-----
94
95 <?xml version="1.0" encoding="UTF-8"?>
96 <beans
97     xmlns="http://www.springframework.org/schema/beans"
98     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
99     xmlns:p="http://www.springframework.org/schema/p"
100    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/sch
101
102
103    <bean id="db" class="DemoBean" init-method="myInit" destroy-method="myDestroy">
104        <property name="msg"><value>hel</value></property>
105    </bean>
106    <!-- <bean id="mpp" class="MyPostProcessor"/> -->
107
108
109 </beans>
110
111 -----DemoClient.java-----
112
113 import org.springframework.beans.factory.xml.XmlBeanFactory;
114 import org.springframework.core.io.ClassPathResource;
115
116
117 public class DemoClient {
118
119     public static void main(String[] args) {
120         /*ClassPathXmlApplicationContext ctx=new ClassPathXmlApplicationContext("DemoCfg.xml");
121         Demo bobj=(Demo)ctx.getBean("db");
122         System.out.println("Result is"+bobj.sayHello());*/
123
124         XmlBeanFactory factory=new XmlBeanFactory(new ClassPathResource("DemoCfg.xml"));
125         factory.addBeanPostProcessor(new MyPostProcessor());
126         Demo bobj=(Demo)factory.getBean("db");
127         System.out.println("Result is"+bobj.sayHello());
128
129
130
131
132
133
134     }
135
136 }

```

- those are
- ① public object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException
  - ② public object postProcessAfterInitialization(Object bean, String beanName) throws BeansException.

→ ① executes before afterPropertiesSet() / init() life cycle method.

② executes after afterPropertiesSet() / init() life cycle method.

③ It's quite useful to change bean property values if they are not injected by container as expected. This logic is going to be common for multiple beans managed by that container.

In the above methods : bean → Spring Bean class object.

beanName → Bean Id

### Example Application :-

Resources :-

- Demo.java → same as previous
- DemoBean.java
- DemoCfg.xml
- MyBeanprocessor.java
- DemoClient.java

The logic of Beanpostprocessor will be developed outside

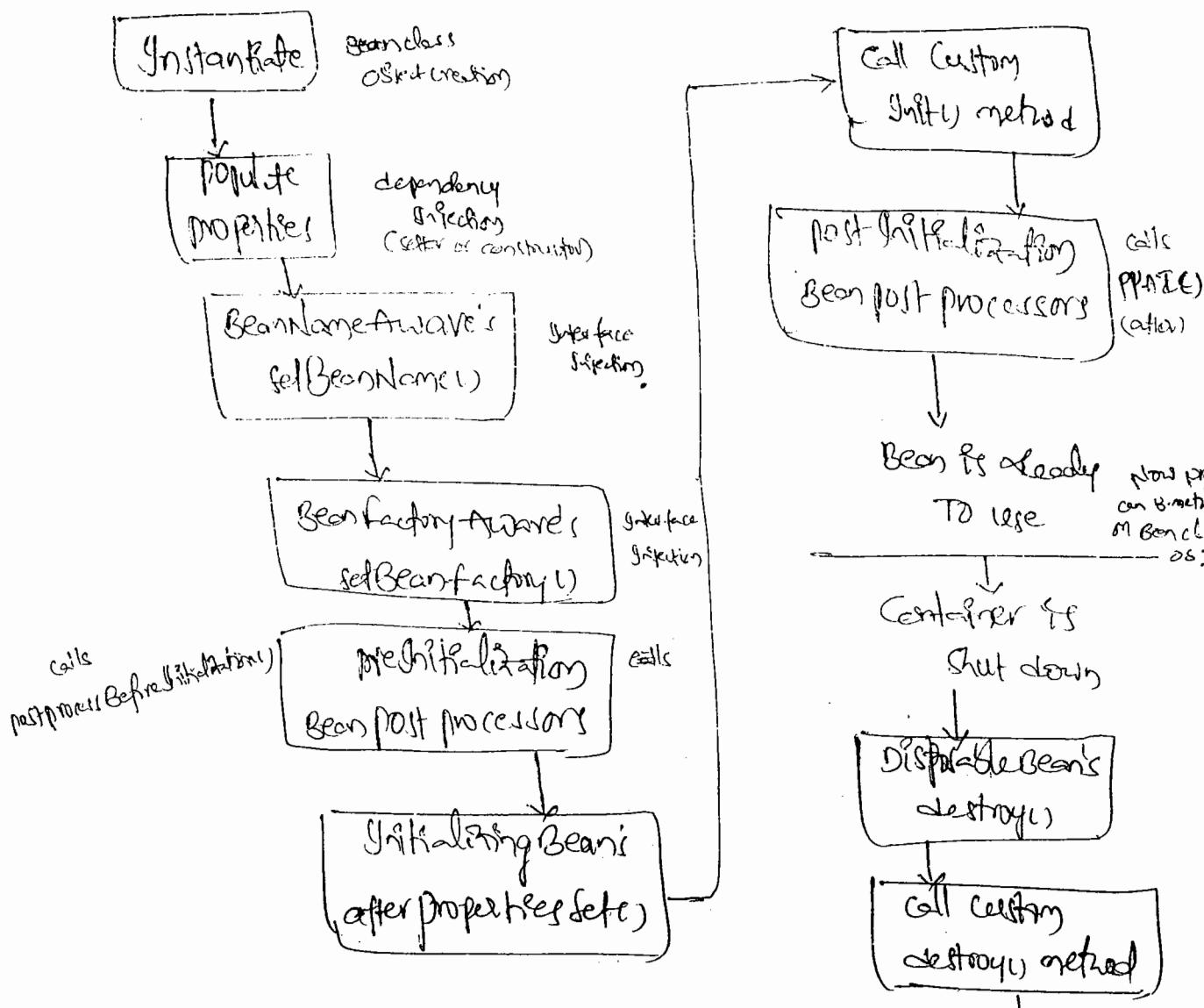
The Spring Bean class is a separate Java class that implements org.springframework.beans.factory.config.BeanPostProcessor interface.

\* for source code of the above application that deals with Beanpostprocessors refer supplementary handout given on 28/09/11.

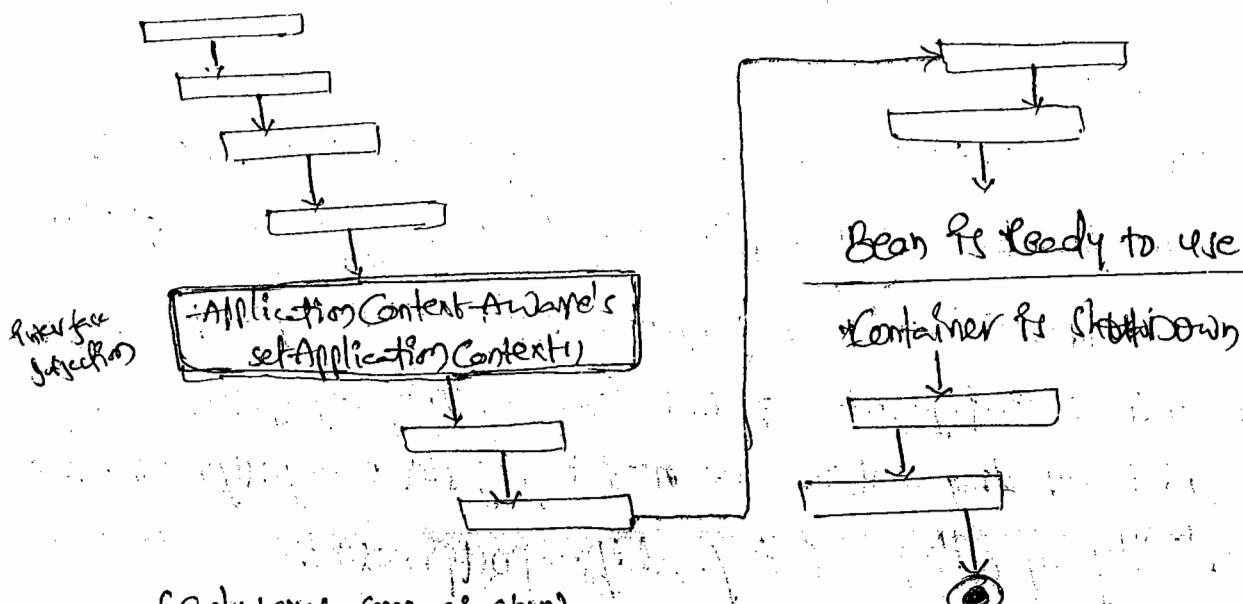
ApplicationContext container performs automatic Bean post processor registration with just configuration of that class in Spring cfg file.

for Beanfactory Container we need to register Beanpostprocessor class explicitly by calling factory.addBeanPostProcessor( ).

# Spring Bean life cycle diagram within a Spring Bean factory Context



for Application Context Container:-



when underlying Container for ApplicationContext Container

then the programmer can inject both Beanfactory, ApplicationContext containers to spring bean through Interface Injection process by implementing both BeanfactoryAware, ApplicationContextAware interface on Spring bean class. This operation is possible bcoz the ApplicationContext container is extension of Beanfactory container.

for better performance it is always recommended to activate Spring Container and get Spring Bean class objs from container by keeping code in onetime execution blocks (like Constructors). It is recommended to call business methods on Spring Bean class objects from repeatedly executing blocks. These two recommendations are given to achieve better performance.

O	Spring's Client Apps	blocks to place & to get Spring Bean class obj from Container	blocks to call business methods on Spring Bean class objects.
O	AWT / Swing frame	constructor / static blocks	event handling methods
O	Applet / JApplet	constructor / static blocks / init()	" "
O	Servlet	constructor / init() / static blocks	service method / doXXX()
O	JSP	<%! public void JspInit() { --- } %>	<% ----- %>
O	Struts	constructor / static blocks of struts Action class	execute(----) of Action class.
O	JSF	constructor / static blocks of managed Bean	Business methods of Managed Bean

03/10/11

we cannot send JDBC Resultset object over the networks.  
Bcoz it is not a serializable object.

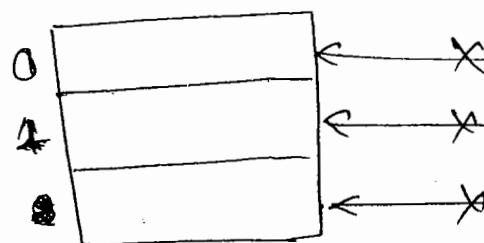
To solve the above problem and to send Resultset obj data over the networks, copy Resultset obj data to collection f/w datastructure and send that collection f/w datastructure over the networks, like ArrayList.

Note: All collection f/w datastructures are serializable objects by default. So they can be sent over the nw directly.

Understanding problem that is involved towards moving the records of JDBC resultset object to elements of ArrayList obj:

In each record of JDBC resultset object there is a possibility of having multiple objs and primitive values, so that record cannot be moved to single element of ArrayList, bcoz each element of ArrayList can allow to store only one obj at a time.

al (ArrayList obj)



rs (Resultset obj)

1	2	3
123 Nani Hyderabad		
143 Sunil Bangalore	2	
362 Kamlesh Mumbai		3

To solve the above problem read each record values into userdefined Java class obj and add that obj to elements of ArrayList. In this process this userdefined Java class is called as DTO class or VO class. Generally this class will be developed as JavaBean.

DTO class / VO class :-

Public class EmpBean implements Serializable

? Int no;

String name;

String address;

↑  
(mandatory)

// write setXXX() and getXXX()

}

Logic to transfer Resultset obj data to ArrayList obj :-

ArrayList al = new ArrayList();

Resultset rs = st.executeQuery("select \* from student");

{  
while(rs.next())

// Store each record data to DTO class obj

EmpBean eb = new EmpBean();

eb.setNo(rs.getInt(1));

eb.setName(rs.getString(2));

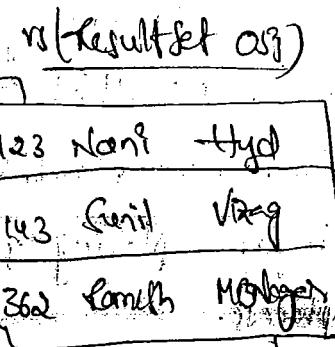
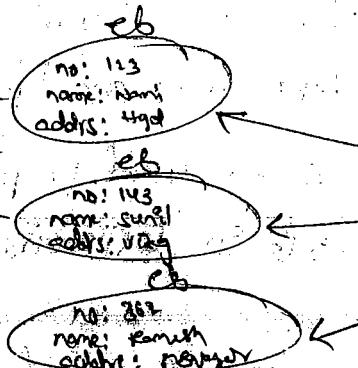
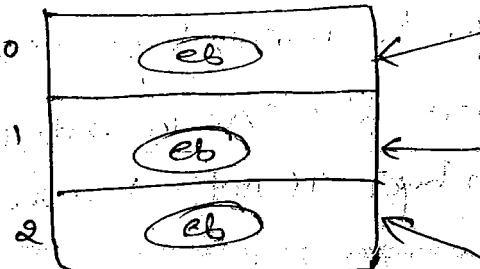
eb.setAddress(rs.getString(3));

// Now add each DTO class obj to ArrayList element

al.add(eb);

}

al (ArrayList obj)



rs (Resultset obj)

In the above code EmpBean class objects are representing single value by combining multiple values, so EmpBean class is called as Value Object (VO) class.

In the above code EmpBean class is used to transfer Resultset obj data to ArrayList obj, so it is called as DTO class.

In order to send collection/flat data structures over the net the objects added to these data structures as elements must be taken as serializable objects.

In a webapp if JSP program wants to work with userdefined Java class then it must be placed under userdefined packages created in WEB-INF classes folder.

Mini project:- (MVC or AOP)

Model - Business logic and persistence logic → spring core Module.

View - presentation logic → JSP program

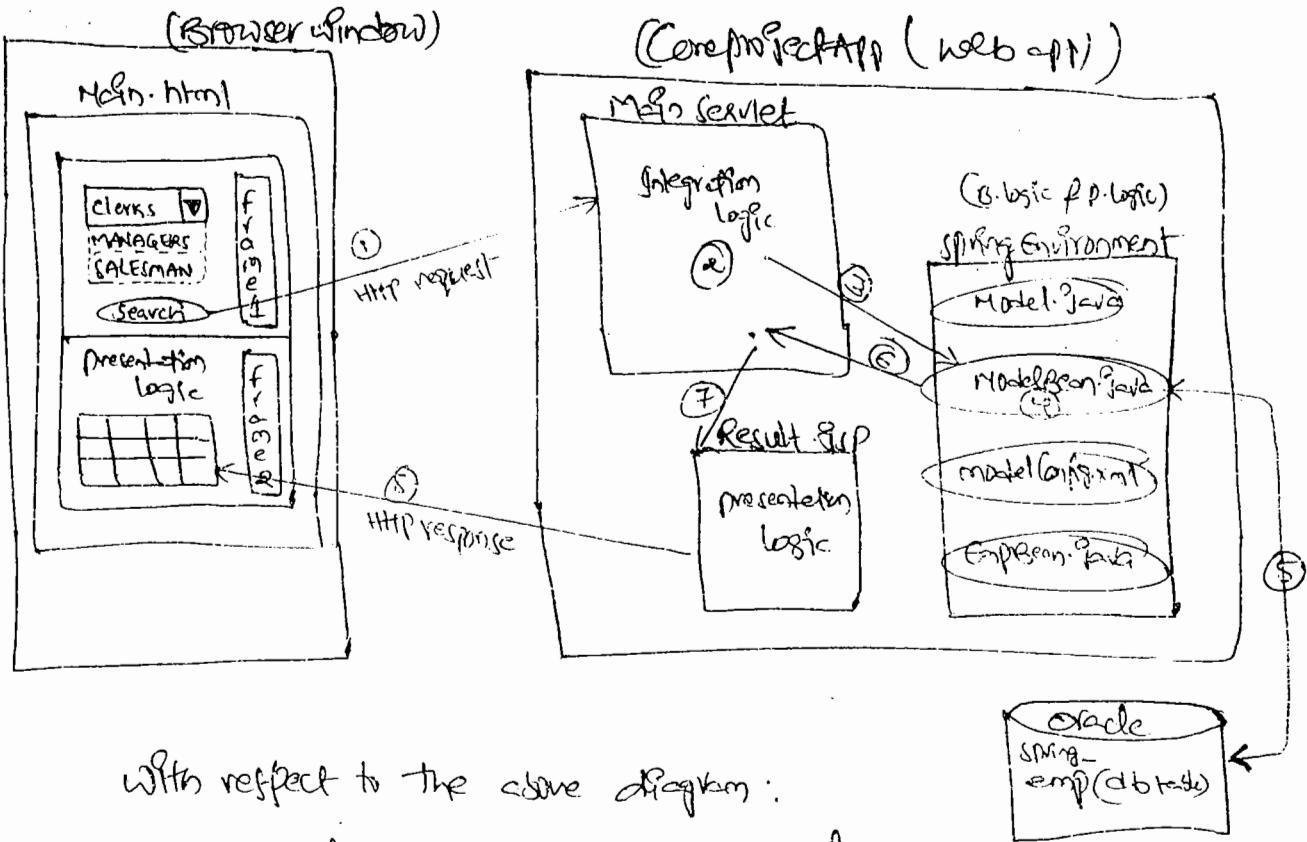
controller - Integration / connectivity logic → servlet program.

Integration / connectivity logic is responsible to get communication b/w View layer, Model layer resources through controller layer.  
This logic monitors and controls every operation of web app.

A Layer is a logical position in the app project representing certain category of logics.

Even though it is not there it is never recommended to send Resultset obj data from one layer to another layer, bcoz to receive Resultset obj we need to place code in destination layer, placing code in destination layer is not a recommended process.

## Diagram:-



With respect to the above diagram:

(1) → End user selects one designation from select box, and submit request to web app.

(2) → The controller servlet of web app traps the req and takes the res → reads form data from page → activates

Spring container → gets Spring Bean class obj from Spring container

(3) → servlet program calls the business method of SpringBean class.

(4) & (5) → The business logic and persistence logic of SpringBean collects the employee records from db table based on given designation.

(6) → Business method takes the support of DAO class (EmpBean) and sends these employee records to controller servlet in the form of ArrayList obj.

(7) → Controller servlet passes this result (ArrayList obj) to the View layer Result.jsp.

(8) → The Result.jsp formats the result by using presentation logic and sends that result to frames of main.html in the form of HTML table.

If the web resource programs of web app (servlet, JSP) are using spring API then the spring API related main jar files (spring.jar) should be added to classpath. The spring API related main and dependent jar files (spring.jar, commons-logging.jar) should be added to WEB-INF/lib folder of web app.

outlook

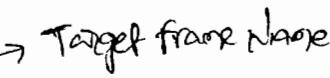
frame with name is called Named frame.

We can make response of form page generated request or hyperlink generated request to be displayed in specific frame by taking the suppose of Target attribute.

```
<form action="testURL" method="get" target="f2">
```

```
</form>
```

The response generated for this form page based request will be displayed in a frame whose name is "f2".

 Target frame name

```
<a href="testURL" target="f2"> go clas </a>
```

To pass data from servlet program to JSF program we can use request attributes if they are using same request, response objects (when they are communicating by using request dispatcher object).

factory class :-

The Factory class that is capable of returning one of its subclass or relevant class obj is known as factory class.

When normal Spring Bean is configured as dependent value to BeanFactory then normal bean class will be injected to that BeanFactory.

When factory spring bean is configured as dependent value to beanfactory then the property that the bean class generated resultant obj will be injected, but not the factory Bean class itself.

Spring Config file:

```
<beans>
    <bean id="a1" class="ABCBean"/>
    <bean id="b1" class="XYZBean"/>
    <property name="pi"> <ref bean="a1"/> </property>
</beans>
</beans>
```

If ABCBean is Simple/normal bean then the "pi" property will be injected with "ABCBean" class obj.

If ABCBean is factory Bean then the "pi" property will ~~not~~ be injected with ABCBean class. ABCBean gives the resultant object as the return value of getobject() method.

To develop above defined beans class as factory Bean of spring Environment Then that class should implements org.springframework.factory.FactoryBean interface.

This interface contains three methods.

- 1) Public object getobject() throws exception → returns resultant obj
- 2) Public class getobjectType() → returns the class name of resultant obj
- 3) Public boolean isSingleton() → returns true if Singleton else

is Singleton otherwise false.

FactoryBean required as FactoryBeans for resultant obj to exposed not directly as a normal Bean interface that will be exposed it self.

## Ex-1:

Resources : Demo.java  
 DemoBean.java  
 TestBean.java - (StringBean's factory Bean)  
 springcfg.xml  
 DemoClient.java

### Demo.java :

```
public interface Demo
{
    public String sayHello();
}
```

### DemoBean.java :

```
import java.util.Date;
public class DemoBean implements Demo
```

```
{ Date d;
```

```
public void setD(Date d)
{
    this.d = d;
}
```

```
public String sayHello()
{
    return "Good Evening" + "d=" + d.toString();
}
```

### Springcfg.xml :

#### beans

```
<bean id="tb" class="TestBean"/>
```

```
<bean id="db" class="DemoBean">
```

```
    <property name="d" ref="tb"/>
```

Final Output

#### beans

```
</beans>
```

## TestBean.java

```
import java.util.Date;  
import org.springframework.factory.FactoryBean  
  
public class TestBean implements FactoryBean  
{  
    public Object getObject() throws Exception  
    {  
        return new java.util.Date();  
    }  
  
    public Class getObjectType()  
    {  
        return java.util.Date.class;  
    }  
  
    public boolean isSingleton()  
    {  
        return false;  
    }  
}
```

## DemoClient.java

```
public class DemoClient  
{  
    public void main (String args[]) throws Exception  
    {  
        FileSystemXmlApplicationContext ctx = new FileSystemXmlApplicationContext  
        ("springcfg.xml");  
        Demo bob = (Demo) ctx.getBean("db");  
        System.out.println("Result : " + bob.sayHello());  
    }  
}
```

Normal Beans are called Self Beans bcz they always return their own obj to the configure bean properties.

factory beans are selfless beans it never injects its own bean class obj to bean property more over it always injects the generated result obj.

→ So for factoryBean based dependency injection if bean property dependent value has to be searched it's injected dynamically.

In a running Java program int type holding Integer value. String object holds String value, if the obj of java.lang class holds a class or interface we can use that obj to perform various operations on that represented class or interface.

There are three ways to create obj of java.lang. class

① by calling getClass() of java.lang. Object

```
Button b = new Button("OK");
```

```
Class c = b.getClass();
```

Here 'c' is the obj of java.lang class but not obj of Button class. Using 'c' we can create one more obj of Button class, or we can call methods of Button class.

② by using class.forName()

```
Class c = Class.forName("java.awt.Button");
```

③ by using "Class" property

```
Class C1 = Button.class; → C1 represents Button class
```

```
Class C2 = Runnable.class; → C2 " Runnable interface"
```

class levels are described below

10  
11/10/11

## JNDI :- (Java Naming and Directory Interface)

DBSLW manager information in the form of DB tables.

Mail Server manager email accounts and after email messages.

Ex: James Mail Server, MS Exchange Server, SMTP One server etc.

Registry SLW can manage objects or OSGI references with nick names or alias names for global visibility.

Ex: RMI Registry, COR Registry, JNP Registry, WebLogic Registry, Glassfish Registry, DNS Registry etc...

Java App uses Jndi API → DB SLW

Java App uses Jndi API → Registry SLW

Java App uses JavaMail API → Mail Server SLW

JDBC API and JNDI API are part of JSE module and JavaMail API is part of JEE module.

Registry SLW's are also called as JNDI registries or naming / directory services.

In order to provide global visibility to certain OSGI or its reference then keep that one in Registry SLW.

JNDI / Nick Name	Registry SLW	Java OSGI / Plain OSGI .rcf
St1	Student class OSGI	
St2	Customer class OSGI reference	
Name	JDBC datasource OSGI reference	
Blue	Date class OSGI	

JNDI API means working with classes, interfaces, enums, annotations that are available in Java Naming and its sub packages.

The DNS Registry maintains the home page URL's of diff websites along with their domain names.

### DNS Registry

Domain Name (Name)	Home Page URL (Value)
www.yahoo.com	http://180.44.38.46:8076/yahooApp/index.html
www.gmail.com	http://316.24.1.56:7080/GmailApp/index.jsp

Evey Registry shw maintains its details in the form of key-value pairs / name-value pairs.

Evey Java web server / application server shw gives one built-in Registry shw.

Weblogic shw → Web logic Registry shw

TBoss server shw → JNP Registry shw (Java naming protocol)

WebSphere server shw → cos Registry shw (Common obj service)

→ In real world while working with server managed JDBC connection pools and while developing distributed apps we need to deal with Registry shw.

### Registry shw's

Naming Registry shw

on

Naming Service shw

Directory Registry shw

on

Directory Service shw

What is the diff b/w Naming Registry JNDI and Directory Registry JNDI?

Ans:- Naming Registry maintains details in the form of Name-Value pairs and we must know names in order to get the values.

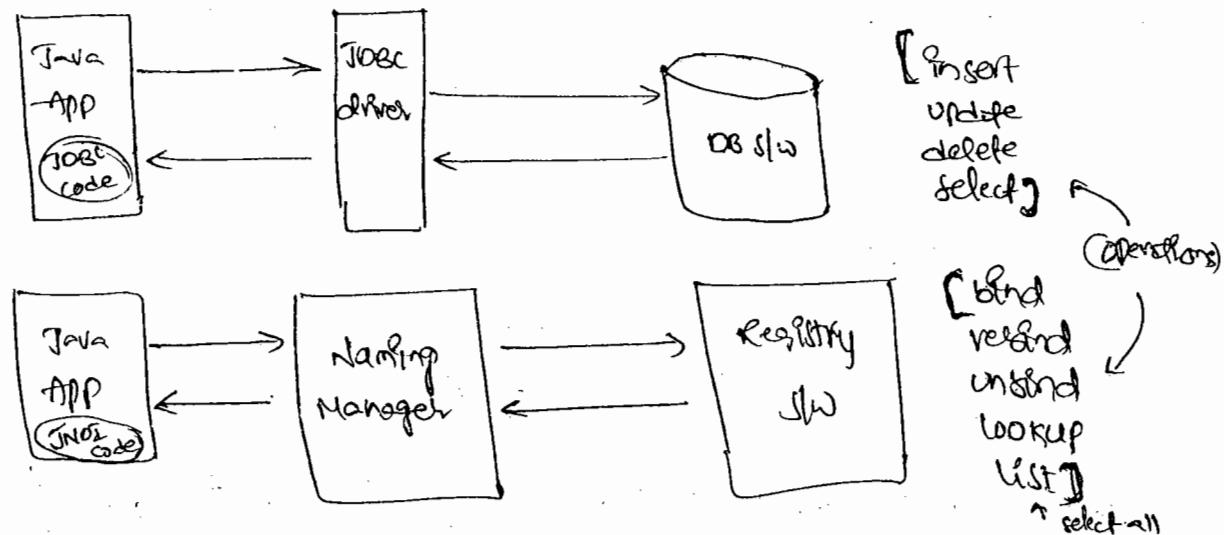
Ex:- Attendance Register, Telephone books ( Maintains ph nos alphabet wise)

tech ex:- RMS Registry, CAF Registry etc.

Directory Registry's are enhancements of Naming Registers where details will be maintained as Name-value pairs, and these values also contain additional properties, so we can use either names or additional properties to search and gather values.

Ex:- Yellow pages books ( Informedia, CSRM etc), windows file search (Maintains ph numbers category wise)

tech ex:- DNS Registry, Weblogic Registry etc...



JDBC driver is bridge b/w Java App and DB S/W, and it will be changed based on the db s/w we use.

Naming Manager is bridge b/w Java App and Registry S/W, and it will be changed based on the Registry S/W we use.

By using JDBC, Java App performs insert, update, delete, select operations on db s/w, and these operations are called JDBC persistence operations.

By using JNDI code Java app performs Bind, Rebind, Unbind, Lookup and List operations on registry slw

Bind → keeps Obj/obj ref in registry slw along with nickname.

Rebind → replaces existing Obj/obj ref of registry slw

Unbind → removes Obj/obj ref from registry slw

Lookup → Gets Obj/obj ref from registry slw using nickname.

List → Gets all Obj/obj ref their nicks names from registry slw.

JDBC connection obj represents the connectivity b/w Java app and db slw, to create this con obj we need the following JDBC prop  
- driverclassname, url, dbusername, dbpassword.

Initial Context obj represents the connectivity b/w Java app and Registry slw, and provides environment to perform opns to perform Bind, Rebind etc opns on Registry slw. To create this Initial Context obj we need the following JNDI properties-

- Initial Context factory class name
- provider url
- principal name (username)
- credentials (password of registry slw)

Note: These details will be changed based on the Registry slw we use.

Qn 14:

Procedure to create our own domain server in weblogic 10.3 :

start → programs → oracle weblogic → quick start → create new

weblogic domain → next → generated domain → next →

domain name: Java1 → next → username: JavaBoss (min 8) & last

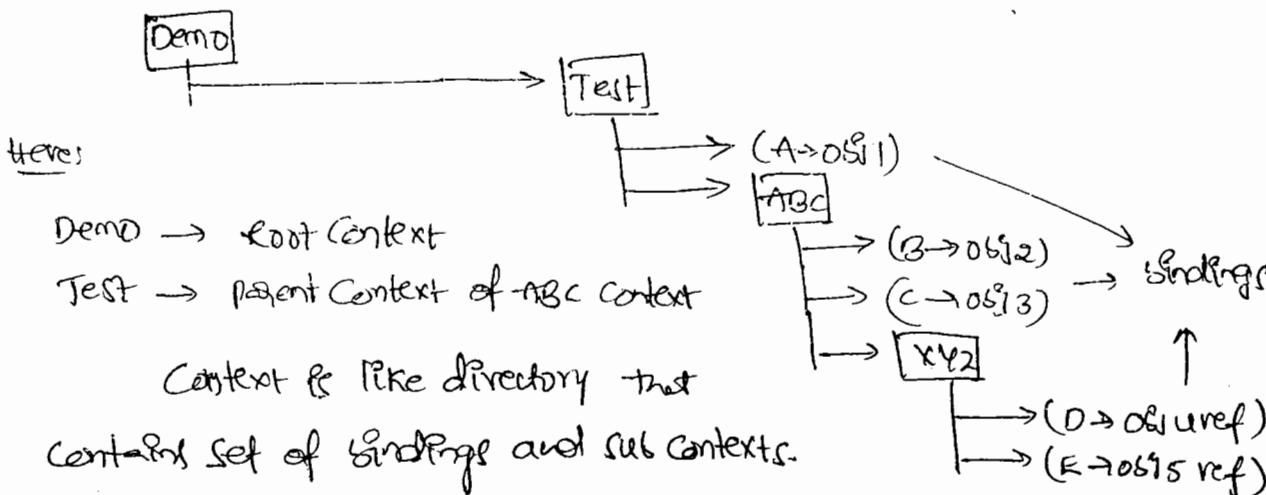
password: JavaBoss12 (min 8 + one digit)

Done & create < Next < Work ← AdminServer ← Next

procedure to observe the weblogic registry of WLS domain server of weblogic :-

start → programs → Oracle WebLogic → WLS project's → WLS domain → start Admin server for weblogic.  
(This starts WLS domain servers) → open admin console of WLS  
(open browser → type  
HTTP://localhost:7001/console →  
username : { } → login →  
Environment → servers → select Admin server → view JNDI Tree.

### JNDI Tree of Any Registry s/w



The context from which search operation is started for a particular binding is called as Initial Context.

Ex:- If search operation is started from Test context for 'OSI3' based on its nickname 'C' then 'Test' is called as Initial Context.

JNDI properties of weblogic registry s/w :-

InitialContextfactory class : weblogic.jndi.wlInitialContextFactory

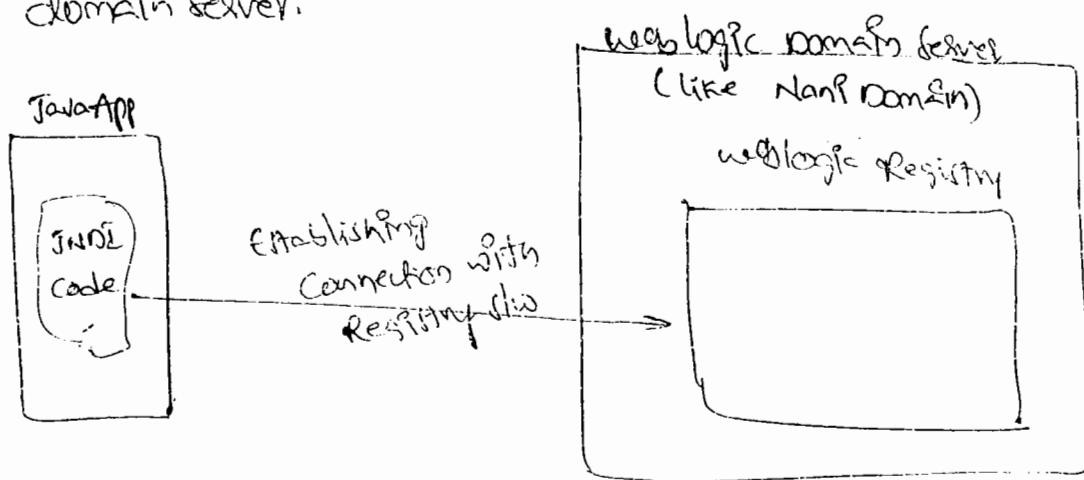
Provider URL : t3://hostname:port : <port> Ex: t3://machine30:7001

Principal Name : ~~username~~ <username> : DataBase

Credential : <password> : JavaBase

Note: t3 is application level protocol to get communication b/w Java app and weblogic registry.

weblogic registry uses same port number, same username, password of its domain server.



public static final variables of certain Java class or Interface are called as Constants.

In order to create InitialContext object we need to supply JNDI properties as Hashtable object element values.

To place every JNDI property value one fixed JNDI property name is given, which is constant of javax.naming.Context interface.

\* `InitialContext` implements `java.naming.Context`.

→ Example code to establish connection with weblogic registry.

// prepare JNDI properties.

```
Hashtable ht = new Hashtable();
```

```
ht.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.  
WlInitialContextFactory");
```

```
ht.put(Context.PROVIDER_URL, "t3://localhost:7001");
```

```
ht.put(Context.SECURITY_PRINCIPAL, "JavaBoss");
```

```
ht.put(Context.SECURITY_CREDENTIALS, "JavaBoss2");
```

// Create InitialContext object.

InitialContext ic = new InitialContext(ht);

(represents connectivity with weblogic registry s/w)

Example App to establish the connection with weblogic Registry (lw) :-

```
import javax.naming.*;
```

```
import java.util.*;
```

```
public class InfolConnTest
```

```
{ public void (String args[]) throws Exception
```

```
{
```

```
Hashtable < String, String > ht = new Hashtable< String>();
```

```
ht.put ( Context.INITIAL_CONTEXT_FACTORY, " " );
```

```
ht.put ( Context.PROVIDER_URL, " " );
```

```
InitialContext ic = new InitialContext ( ht ); → points to
```

```
if (ic == null)
```

Root Context

```
System.out.println (" Connection is not established");
```

```
else
```

```
System.out.println (" Connection is established");
```

```
}
```

→ keep weblogic's NonI domain server in running mode

→ add weblogic.jar to classpath.

Output

Sample code to perform List operation:-

// InfoOperationsTest.java

```
public class InfoOperationsTest
```

```
{ public void (String args[]) throws Exception
```

```
{ InitialContext ic = new InitialContext (); → points to Root Context
```

InitialContext ic = new InitialContext(ht);

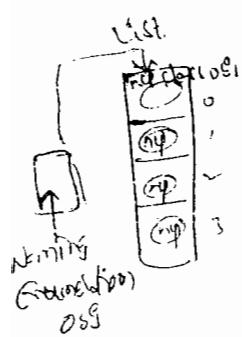
// list operations on root context of Naming Registry.

NamingEnumeration ne = ic.list(""); // represents root context.

((on NamingEnumeration ne = ic.list("Java"); // represents Java context.

((on NamingEnumeration ne = ic.list("Java/translation");

// process results.



while(ne.hasMore())

\* NameBindingPair neP = (NameBindingPair) ne.next();

// gives each bindings

s.o.p(neP.getName() + " → " + neP.getClassName());

↓  
give NickName

of each binding

↓  
give className of  
each bound object.

\* Each NameBindingPair obj can give its name and bound obj class name of each binding.

→ To perform Bind operation on Registry :-

Bind:- ic.bind("today", new Date());

ic.bind("apple", new String("It is Green"));

Rebind:- ic.rebind("Apple", new StringBuffer("It is red"));

Lookup:- Date d = (Date) ic.lookup("today");

s.o.p("lookup for today:" + d.toString());

s.o.p("lookup for Apple:" + ic.lookup("apple"));

unbind:-

ic.unbind("today"); (log: s1 to s2)

for example: now mind the return value TestInitialBindTestInitialPrint

JBoss :- (Application Server) (Apache foundation) (5.x) → 1.6.3, etc

OpenSource. Port No: 8080 (changeable)

→ default domain: default, web, all, standard, minimal

→ Jar file that represents whole See api's is: JBoss-Javaee.jar

→ Built-in registry s/w: Jnp Registry (Port no: 1099)

To install JBoss do extract the zip file to a folder (JBoss-version.zip)

To start JBoss server use JBossHome/run.bat file. (remove "findstr" in this file)

Procedure to view JNDI Tree of JBoss server's Jnp registry:-

Step①: Start JBoss server as shown above.

Step②: Open admin Console of JBoss server and observe JNDI Tree

Open Browser → http://localhost:4440/Jmx-Console → service=JNDI View → list() → invoke → observe global JNDI Name space

Properties:  
InitialContextFactoryName: org.jnp.interfaces.NamingContextFactory  
ProviderURL: jnp://localhost:1099

Glassfish: (OpenSource)

Type: Application Server s/w

Vendor: Sun MS

Version: 2.x (compatible with Glass 1.5/1.6)

default port no for admin console: 4848

Registry s/w name: Glassfish Registry

Jar file that represents whole See api's: Javaee.jar

The default domain is: domain1, mydomain, myapp

Don't install Glassfish 2.x separately. Install NetBeans 6.7.1 and

it gives built-in Glassfish 2.x and we operate this Glassfish server with/without IDE

Note:- In our JNDI apps the JNDI property values will be changed based on the Registry s/w we use, but the code is same for all (different JBoss, Glassfish, JBoss2, Glass, Glassfish2, Glassfish3)

procedure to observe JNDI Tree of Glassfish registry. In Glassfish server related "domain" server :-

Step(1) :- start the domain server

start → programs → SunMicrosystems → AppServer 2.1 → start default server

Step(2) :- Launch admin console of Glassfish server domain1

open browser window → type : `Http://localhost:4848` → username : `admin` →  
password : `adminadmin`

Step(3) :- Launch Jndi tree of Glassfish Registry.

admin console → appserver → Jndi browsing → Jndi Tree

Jndi properties of Glassfish Registry :-

InitialContextFactory className : com.sun.enterprise.naming.

(over internet orb protocol)

InitialContextFactory

Provider URL : `Riop://localhost:4848/c848` (Home/sun/AppServer/1.6/Amfear-rt)  
(ear files location)

principalName : `admin`

} optional

Credentials : `adminadmin`

Ex code that establishes connection with Glassfish registry of Glassfish :-

// prepare Jndi properties having values.

```
Hashtable ht = new Hashtable();
```

```
ht.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.enterprise.naming");
```

```
ht.put(Context.PROVIDER_URL, "Riop://localhost:4848");
```

```
ht.put(Context.SECURITY_PRINCIPAL, "admin");
```

```
ht.put(Context.SECURITY_CREDENTIALS, "adminadmin");
```

// Create Initial Context

```
InitialContext ic = new InitialContext(ht);
```

Properties : `amfear-rt.jar` (main package) dependent to main

15/10/11

JDBC connection pool is a factory that maintains set of readily available JDBC connection objects before actually being used.

There are two types JDBC connection pools.

1. Driver managed / stand alone JDBC connection pool
2. Server managed JDBC connection pool (created / managed by Application / web tiered)

This DataSource object represents server managed JDBC connection pool, and for global visibility this datasource object will be placed in registry / w/ having Jndi name / nisq name.

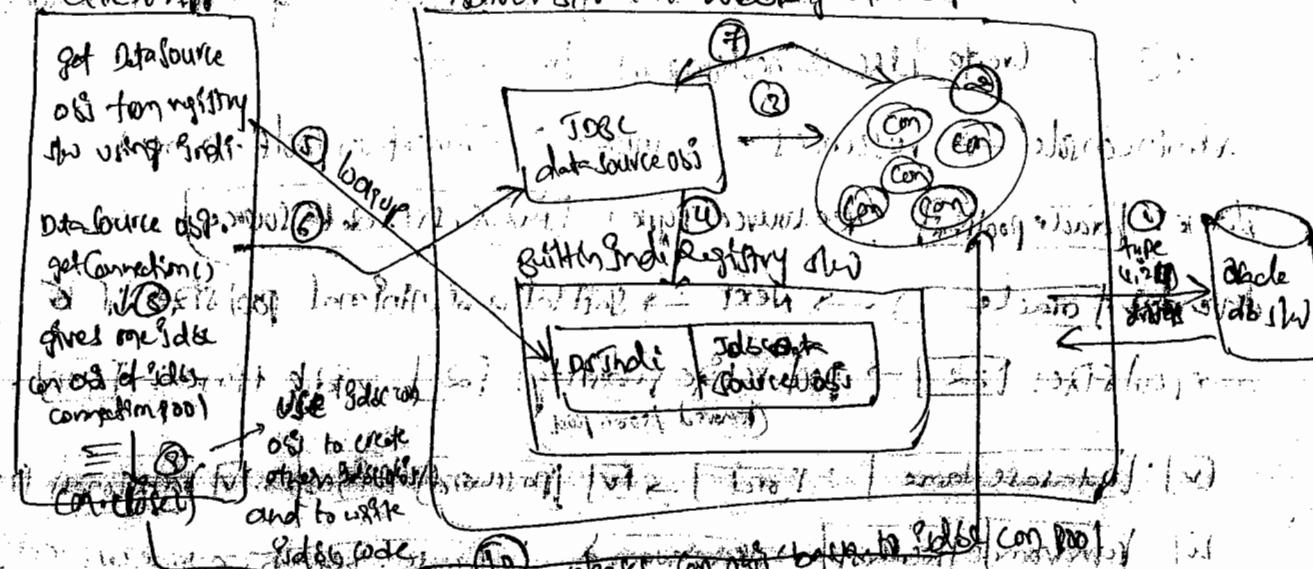
JDBC Datasource OBI means it is the object of a Java class that implements javax.sql.DataSource interface.

Client applications will use JDBC datasource obj to access the JDBC connection objects from JDBC connection pool.

All JDBC connection objects of JDBC connection pool represents connectivity with same db (w).

Ex:- Jndi connection pool - for oracle means all JDBC connection objects in this pool represents connectivity with Oracle db (w).

client app server side like weblogic, websphere etc



steps:

Step ① & ②: In server environment, MySQL user, type 1 or 4 JDBC drivers to create JDBC connection pool for certain DBs who having JDBC connection objects.

③ & ④: PL/SQL creates JDBCDataSource obj in server environment representing JDBC connection pool and also registers that dataSource obj in JNDI Registry also having JNDI name.

⑤: Client App performs JNDI lookup operation on registry JNDI and gets JDBC DataSource obj.

⑥ & ⑦: Client App calls getConnection() method on dataSource obj, this call gets one JDBC connection obj of ConnectionPool through dataSource obj (server managed).

⑧ & ⑨: Client app -----> refer diagram.

The released Connection object now becomes ready to give service to other clients and new requests.

Procedure to create JDBC Connectionpool for Oracle in Glassfish:-

Step ①:- Keep odbc4j.jar file in Glassfish\home\APPServer\domains\domain1\lib\ext folder.

Step ②:- Start domain1 server of Glassfish and open its Admin Console.

Step ③:- Create JDBC Connectionpool for oracle

Admin Console → resources → JDBC → Connection pools → new →

Name: oracle pool Name: Resource Type: javax.sql.DataSource

driver: oracle → Next → initial and minimal pool size: 10

max pool size: 50 → pool resize quantity: 2 → idle time: 1000 → (Create from pool)

✓: Database name: Oracle → ✓: Password: 1234567890 → ✓: Port number: 1521

✓: Test connection on creation → ✓: Max error count: 1000 → ✓: Max idle time: 1000

→ launch oraclepoolnani → ping → save

Step ④ Create Jdbc datasource representing the above IdsCConnPool.

Admin Console → Resources → JDBC → JDBC Resources → New →

JNDI Name: DCINDE → Pool Name: Oracle poolnani → OK

Note: When the above OK button is clicked the created JDBC datasource obj will be registered with Glassfish registry automatically with JNDI Name : DCINDE.

Standalone client application to access the JDBC connections of the above server managed JDBC connection pool.

PoolTest.java:

```
import javax.naming.*;  
import java.util.*;  
public class PoolTest  
{  
    public static void main(String args[]){  
        Hashtable ht = new Hashtable();  
        ht.put(Context.INIT_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");  
        ht.put(Context.PROVIDER_URL, "ldap://192.168.1.11:389");  
        InitialContext ic = new InitialContext(ht);  
        // get DataSource obj from global registry through lookup()  
        DataSource ds = (DataSource)ic.lookup("DCINDE");  
        System.out.println(ds.getClass());  
        // get Connobj from Jdbc connection pool  
        Connection con = ds.getConnection();  
        // write some persistence logic  
        Statement st = con.createStatement();  
        ResultSet rs = executeQuery("select count(*) from EMP");
```

```

    if (rs.next())
        soap(" No. of records in emp table : " + rs.getInt(1));
    // release psdtc can also work to psdsc con pool
    psdtc.close();
}

```

Add following Jar files to classpath before executing the above app.

- |                        |  |
|------------------------|--|
| 1. appserver-rt.jar    | available in GlassfishHome/appserver/lib                                 |
| 2. Javaee.jar          |  |
| 3. appserver-admin.jar | available in GlassfishHome/appserver/lib<br>Install Applications/glassva |
| 4. EntityManager.jar   |  |
| 5. desccore.jar        |  |

While working with Glassfish registry we can create InitialContext obj without TNSL properties related map object. (not recommended)

```
InitialContext ic = new InitialContext();
```

The above statement worked app execution you should add Javaee.jar, appserver-rt.jar files to classpath.

procedure to create JNDI Connection pool for Oracle JDBC datasource in weblogic 10.2 server:-

Step① : start the Name domain server of weblogic and open its admin console.

Step② : Create JDBC connection pool for Oracle along with JDBC datasource.

Admin Console → services → Jdbc → Datasources → new → name : [myds]

JNDI Name : [orclIndi] database Type : [Oracle] driver [oracle.jdbc.driver.OracleDriver]

→ Next → DBName : [orcl] hostName : [localhost] port : [1521]

DBUsername : [scott] pdl : [tiger] confirm pdl : [tiger] → next →

Test Configuration → next → [Select Admin Server] → finish

→ [Finish] → [Finish] → [Finish] → [Finish]

Note: Having defined an EJB3 name the above created data source OSGI that represents JDBC connection pool for Oracle will be registered automatically with weblogic registry.

Step③: specify additional parameters to the above created Conn pool.

Admin console → services → JDBC data sources → MyDS → Connection pool Tab →  
Initial Capacity:  Max Capacity:  Capacity Increment:  →  
Advanced → Shrink Frequency:   
(idle time)

Writing Standalone Test App to access the JDBC Connection Pool of the above JDBC connection pool:-

poolTest.java

Same as previous app, but <sup>ref</sup> properties should be change to Weblogic

```
    Hashtable ht = new Hashtable();
    ht.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFactory");
    ht.put(Context.PROVIDER_URL, "t3://localhost:7001");
```

To execute the above app add weblogic.jar to classpath.

Spring JNDI:-

A template class that contains automated code taking care of common activities of app development, so programmer just concentrate on the specific app development.

Spring provides lots of template classes to provide abstraction layer on plain Java Tree technologies by taking care of the common activities of app development. So in Spring environment the programmers just need to worry about application specific requirements.

In Spring Indie framework → Indie Complete

In Spring 2011 (DA) env → 1 TDec template

In Spring open env → Efklende Template, JDO Template, TopLink template.

## SPRING 2010

1. Provides abstraction layer on plain Indi program.
  2. Use IndiTemplate class to take care of the common coding operations of Indi programming.
  3. Takes care of exception handling, and also ~~throws~~ ~~catches~~ ~~creates~~ ~~generated~~ ~~unhandled~~ exceptions using exception ~~returning~~ concept.
  4. Overall it simplifies Indi programming without using plain Indi API.

~~Platfor~~mid~~~~ code execution internally takes care of the ~~platfor~~mid~~~~ code and converts ~~platfor~~mid~~~~ code generated checked exceptions into unchecked exceptions.

Public void on()

? try 12 =

Catch( IOException e)

{ } new, new. NumberformedException ("Exception Raised");

This was not there  
in India but there  
is TOSB (Springboard)  
Templates

~~while selling mill~~ while selling mill we need not handle Exception, bcoz

we can handle the unchecked exception (NumberFormatException) optionally, if we want to do so.

18/10/11

org.springframework.jdbc.JdbcTemplate class supplies all basic methods to perform Jdbc operations but it doesn't supply list() to perform listing operation so this list() functionality can be implemented explicitly through JdbcCallback interface.

The spring even allows to perform dependency injection on a class that acts as client app by activating spring container.

for example on Spring Jndi refer app 12 of material : page : 58-60

while working with JNDI template class - the InitialContextFactory class name related Jndi property must be taken as "InitialContextFactory" and must not be taken as "initial-context-factory"

Note :- The JNDI Template class springJndi environment is not converting checked exceptions generated underlying plain Jndi through unhandled exceptions , so exception handling is mandatory in spring Jndi environment .

```
→ addWindowListener( new WindowAdapter() {  
    public void windowClosing( WindowEvent we)  
    {  
        //  
    }  
});
```

In the above statement in the parenthesis of addWindowListener() method one anonymous inner class is created extending from WindowAdapter class having overriding of windowClosing method moreover that anonymous inner class obj is created and passed as argument value of addWindowListener method call.

The Template classes of spring programming supplies callback interfaces allowing the programmer implementing certain functionality using underlying plain technology app when that functionality is missed in template class

The Indi Template class gives execute() to execute the functionality defined in Callbacks Interface Implementation.

(IndiCallbacks)

Ex - App on Indi Callbacks Interface Implementation to provide list() method functionality by using Java NNP.

democfg.xml : → same as previous app (app12)

IndiTemplateTest.java

```
import org.springframework.context.support.*;  
import org.sf.indi.*;  
import javax.naming.*;  
  
public class IndiTemplateTest  
{  
    static IndiTemplate template;  
  
    public void setTemplate(IndiTemplate template)  
    {  
        this.template = template;  
    }  
  
    public void main(String args[]) throws Exception  
    {  
        FileSystemApplicationContext ctx = new  
            FileSystemXmlApplicationContext("democfg.xml");  
        template.execute(new IndiCallback());  
    }  
}
```

public Object doInContext(Context ct) throws IndiException  
→ represents ApplicationContext of Indi env.

{ If list() with plain Indi. NamingException

NamingEnumeration ne = ct.list();

while (ne.hasMore())

NamingContext np = (NamingContext) ne.nextElement();

String name = np.getNames();

String value = np.getValues();

In the above ex template.execute() is called having anonymous class obj as argument value and that anonymous class implements TestCallback interface.

QUESTION

In spring application we can work with 3 types of JDBC Connection Pool.

1. Spring's builtin connection pool using DriverManagerDataSource class or SingleConnectionDataSource class.
2. Third party managed connection pool like Apache DBCP or C3P0
3. Web/application server managed JDBC connection pool

Note:- The ~~DriverManagerDataSource~~, SingleConnectionDataSource related connection pool does not actually pool connection objects moreover it creates new JDBC connection objects in Connection pool for every call of ds.getConnection().

This connection pool is not suitable in modern projects.

Q: What is the JDBC connection pool that you have used in your spring project?

A:- If your spring app is standalone environment project then use third party SWI based connection pools like Apache DBCP or C3P0.

If your spring project is deployable application in the servers like webapplications then use server managed JDBC connection pool.

→ The TomcatHome\lib\Tomcat-dbcp.jar file represents Apache DBCP SWI, this jar file contains org.apache.tomcat.dbcp.BasicDataSource class whose dataSource obj represents one JDBC Connection pool.

Java Bean properties of BasicDataSource class:

- |                    |                    |
|--------------------|--------------------|
| 1. username        | 5. initialSize     |
| 2. password        | 6. maxActive       |
| 3. url             | 7. maxWait         |
| 4. driverClassName | 8. removeAbandoned |

Example Application:— (Tomcat DBCP)

Demo.java:

```
public interface Demo  
{  
    public int fetchSalary(int eno);  
}
```

DemoBean.java

```
import java.sql.*;  
import javax.sql.*;  
  
public class DemoBean implements Demo  
{  
    DataSource ds;  
    // refxx() for setter injection  
    public void setDs(DataSource ds)  
    {  
        this.ds = ds;  
    }  
  
    public int fetchSalary(int eno)  
    {  
        int sal = 0;  
        try {  
            Connection con = ds.getConnection();  
            PreparedStatement ps = con.prepareStatement("select  
                sal from temp where eno = ?");  
            ps.setInt(1, eno);  
            ResultSet rs = ps.executeQuery();  
            if (rs.next())  
                sal = rs.getInt(1);  
        } catch (Exception e)  
        {  
            e.printStackTrace();  
        }  
        return sal;  
    }  
}
```

## Demo Cfg.xml :-

<beans>

```
<bean id="dbcp" class="org.apache.tomcat.dbcp.dbcp.  
BasicDataSource" destroy-method="close">  
    <property name="driverClassName" value="oracle.jdbc.  
        driver.OracleDriver"/>  
    <property name="url" value="jdbc:oracle:thin:@localhost:1521:ORCL"/>  
    <property name="username" value="scott"/>  
    <property name="password" value="tiger"/>  
    <property name="initialSize" value="2"/>
```

</beans>

```
<bean id="ds" class="DemoBean">  
    <property name="ds" ref="dbcp"/>
```

</bean>

</beans>

## DemoClient.java :-

```
import org.springframework.context.*;  
public class DemoClient  
{  
    public void main(String args[]){  
        ClassPathXmlApplicationContext ctx = new  
            ClassPathXmlApplicationContext("DemoCfg.xml");  
        Demo bobbi = (Demo) ctx.getBean("ds");  
        System.out.println("Result Salary is :" + bobbi.fetchSalary(425));  
    }  
}
```

Note:- Make sure that the following jar files are added to classpath.

- 1) Spring.jar
- 2) Commons-logging.jar
- 3) tomcat-dbcp.jar
- 4) Oracle14.jar

\* The HB 3.2.5 Home | lib | c3p0-0.9.1.jar file represents C3P0 JAR.  
This jar file contains com.mchange.v2.c3p0.ComboPooledDataSource class. This class obj is GlobalDataSource object representing one C3P0 JDBC Connection pool.

This class having the following prop properties.

- 1) user
- 2) password
- 3) JDBCURL
- 4) driverClass
- 5) maxPoolSize
- 6) initialPoolSize

Example application: (C3P0 pool)

Demo.java, DemoBean.java, DemoClient.java → Same as previous

DEMOCFG.XML :-

<beans>

```
<bean id="c3p0" class="com.mchange.v2.C3P0.  
ComboPooledDataSource" destroy-method="close">  
    <property name="driverClass" value="oracle.jdbc.. .."/>  
    <property name="JDBCURL" value="jdbc:oracle:..."/>  
    <property name="user" value="scott"/>  
    <property name="password" value="tiger"/>
```

</bean>

```
<bean id="db" class="Demo Bean">
```

```
    <property name="ds" ref="c3p0"/>
```

</bean>

</beans>

Note: Make sure that the following jar files are added to classpath,

- 1) spring.jar
- 2) commons-logging.jar,
- 3) c3p0-0.9.1.jar
- 4) ojdbc4.jar.

~~20/11~~  
org. sf. Indi. IndiObjectfactoryBean is a predefined factory

bean class that looks up for a object in Indi registry and exposes / projects found out in Indi Registry for our Bean class properties as dependent value.

Since it is a factory Bean when its beanId is configured as dependent value to our Bean class property then IndiObjectfactoryBean class obj will not be projected to that property but the object that is gathered from Indi Registry will be injected to our Bean class property.

The important two properties of IndiObjectfactoryBean class are 1. JNDIName 2. IndiEnvironment. (for specifying Indi properties) (for lookup)

The IndiObjectfactoryBean class is popularly used to inject the dataSource obj gathered from registry into to our Bean class properties.

Example Application:-

Demo.java, DemoBean.java, Democlient.java are some of previous app.

Democfg.xml:-

<beans>

<bean id="jofb" class="org. sf. Indi. IndiObjectfactoryBean">

<property name="IndiName" value="DsTnali" />

<property name="IndiEnvironment">

→ Indi properties of weblogic registry

<props>

<prop key="java.naming.factory.urlprefix">

weblogic.Indi.WLJndiContextFactory </prop>

<prop key="PROVIDER\_URL">t3://localhost:7001</prop>

</props>

</property> </bean>

<bean id="db" class="Demo Bean">

<property name="ds" ref="jofb" /> </bean>

</beans>

there 'ds' project's Jdbc Datasource obj gathered from Registry s/w  
will not 'go'.

Jar files required : (1) spring.jar (2) commons-logging.jar  
(3) weblogic.jar (for Jndi properties).

It is not recommended to use server managed Jdbc Connection pool  
in standalone / desktop spring / Java apps. When your app is deployable  
and executable on the server then only it is recommended to use  
server managed Connection pool in that app (like webapp). In this scenario  
there is no need of specifying Jndi properties in the application.

## Spring DAO

This module provides abstraction layer on plain JDBC  
programming and simplifies the process of developing persistence  
logic by specifying JdbcTemplate class. This class takes care  
of common activities of JDBC program, concentrate on application  
specific activities like preparing SQL query, executing query,  
gathering and processing the Resultset.

### plain JDBC

1. Load JDBC driver class and register with DriverManager service.
2. Establish connection with db shw.
3. Create JDBC Statement obj.
4. Prepare query, send and execute that SQL query to db shw.  
common
5. gather query results and process the results.  
app specific
6. Take care of Exception Handling.
7. Take care of Transaction Management if necessary.
8. close JDBC obj along with JDBC connection.

The code of common activities will always remain same more or less in all applications. Whereas the app specific activities code will change from application to application.

In Spring Jdbc / Dao environment The org.springframework.jdbc.core.JdbcTemplate class takes care of common activities / work flow of Jdbc programming and makes programmer to just take care of app specific activities.

Spring Jdbc :-

- 1) Get JdbcTemplate class obj through dependency injection.
- 2) Use JdbcTemplate obj to prepare sql query, send and execute that sql query to db sql.
- 3) Gather sql query results and process the results.

JdbcTemplate class internally performs :-

- 1) Common activities of Jdbc programming.
- 2) Exception Handling and Transaction Management.
- 3) Converts plain Jdbc generated checked exceptions to unchecked.

\* The Spring Dao / Jdbc persistence logic internally uses plain Jdbc and converts plain Jdbc generated checked exceptions to unchecked exceptions by using exception rethrowing concept.

DAO is a design pattern which is nothing but a Java class / comp in our app that separates persistence logic from other logics of the app and makes the persistence logic as flexible logic to modify.

Using spring DAO or plain Jdbc or plain Hibernate or spring ORM module we can develop the persistence logic of DAO class.

Using spring DAO module we can develop persistence logic with or without implementing DAO design pattern.

In plain JDBC programming the select query gives JDBC Resultset obj and it is not serializable obj by default so we cannot send this obj over the network.

In Spring DAO environment we got diff varieties of query(), queryXXX() fn JDBCTemplate class to get select query results in programmer choice format.

- Ex queryForInt(), queryForLong(), queryForRowSet(), queryForMap(), queryForObject(), queryForList() . . . etc..

To create JDBCTemplate obj we need JDBCDataSource obj.

In spring cfg file:-

```
<bean id="dbcp" class="org.apache.tomcat.dbcp.dbcp.  
BasicDataSource">  
    <!-- Properties -->  
    <property name="dataSource" ref="dbcp"/>  
</bean>  
  
<bean id="temp" class="org.springframework.jdbc.core.JdbcTemplate">  
    <property name="dataSource" ref="dbcp"/>  
</bean>  
  
<bean id="db" class="DemoBean">  
    <property name="jt" ref="temp"/>  
</bean>
```

With respect to above code our SpringBean class DemoBean is injected with JDBCTemplate class obj (to "jt" property), so the business methods of DemoBean class we can use the injected JDBCTemplate class obj to send and execute sql queries.

21/10/11

In JdbcTemplate class environment,

① use query() or queryForXXX() for executing queries.

a) queryForInt(), queryForLong() :- To get numeric values  
or sql select query results.

select count(\*) from emp;

select \* from emp where empno = 7488;

b) queryForMap(): to get one record from Hashtable by executing query  
select \* from emp where empno = 7488;

c) queryForList(): to execute sql select query that return/execute  
multiple records in List object.

select \* from emp;

d) queryForRowSet(): To execute sql select query that returns  
multiple records, gives records in RowSet object.

select \* from emp;

Note: - Resultset obj is not a serializable obj, where as Rowset obj  
is a serializable obj.

e) queryForObject(): To execute sql select query and to  
gather result in customized Java class obj.

② Use update() to execute all non-select sql queries.

③ Use batchUpdate() for batch processing of sql queries.

Note: - Using Spring DAO we can also call pl/sql procedures/functions of db side.  
Both JDBC and Spring DAO persistence logic are db side dependent  
persistant logics, bcoz they use the db side dependent sql queries.

Ex-App:- Demo.java, DemoBean.java, democonfig.xml, DemoClient.java

while developing this app by using myeclipse IDE choose following  
libraries, 1) spring core libraries 2) persistiance java libraries  
3) persistiance core libraries

## DemoCfg.xml :-

<beans>

```
<bean id="dbcp" class="org.apache.tomcat.dbcp.dbcp.  
BasicDataSource" destroy-method="close">  
    <property name="driverClassName" value="oracle.jdbc.driver.  
        OracleDriver"/>  
    <property name="url" value="jdbc:oracle:thin:@localhost:1521:orcl"/>  
    <property name="username" value="scott"/>  
    <property name="password" value="tiger"/>  
</bean>
```

```
<bean id="template" class="org.st.9dte.core.JdbcTemplate">  
    <property name="dataSource" ref="dbcp"/>
```

</beans>

```
<bean id="ds" class="DemoBean">  
    <property name="tf" ref="template"/>  
</bean>
```

</beans>

Demo.java :-

```
import java.util.*;  
import java.sql.*;  
public interface Demo  
{  
    public long fetchEmpId (int emp);  
    public int countEmp();  
    public Map fetchEmpDetails(int emp);  
    public List fetchEmpDetails(String job);  
    public RowSet fetchEmpDetails(long deptno);  
    public boolean registerEmp(int emp, String ename, String job, double  
        salary);  
    public boolean hikeSalary (int emp, int percentage);  
    public boolean promoteEmp (int salRange);  
}
```

DemoBean.java :-

```
import java.util.*;  
import javax.ssi.ResultSet;  
import org.if.jdbc.core.*;
```

public class DemoBean implements Demo

```
{ JdbcTemplate jt;
```

```
public void setJt(JdbcTemplate jt) { this.jt = jt; }
```

```
public int countEmp()
```

```
{ int count = jt.queryForInt("select count(*) from emp");  
    return count;
```

```
public long fetchEmpSal(int eno)
```

```
{ long salary = jt.queryForLong("select sal from emp where  
    empno=?", new Object[] { new Integer(eno) });  
    return salary;
```

(\* Most of the queryforxxx() methods with single argument value internally uses simple Statement obj to execute sql query where some methods with more than one arg values internally use preparedStatement obj to execute sql queries. \*/

```
public Map fetchEmpDetails(int eno)
```

```
{ Map m = jt.queryForMap("select * from emp where empno=?",  
    new Object[] { new Integer(eno) } );  
    return m;
```

Column names  
as key {

empno	7698	0
ename	Alex	1
Job	salesman	2
sal	1600	3
mgr	7698	4
hiredate	20-feb-82	5
Comming	300	6
Address	1A	7

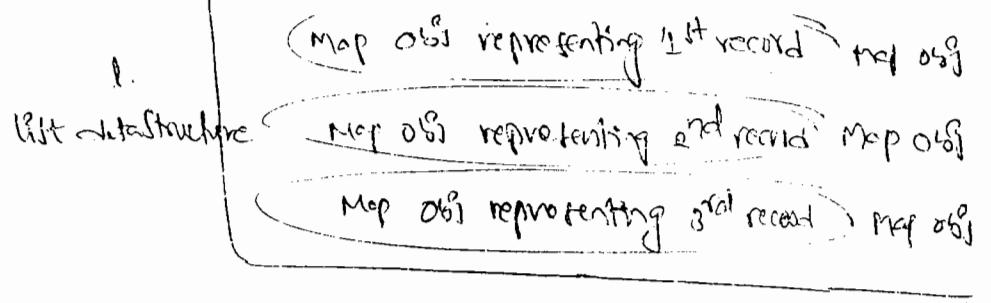
} map datastructure  
column values as  
values.

```

public List fetchEmpDetails( String job)
{
    List l = qf.executeQuery("select * from emp where job= ?",
                           new Object[] {job});
    return l;
}

```

22/10/11



```

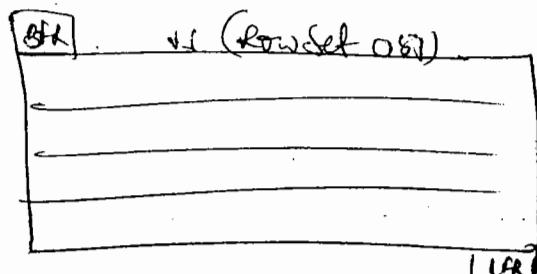
public RowSet fetchEmpDetails( long dept)

```

```

Rowset rs = qf.executeQuery("select * from emp
                           where deptno = ?",
                           new Object[] {new Long(dept)} );
return (Rowset)rs;

```



```

public boolean registerEmp( int empno, String ename, String job
                           int sal)

```

```

int res = qf.executeUpdate("Insert into emp(empno,ename,job,sal)
                           values(?, ?, ?, ?)",
                           new Object[] {new Integer(empno),
                           ename, job, new Integer(sal)} );

```

If (res == 0)

return false;

else

return true;

```
public boolean fireEmp( int valrange ) {  
    int res = qf.update( "delete from emp where val >= ",  
        new Object[] { new Integer( valrange ) } );  
    if ( res == 0 )  
        return false;  
    else  
        return true;  
}
```

```
public boolean hikeSalary( int end, int percentage ) {  
    int sal = qf.queryForInt( "select sal from emp where  
        empno = ? ", new Object[] { new Integer( end ) } );  
    float newSal = sal + ( sal * ( percentage / 100 ) );  
    int res = qf.update( "update emp set sal = ? where  
        empno = ? ", new Object[] { new float( newSal ),  
            new Integer( end ) } );  
    if ( res == 0 )  
        return false;  
    else  
        return true;  
}
```

## Demo Client.java

```
public class DemoClient  
{  
    public static void main(String args[])  
    {  
        ClassPathXmlApplicationContext ctx =  
            new ClassPathXmlApplicationContext("Demo.xml");  
        Demo bob = (Demo) ctx.getBean("ds");  
  
        System.out.println("Count of Emps" + bob.getCount());  
        System.out.println("First Emp salary" + bob.fetchEmpSalary(1));  
        System.out.println("Emp Registered ?" + bob.registerEmp(100, "John",  
            "clerk", 1500));  
        System.out.println("First Emp Details are " + bob.fetchEmpDetails(1));  
        System.out.println("clerk Dept Emp details are" + bob.fetchEmpDetails("clerk"));  
  
        System.out.println("10th department employee details are ");  
  
        Set<Employee> rs = bob.fetchEmpDetails(10);  
        while(rs.iterator().hasNext())  
        {  
            System.out.print(rs.iterator().next().getFirstName() + " " +  
                rs.iterator().next().getLastName() + " " +  
                rs.iterator().next().getSalary());  
        }  
        System.out.println("Total Emp salary hired ? "+  
            bob.fetchTotalSalary(100, 10));  
    }  
}
```

To develop persistence logic in our class by using spring module then we need to inject JDBC template class obj to that DAO class.

### Ex APP

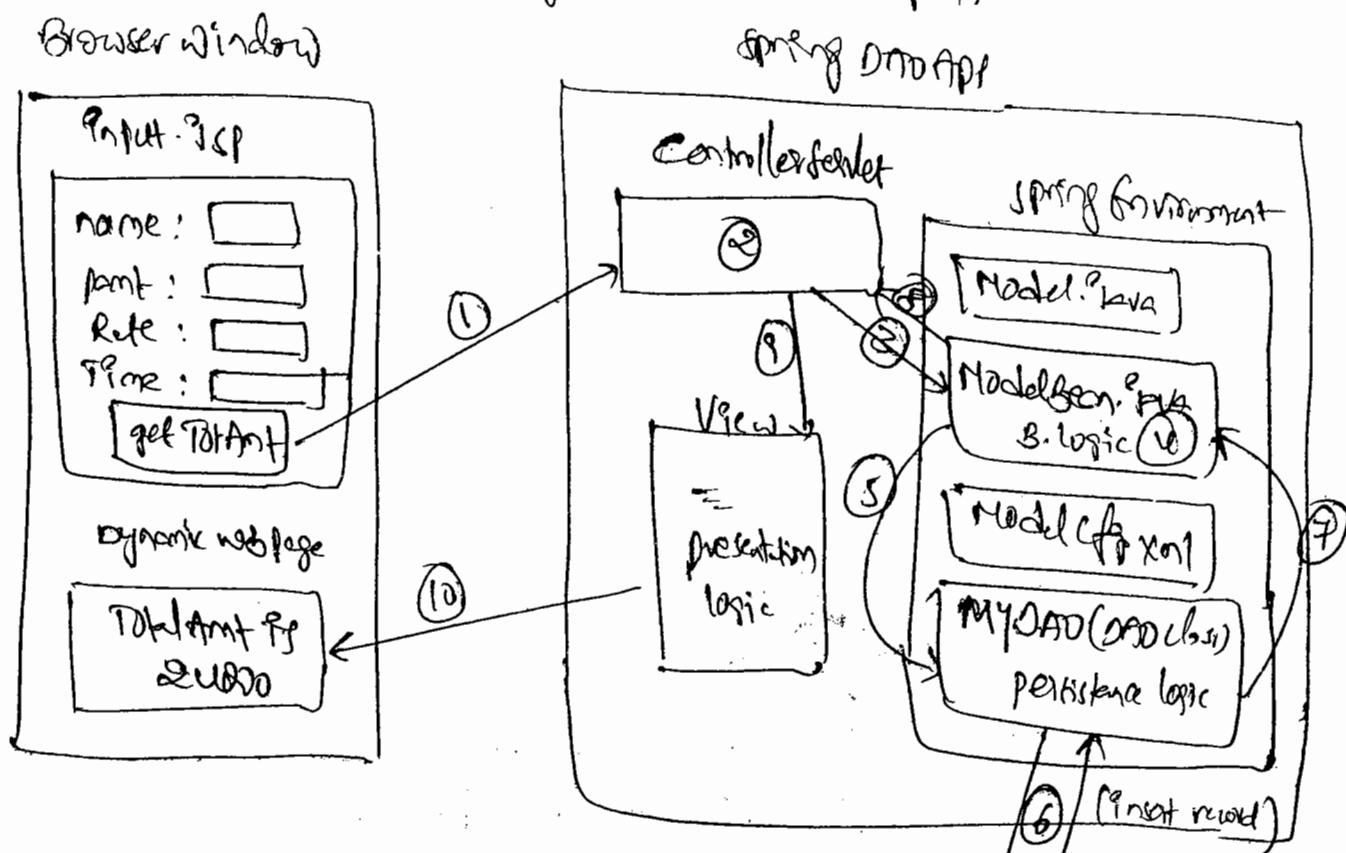
- MVC Arch Based web application.

Model → Spring Bean + DAO class

Controller → servlet

View → JSP

- User SpringDAO module based DAO class having persistence logic.
- Used container managed JDBC connection pool.



In the above app calculating interest amt is business logic and insert record into db table is persistence logic.

DB S/W db table I				
name	amt	rate	time	IntAmt
abc	10000	2	12	24000

procedure To develop above app using MyEclipse.

Step①: Create web proj having name SPRINGDADAPP.

Step②: Add Spring Capabilities to the project having

Spring 2.5 persistence,

Spring 2.5 core library,

Spring 2.5 Relational Jdbc library and also

choose → next → modelcfg.xml or springcfg file.

Step③: Add weblogic.jar file to the buildpath of project.

Note: The .jar files added to myComputer's classpath are not  
visible and accessible to the proj in IDE, so we must add like above.

24/10/11

Step④: Create spring-customer table in oracle db (sql)

> create table spring-customer (ename varchar(15), dept number(18),  
rate number(4), time number(4), amount number(4));

Step⑤: Develop DAO class in src folder (myDAO.java)

import org.springframework.jdbc.core.JdbcTemplate;

public class myDAO

{ JdbcTemplate jt;

public void setJt (JdbcTemplate jt)

{ this.jt = jt;

public boolean insertInto (String ename, int dept, int rate,  
int time, int amount)

{ int res = jt.update ("insert into spring-customer  
values (?,?,?,?,?)", new Object[] {ename,

new Integer(dept), new Integer(rate),  
new Integer(time), new Integer(amount))");

new Integer(amount))});

```

    if( ref == null)
        return false;
    else
        return true;
}

```

**Step④:** Add spring Interface , Spring Bean class to the project.

```

public interface Model {
    public int calcIntramt (String ename, int pamt, int time,
                           int rate);
}

```

**Step⑤:** Music class ModelBean implements model

```

{
    MyDAO dao = null;
    public void setDao(MyDAO dao) {
        this.dao = dao;
    }
}

```

```

public int calcIntramt (String ename, int pamt, int time,
                       int rate)
{
    int intramt = (pamt * time * rate) / 100;
}

```

// use the persistency logic of DAO class  
 dao.insertInto(ename, pamt, time, rate);

```

    return intramt + pamt;
}

```

}

**Step⑥:** Write following Configurations in spring Config file.

ModelConfig.xml :-

```
<beans>
```

```

    <bean id="jots" class="org.springframework.jdbc.ObjectFactoryBean">

```

```

        <property name="dataSource" value="DcJndi"/>
    </bean>

```

```

    <bean id="template" class="org.springframework.jdbc.core.JdbcTemplate">

```

```

        <property name="dataSource" ref="jots"/>
    </bean>

```

```
</beans>
```

```

<bean id="dao1" class="myDAO">
    <property name="rt" ref="template"/>
</beans>

<bean id="m" class="ModelBean">
    <property name="dao" value="dao1"/>
</beans>

</beans>

```

Note: If the application wants to communicate with Grid registry of server resides outside the server then there is no need of applying Jndi properties.

Step①: Add Input.jsp to webroot folder of the project.  
 Right click on webroot folder → new → JSP → Input.jsp

Input.jsp:-

```

<form method="get" action="controller" name="f1">
    Name: <input type="text" name="name"/> <br>
    Amount: <input type="text" name="amount"/> <br>
    Rate : <input type="text" name="rate"/> <br>
    Time : <input type="text" name="time"/> <br>
    <input type="submit" value="getTotalAmount"/>
</form>

```

<Form>

Step②: Add Controller servlet to the project src folder.

Right click on src → add → Add servlet → ControllerServlet → Select doPost → next → mapping url: /controller → finish.

```
public class ControllerServlet extends HttpServlet
```

```
{ model obj = null;
```

```
    public void init()
```

```
{ ClassPathXmlApplicationContext ctx = new
```

```
ClassPathXmlApplicationContext("model.cfg.xml");
```

```
// get Spring Bean obj from container
```

```
} obj = (Model) ctx.getBean("obj");
```

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
```

```
{ throws ServletException, IOException
```

```
// Read form data
```

```
String name = Integer.parseInt(req.getParameter("name"));
```

```
int amt = Integer.parseInt(req.getParameter("amt"));
```

```
int rate = Integer.parseInt(req.getParameter("rate"));
```

```
int time = Integer.parseInt(req.getParameter("time"));
```

```
// general settings
```

```
PrintWriter pw = response.getWriter();
```

```
response.setContentType("text/html");
```

```
// call business method of spring bean
```

```
int totalAmt = obj.calcIntAmt(name, amt, time, rate);
```

```
// keep the result to request attribute
```

~~request.setAttribute("total", totalAmt);~~

~~request.setAttribute("total", totalAmt);~~

// send request to result page

```
RequestDispatcher rd = req.getRequestDispatcher("/result.jsp");
rd.forward(req, res);
}
```

```
public void doPost(HttpServletRequest req, HttpServletResponse
    res) throws ServletException, IOException
```

```
{ doGet(req, res);
}
```

```
public void destroy()
```

```
{ web = null;
}
```

Step ⑨: Add result.jsp to the webroot folder of the project.

<b> The Total Amount : </b> <?> = request.

getAttribute("total") %>

Step ⑩:- Configure string batch domain of web logic with myeclipse  
window menu → preferences → myeclipse → servers →  
weblogic → weblogic 10.x → enable →

BEA home directory :

WL Installation directory :

Admin Username :

Admin password :

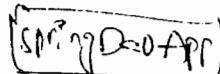
executing domain : .

→ Apply → OK

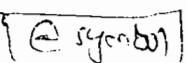
Step 11: Select server from  in the Tool bar →

Weblogic 10.3 → Start

\* Deploy the project in weblogic server

Go to deploy icon of the tool bar  Project: 

→ add server: weblogic 10.x → finish → OK

\* Open browser window:  and type URL.

Procedure to Configure Glassfish Server:-

Windows → preferences → My Eclipse → servers →  
Glassfish → Glassfish 2.x →  enable →

\* The webapps of Glassfish should be accessed by  
using port No: 8080 and it can be changed through  
Glassfish - Home | appserver | domain | domain 1 | config |  
domain.xml file related <http> > tag port attribute.

25/July

JDBC supports only positional parameters as part of their queries.

where as spring DAO allows the programmer to specify named parameters and positional parameters in SQL queries.

To work with named parameters we can use NamedParameterJdbcTemplate class.

The NamedParameterJdbcTemplate class is no way related with the regular JdbcTemplate class.

To supply named parameter values we can work with various methods of MapSqlParameterSource class.

SingleConnectionDataSource class also represents a dummy JDBC connection pool. This class super class is DriverManagerDataSource class.

The connection obj of DMS related connection pool will be closed automatically at the end of the execution but, while working with SCDs we need to close connection obj explicitly ~~so~~ it will not be closed automatically.

for example app on named parameters refere app@ of material.  
(Here no stringContainer feature is taken in app development)

SqlTemplate class based SQL queries can have only positional parameters, where as NamedParameterJdbcTemplate class based queries can have named parameters.

We can not place both named and positional parameters in single SQL query of spring DAO environment.

If certain functionalities are directly not possible with JdbcTemplate class then spring DAO allows us to implement that

functionality through Callbacks; Interface Implementation.) During this implementation we can take the support of plain JDBC API.

Some important callback interfaces of Spring DAO module:-

- ResultSetExtractor → To deal with total ResultSet obj
- RowMapper → To deal with single row
- PreparedStatementCreator
- PreparedStatementSetter ... etc. } to works with PreparedStatement obj

for example app on springDAO based Select operation with Callbacks Interfaces Implementation refer App ⑦ of book.

The queryForObject() of SpringDAO environment allows the programmers to gather select query execution result in programmer choice format/objects. (refer q10 to q13 of app).

Use the RowMapper callback interface implementation in order to store the selected single record of SQL select query in application specific userdefined Java class obj

Use ResultSetExtractor callback interface to copy (multiple selected) records of resultset obj to programmer choice object like ArrayList

Use PreparedStatementCreator callback interface to execute given SQL queries through PreparedStatement obj.

While implementing all these callbacks interfaces of SpringDAO with plain JDBC code, there is no need of performing Exception Handling bcoz the JdbcTemplate class takes care of that process.

Note :- If you want to make any queryForXXX() executing of query without parameters by using JDBC PreparedStatement obj then pass "null" at the second argument value. (queryForInt(query, null) → PreparedStatement)

~~sol~~ procedure to develop Spring app (1<sup>st</sup> app) by using NetBeans

Step①: Create Java project in NetBeans IDE having name TestProj.

Step②: Add Spring libraries to the project.

Right click on Libraries → add library → Spring framework 2.5 → add library

Note:- The above step adds spring.jar, commons-logging.jar files to Libraries.

Step③: Add Spring Interface to the project.

Right click on Project → new → Java Interface → Name: Demo → finish

public interface Demo

```
{ public String sayHello(); }
```

Step④: Add Bean class to project.

Right click on Project → new → class → Name: DemoBean → finish

public class DemoBean implements Demo

```
{ String msg;
```

public void setMsg(String msg)

```
{ this.msg = msg; }
```

public String sayHello()

```
{ return "GoodMorning" + "msg = " + msg; }
```

```
}
```

Step⑤: Add Spring Config file to project

Right click on Proj → new → other → XML → XML document → next →

Name: DemoCfg.xml folder: [src] → next →  DTD constraint → next

Spring DTD Beans 2.0. Document Name: beans → finish → write

The Step⑥ following code in DemoCfg.xml

```
<bean id="db" class="DemoBean">
```

```
    <property name="msg" value="hello"/>
```

```
</bean>
```

(as recommended to clear doctrine midvalue in DemoCfg.xml file)

Step ⑥: Add client application to the project

Right click on project → new → Java class.: DemoClient.

public class DemoClient

{  
    Pstmt (String args[])

( press + tab + space  
    + )

Pstmt (String args[]))

classpathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("demo.xml");  
Demo bob = (Demo) ctx.getBean("bob");  
} } bob.sayHello();

Step ⑦: Run the client application

Right click on DemoClient.java source code → run file

27/01

To centralize persistence logic or to make persistence logic as reusable logic by keeping those logics in db then take the support of PL/SQL procedure / function.

Spring DAO environment allows the programmer to call PL/SQL procedure or function.

The subclass object of org. if. Jdbc. object. StoredProcedure class can represent a PL/SQL procedure/ function.

This StoredProcedure class use execute() using which we can call PL/SQL procedure / function.

- public Map execute(Map params) throws Exception

Represents the return values      Represents the parameter values of PL/SQL of PL/SQL procedure/ function.

procedure/function

Some Inherited methods of StoredProcedure class:

compile() → compiles the query that calls PL/SQL procedure/function,

setFunction(true/false) → represents whether the subclass object of SP class points to PL/SQL

& parameter(parameter) → adds parameters that are registered with Jdbc datatype to the subclass obj of SP class.

By using `setOutParameter` class object we can register out parameters of pl/sql proc/fun with JDBC Types / SQL Types.

Similarly By using `SQLParameter` class we can register the In parameters.

\* for example of an spring DAO based pl/sql proc/fn refer off ⑯

Conclusion on : Spring DAO :-

Spring DAO persistence logic is db independent, bcoz it uses db specific SQL queries.

All ORM tools like Hibernate can be used to develop db independent persistence logic.

Spring ORM module allows the programmer to develop db independent persistence logic by providing abstraction layer on all ORM tools.

Java Mail API :-

- DB tools are responsible to maintain Data
- Registry tools are responsible to maintain obj's and obj references.
- Mail server tools are responsible to maintain email accounts and email messages.

e.g.: James Mail server, SMTP mail server, Lotus Notes, Microsoft Exchange.

Note: db tools and web servers/app servers are not responsible to maintain email accounts and email messages.

To talk with mail servers the Java applications use JavaMail API.

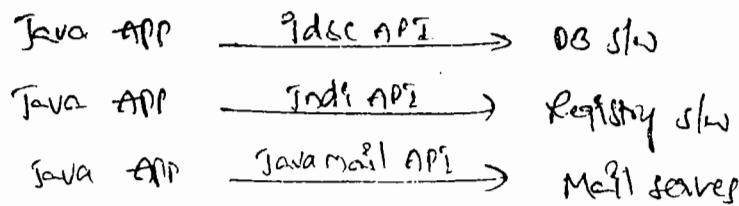
Java Mail API is not JMS.

Java Mail API:

`javax.mail.*` (is part of Jee module)

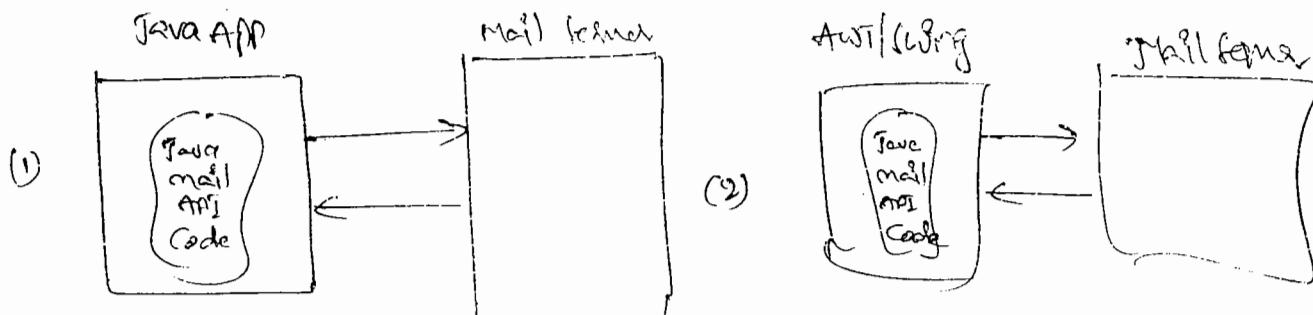
`javax.mail.internet.*`

`javax.activation.*` (JAF)



Ex(10/11)

We can develop different types of client applications using Java Mail API to interact with mail server and to perform send mail, read mail, delete mail operations on email accounts.



Every Mail server contains one Incoming server and one Outgoing server. Incoming server is responsible to receive email messages and to store them in the inbox of email accounts. whereas Outgoing server is responsible to send email messages from one ac to another ac.

Microsoft Outlook, Microsoft Outlook express, etc are the Mail Client s/w's to interact with mail servers.

Based on instructions received from Mail Client the Outgoing server takes the responsibility of sending email message s/w two email accounts of same mail server or diff mail servers.

In Company Environment employees generally configure the company supplied email account with Mail client like Outlook Express and use that to check the emails as the first work of the everyday.

Mail client app/slw's use the protocol SMTP to interact with Outgoing server, and POP3 or IMAP to interact with Incoming server

The POP3 based Incoming server delete the mail from inbox and sends to mail client machine once mail client reads the email msg from inbox.

IMAP based Incoming server maintains email messages forever even after they are read by mail clients.

In company level Internet Mailing system the POP3 based Incoming server will be used. In large scale email operations environment like Gmail, yahoo, the IMAP based Mail server is used.

In company environment the programmes will get the assignment through mails, but these mails related email accounts will be managed by IMAP based Incoming server.

### James : (open source)

Type : Java based Mail Server

Version : 2.2 (compatible with JDK 1.4+)

Vendor : Apache SW foundation

ports : 143, 25, 465, 587

[slw : www.apache.org]

(zip file)

Incoming server (POP3) : 110

Outgoing server (SMTP) : 25

Ports : 113, 465, 587

To install James Mail Server we extract James-2.2.0.zip file.

\* To maintain news items we need news server and to communicate with that server we use the protocol, NNTP.

To start James Mail Server use run.bat file of bin directory.

In James Mail Server since "fetch pop" is disabled, the inbox maintains email messages even after they are read by mail client.

Procedure to create Gmail accounts by James Mail Server :-

Step① : Start James Mail Server.

Step② : Launch Telnet tool (built-in tool of windows)

Start → Run → Telnet

Step③ : Connect to remote manager service of James Mail Server from TelNet.

Telnet > Open localhost 4555 ↵

Login id : root

Password : root

Step④ : Add new users (email id's)

addUser Nani 002

addUser Kkr rddy

listUsers : display the list of all users

del user Nani : delete the user nani

verify Nani : verify whether Nani is there or not.

quit : to exit.

Javamail API is not part of JSE module. It is part of JEE module.

And that Jee is not an installable file, bcz it is given as a specification containing set of API's, rules and guidelines to develop web server, app server etc. So every web and app layer should contains jar files representing Jee API's.

- In weblogic server the weblogic.jar represents Jee API's (but not Java Mail API).
- In Glassfish servers javamail.jar represents Jee API's (With JavaMail API)
- In JBoss servers JBoss-Javamail.jar represents Jee API's (without Java mail API)  
(In this it is as mail.jar)

29/10/11

A session object of JavaMail API based coding represents connectivity b/w Java app and Mail servers and this session obj is not at all related with HttpSession obj of servlet programming, session obj of JSP programming and the session obj of Hibernate programming.

To create session obj of JavaMail programming we need mail properties. These mail property names are fixed, but values will be changed based on the Mail server we use.

1) Mail.transport.protocol.

2) Mail.smtp.host }      u) Mail.pop.host }

3) Mail.smtp.port }      u) Mail.pop.port } Incoming server  
out going .

while creating session obj of JavaMail programming we need to pass the Mail properties and their values (related to Incoming server/ Outgoing server) as key-value pairs in the elements of java.util.properties class obj.

→ write a JavaMail API based application to establish connection with James mail server's Outgoing server?

Step①: Start James Mail Server

Step②: Add Glassfish Home/ AppServer/ lib/ jar file to the classpath

Step③: Develop Java application having JavaMail API code as shown below.

ConnTest.Gava

## public class Contest

<sup>2</sup> `Print(String args[])` throws exception

1

Properties P = New Properties (7)

```
p.put("mail.transport.protocol", "smtp");
```

```
Put("mail.smtp.host", "localhost");
```

\$.put("mail.smtp.port", "25").

```
function fcs = fcsim.getInstance(p);
```

if ( \$es == null )  
    ↑  
    (factory method)

5.0.1 ("Connection not established")

else

5.0. pl" Connection established"),

1

Step 10:- Compile and execute the Application.

→ `fwax-mail.internet`. InternetAddress class obj can represent one email address / email address.

To create email message programmatically, instantiate ~~JavaMail~~  
JavaMail Internet MimeMessage class and fill that Obj with data  
(headers and body).

The `Transport.send()` of JavaMail API can be used to send email message from one Email account to another email accounts.

for example app on play JavaMail API refer app! (13)

31/10/11

Inorder to delete the final messages of 'Inbox' then marks them for deletion and close the inbox with flag "true".

When inbox is closed with "true" flag all marked messages will be deleted.

`myinbox.close(true);`

When inbox is closed with flag "false" - then email messages will not be deleted even though they are marked for deletion.

`myinbox.close(false);`

**SpringMail :-**

Spring mail provides abstraction layer on Java Mail API and simplifies the process of mail operations.

As of now SpringMail is designed only to perform mail sending operations. That means we can not perform Read mail, delete mail operations.

Spring Mail API means working with ~~java.org~~.st-mail package, org.st.mail.javamail package.

org.st.mail package gives Spring's own ~~infrastructure~~ Infrastructure for dealing with Mails.

org.st.mail.javamail ~~uses~~ → Spring internally uses javaxmail API for dealing with mails.

org.st.mail.SimpleMailMessage allows us to design <sup>Simple</sup> email msg (without attachment) whereas org.st.mail.javamail.MimeMailMessage class allows us to deal with more complex email messages like message with attachment.

While working with SpringMail questions we can deal with Spring's dependency injections by preparing sender obj's and Mail message.

org. sf. mail. JavaMail! JavaMail SenderImpl class obj can be used to send email messages from one ac to another account. It underlying uses JavaMail API support.

o/p 11 To add attachment to the body of Email message prepare that email msg body with multiple parts and each part by containing text data or image data attached file.

Ex on sending email with attachment refer @ APP ① of o/p 11/11

To send Email msg with attachment we need to use the callbacks interface org. sf. mail. JavaMail. MimeMessagePreparator and class called org. sf. mail. JavaMail. MimeMessageHelper class in SpringMail environment.

Ex on send email with attachment refer @ APP ② o/p 11/11

The outgoing server details of Gmail.com mail server:-

mail.smtp.host → smtp.gmail.com

mail.port → 465

mail.smtp.auth → true

mail.smtp.socketFactory.class → javax.net.ssl.SSLSocketFactory

while creating service obj that point to the Mail server of network environment like smtp.gmail.com we need to pass an authentication class obj along with mail property.

javax.mail.Authenticator is an abstract class, so its sub class required while creating the service obj.

Q3/11/11

## The Incoming Server details of gmail.com

mail.pop3.host → pop.gmail.com

mail.pop3.port → 995

mail.pop3.auth → true

mail.pop3.socketfactory.class → javax.net.ssl.SSLSocketFactory

Ex: tip on gmail.com refer Handout given on 02/11/11.

\* procedure to configure gmail with microsoft outlook:-

Step①: login to gmail.com website's email account.

Step②: change settings of Email account.

settings → mail settings → forwarding and POP/IMAP →  enable POP for all mails → save changes.

Step③: launch and work with Microsoft Outlook 2007.

Start → programs → ms office → outlook 2007 → next →  Yes → next →

Your Name: Nani Email address: nani002@gmail.com : pwd: 111443

re-pwd: 111443 → next → finish.

\* To Compose Mail through outlooks.

file → new → mail message → To: nani@gmail.com →

subject: resume body: - - - → Attach symbol (To add attachment with mail) → send

\* password protection for microsoft outlook.

Tools → Account settings → data file tab → settings → change pwd →

new pwd: nani002 → verify new pwd: nani002 →  → OK

→ for 03/11/11 to 03/11/11 data refer 0st page to 13<sup>th</sup> page of xerox (back side)

⑦ Step ⑩ - Add form page Input.html to the webroot folder of project

Right click on webroot folder → html → [Input.html]

```
<form action="controller" method="get">  
    <input type="text" name="name"/>  
    <input type="text" name="param"/><br/>  
    <input type="text" name="date"/><br/>  
    <input type="text" name="time"/><br/>  
    <input type="submit" value="getTotalAmount"/>  
</form>
```

Step ⑪: Add controller servlet to the src folder of the project.

Right click on project → servlet → [ControllerServlet] → select HttpServlet

destroy(), doGet(), doPost() methods → next → servlet(Servlet mapping)

URL: /controller → finish

Public class ControllerServlet extends HttpServlet

```
{  
    Model model = null;  
  
    public void print()  
    {  
        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext  
            ("springcfg.xml");  
        model = (Model) ctx.getBean("model");  
    }  
}
```

public void doGet(HttpServletRequest req, HttpServletResponse res)

throws ServletException, IOException

Read form data

String name = request.getParameter("name");

```
int amt = Integer.parseInt(request.getParameter("amt"));
int rate = Integer.parseInt(request.getParameter("rate"));
int time = Integer.parseInt(request.getParameter("time"));

int totamt = bobl.calcIntAmt(name, amt, rate, time);
```

// send result to result.jsp page

```
request.setAttribute("res1", totamt);
```

// forward request to result.jsp

```
RequestDispatcher rd = request.getRequestDispatcher("result.jsp");
```

```
rd.forward(res, res);
```

}

```
public void doPost(, ) throws (, IOException)
```

```
{  
    doGet(res, res);  
}
```

```
public void destroy()
```

```
{  
    bob = null;  
}
```

}

Step 18 Add result page to the webroot folder of the project.

Right click on web root folder → new → JSP → result.jsp

<6> The result is : <6> res = request.getAttribute("res1")/>

Step 19: Start Tomcat server from MyEclipse IDE

Step 20: Deploy web app to Tomcat server from MyEclipse

Step 21: Test the web app from the browser window of MyEclipse

http://localhost:8080/springH3APP3/input.html

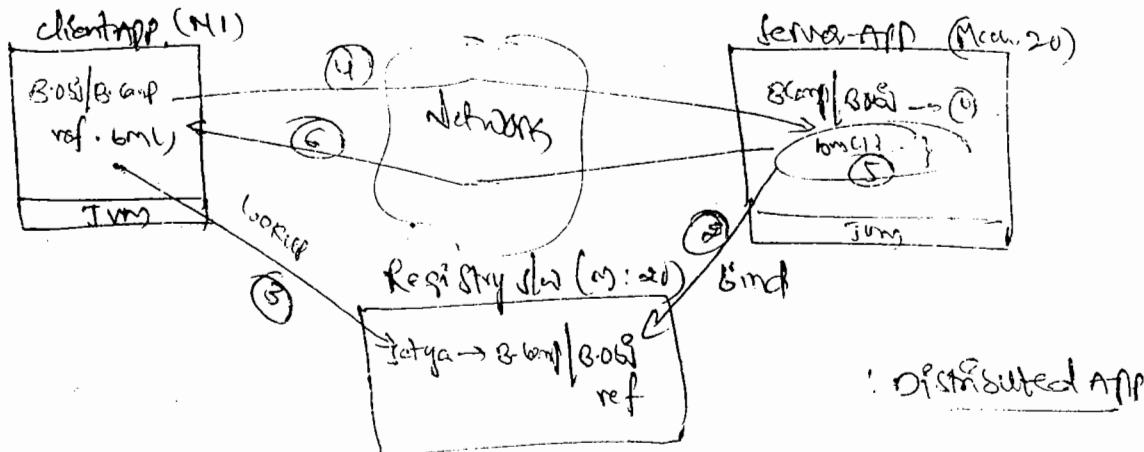
## Spring JEE Module:

Spring JEE module is given having the following facilities.

- 1) gives ApplicationContext container
- 2) gives Spring Indi providing abstraction layer on plain Indi
- 3) gives Spring mail, providing abstraction layer on plain JavaMail API
- 4) Gives environment to enable scheduling on spring-applications
- 5) Gives abstraction layer to the following plain distributed technologies to develop distributed application.
  - i) spring RMI provides abstraction layer on plain RMI.
  - ii) spring EJB " " plain EJB.
  - iii) spring webservices " " plain web services.
  - iv) spring Hessian, and BURLP .. plain Hessian, Burlp.
- 6) Gives its own direct distributed technology called  
Http Invoker.
- 7) Gives spring JMS providing abstraction layer on plain JMS  
to develop application that communicates through message-  
queuing programming based client-server app and JMS apps  
interacting with also are called as traditional client-server apps.  
These traditional client server apps are location dependent  
apis bcoz any change in the server app location must be  
performed to all the client applications. Then only they can  
access the business logic of methods available in business  
component of server application.

The server app of traditional client-server app allows remote clients but the whole app itself location dependent.

To solve the location dependency problem develop your applications as distributed applications.



Distributed Applications are location transparent w.r.t of Registry serv. The reason is no client is directly communicating with server application, they are getting business obj ref from Registry and using that ref to communicate with server app.

So if any change is there in server app location or other details we need to inform to all the clients. If we just inform to registry serv all clients will receive those details dynamically.

With respect to the diagram in server app the business obj will be developed having business logic in business methods.

(1) server app binds B-Obj with registry serv having nickname.

(2) client app gets B-Obj ref from registry through lookup operation.

(3) client app calls bmethod on business obj reference.

(4) The bmethod of B-Obj available in server app executes.

(5) The results generated by bmethod goes to client app.

10 14/11/11

The class that extends from Java.util.TimerTask's class can have a task or logic that can be scheduled for one time or repeated execution by a Timer.

Java.util.Timer class is given to enable Timer service on given tasks (use schedule() method of this class).

for example app on scheduling by using Jobs level concepts refer off: ⑫

Spring support scheduling operations, it internally uses quartz algorithm for this scheduling. The Spring scheduling gives following benefits when compared to jobs level scheduling.

1. Allows to enable scheduling on multiple tasks at a time.
2. Allows to enable scheduling on user defined methods of user defined classes. (class need not to extend from TimerTask class and any name can be taken for methods).

The org.springframework.scheduling.timer.ScheduledTimerTask is a Spring Bean that represents a task on which timer service can be enabled having initial delay, period parameters, but this task must extend from all tasks defined in a Java class that extends from Java.util.TimerTask class.

The org.springframework.scheduling.timer.TimerFactoryBean starts timer service on the specified task. This Bean initializes the Timer on the startup of application context container and cancels the Timer on the destruction of application context container.

for ex application i on spring based scheduling refer off ⑬.

The org.springframework.scheduling.timer.TimerTaskFactoryBean is a factory Bean that returns TimerTask class object pointing to specific user defined method of user defined Java class.

That means it converts ordinary POJO class into TimerTasks enabled obj.

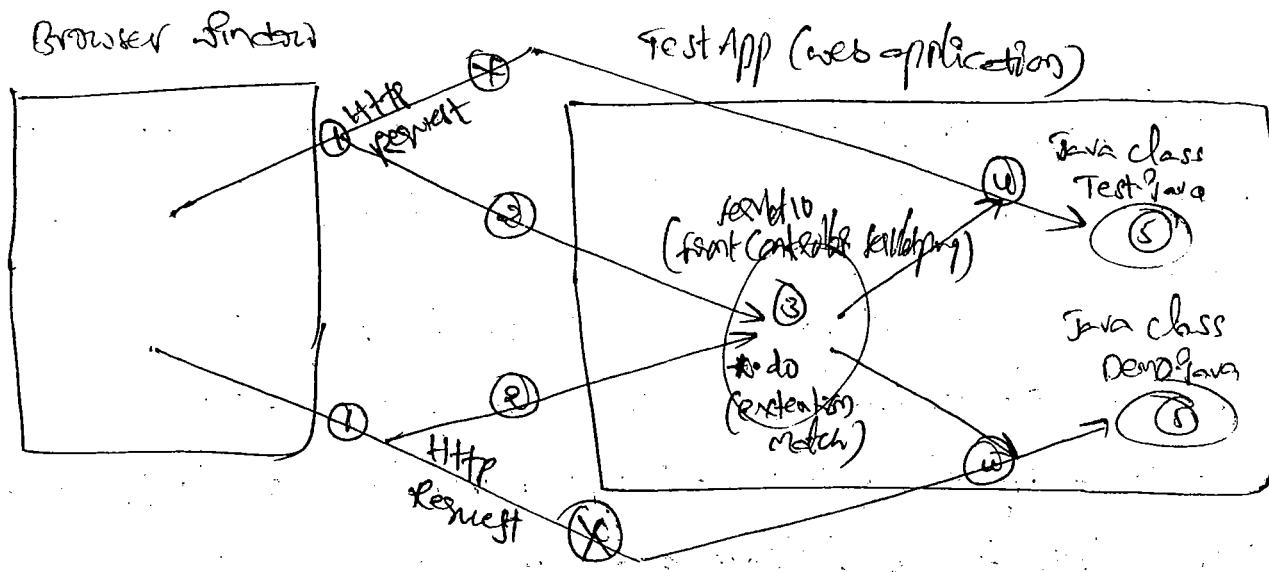
for example application ② on Spring-based Timer app refer off ②

1) URL

A special web resource program in the web application that is capable of trapping and taking request going to other web resource programs or resources of the web application is called as front controller.

A front controller servlet program or JSP program must be configured in web.xml file with extension match or directly matches URL pattern.

The Java classes of web application can not take HttpServletRequest directly. So it takes the support of a front controller servlet/JSP to trap that request and to pass that request to Java class.



In Struts application ActionServlet will be taken as front controller servlet to take the HTTP request from browser window and to pass them to the Java class called struts Action class.

In spring based web applications a predefined servlet called `org.springframework.web.servlet.DispatcherServlet` will be taken as front controller servlet to take the `HttpServletRequest` from browser window and to pass them to the Java class of web application.

When servlet container instantiates `DispatcherServlet` it automatically activates web application context container taking `<DispatcherServlet logical name>` - `web.xml` as spring configuration file.  
Ex If DispatcherServlet logical name in `web.xml` is abc  
then spring config file name is "abc-servlet.xml"

`DispatcherServlet` uses one or other `url-handlers` mapping done in spring config file (like `SimpleUrlHandlerMapping`) to pass the trapped http request of browser window an appropriate Java class of web application.

In `spring.cfg.xml` :-

```
<bean id="curl" class="org.springframework.web.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <mappings>
            <prop key="my.do">/xyz</prop>
            <prop key="xyz.do">/abc</prop>
        </mappings>
    </property>
</bean>

<bean name="/xyz" class="MyDemo"/>
<bean name="/abc" class="MyDemo"/>
```

→ Java.util.Properties type property  
word by request url  
→ URL Handler mapping required  
for DispatcherServlet  
to locate the definition  
Java class for  
trapped request]

Note: According to above code DispatcherServlet passes trapped HTTP Request to MyDemo class when that request URL contains my.do word.

HTTP Invoker:-

- Spring's own distributed Technology to develop distributed app.
- web based distributed Technology.
- Uses Spring Container itself as registry svr.
- The server app of this distributed app must be developed as web application and client app must generate HTTP requests to that generates server app.

- The server app uses spring xml specified dispatcher servlet as front controller to trap the http request and to pass them to implementation classes.

\* The server application of HTTPInvoker uses predefined org.springframework.http.HttpInvokerHttpInvokerServiceExporter to convert ordinary Java class obj to HttpInvoker service obj and registers that service obj reference with Spring Container (Registry svr)

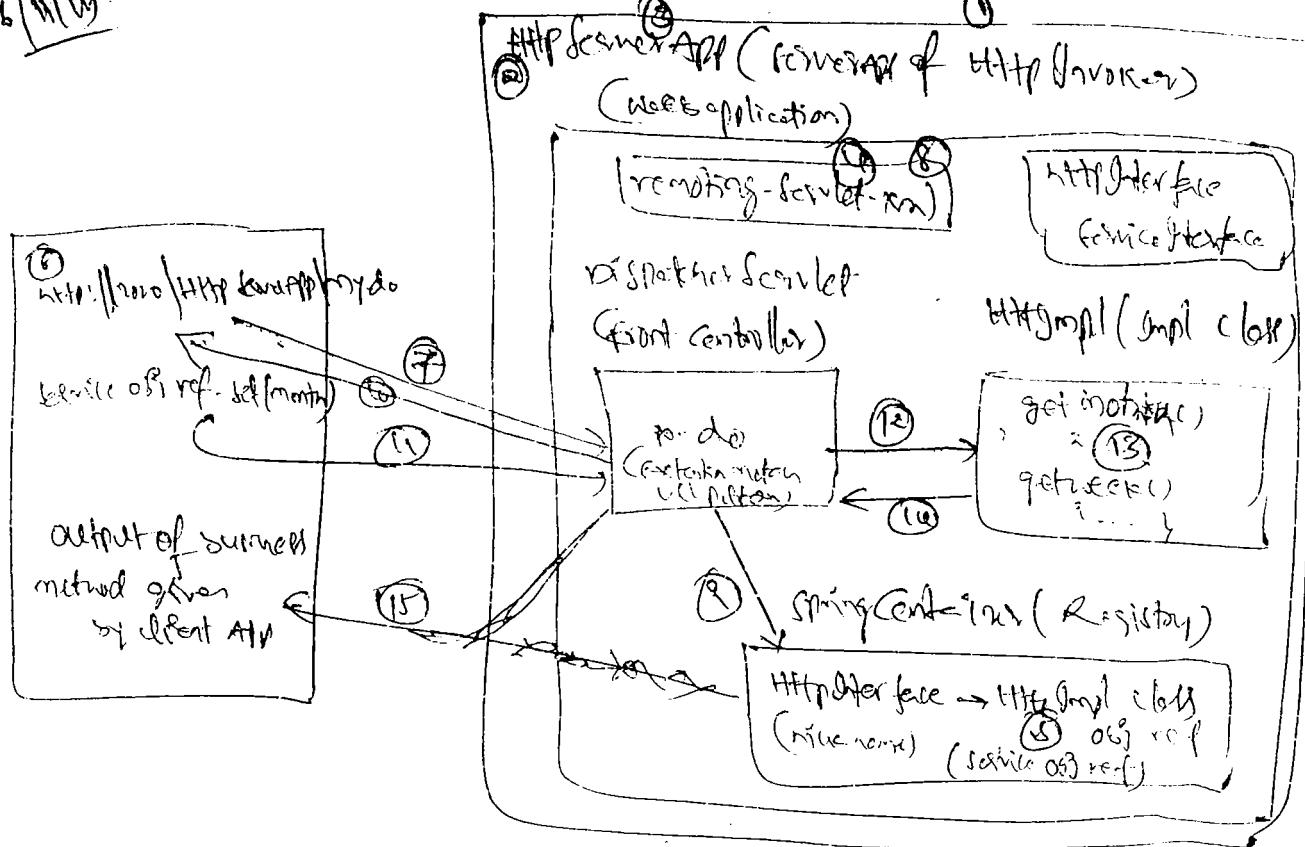
The client app of HTTPInvoker uses org.springframework.http.HttpInvoker

HttpInvokerProxyFactoryBean to get service obj reference from registry (Spring Container) and to inject that service obj reference to obj reference to a bean property of specified bean class.

Not only browser windows, we can see stand alone Java applications generating HTTP Requests.

# Tomcat Server (on my machine)

16/11/14



with respect to the diagram:

- ① Programer deploys `HTTP Server App` in tomcat server
- ② Based on loc of `HttpServlet` enable in `DispatcherServlet`, A servlet container initialize dispatcher servlet either ~~by config~~ during server startup or during the deployment of web app.
- ③ Dispatcher servlet internally activates app context container by taking `remoting-servlet.xml` as spring config file  
(D.S's logical name)
- ④ The activated app context container performs preinitialization on all the spring beans of spring config file.
- ⑤ During preinitialization process : The `Http Chopper` service explorer configured by spring config file registered

Service obj reference by Registry Inv for track, it converts http Uml class obj to service obj

⑥ Client App generates request via through Http Invoker proxy factoryBean.

⑦ Since D.S URL pattern is `/*.do` and client generate request via URL having my.`do`. The D.S traps and take the client req.

⑧ D.S URL mapping is done. Spring Config file to locate service obj ref in the Registry.

⑨ D.S gets service obj ref from Registry.

⑩ D.S sends service obj ref to client app.

⑪ Client app calls businessmethod() on service obj ref.

⑫ D.S prepares these trapped req to service

(Http Uml) service class obj.

⑬ The business method service class get() is executed.

⑭ Business method request came to D.S.

⑮ D.S returns the result to client app.

for the above diagram based Http Invoker ex app @ app ⑯

As a service client of Http Invoker based client app development gather following details from service provider.

1. service url to get the service obj ref from registry inv
2. A copy of service interface file.
3. Documentation on Business method.

Conclusion: prefer working with spring web services to develop distributed application, bcoz it allows us to develop interoperable & compatible giving the ability to write client app in any language like Java, .net, etc.

Next prefer Http Chopper to develop spring distributed application,

17/11/10

## Spring WEB Module

part 1 → Contains facilities and plugins to make spring apps  
communicable from web flow like based applications  
like struts app, Jsf app and etc..

part 2 → It is spring web mvc flow. It is given to develop  
mvc2 architecture based web applications. Spring web mvc  
is spring's own web flow to develop mvc2 webapps.

part 1 :-

struts and spring integration :-

In struts and spring integration struts app represents View  
layer and controller layer, whereas spring application represents Model  
layer. Struts app contains presentation logic and integration logic.  
Spring app contains business logic and persistence logic.

Struts and spring integration can be done in two forms

form 1) As struts with spring app

form 2) As struts app to spring app.

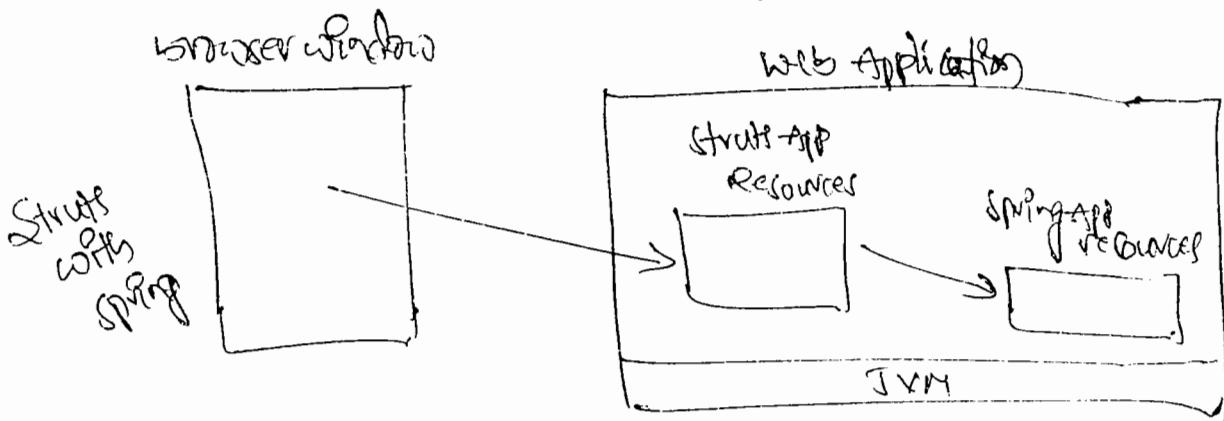
In form 1 :- Struts and spring app resides on the same JVM.

struts app acts as local client to spring. So spring app can be developed  
as non-distributed applications.

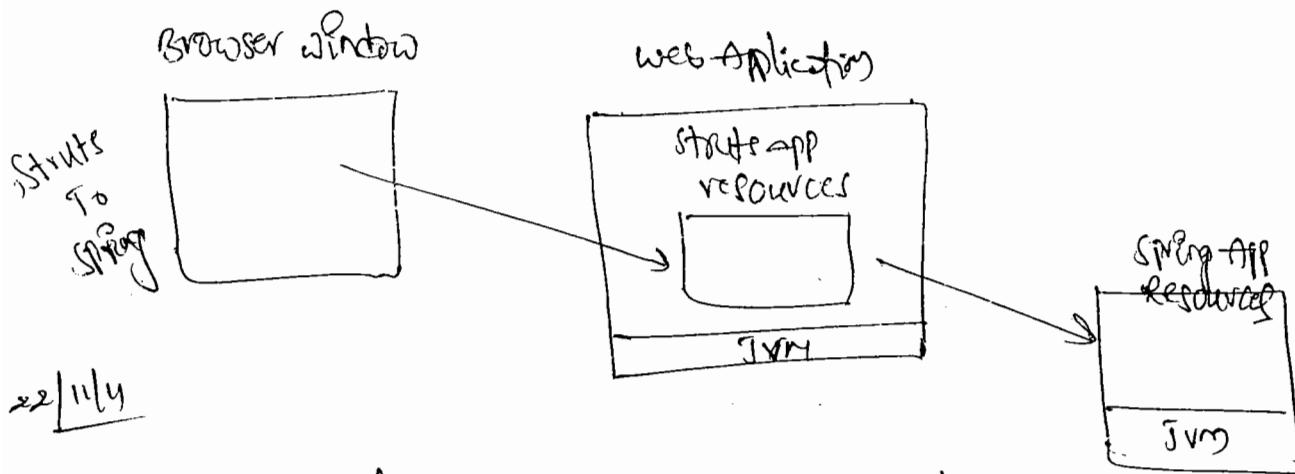
In form 2 :- Struts app and spring app resides on two diff JVM's.

struts app acts as remote client to spring. So spring app must be  
developed as distributed app (use spring jee module)

## form1 based Struts and Spring Integration.



## form2 based Struts and Spring Integration



In struts and spring integration based application we need to configure one spring plugin supplied plugin in struts config file, i.e. org.apache.web.struts.ContextLoaderPlugin. This plugin will be activated along with ActionServletInitialization during either server startup or during deployment of struts application.

This plugin internally activates web app context container by taking <ActionServlet logicalName> servlet.xml of spring configuration file. And performs preInstantiation on all the beans configured in spring configuration file.

Ex: In struts config file:

```
<plugin> class name = "org.apache.web.struts.ContextLoaderPlugin"/>
```

(It's to fix the fixed notation and error)

Ex

```
<plug-in classname="org.apache.struts.ContentLoaderPlugin">  
  <set-property property="contentConfigLocation"  
    value="WEB-INF/mycfg.xml WEB-INF/mvcfg.xml"/>  
</plug-in>
```

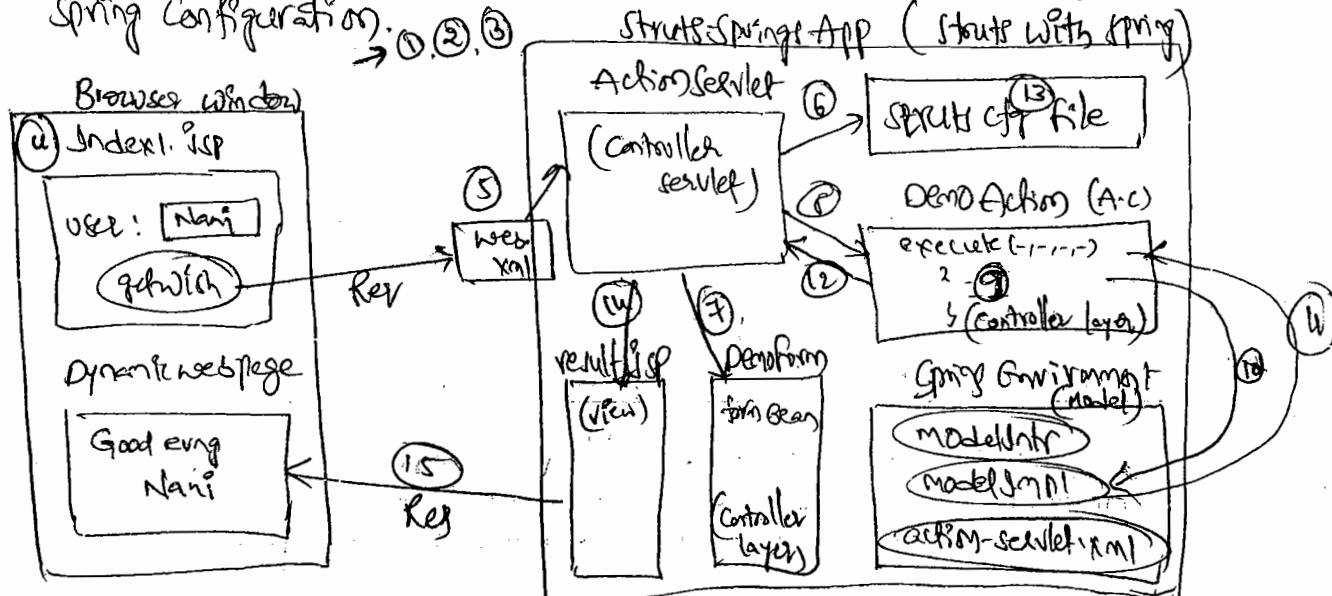
Here it allows us to take programmatic choice names based on one or multiple files as config files.

In Struts and Spring Integration based application we can activate SpringContainer manually in Struts Action class but it's not a recommended process. ~~so~~ It demands you to activate SpringContainer in every Struts Action class, so use Content Loader plugin to activate SpringContainer only once during server startup or deployment of Struts App.

There are two approaches to develop Struts-Spring Integration based applications.

Approach 1:- Make Struts Action class as Spring aware class with the support of ~~xxxSupport~~ classes, like -ActionSupport, DispatchActionSupport, LookupDispatchActionSupport .....

Approach 2:- By marking Struts Action class as Spring Bean in Spring Configuration.



with respect to the diagram

- ① Programer deploys Struts App in web server or App Server.
- ② Based on load-on-startup the Servlet Container creates obj of ActionServlet either during server startup or deployment of struts app.
- ③ ActionServlet activates ContextLoader plugin and this plugin activates Spring Container, this container performs preInstantiation.
- ④ Spring beans of struts cfg file.
- ⑤ End user launches the form page in browser window.
- ⑥ Based on the configuration done in web.xml file the ActionServlet traps and takes the request.
- ⑦ ActionServlet uses struts cfg file entries to decide the formBean and Action class towards processing request.
- ⑧ ActionServlet creates or locates ActionClass obj.
- ⑨ ActionServlet calls execute() of Action class, and gets spring bean class ~~obj~~ from Spring Container using diff techniques.
- ⑩ Action class calls b-method of Spring Bean from execute().
- ⑪ The b-logic generated result comes to execute() of Action class.
- ⑫ execute() returns the control to ActionServlet.
- ⑬ ActionServlet uses ActionForward Configuration of struts cfg file to decide the result page.
- ⑭ ActionServlet sends the control to Result page.
- ⑮ presentation logic of Result page formats the result and sends it to browser window as dynamic web page.

## ~~Required Team members~~

All ~~xxx~~Support classes of spring environment internally extends from ~~xxx~~Action classes of struts environment. So when we develop our Java classes extending from ~~xxx~~Support classes of, they not only become struts action classes, they also acts as spring aware classes.

ActionSupport class of spring API internally extends from Action class of struts API.

DispatchActionSupport class of spring API internally extends from DispatchAction class of struts API.

Procedure to develop above diagram based application by using Approach①:

Step①: Configure ContextLoaderPlugin in struts cfg file like previous.

Step②: Develop your struts action class (DemoAction) extending from org.apache.struts.ActionSupport class.

Step③: Get access to spring container, activated by ContextLoaderPlugin from execute() of Action class of struts and call the methods of Spring Bean.

public class DemoAction extends ActionSupport

{  
    public ActionForward execute(...,...)

        {  
            // get access to AppContext Container

            WebApplicationContext ctx = getWebApplicationContext();

            // get access to spring bean class obj from Spring Container

            ModelInt bobj = ctx.getBean("mb");

            // call some method of SpringBean class

            bobj.bmi();

        }

Step④ Develop SpringInterface, SpringBean and Configure them in Spring configuration file.

Step⑤ Develop resources of Struts App in a regular manner and configure them in Struts config file.

Step⑥ Add struts api and spring api related jar files to WEB-INF/lib folder (struts → 1.0 + spring → 2.0) of struts app.

[springtime/dist/module] → [spring-web-mvc-struts.jar]

for above diagram based approach 1 style Struts with Spring app

~~source code side~~ ~~app~~ (23) pgm 75-77

In Struts and Spring Integration based on approach ② the Struts Action class will be configured in Spring configuration file as SpringBean and the other original SpringBean class obj will be injected to Struts action class through dependency injection. So Struts action class can use that injected SpringBean class obj to call the methods of Spring Bean class. In this situation we configure "org.of.web.struts.DelegatingActionProxy" class in Struts config file as proxy for Struts Action class to receive the request from clients and to pass that request to the Struts Action class of Spring config file. (Spring managed Struts Action class).

In Struts cfg file:

```
<struts-config>
```

```
  <form-beans>
```

```
    <form-bean name="donotfm" type="org.of.web.StrutsDelegatingActionProxy"/>
```

```
  </form-beans>
```

```
  <action path="/demo" type="org.of.web.StrutsDelegatingActionProxy">
```

```
    <name>"donotfm"</name>
```

```
    <forward name="success" path="/result.jsp"/>
```

```
  </action>
```

In spring.cfg file :

In spring config file:  
<beans> → must match with the action path of  
DelegatingActionFactory class

```
<bean name="demo" class="deanplus.DemoAction">
```

<Property name="obj" ref="d1"/>

<(seen)>

<button pd="di" class="demopak-petondelImp1">

*(figure 8)*

## Demo Action: Java

public class Demolition extends Action

۲

Model Intr 1059

Public void setBook(ModelBook book)

} Thus  $\text{K}_1 = \text{K}_2$ ;

Public Action formed executive -

1) calling subroutines and other logic

3) String ref = bdsi.getFisch()

卷之三

24/11/04

procedure to develop state with spring off based on approach (2).

Step①: Develop spring interface, spring bean class and configure them in spring config file.

Step②: Configure contextloader plugin in struts configuration file  
(Refer previous discussion)

Step ③: Configure org. sf. web. struts delegating action proxy class by struts config file as proxy to receive original struts action class req.

Location path = "/demo", type = "org.st. web. starts. Deligating

```
<forward name="success" path="/reflect.jsp"/>
```

Step① Configure Struts Action class as spring bean in struts config file and inject other Spring Bean class obj to this Struts Action class property through dependency injection.

Step② call B method of Spring Bean class from execute() of Struts Action class by using the injected Spring Bean obj.

Step③ Develop the resources of the Struts app in regular manner.

Step④ Keep war files that represents Struts app and Spring app in WEB-INF/lib folder. (10 Struts + 2 Spring + 1 other).

\* for above steps based (Approach②) Struts with Spring application refer app② of page no: 73 to 75

for related information on Integrating Struts and Spring refer . page no. 26 to 34.

Note: use Approach① for developing Struts with Spring Integration based applications.

Developing Struts To Spring with Hibernate Application:-

With respect to the diagram:

- ① programmer deploys the HttpServlet Server App in web server as web application. bcos of preinitialization and Initialization process that takes place by server during initial server startup or during deployment of web app, the bootstrap reference (Proprietary obj) will be registered automatically with Registry (W).

# Struts Application



① programmer deploys struts app in webserver, due to initiliazation and preinitialization process the action servlet activated spring container through context loader plugin and that @ModelAttribute spring container performs preinitialization on spring beans of client Cfg.xml .

- ② form page submits the req to struts app
- ③ action servlet traps and takes the request of form page
- ④ as user struts Cfg file entered to decide from bean and action to process the request

- ④: A.S. writes form page form data to formbean class object.
- ⑤: A.S. calls execute() of A.C.
- ⑥ & ⑦: The client API of HttpInvoker gets business obj ref from registry given by server app and gives that b.obj ref to execute() of Struts A.C.
- ⑧: execute() of A.C. uses that b.obj ref and calls b.method of server app.
- ⑨ ⑩ ⑪ ⑫: The b.method of server app generates the results through b.logic (Gives gives total, avg, ranks for student) and uses A.B related DAO class to insert student details to db table.
- ⑬: Logic of b.method results of server app goes to A.C execute() method.
- ⑭: A.C returns the control to A.S.
- ⑮ ⑯: A.S. uses struts config file entries and forwards the control to result page.
- ⑰: The result page generates dynamic web page displaying formatted results.

26/04/11 procedure to develop the above diagram based Struts to spring with hibernate application.

- Part ①: developing HttpInvoker server Application
- Step ①: Create the following db table in oracle
- ```
SQL> create table fb-student(mno number(4) primary key,
      sname varchar2(20), total number(4) number(7),
      avg number(7,2), result varchar2(5));
```

Step②: Create db(explorer) profile for oracle by using Myeclipse db explorer

Step③: Create web project (HttpServerApp) in Myeclipse IDE.

Step④: Add Hibernate capabilities to the project.

Right click on project → Myeclipse → Add Hibernate capabilities

→ Hibernate 3.2 → next → Config file name: [hibernate.cfg] → Next

DGDriver: [oracle] (DB profile) → Pwd: [tiger] → Next →  deselect

Check box → finish → Add show-sql, autoCommit properties

Step⑤: Add spring capabilities to the project.

Right click on project → myeclipse → Add spring Capabilities →

① Spring 2.5 → Select core libraries, persistent core libraries, persistent jdbc libraries, remoting libraries, service libraries, web libraries →

Next →  deselect ADP builder → file: [remote-fewlet.xml] →

Next → Name of the session factory id: [session] → finish

Step⑥: Perform hibernate reverse engineering on HB-Student table to generate hibernate pojo class, mapping file, DAO class.

Right click on Graph → Right click → open connection → expand one

→ Right click on HB-Student table → Hibernate reverse engineering → Java source folder: [HttpServerApp/src] → select all the three main check boxes with spring DAO option → Next → ② Hibernate Types

Id Generator: [Increment] → Next → finish

The above step generates HBStudent.java, HBStudentDAO.java, HBStudent.hbm.xml dynamically.

Step⑦: Configure basic datasource class of Apache DBCP in spring.cfg file and inject the generated trnsaction manager to the property of local session factory bean configuration.

## Go remote - sevlet.xml :

```
<bean id="dbcp" class="org.apache.tomcat.dbcp.PoolingDataSource">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url" value="jdbc:oracle:thin:@localhost:1521/orcl"/>
    <property name="username" value="system"/>
    <property name="password" value="tiger"/>
</bean>
<bean id="service">
    <property name="dataSource" ref="dbcp"/>
</bean>
```

Step ⑦: Add the apache dbcp related tomcat-dbcp.jar file to the build path of the project.

Step ⑧: Develop service interface, implementation class of HttpServlet Server App.

### Model.java

public Interface Model

```
{ public String findResult(String name, int m1,
                           int m2, int m3); }
```

### ModelBean.java

public class ModelBean implements Model

```
{ StudentDAO dao =
```

1) self method for dao:

```
public String findResult(String name, int m1, int m2, int m3)
```

```
{ // logic
```

int total = m1 + m2 + m3;

```

float avg = total / 3.0f;
String res = null;
if (avg < 35)
    res = "fail";
else
    res = "pass";

```

// Use dao class persistence logic to insert record into db table.

```
HBStudent st = new HBStudent();
```

```

st.setName(sname);
st.setTotal(new Integer(total));
st.setAvg(new Double(avg));
st.setResult(res);
dao.save(st);

```

```
return res;
```

```
}
```

Step①: Configure Implementation class(model bean class) in spring cfg file. In remote-servlet.xml:

```

<bean id="ms" class="modelBean">
    <property name="dao" ref="HBStudentDAO"/>
</bean>

```

Step②: Configure dispatcher servlet in web.xml file having the logical name 'remote' and extension match url pattern '\*.\*.do'

Refer web.xml file of Page No: 66 of app ② ⑦

Step④: Configure `HttpInvokerServiceExporter`, `SimpleUrlHandlerMapping` classes in spring cfg file.

```

<bean name="/service" class="org.sf.remoting.HttpInvoker.HYSE">
    <property name="service" ref="ms"/>

```

```

<property name="serviceInterface" ref="model"/>
</bean>

<bean id="sh" class="org.springframework.web.servlet.HandlerServlet">
    <property name="mappings">
        <props>
            <prop key="my.do">/service</prop>
        </props>
    </property>
</bean>

```

Step ⑪ Configure Tomcat server with myeclipse IDE.

Step ⑫ Start Tomcat server and deploy this application.

with

part ② : Developing Struts Based web app as remote client to the above server app of HttpInvoker.

Step ①: Create web project having name StrutsApp

Step ②: Add Struts Capabilities to the project.

Right click on StrutsApp → myeclipse → Add Struts Capabilities  
 → select Struts 1.3 → Empty the base package text box → finish

Step ③: Add Spring Capabilities to the project.

Right click on StrutsApp → myeclipse → Add Spring Capabilities → Spring 2.5 → select core libraries, jee libraries, remoting libraries, miscellaneous libraries, web libraries → next → deselect App Builder → file: clientcfg.xml → finish.

Step ④: Add ClientInit, ClientBean resources to StrutsApp.

to hold and return EJB reference of http server application,  
 given by HttpInvokerProxyFactoryBean.

## clientIntf.java

```
public interface ClientIntf  
{  
    public Model getBoref();  
}
```

## clientBean.java

public class ClientBean implements ClientIntf

{

Model boref;

```
    public void setBoref(Model boref)  
    {  
        this.boref = boref;  
    }
```

```
    public Model getBoref()  
    {  
        return boref;  
    }  
}
```

Step⑤: copy Model.java from serverApplication to client  
street application's 'src' folder.

Step⑥: write following entries in config file (Clientcfg.xml)

```
<bean id="pb" class="org.hf.remoting.HttpInvoker.
```

HttpInvokerProxyFactoryBean>

```
    <property name="serviceURL" value="http://localhost:8080/  
    http://serverApp/my/do"/>
```

```
    <property name="serviceInterface" value="Model"/>
```

</beans>

```
<bean id="cb" class="ClientBean">
```

```
    <property name="boref" value="pb"/>
```

</beans>

Step②: Configure ContextLoader plugin in struts-config.xml file.

```
<plug-in class-name="org.apache.struts.ContextLoaderPlugin">
  <set-property property="contextConfigLocation"
    value="/WEB-INF/clientcfg.xml"/>
</plug-in>
```

Step③: Add form page (Input.jsp), form bean class (Inputform.java) and action class (InputAction.java) to strutsapp.

Right click on project → New → Other → myeclipse → web → Struts → Struts 1.3 → Struts 1.3 Form, Action & JSP → Next → Name:

Super class:  → formType:  → Add (The following formbean properties). → name, m1, m2, m3 → Jsp Tab

→  select the check box:  → Path:

Super class:  type:  → form tab → name:  → forwards tab → add → name: success → Path:  → Add → finish

Step④: Develop StrutsAction class as shown below.

InputAction.java:

1. Public class InputAction extends ActionSupport

2. Public ActionForward execute(...)

    InputForm ff = (InputForm) form;

    String name = ff.getName();

```
int m1 = Integer.parseInt(pf.getParam1());
```

```
int m2 = Integer.parseInt(pf.getParam2());
```

```
int m3 = Integer.parseInt(pf.getParam3());
```

|| get access to the context loader plugin activated spring container

```
WebApplicationContext ctx = getWebApplicationContext();
```

|| get access to Client Bean class object.

```
ClientImpl cb = (ClientImpl) ctx.getBean("cb");
```

|| get basic ref of server app from Client Bean

```
Model boRef = cb.getBoRef();
```

|| call b-method of server app

```
String res = boRef.findResult(name, m1, m2, m3);
```

|| keep result of b-method in request attribute to send to result page (result.jsp)

```
request.setAttribute("result", res);
```

```
} return mapping.findForward("success");
```

```
}
```

Step 10 : Add Result.jsp to the web root folder of the project.

Result.jsp :

```
<% Object result = request.getAttribute("result"); %>
```

Step 11 : Start Tomcat server and deploy the above StrutsApp project in that particular server.

Step 12 : Test the application

- 1. Make sure that http://serverIP:8080/StrutsApp/ is running mode.
- 2. Use following respectively for browser window.

http://localhost:8080/StrutsApp/

### Note! After part ①:

- 1) Move remote-servlet.xml to web-inf/ from src folder
- 2) remove myeclipse EJB generated libraries and add the following jar files manually to the build path of the project.

- 2 - regular spring jar files
- 8 - regular struts libraries jar files,
- spring-webmvc.jar
- spring-webmvc-struts.jar

### After part ②:

- 1) Move clientcfg.xml file from src folder to /WEB-INF folder.
- 2) Add spring-webmvc-struts.jar file to part ② project

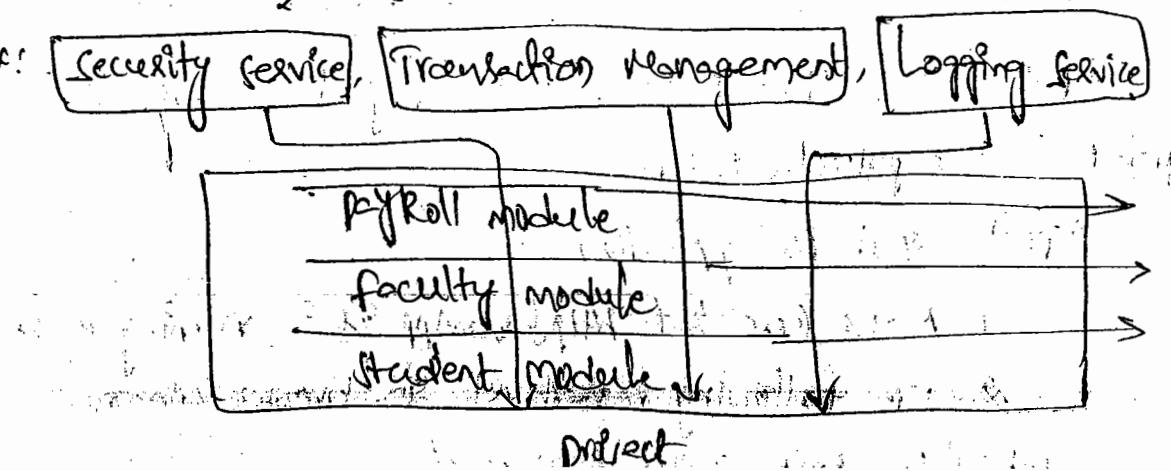
28/11/11

## Spring AOP

Spring AOP is given configure and apply middleware services on Spring application. At Spring level middleware services are called as Aspects.

The additional services that are configurable on the application to make our applications more perfect and accurate are called as middleware services.

Ex: Security service, Transaction Management, Logging service etc..



Middleware services are reusable which can be applied on multiple apps of each module, on multiple modules of each project from outside of the application code without disturbing the application code. Due to this m/w services are called as cross cutting concerns of project.

Spring trop is noway related with OOP. OOP is the methodology to create programming languages, whereas the trop is the methodology to apply m/w services on spring applications.

What is the diff b/w Jee m/w services and spring trop m/w?

Ans:-

### Jee m/w services

1) Allows only to work with server managed m/w services.

2) These m/w services are applicable only on deployable applications.

3) Cannot control the start and end point of m/w service launch.

4) No provision to develop user-defined m/w services outside the server environment.

\* It is not recommended to mix up m/w services code with application/business logic of b-methods as shown in page: 100 & 101.

It is recommended to separate m/w services from logic of b-methods that means extend them and link them with b-methods.

### spring trop m/w services

1) Allows to work with server managed, user defined, third party supplied, spring supplied m/w services.

2) These m/w services applicable on stand alone, deployable app's.

3) can control.

4) Allows us to develop user-defined m/w services.

using xml files or annotations through spring mdp.

(The problem and solution scenario given on page 100 to 102.)

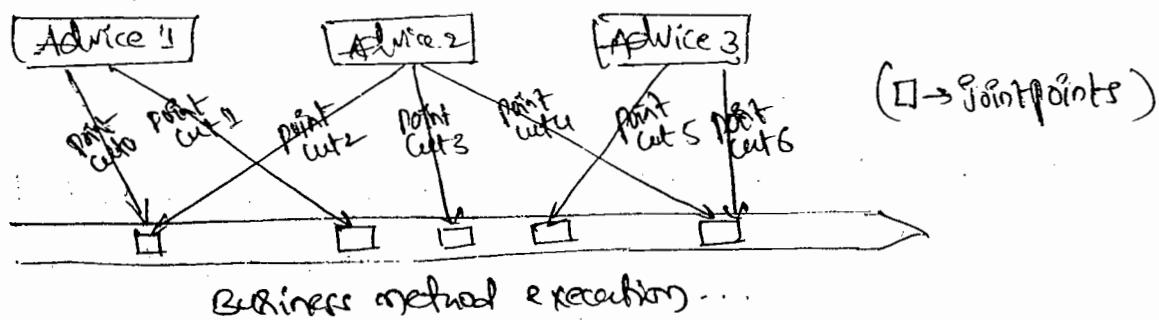
### Spring App Terminology:-

- 1) Aspect 2) Advice 3) Jointpoint 4) pointcut 5) Advisor
- 6) weaving 7) wiring 8) Target Object 9) proxy object

① Aspect: Aspect is the plan to implement my service.

② Advice: Advice is the real implementation of my service.

Ex: Logging service, TX Management are advices that are implemented based on plan kept in their aspects.



③ Jointpoint: The possible positions in business method execution where the advice will be applied is called as Jointpoint.

Ex: Starting of a method, end of a method, when a method returns a value, when a method raises exception etc.

④ pointcut: pointcut is a xml entry in springcfg file that links particular advice with particular Jointpoint position.

⑤ Advisor: Advisor is the combination of Jointpoint and pointcut, pointing to an Advice.

⑥ weaving: The process of configuring aspects / advices on Spring Bean class B methods by using spring app is called weaving.

Note: Spring AOP supports only method level weaving, it doesn't support instance variable (field) level weaving.

④ wiring: configuring spring bean properties with dependent beans through IOC / dependency injection is called as wiring.

⑤ Target object: The spring bean class obj on which we are targeting to configure m/w services is called as Target Object.

⑥ proxy object: The spring bean class obj that is ready with m/w services is called as proxy object.

Note: When b-methods are called on Target object then, only b-logic will executes (Target obj is pre-aop obj), when b-methods are called on proxy object (Post AOP obj) the b-logic executes along with m/w services.  
(for above terminology refer page: 102 to 104)

29/11/11

We can develop 4 types of advice:

1) Before Advice    2) After Advice    3) Throw Advice    4) Around Advice

① executes at the beginning of business method execution.

② executes when business method returns a value

③ executes when b-method throws exception.

④ executes at the beginning of b-method and at the end of b-method execution.

Based on the type of the advice that we develop the join point position for the b-method will be decided automatically.

To know about developing diff types of advice refer page: 104 to 106.

EasyAdvice is a Task class implementing XXX interface of Spring-Step API.

- \* BeforeAdvice type advices are very useful to keep security authentication logic of 5 methods outside the spring bean class.
  - \* AfterAdvice is useful to perform cleanup operations at the end of 5 method execution being from outside the 5 method, like removing userdata from session when the user logout.
  - \* ThrowAdvice is useful to delete the temporary, and incomplete files when exception is raised in the middle of the 5 method execution.
  - \* The logics placed in AroundAdvice execute at the beginning and at the end of 5 method execution.

Spring AOP is borrowing AroundAdvice functionality and API from Alliance Company. Alliance is also another company to give Java libraries supporting AOP.

To develop Around Advice take a Two class Implementing org. `com.alliance.intercept.MethodInterceptor`.

Ex:- public class MyAdvice implements MethodInterceptor

<sup>2</sup> public Object invoke(MethodInvocation mi)

Executes at the beginning of b-method → } logic To begin Transaction management.  
m. procedure; → calls b-method.

End of method → {init(); logic to commit() or rollBack()}

Following the finding of the Court, the FAI based on the return value  
of something

`obj` → represents the current executing environment of the code.

To link developed advices with b-methods of spring bean class we use pointcut Advisor in spring configuration file.

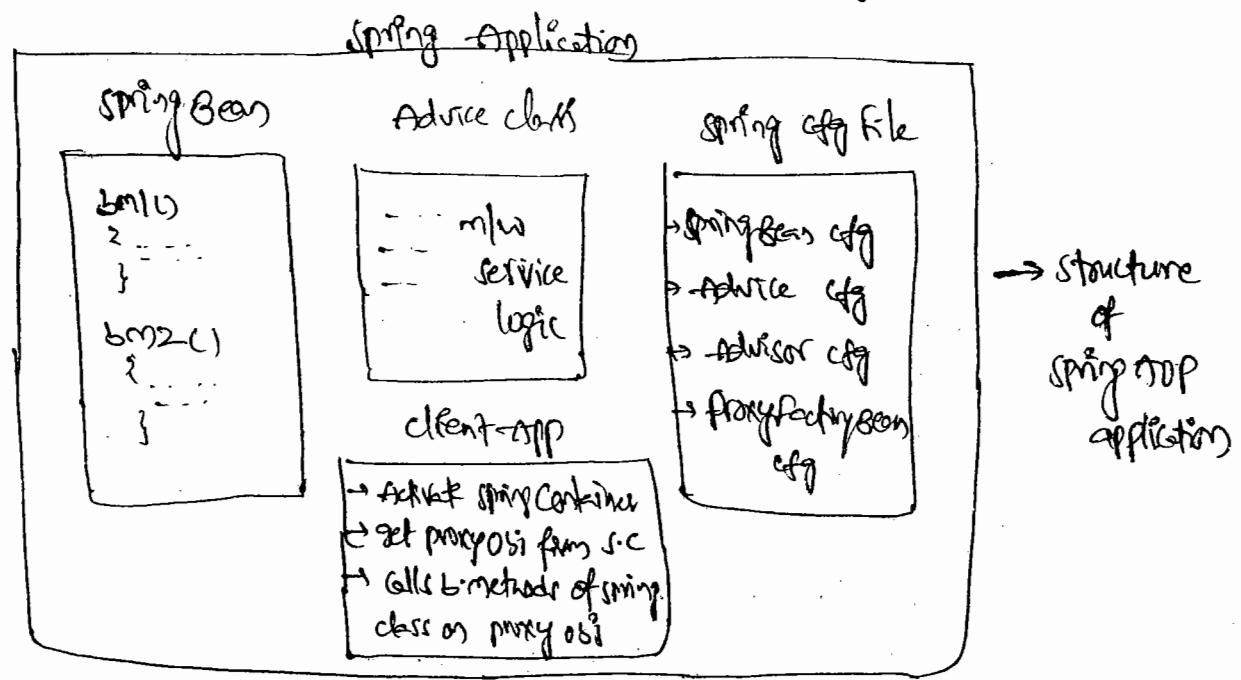
There are two types of pointcut advisors.

① NameMatchMethodPointCutAdvisor → specify b-method name manually to apply the advice.

② Regular Express method PointCut-Advisor → we can use the regular expression characters like 'x', . etc to configure advices with business methods.

for related information in these two advices refer page: 108 to 109

The org. h. aop. framework proxyfactory Bean takes advice to Advisor configuration and returns one proxy object of springbean class by applying advices on the b-methods springBean.



## ~~30/11/16~~ 10/12/2016 Procedure To develop spring AOP based Advises:-

- Step①: develop spring Interface having @method declarations.
- Step②: develop SpringBean class implementing spring Interface.
- Step③: develop Java class as advice having middleware service logic
- Step④: develop spring cfg file
  - Configure SpringBean
  - Configure AdviceBean
  - Configure Advisor Bean linking advice with @methods
  - Configure org.springframework.framework.proxyfactorybeanto generate proxy object based on SpringBean class object.
- Step⑤: Develop client Application.
  - activate spring container
  - Get proxy obj from spring container given by proxyfactorybean
  - Call @method of SpringBean on proxy object.

### Ex-App:

- Demo.java
- DemoBean.java
- ~~spring~~ myAdvice.java (Before-Advice)
- DemoCfg.xml
- ClientApp.java

### Demo.java

```
public interface Demo  
{  
    public void sayHello();  
}
```

### DemoBean.java

```
public class DemoBean implements Demo  
{  
    public void sayHello(String name)  
    {  
        System.out.println("Name! " + name);  
        System.out.println("Hello");  
    }  
}
```

## MyAdvice.java

```

import org.springframework.*;
import java.lang.reflect.*;
import java.util.*;

public class MyAdvice implements MethodBeforeAdvice {
    public void before(Method method, Object[] args, Object target) {
        System.out.println("go before() of MyAdvice class");
        // gathering details of b.method
        System.out.println("B method name is :" + method.getName());
        System.out.println("Name of the Bean class is :" + target.getClass());
        System.out.println("Method getVar() execution started at :" +
                           new Date().toString());
        // changing b.method arg value
        String s = (String) args[0]; // gathering b.method IS
        if (s == null || s.length() <= 3)                         argument value
            args[0] = "mani";
    }
}

```

## DemoCfg.xml

```

<beans>
    <bean id="targetObj" class="DemoBean"/>
    <bean id="adv" class="myAdvice"/>
    <bean id="adviser" class="org.springframework.support.RegexpMethod
                                  advice", ref="adv">
        <property name="advice" ref="adv"/>           PointCutAdvisor
        <property name="pattern" value=".*"/>          → ①
    </bean>   (refers to all b.methods)

```

↑ Config of PointCutAdvisor

```

<bean id="pfb" class="org.springframework.framework.ProxyFactoryBean">
    <property name="proxyInterfaces" value="Demo"/>
    <property name="targetObj" ref="targetObj"/>
    <property name="interceptorNames">
        <list>
            <value>advise
        </list>
    </property>
</beans>

```

(This config gives proxy obj based on target obj by applying advice)

### Client App.java:

```

import org.springframework.context.support.*;
public class ClientApp
{
    public static void main (String [] args)
    {
        FileSystemXmlApplicationContext ctx =
            new FileSystemXmlApplicationContext ("Democfg.xml");
        Demo proxyobj = (Demo) ctx.getBean ("pfb");
        proxyobj .sayHello ("KNReddy");
        proxyobj .sayHello ("KRR");
    }
}

```

classpath: spring.jar, commons-logging.jar.

If client called to method is having invalid argument values  
they can be collected by taking the support of BeforeAdvice.

\* working with "NameMatchMethodPointCutAdvisor", add this  
in place of ①: <property name="mappedName" value="sayHello"/>

\* Diff b/w filesystem/classpath XML ApplicationContext and webApplicationContext spring container?

Ans:- first container looks for spring cfg file in the specified path of filesystem / for the class path, and this container can be activated within the server and also outside the server.

The second one ~~looks~~ should be activated only from web application and looks for spring cfg file in web-INF folder of web app.

Note: Always develop your Java class by having high cohesion (Perfect structure) and low coupling (less dependency with other classes).

01/12/11 : for AroundAdvice based spring app application refer page: n 118, app @① to ⑩

\* If two advices configured on the defines method are capable of executing at same position, then they will be executed in the order they are configured.

The confirmation statements generated by application to understand flow and status of execution are called as Log messages.

In real world projects no prefer to write log messages using `System.out.println()`. It is recommended to use Log4J tool to generate these log messages.

for ex: app that performs logging on something from outside the method by using all the four types of Advices refer app ① of page 118 to 121

In Log4J: 3 imp. class will be there.

① `logger.info` → enables logging on the class, generates five types of logs having priorities: (debug < info < warn < error < fatal)

→ allows to set logger level while retrieving log msg.

Note: If logger level to refine log msgs. Is taken as 'WARN' Then application gives only those log msgs whose logger level  $\geq$  WARN.

- ① Appender obj  $\rightarrow$  To decide destination place of writing log msgs.
- ② Layout obj  $\rightarrow$  To decide the pattern and format of each log msg.

## Q2/10/11 Transaction Management:-

Combining set of operations into single unit and executing them by applying do everything or nothing principle is called as Transaction Management.

Combining withdraw amount, deposit amount operations of transfer money tasks into a single unit and executing them with do everything or nothing principle comes under TX management.

Sample code for TX management:-

```
public boolean transferMoney (int srcAccno, int destAccno, float amt)
```

```
? try {
```

```
    BeginTransaction (tx)
```

```
    // withdraw amt from source account
```

```
    - - - } operation 1
```

```
    // deposit amount to destination account
```

```
    - - - } operation 2
```

```
    Commit Transaction (tx)
```

```
} return true;
```

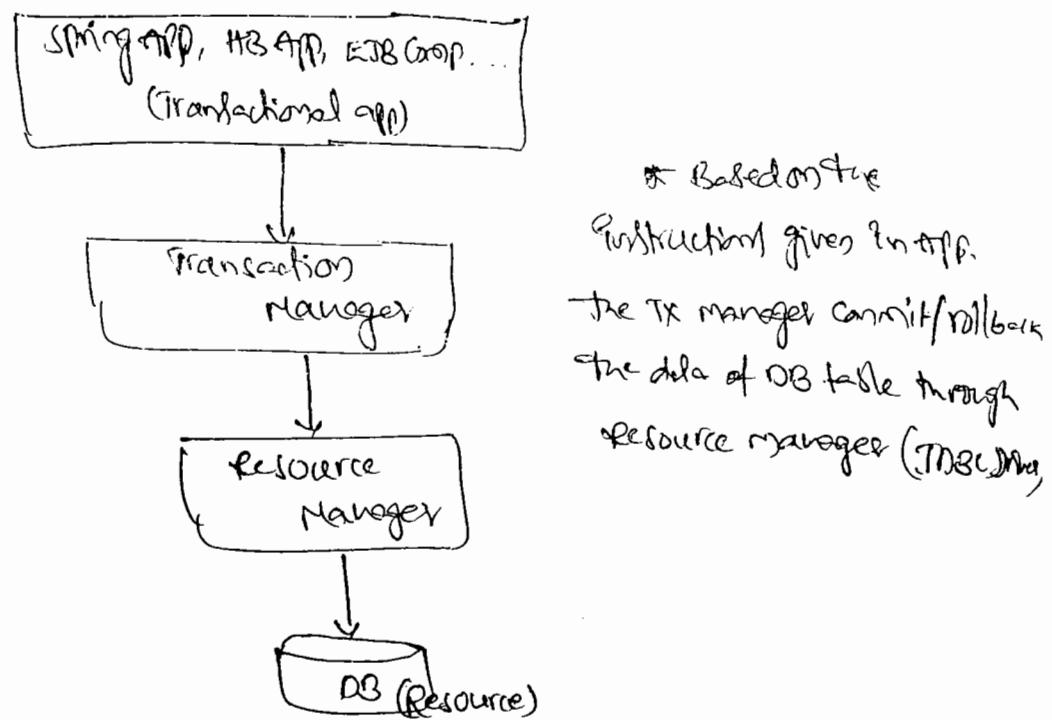
```
catch (Exception e)
```

```
return false;
```

We can bring the effect of Tx management through manually added conditions, return stmts, try catch blocks. This may spoil our code and <sup>may</sup> make the code as unmanageable code.

To overcome this problem we Tx M/H service given by server or underlying db which runs from outside the code and applies Do everything or nothing principle on the code.

Architecture of Tx management:-



The application on which tx service is enabled is called as Transactional Application.

TX management gives support for ACID properties implementation on db.

- Combining set of related operations (Individual) into single unit → Atomicity
- Executing logic of the b-methods without violating the rules kept in application data is called as developing consistent business logic.
- \* Even though application data rules are violated in the middle of the b-method statements if there is a guarantee for correction of that all data at the end of Tx/b-method (committed/rollback) is called as consistency.

→ When multiple apps/users/threads manipulate same data of object or db table simultaneously or concurrently then there is a chance of getting data corruption. To overcome that use isolation and allow one thread/application/user at a time to manipulate object data or db data through locks.

Isolation process prevents concurrent and simultaneous access of application data.

→ Getting the ability of reconstructing application data or db data through log files or backup files even though app or db is crashed comes under maintaining data having durability.

Based on the no. of resources that are involved in tx mgmt there are two types of tx mgmt.

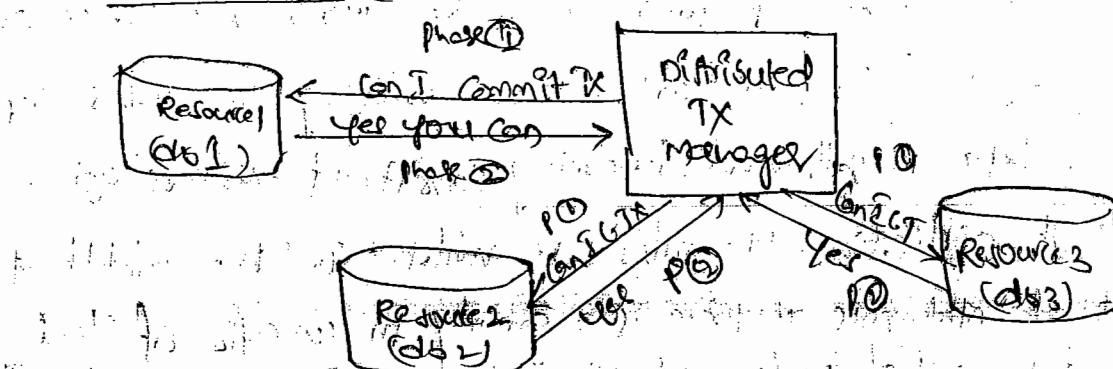
1) Local Tx management → uses single resource (db) for all the operations of tx management.

Ex: Transfer money operation on two site of same bank.

2) Distribute Tx management → uses multiple resources (db's) for all the operations of tx management.

Ex: Transfer money operation on two site of two diff banks.

The distributed tx manager runs distributed tx's based on "2PC protocol" (2 Phase Commit).



Spring, EJB support local, distributed Tx's, whereas JBoss supports only local Tx management.

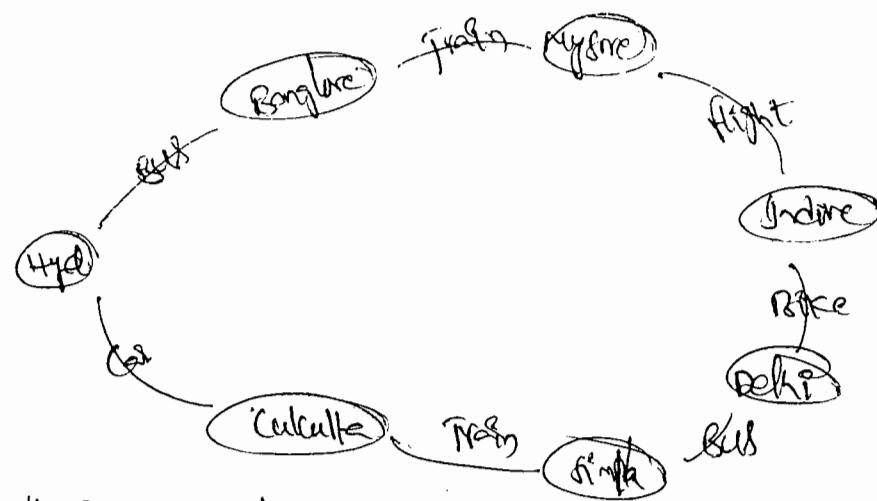
In Spring apps we can use Spring's own supplied Tx service or app server supplied Tx service for both local and distributed Tx's.

Spring's own managed Tx service is good for local Tx management in Spring applications.

App server's managed Tx service is good for Distributed Tx mgmt.

### TX management Models:-

1. flat Tx model
2. Nested Tx model



When the above India tour plan is given to a JBoss application that runs with a flat Tx model, if one or other Journey tickets are not confirmed, it will cancel the remaining all other Journey tickets (as in flat Tx model all Journey ticket bookings will be taken as direct operations of main Tx., as shown below). So the failure of one operation rollbacks whole Tx.

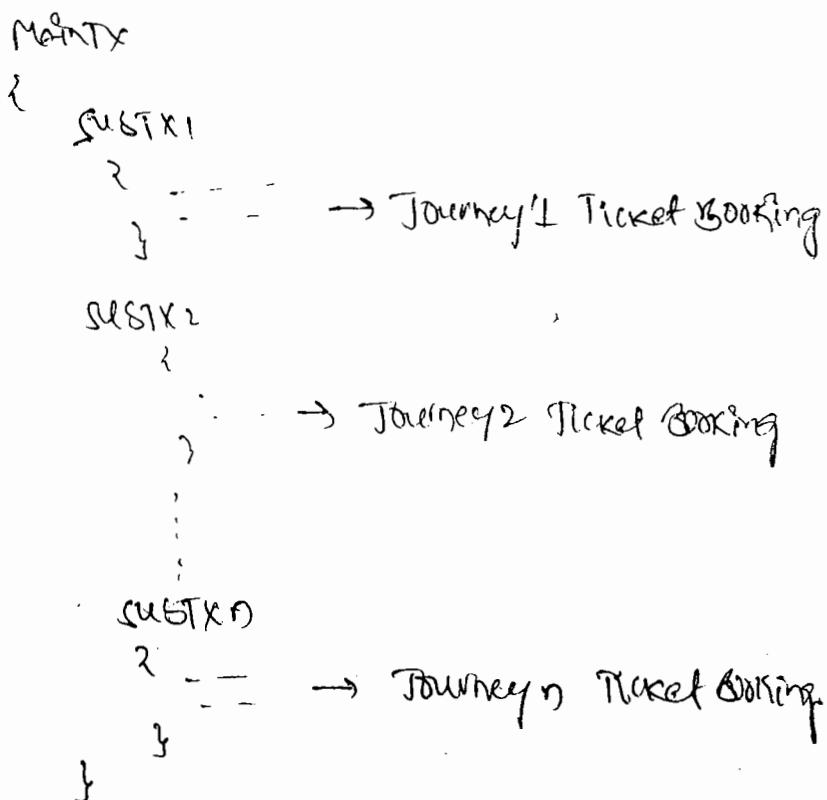
MainTx

1 Journey 1 Ticket Booking (Operation 1)

Journey 2 Ticket Booking (Operation 2)

Journey 3 Ticket Booking (Operation 3)

When above tour plan is given to Nested TX model based sub application, even though one or other Journey tickets are not confirmed it will confirm the remaining Journey tickets. Because in nested TX model each Journey ticket booking operation will be taken as the sub TX of main TX. So the success or failure of one sub TX doesn't effect others sub TX.



→ EJB, HB supports only flat TX. Spring framework both flattened.

→ for 03/12/11 to 07/12/11

pages 25 to 52

08/12/11

Spring 2.5 supplies declarative TX management on methods of spring bean class. While working with this annotation we can also specify TX attributes.

Ex: TestIntf.java:

```
public interface TestIntf  
{  
    void sm1() throws RuntimeException;  
}
```

TestBean.java:

```
import org.springframework.annotation.*;  
import org.springframework.jdbc.core.*;  
public class TestBean implements TestIntf  
{  
    JdbcTemplate jt;  
    public void setJt(JdbcTemplate jt)  
    {  
        this.jt = jt;  
    }
```

@Transactional(propagation = REQUIRED)

public void sm1() ↓ TX attribute

```
{  
    int res1 = jt.update("update emp set sal = 7777 where  
                           enpro = 7834");
```

```
    int res2 = jt.update("update emp set fName='Satya' where  
                           dghno = '10'");
```

if( res1 == 0 || res2 == 0)

```
{  
    System.out.println("TX rolled back");
```

```
} throw new RuntimeException;
```

else

```
{  
    System.out.println("TX Committed");  
}
```

## Spring Config.xml

```
<beans> . . . <spring-beans.xsd, spring-tx.xsd, spring-dao.xsd>

<bean id="dmds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="..."/>
</bean>

<bean id="txmgr" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dmds"/>
</bean>

<bean id="template" class="org.springframework.jdbc.core.JdbcTemplate">
    <constructor-arg ref="dmds"/>
</bean>

<bean id="ts" class="TestBean">
    <property name="jt" ref="template"/>
</bean>

<tx:annotation-driven transaction-manager="txmgr"/>
</beans>
```

↓  
(Applies tx service on h-methods based on  
@Transactional annotation.)

## client.java:

1. develop as a stand alone application to activate spring container and to get Spring bean class obj (TestBean) and all its method.

Jar files in classpath: spring.jar , commons-logging.jar



OB/12/11

1 Title: Spring 3.x based App for annotations based Dependency Injection  
2  
3 -----TestInter.java-----  
4 package p1;  
5  
6 public interface TestInter  
7 {  
8 public String sayHello();  
9 }  
10 -----TestBean.java-----  
11 package p1;  
12 import java.util.Date;  
13 import java.util.List;  
14 import javax.annotation.Resource;  
15 import javax.annotation.Resources;  
16 import org.springframework.beans.factory.annotation.Value;  
17 import org.springframework.stereotype.Component;  
18 import org.springframework.stereotype.Service;  
19  
20 @Component("tb") → Recommended to use @ Service annotation.  
21 public class TestBean implements TestInter  
22 {  
23 //Bean Property  
24 UserBean ul;  
25  
26 @Value("10")  
27 int no;  
28  
29 Date d1;  
30  
31 @Value("#{T(java.util.Arrays).asList('India','Bharat','Hindustan')})"  
32 String nicknames[];  
33  
34 @Value("#{T(java.util.Arrays).asList('Red','Blue','Green')})"  
35 List colors;  
36  
37 @Resource(name="dfb") → It is like ref tag/attribute  
38 public void setD1(Date d)  
39 {  
40 d1=d;  
41 }  
42  
43 @Resource(name="ub") → Bean id of UserBean  
44 public void setU1(UserBean u1)  
45 {  
46 this.ul=u1;  
47 }  
48  
49 public String sayHello()  
50 {  
51 System.out.println("no"+no);  
52 System.out.println("d1"+d1.toString());  
53 System.out.println("nicknames");  
54  
55 for(int i=0;i<nicknames.length;i++)  
56 System.out.println(nicknames[i]);  
57  
58 System.out.println("Colors="+colors.toString());  
59 System.out.println("roles="+roles.get("ravi"));  
60  
61 return "Good Morning";  
62 }

*Object of TestBean*

*Object of UserBean*

*Object of Date*

*Object of List*

*Object of String*

*Object of Properties*

Spring 3.x is given to provide full fledged support for annotations and AnnotationConfigApplicationContext container is introduced to work with annotations based dependency injection.

To know more features of Spring 3.x refer `spring-3.x-home/changelog.txt` file.

for new features of Spring 3.x refer part 2 of pdf file

for high level architecture diagram of Spring 3.x refer `log-3.pdf` file.

Spring 3.x contains the following modules.

1. Core Container
2. Data Access/ Integration module
3. WEB module
4. AOP and Instrumentation module.
5. Test module (for unit Testing)

In Spring 3.x the Java class becomes `SpringBean` when they are annotated with `@Service` or `@Component`.

To inject values to reference type bean properties we need to use `factoryBean` support, especially if those Java classes are third party API or Java API supplied Java classes.

for ex: app on Spring 3.x based application that uses annotations for dependency injection refer handout given on 08/12/11.

for main SpringBean classes use `@Service` annotation.

for sub Spring Bean classes whose objects will be injected to the bean properties of main SpringBean classes use `@Component` annotation.

07/12/11

## Spring WEB MVC

Spring web mvc is part of spring web module which allows to develop mvc architecture based web app's alternative to struts. Isp kind of web flow sl/w.

In web flow slw the market leader is struts, the spring web mvc is gives alternative to struts.

Spring is popular to develop model layer to develop logic and persistence logic. Spring web mvc is not that much popular to develop view and controller logic of web app.

feature of spring web mvc:-

- \* Allows to develop mvc2 arch-based web app.
- \* Activate spring container during server start up or deployment of web application through DispatcherServlet.
- \* Gives more support for dependency injection.
- \* Allows to develop resources of POJO and POJ2.
- \* Allows to develop presentation logic of view layer using multiple technologies like JSP, freemarker, velocity... etc.
- \* Gives JSP Tag libraries to simplify the programming.
- \* Allows to use other spring module's features.

### struts

Controller - ActionServlet  
formBean  
Action class

### spring

DispatcherServlet  
Command class  
CommandController class

struts cfg file

Action Mapping

Actionforward cfg

spring cfg file

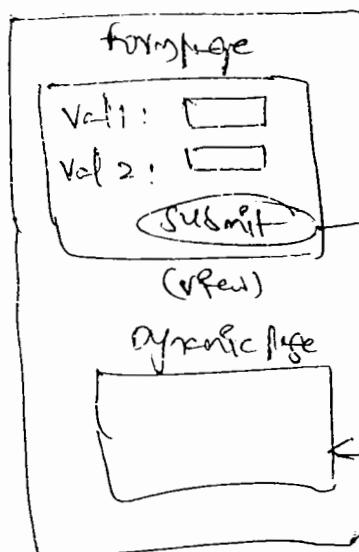
URL Handler

View Resolvers

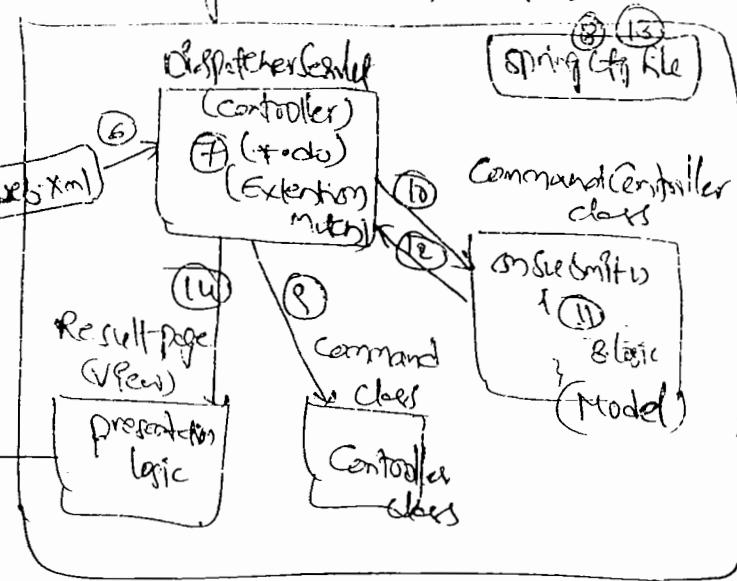
flow of execution of spring web mvc :-

①②③

Browser window



Spring web mvc application



If b-logic is directly based in CommandController class

then it is called as Model layer resource.

If CommandController class contain logic to communicate with other model layer components like EJB, spring jee... Then CommandController class comes under Controller layer resource.

With respect to the diagram,

- ① Program deploys spring web app in web app server.
- ② Based on load-on-startup, servlet container, front-side Dispatcher class of either during server startup or during deployment of app.
- ③ Dispatcher servlet activates core app context (spring container) and container performs preInstantiation of components of spring cfg file, whose name is DispatcherServlet logicalName.xml.

- ⑩ End user launches form page on browser window.
- ⑪ End user submits request from form page.
- ⑫, ⑬ : Based on Configuration done in web.xml, DispatcherServlet traps and takes care of req and also reads form data.
- ⑭ Dispatcher uses UI Handler configuration to decide Command class and Command Controller class that are required to process req.
- ⑮ DispatcherServlet writes form Data to Command class obj.
- ⑯ D.s calls onsubmit() on CommandController class.
- ⑰ B logic of onsubmit() process and generates the result.
- ⑱ onsubmit() returns Model & View class obj to D.s.
- ⑲ D.s use ViewResolver config to decide Result page and Command Controller layer.
- ⑳ D.s forwards control to the result page.
- ㉑ Result page uses presentation logic and finds the logic, sends formatted result to browser window (Dynamic web page)

### struts

- 1) Less support for dependency injection.
- 2) Given by Apache foundation
- 3) Given Validator plugin to perform form validations
- 4) Built-in support for AJAX
- 5) Allows to use EL (by oracle Struts2.x) in JSP programs.
- 6) form components name and formBean class property name must same.

### Spring web mvc

- 1) More support
- 2) By Interface 2.1
- 3) form validation should be performed explicitly.
- 4) No built-in support for AJAX
- 5) Given only EL to use in JSP program.
- 6) No need to match

If D.R logic name is 'abc' the spring cfg file name  
should be "abc-servlet.xml". And this contains,

1. UI Handler Bean cfg.
  2. Command class cfg
  3. CommandController class cfg.
  4. View Renderer cfg
- etc...

### ① UIHandler cfg:-

It helps DispatcherServlet to link client generated req  
to CommandController class. There are two types

i) SimpleUI Mapping → Allows to map the req trapped by  
D.R to CommandController class by using bean id or bean name  
of CommandController class. It is recommended to use.

ii) org.springframework.web.handler.BeanNameUIHandlerMapping →  
Allows to map the req trapped by D.R to CommandController class  
by using only bean name of CommandController class

### ② Command class cfg:-

No separate config required for this, bcz it will be  
configured along with CommandController class config.

### ③ Command Controller class cfg:-

This is a Java class which must extend from  
xxxController class and must be configured in Spring cfg file  
with various details.

## Some Important CommandController classes:

- ① AbstractFormController
  - ② SimpleFormController
  - ③ ConcreteFormController
  - ④ ValidatorAwareView Controller
  - ⑤ BaseCommandController
  - ⑥ org.springframework.web.servlet.mvc.multiaction.MultiActionController  
(to handle multiple submit button based)
- To handle single submit button based form page generated request  
Available in org.springframework.web.servlet.mvc pkg.

All predefined xxxController classes of spring API

implements "org.springframework.web.servlet.mvc.Controller" directly or indirectly.

The SimpleFormController and AbstractFormController based CommandController classes launch form page in the browser window when they get req from browser window having Http Request method "GET". But they process the req given by browser window when the Http Request method is "POST".

while Configuring SimpleFormController type classes we can specify its form page by using "formView" property and specify the result page by using "successView" property.

### ④ ViewResolver cfg:-

ViewResolver helps the D.S. to decide the technology that is required to view layers to develop presentation logic.

Every ViewResolver is a Java class and this name will be changed based on the Technology of View layer

Technology View Resolver doesn't work

JSP → org. sf. <sup>web</sup> servlet. Internal Resource View Resolver.

Velocity → org. ft. web scrlf. Rev. velocity. Velocity View Residue

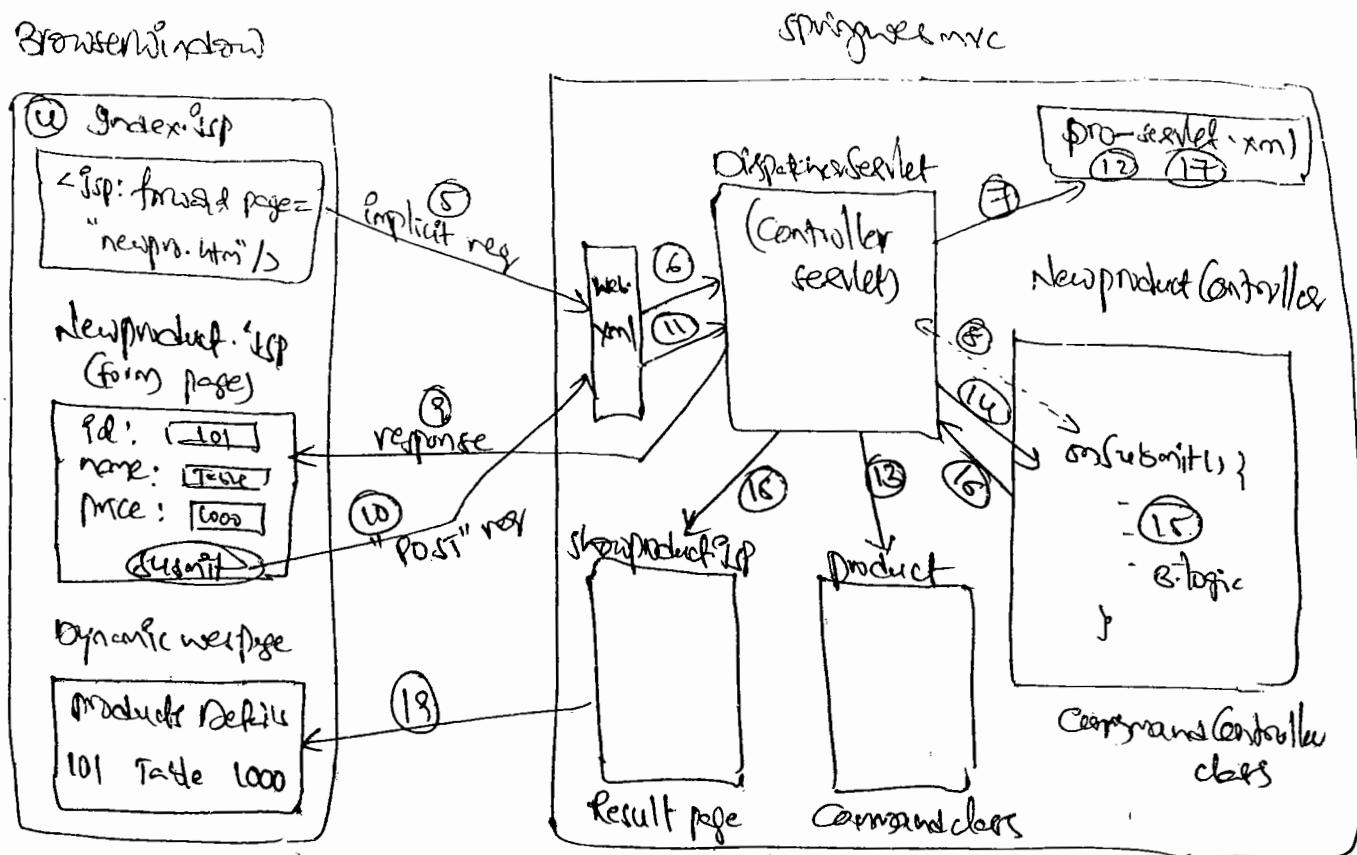
freemarker → org. lsf. web seaflet. view. freemarker. freemarkerViewResolver

Xlist → org. ff-web-seamlet View. Xlist. XlistViewResolver

JasperReports → org. st. webseitl. view. Jasper. Jasper Report View Resolver

Ex: App on Simple form Controller based c.c class :-

for ex. off on this refer off 2u of bookstaf



To make SimpleFormController type classes generating form page, we make the welcome page of controller class generating Implicit request, having the HTTP Request method "GET" as shown in diagram from step ④ to ①.

W.R.T. Diagram:

① ② ③ : Are same as previous Spring MVC Arch.

④ ⑤ : The Prifiled req given to the web app, makes welcome page to generate Implicit request to web app.

⑥ : Based on cfg alone in web.xml or takes the req.

⑦ : D.S uses URLHandlerMapping of Spring config file to decide the CommandController class to process the req.

⑧ : Since req method is "GET" and CommandController class type is SimpleFormController

⑨ : It makes D.S to launch the form page on Command Controller class on Browser window.

⑩ : End user submits form page having req. method of "POST".

⑪ Same as ⑥ of previous.

⑫ Same as ⑦ but CommandController becomes ready to process the request, since the req method is "POST".

⑬ to ⑯ : Same as ⑨ to ⑮ of previous diagram (Spring MVC arch).

\* Spring is having its own TLD tag library whose file is "spring.tld" available in `springHome/lib/resources` folder.

11/27/0

## Flow of execution of app② of booker

Ⓐ Ⓑ Ⓒ → same as spring architecture diagram given in class.

Ⓓ → end user gives request to the deployed product MVC application

→ the default welcome page index.jsp will be executed automatically.

Ⓔ index.jsp generates implicit request to web application having default http req method "GET". (ref 2658)

This request related request url contains newprod.htm word

Ⓕ Based on the configuration done in web.xml file, the DispatcherServlet trap and takes the request (url pattern is \*.htm and implicit request url contains "newprod.htm") (ref: 2702 → 2705, 2655 → 2705)

Ⓖ DispatcherServlet uses simpleFormHandler mapping id to link implicit request with NewProductController class. (ref 2702)

<prop key = "newprod.htm" > Controller</prop>

H Since the CommandController class type is SimpleFormController,

I the received request method is GET, the DispatcherServlet renders form page on the browser window. (ref 2725, 2729). This time internally the ViewResolver will be utilized

J DispatcherServlet launches newproduct.jsp on browser window as form page. (2659 → 2687)

K End user fills up the form page and submits the request. (2684)

L same as F

M same as G

N Since the CommandController class type is SimpleFormController and request method of form page is "POST", the NewProductController class becomes ready to process the request.

- (N) DispatcherServlet writes the received form data of form page to Command class obj by calling .setXXX() on that object.
- (O) DispatcherServlet calls onSubmit(-,-,-,-) on NewProductController class obj to process the request. (Q761, Q772).
- (P) onSubmit() returns ModelAndView class obj to DispatcherServlet having logical name "showproduct".
- (Q) DispatcherServlet uses "successView" property <sup>value</sup> of NewProductController class to decide result page of . During this operation the ViewResolver configuration support will be taken.
- (R) DispatcherServlet forwards the control to result page.
- (S) showproduct.jsp sends response to browser window.

12/12/11

form validation logic in Spring Web MVC applications is possible only at server side by programmatic approach.

for this we need to take a Java class Implementing org.springframework.validation.Validator interface and place form validation logic in validate() definition. while writing this logic we can take the support of rejectXXX()'s of ValidationUtils class.

The form validation logic written for this Java class not only can be used in springweb mvc applications, it also can be used outside the spring webmvc applications.

for example app or validations based <sup>mvc</sup> springweb app refer app @25.

procedure to add programmatic server side form validations in app :-  
 step① create a Java class Implementing org.springframework.validation.Validator interface having the Java code based form validation logic.

refer ProductValidator.java of page 17

② Configure this Validator class as Bean property value of Command controller class.

refer Validator property config of NewProductController class.

③ Write following logic in form page to display the generated form validation errors

④ The remaining resources of the application are same as app 24

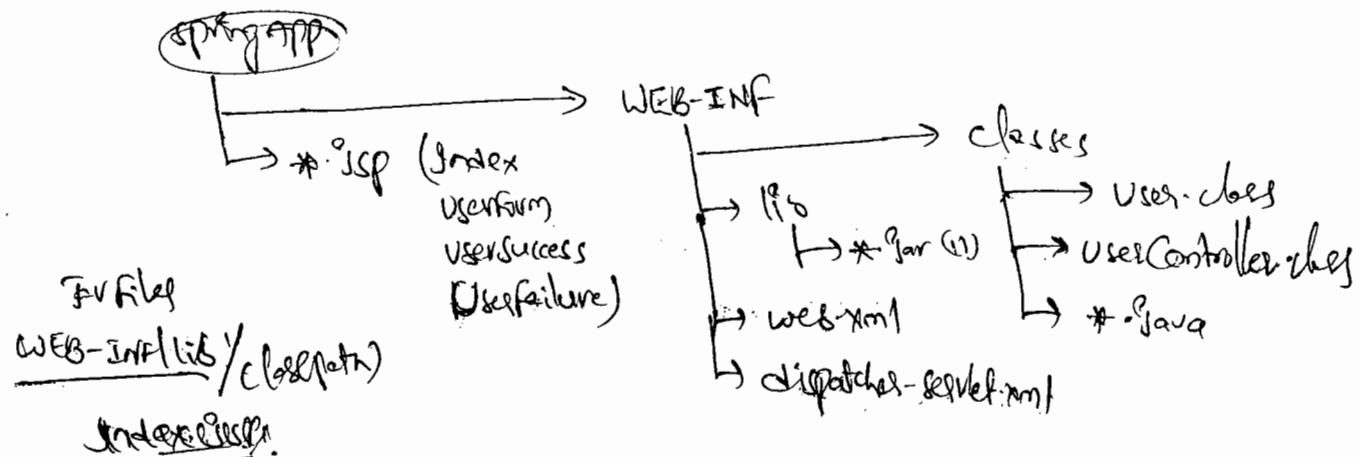
\* Spring supplies more number of annotations to develop spring web MVC applications.

@Controller annotation marks our form class as CommandController class

@RequestMapping annotation links web request with our Command controller class.

@ModelAttribute annotation gathers data & kept in model map class object.

Deployment directory structure for annotations based spring web mvc (Spring 3.x) :-



### Index.jsp

1. antlr-runtime.jar
  2. common-logging.jar
  3. org.springframework.jar
  4. " beans.jar (cp) 10) " web-servlet.jar => \* (cp)
  5. " context.jar (cp)
  6. " context-support.jar (cp)
- 7) " core-3.x.jar (cp)
  - 8) " expression-3.x.jar
  - 9) " web-3.x.jar => \* (cp)

## Index.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
```

### Web.xml :

Configure dispatcher-servlet class having DispatcherServlet as logical name and \*.htm as url-pattern.

### dispatcher-servlet.xml :

```
<beans>
    <!-- Spring-Context.xml -->
    <Context:component-scan base-package="pl" />
</beans>
```

## Userform.jsp

```
<form method="post" action="pro.htm">
    <input type="text" name="name" />
    <input type="password" name="password" />
    <input type="submit" />
</form>
```

## User.java

```
package pl;
public class User {
    private String name, password;
    // If getXXX(), setXXX() methods
```

## UserController.java

Note RequestMapping annotation can be used to link specific http method based request with specific java method of CommandController class.

UserSuccess.jsp : valid Credentials  
UserFailure.jsp : Invalid Credentials

## UserController.java

package pl;

import org.springframework.stereotype.Controller;

import org.springframework.ui.ModelMap;

import org.springframework.web.bind.annotation.\*;

@Controller

@RequestMapping("pr.htm")  
(method = RequestMethod.GET)

public class UserController

@RequestMapping(method = RequestMethod.GET)

public String showUserForm(ModelMap model)

{ System.out.println("showUserForm()");

User u1 = new User();

model.addAttribute(u1);

} return "userform.jsp";

@RequestMapping(method = RequestMethod.POST)

public String mySubmit(@ModelAttribute("user") User user)

// Read form data from Command class obj.

String uname = user.getName();

String pass = user.getPassword();

If (uname.equals("satyajit") & pass.equals("tech"))

return "userSuccess.jsp";

else

return "userFailure.jsp";

represents the  
command class obj  
kept in modelmap

13/12/14

In struts, we can use "LookupDispatchAction" class to handle multiple submit button generated request by using multiple user defined methods of action class.

In spring web mvc we can use MultiActionController to perform the same operation.

The MultiActionController can have multiple user defined methods to process the req generated by multiple submit buttons of form page, But the form page must send methodName as additional request parameter value.

To configure this additional req parameter name use parameterMethodNameResolver Bean.

→ for ex: off on this refer ~~app~~ Handout of 13/12/11.

The Java class that contains persistence logic and separate this logic from other logic is called DAO class.

We can link the form page generated req with Command controller class without having url and but it is not recommended to process like when multiple form pages and commandcontroller classes are there, those generates those procedure gives problems.

In spring web mvc app the form components are directly developed and not bound with Command class properties using <spring:bind> tag. Then form components names should be taken as Command class properties.

1/12/11

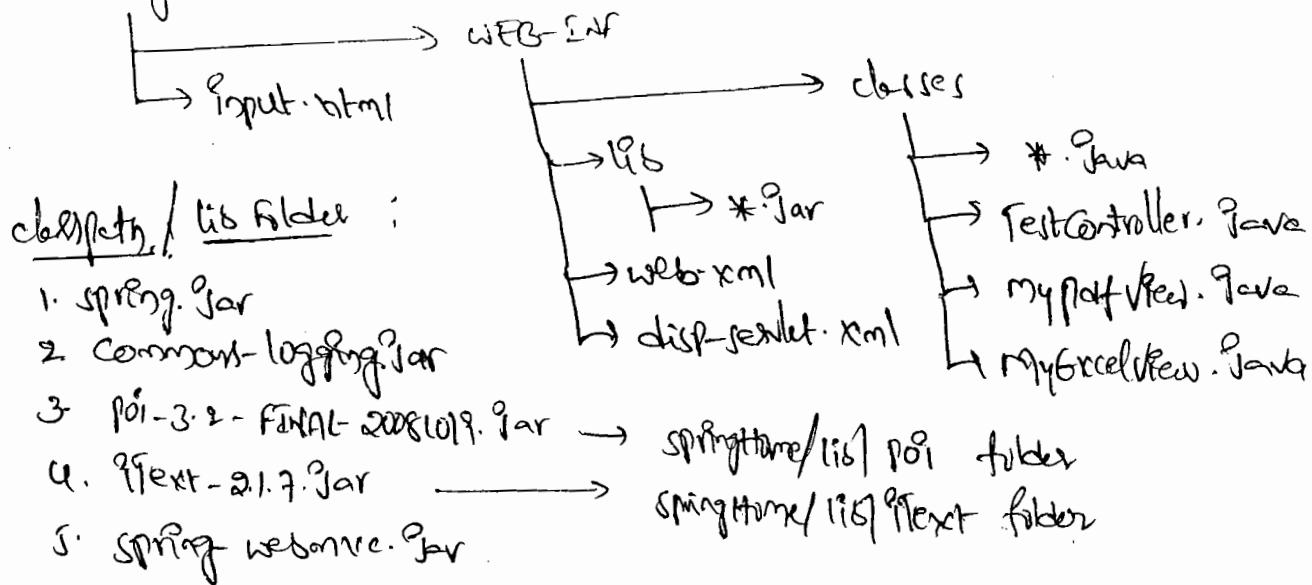
Spring API supplies AbstractxxxView classes to generate excel document based view and spring web mvc applications. These classes are like org.springframework.web.servlet.view.document.AbstractExcelView, AbstractJExcelView, AbstractPdfView and etc..

AbstractExcelView internally uses poi mechanism to generate excel doc based views.

AbstractpdfView internally uses IText mechanism to generate pdf doc based views.

Ex: off on making spring web mvc application generating pdf, excel document based results.

springpdfapp:



Input.html:

```
<form action="spring.do">
```

```
  <input type="submit"/>
```

```
</form>
```

web.xml:

Configure DispatcherServlet having logical name "disp".

url-pattern "\*.do" and also enable load-on-startup.

## Step - Test Controller, Java

```
import javax.servlet.http.*;
import org.springframework.web.context.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
import java.util.*;

public class TestController implements Controller {
    public ModelAndView handleRequest(HttpServletRequest req,
                                      HttpServletResponse res)
        throws Exception {
        List l = new ArrayList();
        l.add("Sandeep");
        l.add("Mayara");
        l.add("Keddy");
        return new ModelAndView("viewbean", "result", l);
    }
}
```

Bean id of spring beans that  
contain view generation  
logic

Note: org.springframework.web.servlet.ViewNameViewResolver makes spring webmvc application to use spring bean class as View layer resource. It is like InternalResourceViewResolver which uses JSP's in the View layer.

### dispatcher-servlet.xml:

```
<beans>
    <bean id="url" class="org.springframework.web.handler.SimpleUrlMappingHandler">
        <property name="mappings">
            <props>
                <prop key="spring-db">controller</props>
            </property>
        </beans>
        <bean id="controller" class="TestController"/>
        <bean id="view" class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
        <bean id="..." />
    </beans>
```

### MyExcelView.java :

```
import java.util.*;  
import javax.servlet.http.*;  
import org.springframework.web.servlet.view.document.*;  
import org.apache.poi.hssf.usermodel.*;  
  
public class myexcelview extends AbstractExcelView  
{  
    protected void buildExcelDocument(Map m, HSSFWorkbook wb,  
        HttpServletRequest req, HttpServletResponse res) throws Exception  
    {  
        HSSFSheet sheet = wb.createSheet("Names");  
        ArrayList al = (ArrayList) m.get("result");  
        getCell(sheet, 0, 0).setCellValue(new HSSFRichTextString("User Name"));  
        getCell(sheet, 1, 0).setCellValue(new HSSFRichTextString(l.get(0) + " "));  
        getCell(sheet, 2, 0).setCellValue(new HSSFRichTextString(l.get(1) + " "));  
        getCell(sheet, 3, 0).setCellValue(new HSSFRichTextString(l.get(2) + " "));  
    }  
}
```

### MyPdfView.java :

```
import com.lowagie.text.*;  
import com.lowagie.text.pdf.PdfWriter;  
  
public class mypdfview extends AbstractPdfView  
{  
    protected void buildPdfDocument(Map m, Document doc, PdfWriter pw,  
        HttpServletRequest req, HttpServletResponse res) throws  
    {  
        ArrayList al = (ArrayList) m.get("result");  
        Paragraph p = new Paragraph("User Details");  
        p.setAlignment("center");  
        doc.add(p);  
        Table t = new Table(1);  
        t.addCell(al.get(0) + "");  
        doc.add(t);  
    }  
}
```

## JMS

In client-server communication, if client is obliged to generate next request until given request related response comes from server then such communication comes under synchronous communication.

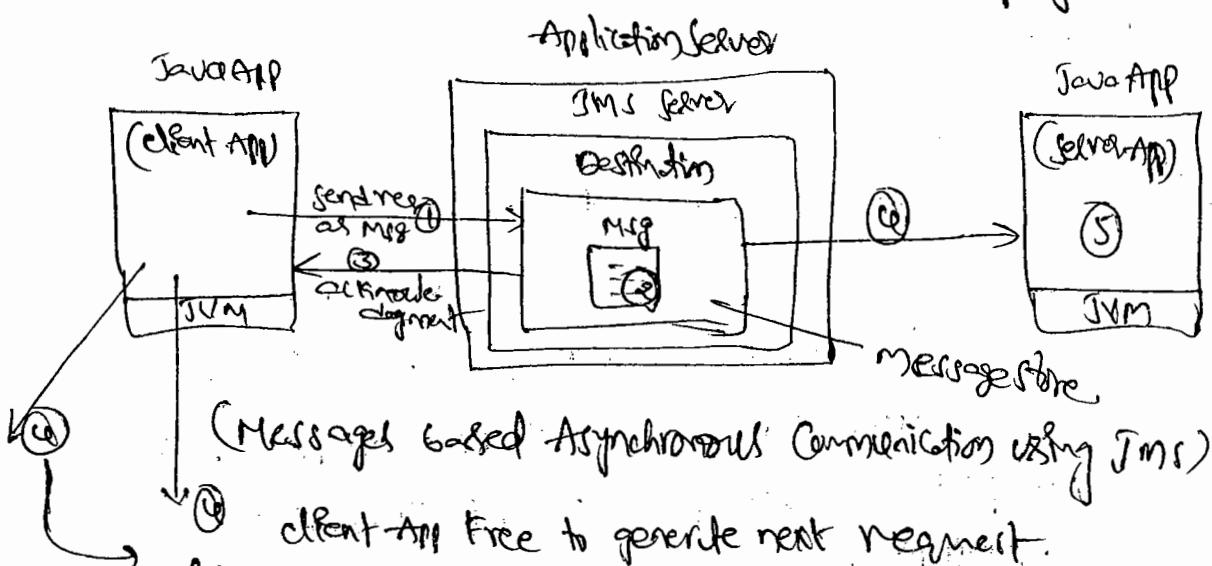
In client server communication if client is free to generate next request or to perform client side operations without waiting for given request related response from the server then such is called as Asynchronous Communication.

Generally request-response model communication is synchronous and message based communication is Asynchronous Communication.

15/12/11

To achieve Asynchronous Communication in web environment use AJAX or AJAX Tool Kits or PORTLETS.

To achieve message based Asynchronous Communication between two Java applications use JMS (Java messaging service).



⑥ client app free to generate next request.  
client is free to perform client side operations

In the diagram client P1 sending request as message and that message is stored in the destination, client P2 getting acknowledgement from destination.

After receiving acknowledgement client P1 free to generate next request asynchronously.

When server is free it gathers new msg from destination and ~~process~~ process the request. Server app sends result to client app as msg <sup>through</sup> destination.

In synchronous communication client and server application are tightly coupled. In asynchronous communication the client and server applications are loosely coupled.

JMS is a sun microsystems supplied specification having set of rules and guidelines in the form of JMS API. Vendor companies use JMS API to develop JMS servers and destinations. Programmers use JMS API to develop client and server applications having the ability to create messages, send and receive messages.

Vendor company supplied JMS server, destination etc together is called as JMS provider. IBM fulfilled JMS provider its name is MQ series whereas Microsoft supplied its name is MS MQ. But every application has one JMS provider.

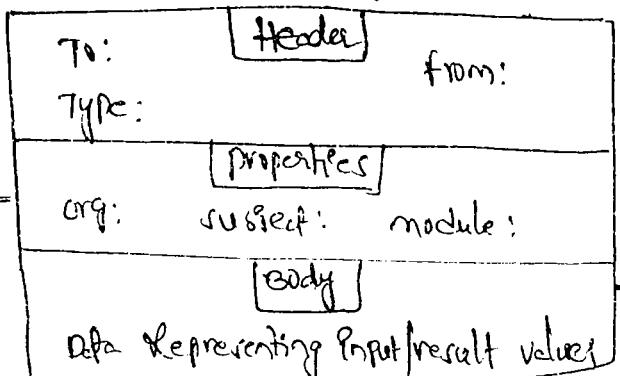
There are two messaging domains,

- 1) PTP (Point To Point Messaging domain)
- 2) Pub/Sub (Publisher and Subscriber Messaging domain)

(Refer page no. 1, 2 of Handout to know more about the above discussed topics).

## JMS Message

Properties are manually assigned values by program as additional info.



→ Header names will be fixed and values will be generated automatically

→ Body represents real data of JMS message

We can develop 5 types of messages in JMS.

They are → Text msg, Map msg, Byte msg, Stream msg, Object msg.

(For related info refer page 34 of Handout)

→ Every JMS provider of App server allows us to create JMS connection factories and destinations / message stores and they will be identified by outsiders through their JNDI names.

Procedure To Create Topic Connection factory, Queue Con. factory, Topic definition, Queue destination in domain 1 server of Glassfish ex:-

Step ①: Start domain 1 server of Glassfish and open its Administration console.

Step ②: Create JMS Topic Con. factory

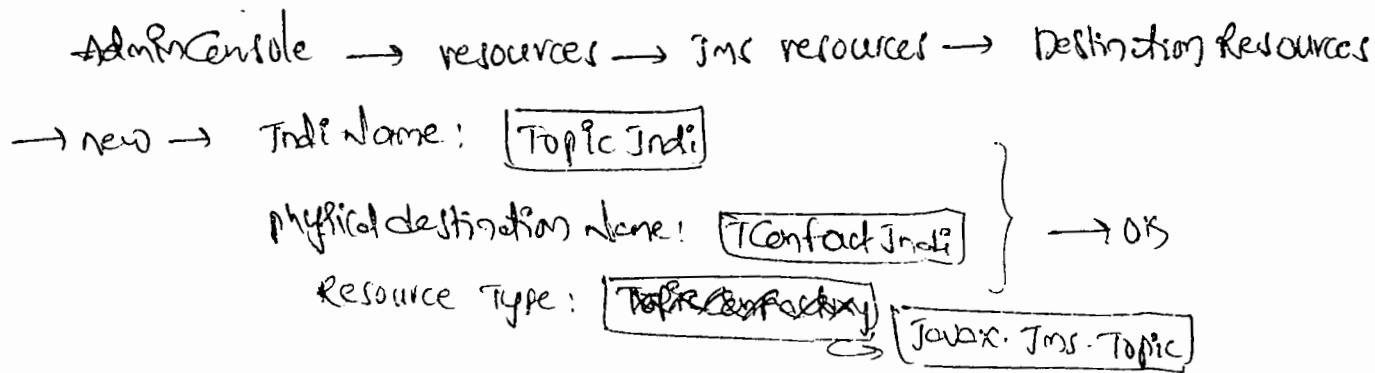
Admin console → resources → JMS resources → Connection factories → new → Jndi Name: **Contact Indi** Resource type: **[TopicCon\_factory]** → ok

Step ③: Create Queue Connection factory.

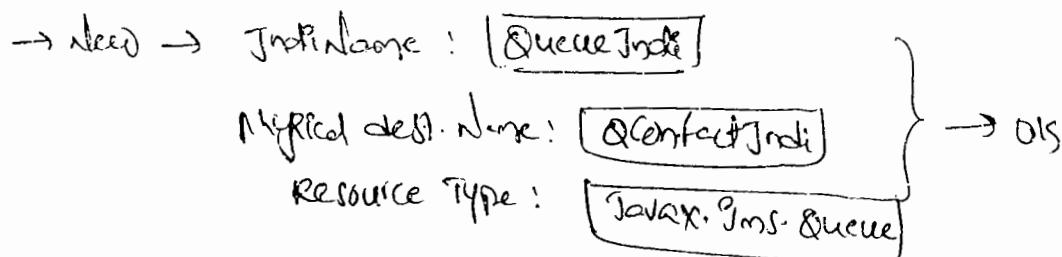
Admin console → resources → JMS resources → Connection factories → new → Jndi Name: **QContact Indi** → Resource type: **[QueueConfactory]** → ok

Step ④: Create Topic Definition,

Topic name: **testTopic** and Subscriptions for it.



Step⑥: Create Queue Definition



Note: All the above four resources will be registered with Registry automatically.

JMS API means JavaX.Jms package.

Every server supplied jar file that represents Javaee app's contains this JMS API.

- In Glassfish → Javaee.jar
- weblogic → weblogic.jar
- JBoss → JBoss-Javaee.jar

16/12/11 For ex app on plain Jms based PTP domain messaging

refer page no 15, 16 of Handout.

We can develop the Receiver/Subscriber application to read the msgs from destination either in synchronous or asynchronous mode.

In Asynchronous mode the msg listener should be registered with destination whenever the a msg arrives to the destination the JMS provider delivers that msg by calling onMessage() of

Message listener which is implemented in <sup>(Receives)</sup> client application.

The receiver/listener application can use message selector as small queries to filter messages that are required to be received from destination.

### Spring JMS

In Spring JMS programming JMS template class is there providing abstraction layer on plain JMS programming.

To create the code obj connection factory obj, destination object (Topic or Queue) are the basic objects.

for Spring JMS based JMS messaging domain application refer page: 6 to 9 of handout.

The Receiver/listener applications can consume the messages from destination synchronously or asynchronously. And this process never disturbs the Asynchronous Communication. By nature JMS allows only Asynchronous Communication b/w sender and receiver.

Receiver can consume msg from destination synchronously through receive() method (all). And we can use the MessageListener's onmessage() method to receive asynchronously,

while running Pub/Sub model programs first start subscriber program (one or more) then run publisher program.

In pub/sub model there is a timing dependency b/w publisher and subscriber, but it still gives Asynchronous Communication.

7/11/11

## WEB SERVICES

Web services is a distributed technology, it allows us to develop interoperable distributed applications.

language      Architecture      Platform Independent etc  
Independent      independent

Generally the server app of web services is web app and client app is stand alone or web application.

The server app of web service contains 6 methods and client app calls them from local or remote location.

Some Terminologies,

WSDL → web service description language

UDDI → Universal Discovery and Description and Integration

SOAP/REST → (Simple Object Access Protocol) / Representational State protocol.

Service Interface in webservice environment will be there in the form of WSDL document.

webservice client uses WSDL doc to develop his client app.

webservice client interacts with webservice server app by using SOAP.

The 6-components details webservice server app will be registered with UDDI registry in the form of WSDL doc.

To develop webservice server and client app in Java environment

we can use Jax-rpc, Jax-ws api's or we can use Axis, Metro kind of Jax-ws based frameworks.

Jax-ws api is built-in api of Jdk 1.6,

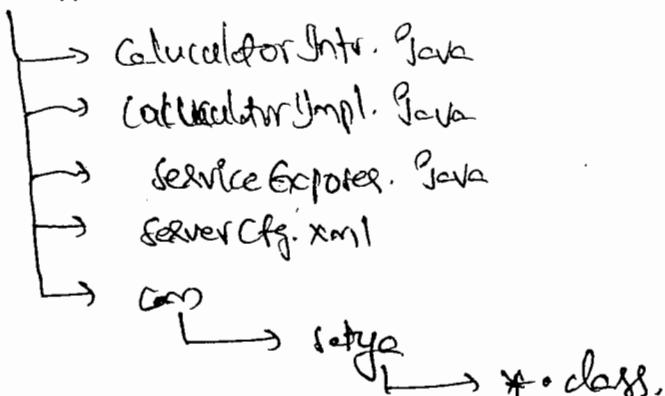
Spring web services provide abstraction layer on plain Jax-ws based web services programming.

- spring supplies built-in webserver / container in spring s/w itself to manage sever app of webservices
- supplier built-in UDDI Registry
- Allows to export webservices related to components through dependency injection.
- Allows to get webservices related to component reference in client app through dependency injection.

\* The Server app of webservices in Spring Environment :-

E:\APPS\WS

  └ serverapp



### CalculatorIntf.java

```

package com.factory;
public interface CalculatorIntf {
    public String sum(int x, int y);
}
  
```

### CalculatorImpl.java

```

package com.factory;
import java.rmi.Remote;
import java.rmi.RemoteException;
import org.springframework.ws.soap.binding.SoapBinding;
  
```

```

import org.springframework.service;
  
```

@ Soap Binding ( style = soapBinding.style.RPC, use = soapBinding,  
use.LITERAL, parameterStyle = soapBinding.parameterStyle.  
WRAPPED )

④ WebService (ServiceName = "craigws")

@ service ("cataly")

public class CalculatorImpl implements CalculatorInt

۱۰

@ webMethod

```
public String sum(int x, int y)  
{  
    return "" + (x+y);  
}
```

Here:

@ soapBinding → maps the given web service details with soap msg protocol

@WebService → Marks current class as web service related to component.

@Service → Makes current class as Spring Bean.

`@WebMethod` → Makes current method as operation of web service.

## springCfg.xml

<beans> . . . . . spring-bean-3.0.xsd spring-context-3.0.xsd >

```
<context:component-scan base-package = "com.satyu"/>
```

<bean id="axis" = "org.apache.axis.AxisEngine"> <!-- org.apache.axis:SimpleAxisServiceExporter -->

< property name="baseAddress"

```
</beans>           value="http://localhost:5678"/> />
```

</beans>

starts @ west service & springs built-in

well feathered at 5678 post number and exposures

@WebService based Java class to Registry as web service

(Collector impl.)

## Service Exposer. Java

```

package com.setya;
import org.springframework.util.*;
public class ServiceExposer
{
    public void main(String [] args)
    {
        FileSystemApplicationContext ctx = new
            FileSystemApplicationContext("springcfg.xml");
        System.out.println("Service Exposed successfully");
    }
}

```

## Jar files in classpath:

- 1. org.sf.context-3.x.jar
  - 2. org.sf.beans-3.x.jar
  - 3. org.sf.core-3.x.jar
  - 4. org.sf.expression-3.x.jar
  - 5. org.sf.web-3.x.jar
  - 6. org.sf.asm-expression-3.x.jar
  - 7. common-logging.jar → collect from spring 3.x.
- Collects from spring 3.x

\* To run the application :> Java com.setya.ServiceExposer

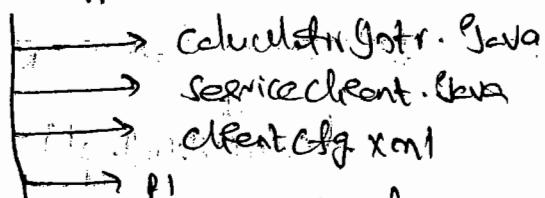
\* To see the wsdl document of above web service :

→ <http://localhost:8080/Calculator>

## Client Application:

E:\apps\ws

↳ ClientApp



## calculatorIntf.java (Develop Based on server app)

```

package pl;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC, use = SOAPBinding.Use.LITERAL,
    parameterStyle = SOAPBinding.ParameterStyle.WRAPPED)

```

### public interface calculatorIntf

```

    public String sum(int x, int y);

```

#### <beans>

(Gathers basic reference  
of web service from  
WSDL Registry)

<beans> . . . . . (same as server app) . . . . . >



<bean id="gs" class="org.gf.reading.jaxws.JAXWSPortProxy

factoryBean">

<property name="wsdlDocumentUrl"

value="http://localhost:8081/calculator?wsdl"/>

<property name="serviceName" value="CalculatorWS"/> w.s. logical name

<property name="namespaceURI" value="http://sdyा.com"/> collect  
from wSDL doc

<property name="serviceInterface" value="pl.calculatorIntf"/>

<property name="portName" value="calculatorImplPort"/>

</beans>

</beans>

### serviceclient.java

```

package pl;

```

### public class serviceclient

```

    p.s.r.m (String arg1)

```

filesystemApplicationContext ctx = new filesystemApplicationContext("client.xml");

calculatorIntf obj = (calculatorIntf) ctx.getBean("gs");

```

    } s.o.n(Mbi.sum((x,y)));

```

Jar files (same as before)

④ org.gf.aop-3.x.jar

④ aopalliance-1.0.jar  
(from internet)

## Spring security

15/12/11

Programmers are not responsible to take care of App security

They are most responsible for Application level security

App level security is all about performing authentication and authorization. Checking the identity of user by using username and password is called as Authentication. Checking the access permissions of a user on a particular resource of app is called as Authorization.

Ex: ~~else~~ To get into bank app every emp must be authenticated. (His uname, pwd will be verified). To get into each module the Access permissions of a user on that particular module will be verified.

Initially spring has given "acegi" fw as security fw to secure spring based applications.

Now "acegi" has become spring security fw from spring 2.5.

The org.springframework.context.ContextLoaderListener is a servlet-Context listener, which activates spring's web app context container either during server startup or deployment of webapp, when Context-obj is created.

org.springframework.web.filter.DelegatingFilterProxy is the predefined servlet filter that takes all the req coming to web app and passes to ~~through~~ the springbeans that are configured in spring config file.

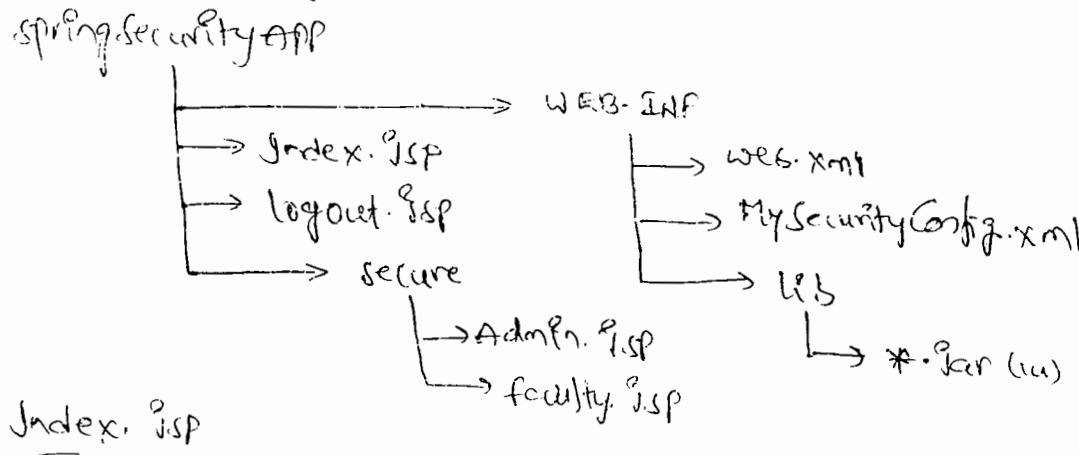
In this spring config file, we specify entries related to authentication and authorization by defining users and their roles.

## Ex-APP:

Step ①: Keep Spring 2.5 or 3.x jar ready.

Step ②: Keep spring-security-3.0.5-release.zip file ready (download from internet).

Step ③: Develop Spring Based webapp as shown below.



### Index.jsp

Any one can view this page.

<p><a href = "secure/Admin.jsp"> Admin page </a></p>

<p><a href = "secure/faculty.jsp"> faculty page </a></p>

→ These two hyperlinks generated resp will be trapped by  
servlet filter.

### web.xml :

<web-app>

<listeners>

<listener-class> org.springframework.web.context.ContextLoaderListener </listener-class>

</listeners>

<context-param>

<param-name> ContextConfigLocation </param-name>

<param-value> /WEB-INF/mysecurityConfig.xml </param-value>

</context-param>

<filters>

springSecurityFilterChain

<filter-name> </filter-name>

<filter-class> org.springframework.web.filter.DelegatingFilterProxy </filter-class>

```
<filter-mappings>
    <filter-names> spring security filter chain
        <filter-name>
            <url-pattern> /* <url-pattern>
        </filter-name>
    </filter-mappings>
</web-app>
```

↓  
Specify url-pattern to make security filters

## mySecurityConfig.xml

Specify URL-pattern to make `ServletFilter` trapping and taking all the requests, and responses.

<sec:global-methods ...>

<http use-expressions="true"/>

} on access

<intercept-url pattern="/index.jsp" access="permitAll"/>

<intercept-url pattern="/secure/admin.jsp" access="hasRole("admin")"/>

<intercept-url pattern="/secure/facility.jsp" access="hasRole("faculty")"/>

<form-login/> → launches a predefined template based form page  
<logout/> for authentication (username, password)

<remember-me/> → remembers the user during a session

<session-management invalid-session-url="/logout.jsp"/>

<concurrency-control max-sessions="3" error-if-maximum-exceeded="true"/>

</session-management>

</http>

<authentication-managers>  
<authentication-provider>  
<user-services>

defining users and roles ← { < User name = "sctya" password = "tech" authorities =  
"admin, faculty" />  
< User-name = "mfaor2" pwld = "mfaor2" authorities = "faculty" />  
</User-accounts> </dr-p> </arms>  
</beans>

## Admin.jsp

<h1> Admin page </h1>

Welcome Mr. <% = request.getAttribute("UserPrincipal").getUsername() %> <br>

Want go to faculty page <a href = "faculty.jsp"> Click here </a>

<p> <a href = ".. /j-spring-security Logout"> Logout </a>

## faculty.jsp

<h1> Faculty page </h1>

Welcome Mr. <% = request.getAttribute("UserPrincipal").getUsername() %> <br>

Want go to admin page <a href = "Admin.jsp"> Click here </a>

<p> <a href = ".. /j-spring-security Logout"> Logout </a>

## Logout.jsp

You have successfully loggedout <br>

<a href = "index.jsp"> Start again </a>

## Jar files in classpath:

1. Commons-logging.jar
2. SPRING-aop-3.0.3.RELEASE.jar
3. SPRING-aopm-3.x.jar
4. SPRING-beans-3.x.jar
5. SPRING-context-3.x.jar
6. SPRING-context-support-3.x.jar
7. SPRING-core-3.x.jar
8. SPRING-expression-3.x.jar
9. SPRING-security-config-3.x.jar
10. SPRING-security-core-3.x.jar
11. SPRING-security-web-3.x.jar

13. SPRING-web-3.x.jar

14. SPRING-webmvc-3.x.jar

## lib folder

same as class path  
total (14).

from nine faculty related zip file

\* procedure to add plugins to Eclipse to develop web app!:-

Step①: Install basic Eclipse 3.4.2 software.

Step②: Keep Internet Connection Ready.

Step③: Select and update the plugins, related to web app development.

Help menu → Software updates → Available software → select Java development and select web and GWT development → Install. → Close.

\* procedure to create web project in Eclipse IDE:-

Step①: Create web project

File menu → New → Other → Web → Dynamic Web Project → Next →

New Proj → Next → Finish.

Step②: Add a JSP program to webproj (NewProj)

Right click on project → New → Other → Web → JSP → Next →

Index → Next → Finish.

Step③: Configure Tomcat server with Eclipse IDE.

Window menu → Preferences → Server → Runtime environments → Add → Apache → Apache Tomcat 6 → Finish → OK

Step④: Run the project.

Right click on project → Run as → Run on server → Select

Tomcat → Next → Finish.

End of Slides

## Example on annotations based autowiring

① public interface Test  
  { public String sayHello();  
  }

② // TestBean.java

```
import org.springframework.*;  
import org.springframework.beans.factory.annotation.*;  
import java.util.*;
```

```
@Service  
public class TestBean implements Test
```

```
{ Date d;  
@Autowired // to perform autowiring on bean property  
public void setD( Date d )  
{ this.d = d; }  
public String sayHello()  
{ return "goodnight" + d.toString(); }  
}
```

③ democfg.xml

<beans>

<!-- Makes Spring container to recognize annotations based configs -->

<!--<context:component-scan base-package = ". " /-->

<bean id = "t b" class = "TestBean" />

<bean id = "dt" class = "java.util.Date" />

</beans>

④ TestClient.java

\* Activate Spring container (normal manner) → go to TestBean class obj to call sayHello on

that obj

first in classpath - Spring.jar, commons-logging.jar

Note: for annotations based aspectJ user defined advises creation using annotations refer app given in page # 10 of 5<sup>th</sup> dec test handout

07/12/11

## Annotations :-

- \* Annotations are java statements which will be written along with java source code as alternate for xml files for metadata operations and for resources config.

### Syntax of annotation:

@annotation-name(param1 = value1, param2 = value2...)

- \* In Java there are two types of annotations

1. Documentation annotations (available from gdk 1.1 version onwards). we use these annotations in documentation comments while generating api documentation.

Eg: @Author, @Param, @Return, @see. (/\* ... \*/)

2. Programming annotations: Available from gdk 1.5 onwards. we use these annotations

In stand alone app to make jvm or JRE interpreting code easily. Eg: @Override we use these annotations in high end app to config resources as alternate for xml files.

- \* All the java technologies that are given based on gdk 1.5 gives support for annotations for resources configurations like penellet 3.x, hibernate 3.3.x, spring 2.5.x, struts 2.x
- \* XML files based resources config gives good flexibility of modification without disturbing source code. But gives bad performance. Beacuz the XML parser is heavy weight &/co.
- \* Annotations based resources config gives good performance but bad flexibility of modifications Beacuz they will be written directly in java source code.
- \* Params of annotations are like attributes of xml files.

- \* In spring 2.5 while working with annotations we should also work with xml files. as in spring 3.0 we can take annotations as complete alternate for xml files.

- \* Annotations can be applied in 3 levels

1. field level, 2. method level, 3. resource level. (class/abstract/interface)

④ @component / @service annotations can be used to config java class as spring bean.

④ @autowired annotation can be used to config bean property for autowiring.

Ex:-

6/12/11

In spring AOP while developing advised we need to make our classes implementing the spring aop supplied interfaces. In aspectJ programming advised can be developed as spring aop independent classes.

- \* AspectJ advised can be linked with spring beans either by using schema based xml files or annotations.
- \* To work with AspectJ you gather the following two additional jar files from [www.springframework.org](http://www.springframework.org) (aspectjrt-1.6.0.jar, aspectjweaver.jar).
- \* For example app on distributed transaction management using 'AspectJ' on hibernate persistent logic refer app given in handout at 6/12/11
- + Jar file in class path →
  - 8 → H.B. jar file
  - 2 → spring jar file
  - 1 → jdbc4.jar
  - 1 → mysql-connector-java-3.0.8-stable-bin.jar
  - 2 → aspectjrt-1.6.8.jar, aspectjweaver.jar

### DB table in oracle:

#### Account\_table

(acno → number)	— 101
acname → varchar(20)	— rafy
balance → number	— 10000.

~~account~~

#### DB table in mysql → (logical db)

#### account\_table

(acno → number int)	— 101
acname → varchar(20)	— rafy
balance → number int	— 50000.

- + For spring AOP aspectJ based advised user defined advised development through xml schema entries refer app given in page nos 9 & 10 of 5th Dec handout

### Jars in class path

spring.jar  
commons-logging.jar  
aspectjrt-1.6.8.jar, aspectweaver.jar

卷之三

48

attach to the handout

### MyAspect.java:

```

import org.aspectj.lang.annotation.*;
@Aspect
public class MyAspect
{
    @Pointcut("execution(* TestBean.*(..))")
    private void testMethod() {} // dummy method as advice which will be used as a reference
                                // to configure other advices.

    @Before("testMethod()")
    public void beforeMethod()
    {
        System.out.println("am before advice");
    }

    @AfterReturning("testMethod()")
    public void afterReturningMethod()
    {
        System.out.println("am after advice");
    }
}

```

acts as before advice

acts as after advice

### ④ Other annotations to develop other types of advices

- ① `@AfterThrowing` → to make Java method as throwing advice
- ② `@Around` → to make Java method as around advice

### Spring.xml:

```

<beans>
    <!-- spring-app-2.5.xsd -->

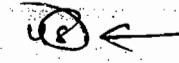
    <bean id="qd1" class="TestBean"/>
    <bean id="qd2" class="MyAspect"/>
    <prop:aspect-autoproxy/> // to convert normal spring bean class obj as aspect obj based
    </beans>

```

### ④ client.java Same as client.java of page no - 10 belongs to 5<sup>th</sup> Dec handout

Jars in class path:

- spring.jar
- commons-logging.jar
- aspectjrt-1.6.0.jar
- aspectweaver.jar



\*2 This applies throws method based throws advice of myAdvices class on demo() method of TestBean class.

```
129  http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
130      http://www.springframework.org/schema/aop
131      http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
132
133  <bean id="id1" class="TestBean"/>
134  <bean id="id2" class="MyAdvices"/> pointing to advice class
135  <aop:config>
136  <aop:aspect ref="id2"> //refers to id2. l34
137  <aop:pointcut id="pt1" expression="execution(* TestInter.demo(..))"/> || this pointcut points to demo method of testInter
138  <aop:after-throwing pointcut-ref="pt1" method="throwsMethod" throwing="nfe"/> demo method of testInter
139
140  <aop:pointcut id="pt2" expression="execution(long TestInter.*(..))"/> || points to findfact() method of testInter class
141  <aop:around pointcut-ref="pt2" method="aroundMethod"/> whose return type is long
142  </aop:aspect>
143  </aop:config>
144  </beans>
145 ----- client.java -----
146 import org.springframework.beans.factory.*;
147 import org.springframework.context.*;
148 import org.springframework.context.support.*;
149 public class client {
150     public static void main(final String[] args) {
151         BeanFactory ctx = new ClassPathXmlApplicationContext("spring.xml");
152         TestInter ti=(TestInter)ctx.getBean("id1");
153         try
154         {
155             ti.demo("sathya");
156         }
157         catch(Exception e)
158         {}
159         System.out.println("-----");
160         long m1 = ti.findfact(20);
161         System.out.println(m1);
162         System.out.println("-----");
163         long m2 = ti.findfact(58);
164         System.out.println(m2);
165     }
166 }
167 }
168 ----- App --> AspectJ annotations -----
169 ----- TestInter.java -----
170 public interface TestInter
171 {
172     void demo(String name);
173     long findfact(int k);
174 }
175 ----- TestBean.java -----
176 public class TestBean implements TestInter
177 {
178     public void demo(String name) HB-method 1
179     {
180         int k=Integer.parseInt(name);
181     }
182     public long findfact(int k) HB-method 2
183     {
184         long f=1;
185         for(int i=1; i<=k; i++)
186         {
187             f = f * i;
188         }
189         return f;
190     }
191 }
192 }
```

\* In this application on demo & method of TestBean class throws advice is applied whose implementation is done in throwsMethod() method of myAdvices class.

My aroundAdvice is applied on findfact() method of TestBean class whose implementation is done in aroundMethod() method of myAdvices class.

\* The following jar files in the classpath:  
Spring.jar, Commons-logging.jar, AspectJ.jar, AspectJweaver.jar

\*3. This lines applies around method based around advice of myAdvices class on findfact() method of testBean class.

```

65     FooService foo = (FooService) ctx.getBean("fooService");
66     foo.getFoo("raja", 12);
67     foo.getAfter();
68     foo.getBefore("raja");
69   }
70 }
71 -----
72 App --> Spring AOP schema support for throws and around advices
73 -----
74 -----TestInter.java-----Spring Interface
75 public interface TestInter {
76 {
77     void demo(String name); } two B-methods
78     long findfact(int k);
79 }
80 -----TestBean.java-----Spring Bean
81 public class TestBean implements TestInter
82 {
83     public void demo(String name)
84     {
85         int k = Integer.parseInt(name);
86     }
87     public long findfact(int k)
88     {
89         long f=1;
90         for(int i=1; i<=k; i++)
91         {
92             f = f * i;
93         }
94         return f;
95     }
96 }
97 } one Advice class having both throws advice, around advice related methods.
98 -----MyAdvices.java-----
99 import org.aspectj.lang.ProceedingJoinPoint;
100 import org.aspectj.lang.JoinPoint;
101 public class MyAdvices {
102     public void throwsMethod(JoinPoint jp, NumberFormatException nfe)
103         throws Throwable
104     {
105         System.out.println("this advice is applied for: "
106                         +jp.getSignature().getName()); gives current B-method name on which this advice is applied.
107         System.out.println("The exception occurred is : "+nfe.getMessage());
108         System.out.println("advice from throws method");
109     }
110 }
111 public Object aroundMethod(ProceedingJoinPoint pjp) throws Throwable
112 {
113     System.out.println("this advice is applied for "
114                         +pjp.getSignature().getName()); holds current B-method details and also useful to call the B-method from the advice.
115     long x = System.currentTimeMillis();
116     Object retval=pjp.proceed();
117     long y = System.currentTimeMillis();
118     System.out.println("The method execution taken "
119                         +(y-x)+" milliseconds"); gives current B-method name.
120     System.out.println("The above service is from around advice");
121     return retval;
122 }
123 }
124 -----spring.cfg file-----
125 <beans xmlns="http://www.springframework.org/schema/beans"
126   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
127   xmlns:aop="http://www.springframework.org/schema/aop"
128   xsi:schemaLocation="http://www.springframework.org/schema/beans

```

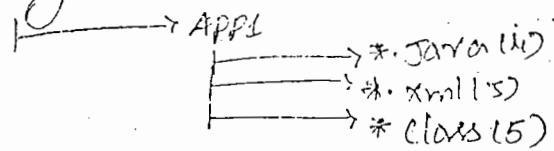
```

1 -----App--> Spring AOP with schema support for before and after advices
2
3 -----
4 -----FooService.java-----
5 public interface FooService {
6     FooService getFoo(String fooName,int age);
7     void getAfter();
8     void getBefore(String myName);
9 }
10 -----DefaultFooService.java-----
11 public class DefaultFooService implements FooService {
12     public FooService getFoo(String fooName, int age) {
13         System.out.println("getFoo(fooName,age) business logic");
14         return null;
15     }
16     public void getAfter() {
17         System.out.println("getAfter() business logic");
18     }
19     final public void getBefore(String myName) {
20         System.out.println("getBefore(myName) business logic");
21     }
22 }
23 -----SimpleProfiler.java-----
24 public class SimpleProfiler {
25     public void afterMethod() throws Throwable {
26         System.out.println("After the method call");
27     }
28     public void beforeMethod(String myName){
29         System.out.println("My name is "+myName);
30     }
31 }
32 -----spring.xml-----
33 <beans xmlns="http://www.springframework.org/schema/beans"
34     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
35     xmlns:aop="http://www.springframework.org/schema/aop"
36     xsi:schemaLocation="http://www.springframework.org/schema
37     /beans http://www.springframework.org/schema/beans/spring-beans
38     -2.5.xsd http://www.springframework.org/schema/aop http:
39     //www.springframework.org/schema/aop/spring-aop-2.5.xsd">
40 <!--this is the object that will be proxied by Spring's AOP infrastructure-->
41 <bean id="fooService" class="DefaultFooService"/>
42
43 <!-- this is the actual advice itself -->
44 <bean id="profiler" class="SimpleProfiler"/>
45
46 <aop:config>
47     <aop:aspect ref="profiler">
48         <aop:pointcut id="aopafterMethod" expression=
49             "execution(* FooService.*(..))"/>
50         <aop:after pointcut-ref="aopafterMethod" method="afterMethod"/>
51         <aop:pointcut id="aopBefore" expression=
52             "execution(* FooService.getBefore(String)) and args(myName)"/>
53         <aop:before pointcut-ref="aopBefore" method="beforeMethod"/>
54     </aop:aspect>
55
56 </aop:config>
57 </beans>
58 -----client.java-----
59 import org.springframework.beans.factory.*;
60 import org.springframework.context.*;
61 import org.springframework.context.support.*;
62 public class client {
63     public static void main(final String[] args) throws Exception {
64         BeanFactory ctx = new ClassPathXmlApplicationContext("spring.xml");

```

```
193     if(res)
194         System.out.println("Money Transferred");
195     else
196         System.out.println("Money not Transferred");
197     }
198 }
199
200
```

E:\Apps → Spring Tool



C:\APP3\Spring\Tx\APP3> javac \*.java

> Java clientAPP

## DB table

### HB-Account

acid numbers  
holdername vaguchatziad

balance number (8)  
two records into HB-Account

insert two records into AB-Album

UVA files in classpath: 2-Spring Jars, 1-objback.jar and 8-HB jars file.

```

129     } //try
130
131     catch(Exception e)
132     {
133         status=false;
134         ts.setRollbackOnly();
135         System.out.println("Tx is rolledback");
136     }
137     return new Boolean(status);
138     } //doInTransaction()
139 }
140
141     return result.booleanValue(); // returns simple true or false as the return
142     } //transferMoney()
143
144 } //class
-----Spring.cfg.xml-----
145 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
146 "http://www.springframework.org/dtd/spring-beans.dtd">
147 <beans>
148     <bean id="myds" class="org.springframework.jdbc.datasource
149         .DriverManagerDataSource"> Spring Bean giving Jdb
150         <property name="driverClassName" > <value>
151             oracle.jdbc.driver.OracleDriver</value></property> data&source desct.
152         <property name="url" > <value>
153             jdbc:oracle:thin:@localhost:1521:satya</value></property>
154         <property name="username" > <value>scott</value></property>
155         <property name="password" > <value>tiger</value></property>
156     </bean>
157     <bean id="mySessionFactory" class="org.springframework
158         .orm.hibernate3.LocalSessionFactoryBean"> Spring Bean giving HB
159         <property name="dataSource" ref="myds"/> refer Lno:149
160         <property name="configLocation" > <value>
161             classpath:mycfg.xml</value></property> factory Object.
162     </bean> HB Cfg file:
163     <bean id="template" class="org.springframework.orm
164         .hibernate3.HibernateTemplate">
165         <constructor-arg> <ref bean=
166             "mySessionFactory"/> </constructor-arg> refer Lno:158
167     </bean>
168     <bean id="hbt" class="org.springframework
169         .orm.hibernate3.HibernateTransactionManager">  Transaction manager that can
170         <property name="sessionFactory" ref="mySessionFactory"/> perform tx mgmt on HB
171     </bean> refer Lno:158 persistence logic.
172
173     <bean id="tt1" class="org.springframework.transaction
174         .support.TransactionTemplate">
175         <property name="transactionManager" > <ref bean="hbt"/></property>
176     </bean> refer Lno:169
177     <bean id="db" class="DemoBean">
178         <property name="ht" > <ref bean="template"/></property> refer Lno:169
179         <property name="tt" > <ref bean="tt1"/></property> refer Lno:174
180     </bean> * Here two objects are injected to our Spring Bean class properties.
181
182 </beans>
-----ClientApp.java-----
183 import org.springframework.context.support.*;
184 public class ClientApp
185 {
186     public static void main(String args[])
187     {
188         FileSystemXmlApplicationContext ctx=
189         new FileSystemXmlApplicationContext("SpringCfg.xml");
190         Demo bean=(Demo)ctx.getBean("db");
191         boolean res=bean.transferMoney(101,102,3000);

```

```

65      <property name="holdername" column="HOLDERNAME"/>
66      <property name="balance" column="BALANCE"/>
67  </class>
68 </hibernate-mapping>
69 ----- Demo.java -----
70 //Demo.java
71 public interface Demo {
72 {
73     public boolean transferMoney(int srcid,int destid,float amt); // Declaration of the B-method.
74 }
75 ----- DemoBean.java -----
76 //DemoBean.java
77 import org.springframework.transaction.support.*;
78 import org.springframework.transaction.*;
79 import org.springframework.orm.hibernate3.*;
80
81 public class DemoBean implements Demo {
82 {
83     TransactionTemplate tt;
84     HibernateTemplate ht;
85     public void setHt(HibernateTemplate ht) {this.ht=ht;} // &txxxxl) methods supporting Setters
86     {
87         this.ht=ht;
88     }
89     public void setTt(TransactionTemplate tt)
90     {
91         this.tt=tt;
92     }
93     public float fetchBalance(int acid)
94     {
95         Account ac1=(Account)ht.get(Account.class,new Integer(acid));
96         return ac1.getBalance();
97     }
98     public boolean transferMoney(final int srcid,final int destid,final float amt)
99     {
100
101     Boolean result=(Boolean)tt.execute(new TransactionCallback()
102     {
103         public Object doInTransaction(TransactionStatus ts)
104         {
105             boolean status=false;
106
107             try
108             {
109                 //transferMoney Logic
110                 int r1=ht.bulkUpdate("update Account
111                 a1 set a1.balance=a1.balance-? where a1.acid=?",
112                 new Object[]{new Float(amt),new Integer(srcid)}); // Hal update query performing
113
114                 int r2=ht.bulkUpdate("update Account a1 set
115                 a1.balance=a1.balance+? where a1.acid=?",
116                 new Object[]{new Float(amt),new Integer(destid)}); // withdraw amt on source a/c.
117
118                 if(r1==0 || r2==0) // (operations) depositing amount
119                 {
120                     status=false;
121                     ts.setRollbackOnly();
122                     System.out.println("Tx is rolledback");
123                 }
124                 else // (operations) in destination a/c.
125                 {
126                     System.out.println("Tx is committed");
127                     status=true;
128                 }
129             }
130         }
131     }
132 }

```

To make outer class parameters visible to inner classes defined in that class  
 the parameters must be taken as final.

(12) ←

1 App1 (Programmatic local transaction management) of HB persistence logic

```

2 =====
3 <!--mycfg.xml-->
4 <!DOCTYPE hibernate-configuration PUBLIC
5   "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
6   "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
7
8 <hibernate-configuration>
9   <session-factory>
10    <property name="hibernate.connection.driver_class">
11      oracle.jdbc.driver.OracleDriver</property>
12      <property name="hibernate.connection.url">
13        jdbc:oracle:thin:@localhost:1521:satya</property>
14      <property name="hibernate.connection.username">
15        scott</property>
16      <property name="hibernate.connection.password">
17        tiger</property>
18      <property name="hibernate.dialect">
19        org.hibernate.dialect.Oracle9Dialect</property>
20      <property name="show_sql">true</property>
21      <mapping resource="Account.hbm.xml"/>
22    </session-factory>
23 </hibernate-configuration>
24 ----- Account.java -----
25 public class Account          HB POJO Class
26 {
27     int acid;
28     String holdername;
29     float balance;
30
31     public void setAcid(int acid)
32     {
33         this.acid=acid;
34     }
35     public int getAcid()
36     {
37         return acid;
38     }
39     public void setHoldername(String holdername)
40     {
41         this.holdername=holdername;
42     }
43     public String getHoldername()
44     {
45         return holdername;
46     }
47     public void setBalance(float balance)
48     {
49         this.balance=balance;
50     }
51     public float getBalance()
52     {
53         return balance;
54     }
55 }
56 ----- Account.hbm.xml ----- mapping file
57 <!DOCTYPE hibernate-mapping PUBLIC
58   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
59   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
60 <hibernate-mapping>
61   <class name="Account" table="HB_Account">
62     <id name="acid" column="ACID">
63       <generator class="increment"/>
64     </id>

```

PROPAGATION - REQUIRED

PROPAGATION - REQUIRESNEW

PROPAGATION - SUPPORTS

PROPAGATION - NOT SUPPORTED

PROPAGATION - NEVER

PROPAGATION - MANDATORY

```
129 <property name="url"> <value>  
130   jdbc:oracle:thin:@localhost:1521:satya</value> </property>  
131 <property name="username"><value>scott</value></property>  
132 <property name="password"> <value>tiger</value></property>  
133 </bean>  
134  
135 <bean id="mySessionFactory" class="org.springframework.orm  
136   .hibernate3.LocalSessionFactoryBean">  
137   <property name="dataSource" ref="myds"/> refers L00:125 Sessionfactory object  
138   <property name="configLocation"><value>  
139     classpath:mycfg.xml</value></property>  
140 </bean>  
141  
142 <bean id="template" class="org.springframework  
143   .orm.hibernate3.HibernateTemplate">  
144   <constructor-arg><ref bean="mySessionFactory"/></constructor-arg>  
145 </bean> refers L00:135  
146  
147 <bean id="hbt" class="org.springframework.orm.hibernate3  
148   .HibernateTransactionManager">  
149   <property name="sessionFactory" ref="mySessionFactory"/>  
150 </bean> refers L00:135  
151 <bean id="db" class="DemoBean">  
152   <property name="ht"><ref bean="template"/></property> refers L00:125 Spring Bean 1/2  
153 </bean> refers L00:142  
154 <bean id="tas" class="org.springframework.transaction  
155   .interceptor.NameMatchTransactionAttributeSource">  
156   <property name="properties"> pointcutAdvisor applying tx attribute on the B-method  
157   <props>  
158     <prop key="transferMoney">PROPAGATION_REQUIRED</prop> Propagation attribute  
159   </props>  
160   </property>  
161 </bean>  
162 <bean id="tfb" class="org.springframework.transaction  
163   .interceptor.TransactionProxyFactoryBean">  
164   <property name="target"><ref bean="db"/></property> refers L00:151  
165   <property name="transactionManager"><ref bean="hbt"/></property> refers L00:147  
166   <property name="transactionAttributeSource">  
167   <ref bean="tas"/></property> refers L00:154  
168 </bean> gives DemoBean class object as proxy object by enabling tx service on it.  
169 </beans>  
170  
ClientApp.java  
171 import org.springframework.context.support.*;  
172 public class ClientApp  
173 {  
174   public static void main(String args[ ])  
175   {  
176     FileSystemXmlApplicationContext ctx=new  
177       FileSystemXmlApplicationContext("SpringCfg.xml");  
178     Demo bean =(Demo)ctx.getBean("db");  
179     try  
180     {  
181       boolean res=bean.transferMoney(101,102,3000);  
182       if(res)  
183         System.out.println("Money Transferred");  
184       else  
185         System.out.println("Money not Transferred");  
186     }  
187     catch(Exception e)  
188     {  
189       System.out.println("Money not Transferred");  
190     }  
191   }  
192 }
```

- \* This application compilation and execution process is same as App ①
- \* This client application always calls the B-method without Tx. So, the B-method transferred money always runs with new transaction because the transaction attribute required

```

65      <id name="acid" column="ACID" >
66          <generator class="increment"/>
67      </id>
68      <property name="holdername"/>
69      <property name="balance"/>
70  </class>
71 </hibernate-mapping>
72 -----Demo.java-----
73 //Demo.java
74
75 public interface Demo
76 {
77     public boolean transferMoney(int srcid,int destid,float amt)
78             throwsException;
79 }
-----DemoBean.java-----
80 //DemoBean.java
81
82 import org.springframework.transaction.support.*;
83 import org.springframework.transaction.*;
84 import org.springframework.orm.hibernate3.*;
85
86 public class DemoBean implements Demo
87 {
88     HibernateTemplate ht;
89     public void setHt(HibernateTemplate ht) // 8th(xxx) methods supporting better injection
90     {
91         this.ht=ht;
92     }
93     public boolean transferMoney( int srcid, int destid, float amt)
94             throwsException
95             {
96                 boolean status=false;
97                 int r1=ht.bulkUpdate("update Account a1 set a1.balance
98                             =a1.balance-? where a1.acid=?",
99                             new Object[]
100                             {new Float(amt),new Integer(srcid)});
101                int r2=ht.bulkUpdate("update Account a1 set a1.balance
102                             =a1.balance+? where a1.acid=?",
103                             new Object[]{new Float(amt),new Integer(destid)});
104                if(r1==0 || r2==0)
105                {
106                    status=false;
107                }
108                else
109                {
110                    System.out.println("Tx committed");
111                    status=true;
112                }
113                if(status==false){
114                    throw newException();
115                }
116                return status;
117 } //transferMoney()
118
119 } //class
120 -----SpringCfg.xml-----
121 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
122 "http://www.springframework.org/dtd/spring-beans.dtd">
123
124 <beans>
125     <bean id="myds" class="org.springframework.jdbc
126             .datasource.DriverManagerDataSource">
127         <property name="driverClassName"> <value>
128             oracle.jdbc.driver.OracleDriver</value></property>

```

2

giving  
Spring Bean having JDBC Data Source  
object

04/10/11

K. KAVI +91-8019875108



## \* Spring With Hibernate Apps With Transaction mgmt.

1 App2 (Local declarative Tx management by using HB Tx manager).

```
2 -----
3 -----mycfg.xml-----
4 <!DOCTYPE hibernate-configuration PUBLIC
5 "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
6 "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
7
8 <hibernate-configuration>
9   <session-factory>
10    <property name="hibernate.connection.driver_class">
11      oracle.jdbc.driver.OracleDriver</property>
12    <property name="hibernate.connection.url">
13      jdbc:oracle:thin:@localhost:1521:satya</property>
14    <property name="hibernate.connection.username">
15      scott</property>
16    <property name="hibernate.connection.password">
17      tiger</property>
18    <property name="hibernate.dialect">
19      org.hibernate.dialect.Oracle9Dialect</property>
20    <property name="show_sql">true</property>
21    <mapping resource="Account.hbm.xml"/>
22  </session-factory>
23 </hibernate-configuration>
24
25 -----Account.java-----
26 public class Account {
27 {
28     int acid;
29     String holdername;
30     float balance;
31
32     public void setAcid(int acid)
33     {
34         this.acid=acid;
35     }
36
37     public int getAcid()
38     {
39         return acid;
40     }
41     public void setHoldername(String holdername)
42     {
43         this.holdername=holdername;
44     }
45     public String getHoldername()
46     {
47         return holdername;
48     }
49     public void setBalance(float balance)
50     {
51         this.balance=balance;
52     }
53     public float getBalance()
54     {
55         return balance;
56     }
57 }
58 -----Account.hbm.xml-----
59 <!DOCTYPE hibernate-mapping PUBLIC
60 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
61 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
62
63 <hibernate-mapping>
64   <class name="Account" table="HB_Account">
```

JAR files in classpath :-

- o - HIS JAR file
- o - Spring JAR files
- 1 → object4J.jar
- 1 → mysql-connector-java-3.0.8-stable-bin.jar
- 2 → aspectJlt-1.6.8.jar, aspectjweaver.jar

DB table in oracle :-

Account-table

<u>acno (int)</u>	<u>acname (vc)</u>	<u>balance (n)</u>
101	Raju	90000

DB table in mysql :-

Account-table (Logical DB name db)

<u>acno (integer)</u>	<u>acname - vc</u>	<u>balance (n)</u>
102	Ravi	60000

```

139     <value>
140         hibernate.dialect=org.hibernate.dialect.OracleDialect
141     </value>
142   </property>
143 </bean>
144 <bean id="betaSessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
145   <property name="dataSource" ref="drds2"/> // refer L00: 142
146   <property name="mappingResources"> // gives HB Sessionfactory object pointing to
147     <list>
148       <value>account2.hbm.xml</value> mysql
149     </list>
150   </property>
151   <property name="hibernateProperties">
152     <value>
153         hibernate.dialect=org.hibernate.dialect.MySQLDialect
154     </value>
155   </property>
156 </bean>
157
158 <bean id="temp1" class="org.springframework.orm.hibernate3.HibernateTemplate"> // HB Template class object
159   <property name="sessionFactory" ref="aliasSessionFactory"/> // refer L00: 142
160 </bean>
161
162 <bean id="temp2" class="org.springframework.orm.hibernate3.HibernateTemplate"> // HB Template class object
163   <property name="sessionFactory" ref="betaSessionFactory"/> // refer L00: 144
164 </bean>
165
166 <bean id="tb" class="testbean"> // Injecting two hibernate template class objects to class
167   <property name="ht1"> <ref bean="temp1"/> </property>
168   <property name="ht2"> <ref bean="temp2"/> </property> Spring bean class
169 </bean>
170
171 <tx:advice id="txAdvice" transaction-manager="txmgr"> // refer L00: 147
172   <tx:attributes>
173     <tx:method name="" propagation="REQUIRED"/> // The transaction attribute required is applied on
174     </tx:attributes> B methods of Spring bean class.
175   </tx:advice>
176
177 <aop:config>
178   <aop:pointcut id="dtxops" expression="execution(* testinter.*(..))"/> // pointcut configuration pointing to
179   <aop:advisor advice-ref="txAdvice" pointcut-ref="dtxops"/> the spring bean (testbean) that
180   <aop:config> refer L00: 172 179 implements testinter interface.
181 </beans>
182
183 <client.java>
184
185 import org.springframework.core.io.*;
186 import org.springframework.beans.factory.*;
187 import org.springframework.beans.factory.xml.*;
188 public class client
189 {
190     public static void main(String args[])
191     {
192
193         Resource res=new ClassPathResource("demo.xml");
194         BeanFactory factory=new XmlBeanFactory(res);
195         testinter ob=(testinter)factory.getBean("tb"); // refer L00: 167
196         ob.transferMoney(101,102,3000); // calls the B method from client application without Txn
197         // the transferMoney() method always runs with new tx because the transaction
198         // attribute that is configured is required.
199
200
201

```

- \* talks about AspectJ style of Configuring transaction service on Spring bean class & methods.
- \* At L00: 180 the aop:advisor tag is applying transaction service L00: 172 on pointcut based spring specified at L00: 179.

```

70
71     int r1=ht.bulkUpdate("update Account ac set ac.balance=ac.balance-? where ac.acno=?", 1 operation withdraw
72             new Object[]{new Double(amt),new Integer(srcid)}); amount operation
73     int r2=ht1.bulkUpdate("update Account ac set ac.balance=ac.balance+? where ac.acno=?", 1 operation
74             new Object[]{new Double(amt),new Integer(destid)}); 2 operation
75     if(r1==0 || r2==0)
76     {
77         System.out.println("Tx is rolledback");
78         throw new Exception();
79     }
80     else
81     {
82         System.out.println("Tx is committed");
83     }
84 } //transferMoney
85
86 }
87 -----demo.xml----- Spring cfg file
88 <?xml version="1.0" encoding="UTF-8"?>
89 <beans xmlns="http://www.springframework.org/schema/beans"
90   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
91   xmlns:aop="http://www.springframework.org/schema/aop"
92   xmlns:tx="http://www.springframework.org/schema/tx"
93   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/
94   http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
95   http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
96
97 <bean id="drds1" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
98   <property name="driverClassName">
99     <value>oracle.jdbc.driver.OracleDriver</value> // bean giving JDBC DataSource object pointing to
100    </property> oracle
101   <property name="url">
102     <value>jdbc:oracle:thin:@localhost:1521:satya</value>
103   </property>
104   <property name="username">
105     <value>scott</value>
106   </property>
107   <property name="password">
108     <value>tiger</value>
109   </property>
110 </bean>
111
112 <bean id="drds2" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
113   <property name="driverClassName">
114     <value>org.gjt.mm.mysql.Driver</value> // bean giving JDBC DataSource object pointing to
115   </property> mysql
116   <property name="url">
117     <value>jdbc:mysql://localhost:3306/db1</value>
118   </property>
119   <property name="username">
120     <value>root</value>
121   </property>
122   <property name="password">
123     <value>root</value>
124   </property>
125 </bean>
126
127 <bean id="txmgr" class="org.springframework.transaction.jta.JtaTransactionManager"> // Special tx manager to
128   <!-- distributed tx mgmt. -->
129   <bean id="alfaSessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
130     <property name="dataSource" ref="drds1"/> // prefer LWD: q7
131     <property name="mappingResources">
132       <list>
133         <value>account1.hbm.xml</value> // gives SessionFactory object pointing to
134       </list> oracle
135     </property>
136     <property name="hibernateProperties">
137       <value>
```

05/12/11 K. Ravi +91-8019875108



## Application on Distributed Transactions.

```
1 public class Account {  
2     private int acno;  
3     private String acname;  
4     private double balance;  
5     public int getAcno() {  
6         return acno;  
7     }  
8     public void setAcno(int acno) {  
9         this.acno = acno;  
10    }  
11    public String getAcname() {  
12        return acname;  
13    }  
14    public void setAcname(String acname) {  
15        this.acname = acname;  
16    }  
17    public double getBalance() {  
18        return balance;  
19    }  
20    public void setBalance(double balance) {  
21        this.balance = balance;  
22    }  
23 }
```

Account.java  
HIB POJO class

\* Declarative distributed tx mgmt on mysql, oracle DB by using AspectJ.

```
25 }  
26 -----account1.hbm.xml----- mapping file pointing to source-table
```

```
27 <?xml version="1.0"?>  
28 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
29 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  
30 <.hibernate-mapping>  
31 <class name="Account" table="account_tab1">  
32     <id name="acno"/> source table name holding source Ac details  
33     <property name="acname"/>  
34     <property name="balance"/>  
35 </class>
```

```
36 </hibernate-mapping>
```

```
37 -----account2.hbm.xml----- mapping file pointing to destination table
```

```
38 <?xml version="1.0"?>  
39 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
40 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
```

```
41 <hibernate-mapping>
```

```
42     <class name="Account" table="account_tab2">  
43         <id name="acno"/>  
44         <property name="acname"/>  
45         <property name="balance"/>  
46     </class>
```

```
47 </hibernate-mapping>
```

```
48 -----testinter.java----- Spring interface
```

```
49  
50 public interface testinter {  
51     public void transferMoney(int srcid,int destid,double amt) throws Exception;  
52 }
```

```
53 -----testbean.java----- Spring Bean Runtime
```

```
54 import org.springframework.orm.hibernate3.*;  
55 public class testbean implements testinter
```

```
56 {  
57     HibernateTemplate ht;
```

```
58     HibernateTemplate ht1;
```

```
59     public void setHt(HibernateTemplate ht)
```

```
60     {  
61         this.ht=ht;
```

```
62     }  
63     public void setHt1(HibernateTemplate ht1)
```

```
64     {  
65         this.ht1=ht1;
```

```
66     }  
67 }
```

```
68  
69     public void transferMoney(int srcid,int destid,double amt) throws Exception
```

Runtime

88

For example application local transaction management or HB persistence logic  
by using HB transaction manager refer application ① & App ② of handout  
Given of 04/12/11.

→ who to make Tx manager to rollback the transaction Even though checked  
exception is raised in B method in Declarative transaction management

→ The B method catch block catch that checked Exception and return  
as unchecked exception.

In distributed transaction management multiple DB's involved so  
we need to use the Jta transaction management.

AspectJ is the enhancement of spring AOP to apply advices on spring bean  
and to generate proxy object.

AspectJ allows both XML, annotations based configurations.

05/12/11

In Spring AOP while developing advices we need to make our classes implement  
the Spring API supplied interfaces.

In AspectJ programming Advices can be developed on Spring API  
independent classes.

AspectJ Advices can be linked with Spring beans either by using schema  
based XML files or annotations.

To work with AspectJ programming gather the following two additional  
jar files from [www.springframework.org](http://www.springframework.org) website.

\* AspectJrt-1.6.0.Jar  
\* AspectJweaver.Jar

For example application on distributed transaction management  
using AspectJ on Hibernate persistence logic refer application given  
in handout of 05/12/11

## Summary table on Transaction attributes:-

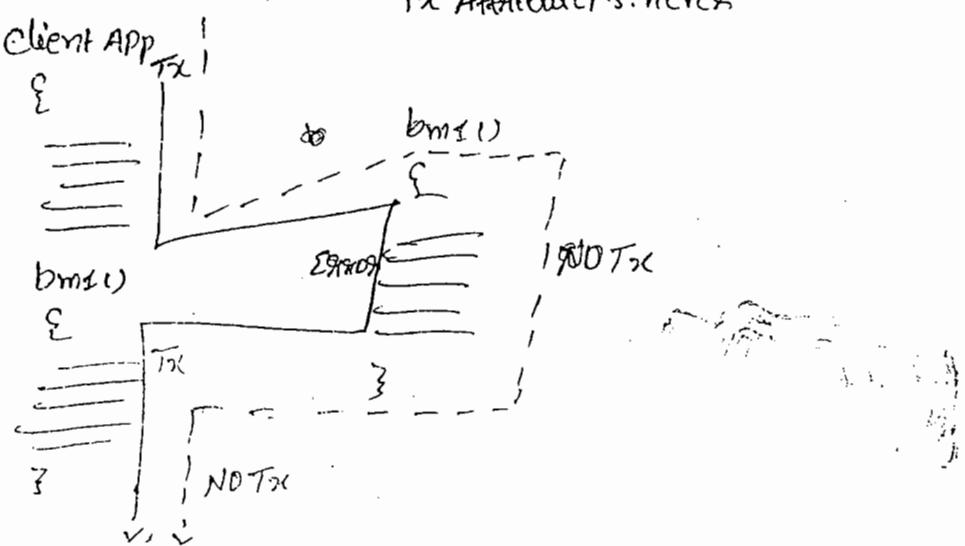
Transaction attribute	clients Transaction	Bean's Transaction
Required	none	$T_2$
	$T_1$	$T_1$
requiresnew	none	$T_2$
	$T_1$	$T_2$
Supports	none	none
	$T_1$	$T_1$
Mandatory	none	throws
	$T_1$	$T_1$
NotSupported	none	none
	$T_1$	none
Never	none	none
	$T_1$	throws

- \*  $T_2$  in Bean's transaction indicates that our B-method runs new  $T_2$ .
- \*  $T_1$  in Bean's transaction indicates that our B-method runs in the client started  $T_1$ .

\* For example app or Local declarative  $T_1$  might on spring DAO persistence logic refer Application(1) of the page nos 130 to 131

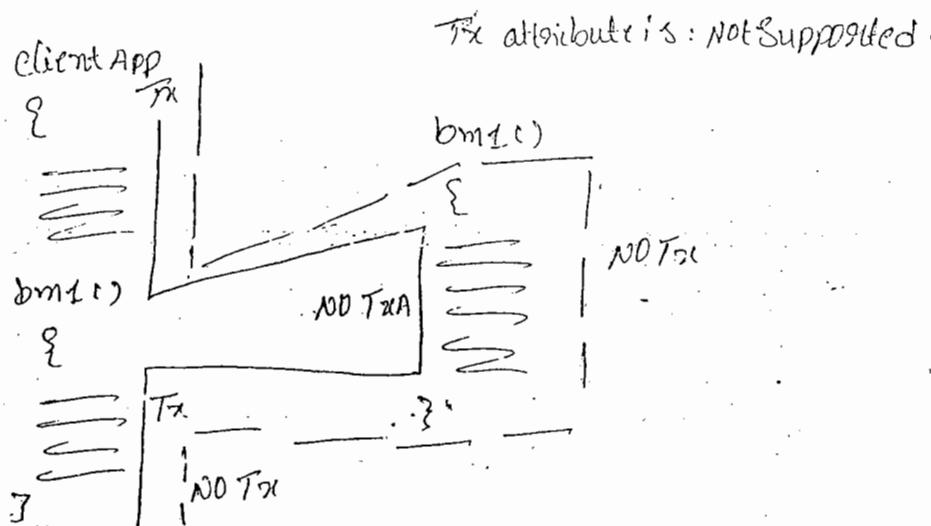
\* In declarative transaction management the transaction manager doesn't rollback transaction with B-method raises checked Exception, but it rollbacks the transaction when B-method raises unchecked Exception (Runtime Exception & its subclasses)

\* To use outer class Java method parameters inside the inner class that is defined in that outer class Java method the outer class parameters should be taken as final:



This indicates client must not call the B-method of Spring Bean with Tx. Otherwise B-method raises the exception.

IT Supposed

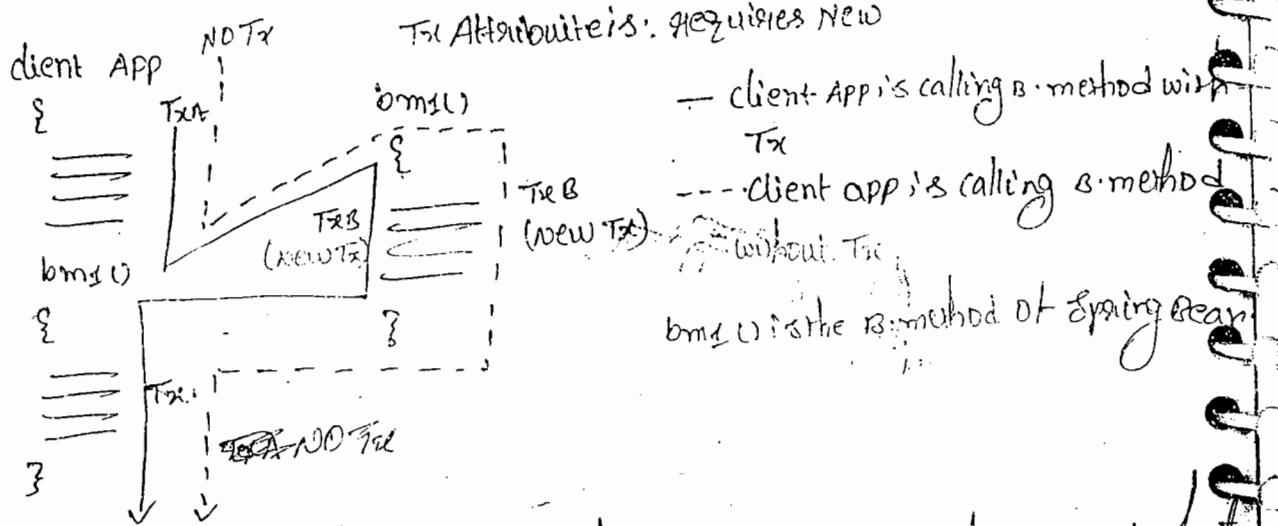


This indicates B-method runs without Tx irrespective of whether client is using B-method with or without Tx.

In declarative transaction the manager rollbacks the transaction with B. method through runtime exception. otherwise it commits the transaction.

### Requires New:

- \* makes B-method always running with new Tx irrespective whether that B-method is called with or without Tx by Client App.



— client App is calling B-method with Tx

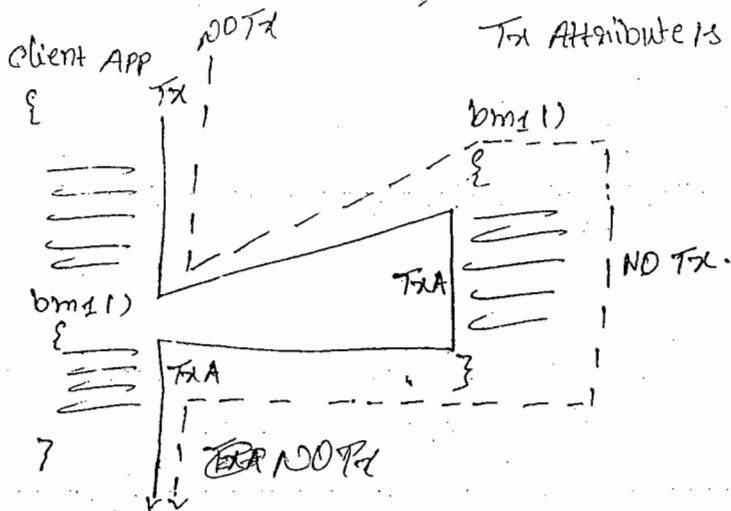
--- client app is calling a method without Tx

bm2() is the B-method of Spring Bean

Supports: - If client calls the B-method with Tx then B-method also runs in the same Tx.

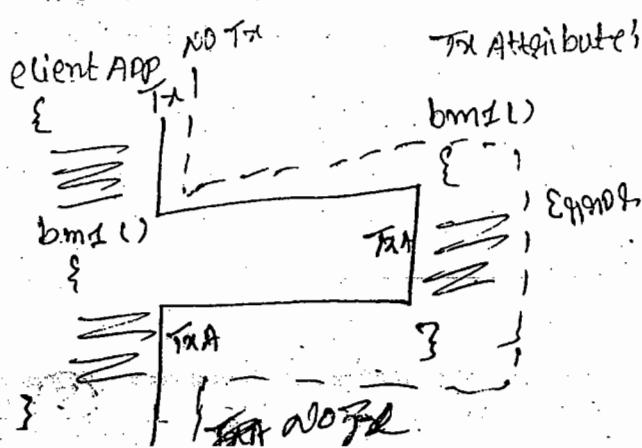
\* If client calls B-method without Tx then B-method also runs without Tx.

### Tx Attribute is: Support



Mandatory: - This indicates the client must call the B-method of Spring Bean with Tx otherwise Error will be raised during the B-method execution.

### Tx Attribute is: mandatory



(24) ←

- never The transaction attribute must be given in a method.
- Supports (default)
- not supported.

For declarative transaction management use org. s.f. transaction.interceptor.

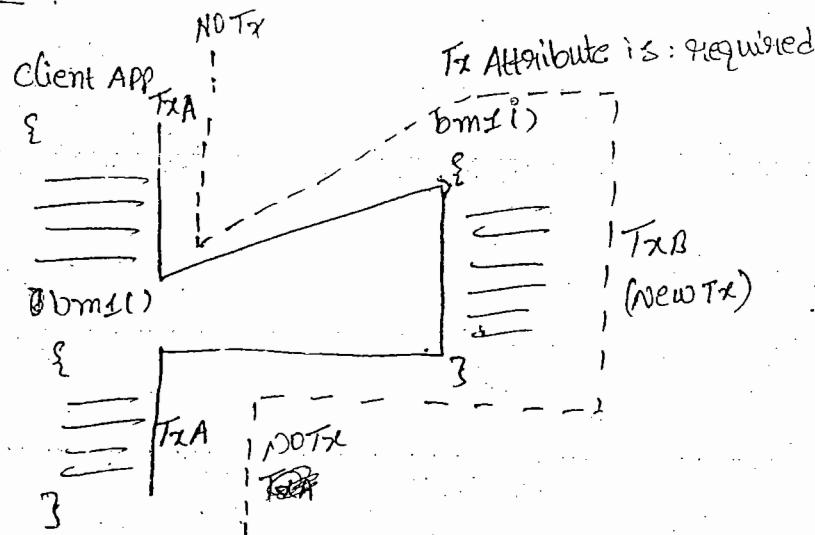
TransactionProxyFactoryBean that gives proxy object by application Tx advice on B-methods.

In cfg linking transaction attributes on B-methods we can use a pointcut advisor named org.s.f.transaction.interceptor.NameMatchAttributeSource.

2/11

Understanding Tx attribute behaviour of declarative Tx mgmt:-

e.g:- In programmatic tx mgmt there is no need of working with tx attribute required;



bmi() is the B-method of Spring Bean.

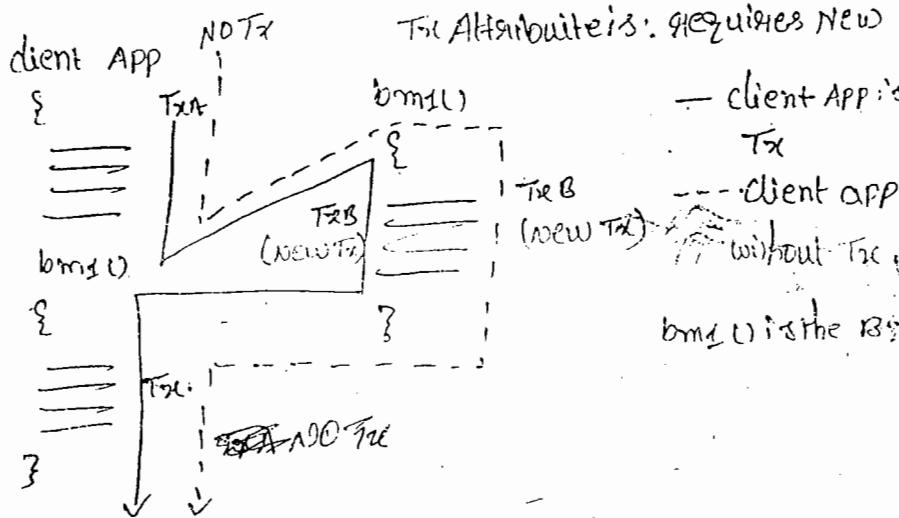
— Client APP is calling B-method with Tx.

--- Client APP is calling B-method without Tx:

If client application calls B-method of Spring Bean with Tx then B-method runs within that tx. otherwise new Tx will be started for B-method

### Requires New

- \* makes B-method always running with new Tx irrespective whether that B-method is called with or without Tx by client APP.



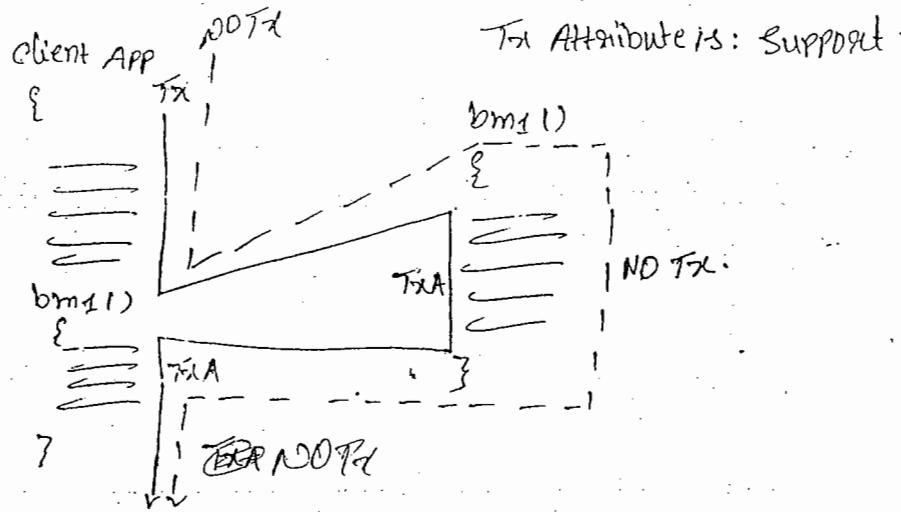
- client APP is calling B-method with Tx

- client APP is calling B-method without Tx

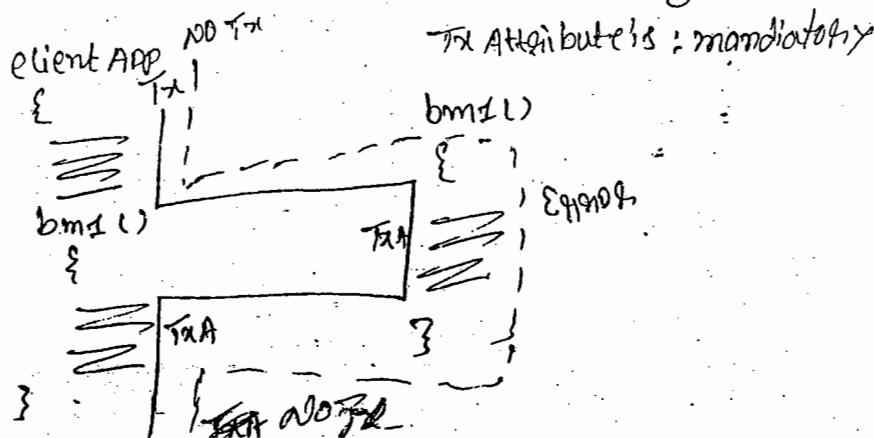
**bmi1** is the B-method of Spring Bean

Supports: - If client calls the B-method with Tx then B-method also runs in the same Tx.

\* If client calls B-method without Tx then B-method also runs without Tx.



Mandatory: - This indicates the client must call the B-method of Spring Bean with Tx otherwise **Exception** will be raised during the B-method execution.

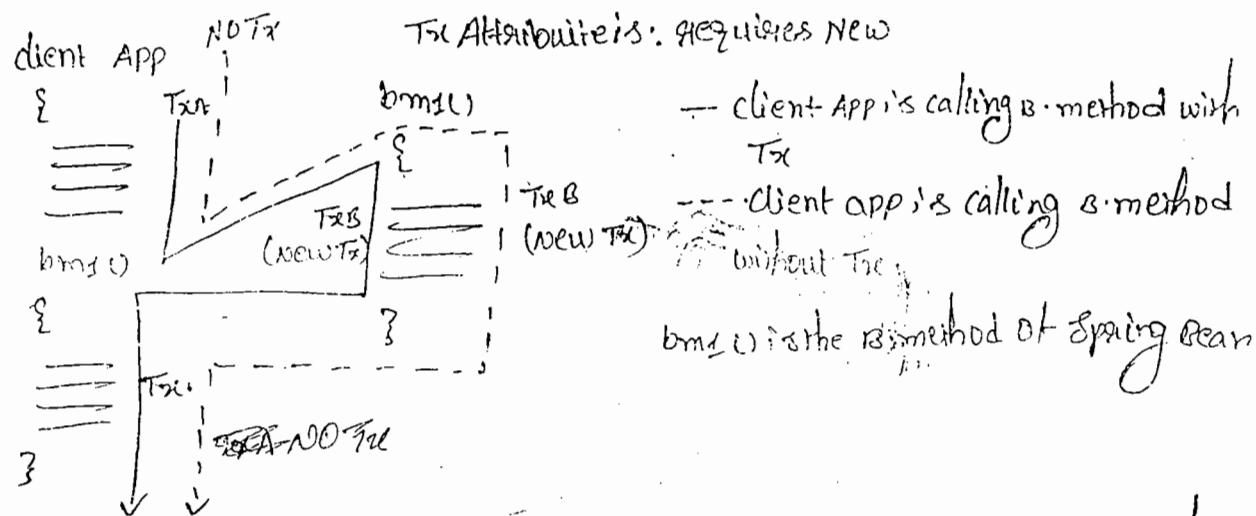


**TxA Attribute is: mandatory**

**Exception**

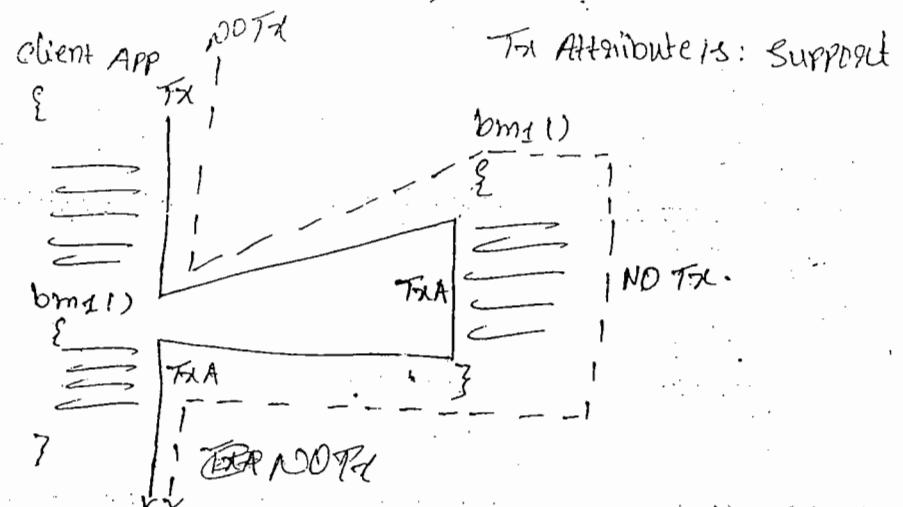
### Requires New

- \* makes B-method always running with new Tx irrespective whether that B-method is called with or without Tx by client APP.

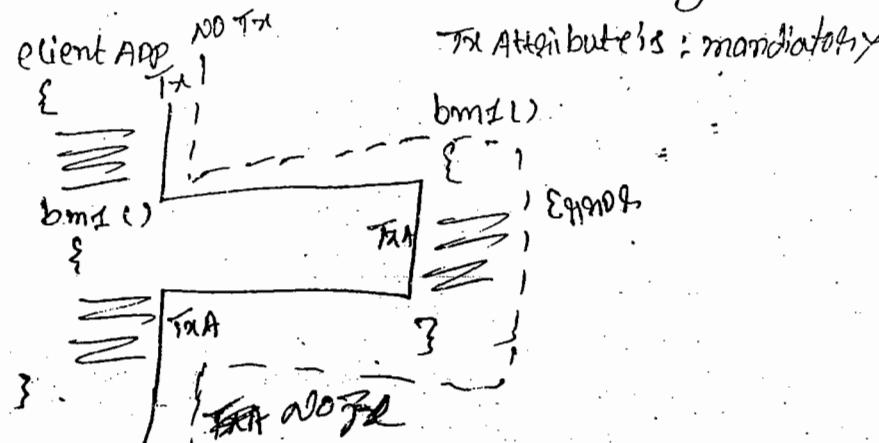


Supports:- If client calls the B-method with Tx then B-method also runs in that same Tx.

\* If client calls B-method without Tx then B-method also runs without Tx.



Mandatory:- This indicates the client must calls the B-method of Spring Bean with Tx otherwise Error will be raised during the B-method execution.



~~EJB, Hibernate~~ transaction attribute that Configured the B-method.  
Nested Tx's.

(ii)

23/12/11

\* Tx.

Spring  
Beans

transaction management use org. s. f. transaction interceptors.  
Proxy Bean that gives proxy object by application Tx

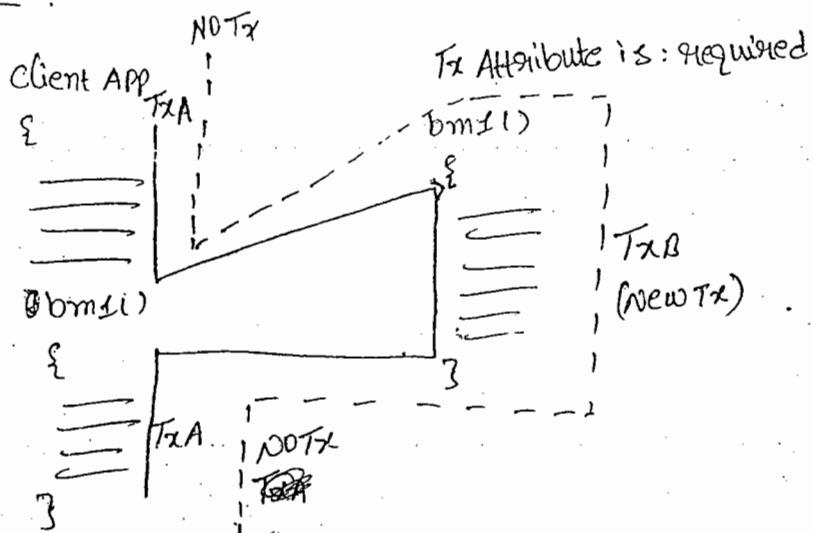
To link transaction attributes on B-methods we can use a pointcut advisor  
called org. s. f. transaction. InterceptorNameMatchTransactionAttributeSource.

24/12/11

Understanding Tx attribute behaviour of declarative Tx project:-

Note:- In programmatic tx mgmt there is no need of working with tx attribute.

Required:-



bms(i) is the B-method of Spring Bean.

— client APP is calling B-method with Tx.

-- client APP is calling B-method without Tx.

If client application calls B-method of Spring Bean with Tx then B-method runs within that tx. otherwise new Tx will be ~~start~~ started for B-method

Compile & execute the client app.

Client > javac -d . HelloClient  
> java HelloClient

Note: Make sure that spring batch domain server of weblogic is in running mode while running the above client app.

\* While developing plain EJB client app we must gather following details from service provider (the EJB Component developer)

1. A copy of Business interface / service interface & Home interface

2. JNDI name of Home obj reference.

3. JNDI properties to interact with registry

4. Documentation about B. methods ..

\* We can develop Spring style EJB client app only for stateless session bean components. For this purpose we need to use "org.raj.ejb.access.SimpleRemoteStatelessSessionProxyFactoryBean" class.

This class generates B. obj reference of EJB component and can inject that reference to specified Bean property of specified spring bean class through dependency injection process.

\* While developing Spring based EJB client app there is no need of keeping Home interface of class side. But remaining all details of plain EJB client app development are required as it is.

\* For Spring based EJB client app refer page no 90 & 91 of the booklet.

Procedure to execute the Spring based client app given in 90 & 91 of booklet

Step 1: Make sure that the plain EJB component is in active mode

1. Develop Spring based EJB client app as shown below.

C:\app\raj\springEjbClient  
    |  
    | Demo.class  
    |  
    | DemoBean.class  
    |  
    | ejb.xml  
    |  
    | DemoClient.class

2. Make sure that following jar files are added to the class path  
weblogic.jar, spring.jar, commons-logging.jar.

Compile & execute Client app

\* We cannot develop Spring based EJB client apps for statefull session bean components, Entity bean components & mdb components.

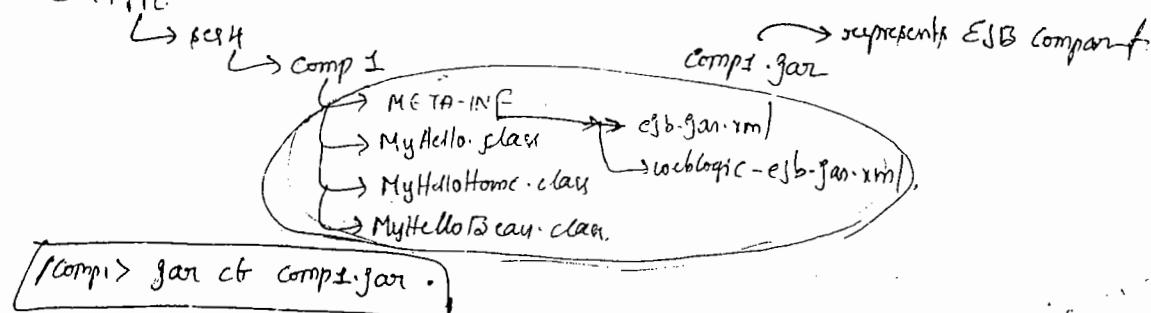
\* To create certain task of app at specific time(s) or repeatedly at regular intervals we need scheduling. This scheduling allows us to enable timer on our app.

- \* Using spring gee module we cannot develop EJB component in spring style - But, because to execute EJB components EJB Container is required to spring container cannot act as EJB container. But, we can develop the client app of EJB component in Spring style by getting abstraction on plain EJB based client app development process
- \* for example app on plain EJB component and plain EJB client app refer page # 88 to 89 app (28)

Procedure to deploy plain EJB 2.X component at prof # 88 & 89 in weblogic 10.3.1

- step 1: Add weblogic.jar to class path (to work with EJB api)
- step 2: prepare jar file on the deployment directory structure of EJB comp representing EJB component.

step 3: E:\AppE



13/11/11

Step 3: Start Spring Batch domain server of weblogic.

start → programs → oracle w.l → user projects → SpringBatch domain → start administrator

Step 4: Deploy the above developed EJB component in springbatch domain server.

Step 5: open browser window → http://localhost:2020/console → deployments → install

Step 6: → upload ur file → deployment archive → browse → select the above created comp1.jar  
→ next → next → next → next → finish → save.

\* For plain EJB client app refer HelloClient.java of page no - 89 & 90 of book left

\* Procedure to execute plain EJB client app i.e., given in 89 & 90 (HelloClient.java).

Step 1: Develop the client app

E:\AppE\src\client\HelloClient.java

Step 2: Copy service interface GIG (MyHello.class), Home Interface GIG (MyHelloHome.class) to client code. (Copy to E:\AppE\src\client folder from comp1 directory of ~~WebX Server~~)

Step 3: Add weblogic.jar to class path

(23) ←

12/11/11

For spring RMI app that uses spring DAO module based persistence logic write app 16 of  
the booklet page no # 64 & 65

- + while developing Spring core module based distributed apps we can use any other spring module.
- (Ex) any other technology except to develop b. logic & persistence logic in Service side implementation class.

### Limitations of RMI

- + language dependent (both server & client apps must be written in java)
- + RMI is not suitable for developing large scale distributed apps.
- + RMI doesn't supply much middle ware services.
- + RMI registry cannot handle huge amount of bind & lookup operations.
- + RMI app cannot use internet n/w as communication channel b/w client & server apps  
(becoz of it internally OS looks for that purpose).

To overcome certain problems of RMI we can use EJB:

### EJB :-

- \* Lang dependent distributed technology
- \* needs EJB container for execution.
- \* Allows to work with more server managed middleware services
- \* Allows to work with messaging n/w
- + Suitable for developing large scale distributed app
- \* Allows to use internet n/w as communication channel.

J

Q: What is the difference b/w JavaBean & EJB?

Ans. JavaBeans

EJB

- |                                                                        |                                                               |
|------------------------------------------------------------------------|---------------------------------------------------------------|
| + Simple java class having getter & setter methods                     | * Distributed technology for distributed apps                 |
| * Needs JVM for execution                                              | * needs JVM + EJB container for execution                     |
| * Light weight resource                                                | * Heavy weight B. component                                   |
| * Useful as helper resource in model layer<br>(DTO, HIBERNATE etc....) | * Can be used in model layer to develop B & persistence logic |

(22) R

- \* The org. st. remoting. Rmi. RmiprxyfactoryBean can inject Rmi B-object Reference to specified bean property of Specified bean class, by gathering that B-object reference from Rmi Registry.
- \* The import Bean properties of this class are -
  - \* url → url required for lookup operation on Rmi Registry to get B-object reference.
  - \* ServiceInterface → name of the ServiceInterface.

III

- \* Post Spring Rmi based distributed application refers the app⑮ of the page no's 62 to 64.
- \* The Spring Rmi application can utilize the already started Rmi registry of at certain port number or can start new Rmi registers explicitly at specified port numbers.
- \* while developing Spring Rmi applications we see the service client needs to gather following details from service provider-
  - A copy of Service interface file.
  - nick name or alias name of B-object reference.
  - host name/ IP address and port number of Rmi Registry
  - Documentation about B-methods.

- plain Rmi Server app → ① → yes ← plain Rmi Client app.
- plain Rmi Server app → ② → yes ← Spring Rmi Client app
- Spring Rmi Server app → ③ → yes ← Spring Rmi Client App
- Spring Rmi Server app → ④ → Not ← plain Rmi Client APP.
- \* The last combination is not possible because Spring Rmi Server application does not generates stub file explicitly but the plain Rmi Client application expects the stub file at Client side.

- \* while developing second combination the plain Rmi Server application stub file should be copied to the Spring Rmi Client application.

### ⑤ Documentation about B-methods.

#### Spring Rmi:

- \* Spring Rmi provides abstraction layer on plain Rmi and allows us to develop Spring style Rmi based distributed applications.
- \* The following benefits are there with Spring Rmi
  - ⇒ no need of working with plain Rmi api.
  - ⇒ Service interface, and the implementation class that contains B-methods can be developed as POJO & POJZ classes.
  - ⇒ No need of generating stub file explicitly.
  - ⇒ Starts / Locates Rmi Registry internally (No need of working with Rmi Registry Explicitly)
  - ⇒ ~~No need of port~~,  
Exception handling is optional because Spring Rmi throws unchecked exceptions.

Note:- Spring Rmi internally catches the plain Rmi generated checked exceptions and rethrows them as unchecked exceptions.

⇒ The Server & Client Applications can perform binding operation, lookup operations on Rmi Registry ~~directly~~ through IOC or dependency injection process.

\* The org.springframework.rmi.RmiServiceExporter can registers given B-object with Rmi Registry having nickname by starting Rmi Registry & w, generating Stub file.

\* The important Bean properties are

- \* service → B-object beanid.
- \* serviceInterface → name of the Service Interface / B-interface.
- \* servicePort → port number of Rmi Registry.
- \* serviceName → nick name B-object.
- \* registerPort → port numbers of Rmi Registry
- \* registryHost → host name or IP address of machine where Rmi Registry resides.

In the above application client & Server applications will execute on two different JVMs by taking Rmi Registry as mediator b/w client & server app.

- If multiple Java applications are executed ~~simultaneously~~ from a single computer then multiple JVMs will be launched in that computer simultaneously.

The process of converting Java notation data to network notation data is called as marshalling and reverse is called as unmarshalling.

In RMI application we need to place stub file at client side & server side, the stub file of client side performs marshalling and unmarshalling applications for client application.

The stub file of stub file marshalling & unmarshalling on application in server application.

In RMI application the stub file of client side acts as proxy for client application & the stub file of server file acts as proxy for server app.

On behalf RMI client application and RMI server application their stub files actually participates in communication and gives the ~~feeling~~ feeling i.e RMI client & server applications are integrating with ~~each other~~ each other.

~~new change~~ we can change the port number of RMI Registry as shown below

> start Rmiregistry 2222 new port number.

As a service provider who develops RMI client application from remote place, we must gather following details from service provider who develops server application.

D A copy of stub file.

D A copy of Service interface file.

> IP address/ host name and port number details RMI Registry.

The nick name assigned to B object reference.

10/11/11

### II FacClientApp.java

```
import java.rmi.*;  
public class FacClientAPP  
{  
    public static void main(String args[]) throws Exception  
    {  
        // get B-object from Rmi Registry through lookup operation  
        // (fall)  
        BObject bobject = Naming.lookup("rmi://localhost:1099/india");  
        // call B-method on B-object reference.  
        Long result = bobject.findFactorial(8);  
        System.out.println("The factorial value is: " + result);  
    }  
}
```

### Steps for Executions:-

Step①:- Compile resources of Server Side.

E:\APP\ses1\sever>javac \*.java

Step②:- generate stub file based on implementation class.

E:\APP\ses1\sever>rmiic -Rimpl

—> gives Rimpl-Stub.classfile.

Step③:- Start rmi registry.

E:\APP\ses1\sever>start ~~rmiregistry~~ rmiregistry

↳ to start rmiregistry service in a new window

Step④:- Run the server application

E:\APP\ses1\sever>java RattServerApp

Step⑤:- copy the service interface(fall.java) Stubfile (Rimpl-Stub.class) files to client folder from sever folder.

Step⑥:- Compile the resources of client side

E:\APP\ses1\client>javac \*.java

Step⑦:- Run the client application.

(KK)

1) Implement B methods.

public Long findFactorial (int val) throws RemoteException.

{

Long res = 1;

for (int i=1; i<=val; ++i)  
res = res \* i;

return res;

}

2) PartServerApp.java:

public class PartServerApp

{

P.S.W. main (String args[]) throws Exception

{

1) Create obj for Impl class (B obj)

PartImpl bobj = new PartImpl();

1) Bind bobj reference with Rmi Registry.

Naming.bind ("rmi://localhost:1099/India", bobj);

Protocol      Hostname & port no of rmi registry

Name      Obj to be bound.

System.out.println ("Bobj is bound and server app is ready");

}

}

Note:- "Rmi" is application level protocol that can be used by client to interact with rmi registry to perform bind, rebind, lookup and etc operations.

Ans: The methods of Local object can be invoked only from local clients whereas the methods of Remote object can be invoked from both local & remote clients.

- \* When Java class implements a marker interface called `java.rmi.Remote` then all the objects of that class acts as remote objects.
- \* The interface that gives special runtime capabilities to the object of its implementation class is called as ~~marker~~ interface.

Ex: `java.io.Serializable`.

`java.rmi.Remote`.

`java.lang.Cloneable`

`java.lang.Runnable` and etc.

- \* A marker interface can be there-with or without methods.

#### FAC.java

```
import java.rmi.*;  
public interface Fac extends Remote  
{  
    public Long findFactorial(int val) throws RemoteException;  
}
```

declaration of B method.

#### II factServerApp.java

```
import java.rmi.*;  
import java.rmi.server.*; // allows the objects of its subclasses  
// registerable with Rmiregistry.  
class FactImpl extends UnicastRemoteObject implements Fac  
{  
    UnicastRemoteObject  
    public FactImpl() throws RemoteException  
    {  
        System.out.println("No-arg Constructor of FactImpl class");  
    }  
}
```

/\* If Superclass Constructor throws Exception the Sub class Constructor can not catch and handle them using try, catch statements so, the Sub class Constructor should declare that exception to be thrown using throws statement \*/

- \* In the registry SW's of distributed application multiple B. Component references will be placed and that registry SW will reside in fixed location.
- \* RMI, EJB, CORBA can be used to develop non web based distributed application.
- \* http invokers and webservices can be used to develop web based distributed applications.
- \* In distributed application development four important practices will be involved.

Service provider → develops server application having B. Comp/B. obj.  
                           → Binds B. obj ref with Registry SW.

Service clients → develops client application calling B. methods.  
                           → gets B. obj ref from Registry to lookup operation

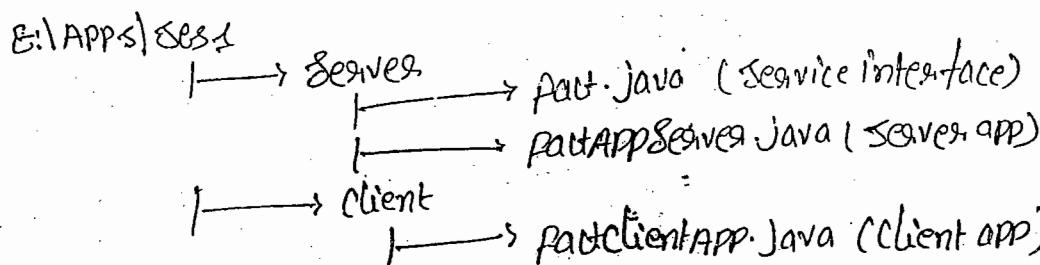
Service interface → understanding document b/w Service provider & Service Client having declaration of B. methods.  
                           → It is an Java interface.

Registry SW → manages B. object ref having global visibility.  
                           → makes distributed applications as Location transparency app.

### plain RMI

- \* Built-in technology in JDK SW.
- \* uses RMI registry as registry SW. (default port No: 1099)
- \* The core RMI API is Java.rmi package & Java.rmi.server package.
- \* Basic JDK SW is enough to develop RMI applications.

### plain RMI application

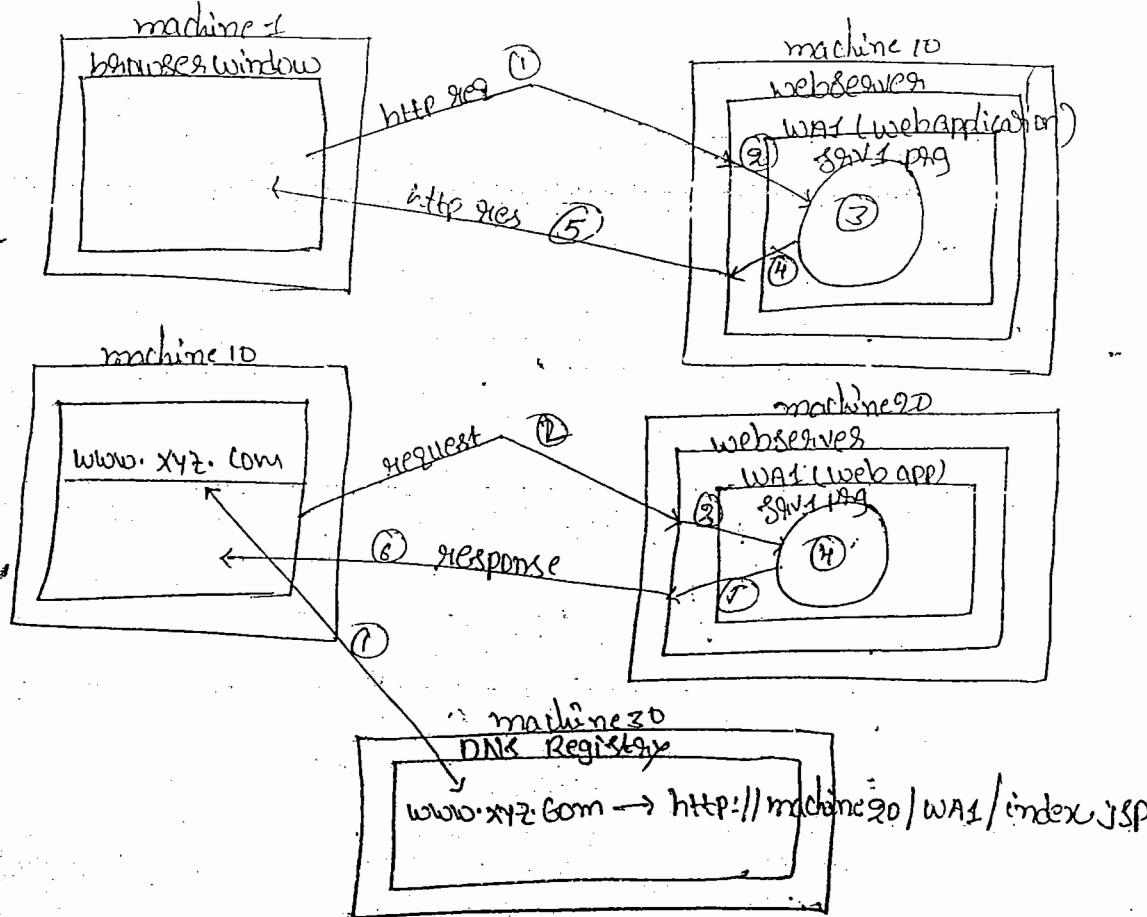


- \* In RMI application the B. objects of servers application should be developed as Remote Objects.

- ④ Client application calls B-method on B-object reference
- ⑤ The B-method of B-object available in Server application executes.
- ⑥ The results generated by B-method goes to Client Application

09/11/11

- \* The client Server application that gives Location Transparency is called as distributed application, RMI, EJB, CORBA, webservices are the distributed technologies to develop distributed applications.
- \* A distributed application can be developed either as web application or Non-webapp
- \* The web application with DNS Registry support are called as distributed app.
- \* A webapplication without DNS (Domain Naming Services) Support is called as Client Server application.
- \* All web sites that hosted on the internet network are called as distributed app
- \* All web applications that are developed at classroom level (NO DNS Registry Support) is called as client server application.



web application or distributed application

116 ←

// write b.logic

int iamt = (pamt \* state \* time) / 100;

// write the persistence logic DAO class

HbCustomer hc = new HbCustomer();

~~hc.setPamt~~

hc.setName(name);

hc.setPamt(~~new Integer(pamt)~~);

hc.setIntamt(~~new Integer(amt)~~);

dao.save(hc);

// return total amt

return (pamt + iamt);

3 // method.

}

Step 10: - Configure the above created SpringBean class in SpringH3.xml file.

```
<bean id="mb" class="moddBean">
    <property name="dao" ref="HbCustomerDAO"/>
</bean>
```

\* injects the IDE generated DAO class object to our SpringBean class project.

Step 11: - Since incantment algorithm can not generate object type identifiers values change the 'cno' property of HbCustomer.java from java.lang.Integer to simple "int".

Step 12: - Configure tomcat 6.0 server with myEclipse IDE.

Window menu → preferences → servers → tomcat : tomcat 6.0

① enable : Browse and select home directory of tomcat

08/11/11

Step 12: - Add the form page input.html to the webroot folder of the project.

Right click on webroot folder → new → html : input.html

use pattern of Controllers Design program

```
<form action="Controller1" method="get">
```

```
    <b> Name : <input type="text" name="tname" /> <br>
```

```
    <b> Pamt : <input type="text" name="tpamt" /> <br>
```

Step@: Perform HB Reverse Engineering on HB-Customer table of the above created OraP DB profile.

go to DB Browser window → RC on OraP profile → open Connection → Expand OraP → Expand Connected to OraP → Select → table → HB-Customer  
RC on HB-Customer table → hibernate reverse engineering → java source folder: /SpringHBAPP3/src

{  
    if Create POJO ---  
    if Java DAO Object ---  
    if Delete Create abstract class  
    if Java Data Access Object  
        DAO type: ① Spring DAO  
        Spring config file: src/springHB.xml  
        Session factory ID: SessionFactory  
    }  
→ Next

Type mapping: ① HB Types } → Next → finish  
ID generator: Increment }

\* The above step generates mapping file (HBCustomer.hbm.xml) POJO classes (HBCustomer.java) and DAO class (HBCustomerDAO.java)

Note:- Add connection.autocommit property in hibernate.cfg.xml file.  
<property name="connection.autocommit"> true </property>

Step@: Add Spring resources for the project (Model.java, modulBean.java)  
modul.java:- RC src → new → Interface → modul.java

public interface modul  
{

    public int calcIntRate (String cname, int point, int time, float rate);

}

RC on src → new → Class → modulBean.java → implements → (modul)

modulBean.java,

public class modulBean implements modul,

{

    HBCustomerDAO dao;

    public void setHBCustomerDAO(HBCustomerDAO dao)

    { this.dao = dao;

}

    public int calcIntRate (String cname, int point, int time, float rate)

{

Step ①: Create web project in myeclipse IDE having name SpringHBAPPJ.

File menu → New → webproject → project name: SpringHBAPPJ → Finish.

Step ②: Add HB capabilities to the project.

R.C on project → myeclipse → Add HB capabilities → hibernate3.20  
→ next → next → Datasource → use JDBC driver → DB Driver: Oracle,  
→ next →  deselect checkbox → finish → Add → property: Show SQL  
Value: True → Save.

Step ③: Add Spring capabilities to the project.

R.C on project → myeclipse → add Spring capabilities → ~~hibernate~~  
~~Spring 2.5 core libraries~~ & ~~Spring 2.5 persistence core libraries~~  
~~Hibernate 3.2 core libraries~~  
Annotations } → next →  deselect Enable HQL builder →  
file: SpringHB.XML → next → restart → finish.

becomes bean id of LocalSessionFactory  
bean Configuration.

Step ④: Add Apache DBCP related jar file to classpath / build path of the project.  
(tomcat-dbcp.jar)

R.C on project → buildpath → add External Archives → Browse and  
Select tomcat-home/lib → tomcat-dbcp.jar → OK.

Step ⑤: Configure the apache dbcp related BasicDataSource class in Spring HB.XML file.

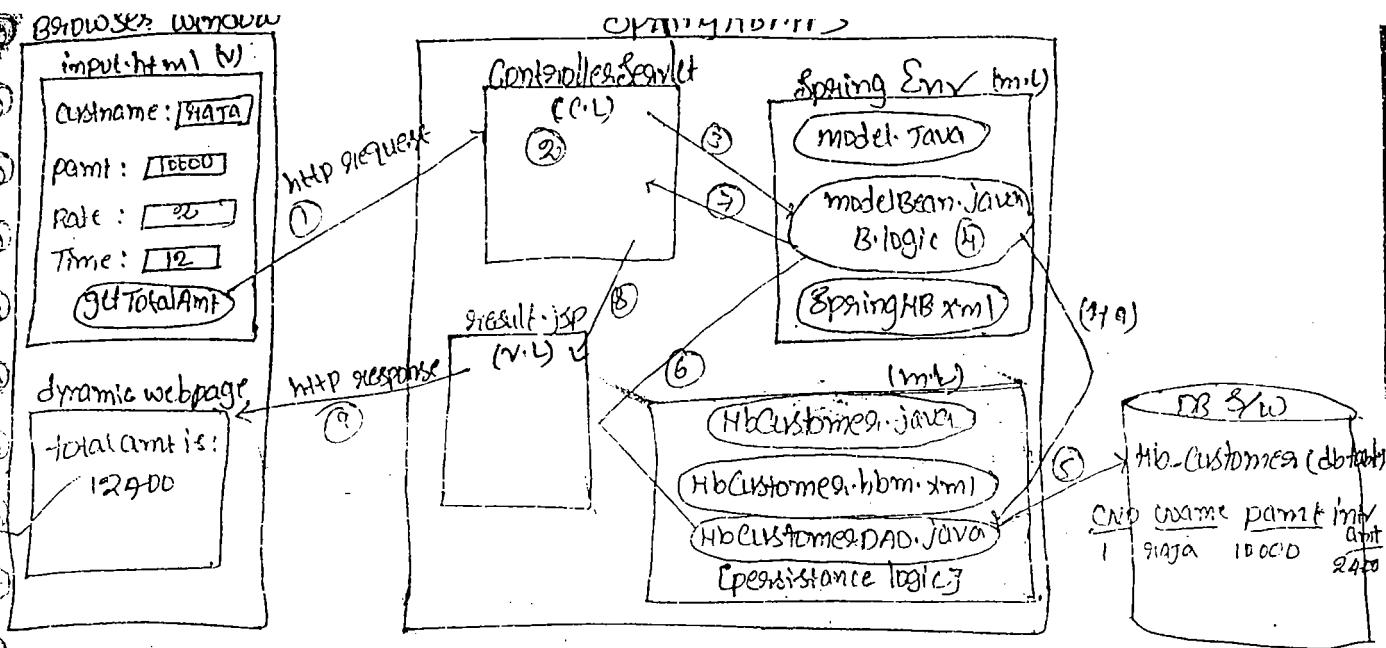
In SpringHB.XML:

```
<bean id="dbcp" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    <p name="url" value="jdbc:oracle:thin:@localhost:1521:ORCL"/>
    <p name="username" value="scott"/>
    <p name="password" value="tiger"/>
</bean>
```

Inject the above bean generated datasource object to datasource property of  
LocalSessionFactoryBean cfg in SpringHB.XML file.

```
<bean>
    <property name="datasource" ref="dbcp"/>

```



In the above diagram B-logic is calculating the ~~interest~~ interest amount and the persistence logic is inserting customer details.

07/11/11

licence key of myeclipse :-

Subscriber: Satya Technologies

Subscription code: FLR8ZG-855-55-6966505931894727

\* procedure to develop the above diagram based application by using myEclipse IDE:-

Step 0:- Create DB profile from Oracle in myeclipse IDE:

Window menu → open perspective → myeclipsr. DB Explorer → DB because Window → Rightclick → new → driver template: Oracle thin drivers,

Driver name: 09040 → user: scott → Jdbc Oracle: thin @ localhost: 1521: ORCL

User name: scott password: tigress → Add JARs → Browse & select DJdbc4JAR → Finish → Save password → Next → Finish.

Step 1:- Create Hb-Customers table in Oracle DB having primary key Constraint column.

SQL > create table Hb-Customers (cno number(5) primary key, cname varchar2(15) pamt number(7), interestRate number(7));

(10) ←

Executing native SQL query directly using HBTemplate & hibernateCallback (2)

Step 1:- In gital() method of demoimpl.java

public Iterator getdata()

{ List

List l = ht.execute(new HibernateCallback(2))

{

public Object doInHibernate(Session ses) throws HibernateException,  
DataException:

{|| executing native SQL query directly

SQLQuery q = ses.createSQLQuery("select \* from users"),

q.addEntity(User.class),

List l = q.list(),

return l;

} || method

},

Iterator it = l.iterator();

return it;

} getdata()

\* Here ht.execute() method is called having object of ~~some~~ anonymous inner class. That anonymous inner class is implementing a ~~new~~ callback interface called org.sf.orm.hibernate3.HibernateCallback(2)

\* my Eclipse IDE is capable of generating hibernate based Spring ORM module style persistence logic based DAO class dynamically through HB Reverse Engineering but that project must be enabled with Spring, hibernate capabilities.

\* Example application:-

- \* while working with plain HIB we need to perform transaction management while executing non-select SQL queries.
- \* while working with HIB template class there is no need of performing Transaction management towards non-select queries execution.

### In getdata()

|| to insert record.                            || to delete the record

```
User u1 = new User();
u1.setId(157);
u1.setUname("Ravi");
u1.setRole("t.i.u");
ht.save(u1);
```

```
User u1 = new User();
u1.setId(157);
ht.delete(u1);
```

- \* performing bulk non-select operation using HQL non-select query having POSITIONAL PARAMETERS:

```
int res = ht.bulkUpdate("update User u1 set u1.role=? where u1.id=?",
    new Object[] {"p1", new Integer(200)},
    s.b.plm("no. of records that are effected:" + res),
```

Operations that are not different not possible using HibernateTemplate class obj:

- \* native SQL queries can not executed directly without making them as named queries.
- \* HQL non-select bulk operations are not possible with named parameters.
- \* native SQL based non-select operations are not possible.
- \* Non-select HQL, native SQL queries can not execute as named queries.

### Note:-

To perform all these operations in HibernateTemplate class use HibernateCallback interface of Spring ORM module.

- \* while working with hibernate callback interface implementation the doInHibernate(-,-) method of that callback interface exposes us the underlying hibernate related hibernate Session object using that object we can ~~not~~ develop our choice code.

Executing named native SQL select query having named parameters :-

P ①:- In user.hbm.xml:-

```
<sql-query name="myq2">
    <return class="User"/>
    <!CDATA[select * from users where userid = :P1 and
    userid <= :P2 ]>
```

P ②:- In getdata() method of demoimpl.java .

```
List l = ht.findByNameQueryAndNamedParam ("myq2", new String[] {
    "P1", "P2"}, new Object[] {new Integer(100), new Integer(200)});
```

Using PL/SQL procedure of oracle using HibernateTemplate class obj:-

Q ①:- write PL/SQL procedure in oracle as required - for HB (follow rules)

PL/SQL Create or replace procedure getEmpDetails (mycur OUT SYS.REFCURSOR,  
desg in varchar)

AS

BEGIN

```
OPEN mycur FOR
    Select * from users where role = desg;
```

END;

execute in SQL prompt of oracle .

Q ②:- call the above PL/SQL procedure from HB mapping file as named native SQL query.

```
user.hbm.xml:<sql-query name="myq2" callable="true">
    <return class="User"/>
    {call getEmpDetails (?,?)}
```

```
</sql-query>
```

P ③:- write following code in getdata() method of demoimpl class

```
new Object[] {"b1"});
```

\* HibernateTemplate class provides abstraction layer on plain hibernate and also converts the underlying hibernate's generated checked exceptions into un-checked exception.

\* executing HQL Select query with position parameters -

In getdata() method of demopl class:-

```
public Iterator getdata()
```

```
{
```

HQL query with position parameters.

```
List l = ht.find("from User u where u.uid=? and u.uid<?",  
new Object[]{new Integer(100), new Integer(200)});
```

Iterator it = l.iterator(); Java.lang.Object class obj[] supplying argument values.  
return it;

executing HQL query with namedparameters:-

\* List l = ht.findByNameParam("from User u where u.uid >=:p1 and u.uid <=:p2",  
new String[]{"p1", "p2"}, new Object[]{new Integer(100), new Integer(125)});  
Iterator it = l.iterator();  
return it;

\* Executing named nativeSQL query with position parameters:-

Step 1:- prepare named nativeSQL query in hibernate mapping file.

In User.hbm.xml:-

```
<sql-query class="User">  
mapping SQL <sql-query name="mypy">  
query selects with return class="User"/> datatable name. column name.  
HB POJO classes (User) <sql-query name="mypy" /> <select> * from users where user_id=? and user_id=?  
<!-- COALESCE user_id, user_id -->  
</sql-query>
```

Step 1:- write the following code in the getdata() method of demopl class

```
List l = ht.findByNameQuery("mypy", new Object[]{new Integer(100),  
new Integer(125)});
```

Step 4: Configure our Spring bean class in Spring config file Injecting HibernateTemplate obj in Spring config file.

```
<bean id="dl" class=" demo9impl">  
    <property name="ht" ref="template" />
```

The bean id of HibernateTemplate class which is configured above.

Step 5: Write spring style HB persistence logic in our Spring Bean class by using the injected HibernateTemplate class obj (ht).

Ex: public class Demo9impl implements DemoInter  
{  
 HibernateTemplate ht;  
 public void setHT(HibernateTemplate ht)  
 {  
 this.ht = ht;  
 }  
 public void bmt()  
 {  
 // use the methods of HT class to write HB persistence logic  
 }  
}

Step 6: Develop the remaining steps of app in regular manner.

Understanding Various methods of the predefined HibernateTemplate (c) for persistence

save(-), persist(-) for insertion of record

update(-) to update the record

SaveOrUpdate(-), merge(-) to insert / update

load(-), get(-) to select the record

delete(-) to delete a record.

find(<>) → to execute HQL Queries or Select

getNamedQuery(<->) → to execute hql select queries

findByXXX(<->) → to execute native sql queries

findByCriteria(<->) → to execute logic of criteria API.

bulkUpdate() → to execute non-select hql queries

The call back interface in HTTemplate class environment is "org.hibernate.Callbacks".

HibernateCallback"

④ For example app on approach no 2 based Spring with HB app refer the handout given on November - 04 - 2011 (today).

```

public class DemoImpl implements DemoInter
{
    Sessionfactory factory;
    public void selffactory(Sessionfactory factory)
    {
        this.factory = factory;
    }
    public void bmic() // B.method
    {
        try
        {
            Session rep = factory.openSession();
            // use HS session obj to write HB persistence logic
        }
        catch (Exception e)
        {
            // bmic()
        }
    }
}

```

Step 5: Develop the remaining properties of the app in regular manner as required for the app.

For above 2 steps & diagram based Spooling with HB app refer page # 70 app # 21 (Approach-1)

OH|U|H Seftwark

- \* If business component contains persistence logic as main logic of the app then the persistence logic itself acts as the business logic of the app. (method `getDatas()` business method of `page1`)

The limitation with approach #1 based Spring with HS app we still need to use HS app in spring bean class to develop hibernate persistence logic.

Approach no #2

- \* Procedure to develop Spring with HB app based on approach B. (by injecting ~~the~~)  
Hibernate(Persistent class obj)

Steps 1 & 2 are same as approach no 1

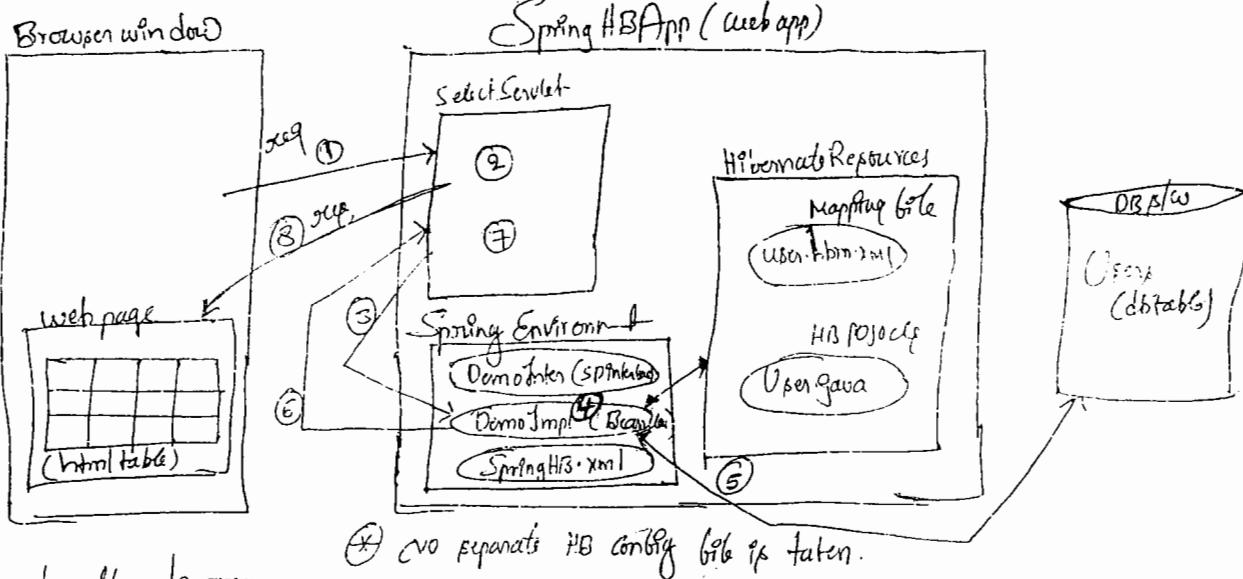
~~Step 3~~ - config Use the local SessionFactoryBean generated HIB SessionFactory obj as base obj and configure org.springframework.hibernate.HibernateTemplate class in spring config file.

8th Sample code: In Spring config file

```
<bean id="template" class="org.hibernate.HibernateTemplate">
    <property name="sessionfactory" ref="SessionFactory"/> // Inject HB Sessionfactory obj.
</bean>
```

- \* If you are developing spring ORM module app based on the existing resources db-hibernate then go for example 2 style of Spring config file

Fightback



W.r.t to the diagram

- ① Browser window gives blank request to Select Servlet.
- ② Select Servlet takes the request, activates the Spring container, gets Spring bean class obj.
- ③ Select Servlet calls B method of Spring bean class.
- ④ The Spring Bean class uses either approach 1 or 2 to develop HIS persistence logic based on HIS resources.
- ⑤ This persistence logic of Spring bean class B method will interact with db pl/w & gets records from the table.
- ⑥ The B method of Spring bean class sends the result to the Servlet prog.
- ⑦ ⑧ The servlet prog formates the result by using presentation logic and sends the result to browser window as web page containing html table.

Procedure to develop the above diagram based app (Spring with HIS) by using approach 1

Steps: Config bean that gives Jdb DataSource obj

Step2: Config org.hibernate.SessionFactoryBean that gives hb SessionFactory obj

Step3: Config own Spring Bean class (DemoImpl) injecting HIS SessionFactory obj given by the above LocalSessionfactory Bean

(for sample code refer example 2 Spring config in the previous page once)

Step4: Create HIS Session obj in the B.method of Spring bean based on the injected HIS Session Factory obj.

```

<bean id="perfact" class="org.hibernate.SessionFactory">
    <property name="DataSource" ref="dsrdbf"/> 
    </bean>
    <bean id="db" class="DemoBean">
        <property name="factory" ref="perfact"/> // our Bean class obj will be injected with
    
```

HB config file is specified here.

```

        </bean>
    </beans>

```

the local Sessionfactory bean generated  
HB Sessionfactory obj.

Note: Here Spring config & HB config file are two different xml file.

### Approach 2 - Example 2:

```

<beans>
    <bean id="dsrdbf" class="org.apache.tomcat.dbcp.dbcp.BasicDataSource">
        <!--
        </bean>
        <bean id="perfact" class="org.hibernate.LocalSessionFactoryBean">
            <property name="dataSource" ref="dsrdbf"/>
            <property name="mappingResource">
                <list>
                    <value>Employee.hbm.xml</value> // hb mapping file
                </list>
            </property>
            <property name="hibernateProperties">
                <props>
                    <prop key="hibernate.dialect">org.hibernate.oracle.OracleDialect</prop>
                    <prop key="show_sql">true</prop>
                </props>
            </property>
        </bean>
        <bean id="db" class="DemoBean">
            <property name="factory" ref="perfact"/>
        </bean>
    </beans>

```

\* Here separate hb config file is not required bcoz the spring config file itself containing hibernate config details.

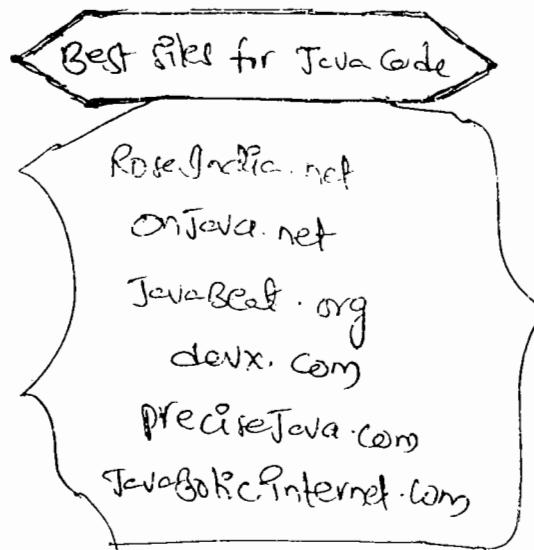
\* HB persistence class(pojo), hb mapping file, hbconfig file are called as hb resources

\* If Spring app is developed from scratch level then go for example 2 style Spring config file.

→ This page will read from this page to books

These are the gaps of

$$\begin{array}{l} 03/11 \rightarrow 07/11 \\ 09/11 \rightarrow 13/11 \\ 03/12 \rightarrow 07/12 \end{array} \quad \left. \right\} \text{p}$$



03/11/11 Spring with Hibernate App

- \* The two approaches of developing Spring with HB app

### I. Approach 1

- [Change HB Sessionfactory obj to Spring Bean]

- + Here we need to use plain hibernate in Spring Bean class to develop persistence logic

## d. Approach P

- C object Hibernate Template class obj in Spring(Backend)

- \* Here there is no need of using plain HB API in Splicing Beam class to develop persistence logic

- ④ Approach II is recommended to use

- To work with approach no 1 we use "org.hibernate.LocalSessionFactoryBean" class which is capable of generating and injecting HIB Sessionfactory obj to the specified property of Spring bean class.

- \* While configuring this class Data Source obj is mandatory and pf. config location property is specified then we can work with separate hibernate configuration file. Otherwise the details of HIB config file must be specified in Spring config file it self while configuring this class.

Example 1 (In Springy case g(6))

```
<beans>
    <bean id="dataSource" class="org.apache.tomcat.dbcp.dbcp.BasicDataSource">
        <!-->
    </bean>
</beans>
```