25/08/11

Springs

Framework is a special s/w, 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 logics development. This increases productivity in application development.

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

web flw s/w's :-

To develop mvc II architecture based webapplications.

Ex: struts, webwork, jsf, spring mvc ....

These are given based on servlet, jsp core technologies.

orm flw s/w's :-

Hibernate, iBatis, Toplinks, JDO, OJB .....

These use JDBC as underlying core technology.

orm flw s/w based persistance logic db s/w independent.

JDBC persistance logic is db dependent.

Java-Jee flw s/w's :-

Spring

(Allows) provides environment to develop all Java, Jee applications in frameworks style.

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, Adter apps, etc

Plain Jdbc application :-

1. Register driver with DriverManager service.
2. Establish connection with db s/w.
3. Create Statement object.

→ Common logics

4. Prepare sql query, send and execute sql query in db.
5. Gather results and process results.

→ app specific logic

6. perform exception Handling.
7. perform Transaction Management.
8. close Jdbc object (especially connection object)

→ common logics

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 s/w
3. Gather results and process results.

Note : In spring Jdbc application the JdbcTemplate will take care of all the common logics generation, and makes programer 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 programer 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.

Diff b/w Struts and spring

Struts	spring
1. It is a web flw slw to develop mvc 2 arch web apps.	1. It is a Java-Jee flw slw to develop any kind of apps including web
2. we must need web/application server to execute struts apps	2. Some spring apps can be executed without web/application servers.
3. Struts app resources are struts API dependent.	3. spring app resources are spring API independent.
4. Allows only Jsp programs for view layer.	4. In spring based web apps Velocity, freemarker and Jsp programs can be taken for the view layer.
5. Doesnt provide built-in middleware services.	5. provides builtin middleware services.
6. Runs on MVC2 principles.	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 logics 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 (16+ middleware services are there)...

spring:

type : Java-Jee fw sfw to develop any Java, Jee applications

version : 2.5 (compatable with Jdk 1.5/1.6), 3.x (compatable with Java 1.6 + ...)

vendor : Interface 21

creator : Mr. Rod Johnson

It is open source sfw, download sfw as zip file from www.springframework.org website

for good online tutorial www.springframework.org
www.roseindia.net.

Reference book : spring live, spring in action, ...

The seven modules of spring 1.x :-

1. spring core (base for all other)
2. spring dao
3. spring orm
4. spring web
5. spring web mvc
6. spring AOP

Spring 2.x 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 programing)

* AOP is nowhere related with OOP. (Object oriented programing)
  → OOP is the methodology of creating programing 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 meterial from Institute by requesting for it.
(meterial ↳ dependent value; requesting ↳ gathering explicitly.)

If underlying container fw or tfw s/w or server s/w is dynamically assigning values to resource/application then it is called as dependency Injection/ Inversion of control.

In dependency injection the underlying fw slw or server slw 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 slw or slw 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 (class) life cycle)

In spring we have two built in containers,

There are 1. BeanFactory (part of spring core module)
2. ApplicationContext (part of spring jee/context module)

Spring containers are light weight containers.

Servlet Container, Jsp container and EJB containers are slw applications of web server, application server slw so they are heavy weight containers.

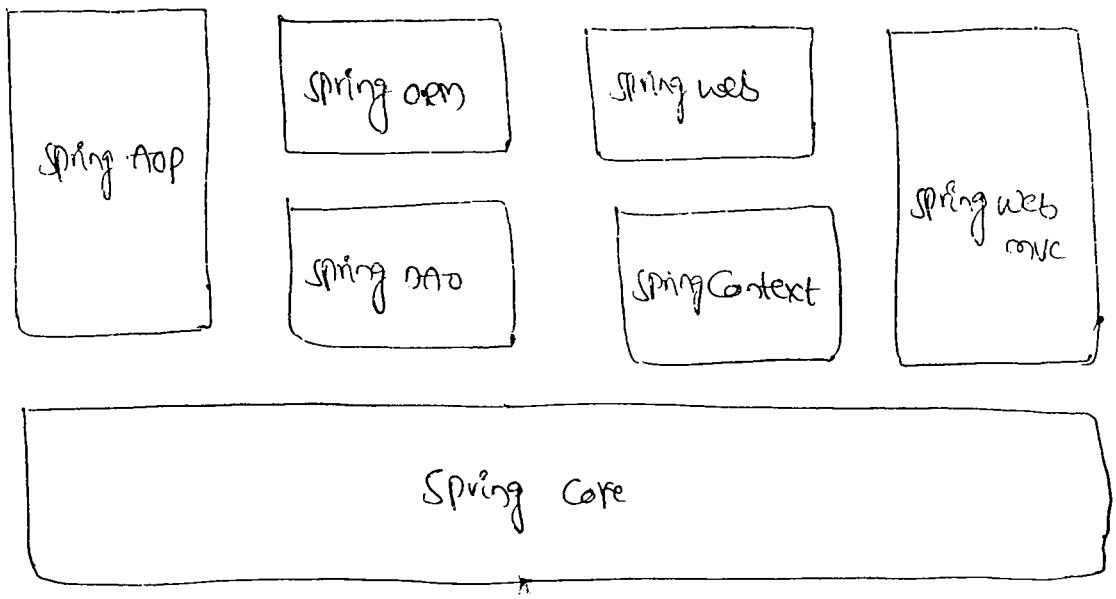
Spring containers are given as predefined Java classes. by just creating objects for these classes we can activate spring containers any where. This indicates spring containers are light weight containers.

The two containers of spring are no way alternate for servlet container, Jsp container and EJB container.

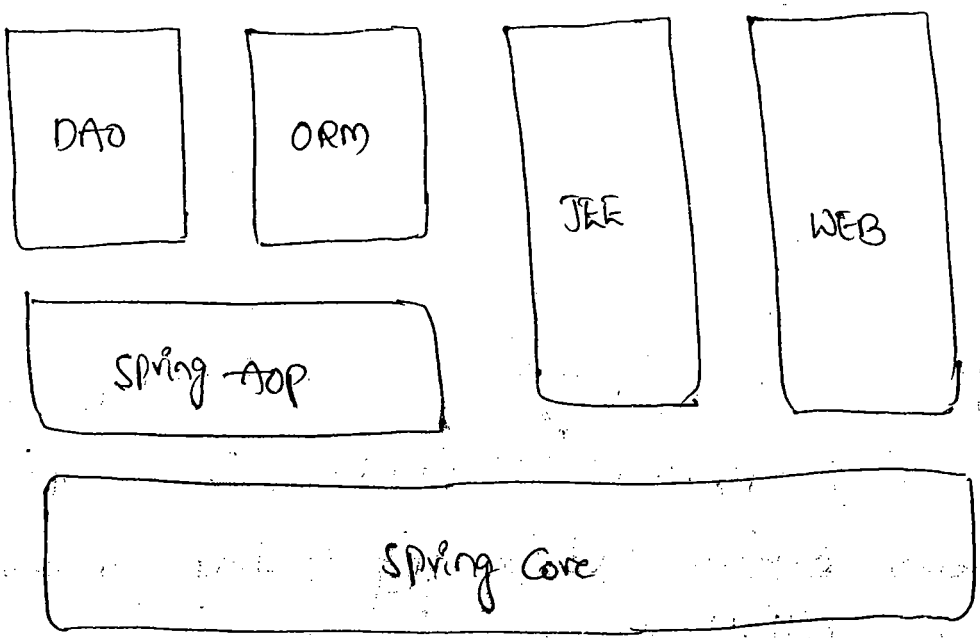
Resource means a program or file of a application.

Spring resources means the classes and Interfaces of spring application.

Spring 1.x High level Architecture Diagram :-



Spring 2.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 programer to develop framework style JDBC persistance logic by providing abstraction layer for the core JDBC technology.

Spring ORM module provides abstraction layer on ORM frameworks fw's like Hibernate, JPA, Toplink, JDO etc. It provides environment to develop db s/w Independent ORMapping persistance 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 communicatable from the web fw s/w apps like struts apps, JSF application etc.

b) provides springs own web fw called springmvc / springwebmvc to develop MVC2 arch based web applications.

Spring AOP provides environment to develop new middleware services and to apply existing m/w services on spring applications.

29/05

The client server applications with location transperency (Independency) are called as distributed applications.

Location Transperancy means the change in server application location will be recognized by clients dynamically.

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

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

The web app without location Transperancy are called client-server application (ex: class room level web apps)

The app that deals with complex, heavyweight, 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 layer → Business logic + persistance logic

View layer → presentation logic

Controller layer → Integration logic / Connectivity logic

The logic that gives communication b/w View and model layer resources, that monitors and controlls 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 persistance logic. (JDBC code, Hibernate code etc)

Now a days all companies are preferring to develop their projects as MVC2 architecture based projects.

Ex: ① JSP ⟶ servlet ⟶ Java Class / JavaBean ⟶ DB s/w
(View) (Controller) (Model)

② JSP ⟶ servlet ⟶ Java Class / JavaBean ⟶ DAO class ⟶ DB s/w
(View) (Controller) (Business logic) (persistance logic)
(Model)

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

③ JSP ⟶ servlet ⟶ EJB session Bean ⟶ EJB entity Bean ⟶ DB s/w
(View) (Controller) (Business logic) (persistance logic)
(Model)

④ ③ Struts ⟶ EJB session Bean ⟶ DAO class ⟶ DB s/w
(View & Controller) (Business logic) (persistance logic)
(Model) (JDBC based)

⑤ ④ Struts APP ⟶ EJB session Bean ⟶ Hibernate ⟶ DB s/w
(V & C) (B. Logic) (M) (P. logic)

*: EJB components are heavy weight, because they need EJB container and application server s/w kind of Heavy weight s/w's for execution.

① popularity

② Struts App ⟶ Spring app / Spring 9ee app ⟶ Hibernate ⟶ DB s/w
(V & C) (b.logic) — (M) — (p.logic)

③ Spring web mvc ⟶ Spring 9ee app ⟶ spring DAO/ORM app ⟶ DB s/w
(V & C) (b.logic) — (M) — (persistance logic)

Spring can be used in every layer of MVC2 architecture based projects to develop all the logics, as of now industry is using spring only in the model layer to develop business logics as alternate for ejb session Bean.

* The possible technologies of View layer:

JSP, HTML, velocity, freeMarker, XSLT, etc

Controller layer:

Servlet, servlet filter

Web fw s/w's to develop View & Controller layer logics:

Struts, webwork, JSF, Wicket, Xwork, Cocoon, Tapestry, spring web mvc etc.

Model layer to develop Business logic

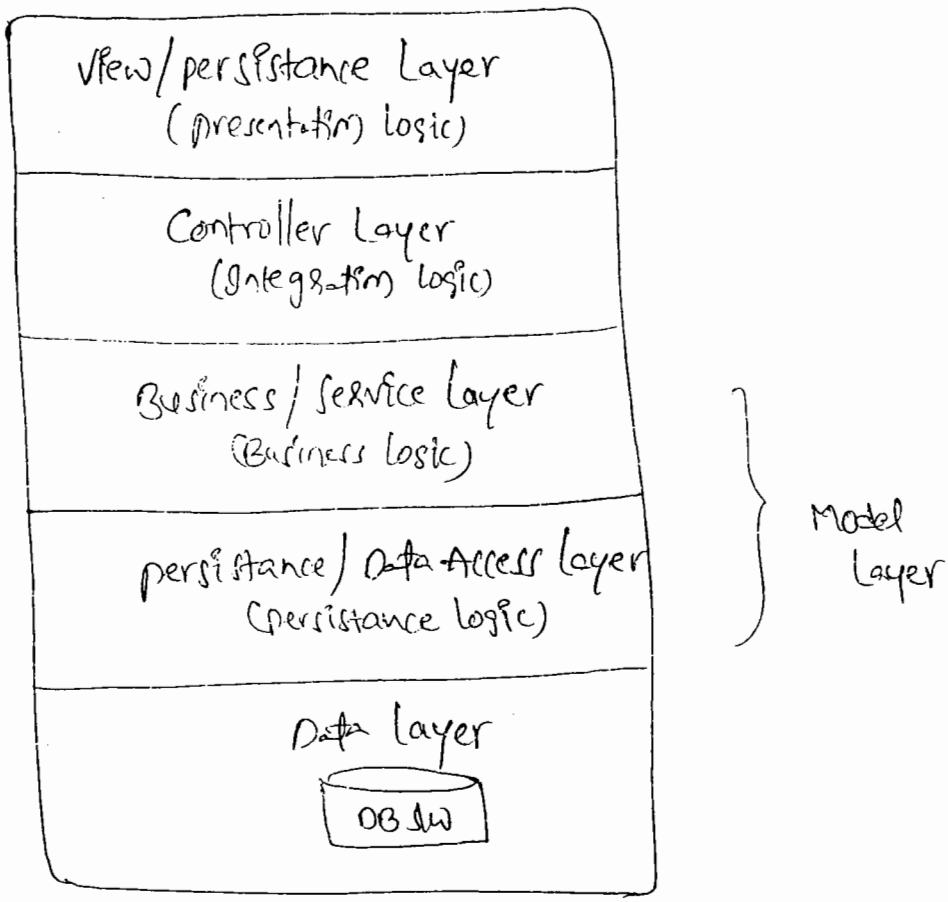
EJB session Bean, spring 9ee module, Java class, JavaBean, RMI, CORBA, WEBservices, HTTP Invoker etc

Model Layer to develop persistance logic:

JDBC, Hibernate, iBatis, 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 :-



Ex struts-spring-hibernate combination based projects:

JSP → View/presentation layer

Action servlet & resources → Controller layer

Spring core APP/ spring Jee APP → service/Business layer

spring orm/ hibernate → persistance layer

Oracle/MySQL → Data layer

Spring :- spring is an open source, light weight, loosly coupled, Aspect oriented, dependency injection based Java-Jee flw sw to develop all kinds of applications in Java.

3 days

Open Source

Open Source s/w are not only free s/w but they also expose their source code to programmers when s/w is installed.

To install Spring 2.5 s/w extract "Spring-framework-2.5.1-with-dependencies.zip" file to a folder.

Lightweight: spring-home/src folder contains the source code of Spring s/w.

Spring s/w 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 Bean-factory or ApplicationContext Containers which can be activated any where 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 Containers/servers (like web and App servers) for execution and the resources of those components are their respective-API dependent.

Loosly Coupled: If degree of dependency is very less b/w two components then they are called as loosly coupled components.

If degree of dependency is high/more then they are called as tightly coupled components.

Ex: CPU and monitor are tightly coupled divices
TV and remote are loosly coupled divices.

Spring fw is loosly coupled fw, because

1. We can use specific spring module or all spring modules in application development. (The degree of dependency is less fw 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 fw provides facilities for the programer to works with builtin middleware services and third party middleware services. This module also allows the programer to develop user defined m/w services.

These features of spring makes the spring fw 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 fw. fw / 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 fws 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 Work, samples folders of spring home 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 fw recognizing 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.

02/09/11

Spring application resources are spring API independent That means the classes and interfaces of spring application can be developed as POJO classes and POJI'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, Hibernate 3.x are designed supporting POJO/POJI model programming.

Every s/w technology contains API, API is the base for programer to develop that technology based s/w 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.event packages.

POJO class:- If Java class that is taken as resource of certain s/w 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 { } ⇒ based on spring

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

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 POJI.

If Java app uses third party API (other than JDK API's) then the third party API related main and dependent Jar files must be added to classpath to make Java tools (Javac, Java etc) to recognize and use third party APIs

* when Java application uses spring API the 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 fw sw to develop all kinds of Java, Jee and other applications in this style.

Supports POJO/POJI model programming

supports both xml, Annotations based configurations.

Allows to develop standalone, desktop, two tier, web distributed, enterprise ... appl.

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

Allows to works 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, jndi, servlets, jsp, rmi, jms, javamail, ... to simplify the development process.

provides its own sub's to develop applications directly.

Ex: HttpInvoker : springs own distributed technology to develop distributed apps.

Spring webmvc : spring's own web fw sub to develop mvc arch based projects.

provides light weight containers which can be activated without using web server or app server sub.

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 sub to develop orm persistance logic.

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

Spring Beans are no way related with JavaBeans. But JavaBean can also become as springBean (every springBean need not be JavaBean)

SpringBean can be a pojo or non pojo class.

predefined or Userdefined or Thirdparty supplied Java classes that can be instantiated by spring Container is called as SpringBean.

-All SpringBeans must be configured in spring configuration file

Ex: SpringBean class

```
1. Java.Util.Date.
2. public class Test
   {
      // some properties
      // some methods
   }
```

Any <file name>.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.xmlBeanFactory".

Ex:

```
FileSystemResource res = new FileSystemResource("demo.xml");
XmlBeanFactory factory = new XmlBeanFactory(res);
```

Spring Core Module: —

→ Gives Beanfactory container

→ Allows to observe diff modes of dependency Injection & bean life cycle

→ Allows to develop app that can have only local clients

Note: —

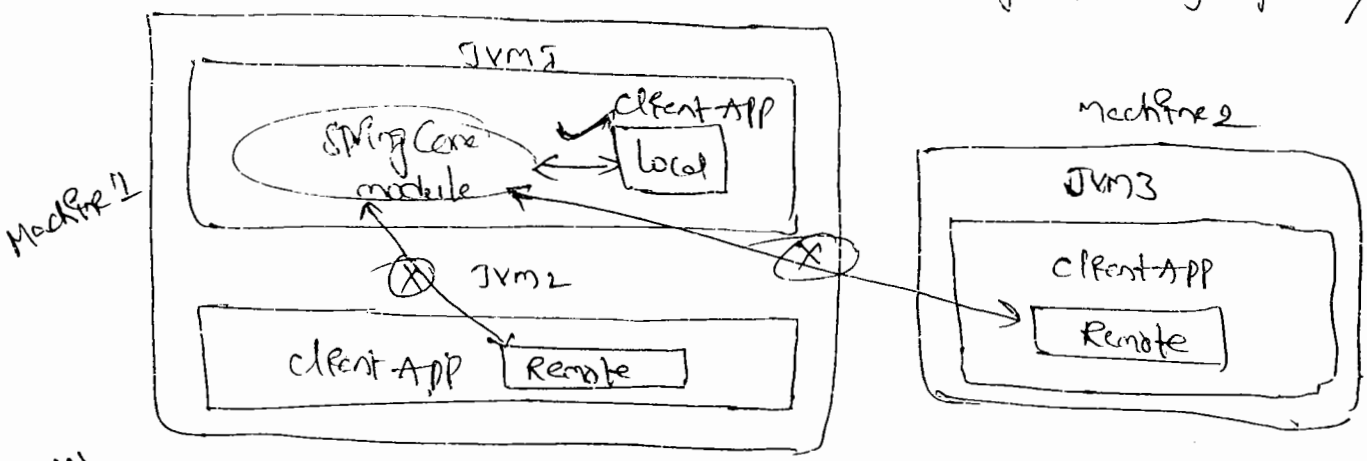
In spring environment any reusable and pattern ~~pattern~~ Java class is called as springBean

App and its client in same JVM → local, different → remote client

Distributed apps allow both 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 support in that computer. i.e in one computer multiple JVM's can be activated simultaneously or parallely



Resources of Spring Core Module App :-

Spring core module app resources

1) Spring Interface (can be a POJI) (.java) → optional
2) Spring Bean class (Can be a POJO class) (.java)
3) Spring Bean Configuration file (Any<filename>.xml) (.xml)
4) Client Application (Must be local client app) (.java)

↓ Client to Core module application.

Spring Interface :-

→ It is POJI (It can be a POJI)
→ Contains the declaration of business methods or utility methods.
→ Acts as common understanding document b/w spring bean class 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 for spring developer
- → 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 <filenames.xml can be there as spring config file.
→ No default file name exists as 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 process the spring Container dynamically assigns dependent values to spring Bean.

Advantages of Dependency Loosely :-

→ Resource/SpringBean can search and gather only the needed dependent values.

DisAdvantage :

→ Resource/SpringBean needs to spend time to search and gather dependent values.

Advantage of Dependency Injection :-

→ Resource/springbean 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 springbean)

2. Constructor Injection (Through parameterized constructors of springBean class)

3. Interface Injection (By implementing special interfaces on springBean 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 springBean 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;
    // setXXX(-) supporting Setter Injection
    public void setMsg(String msg)
    {
        this.msg = msg;
    }
    // write business method
    public void bm1() {
        ..... } → Business logic of bussins method the injected 'msg' property value can be used here.
    }
```

Demo.xml (Spring config file)

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

value to Inject

When above xml 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. (id attribute value) [like DemoBean ds = new DemoBean();]

5) Spring container dynamically calls setMsg() with argument value "hello" on Beanclass 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 ConstructorInjection :-

DemoBean.java

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

Demo.xml :

```
<beans>
    <bean id="ds" class="DemoBean">
        <constructor-arg>
            <value>Hello</value>
        </constructor-arg>
    </bean>
</beans>
```

Note: If Bean property is configured by using <property> tag then spring container performs setterInjection on that property.

If Bean property is configured by using <constructor-arg> tag then spring container performs ConstructorInjection on that beanproperty.

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");

06/09/11

Naming Conventions of Spring Core module App:

x ⟶ Spring Interface Name
xBean ⟶ Spring Bean class Name
xcfg.xml ⟶ Spring configuration file Name
xclient ⟶ client Application

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

Spring Core Module Application:

(To demonstrate setter Injection)

es1 Apps
└→ Spring
└→ fes1
→ Demo.java
→ DemoBean.java
→ democfg.xml
→ Democlient.java

Procedure to Create and execute the above CoreModule App:

Step0: Develop the above resources related source code.

Demo.java (POJI):

```
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)
    {
        S.O.P("DemoBean : setMsg(-)");
        this.msg = msg;
    }
// Implements business methods having business logic
    public String generateWishMsg(String uname)
    {
        SO.P("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){
    S.O.P("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">
```

[Gather it from Spring-home/dist/resources/spring-beans-2.0.dtd file]

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

SpringBean Configuration

Bean property Configuration for setter Injection.

* SpringConfiguration 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
{
   psvm (String args[]) throws Exception
   {
      // locate spring configuration file
         FileSystemResource res = new FileSystemResource ("Democfg.xml");
      // Activate spring container
         XmlBeanFactory factory = new XmlBeanFactory (res);
      // get SpringBean class obj from spring container
         Demo dobj = (Demo) factory.getBean("ds");
```

```
// Call business methods on spring bean class object
        S.O.P("With msg is :" + obj.generateWishMsg("Nani"));
        S.O.P("factorial value of :" + obj.findFactorial(5));
    }
}
```

step ② : 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 obj = (Demo) factory.getBean("db");
```

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

a) Makes BeanFactory container to load DemoBean class based on the bean id "db".

b) Makes spring container to create DemoBean class object using zero argument constructor.

c) Makes spring container to call setMsg("Hello") on DemoBean class obj to perform setter injection on 'msg' property.

d) 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.

An Interface reference variable can refer its Implementation class object and can be used to call the Implemented methods of Implementation class.

Super class reference variable can refer its one sub class obj and can be used to call the overriden 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 refered by using

a) DemoBean class reference variable

DemoBean bobj = (DemoBean) factory.getBean("ds")

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

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

problems and solutions related to factory.getBean(-) call :—

problem:

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

Solution 1 :

```
DemoBean bobj = (DemoBean) factory.getBean("ds");
bobj.findFactorial(5);
bobj.generateWishMsg("Nani");
```
} valid

Solution 2:

```
Demo bobj1 = (Demo) bobj;
```
(or)
```
Demo bobj1 = (Demo) factory.getBean("ds");
bobj1.findFactorial(5);
bobj1.generateWishMsg("Nani");
```
} valid

Spring Container perform dependency Injection only on those springBean objects that are created by Spring Container. It Cannot perform that dependency Injection operations on programer supplied and explicitly created springBean 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 just reads and verifies xml entries of spring config file against syntax rules and dtd rules but it never creates springBean class object in this process.

Beanfactory Container creates springBean class object only when client app calls getBean(-) with BeanId.

08/08/11

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

If certain bean property of springBean 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 setterInjection will be effected as final value.

If all Bean properties of springBean are configured for only setterInjection then spring container uses zero param constructor to create Bean class object.

If few are all Bean properties of springBean are configured only for ConstructorInjection then springContainer 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 JEE/JVM dynamically constructor and assigns initial values to instance variables when Java object is created as comes 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:2:- (On setter Injection) :-

Resources:

Demo.java → spring Interface
DemoBean.java, TestBean.java → Spring Beans
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 SetterInjection

public void setAge(int age) {
        this.age = age;
    S.O.P("DemoBean: setAge(-)");
    }

public void setD(Date d) {
        this.d=d;
    S.O.P("DemoBean: setD(-)");
    }

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("zeroargConstructor"); }
    public void setMsg(String msg) {
        this.msg = msg;
    }  S.O.P("TestBean: setMsg(-)");
    public String toString() {
    } return "TestBean.msg =" + msg;
}
```

DemoCfg.xml :—

```
<!DOCTYPE . . . . . .

<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>
   </bean>
   <bean id ="dti" 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 ="ti"/> </property> **
   </bean>
</beans>
```

predefined Java class as spring bean

* Spring Container injects java.util.Date class object (dt) to "d" property of DemoBean class.

** Spring container injects TestBean class object (ti) to "tb" property of DemoBean class.

DemoClient.java :—

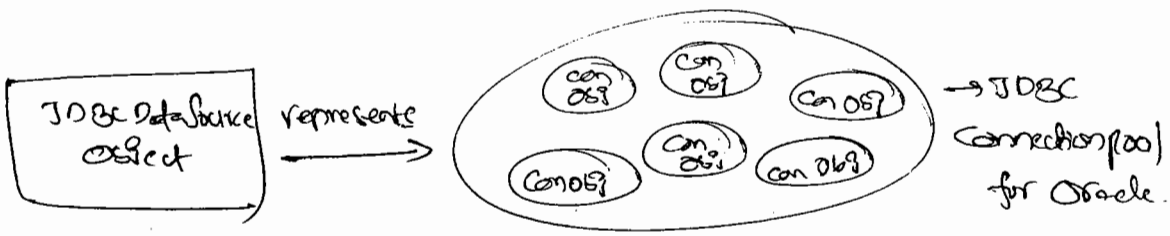
```
import org.springframework.core.io.FileSystemResource;
import org.springframework.beans.factory.xml.XmlBeanFactory;
   public class DemoClient
   {
      p.s.vm (String s[],) throws Exception
```

```
        FileSystemResource res = new FileSystemResource("democfg.xml");
        XmlBeanFactory factory = new XmlBeanFactory(res);
        Demo bobj = (Demo) factory.getBean("db");
        s.o.p("Result is :" + bobj.sayHello());
    }
}
```

6/9/17

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

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



Spring supplies org.sf.jdbc.datasource.DriverManagerDataSource class which gives JDBC datasource obj representing the builtin JDBC connectionpool 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 SpringBean through setterInjection.

Resources of Application :

DBOperation.java (Spring Interface) → POJI

DBOperationBean.java (Spring Bean class) → POJO

SpringCfg.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 setterInjection

    // Implementing business methods
    public float fetchSalary(int eno)
    {
       float sal = 0.0f;
       try {
    // use jdbc datasource obj to get jdbc con obj from jdbc con pool
```

```
// write Jdbc persistance logic

Statement st = con.createStatement();
ResultSet rs = st.executeQuery("select sal from emp
                                where eno =" + eno);
if(rs.next())
    sal = getfloat(1);
// close Jdbc objects
    rs.close();
    st.close();
    con.close();
}
catch(Exception e)
{
    e.printStackTrace();
}
return sal;
}

public String fetchName(int eno)
{
String name=null;
try {
    Connection con = ds.getConnection();
    Statement st = con.createStatement();
    ResultSet rs = st.executeQuery("select ename from
                                emp where eno =" + eno);
    if (rs.next())
        name = rs.getString(1);
} rs.close(); st.close(); con.close();
Catch (Exception e) { e.printStackTrace(); }
    return name;
}
}
```

spring(cfg.xml) :—

```
<!DOCTYPE . . . . .

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

predefined class supplied by spring API (Non POJO) to configured as springBean

our spring Bean class configuration (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.springframeworks.core.io.*;
import org.springframeworks.beans.factory.xml.*;
  public class DBClient
  {
    psvm(String args[]) throws Exception
```

```
// Activate Container by locating spring cfg file
FileSystemResource fes = new FileSystemResource("SpringBeanCfg.xml");
    XmlBeanFactory factory = new XmlBeanFactory(res);
// get our SpringBean class obj
DBOperation bobj = (DBOperation) factory.getBean("dbob");
// Call business methods
    S.O.P("Employee salary is:" + bobj.fetchSalary(4225));
    S.O.P("Employee name is:" + bobj.fetchName(4225));
  }
}
```

Jar files:-

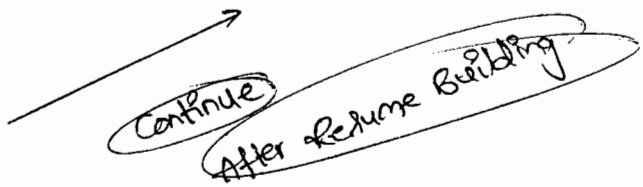
spring.jar

Commons-logging.jar

Oracle.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.



02/10/14

Resume Building

Resume: (14-Bold)

font : Times New Roman, Arial, Vasathana

Size : 9 or 10

Name : 11-Bold

phone : 9, email : 9

documents (Exp)

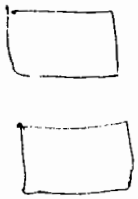
Employee Id
salary slips
Bank statements
form 16
Releaving letter

Consultancy: (freshers)
→ magna IT - Macabtank

HR
Questions: (Exp)

1) Why you are leaving that Company?
2) After how many do you join? (after 1-2 months)
3) CTC → 3.2 lacs
4) Hike → 40 to 50%.

RESUME



Java certification logos

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

Career Objective:

Achievements:

1. Academic ranks related.
2. Point on scsp/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)	Nagarjuna University	2005-08	84%
10+2	Board of Intermediate	2003-05	88%
10th	Z.P. High School (S.S.C)	2003	75%

Technical profile:

programming Languages : C, C++, Java
Operating systems : Windows

Main Academic subjects:

→ DBMS, SE, OOSD

(End of Page 1)

project profile:

project #1 (as MCA final sem project) (Feb 2011 to June 2011)

Title: System to Monitor GSM schemes

client org: product/ Company Name

Environment: - same as exp -

Team size: 1

Role: Involved every where

Description:

(6 lines)

personal profile:

Name: Narayana Reddy Kanala

DOB: 1st May 1988

Alternate email id: knr002@yahoo.com

Alternate Cont no: +91-97031 98472

passport no: 4252425

Language known: English, Telugu, Hindi

(End of page 2)

(End).

CURRICULUM VITAE

NARAYANA REDDY. KANALA
Phone: +91-90303 67416
Email: knr002@gmail.com

Career Objective : (optional)

Experience summary :

- → Having 4+ years Experience on Java, J2ee and frameworks
- → hands on experience to work with struts, spring, hibernate ...
- → Ability to implement design patterns
- → Experienced to work with servers like weblogic, jboss, glassfish, ...
- → Good Teamplayer, self motivator

Achievements :

- → Secure 85% in SCJP 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 till date
- → worked as programer for Infosys from Aug 2008 to Dec 2010

Technical profile

Skill	Experienced	Awareness
programing lang's	Java	C, C++
Operation systems	windows	Unix
Database sw's	oracle	mysql
web technologies	HTML, JavaScript	AJAX
Jse concepts	awt, swing, [illegible]	Struts
Jee concepts	servlets, JSP's	EJB

frameworks	struts, spring	JSP
ORM s/w's	hibernate	—
servers	Tomcat, weblogic	JBoss, glassfish
SOA	webservices	—
IDE's	MyEclipse, NetBeans	—
Logging tool	Log4j	—
Built tool	ANT	MAVEN
CVS 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: CitiBank | Infonce | product

Environment: JDK 1.5, Struts 1.3.8, JSP 2.0, servlets 2.4, Tomcat 5.5 NetBeans 6.7,

Team Size : 8

Duration : 1+ years

Role : programer

discription :

} 6th 7 lines apply (justify) (keep less, speak more)

Responsibilities:

→ Involved in UserInterface designing
→ participated in integration of project
→ done coding for "payRoll" module.
→ Implemented design patterns
→ performed unit Testing by using Junit Tool

end resume

12/09/11

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

Bean property type	tag name
Simple data type (Int, float...)	<value>
java.lang.String	<value>
Bean class (ref type)	<ref>
java.util.List	<list>
array	<list>
java.util.Map	<map>
java.util.set	<set>
java.util.properties	<props>
To set "null" value at Beanpropety	<null/>

Each element of regular Map datastructure like HashTable obj or HashMap obj can take any object as key and any obj as element.

where as the element of java.util.properties Map datastructure allows only string obj as key and string obj as value.

MyEclipse :-

type : IDE for Java Environment

version : 8.x Compatable with Jdk 1.6

vendor : Eclipse org

It is a commercial s/w, allows to configure any external servers with IDE s/w. To download www.myeclipseide.com.

A plugin is a patch s/w or s/w application that can enhance functionalities of existing s/w or s/w applications.

MyEclipse, Eclipse Galileo, EasyEclipse are the advanced ide's that are developed based on Eclipse IDE.

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

step ①: Launch MyEclipse IDE by choosing a workspace folder.

step ②: Create Java project in myEclipse ide

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

step ③: -Add spring Capabilities to the project.

Right clicks on project (MyProject) → MyEclipse → Add spring Capabilities
→ ⊙ spring 2.5 → ☑ select spring 2.5 core libraries → Next →
Bean configuration file: [DemoCfg.xml] → finish

step ④: -Add spring Interface, spring Bean class to project as shown below.

→ Right clicks on project → New → Interface → Name: [Demo] → finish

Demo.java:-

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

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

DemoBean.java:-

(Ctrl + shift + O → to remove import)

```
import java.util.*;
public class DemoBean implements Demo
{
    int age;
    String name;
    Date dt;
    int marks[3];
    List fruits;
    Set phones;
    Map capitals;
```

properties facultes;

// write setXxx(-) supporting setter injection

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() +
           "facultes =" + facultes.toString();
    }
}
```

Step ⑤: Add following content inside the Democfg.xml (In <beans>)

```
<bean id="dt" class="java.util.Date"/>
<bean id="db" class="DemoBean">
    <property name="age" value="25" />
    <property name="name" value="Nani" />
    <property name="d" ref="dt" />
    <property name="marks">
        <list>
            <value>30</value>
            <value>40</value> </list>
    </property>
```

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

<property name = "phones" >
    <set>
        <value> 9030367416 </value>
        <value> 01065538968 </value>
        <ref bean = "dt" />
    </set>
</property>

<property name = "Capitals"
    <map>
        <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>
    </map>
</property>
```

```
<property name = "faculties">
    <props>
        <prop key = "Nataraz"> Java faculty </prop>
        <prop key = "prakash"> .Net faculty <prop>
        <prop key = "Rami Reddy"> Java faculty </prop>
    </props>
</property>
</bean>
</beans>
```

Step ⑥ : Add client Application to the project.

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

DemoClient.java :— [syso ctrl+space → SOP]

```
public class DemoClient {
    P S V m (String args[]) {
        FileSystemResource res = new FileSystemResource (
                                    ".\\src\\DemoCfg.xml");
        XmlBeanFactory factory = new XmlBeanFactory (res);
        Demo ds = (Demo) factory.getBean("ds");
        S.O.println(ds.sayHello("Reddy"));
    }
}
```

Step ⑦ : Run the client Application

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

13/09/11 :

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 to this bean property we use <props> tag as shown below.

In springcfg.xml :-

```
<bean id="drds" class="org.sf.jdbc.ds.DMDS">
  <property name="driverclassname"><value>oracle...</value></p>
  <property name="url"><value>jdbc:oracle:thin...</v></p>
  <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.

Ans:- 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

SpringBean class must have either explicitly placed zero-arg Constructor or Javac compiler generated default zero-arg Constructor in the above said situation.

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

Understanding Constructor Injection :—

In Constructor Injection the SpringContainer uses parameterized constructor to create SpringBean class object and to assign / inject values to Bean properties.

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

Ex: If <Constructor-arg> is written for 3 times under <bean> then spring container uses 3-param constructor to perform Constructor-Injection on Bean properties.

Ex: Demo.java :—

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

DemoBean.java :—

```
import java.util.*;
public class DemoBean implements Demo
{
    private String msg;
    private int age;
    private float avg;
    // parameterized Constructors for Constructor Injection.
```

```
public DemoBean(String msg, int age, float avg)
{
    S.O.P("3 param Constructor");
    this.msg = msg;
    this.age = age;
    this.avg = avg;
}

public DemoBean(int age, float avg)
{
    S.O.P("2-param Constructor");
    this.age = age;
    this.avg = avg;
}

public DemoBean(float avg)
{
    S.O.P("1-Param constructor");
    this.avg = avg;
}

public String sayHello()
{
    return "Good Morning" + "msg=" + msg +
           "age=" + age + "avg=" + avg;
}
}
```

DemoCfg.xml:—

```
<beans>
  <bean id="ds" class="DemoBean">
    <constructor-arg><value> Hello </value></con...>
    <constructor-arg><value> 24 </value></c>
    <constructor-arg><value> 12.34 </value></c>
  </bean>
</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:

Ex:- 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)
     {
        SO.P("3-param constructor");
        this.msg = msg;
        this.age = age;
        this.avg = avg;
     }
   public String sayHello() { return "Good Night"; }
}
```

DemoCfg.xml:—

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

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)
     {
       SO.P(" 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 = "2" ><v> 30 </v> </con-arg>
    <con-arg index = "0" > <v> 60 </v> < /con--arg>
    <con-arg index = "1" > <v> 90 </v> </con-arg>
  </bean>
</beans>
```

Note :— If there is single bean property in springBean class then go for constructor Injection bcoz 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 programer.

Note:- It is not manditory 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 4 setXXX() are enough. But we need to 16 (24) no. of constructors to satisfy constructor Injection in all angles. so setterInjection is prefered compared to Constructor Injection when springBean class is having more than one Bean property.

Note:

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 → BuildPath → Add libraries → MyEclipse libraries → Next → select more spring libraries → finish.

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

demo.java :— same as sep 12th example

DemoBean.java :— same as sep 12th example. but replace 8 setter methods with one 8-param Constructor as shown below.

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

Note: To generate the above parameterized constructor through myEclipse IDE select 8 bean properties of DemoBean class and right click → source → Generate Constructor using fields

democfg.xml:-

same as setpath example but replace Complete <property> (including attribute) with <constructor-arg> tag. Similarly replace </property> tag with </constructor-arg> tag.

DemoClient.java :-

same as setpath example.

we can locate spring config file for BeanFactory container either by using FileSystemResource class or ClassPathResource 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");
↓ relative path
- Locates democfg.xml from current directory.

FileSystemResource res=new FileSystemResource("..\\democfg.xml");
↑
- Locates democfg.xml from parent directory

FileSystemResource res = new FileSystemResource("E:\\abc\\democfg.xml");
↑ absolute path
- Locates democfg.xml from E:\abc folder

The ClassPathResource class locates the given spring 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 this E:/apps folder to environmental variable classpath

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

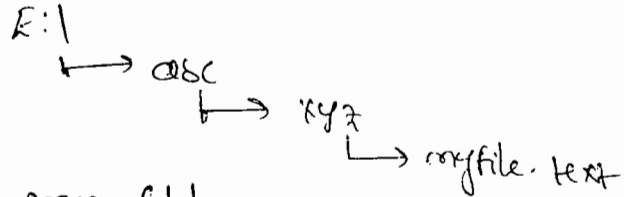
Jar file Based: or create a jar file on E:/apps folder like

E:/apps > jar cf Test.jar . ↵

Then add this jar file E:/apps/test.jar to class path.

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

Ans:- original resource file is like:



1) Current works folder is: e:/abc folder

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.

15/04/U

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 cant create objects for Interfaces and Abstract classes in 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:

DemoCfg.xml :—

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

Here forName returns java.lang.Class object

valueOf returns java.lang.String 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
{
  psvm(String arg[]) throws Exception
  {
    ClassPathResource res=new ClassPathResource("DemoCfg.xml");
    XmlBeanFactory factory = new XmlBeanFactory(res);
    Class cosi = (Class) factory.getBean("c1");
    sop("cosi data is" + cosi.toString());
    sop("cosi class name is" + cosi.getClass());
```

```
String s06j = (String) factory.getBean("s1");
    S.O.P(" s06j data is :" + s06j);
    S.O.P(" s06j class name is :" + s06j.getClass());
Calendar cal = (Calendar) factory.getBean("cal");
    S.O.P(" cl data is :" + cl.toString());
    S.O.P(" cl class name is :" + cl.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 or JRE tries to create multiple objects for that class it just allows to create only one object.

for a normal Java class which actually allows to create multiple Java objects if container or 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 programer) but servlet Container always creates and uses only one object of that class. It doesn't mean programers 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 SpringContainer keeping springBean class

Those are 1. Singleton (default) } Applicable for standalone
2. prototype } and web environment of
3. request } spring apps
4. session } only for web environment.

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

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

<bean id="ds" 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 if 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">

16/09

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.

Application Context Container :—

Application Context Container is extention 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. Preinstantiation of SpringBeans.
3. Support to read Bean property values from properties file.
4. Support for Internationalization.
5. Support for event handling in spring Container through event listeners and etc...

Activating ApplicationContext Container is nothing but creating object for a Java class that implements org.sf.context.ApplicationContext Interface (this Interface is the sub Interface of org.sf.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.sf.Context.support.FileSystemXmlApplicationContext
   - Activates ApplicationContext container by locating given spring configuration file from the specified path of file system.

```
FileSystemXmlApplicationContext ctx = new FileSystemApplicationContext
```

2) org.sf.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.sf.context.web.context.support.XmlWebApplicationContext.

- Useful to activate applicationContext container in webapplication environment.

Executing first spring app with ApplicationContext container:-

Demo.java, DemoBean.java, DemoCfg.xml are same as 1st app.

DemoClient.java:-

```
import org.sf.context.support.*;
public class DemoClient
{
  P S V M(String args[]) throws Exception
  {
    // Activate ApplicationContext container
    FileSystemXmlApplicationContext ctx = new
                    FileSystemXmlApplicationContext("DemoCfg.xml");
    // get Access to SpringBean class obj from Container
    Demo bobj = (Demo) ctx.getBean("db");
    // call b.methods
    S.O.P("Wish message is :" + bobj.generateWishMsg("Nani"));
    S.O.P("factorial value is :" + bobj.findFactorial(5));
  }
}
```

BeanFactory Container cannot take multiple spring Config files, where as ApplicationContext container can works with Multiple spring Configuration files, as shown below.

ApplicationContext Container is supplied in spring Context / JEE modules, but 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 ApplicationContext 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
 }
```

democfg1.xml:-

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

DemoCfg2.xml :-

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

DemoCfg3.xml :-

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

DemoClient.java :-

```
import org.sf.context.support.*;
public class DemoClient
{
  psvm(String args[]) throws Exception
  {
    String cfg[] = {"DemoCfg1.xml", "DemoCfg2.xml",
                    "DemoCfg3.xml" }
    FileSystemXmlApplicationContext ctx =
        new FileSystemXmlApplicationContext (cfg);
    Demo bobj = (Demo) ctx.getBean("db");
    SO.P("Result is" + bobj.sayHello() );
  }
}
```

1-7-09

Making 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 springBeans.

factory.getBean("ds") → 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.

FAQ :- There are ten Beans spring config file, Can 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 Bean properties of first five Bean classes are not using last five springBean class objects.

FAQ →

```
<beans>
  <bean id="t1" class="TestBean" scope="prototype"/>
  <bean id="ds" class="DemoBean" scope="singleton">
    <property name="tb"> <ref bean="t1"/> </property>
  </bean>
```

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 that Bean class Bean id as dependent value to singleton scope springBean related Bean property as shown in the above question related spring 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 dont hardcode any values in your applications that will be changed in future, Moreover pass them to application from outside the application by taking the support of properties files.

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

Beanfactory container doesnt support properties files where as ApplicationContext container supports them).

for security reasons the username and passwords of db sw 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"><value> ${my.age} </value></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 container recognizing place holders and to specify the name of .properties file always configure the following predefined springBean class.

org.sf.beans.factory.config.PropertyPlaceholderConfigurer

for this we use two properties of these class

location → for single property file

locations → ... for multiple properties file.

In spring Config file :-

```
<bean id="propConfig" class="org.sf.beans.factory.config.
                        propertyPlaceholderConfigurer">
    <property name="location"> <value> myfile1.properties </v></p>
</bean>
```

The above code is for single property file configuration.

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

The above code is for multiple properties files configurations

19/04/11.

Procedure to add properties file support for spring App that is given on 10th of this month. (dataSource example).

Resources :
1. DBOperation.java
2. DBOperationBean.java } → same as 10th date example.
3. SpringCfg.xml
4. DBClient.java
5. DBDetails.properties.

DBDetails.properties :

```
my.driver = oracle.jdbc.driver.OracleDriver
my.URL = jdbc:oracle:thin:@localhost:1521:orcl
my.dbuser = scott
my.dbpwd = tiger
```

Spring cfg. xml :-

```
<beans>
    <bean id="propConfig" class="org.sf.beans.factory.config
                                 .PropertyPlaceholderConfigurer">
        <property name="location"><value>DBDetails.properties
                                        </value></property>
    </bean>
    <bean id="drds" class="org.sf.jdbc.datasource.
                           DriverManagerDataSource">
        <property name="driverClassName"><value>${my.driver}
                                        </value></property>
        <property name="url" value="${my.url}" />
        <property name="connectionProperties">
            <props>
                <prop key="user">${my.dbuser}</prop>
                <prop key="password">${my.dbpwd}</prop>
            </props>
        </property>
    </bean>
    <bean id="dbot" class="DBOperationBean">
        <property name="ds" ref="drds" />
    </bean>
</beans>
```

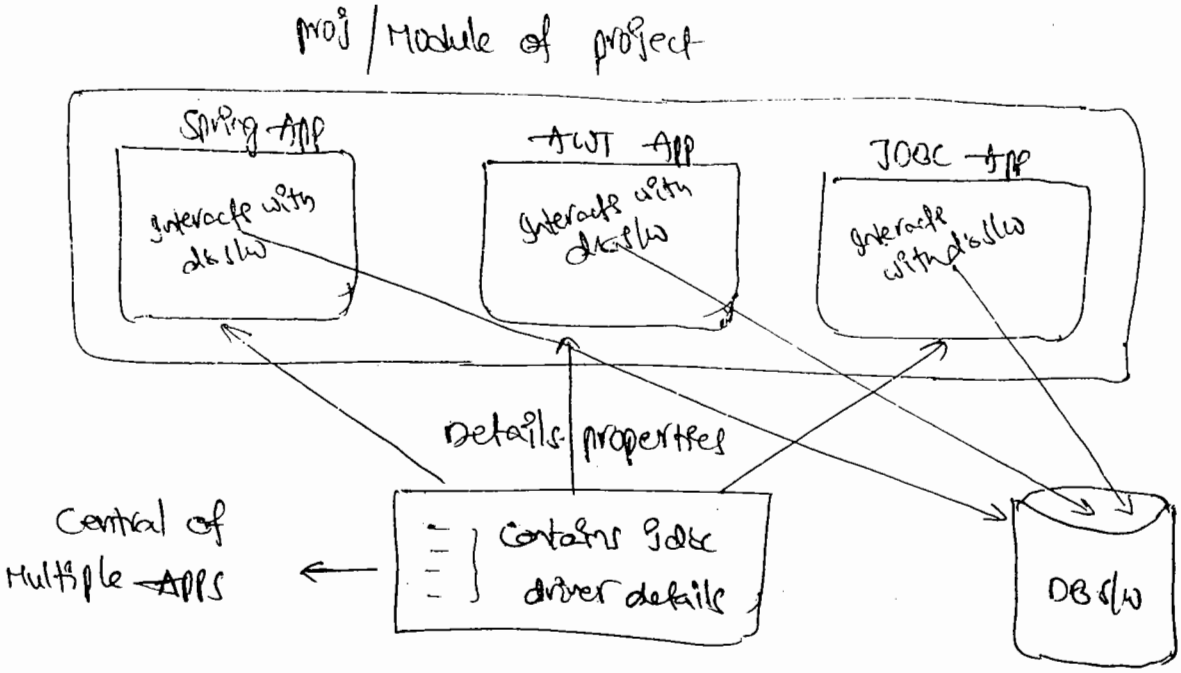
To make spring container to recognize and work with properties file and placeholders

placeholder

DBClient.java :-

Same as serlets example, but ApplicationContextContainer instead of BeanFactory Container.

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



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 communities based client applications is nothing but enabling Internationalization (I18n) on the application.

I18N can deal with only presentation (GUI) 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), native2ascii (Builtin tool of JDK) to get unicode numbers of any word or sentence belonging to any language.

Unicode editor tool can be downloaded from www.baigopi.com website.

Procedure to Getting Hindi words related unicode range numbers By using unicode Editor, Native2Ascii tools.

Step①: Download Unicode Editor tool from bigopi.com and install it.

Step②: Launch Unicode Editor tool (use Index.html)

Step③: Type certain Hindi words in the Editor area.

मिटाओ	–	delete
सुरक्षित	–	save
रुकिए	–	stop
रद्द	–	Cancel

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

Step⑤: use native2ascii to get unicode numbers for above Hindi words

E:\> native2ascii -encoding unicode myword.txt myfile.txt

↑ input ↑ output – unicode nos

Ex: myfile.txt:

Scalps

I18n can enabled on the application always deals with presentation logic of the application and it is no way responsible business logic and persistance logic of the application.

To enable I18n on standalone Java applications use Locale, ResourceBundle 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 = LÖSHEN
str2 = DENN
str3 = ZUG
str4 = ABSAGEN
```

App_fr_CA.properties :-

```
# for french language
str1 = EFFACER
str2 = SAUF
str3 = ARRÊT
str4 = ANNULER
```

App_hi_IN.properties :-

```
# for hindi language
str1 = \u0928\.....
str2 = \u0938\.....
str3 = \u0930\.....
str4 = \u0930\.....
```

→ collect from myfile.txt file

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

We can use Google Translator or some language convertors 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 I18n as shown below

I18nApp.java: (Save encoding style as Unicode)

```
import java.util.*;
import javax.swing.*;
import java.awt.*;

public class I18nApp {
    psvm (String args[]) {
// create Locale class obj based on language, Country code
    Locale l = new Locale (args[0], args[1]);
// pickup properties file based on the locale obj data
    ResourceBundle r = ResourceBundle.getBundle ("App", l);
// create frame window.
    JFrame jf = new JFrame();
    Container cp = jf.getContainerPane();
// create Buttons by getting lables 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.
        cp.setLayOut(new flowLayout());
        cp.add(b1);
        cp.add(b2);
        cp.add(b3);
        cp.add(b4);
        jf.pack();
    } jf.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 IlsnApp.java

> java IlsnApp de DE → for Germany

> java IlsnApp x y → Base file will pickedup, Bcoz lang code, country code are wrong.

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

When Ilsn 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 Ilsn.

Example Application on Ilsn 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.

beanCfg.xml :-

```
<beans>
  <bean id="messageSource" class="org.sf.context.support.RBMS">   ← Fixed Bean Id
    <property name="basenames">
      <list> <value>App</value> <value>App_de_DE</value>
             <value>App_fr_CA</value> <value>App_hi_IN</value>
      </list> </property>
  </beans> </beans>
```

DemoClient.java :-

```
import org.sf.context.support.*;
import java.util.*;
public class DemoClient {
    psvm(String args[]) throws Exception {
        //Locale object
    Locale l = new Locale(args[0], args[1]);
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext
    { //Activates the Spring Container.          ("DemoCfg.xml");
    // get msg from the activated properties file
                                    key in the properties file
        String msg = ctx.getMessage("str3", null, "default msg", l);
                                    ↑*①    ↑*②    ↑*③
            S.O.P("message is :" + msg);
    }
}
```

*① :- Actually it is java.lang.Object class Object[] to supply {n} 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 in All realtime project this ApplicationContext container is used. Moreover the ApplicationContext container gives better support to work with AOP module n/w services when compare to BeanFactory container

28/08

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. EventListener
4. Event Handling method.

AWT / swing Example :-

Source Component → Button
Event → ActionEvent
Event listener → ActionListener
Event-Handling method → public void actionPerformed(ActionEvent ae)
{ ..... }

Web Environment Example :-

Source obj → ServletContext obj
Event → ServletContextEvent
EventListener → ServletContextListener
Event Handling method → 1. public void contextInitialized() { ... }
(executes when servletContext obj is created)
2. public void contextDestroyed() { ... }

Spring Example :-

source obj → ApplicationContext container

Event → ApplicationEvent

EventListener → ApplicationListener

EventHandlingmethod → public void onApplicationEvent(ApplicationEvent ae)
(executes when AppContext 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 eventHandling on AppContext container by taking one seperate java class.

Example-Application :-

G:\Apps\Spring

- DemoJava
- DemoBean.java } → same as first App
- DemoCfg.xml
- DemoClient.java
- MyTestListener.java

DemoCfg.xml :-

```
<beans>
   <bean id="mtl" class="MyTestListener" />
   <bean id="db" class=DemoBean">
      <property name="msg"><value> Hello </v></p>
   </beans>
</beans>
```

MyTestListener.java :-

```
import org.sf.context.*;
public class MyTestListener implements ApplicationListener
```

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

DemoClient.java :-

- Same as firstApp 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.

→ FileSystemXmlApplicationContext ctx = new FSXAC("DemoCfg.xml");

→ starts container

→ ctx.close() → stops container

Note :- Event handling on SpringContainer is very useful when AppContext Container is used in webEnvironment.

We can't explicitly stop 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 configuring them explicitly. Both BeanFactory, AppContext Containers support AutoWiring.

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

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

Auto wiring is possible only on ref type bean properties (The Bean properties which can hold other bean class objects).

Limitations with Auto Wiring :—

① we cannot perform Auto wiring on primitive data type or String type bean properties which expect direct and simple stright values.

Ex: int age, String name, float avg etc.

② There is a possibility of getting Ambiguous state (dalibamma state) error when spring container finds multiple dependent objs for a single bean prop.

26/09/11

To know info about any xml tag or attributes of spring cfg file refer the dtd file rules or schema file rules (xsd file)

In dtd file like spring-beans-2.0.dtd file <springhome>/dist/release

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

<!ATTLIST <tagname> <attributename> → for attributes

Programmer can give instruction to spring Container to perform auto wiring in diff modes, by using the autowire attribute of <bean> 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's id names of dependent Bean class axis must match.

Example :-

E:\Apps\spring

- TestBean.java
- Demo.java
- DemoBean.java
- DemoCfg.xml
- DemoClient.java

[constructor - params
method call - arg]

TestBean.java :-

```
public class TestBean
{
    private String msg;

    public TestBean()
    {
        S.O.P("0-param Constructor of TestBean");
    }
```

```
    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()
    {
        S.o.println("0-param Constructor of DemoBean");
    }
    public DemoBean(TestBean tb)
    {
        this.tb = tb;
        S.o.println("1-param Constructor of DemoBean");
    }
    public void setTb(TestBean tb)
    {
        this.tb=tb;
        S.o.println("DemoBean: setTb() method");
    }
    public String sayHello()
    {
        return "Good morning" + "tb=" + tb.toString();
    }
}
```

DemoCfg.xml :- (Based on this configuration the Spring Container injects TestBean class obj (tb) to the "tb" property of DemoBean class through setter Injection)

```
<beans>
   <bean id="tb" class="TestBean">
      <property name="msg"><value>Hello</value></property>
   </bean>
   <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.st.context.support.*;
public class DemoClient
{
   psvm(String args[])
   {
      FileSystemXmlApplicationContext ctx = new
            FileSystemXmlApplicationContext("DemoCfg.xml");
```

(OR) →

```
      XmlBeanFactory factory = new XmlBeanFactory(new
            FileSystemResource("DemoCfg.xml");
      Demo bobj = (Demo) ctx.getBean("db");
      S.o.p("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 DemoCfg.xml :-

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

Note: Here tb property type of DemoBean class is matching with tb1 bean OS 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>Hai</v></p>
  </bean>
  <bean id="db" class="DemoBean" autowire="byType"/>
</beans>
```

(4) constructor :- Makes spring Container to perform Constructor Injection through parameterized constructor of springBean 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 recomended to avoid autowiring and recomended 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="ds"><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="tb1" class="TestBean">
      <p name="msg"><v> HaiK/v> </p> </bean>
```

```
<bean id="ds" class="demoBean" autowire="byName">
    <property name="to"> <ref bean="to1" /> </property>
</bean>
</beans>
```

(Explicit wiring)

27/08/14

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 logics and life cycle operations related logics in these life cycle methods.

programer 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 clycle that are rised on the obj are called as life cycle methods or container callbacks 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 programer to keep user-defind methods as life cycle methods, but they must specified/configured in spring cfg file during spring Bean Configuration.

In springBean class the life cycle method names are not fixed names, they are programer choice method names.

AppletContainer (AppletViewer or browser window) calls Applet life cycle 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 SpringBean instantiation and dependency injection.

- This method is useful to keep logic

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

ii) That verifies weather 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 springBean 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 msg;
  Date dt;
  public DemoBean() { S.O.P("0-param Constructor"); }
```

```
public void setAge(int age)
  { this.age = age;
    S.O.P(" setAge(-) method");
  }
public void setMsg(String msg)
  {
    this.msg = msg;
      S.O.P("setMsg(-) method");
  }
public void setD(Date d)
  {
    this.d = d;
      S.O.P(" setD(-) method");
  }
  // life cycle methods of springBean
public void myInit() throws Exception
  {
    S.O.P(" myInit life cycle method");
    if(age<=0 || msg==null || d==null)
      throw new Exception(" set values to Bean properties");
  }
public void myDestroy()
  {
    S.O.P(" myDestroy lifecycle method");
      age=0;
      msg=null;     // nullifying Bean properties
      d=null;
  }
} → public String sayHello() { return ... }
```

DemoCfg-xml :—

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

DemoClient.java :-

```
public class DemoClient
{
  psvm (String args[]) throws Exception
  {
    XmlBeanFactory factory = new XmlBeanFactory(new ...
      Demo bobj = (Demo) factory.getBean("ds");
      S.o.println("Result" + bobj.sayHello());
      factory.destroySingletons();
  }
}
```

In the above program, if the container is AppContext container then it is same as previous example.

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

→ Both ApplicationContext, BeanFactory containers support life cycle methods. we cannot design these life cycle methods with parameters.

Limitations of life cycle methods :-

1. If programer 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 errorprone process.

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

1. org.sf.beans.factory.InitializingBean

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

But here there is no need of configuring these method names explicitly in spring cfg file.

2 ways

The org.sf.beans.factory.InitializingBean(i) contains

afterPropertiesSet() → programer implements this method having same logic of init life cycle method.

The org.sf.beans.factory.DisposableBean(i) contains

destroy() → programer implements this method having same logic of destroy life cycle method.

When SpringBean 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 name 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)

Democfg.xml → 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() life cycle methods.

If you keep init() along with afterPropertiesSet() then the spring Container executes init() life cycle method after - afterPropertiesSet(). Like destroy() life cycle method executes after destroy() of DisposableBean interface.

*** Q: How did you use Abstract class and Interface in 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, where as 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 s/w specification gives certain API (classes, & interfaces) having Rules and Guidelines to develop s/w's.

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

Servlet specification contains set of Rules and guidelines to develop Servlet Container s/w.

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

Interface Injection :-

Since life cycle methods, InitializingBean, DisposableBean 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, So any container life cycle methods cannot perform dependency injection bcoz they cannot gather any external data.

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

These special values are like currentBeanId, underlying BeanFactory, AppContext containers and etc..

The xxx-Aware interfaces are :-

① org.sf.beans.factory.BeanNameAware
- useful to inject/get current Bean class BeanId being from that Bean class.

method → public void setBeanName(String name)
(name represents BeanId)

② org.sf.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.sf.context.ApplicationContextAware

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

method → public void setApplicationContext(ApplicationContext ctx) throws BeansException

Note:- SpringBeans use the injected underlying SpringContainers to know about other SpringBeans 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.sf.context.*;
import org.sf.beans.factory.*;
public class demoBean implements Demo, BeanNameAware,
                                  ApplicationContextAware
{
    String bname;
    ApplicationContext ctx;
// Implement methods of BeanNameAware, AppContextAware (i) (not setXxx();)
    public void setBeanName(String bname)
    {
        this.bname = bname;
    }
    public void setApplicationContext(ApplicationContext ctx)
    {
        this.ctx = ctx;
```

```
public String sayHello()
{
    S.O.P("Current Bean class Bean Id is : " + bname);   // gives db
    S.O.P("Current container is managing" +
                ctx.getBeanDefinitionCount() + " Beans");   // gives 3
    S.O.P("Current Container is managing the following Beans");
    String[] names = ctx.getBeanDefinitionNames();
    for(int i=0; i<names.length; ++i)
        S.O.P(names[i]);   → gives db, dt, dt1
    S.O.P("Current Bean Scope is Singleton ?" + ctx.isSingleton(bname));   // true
    S.O.P("Current Bean Scope is prototype ?" + ctx.isPrototype(bname));   // false
    return "Good Morning Nani...";
}
}
```

DemoCfg.xml :— (Here Both dt, dt1 are just demonstration purpose, not related with this)

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

DemoClient.java :—

Same as previous, but uses with AppContext container

* If programer wants to modify Bean property values after Injection and before using them in Business method then we need to works with special org.sf.beans.factory.config.BeanPostProcessor interface.

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

Example app on bean postprocessor

29/08/11

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

there 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 is 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 in a separate Java class that implements org.sf.beans.factory.config.BeanPostProcessor interface.

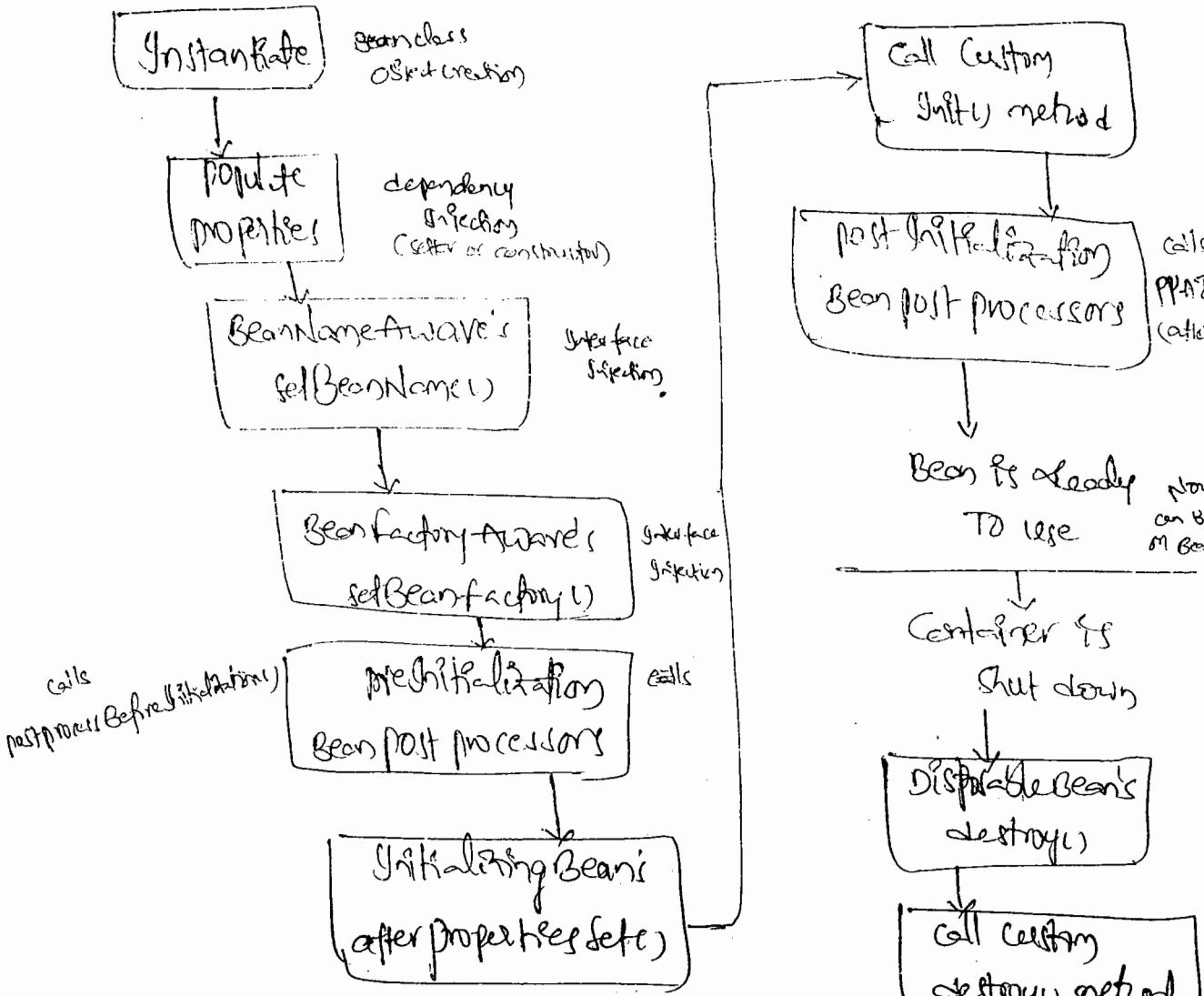
* for source code of the above application that deals with BeanPostProcessors refer supplementary handout given on 28/08/11.

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

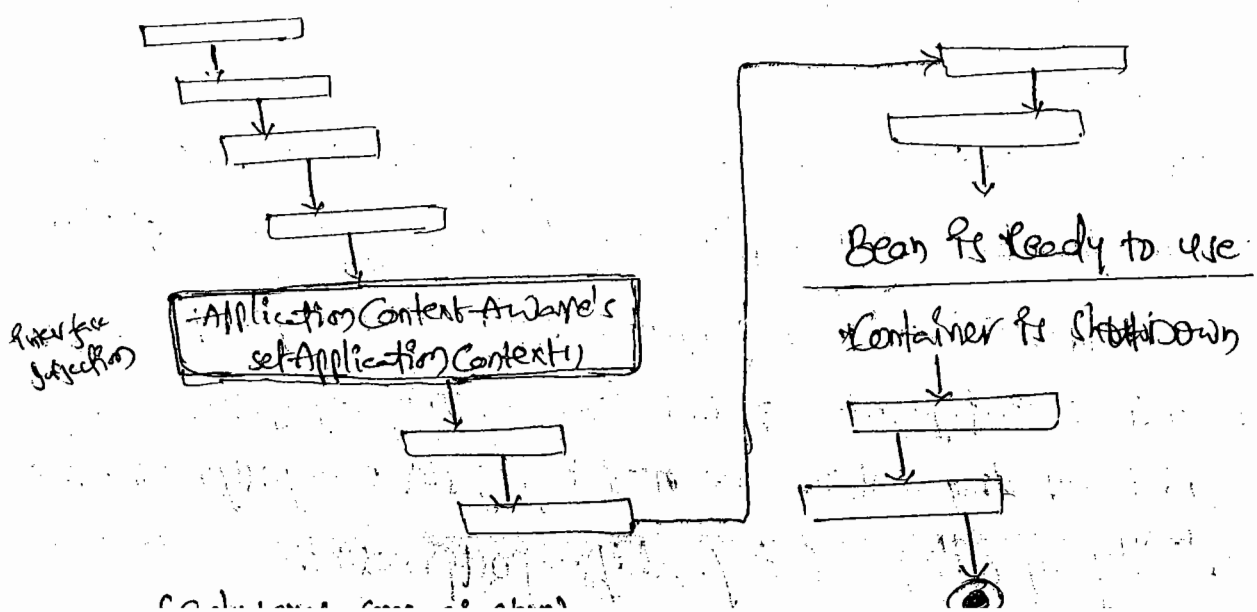
for BeanFactory container we need to register BeanPostProcessor class explicitly by calling factory.addBeanPostProcessor().

4/10/24

Spring Bean life cycle diagram with in a spring Bean factory Container



for ApplicationContext Container:-



When underlying Container is ApplicationContext Container then the programer can inject both BeanFactory, AppContext containers to SpringBean through Interface Injection process by Implementing both BeanFactoryAware, ApplicationContextAware interface on Spring Bean class. This operation is possible bcz the AppContext Container is extension of BeanFactory container.

for better performance it is always recomended to activate spring container and get SpringBean class objs from container by keeping code in onetime execution blocks (like constructors). It is recomended to call business methods on SpringBean class objects from repeatedly executing blocks. These two recomendations are given to achieve better performance.

Spring's Client Apps	Blocks to place & to get SpringBean class obj from container	Blocks to call business methods on SpringBean class objects.
AWT/Swing frame	constructor/static blocks	Event handling Methods
Applet/JApplet	constructor/static blocks/init()	"
Servlet	constructor/init()/static blocks	service method (-,-) / doXXX()
JSP	<%! public void jspInit() { ---- }	<% ------ ---- %>
Struts	constructor/static blocks of Struts Action class	execute(-,-,-,-) of Action class.
JSF	constructor/static blocks of ManagedBean	Business methods of ManagedBean

23/10/11

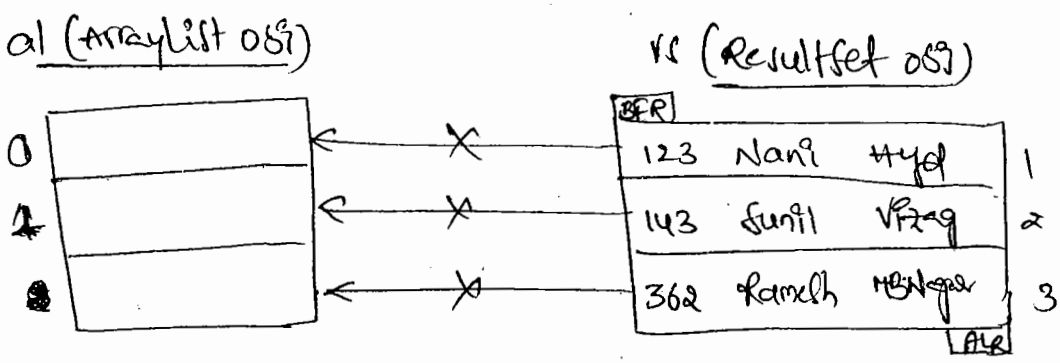
We cannot send Jdbc Resultset Object over the networks. Bcoz it is not a serializable object.

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

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

Understanding problem that is involved towards moving 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 can not be moved to single element of ArrayList, bcoz each element of ArrayList can allow to store only one Java obj at a time.



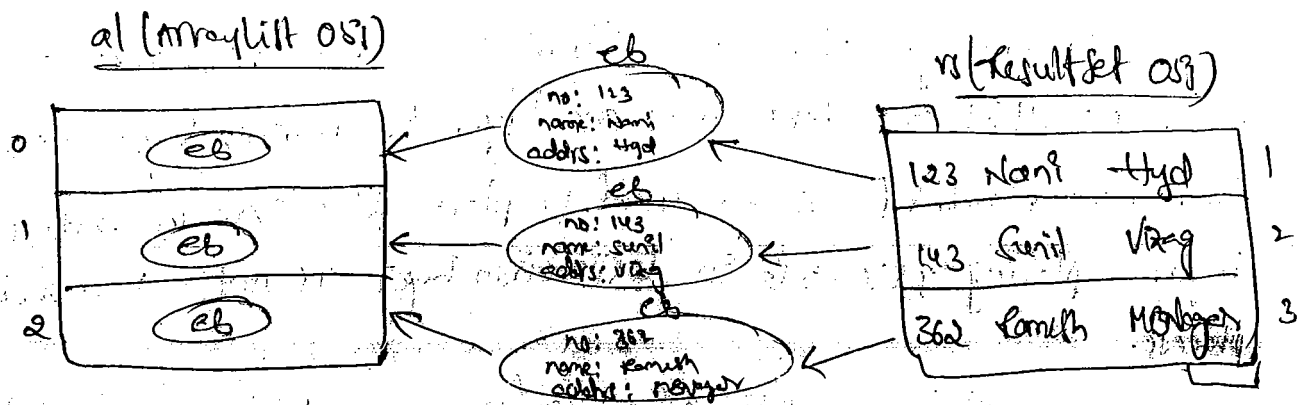
To solve the above problem read each record values into userdefined Java class obj and add that obj to elements of ArrayList. In this process this userdefined Java class is called as DTO class or VO class. Generally this called will be developed as JavaBean.

DTO class / VO class :-

```
public class EmpBean implements Serializable
{ int no;                                   ↑ (mandatory)
  String name;
  String addrs;
    // 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.setAddrs(rs.getString(3));
  // now add each DTO class obj to Arraylist element
      al.add(eb);
}
```



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.

Inorder to send collection/flow data structures over the n/w the objects added to these datastructures as elements must be taken as serializable objects.

In a web app 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 2 Arch)

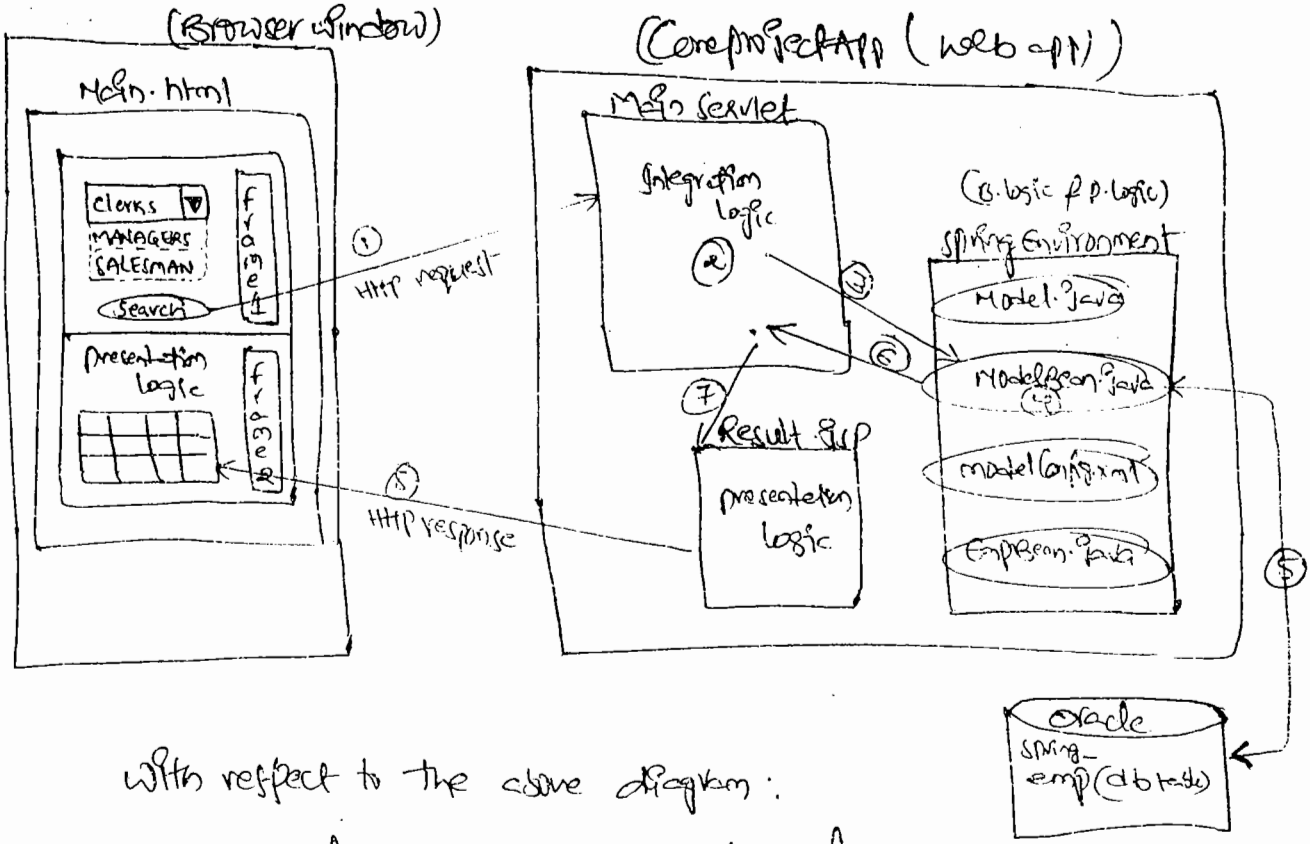
Model — Blogic and persistance logic → spring core Module
View — presentation logic → Jsp prgrm
controller — Integration/connectivity logic → servlet program.

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

A Layer is a logical partition in the appl/project representing certain category of logics.

Even though n/w 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 jdbc code in destination layer, placing jdbc 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 req → reads form data from form 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 persistance logic of springBean collects the employee records from db table based on given designation.

⑥ → Business method takes the support of DTO 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 frame2 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.

04/10/11

frame with name is called Named frame.

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

```
<form action="test URI" method="get" target="fa">
- - - - - -
</form>
```

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

```
<a href="testurl" target="fa"> go </a>
```

→ Target frame Name

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

10/10/11

factory class :-

The java class that is capable of returning one of its subclass or relavent class obj, is known as factory class.

When normal spring Bean is configured as dependent value to Bean factory then normal bean class obj will be injected to that Bean-factory

When factory springBean is configured as dependent value to Beanfactory then the property that the bean class generated resultant obj will be injected, but not the factoryBean class object itself.

spring Config file:

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

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

If ABCBean is factoryBean 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 two class as factoryBean of spring Environment then that class should implements org.sf.bean.factory.FactoryBean Interface.

This interface contains three methods.

1) public object getObject() throws Exception → returns result obj
2) public class getObjectType() → returns the class name of result obj
3) public boolean isSingleton() → returns true if singleton object is singleton otherwise false.

factoryBean is used as factoryBean for Resultant obj to exposed not directly as a normal bean Interface that will be exposed itself.

Ex-III :

Resources : Demo.java
DemoBean.java
TestBean.java - (StringBean's factoryBean)
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"/>
  </bean>
</beans>
```

TestBean.java

```
import java.sql.Date;
import org.springframework.beans.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 static void main(String args[]) throws Exception
    {
        FileSystemXmlApplicationContext ctx = new FileSystemXmlApplicationContext
                                              ("springCfg.xml");
        Demo bobj = (Demo) ctx.getBean("db");
        System.out.println("result is" + bobj.sayHello());
    }
}
```

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

factoryBeans are selflessBeans it never injects its own Bean class obj to bean property more over it always injects the generated result obj.

→ Go for factoryBean based dependency injection if bean property dependent value has to be searched in injected dynamically.

In a running Java program int type holding Integer value. String obj holds String value, ily 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
```

11/10/11

JNDI :- (Java Naming and Directory Interface)

DB s/w manages information in the form of db tables.

Mail server manages email accounts and other email messages.

ex: James Mail server, MS exchange server, SMTP One server etc.

Registry s/w can manage objects or obj references with nick names or alias names for global visibility.

ex: RMI Registry, COS Registry, JNP Registry, WebLogic Registry, Glassfish Registry, ONS Registry etc...

Java App ——uses jdbc api——→ DB s/w

Java App ——uses jndi API——→ Registry s/w

Java App ——uses JavaMail API——→ Mail server s/w

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

Registry s/w's are also called as JNDI Registries or Naming/Directory services.

Inorder to provide global visibility to certain obj or its reference then keep that one in Registry s/w.

Registry s/w

JNDI/Nick Name	Java obj/Java obj ref
st1	Student class obj
st2	Customer class obj reference
Nani	JDBC datasource obj reference
Blue	Date class obj

JNDI api means working with classes, interfaces, enums, annotations that are available in javax.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://180.44.38.46:8016/yahooApp/index.html
www.gmail.com	http://316.24.1.56:7080/GmailApp/index.jsp

Every registry s/w maintains its details in the form of key-value pairs / name-value pairs.

Every Java web server/ application server s/w gives one built in registry s/w.

WEBlogic s/w → WEB logic registry s/w

JBoss server s/w → JNP registry s/w (Java naming protocol)

WEB sphere server s/w → COS registry s/w (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 s/w's.

Registry s/w's

Naming registry s/w (or) Naming service s/w.

Directory registry s/w (or) Directory service s/w

What is the diff b/w Naming Registry s/w and Directory Registry s/w:-

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

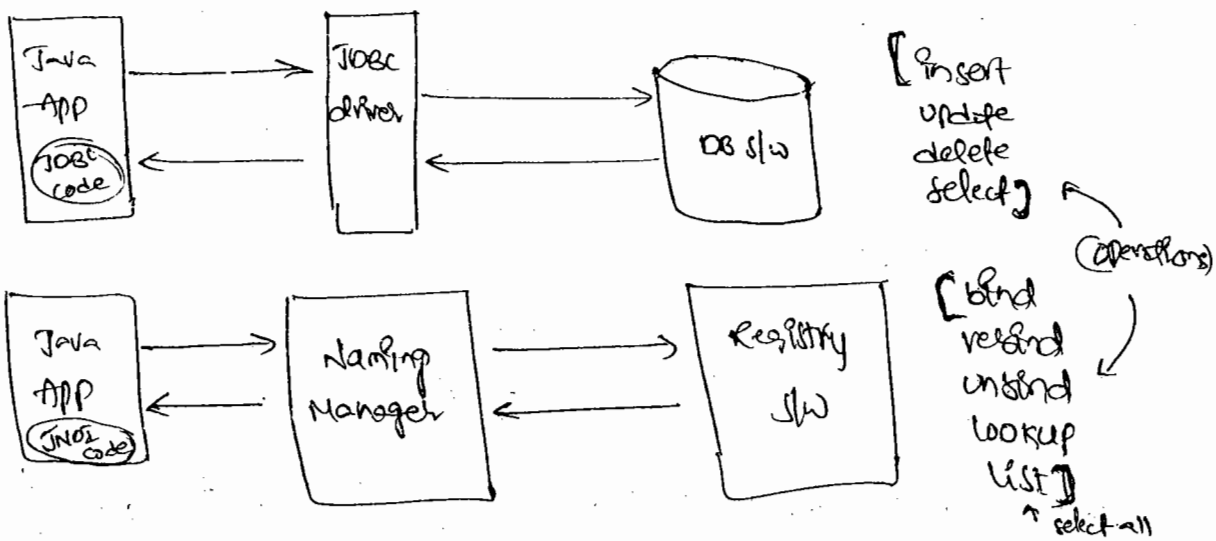
Ex:- Attendence Register, Telephone books (Maintains ph nos alphabet wise)

tech.ex:- RMI Registry, CORBA Registry etc.

Directory Registry's are enhancements of Naming Registry's where details will be maintain 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 (Infomedia, csidm 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.

NamingManager 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 persistance operations.

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

Bind → keeps obj/obj ref in registry s/w along with nick name.
Rebind → replaces existing obj/obj ref of registry s/w
Unbind → removes obj/obj ref from registry s/w
Lookup → Gets obj/obj ref from registry s/w using nick name.
List → Gets all obj/obj ref their nick names from registry s/w.

JDBC connection obj represents the connectivity b/w Java App and db s/w. 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 s/w, and provides environment to ~~Retrieve objs Regis~~ perform Bind, Rebind etc Jndi operations on Registry s/w. 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 s/w)

Note:- These details will be changed based on the Registry s/w we use.

Info:

Procedure to create our own domain server in weblogic 10.3 :-

start → programs → oracle weblogic → Quickstart → create new weblogic domain → next → generated domain ... → next → domain name: [mani] → Next → username: JavaBoss (min 8) at last pwd: JavaBoss2 (min 8 + one digit)

Done ← create ← next ← Next ← Admin server ← Next

procedure to observe the weblogic registry of Nani Domain server of weblogic :-

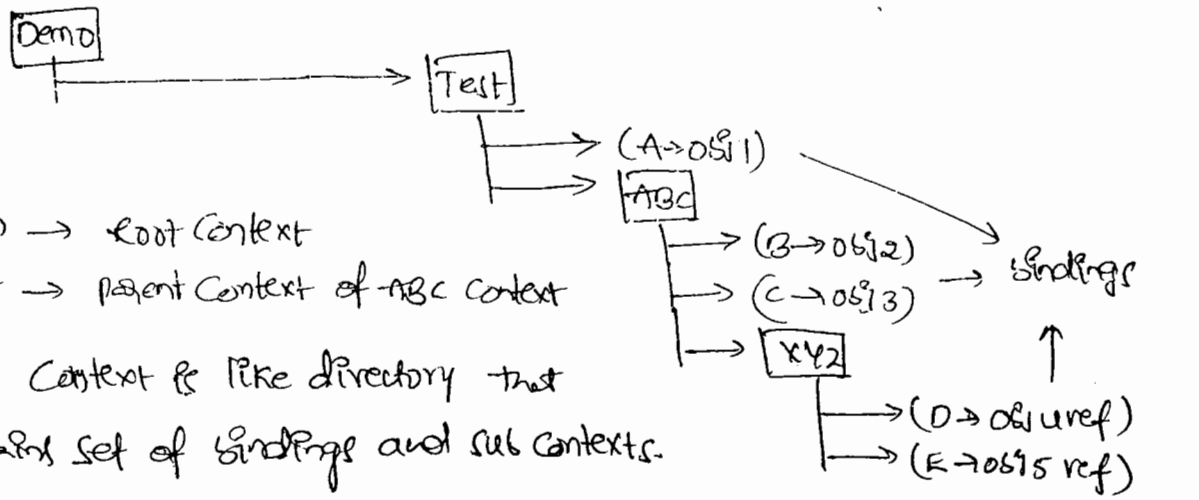
start → programs → Oracle WebLogic → User projects → Nani domain → start Admin server for WebLogic.
(This starts Nani domain server)

→ open admin console of Nani

This gives Admin Console ← (Open Browser → type http://localhost:7001/console → username: / pwd: } → login →

Environment → server → select Admin server → View JNDI Tree.

JNDI Tree of Any Registry s/w



Here:

Demo → Root Context
Test → parent Context of ABC context

Context is like directory that contains set of bindings and sub contexts.

Binding in JNDI Tree is nothing but nick name and OS1/ref that is registered with registry s/w.

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

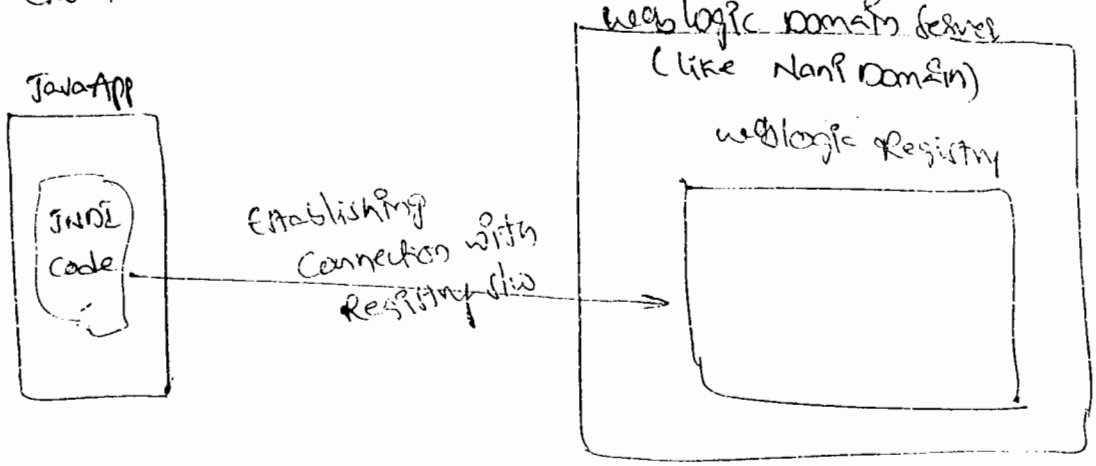
Ex:- If search operation is started from Test context for 'OS3' based on its nickname 'C' then 'Test' is called as InitialContext.

JNDI properties of weblogic registry s/w :-

InitialContextFactory class : weblogic.jndi.WLInitialContextFactory
provider url : t3://<hostname/addr>:<portno> Ex: t3://machine30:7001
principal name : ~~weblogic~~ <username> : JavaBoss
Credential : <password> : JavaBoss@

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

weblogic registry uses same IOR number, same username, pwd's 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.

* javax.naming.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 s/w:-

```
import javax.naming.*;
import java.util.*;

public class JndiConnTest
{
  psvm(String args[]) throws Exception
  {
    Hashtable<String,String> ht = new Hashtable<String,String>();
    ht.put(Context.INITIAL_CONTEXT_FACTORY," ");
    ht.put(Context.PROVIDER_URL," ");
    InitialContext ic = new InitialContext(ht); → points to root context
    if(ic==null)
      so.p("Connection is not established");
    else
      so.p("Connection is established");
  }
}
```

→ keep weblogic's Nani domain server in running mode

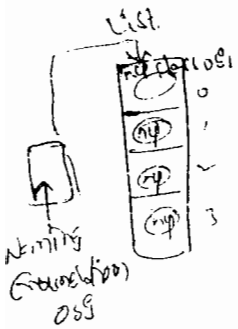
→ add weblogic.jar to classpath.

18/10/11

sample code to perform List operation:-

```
// JNDIOperationsTest.java
public class JndiOperationsTest
{
  psvm(String[] s) throws Exception
  {
    // prepare JNDI properties
```

```
InitialContext ic = new InitialContext(ht);
// list operations on root context of jndi registry.
NamingEnumeration ne = ic.list(""); // represents root context
//(or) NamingEnumeration ne = ic.list("javax"); // represents 'javax' context
//(or) NamingEnumeration ne = ic.list("javax/transaction");
//process results.
while(ne.hasMore())
{
  NameClassPair ncp = (NameClassPair) ne.next();
  // gives each bindings
  so.p(ncp.getName() + "-->" + ncp.getClassName());
}
}
}
```



gives NickName of each binding

gives className of each bound object.

* Each NamingClassPair obj can give nick 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");
so.p("lookup for today:" + d.toString());
so.p("lookup for Apple:" + ic.lookup("Apple"));
```

unbind :- `ic.unbind("today");` (reg: 57 to 58)

* For example, we can develop two more related TestTradePlaza, TestTradeJndi.java

JBoss :- (Application server) (Apache foundation) (5.x) → 1.6 JDK

Open source. Port No: is 8080 (changeable)

→ default domains: default, web, all, standard, minimal

→ Jar file that represents whole Jee api's is: Jboss-Javaee.jar

→ Built-in registry slw: Jnp registry (Port no: 1099)

To install JBoss slw 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:8080/jmx-console → service=JNDIView → list() → Invoke → Observe global JNDI Name space

properties: InitialContextFactory name: org.jnp.interfaces.NamingContextFactory
providerUrl: jnp://localhost:1099

[Establishing Connection is same as Weblogic/Glassfish (same code)]

[Jbossall-client.jar]

Glassfish: (Open Source)

type: Application Server slw
vendor: Sun MS
version: 2.x (compatable with JDK 1.5/1.6)
default port no for admin console: 4848
registry slw name: Glassfish registry
Jar file that represents whole Jee api's: Javaee.jar
The default domain is: domain1

Dnt 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 slw we use, but the code is same for all

procedure to observe JNDI Tree of Glassfish registry in Glassfish server related "domain1" server :—

Step(1) :- start the domain server

start → programs → sunmicro systems → App server 9.1 → start Default server

Step(2) :- Launch admin console of Glassfish server domain 1

open browser window → type : http://localhost:4848 → username : admin
pwd : adminadmin
login.

Step(3) :- Launch Jndi tree of Glassfish Registry.

admin console → app server → Jndi browsing → JNDI Tree

JNDI properties of Glassfish Registry :-

InitialContextFactory class Name : com.sun.enterprise.naming.
SerialInitContextFactory.

provider URL : iiop://localhost:4848/ (home/sun/Appserver/lib | domain-xml)
(server internet ORB protocol) (4848) (config files location)

principalName : admin
credential : adminadmin } optional

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...");
ht.put(context.PROVIDER_URL," iiop://localhost:4848");
ht.put(context.SECURITY_PRINCIPAL," admin");
ht.put(context.SECURITY_CREDENTIALS," adminadmin");
```

// create InitialContext obj

```
InitialContext ic = new InitialContext(ht);
```

jar files : appsrv-ext.jar (main) jaxrpc.jar (dependent to main)

15/10/11

JDBC connection pool is a factory that maintains set of readily available JDBC connection objs 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 App server / web server)

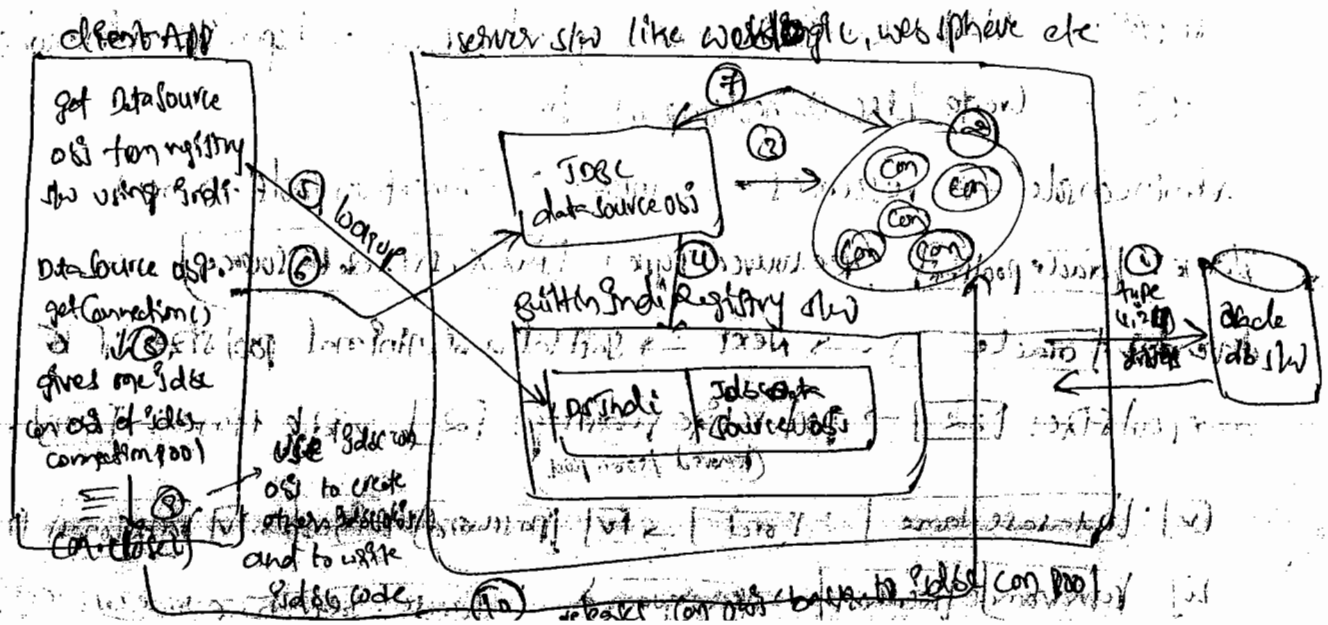
JDBC DataSource object represents server managed JDBC connection pool, and for global visibility this datasource object will be placed in registry s/w having JNDI name / nick name.

JDBC DataSource obj 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 s/w.

Ex:- JNDI connection pool for oracle means all JDBC connection objects of this pool represents connectivity with oracle db s/w.



steps:

step① & ②: In server environment, we'll use Type 1, 2 or 4 JDBC drivers to create JDBC connection pool for certain DBs also having JDBC connection objects.

③ & ④: PLTL creates JDBC DataSource 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 s/w and gets JDBC DataSource obj.

⑥ & ⑦ & ⑧: Client App calls getConnection() method on datasource obj, this call gets one JDBC connection obj of connection pool through datasource obj (server managed).

⑨ & ⑩: 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:-

step①:- keep ojdbc6.jar file in GlassfishHome/AppServer/domains/domain1/lib/ext folder.

step②:- start domain1 server of Glassfish and open the AdminConsole.

step③:- Create JDBC connection pool for oracle

AdminConsole → resources → JDBC → connection pools → new →

Name: oracle-pool | Resource Type: javax.sql.DataSource →

Driver: oracle → Next → Initial and minimal pool size: 16

max pool size: 32 → pool resize quantity: 2 → Idle time: 200 →

(Timed from pool)

Ping: DatabaseName: orcl → User: system → Password: tiger → Properties: URL

→ launch oraclepoolNani → ping → save

Step ③ create JDBC datasource representing the above JDBC Conn pool.

Admin Console → Resources → JDBC → JDBC Resources → new →

JNDI Name : DCJNDI → 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 : DCJNDI.

Standalone client application to access the JDBC connection obj 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.INITIAL_CONTEXT_FACTORY, "");
        ht.put(Context.PROVIDER_URL, "");
        InitialContext ic = new InitialContext(ht);
// get DataSource obj from jndi registry through lookup()
        DataSource ds = (DataSource)ic.lookup("DCJNDI");
            System.out.println(ds.getClass());
// get connobj from JDBC connection pool
        Connection con = ds.getConnection();
// write JDBC persistence logic
        Statement st = con.createStatement();
        ResultSet rs = executeQuery("select count(*) from emp");
```

```
if(rs.next())
    sop("No. of records in emp table :"+rs.getInt(1));
// release jdbc con obj back to jdbc con pool
    con.close();
}
```

Add following jar files to classpath before executing the above app.

1. appserv-rt.jar
2. javaee.jar
3. appserv-admin.jar

} available in Glassfishthree/appserv/lib

4. ImgSimsra.jar → available in Glassfishthree/appserver/lib/Install/applications/Imsra
5. ojdbc6.jar

While working with Glassfish registry we can create InitialContext obj without JNDI properties related map object. (not recommended)

```
InitialContext ic = new InitialContext();
```

The above statement based app execution you should add javaee.jar, appserv-rt.jar files to classpath.

procedure to create JDBC conpool for oracle JDBC datasource in weblogic 10.3 server:-

Step 0: start the Nani domain server of weblogic, and open its admin console.

Step 1: create JDBC connection pool for oracle along with JDBC DataSource.

adminConsole → services → jdbc → DataSources → new → name: mydsr

JNDI Name: DsJndi database Type: Oracle, driver: Oracle's Driver (Thin) Service connections

→ Next → DBName: orcl HostName: Localhost : portNo: 1521

DBUserName: scott pwd: tiger confirm pwd: tiger → next →

Test Configuration → Next → ☑ select admin server → finish

Note: Having defined as Jndi name the above created datasource obj that represents Jdbc connection pool for oracle will be registered automatically with weblogic registry.

Step③: specify additional parameters to the above created con pool.

Admin Console → services → Jdbc → dataSources → mypos → connection pool Tab →

Initial Capacity: [5] Max Capacity: [50] Capacity Increment: [2] →

advanced → Shrink Frequency: [900]
(idle time)

Writing standalone Java App to access the JDBC con obj of the above JDBC connection pool:—

poolTest.java:

same as previous app, but Jndi properties should be change to weblogic

```
Hashtable ht = new Hashtable();
ht.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitial...");
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, Jee technologies by taking care of the common activities of app development. So in spring environment the programmer just need to worry about application specific operations

In spring Jndi environment → JndiTemplate

In spring JDBC/JDBC env → JDBC Template

In spring ORM env → Hibernate Template, JDO Template, Toplink template.

Spring JNDI:

1. provides abstraction layer on plain Jndi program.

2. Use JndiTemplate class to take care of the common coding operations of Jndi programming.

3. ~~Takes care of exception handling, and also converts Checked Exceptions into Unchecked Exceptions using exception Rethrowing concept.~~

4. Overall it simplifies Jndi programming without using plain Jndi API.

~~Spring Jndi code execution internally takes support of the plain Jndi code and converts plain Jndi code generated checked exceptions into Unchecked Exceptions.~~

```
public void m1()
{
  try{
    ---
    ---
    ---
  }
  Catch(IOException ie)
  {
    throw new NumberFormatException("Exception Raised");
  }
}
```

This was not there in Jndi but there in JDBC (Spring based) Templates

while calling m1() we need not handle IOException, bcoz we can handle the Unchecked exception NumberFormatException optionally.

18/10/11

org.sf.jndi.JndiTemplate class supplies all basic methods to perform jndi operations but it doesn't supply list() to perform listing operation so this list() functionality can be implemented explicitly through JndiCallback interface.

The spring even allows to perform dependency injection on a class that acts as client app by activating spring container.

for example refer app 12 of material : page : 58-60

while working with JNDI Template class the InitialContextFactory classname related jndi property must be taken as "java.naming.factory.initial" and must not be taken as "JNITIAL_CONTEXT_FACTORY"

Note:- The JNDITemplate class spring jndi environment is not converting checked exceptions generated underlying plain jndi through unchecked exceptions, so exception handling is mandatory in spring jndi environment.

→
```
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent we)
    {
      ---
      ---
    }
});
```

In the above statement in the paranthesis of addWindowListener() method call 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 programing supplies CallBacks interfaces allowing the programmer implementing certain functionality using underlying plain technology API when that functionality is missed in template class

The JndiTemplate class gives execute() to execute the functionality defined in CallBacks Interface Implementation.
(JndiCallback)

Ex App on JNDI CallBacks Interface Implementation to provide list() method functionality by using Naming JNDI.

DemoCfg.xml : → same as previous App (app12)

JndiTemplateTest.java

```
import org.sf.context.support.*;
import org.sf.jndi.*;
import javax.naming.*;
public class JndiTemplateTest
{
    static JndiTemplate template;
    public void setTemplate(JndiTemplate template)
    {
        this.template = template;
    }
public static void main(String args[]) throws Exception
{
    FileSystemXmlApplicationContext ctx = new
                FileSystemXmlApplicationContext("DemoCfg.xml");
    template.execute(new JndiCallback() {
        public Object doInContext(Context ctx) throws
                                    NamingException
        { // list() with plain Jndi.
        NamingEnumeration ne = ctx.list("");
        while (ne.hasMore())
        { NamingClassPair np = (NamingClassPair)ne.next();
          System.out.println(np.getName() + " " + np.getClassName());
```

→ represents JndiContext object of Jndi env.

In the above ex template.execute() is called having anonymous class obj as argument value and that anonymous class implements JdbcCallbacks interface.

19/10/15

In spring-application we can works with 3 types of jdbc conpool.

1. Springs builtin connection pool using DriverManagerDataSource class or SingleConnectionDataSource class.
2. Third party managed connectionpool like Apache DBCP or C3P0
3. web/application server managed jdbc connectionpool

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 connectionpool is not suitable in realtime projects.

Q: what is the jdbc con pool that you have used in ur spring projects?

Ans:- If your spring proj is standalone environment project then use third party slw based connection pools like Apache DBCP or C3P0.

If your spring project is deployble application in the servers like webapplications then use server managed jdbc connectionpool.

→ The TomcatHome\lib\tomcat-dbcp.jar file represents Apache DBCP slw, this jar file contains org.apache.tomcat.dbcp.dbcp.BasicDataSource class whose dataSource obj represents one jdbcConnectionPool.

Imp Bean properties of BasicDataSource class:

1. username
2. password
3. url
4. driverclassname
5. initialSize
6. maxActive

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;
   // setXXX() 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;
      }
}
```

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:ord"/>
      <property name="username" value="scott" />
      <property name="password" value="tiger"/>
      <property name="initialSize" value="2"/>
   </bean>
   <bean id="ds" class="DemoBean">
      <property name="ds" ref="dbcp" />
   </bean>
</beans>
```

DemoClient.java :-

```
import org.sf.context.support.*;
public class DemoClient
{
   psvm(String args[])
   {
      ClassPathXmlApplicationContext ctx = new
                          ClassPathXmlApplicationContext("Democfg.xml");
      Demo bobj = (Demo) ctx.getBean("ds");
   } sop("Result Salary is:" + bobj.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) jdbc14.jar

* The HB 3.2.5 Home/lib/c3p0-0.9.1.jar file represents c3p0 s/w. This jar file contains com.mchange.v2.c3p0.ComboPooledDataSource class. This class obj is JDBC DataSource 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="DemoBean">
    <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) ojdbc14.jar.

20/10/11

org.sf.Jndi.JndiObjectFactoryBean is a predefined factory bean class that lookup for a object in Jndi registry and exposes / injects found obj from Jndi Registry for our Bean class properties as dependent value.

Since it is a factoryBean when its bean id is configured as dependent value to our Bean class property then JndiObjectFactoryBean class obj will not be injected to that property but the object that is gathered from Jndi Registry will be injected to our Bean class property.

The important two properties of JndiObjectFactoryBean class are 1. JNDIName (for lookup) 2. JndiEnvironment. (for specifying Jndi properties)

The JndiObjectFactoryBean class is popularly used to inject the dataSource obj gathered from registry s/w to our Bean class properties.

Example Application:—

Demo.java, DemoBean.java, DemoClient.java are same as previous app.

Democfg.xml:—

```
<beans>
  <bean id="jofb" class="org.sf.Jndi.JndiObjectFactoryBean">
    <property name="JndiName" value="DsJndi" />
    <property name="JndiEnvironment">          → Jndi properties of weblogic registry
      <props>
        <prop key="java.naming.factory.Initial">
          weblogic.jndi.WLInitialContextFactory </prop>
        <prop key="PROVIDER_URL">t3://localhost:7001</prop>
      </props>
    </property> </bean>
  <bean id="db" class="DemoBean">
    <property name="ds" ref="jofb" /> </bean>
</beans>
```

where 'ds' refers jdbc DataSource obj gathered from Registry s/w such not 'jndi'.

Jar files required : (1) spring.jar (2) commons-logging.jar (3) weblogic.jar (for jndi properties).

It is not recomended to use server managed jdbc ConnectionPool in standAlone / Desktop spring / Java apps. When your app is deployable and executable in the server then only it is recomended to use server managedConnectionpool 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 programing and simplifies the process of developing persistance logic by specifying JdbcTemplate class. This class takes care of common Activities of jdbc program, concentrate on application specific activities like preping sql query, executing query, gathering and processing the ResultSet.

plain jdbc

common
1. Load jdbc driver class and register with DriverManager service.
2. Establish connection with db s/w.
3. Create jdbc statement obj

app specific
4. Prepare query, send and execute that sql query in db s/w
5. gather query results and process the results.

common
6. Take care of Exception Handling
7. Take care of Transaction Management if necessary.
8. close jdbc, objs along with jdbc connection.

The code of common activities will always remain same more or less in all applications. where as the app specific activities code will change from application to application.

In spring Jdbc / DAO environment The org.sf.jdbc.core.JdbcTemplate class takes care of common activities / work flow of Jdbc programing and makes programer 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 sw.
3) Gather sql query results and process the results.

JdbcTemplate class internally performs :-

1) Common activities of Jdbc programing.
2) Exception Handling and Transaction Management.
3) Converts plain Jdbc generated checked exceptions to unchecked.

* The spring DAO / Jdbc persistance logic internally uses plain Jdbc and converts plain Jdbc generated checked exceptions to unchecked exceptions by using exception re-throwing concept.

DAO is a design pattern which is nothing but a Java class / comp in our app that separates persistance logic from other logics of the app and makes the persistance logic as flexible logic to modify.

Using springDAO or plain Jdbc or plain Hibernate or springORM module we can develop the persistance logic of DAO class.

Using springDAO module we can develop persistance logic with or without implementing DAO design pattern.

In plain Jdbc programming the select query gives JdbcResultSet obj and it is not serializable obj bydefault. So we cannot send this obj over the network.

In spring DAO environment we got diff varietics of query(), queryXxx() in JdbcTemplate class to get select query results in programer choise 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)
</bean>
<bean id="temp" class="org.sf.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 as sql select query results.

select count(*) from emp;

select * from emp where empno = 7499;

b) queryforMap() : to get one record from Hashtable by executing query

select * from emp where empno = 7499;

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 plsql procedures/functions of db sw.

Both JDBC and spring DAO persistance logic are db sw dependent persistant logics, bcoz they use the db sw dependent sql queries.

Ex App :- Demo.java, DemoBean.java, democfg.xml, DemoClient.java

While developing this app by using myEclipse IDE choose following libraries, 1) spring Core libraries 3) persistance jdbc libraries.

2) persistance Core libraries

Demo(fg.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.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dbcp" />
  </beans>
  <bean id="ds" class="DemoBean">
    <property name="jt" ref="template" />
  </bean>
</beans>
```

Demo.java :—

```
import java.util.*;
import java.sql.*;
public interface Demo
{
    public long fetchEmpSal(int eno);
    public int countEmp();
    public Map fetchEmpDetails(int eno);
    public List fetchEmpDetails(String job);
    public RowSet fetchEmpDetails(long deptno);
    public boolean registerEmp(int eno, String ename, String job, double
    public boolean hikeSalary(int eno, int percentage);          salary);
    public boolean fireEmp(int empId);
}
```

DemoBean.java :—

```
import java.util.*;
import javax.sql.RowSet;
import org.springframework.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 same
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;
    }
```

empNo	7698	0
ename	Allen	1
job	salesman	2
sal	1600	3
mgr	7698	4
hiredate	20-feb-81	5
comm	300	6
deptno	10	7

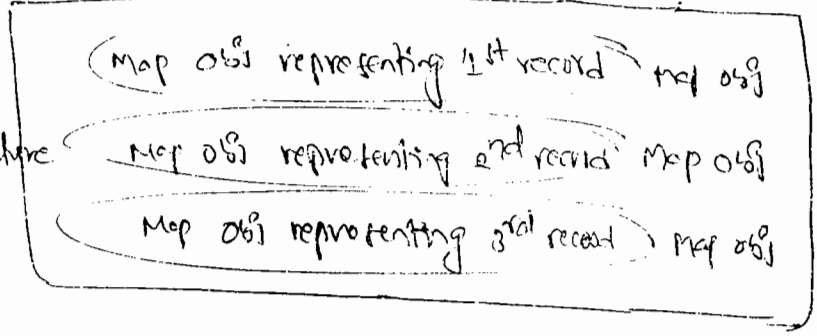
Column names as key

m map data structure column values as values.

public List fetchEmpDetails(String job)

{ List l = jt.queryForInt("select * from emp where job=?",
new Object[]{job});

} return l;

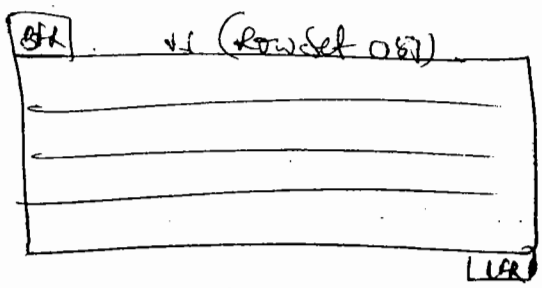


22/10/11

public Rowset fetchEmpDetails(long dept)

{ Rowset rs = jt.queryForRowSet("select * from emp
where deptno=?", new Object[]{new long(dept)});

} return (Rowset)rs;



public boolean registerEmp(int eno, String ename, String job
int sal)

{ int res = jt.update("Insert into emp(empno, ename, job, sal)
values(?,?,?,?)", new Object[]{new Integer(eno),
ename, job, new Integer(sal)});

if(res==0)
return false;
else
return true;

}

```
public boolean fireEmp(int salrange) {
    int res = jt.update("delete from emp where sal>=",
        new Object[] { new Integer(salrange) });
    if(res==0)
        return false;
    else
        return true;
}

public boolean hikeSalary(int eno, int percentage)
{
    int sal = jt.queryForInt("select sal from emp where
        empno=?", new Object[] { new Integer(eno)});
    float newSal = sal + (sal * (percentage/100));
    int res = jt.update("update emp set sal=?", where
        empno=?", new Object[] { new float(newSal),
            new Integer(eno) });
    if(res==0)
        return false;
    else
        return true;
    }
}
```

DemoClient.java

```
public class DemoClient
{
    psvm (String args[])
    {
        ClassPathXmlApplicationContext ctx =
                new (ClassPathXml)ApplicationContext("Demo(fg.xml");
        Demo bobj = (Demo) ctx.getBean("ds");
s.o.p("Count of Emps" + bobj.countEmps());
s.o.p(" 7499 Emp Exists" + bobj.fetchEmpExists(7499));
s.o.p(" Emp Registred ?" + bobj.registerEmp(100, "Rani",
                                "clerk", 15800));
s.o.p(" 7499 Emp Details are " + bobj.fetchEmpDetails(7499));
s.o.p(" clerk desg Emp details are" + bobj.fetchEmpDetails
                                ("clerk"));
s.o.p(" 10th department employee details are");
        SQLRowSet rs = bobj.fetchEmpDetails(long 10);
        while(rs.next())
        {
            s.o.p(rs.getInt(1) + " " + rs.getString(2) + " " +
                    rs.getString(3));
        }
s.o.p(" 7499 Emp salary hiked ?" +
                bobj.hikeSal(7499, 10));
    }
```

To develop persistence logic in DAO class by using spring DAO module then we need to inject JDBCTemplate class obj to that DAO class.

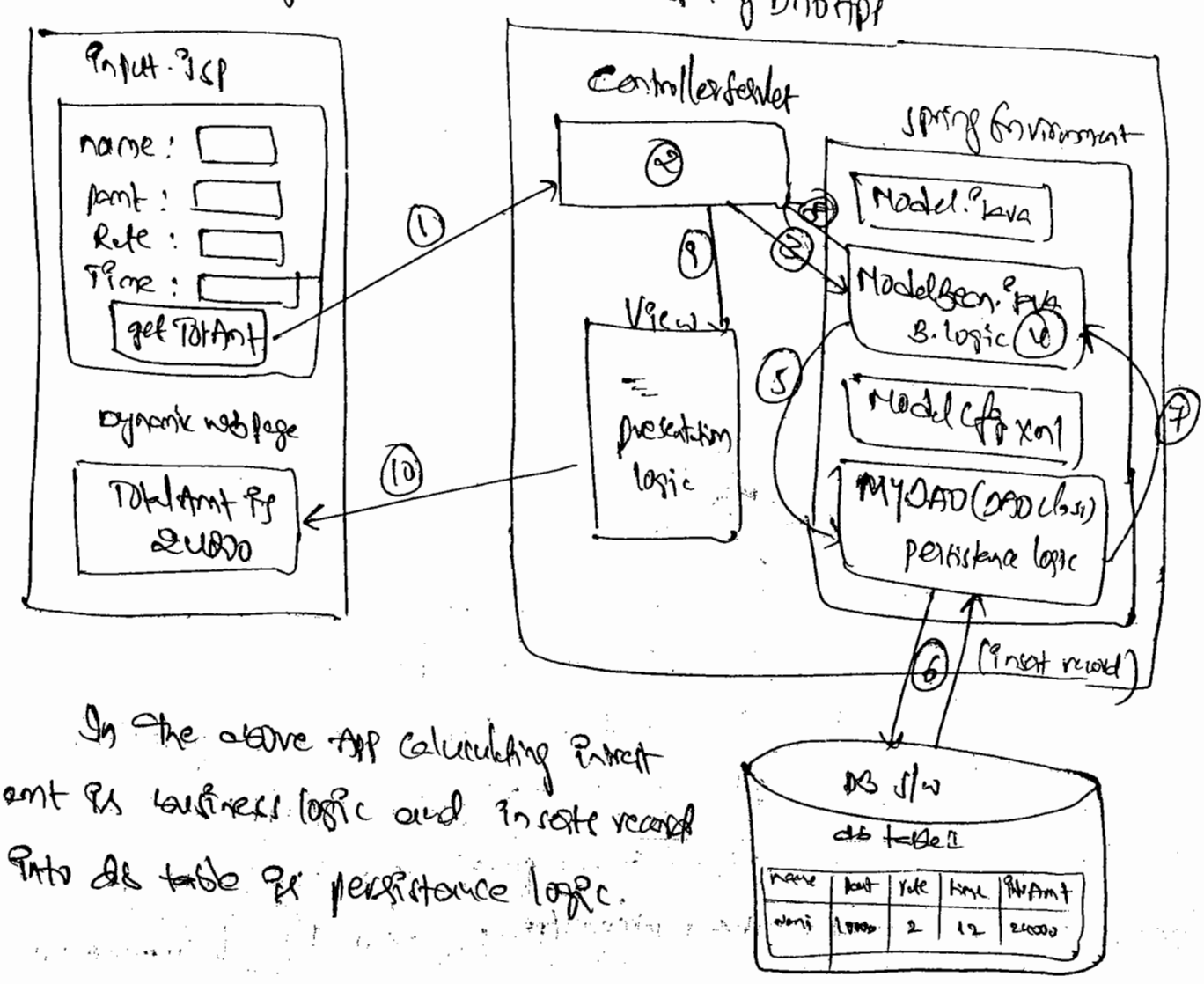
EXAPP :-

① MVC Arch Based web Application.

Model → SpringBean + DAO class
Controller → servlet
View → Jsp

② Use SpringDAO module based DAO class having persistence logic.

③ Use server managed JDBC connection pool.



In the above App calculating interest is business logic and insert record into db table is persistance logic.

procedure To develop above app using myeclipse.

step ①: create web proj having name springDaoApp.

step ②:- Add spring Capabilities to the project having

☑ spring 2.5 persistance,

☑ spring 2.5 core library,

☑ spring 2.5 persistence jdbc library and also

choose → next → model.cfg.xml or spring.cfg file.

step ③ Add weblogic.jar file to the buildpath of project.

Note:- The jar files added to myComputer's classpath are not visible and accesible to the proj in IDE, so we must add like above.

24/10/11

step ②: create spring_customer table in oracle db s/w.

> create table spring_customer (ename varchar(15), psal number(8), vote number(4), time number(4), intrest number(4));

step ③: Develop DAO class in src folder (myDao.java)

```
import org.st.jdbc.core.JdbcTemplate;

public class myDao
{
    JdbcTemplate jt;
    public void setJt(JdbcTemplate jt)
    {
        this.jt = jt;
    }
    public boolean insertInto(String ename, int psal, int vote,
                              int time, int intrest)
    {
        int res = jt.update("insert into spring_customer
            values(?,?,?,?,?)", new Object[]{ ename,
            new Integer(psal), new Integer(vote),
            new Integer(time), new Integer(intrest)});
```

```
        if(rel == 0)
            return false;
        else
            return true;
    }
}
```

Step ④ : Add spring interface, spring bean class to the project.

```
public interface MyBean
{
public int calcIntrAmt(String ename, int pamt, int time,
                       int rate);
}
```

Step ⑤ :

```
public class MyBeanBean implements MyBean
{
    MyDao dao;
    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 persistance logic of DAO class
        dao.insertInto(ename, pamt, time, rate);
        return intrAmt + pamt;
    }
}
```

Step ⑥ : Write following configurations in spring config file.

MyBeanCfg.xml :-

```
<beans>
    <bean id="jofs" class="org.st.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="DsJndi"/>
    </bean>
    <bean id="template" class="org.st.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="jofs"/>
    </bean>
```

```
<bean id="dao1" class="mypack">
    <property name="jt" ref="template" />
</bean>
<bean id="ms" class="modelBean">
    <property name="dao" value="dao1" />
</bean>
</beans>
```

Note: If the application wants to communicate with jndi registry of server replied outside the server then there is no need of supplying 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="fname" /> <br>
principalAmount: <input type="text" name="pamount" /> <br>
    Rate : <input type="text" name="rate" /> <br>
    Time : <input type="text" name="time" /> <br>
        <input type="submit" value="getTotalAmount" />
</form>
```

Step②:- add Controller servlet to the project src folder.
Right click on src → add → add servlet → Controller servlet → select doget, dopost → next → mapping url: /controller → finish.

```
public class ControllerServ extends HttpServlet
{
    Model bobj = null;
    public void init()
    {
        ClassPathXmlApplicationContext ctx = new
                ClassPathXmlApplicationContext("model cfg.xml");
        // get SpringBean obj from Container
        bobj = (Model) ctx.getBean("mb");
    }
    public void doGet(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException
    {
        // Read form data
        String name = Integer.parseInt("pname");
        int amt = Integer.parseInt(req.getParameter("pamt"));
        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 SpringBean
        int totalAmt = bobj.calcIntrAmt(name, amt, time, rate);
        // keep the result in request attribute
        request.setContentType(
                request.setAttribute("attrd", 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()
    {
        bobj = null;
    }
}
```

Step(9): Add Result.jsp to the WebRoot folder of the project.

```
<b> The Total Amount : </b> <%=request.
                        getAttribute("dtf") %>
```

Step(10):- Configure spring batch domain of web logic with myeclipse

window menu → preferences → myeclipse → servers →
weblogic → weblogic 10.x → enable →

BEA Home Directory : [C:\bea] [browse]

WL Installation directory : [ ] [ ]

Admin UserName : [ ]

Admin password : [ ]

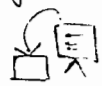
execution domain : [C:\bea\weblogic10.3\user_projects\domains] [browse]

→ Apply → OK

step(4): select server icon  in the Tool bar →

weblogic 10.3 → start

* deploy the project in weblogic server

Go to deploy icon of the tool bar  project: SpringDaoApp

→ add server: weblogic 10.x → finish → OK

* open browser window: @ symbol and type URL.

Procedure to Configure Glassfish server :-

Window → preferences → myEclipse → 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 \ domain1 \ config \ domain.xml file related <http> tag port attribute.

25/04/14

Jdbc supports only positional parameters as part of their queries. where as springDAO allows the programer to specify named parameters and positional parameters in sql queries.

To work with named parameters we can use NamedparameterJdbc Template class

The NamedparameterJdbc Template 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 connectionpool. This class super class is DriverManagerDataSource class

The connection obj of DMDS related connectionpool will be closed automatically at the end of the execution but, while working with SCDS we need to close connection obj explicitly bcz it will not be closed automatically.

for example app on Namedparameters refere app@ of material. (Here No springContainer support is taken in app development)

JdbcTemplate 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 springDAO environment.

If certain functionalities are directly not possible with JdbcTemplate class then springDAO allows us to implement that

functionality through Callbacks Interfaces Implementation. During this Implementation we can take the support of plain Jdbc API.

Some Imp Callbacks 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 bootfg.

The queryForObject() of SpringDAO environment allows the programer to gather select query execution result in programer 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 user-defined Java class obj

Use ResultSetExtractor callback Interface to copy (multiple selected) records of resultset obj to programer choice object like Arraylist

Use preparedStatementCreator callback Interface to execute given sql queries through prepared statement 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 sql query without parameters by using JdbcpreparedStatement obj then pass "null" as the second argument value. (queryForInt(query, null) → uses preparedStatement)

20/1/14

procedure to develop spring App (1st App) by using NetBeans IDE

step①: Create Java project in NetBeans IDE having name TestProj.

step②: Add spring libraries to the project.

Right click on libraries → add library → spring fw 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 (project) → next → ⊙ DTD-constraint → next

spring DTD Beans 2.0 [—] Document root: beans → finish → write

the step⑥ following code in DemoCfg.xml

```
<bean id="ds" class="DemoBean">
  <property name="msg" value="hello"/>
</bean>
```

(as a recommended to close directive manually in DemoCfg.xml file)

Step ⑥: Add client Application to the project

Right click on project → new → Java class : DemoClient

```
Public class DemoClient
{
    Psvm (String args[])
    {
    ClassPathXmlApplicationContext ctx = new CPXAC("Demo(ff.xml");
        Demo boss = (Demo) ctx.getBean("dis");
    }   } S.o.p(boss.sayHello());
```

(Psvm + tab + space
↓
Psvm (String args[]))

Step ⑦: Run the client Application

Right click on DemoClient.java source code → Run file

27/Jan

To centralize persistance logic or to make persistance logic as reusable logic by keeping those logics on db s/w then take the support of PL/SQL procedure / function.

Spring DAO environment allows the programer to call PL/SQL procedure & function.

The subclass object of org.sf.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 value of PL/SQL procedure/function.

↓ represents the parameter value of PL/SQL procedure/function

Some Inherited methods of StoredProcedure class.

compile() → Compiles the query that calls PL/SQL procedure/function.

setFunction (true/false) → represents weather the subclass object of SP class points to PL fn/Proc

setParameter(parameter) → adds parameters that are registed with jdbc datatypes to the subclass obj of SP class.

By using SqlOutParameter class object we can register out parameters of pl/sql proc/fun with JDBC Types / SQL Types.
Ily By using SqlParameter class we can register the in parameters.

* for example app on spring DAO based pl/sql proc/fun refer app (10)

Conclusion on : spring DAO :—

spring DAO persistance logic db s/w dependent, bcoz it uses db s/w specific sql queries.

All ORM s/w's like Hibernate can be used to develop db s/w independent persistance logic.

spring ORM module allows the programer to develop db s/w independent persistance logic by providing Abstraction layer on all ORM s/w's.

Java Mail API :—

- DB s/w is responsible to maintain data
- Registry s/w is responsible to maintain objs and objs references.
- Mail server s/w's are responsible to maintain email accounts and email messages.

ex: James Mail server, SMTP one server, Lotus Notes, Microsoft Exchange.

Note: db s/w's 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 pkgs:

Javax.mail. pkg (is part of JEE module)
Javax.mail.internet pkg
Javax.activation. pkg (JAF)

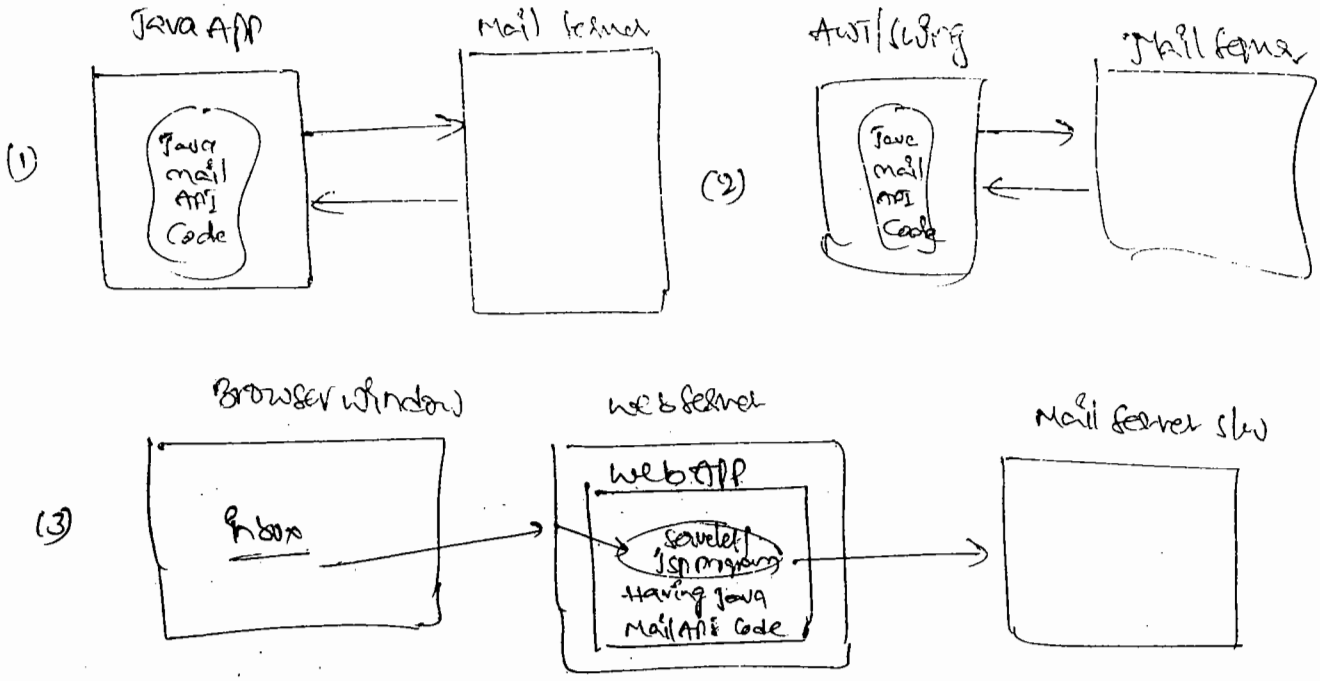
Java App ——JDBC API——→ DB s/w

Java App ——JNDI API——→ Registry s/w

Java App ——Java mail API——→ Mail servers

28/10/11

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 pc to another pc.

Microsoft Out Look, Microsoft Out Look 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 b/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 slw 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 collect the mails from Inbox and sends to mail client Machine once Mail client reads the email msg from Inbox.

IMAP based Incoming servers maintain email messages forever even after they are read by mail clients.

In company level Intranet 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 programers will get the assignment through mails, but these mails related email accounts will be managed in IMAP based Incoming server.

James : (open source)

Type : Java based Mail server slw
version : 2.2 (compatable with jdk1.4+)
vendor : Apache slw foundation [slw: www.apache.org]
(zip file)
ports : for admin console : 4555
Incoming server (POP3) : 110
outgoing server (SMTP) : 25
News server (NNTP) : 119

To Install James Mail Server s/w 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 gmail maintains email messages even after they are read by mail client.

Procedure to create Gmail accounts in 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 a/c's)

adduser Nani 002

adduser [redacted] reddy

listusers : display the list of all users

del user Nani : deletes 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 s/w, bcz it is given as a specification containing set of API's, rules and guidelines to develop web server, app server s/w's. So every web and app server s/w supplies jar files representing JEE API's.

→ In weblogic server the weblogic.jar represents jee.apis (but not Java Mail api).

→ In Glassfish server javaee.jar represents jee api's (with JavaMail API)

→ In JBoss server jboss-javaee.jar represents jee api's (without Java mail API) (In this it is as mail.jar)

28/10/11

A session obj of JavaMail API based coding represents connectivity b/w Java app and Mail server and this session obj is no way related with HttpSession obj of servlet programing, session obj of JSP programing and the session obj of Hibernate programing.

To create session obj of JavaMail programing 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 } outgoing
3) Mail.smtp.port }
4) Mail.pop.host } incoming server
5) Mail.pop.port }

While creating session obj of JavaMail programing 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/javaee.jar file to the classpath

Step③: Develop Java application having JavaMail API code as shown below.

ConnTest.java :

```
public class ConnTest
{
  public static void main(String args[]) throws Exception
  {
    Properties p = new Properties();
    p.put("mail.transport.protocol", "smtp");
    p.put("mail.smtp.host", "localhost");
    p.put("mail.smtp.port", "25");
    Session ses = Session.getInstance(p);
                          ↓
                    (factory method)
    if(ses==null)
      System.out.println("Connection not established");
    else
      System.out.println("Connection established");
  }
}
```

Step④:- Compile and execute the application.

→ Javax.mail.internet.InternetAddress class obj can represent one email a/c / email address.

To create email message programatically instantiate ~~JavaMail~~ Javax.mail.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 a/c to another email accounts.

for example app on plain Javamail API refer app: (13)

31/10/11

In order to delete the email messages of Inbox then mark them for deletion and close the Inbox with flag "true".

when Inbox is closed with "true" flag all marked messages will be deleted.

myfbox.close(true);

when Inbox is closed with flag "false" then email messages will not be deleted eventhough they are marked for deletion.

myfinbox.close(false);

SpringMail :—

Spring mail provides abstraction layer on the Javamail 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 Readmail, delete mail operations.

Spring Mail API means working with org.sf.mail package, org.sf.mail.javamail package.

org.sf.mail package gives spring's own ~~infrastructure~~ infrastructure for dealing with Mails.

org.sf.mail.javamail → spring internally uses Javamail API for dealing with mails.

org.sf.mail.SimpleMailMessage allows us to design simple email msg (without attachment) where as org.sf.mail.javamail.MimeMailMessage class allows us to deal with more complex email messages like message with attachment.

While working with springMail operations we can deal with spring's dependency Injections by preparing sender obj's and Mail messages. (for example

org.sf.mail.javamail.JavaMailSenderImpl class obj can be used to send email messages from one a/c to another account. It underlyingly uses Javamail-api support.

01/11/11 To add attachment to the body of Email message prepare that email msg body with multiple parts and each part can contain text data or image data attached file.

Ex on sending email with attachment refer @ APP① of 01/11/11

To send Email msg with attachment we need to use the call backs 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 @ APP② 01/11/11

The outgoing server details of GMail.com mail server:-

mail.smtp.host ⟶ smtp.gmail.com

mail.port ⟶ 465

mail.smtp.[illegible] ⟶ 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 above said service obj

03/11/11.

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: ~~app~~ on gmail.com refer Handout given on 02/11/11. ✓

* procedure to configure gmail with microsoft outlook :-

step①: login to gmail.com web site'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: [hnr002@gmail.com] pwd: [141443] re-pwd: [141443] → next → finish.

* To compose Mail through outlook.

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 files tab → settings → change pwd → new pwd: [nani002] → verify new pwd: [nani002] → OK → OK

➡ for (03/11/11) to (07/11/11) data refer 01st page to (13)th page of xerox (Books side)

Step (16) - Add form page input.html to the webroot folder of project

Right click on webroot folder → html → [input.html]

```
<form action = "controller" method = "get">
  <b> Name : </b> <input type = "text" name = "tname" /><br/>
  <b> Amt : </b> <input type = "text" name = "amt" /><br/>
  <b> Rate : </b> <input type = "text" name = "rate" /></b>
  <b> Time : </b> <input type = "text" name = "time" /><br/>
    <input type = "submit" value = "getTotalAmt" />
</form>
```

Step (17): Add controller servlet to the src folder of the project.

Right click on project → servlet → [ControllerServlet] → select init(), destroy(), doGet(), doPost() methods → next → servlet URL pattern: [/controller] → finish

```
Public class ControllerServlet extends HttpServlet
{
  Model bobj = null;
  public void init()
  {
    ClassPathXmlApplicationContext ctx = new ClassPathXmlApplication
                                         Context("springcfg.xml");
    bobj = (Model) ctx.getBean("mb");
  }
  public void doGet(HttpServletRequest req, HttpServletResponse res)
                throws ServletException, IOException
  {
    // Read form data
    String name = request.getParameter("tname");
```

```
int amt = Integer.parseInt(request.getParameter("famt"));
int rate = Integer.parseInt(request.getParameter("rate"));
int time = Integer.parseInt(request.getParameter("time"));

    int totamt = bobj.calcIntAmt(name, amt, rate, time);
// send result to result.jsp page
    request.setAttribute("res1", totamt);
// forward result to result.jsp
RequestDispatcher rd = request.getRequestDispatcher("result.jsp");
    rd.forward(req, res);
  }

public void doPost(-,-) throws (IE, IOException)
{
    doGet(req, res);
}

public void destroy()
    { bobj = null;
    }
}
```

step (18) add result page to the webroot folder of the project.

Right click on web root folder → new → jsp → result.jsp

```
<b> The result is : </b> <%= request.getAttribute("res1") %>
```

step (19): Start Tomcat server from MyEclipse IDE

step (20): deploy web app in Tomcat server from MyEclipse

step (21): test the web app from the browser window of myeclipse

http://localhost:2020/springHBAPP8/input.html

Spring JEE Module:

Spring JEE module is given having the following facilities.

1) gives ApplicationContext container

2) Gives Spring Jndi providing Abstraction layer on plain Jndi

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 applications.

i) Spring RMI provides Abstraction layer on plain RMI.

ii) Spring EJB " " plain EJB.

iii) Spring WebServices " " plain WS Services.

iv) Spring Hessian, and Burlap " " plain Hessian, Burlap.

6) Gives its own direct distributed technology called Http Invoker.

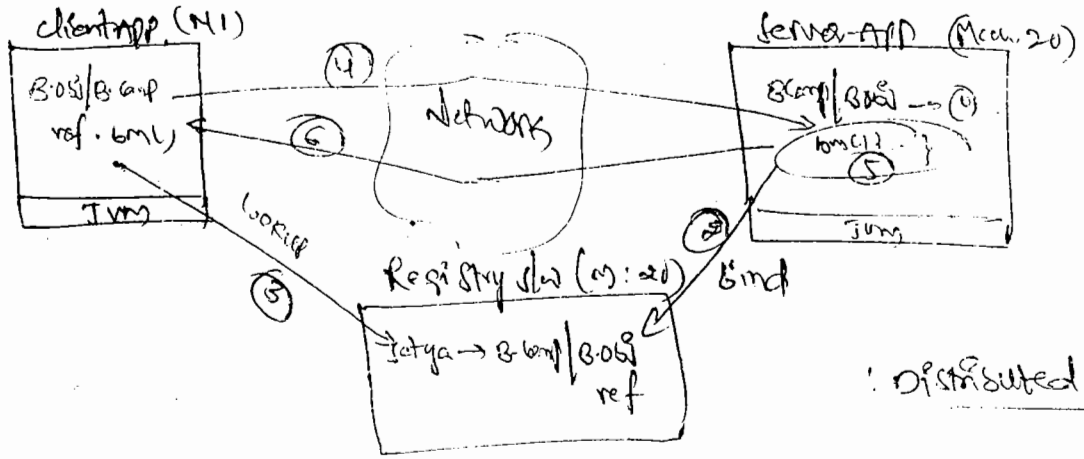
7) Gives Spring Jms providing Abstraction layer on plain Jms to develop Application that communicates through messages.

The socket programming based client-server app and Java apps interacting with ds s/w are called as traditional client-server apps.

These traditional client server apps are location dependent apps coz any change in the server app location must be informed 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 treditional 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 bcoz of Registry slw. The reason is no client is directly communicating with server application, that 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 slw all clients will recieve those details dynamically.

With respect to the diagram ① in server app the business obj will be developed having business logic in business methods.

② server app binds B.obj with registry slw having nick name.
③ client app gets B.obj ref from registry through lookup operation.
④ client app calls B method on business obj reference.
⑤ The B method of B.obj available in server app executes.
⑥ The results generated by B method goes to client APP.

14/11/11

The class that extends from Java.util.TimerTask class can have a tasks 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 task (use schedule(..) method of this class).

for example app on scheduling by using Jdk level concepts refer pg: (18)

Spring support scheduling operations, it internally uses quartz algorithm for task scheduling, The spring scheduling gives following benefits when compared the Jdk 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.sf.scheduling.timer.ScheduledTimerTask is a springBean that represents a tasks on which timer service can be enabled having initial delay, period parameters, but this tasks must extend from all classes defined in a Java class that extends from Java.util.TimerTask class.

The org.sf.scheduling.timer.TimerFactoryBean starts timer service on the specified tasks. this Bean initializes the Timer on the startup of applicationContext container and cancels the Timer on the destruction of ApplicationContext container.

for ex application on spring based scheduling refer pg (19).

The org.sf.scheduling.timer.MethodInvokingTimerTaskFactoryBean is a factory bean that returns timer task class obj pointing to specific User defined method of User defined Java class.

That means it converts ordinary pojo class into Rimoting is enabled obj.

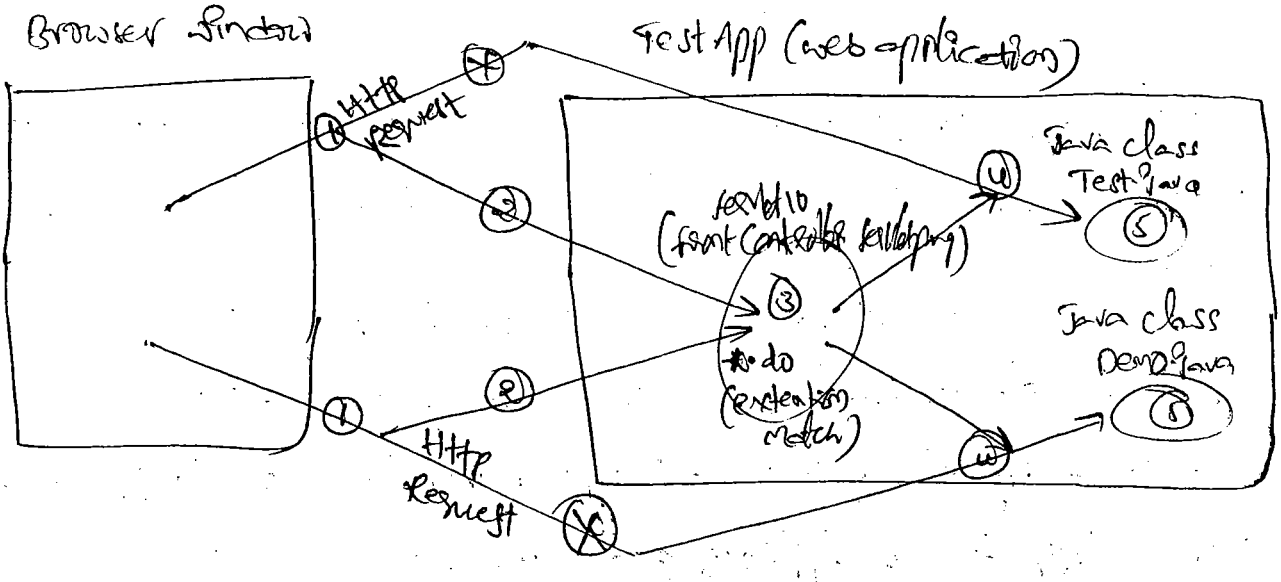
for example application ② on springboard Tuner app refer app @ ⑳

15/11/17

A special web resource program in the web application that is capable of traping 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 extention match or directly match URL pattern.

The Java classes of web application can not take HttpRequest directly. So it take 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.sf.web.servlet.DispatcherServlet will be taken as front controller servlet to take the HttpRequest 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 having <DispatcherServlet logicalName> - servlet.xml as spring Configuration file.

ex: If dispatcherServlet logical name in web.xml is abc
then spring cfg file name is "abc-servlet.xml"

DispatcherServlet uses one or other url-handler 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.sf.web.servlet.handler.SimpleUrl
                        HandlerMapping">
    <property name="mappings">          → Java.util.myprops type property
      <props>                           → word in request url
        <prop key="my.do">/xyz</prop>
        <prop key="xyz.do">/abc</prop>
      </props>
    </property>
</bean>

<bean name="/xyz" class="MyDemo"/>
<bean name="/abc" class="MyDemo1"/>
```

[url HandlerMapping required for DispatcherServlet to locate the destination Java class for trapped request]

Note: According to above code DispatcherServlet passes trapped HttpRequest to MyDemo class when that request URL contains my.do word.

HttpInvoker :—

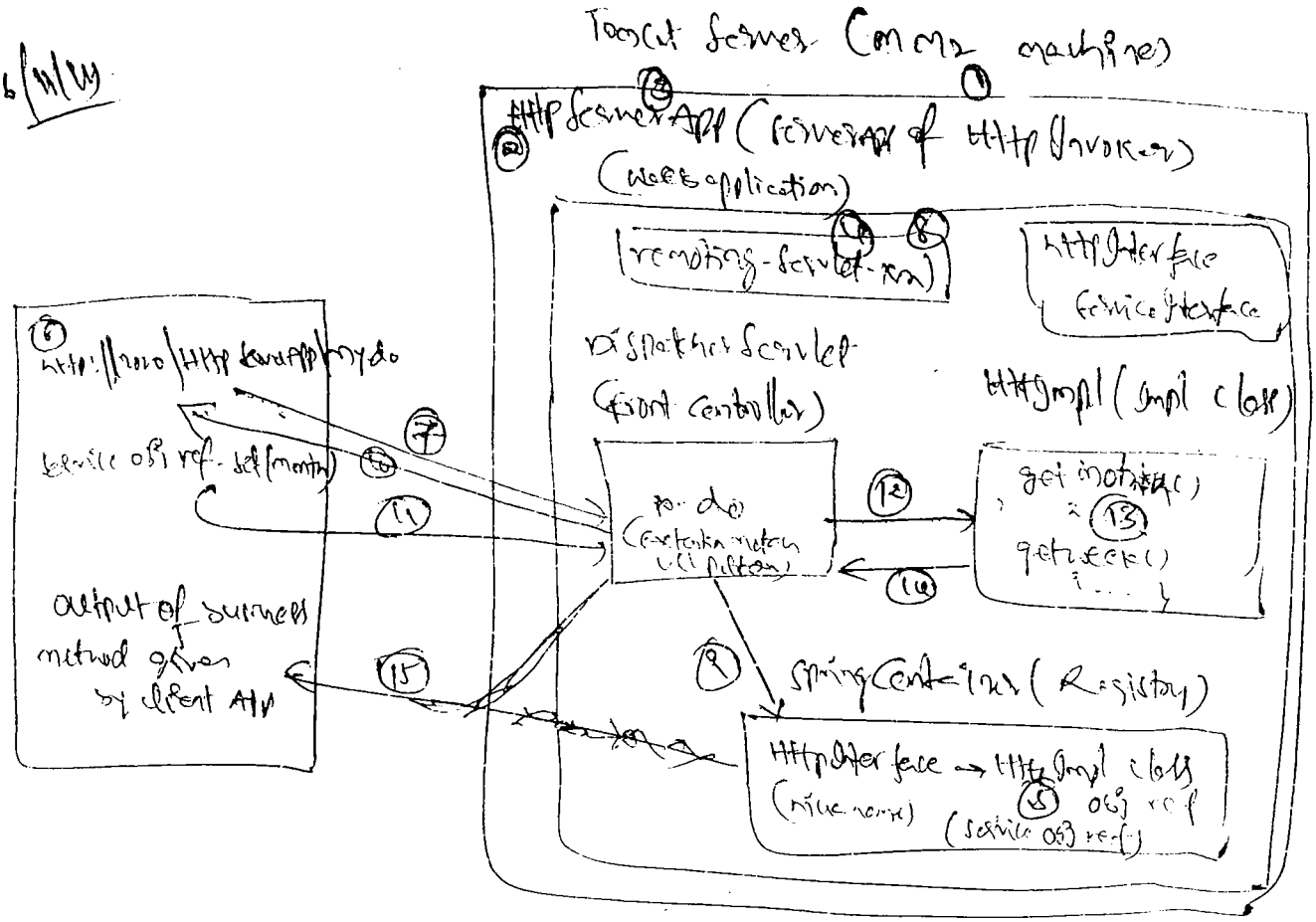
- Spring's own distributed Technology to develop distributed app.
- web based distributed Technology.
- Uses spring Container itself as registry also.
- The server app of this distributed app must be developed as web application and client app must generate Http requests to that ~~generates~~ server App.
- The server app uses spring api specified DispatcherServlet as front Controller to trap the Http request and to pass them to Implementation classes.

* The server application of HttpInvoker uses predefined org.sf.remoting.httpinvoker.HttpInvokerServiceExporter to convert ordinary Java class OSI to HttpInvoker service OSI and registers that service OSI reference with spring Container (Registry also)

The client app of HttpInvoker uses org.sf.remoting.httpinvoker.HttpInvokerProxyFactoryBean to get service OSI reference from registry (spring Container) and to inject that service OSI reference to OSI reference to a bean property of specified bean class.

Not only browser windows, we can use stand alone Java applications generating Http Requests.

16/11/19



with respect to the diagram:

① programmer deploys http server App in tomcat server

② Based on load on startup enable or dispatcher servlet, a servlet container initiate dispatcher servlet either ~~during~~ 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 preinstantiation on all the spring beans of spring config file.

⑤ During preinstantiation process. The HttpInvoker service-Explorer configured in spring config file registers

Service obj reference by Registry (s) for those, it converts HttpImpl class obj to service obj.

(6) client-Appl generates request url through HttpInvoker proxy factoryBean.

(7) Since D.S url pattern is *.do and client generate request url pl 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 get's service obj ref from Registry.

(10) D.S sends service obj ref to client Appl.

(11) client Appl calls businessmethod() on service obj ref.

(12) D.S prepares the req to services (HttpImpl) service class obj.

(13) The business method service class get() executed.

(14) Business method request comes to D.S

(15) D.S returns the request to client app.

for the above diagram based HttpInvoker ex appl @ Appl (17)

As a service client of HttpInvoker based client-Appl development gather following details from service provider.

1. service url to gather service obj ref from registry (s)
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 interoperatable components giving the ability to write client appl in any language, like Java, .net, etc.

client prefer HttpInvoker to develop spring distributed application,

17/11/11

Spring WEB Module

part 1 → Contains facilities and plugins to make spring apps communicatable from web flw sfw based applications like Struts app, Jsf app and etc...

part 2 → It is spring web mvc flw sfw. It is given to develop mvc2 architecture based web applications. Spring web mvc is spring's own web flw sfw to develop mvc2 web apps.

part 1 :—

struts and spring Integration :—

In struts and spring Integration struts app represents View layer and Controller layer, where as spring application represents Model layer. Struts app contains presentation logic and Integration logic. spring app contains business logic and persistance logic.

Struts and spring Integration can be done in two forms

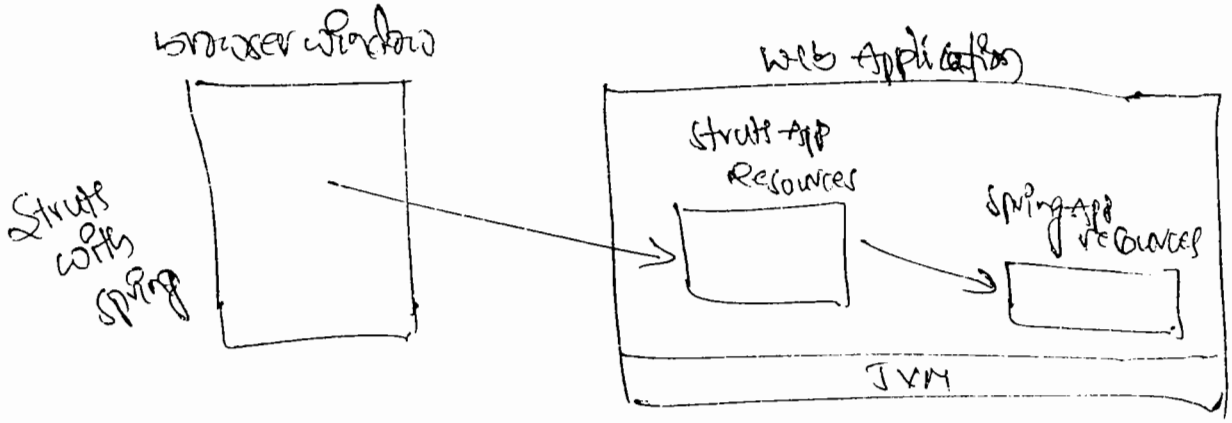
form 1) As struts with spring app

form 2) As struts app to spring app.

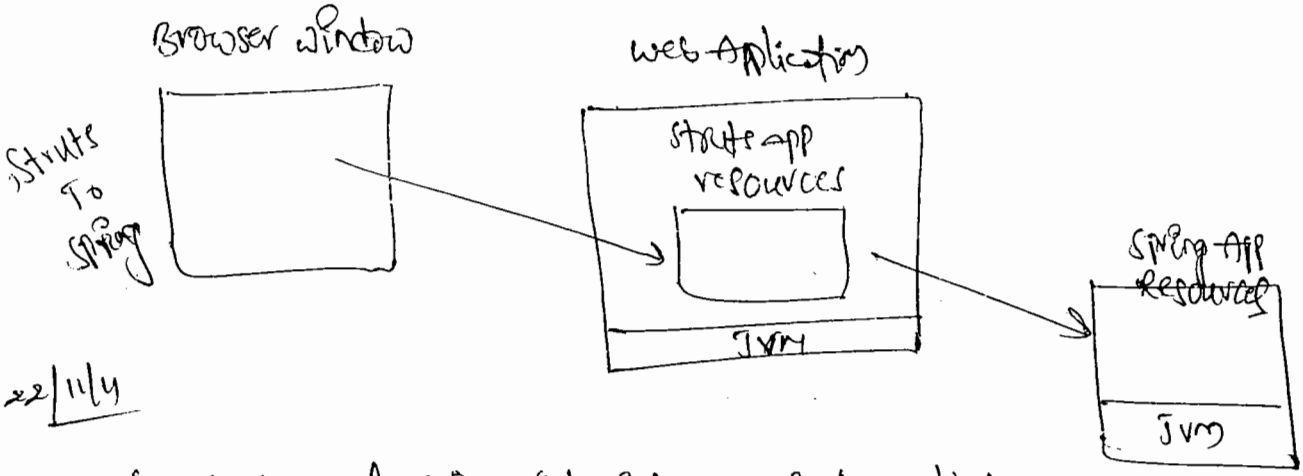
In form 1 :— Struts and spring app resides on the same JVM. Struts app acts as local client to spring. So spring app can be developed as non-distributed applications.

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 modules)

form1 based struts and spring Integration.



form2 based Struts and spring Integration



22/11/14

In struts and spring Integration based application we need to configure on spring s/w supplied plugin in struts config file, i.e org.sf.web.struts.ContextLoaderplugin. This plugin will be activated along with ActionServletInstantiation during either server startup or during deployment of struts application.

This plugin internally activates webappContext container by taking <ActionServlet logicalName>-servlet.xml as spring configuration file. And performs preInstantiation on all the beans configured in spring configuration file.

Ex: In struts config file :-

```
<plugin classname = "org.sf.web.struts.ContextLoaderplugin" />
```

(Looks to take the fixed notation

Ex②

```
<plug-in classname="org.st.web.struts.ContextLoaderPlugin">
    <set-property property="contextConfigLocation"
                  value="/WEB-INF/myCfg1.xml /WEB-INF/myCfg.xml"/>
</plug-in>
```

Here it allows us to take programmer choice names based one or multiple files as spring cfg files.

In Struts and spring Integration based application we can activate SpringContainer manually in Struts Action class but it is not a recomended process. It demands you to activate SpringContainer in every struts-Action class, so use Context loader plugin to activate spring container only once during server startup or deployment of Struts App.

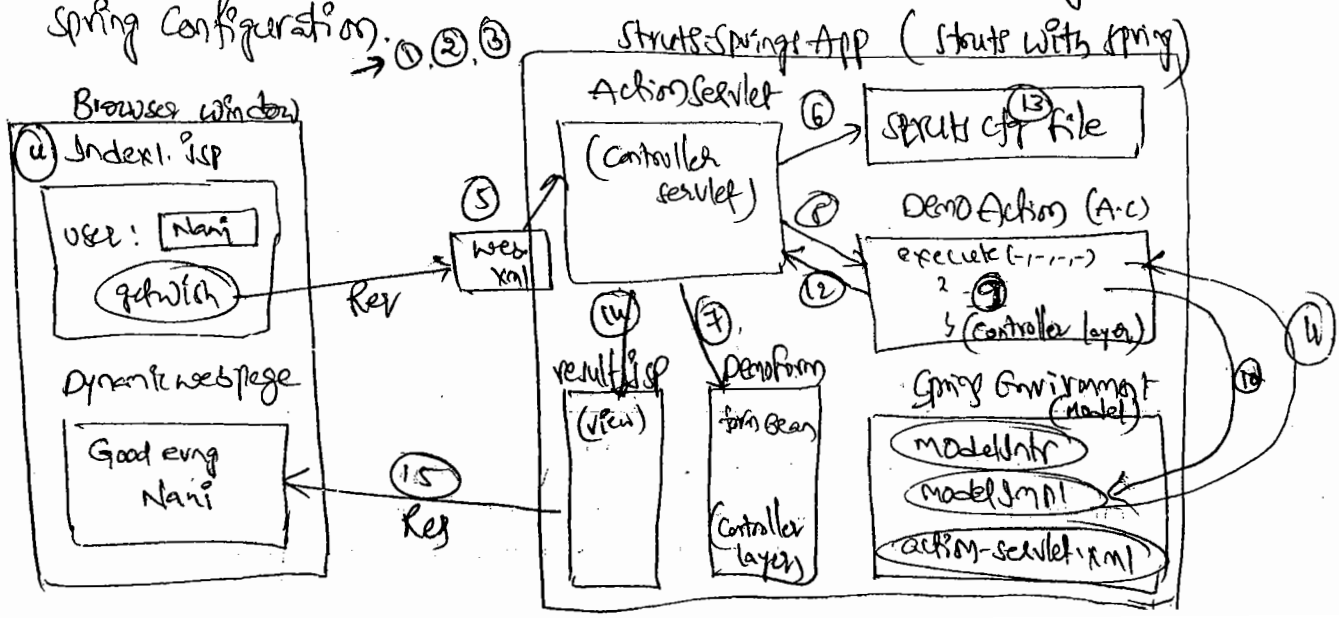
There are two approaches to develop Struts-Spring Integration based applications.

Approach ①:- Make Struts Action class as spring aware class with the support of xxxSupport classes, like

-ActionSupport, DispatchActionSupport, LookupDispatchActionSupport .....

Approach ②:- By making struts Action class as SpringBean in spring Configuration.

Ex



With Reflect to the Diagram

① Programer deploys Struts App in web server or App server

② Based on load-on-startup the servlet Container create obj of ActionServlet either during server startup or deployment of struts app

③ ActionServlet activates contextLoaderplugin and this plugin activates spring Container, this Container performs pre-instantiation on springbean of spring cfg file.

④ End user launches the form page on browser window.

⑤ Based on the configuration done in web.xml file the ActionServlet traps and takes the request.

⑥ ActionServlet uses struts cfg file entries to decide the formBean and Action class towards processing request.

⑦ Action servlet writes the form data of form page to formbean class object.

⑧ ActionServlet creates or locates Actionclass obj.

⑨ ActionServlet calls execute() of Action class, and gets springBean this method class from spring container using diff techniques.

⑩ Action class calls b.method of springBean from execute().

⑪ The b.logic generated result comes to execute() of Actionclass.

⑫ execute() returns the Controle 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.

All xxxSupport classes of spring environment internally extends from xxxAction classes of struts environment. So when we develop our java classes extending from xxxSupport classes of, they not only become struts Action classes, they also acts as spring aware classes.

ActionSupport class of springAPI 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 Approach ①:

Step ①: Configure ContextLoaderPlugin in struts cfg file like previous.

Step ②: Develop your struts Action class (DemoAction) extending from org.sf.web.struts.ActionSupport class.

Step ③: Get access to springContainer, activated by ContextLoaderPlugin from execute() of Action class of struts and call the b-method 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
  ModelIntr bobj = ctx.getBean("mb");
      // call b-method of springBean class
    bobj.bmth();
  }
}
```

Step ③ Develop springInterface, springBean and configure them in spring configuration file.

Step ④ Develop resources of Struts App in a regular manner and configure them in struts config file.

Step ⑤ Add struts api and spring api related jar files to WEB-INF/lib folder (struts → 10 ⊕ spring → 2 ⊕ 1) of struts app.

[springframework/dist/modules] → (spring-webmvc-struts.jar)

23/11/11

for above diagram based approach ② style Struts with spring app source code refer app (23) page 75-76

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 b.methods of spring bean class. In this situation we configure "org.sf.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="demoform" type="demopack.DemoForm"/>
  </form-beans>
  <action path="/demo" type="org.sf.web.struts.DelegatingActionProxy"
          name="demoform">
    <forward name="success" path="/result.jsp"/>
  </action>
```

In spring cfg file:

```
<beans>                                  → must match with the action path of DelegatingActionProxy class
  <bean name="/demo" class="demopack.DemoAction">
    <property name="bo" ref="dl" />
  </bean>
  <bean id="dl" class="demopack.ModelImpl" />
</beans>
```

The Struts A-C that is configured as spring bean

model layer's spring Bean

DemoAction.java

```
public class DemoAction extends Action
{
    ModelIntr bo;
    public void setBO(ModelIntr bo)
    { this.bo = bo;
    }
    public ActionForward execute(-,-,-,-)
    {
        // calling B.Logic and other logic
        String res = bo.getResult();
    }
}
```

supporting code for setter injection to inject spring bean class object.

Steps

procedure to develop struts with spring app based on Approach ②.

Step ①: Develop spring interface, springbean class and configure them in spring config file.

Step ②: Configure contextloaderplugin in struts configuration file. (Refer previous discussions)

Step ③: Configure org.sf.web.struts.DelegatingActionProxy class in struts config file, as proxy to receive original struts Action class req.

```
<action path="/demo" type="org.sf.web.struts.DelegatingActionProxy">
  <forward name="success" path="/result.jsp" />
</action>
```

step② Configure Struts Action class as SpringBean in spring cfg file and inject other SpringBean class obj to this Struts Action class property through Dependency injection.

step③ call a method of SpringBean class from execute() of Struts Action class by using the injected SpringBean class obj.

step④ Develop the resources of the StrutsApp in regular manner.

step⑦ Keep *jar files that represents Struts api and spring api in WEB-INF/lib folder. (10 Struts + 2 spring + 1 other)

* for above steps based (Approach②) Struts with spring application refer app@㉒ of page No: 73 to 75

for related information on integrating Struts and spring refere 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:

⓪ programmer deploys the HttpInvokerServerApp in webserver as web application. Because of predestruction and initialization process that takes place in server during either server startup or during deployment of web app, the bean reference (impl class obj) will be registered automatically with Registry s/w.

Eg:

(0c) programmer deploys Struts app in webserver, due to initialization and preinstantiation process the ActionServlet activates spring container through context-loader plugin and that spring container performs preinstantiation on springbeans of client cfg.xml.

(1) form page submits the req to Struts app

(2) ActionServlet traps and takes the request of form page

(3) AS uses struts cfg file entries to decide formbean and A-C 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 appl 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, rank for student) and uses its related DAO class to insert student details to DB table.

⑬: ~~B.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.

26/4/14

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 HB_student(sno number(4) primary key,
        sname varchar2(20), total ~~varchar2(20)~~ 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 3.0 → next → Config file name: [hibernate.cfg] → next

DB Driver: [orap] (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 → ⊙ spring 2.5 → select core libraries, persistant core libraries, persistant jdbc libraries, remoting libraries, jee libraries, web libraries → next → ☐ deselect AOP builder → file: [remote-servlet.xml] → next → Name of the session factory id: [sessfact] → finish

step⑥: perform hibernate reverse engineering on HB-Student table to generate hibernate POJO class, mapping file, DAO class.

Right click on ~~oracle~~ window → right click → open connection → expand orap → 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 → ⊙ Hibernate Types Id Generator: [increment] → next → finish

The above step generates HBStudent.java, HbStudentDAO.java HB student.hbm.xml dynamically.

step⑦:- Configure Basic datasource class of Apache DBCP in spring cfg file and inject the generated jdbc driver obj to ds property of local session factory bean configuration.

In remote-servlet-xml:

```
<bean id="dbcp" class="org.apache.tomcat.dbcp.dbcp.
                        BasicDataSource">
    <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="serfact"
        <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 HttpInvokerServer App.

Model.java

```
public interface Model
{
    public String findResult(String sname, int m1,
                             int m2, int m3);
}
```

ModelBean.java

```
public class ModelBean implements Model
{
    StudentDAO dao;
    // setter method for dao:
    public String findResult(String sname, int m1, int m2, int m3)
    {   // biz 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 persistance logic to insert record into db table
      HBStudent st = new HBStudent();
          st.setSname(sname);
          st.setTotal(new Integer(total));
          st.setAvg(new Double(avg));
          st.setResult(res);
          dao.save(st);
        return res;
    }
}
```

step⑥: Configure Implementation class(modelBean class) in spring cfg file.

In remote-servlet.xml:

```
<bean id="mb" class="modelBean">
  <property name="dao" ref="HBStudentDAO" />
</bean>
```

step⑦: Configure DispatcherServlet 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 ⑫ ⑰

step⑧: Configure HttpInvokerServiceExporter, SimpleUrlHandlerMapping classes in spring cfg file.

```
<bean name id="/service" class="org.st.remoting.httpInvoker.HYSE" />
  <property name="service" ref="mb" />
```

```
<property name="serviceInterface" value="Model" />
</bean>
<bean id="sh" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <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.

28/11/14

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 clientIntr, clientBean resources to StrutsApp to hold and return b.obj reference of http server application, given by HttpInvokerProxyFactoryBean.

clientIntr.java :

```
public Interface clientIntr
  {
    public Model getBobjref();
  }
```

clientBean.java

```
public class clientBean implements clientIntr
  {
      Model bref;
      public void setBoref(Model boref)
        { this.boref = boref;
        }
      public Model getBobjref()
        {
          return boref;
        }
  }
```

step⑤: copy Model.java from serverApplication to thin struts Application's 'src' folder.

step⑥: write following entries in springcfg file (clientcfg.xml)

```
<bean id = "pfb" class = "org.sf.remoting.httpInvoker.
                          HttpInvokerproxyfactoryBean" >
  <property name = "serviceUrl" value = "http://localhost:8080/
                                         HttpServerApp/my.do"/>
  <property name = "serviceInterface" value = "Model"/>
</bean>
<bean id = "cb" class = "clientBean" >
  <property name = "bref" value = "pfb"/>
</bean>
```

step(7): configure ContextLoaderPlugin in struts-config.xml file.

```
<plug-in className="org.sf.web.struts.ContextLoaderPlugin">
    <set-property property="contextConfigLocation"
                  value="/WEB-INF/clientCfg.xml"/>
</plug-in>
```

step(8): Add formpage (Input.jsp), formBean 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: [if] super class: [ActionForm] → formType: [InputForm] → Add (The following formBean properties) → sname, m1, m2, m3 → jspTab → select the check box: [Input.jsp] → path: [/demo] super class: [Action] type: [InputAction] → form tab → name: [if] → forwards tab → add → name: success → path: [/Result.jsp] → Add → finish

step(9): Develop StrutsAction class as shown Below.

InputAction.java:

```
public class InputAction extends ActionSupport
{
    public ActionForward execute(-,-,-,-)
    {
        InputForm if = (InputForm) form;
        String name = if.getSname();
```

```
int m1 = Integer.parseInt(rf.getM1());
int m2 = Integer.parseInt(rf.getM2());
int m3 = Integer.parseInt(rf.getM3());
// get Access to the ContextLoaderPlugin activated spring Container
WebApplicationContext ctx = getWebApplicationContext();
// get access to clientBean class object.
clientIntr cbi = (clientIntr) ctx.getBean("cb");
// get BO ref of server app from clientBean
Model boref = cbi.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 :

```
Result is : <%= 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 HttpServerApp is in Running mode.
2. Use following request.url in browser window.

http://localhost:2020/StrutsApp/

Note: After part①:

1) Move remote-servlet.xml to WEB-INF/ from src folder
2) Remove MyEclipse IDE generated libraries and add the following jar files manually to the build path of the project.

→ 8 - Regular spring jar files
→ 8 - Regular ~~struts~~ Alternate jar files.
→ spring-webmvc.jar
→ spring-webmvc-struts.jar

After part②:

1) Move clientCfg.xml file from src folder to /WEB-INF folder.
2) Add spring-webmvc-struts.jar file to part② project

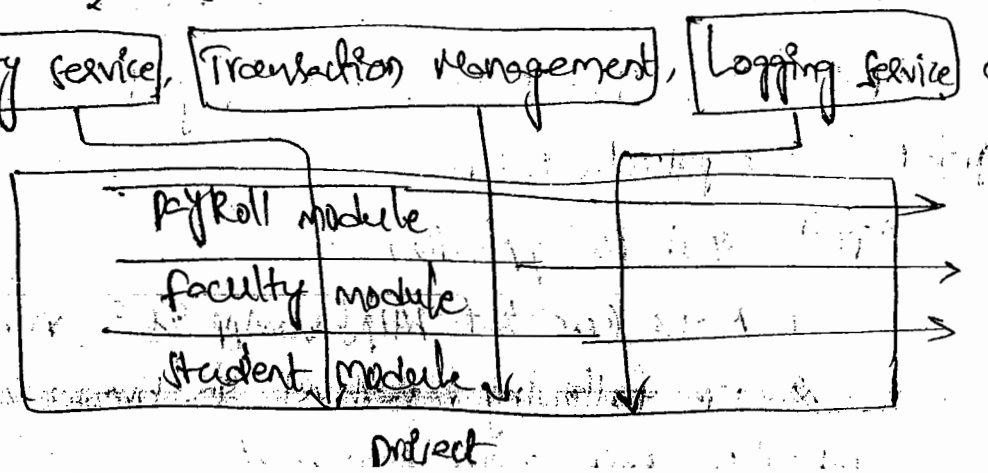
26/11/11

Spring AOP

Spring AOP is given configure and apply Middleware services on spring applications. At spring level Middleware services are called as Aspects.

The additional services that are configurable on the applications 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 out side of the application code without disturbing the application code. Due to this m/w services are called as cross cutting concerns of project.

spring aop is noway related with OOP. OOP is the methodology to create programing languages, where as the aop is the methodology to apply m/w services on spring applications.

what is the diff b/w Jee level m/w services and spring aop m/w?

Ans:-

Jee m/w services	spring aop m/w services
1) Allows only to work with server managed m/w services.	1) Allows to work with server managed, user defined, third party supplied, spring ibs supplied m/w services.
2) These m/w services are applicable only on deployable applications.	2) these m/w services applicable on stand alone, deployable appl.
3) Cannot control the start and end point of m/w service launch.	3) can control.
4) No provision to develop user-defined m/w services out side the server environment.	4) Allows us to develop user defined m/w services.

* It is not recomended to mix up m/w services code with application/business logic of b.methods as shown in page: 100 & 101.

It is recomended to seperate m/w services from b.logic of b.method that means externalize them and link them with b.methods

using xml files or annotations through spring app.

(Ex problem and solution scenario given in page: 100 to 102)

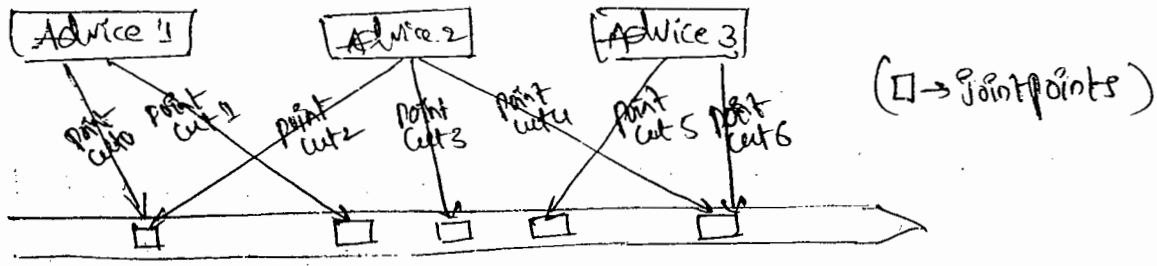
spring app Terminology:-

1) Aspect 2) Advice 3) JoinPoint 4) pointcut 5) Advisor
6) weaving 7) wiring 8) Target object 9) proxy object ...

① Aspect: Aspect is the plan to implement new service.

② Advice: Advice is the real implementation of new service.

Ex: Logging service, TX Management are advices that are implemented based on plan kept in their aspects.



③ JoinPoint: The possible positions in business method execution where the advices will be applied are called as JoinPoint.

Ex: starting of b.method, end of b.method, when b.method returns a value, when b.method raises exception etc...

④ pointcut: pointcut is a xml entry in springcfg file that links particular advice with particular JoinPoint position.

⑤ Advisor: Advisor is the combination of JoinPoint and pointcut, pointing to an Advice.

⑥ weaving: The process of configuring aspects / advices on SpringBean 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.

⑦ Wiring: configuring springbean properties with dependent values through IOC / dependency injection is called as wiring.

⑧ Target object: The springBean' class obj on which we are targeting to configure mlw services is called as Target object.

⑨ proxy object: The springbean class obj that is ready with mlw 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 mlw services.

(for above terminology refer page: 102 to 104)

29/01/11

We can develop 4 types of advices.

1) Before Advice 2) After Advice 3) Throws Advice 4) Around Advice

① Executes at the begining of Business method execution.

② executes when business method returns a value

③ executes when b-method throws exception.

④ Executes at the begining of b-method and at the end of b-method execution.

Based on the type of the advice that we develop the joint point position in the b-method will be decided automatically.

To know about developing diff types of advices refer page: 104 to 106.

Every Advice is a Java class implementing xxx interfaces of spring-AOP api.

* BeforeAdvice type advices are very useful to keep security authentication logics of b.methods outside the spring bean class.

* AfterAdvice is useful to perform cleanup operations at the end of b.method execution being from outside the b.method, like Removing Userdata from session when the User Logedout.

* ThrowsAdvice is useful to delete the temporary and Incomplete files when exception is risked in the middle of the b.method execution.

* The logics placed in AroundAdvice execute at the begining and at the end of b.method execution.

Spring AOP is borrowing AroundAdvice functionality and API from Alliance company. Aliance is also another company to give Java devs supporting AOP.

To develop AroundAdvice take a Java class implementing org.aopalliance.intercept.MethodInterceptor.

Ex:-

```
public class MyAdvice implements MethodInterceptor
{
    public Object invoke(MethodInvocation mi)
    {
        ---------- } Logic to begin Transaction mngment.
        mi.proceed(); → calls b.method
        ---------- } Logic to commit(s) or rollback(s)
    }
```

Executes at the begining of b.method →

End of b.method →

the TX based on the return value of b.method

mi → represents the current executing environment of b.method.

To link developed advices with b.methods of spring bean class we use pointcut Advisors in spring configuration file.

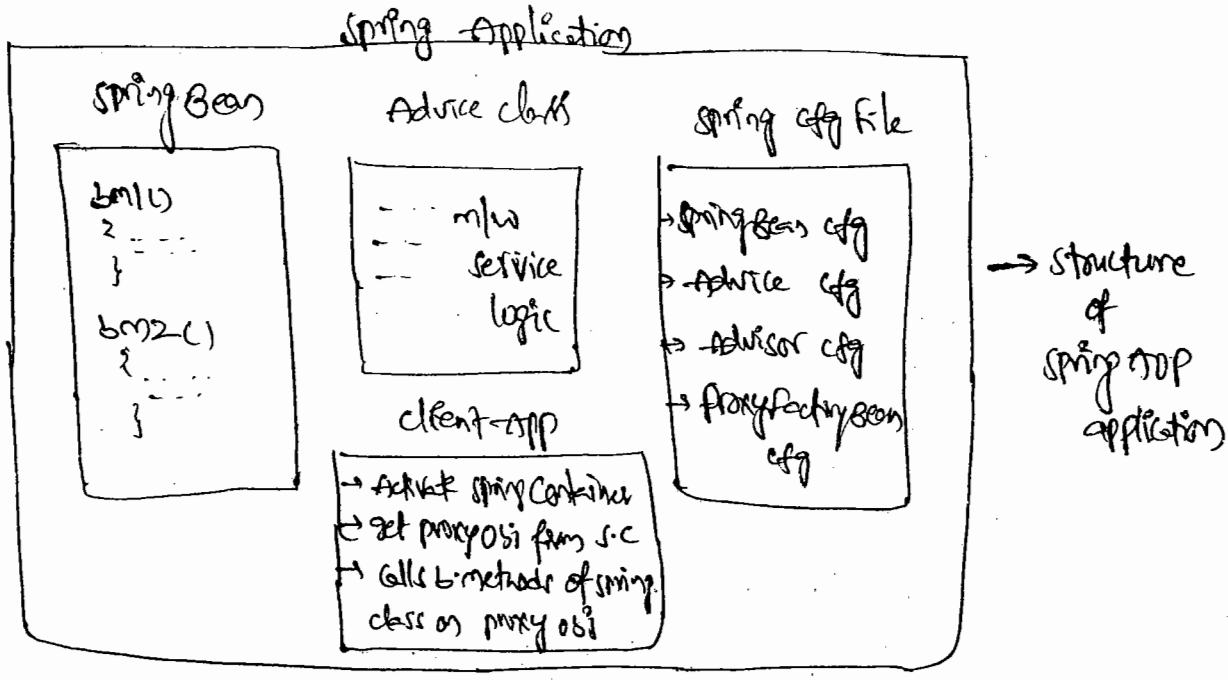
There are two types of pointcut advisors.

① NameMatchMethodPointcutAdvisor → specify b.method name manually to apply the advice.

② RegularExpressionMethodPointcutAdvisor → we can use the regular expression characters like '*', . etc to configure advices with Business methods.

for related information on these two advices refer page: 108 to 109

The org.sf.aop.framework.ProxyFactoryBean takes advice to Advisor configuration and returns one proxy object of springbean class by applying advices on the b.methods SpringBean.



30/11/19

Procedure To develop spring-App based Advices :—

Step①: develop spring Interface having b methods 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 b methods
- Configure org.sf.aop.framework.proxyfactoryBean

to 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 b methods of springBean on proxy object.

Ex-App:

- Demo.Java
- DemoBean.Java
- 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)
    {
        S.o.p("Name:" + name);
    } S.o.p("\n\n\n");
}
```

myAdvice.java

```
import org.sf.aop.*;
import java.lang.reflect.*;
import java.util.*;
public class myAdvice implements MethodBeforeAdvice
{
    public void before(Method method, Object[] args, Object target)
    {
        System.out.println("in before() of myAdvice class");
        // gathering details of b.method
        System.out.println("Bmethod name is:" + method.getName());
        System.out.println("Name of the Bean class is:" + target.getClass());
        System.out.println(method.getName() + " execution started at:" +
                                                new Date().toString());
        // changing b.method arg value
        String s = (String) args[0];  // gathering b.method 1st argument value
        if(s==null || s.length<=3)
            args[0] = "sunil";
    }
}
```

democfg.xml:

```
<beans>
    <bean id="targetObj" class="DemoBean" />
    <bean id="adv" class="myAdvice" />
    <bean id="advisor" class="org.sf.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice" ref="adv" />
        <property name="pattern" value=".*" />  → ①
    </bean>
```

(refers to all b.methods)

↑ Config of PointcutAdvisor

```
<bean id="pfb" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces" value="Demo" />
    <property name="target" ref="targetObj" />
    <property name="interceptorNames">
        <list>
            <value>advisor</value>
        </list>
    </property>
</bean>
</beans>
```

(This config gives proxy obj based on target obj by applying advice)

clientApp.java:

```
import org.springframework.context.support.*;
public class clientApp
{
    psvm(String[] args)
    {
        FileSystemXmlApplicationContext ctx =
                new FileSystemXmlApplicationContext("Democfg.xml");
        // get proxy obj
        Demo proxyObj = (Demo) ctx.getBean("pfb");
        proxyObj.sayHello("KNReddy");
        proxyObj.sayHello("knr");
    }
}
```

classPath: spring.jar, commons-logging.jar.

If client called method by having invalid argument values they can be collected by taking the support of BefreAdvice.

* working with "NameMatchMethodPointCutAdvisor", add this in place of ①: <property name="mappedName" value="sayHello" />

* Diff b/w FileSystem/ClassPathXml ApplicationContext and WebApplicationContext spring Containers?

Ans:- First Containers looks for spring cfg file in the specified path of filesystem / in 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 classes by having high Cohesion (Perfect structure) and low coupling (less dependency with other classes)

01/12/11: for AroundAdvice based springAop applications refer page: 116 to 118, App ① to ⑥

* If two Advices configured on the business 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 we prefer to write log messages using sop statements. It is recomended to use Logur tool to generate these log messages.

for ex: app that perform logur based logging on a method from outside the method by using all the four types of Advices refer app ⑤ of page 119 to 124

In logur: 3 imp objs will be there.

① Logger obj → enables logging on the class, generates five types of log msgs having priorities: (debug < info < warn < error < fatal)
→ Allows to set Logger level while retriving log msgs.

Note: If logger level to retrive log msgs is taken as 'warn' then application gives only those log msgs where logger level >= warn.

② Appender obj → To decide destination place of writing log msgs.
③ Layout obj → To decide the pattern and format of each log msg.

22/12/14 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, disposit amount operations of Transfer money task into a single unit and executing them with do everything or nothing principle comes under TX management.

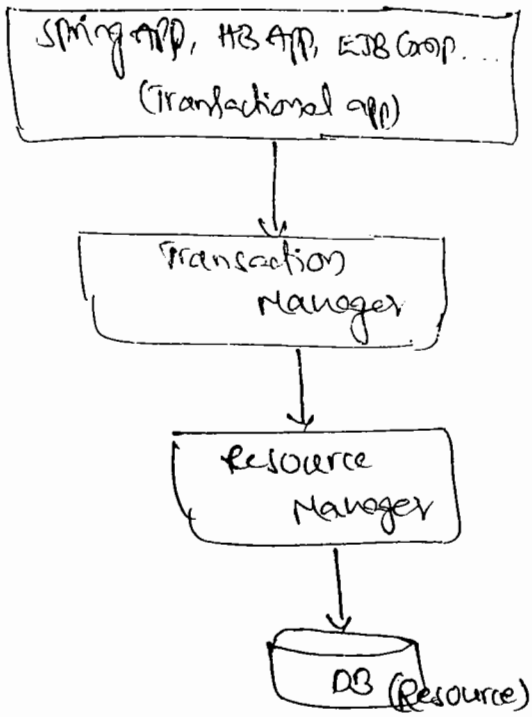
Sample code for TX management :-

```
public boolean transferMny (int srcAcno, int dstAcno, int amt)
{
    try {
        BeginTransaction (tx)
        // withdraw amt from source account
            ---
            ---   } operation 1
        // disposit amount to destination account
            ---
            ---   } operation 2
        CommitTransaction (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 may make the code as unmanagable code.

To overcome this prblem use Tx m/w service given by server or underlying sw which runs from outside the code and applies DO everything or nothing principle on the code.

Architecture of Tx management :-



* Based on the instructions given in app. the Tx manager commit/rollback the data of DB table through resource manager (JDBC)

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 (indivisible) into single unit → Atomicity

→ executing logics of the b. methods without violating the rules kept on 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 garanty for currection of that all data at the end of Tx / b.method (commit/rollback) is called as consistantcy

→ When multiple apps/users/threads manipulate same data of obj or db table synultaniously or concurrently then there is a chance of getting data currupton. 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 synultanious 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 (db's) that are involved in Tx mngmt there are two types of Tx mngmt.

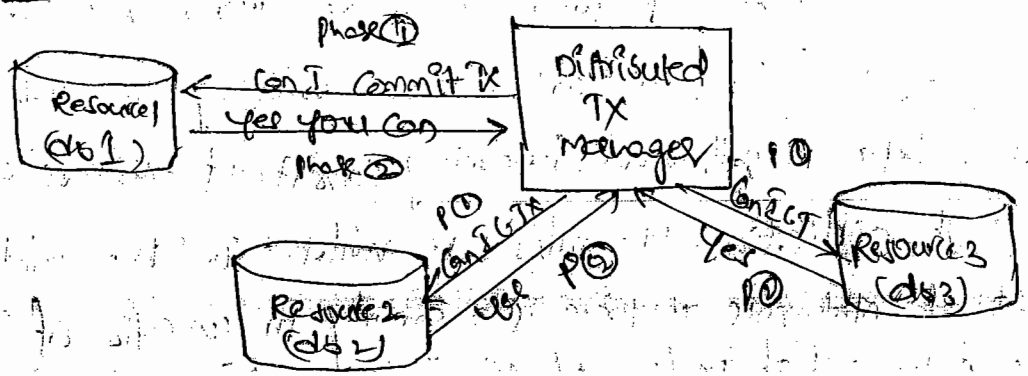
1) Local Tx management → user single resource (db) for all the operations of Tx management.

Ex: Transfer money operation in two a/c of same bank.

2) Distribute Tx management → uses multiple resources (db's) for all the operations of Tx management.

Ex: Transfer money operation in two a/c 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, where as HB supports only local Tx management.

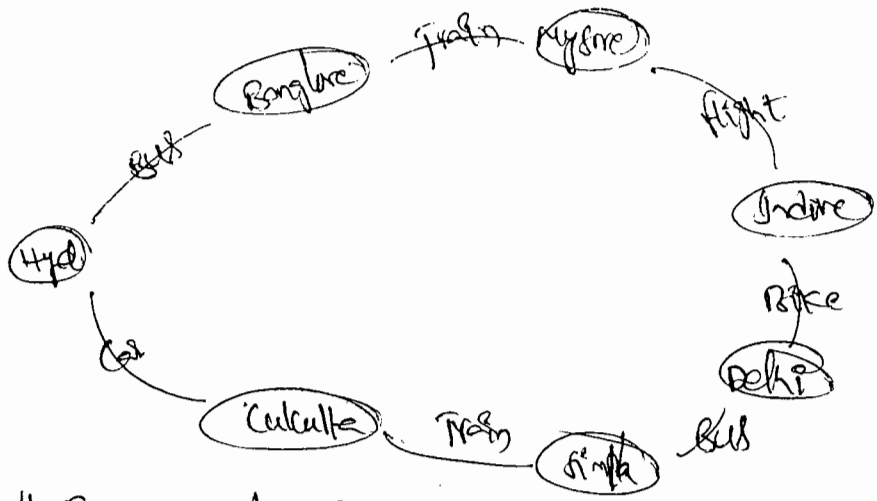
In spring apps we can use spring slw supplied Tx service or app server supplied Tx service for both local and distributed Tx's.

Spring slw managed Tx service is good for local Tx management in spring applications.

App ~~Spring~~ slw managed Tx service is good for Distributed Tx mgmnt.

Tx management Models:-

1. flat Tx model
2. Nested Tx model



When the above India Tour plan is given to a slw application that runs with a flat Tx model, if one or other journey tickets are not confirmed, then it will cancel the remaining all other journey tickets bcz 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
{
    JourneyL TicketBooking(operation 1)
    Journey2 TicketBooking(operation 2)
    :
    JourneyN TicketBooking(operation n)
}
```

When above tour plan is given to Nested TX model based sw application, eventhough one or other journey tickets are not confirmed it will confirm the remaining journey tickets. Bcoz 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 other sub TX.

```
MainTX
{
   SUBTX1
    {
      . . .   → Journey'1 Ticket Booking
    }
   SUBTX2
    {
      . . .   → Journey2 Ticket Booking
    }
    .
    .
    .
   SUBTXn
    {
      . . .   → Journeyn Ticket Booking
    }
}
```

→ EJB, HB supports only flat TX. Spring supports both flat, nested.

→ for 03/12/11 to 07/12/11
page (25) to (52)

08/12/11

Spring 2.5 supplies declarative TX management on methods of springbean class. while working with this annotation we can also specify TX attributes.

Ex: TestIntr.java :

```
public interface TestIntr
{
  void sm1() throws RuntimeException;
}
```

TestBean.java :

```
import org.sf.transaction.annotation.*;
import org.sf.jdbc.core.*;
public class TestBean implements TestIntr
{
  JdbcTemplate jt;
  public void setJt(JdbcTemplate jt)
  {
    this.jt=jt;
  }
@Transactional(propagation=propagation.REQUIRED)
                                  ↓ TX attribute
  public void sm1()
  { int res1=jt.update("update emp set sal=7777 where
                        empno=7839");
    int res2=jt.update("update emp set ename='satya' where
                        deptno='10'");
    if(res1==0 || res2==0)
    {
      s.o.p("TX rolled back");
      throw new RuntimeException;
    }
    else
    {
      s.o.p("TX commited");
    }
  }
}
```

spring Config.xml

```
<beans . . . .   spring-beans.xsd, spring-tx.xsd, spring-aop.xsd >
  <bean id = "dmds" class = "org.sf.jdbc.datasource.
                    DriverManagerDataSource" >
    <property name = "driverClassName" value = " . . .
  </bean>
  <bean id = "txmgr" class = "org.sf.jdbc.datasource.
                    DataSourceTransactionManager" >
    <property name = "dataSource" ~~value~~ ref = "dmds" />
  </bean>
  <bean id = "template" class = "org.sf.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 b.methods based on @Transactional annotation)

client.java:

// develop as a stand alone application to activate spring container and to get SpringBean class obj (TestBean) and call b.method.

jar files in classpath: spring.jar, commons-logging.jar

Jarfiles: 1. commons-logging.jar (spring 2.5)
2. org.springframework.asm-3.x.jar
3. " beans-3.x.jar
4. " context-3.x.jar
5. " context.support-3.x.jar
6. " core-3.x.jar
7. org.springframework.expression-3.x.jar

Available in (SPRING_HOME)/dist folder

```
62          }
63  }
64
```

----------UserBean.java---------- User-defined springBean class.

```
65
66
67  package p1;
68
69  import org.springframework.stereotype.*;
70
71  @Service("ub") -> here @Component can also be used
72  public class UserBean
73  {
74          String msg="Hello World";
75
76          public String toString()
77          {
78                  return "UserBean.msg="+msg;
79          }
80  }
81
```

----------DateFactoryBean.java---------- factoryBean that returns java.util.Date class obj.

```
82
83  package p1;
84
85  import org.springframework.beans.factory.FactoryBean;
86  import org.springframework.stereotype.Component;
87
88  @Component("dfb")
89  public class DateFactoryBean implements FactoryBean
90  {
91          public Object getObject() throws Exception {
92                  return new java.util.Date(); -> The resultant obj of this factoryBean
93          }
94          public Class getObjectType() {
95                  return java.util.Date.class;
96          }
97          public boolean isSingleton() {
98                  return false;
99          }
100
101
102 }
103
```

---------- -TestClient.java----------

```
104
105 package p1;
106
107 import org.springframework.context.annotation.AnnotationConfigApplicationContext;
108 import org.springframework.context.support.*;
109
110 public class TestClient
111 {
112         public static void main(String s[])
113         {
114                 AnnotationConfigApplicationContext ctx = new AnnotationConfigAppli
115                 ctx.scan("p1");  //p1 is the package name
116 launches spring container <-- ctx.refresh();
117                 TestInter test=(TestInter)ctx.getBean("tb");
118                 System.out.println("result is"+test.sayHello());
119
120         }
121 }
```

cationContext();

refresh() -> activates the spring container.

03/12/11

Title: Spring 3.x based App for annotations based Dependency Injection

----------TestInter.java----------

```
package p1;

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

----------TestBean.java---------- Main Springbean class.

```
package p1;
 import java.util.Date;
import java.util.List;
import javax.annotation.Resource;
import javax.annotation.Resources;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

@Component("tb") → recomended to use @service annotation.
public class TestBean implements TestInter
{
        //Bean Property
        UserBean u1;

        @Value("10")
        int no;

        Date d1;

        @Value("#{T(java.util.Arrays).asList('India','Bharat','Hindustan')}")
        String nicknames[];

        @Value("#{T(java.util.Arrays).asList('Red','Blue','Green')}")
        List colors;

        @Resource(name="dfb") → It is like ref tag/attribute
        public void setD1(Date d)
        {
                d1=d;
        }

        @Resource(name="ub") → Reqd id of UserBean
        public void setU1(UserBean u1)
        {
                this.u1=u1;
        }

        public String sayHello()
        {
        System.out.println("no"+no);
        System.out.println("d1"+d1.toString());
        System.out.println("nicknames");

        for(int i=0;i<nicknames.length;i++)
                System.out.println(nicknames[i]);

        System.out.println("Colors="+colors.toString());
        System.out.println("roles="+roles.get("ravi"));
        return "Good Morning";
```

Inserts Date class obj to d1 prop given by DateFactoryBean

Inserts UserBean class obj to U1 property

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 pdf 2 of pdf file

for high level architecture diagram of spring 3.x refer pg: 3 of 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 & 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.

08/12/11

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, Jsp kind of web flw s/w.

In web flw s/w the market leader is struts, the spring web mvc is gives alternative to struts.

spring is popular to develop model layer to develop Blogic and persistance logic. spring web mvc is not that much popular to develop view and controller logic of web app.

feature of spring web MVC:—

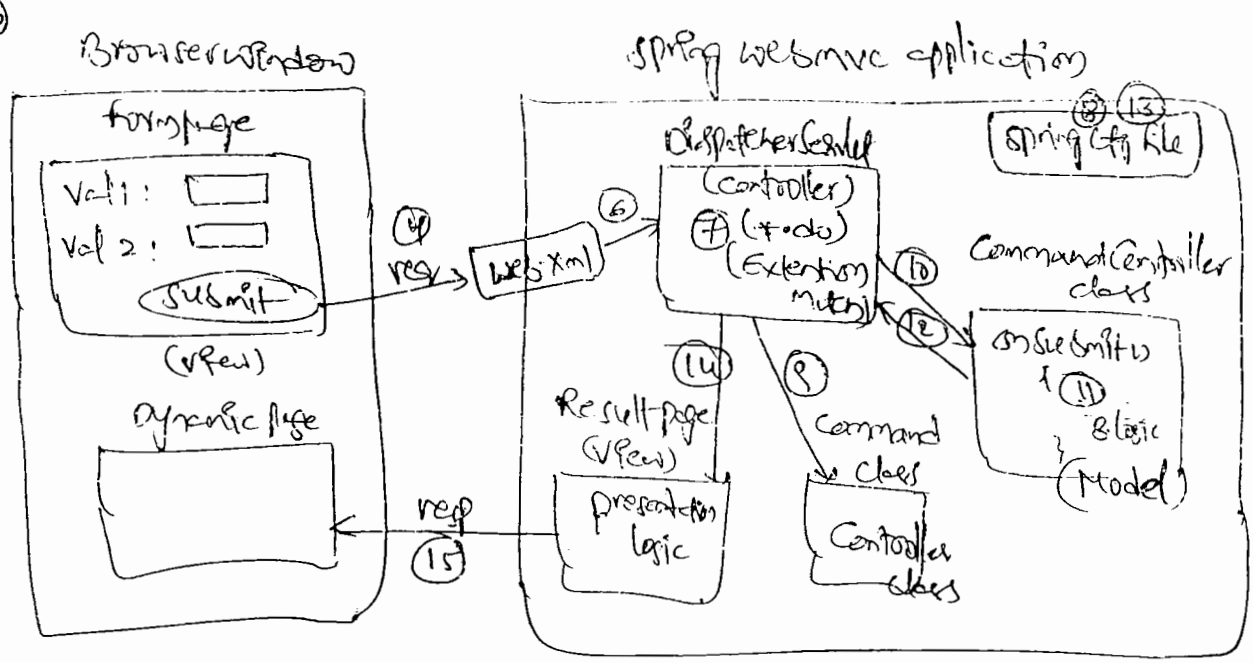
* Allows to develop MVC2 arch based web app.
* Activate spring container during server startup or deployment of web application through DispatcherServlet.
* Gives more support for dependency Injection.
* Allows to develop resources of POJO and POJI.
* Allows to develop presentation logic of view layer using multiple technologies like Jsp, freeMarker, Velocity ... etc.
* Gives Jsp Tag libraries to simplify the programing.
* Allows to use other spring module's features.

Struts	spring
Controller - ActionServlet	DispatcherServlet
formBean	Command class
Action class	CommandController class

Struts cfg file
- Action Mapping
- Action forward cfg

spring cfg. file
- URL handler
- ViewResolvers

flow of execution of spring web mvc :-



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 programs deploys spring web app in web app server.

② Based on load-on-startup, servlet container instantiate DispatcherServlet class obj either during server startup or during deployment of app.

③ DispatcherServlet activates WebApplicationContext (spring container) and container performs pre-instantiation of springbeans of spring cfg file, whose name is <DispatcherServlet logicalname>-servlet.xml.

⑨ End user launches form page on browser window.

① End user submits request from form page.

⑥, ⑦ : Based on configuration done in web.xml, DispatcherServlet traps and traces the req and also reads form data.

⑧ Dispatcher uses UrlHandler configuration to decide Command class and Command Controller class that are required to process req.

⑨ DispatcherServlet writes form Data to Command class obj.

⑩ D.S calls onSubmit() on CommandController class.

⑪ B.Logic of onSubmit() process and generates the result.

⑫ onSubmit() returns ModelAndView class obj to D.S.

⑬ D.S use ViewResolver config to decide Result page and CommandController layer.

⑭ D.S forwards control to the result page.

⑮ Result page uses presentation logic and formats the logic, sends formated result to browser window (Dynamic web page)

Struts	Spring web mvc.
1) Less support for dependency Injection.	1) More support
2) Given by Apache foundation	2) By Interface 2.1
3) Given validator plugin to perform form validations	3) Form validation should be perform explicitly.
4) Built-in support for AJAX	4) No Built-in support for AJAX
5) Allows to use EL (6x) OGNL (Struts 2.x) in JSP program.	5) Given only EL to use in JSP program
6) Form Component name and formBean class property name must same.	6) No need to match

If D.S logic name is 'abc' the spring cfg file name should be "abc-servlet.xml". And this contains,

1. UrlHandler Bean cfg.
2. Command class cfg
3. CommandController class cfg.
4. View resolver cfg

etc...

① UrlHandler cfg:-

It helps DispatcherServlet to link client generated req to CommandController class. There are two types

i) SimpleUrlMapping → Allows to map the req trapped by D.S to CommandController class by using bean id or bean name of CommandController class. It is recomended to use.

ii) org.sf.servlet.handler.BeanNameUrlHandlerMapping → Allows to map the req traped by D.S to CommandController class by using only bean name of CommandController class

② Command class cfg:-

No seperate config required for this, bcz it will be configured along with CommandController class config.

③ CommandController 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 CancellableFormController
④ UrlFilenameViewController
⑤ BaseCommandController

→ To handle single submit button based form page generated request

Available in org.springframework.web.servlet.mvc pkg.

⑥ org.springframework.web.servlet.mvc.multiaction.MultiActionController
(To handle multiple submit button based)

All predefined xxxController classes of spring API implements "org.springframework.web.servlet.mvc.Controller" directly or indirectly.

The SimpleFormController and AbstractFormController based CommandController classes launches form page on the browser window when they get req from browser window having HttpRequest method "GET". But they process the req given by browser window when the HttpRequest method is "POST".

while configuring SimpleFormController type C.C. class we can specify its form page by using "formView" property and specify the result page by using "successView" property.

⑩ 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 class name

Jsp → org.sf.web.servlet.InternalResourceViewResolver,

Velocity → org.sf.web.servlet.view.velocity.VelocityViewResolver

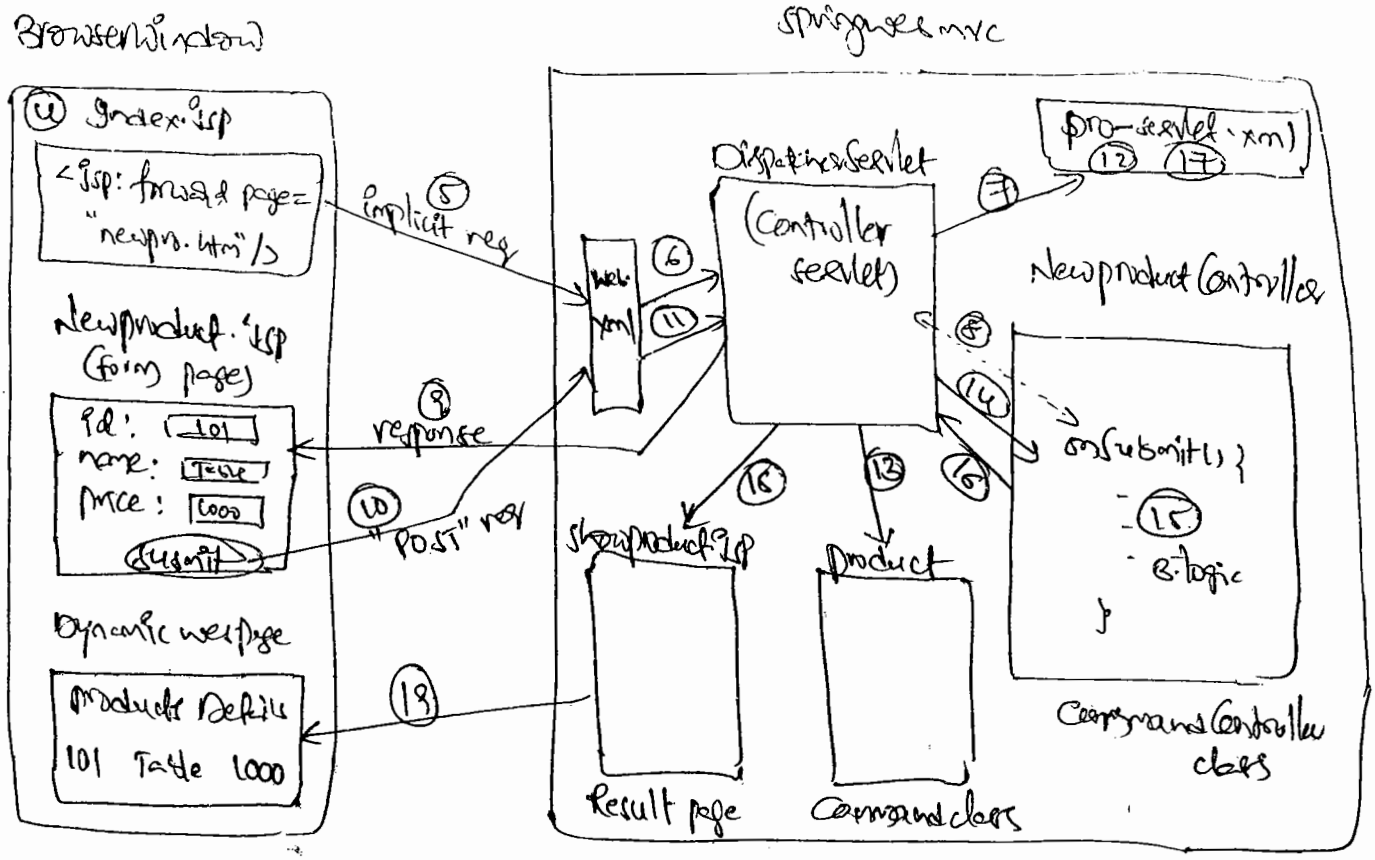
freemarker → org.sf.web.servlet.view.freemarker.FreeMarkerViewResolver

Xlst → org.sf.web.servlet.view.xlst.XlstViewResolver

JasperReports → org.sf.web.servlet.view.jasper.JasperReportViewResolver

Ex: app on SimpleFormController based C.C class :-

for ex. app on this refer app (2w) of bookref.



To make SimpleFormController type C.C class 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 (1) to (9)

W.R.To Diagram:

①②③ : Are same as previous springwebmvc arch.

④⑤ : The initial req given to the web app, makes welcome page to generate implicit request to web app.

⑥ : Based on cfg done in web.xml D.S takes the req.

⑦ : D.S uses UrlHandlerMapping of spring cfg 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 CommandController class on BrowserWindow.

⑩ : End user submits form page having req. method as "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 ⑱ : Same as ⑨ to ⑮ of previous Diagram (springwebmvc arch).

* spring is having its own JSP tag library whose tld file is "spring.tld" available in springHome/dist/resources folder.

11/12/14

Flow of execution of app(2a) of booksket

(A) (B) (E) → same as spring Architecture diagram given in class.

(D) → End user gives request to the deployed productMVC application so the default welcome page index.jsp will be executed automatically.

(E) Index.jsp generates implicit request to web application having default http req method "GET" (ref 2655)

This request related request url contains newpro.htm word

(F) Based on the configuration done in web.xml file, the DispatcherServlet traps and takes the request (url pattern is *.htm and implicit request url contains "newpro.htm") (ref: 2702 → 2705, 2695 → 2700)

(G) DispatcherServlet uses SimpleUrlHandlerMapping class obj to links implicit request with newproductController class. (ref 2720)

<prop key = "newpro.htm" > Controller </prop>

(H) Since the commandController class type is SimpleformController. the recieved request method is GET, the DispatcherServlet renders form page on the browser window. (ref 2725, 2729). This time internally the ViewResolver will be utilized

(I) DispatcherServlet launches new product.jsp on browser window as form page. (2659 → 2687)

(J) End user fills up the form page and submits the request. (2690)

(K) same as (F)

(A) same as (G)

(M) Since the commandController class type is SimpleformController and request method of form page is "POST", the new product Controller class comes ready to process the request.

(N) DispatcherServlet writes the received form data of formpage to Command class obj by calling setXXX() on that object.

(O) DispatcherServlet calls onSubmit(-,-,-,-) on NewProductController class obj to process the request (gpt1, gpt2).

(P) onSubmit() returns ModelAndView class obj to DispatcherServlet having logical name "showproduct".

(Q) DispatcherServlet uses "successView" property value of NewProductController class to decide result page. 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 in programatic approach.

for this we need to take a Java class implementing org.sf.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 in this Java class not only can be used in springmvc mvc applications, it also can be used outside the spring webmvc applications

for example app on validations based springmvc mvc app refer app @ 2.5.

procedure to add programatic server side form validations to app:-

step① develop a Java class implementing org.sf.validation.Validator interface having the Java code based form validation logic.

refer productvalidator.java of page: (21)

(22) configure this validator class as bean property value of command controller class.

refer validator property config of newproductController class.

(23) write following logic in form page to display the generated form validation errors

(24) The remaining resources of the application are same as app1

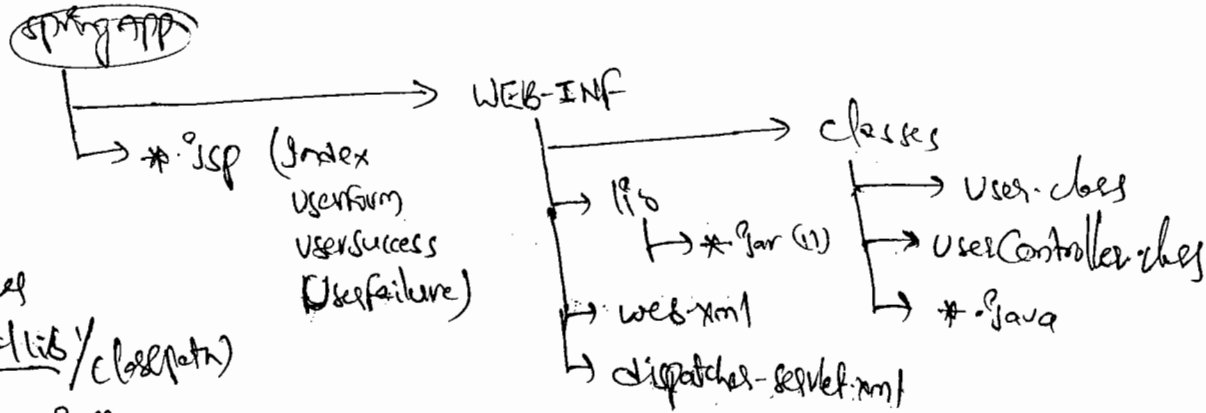
* spring3 supplies more number of annotations to develop spring webmvc applications.

@ controller annotation makes our java class as CommandController class

@ requestmapping annotation links webrequest with one command controller class.

@ modelAttribute annotation gathers one obj kept in modelmap class object.

Deployment directory structure for annotations based spring web mvc (spring 3.x) :-



Jar files
WEB-INF/lib/(classpath)

index.jsp:

1. antlr-runtime.jar
2. Commons-logging.jar
3. org.st.asm.3.x.jar
4. " beans-3.x.jar (cp)
5. " context-3.x.jar (cp)
6. " context.support-3.x.jar (cp)
7. " core-3.x.jar (cp)
8. " expression-3.x.jar
9. " web-3.x.jar → *(cp)
10. " web.servlet-3.x.jar → *(cp)

Index.jsp

```
<jsp:forward page="pro.htm"/>
```

web.xml :

Configure dispatcherServlet class having dispatcher as logical name and *.htm as url-pattern.

dispatcher-servlet.xml :

```
<beans ..... spring-context.xsd>
    <context:component-scan base-package="p1" />
</beans>
```

Userform.jsp

```
<form method="post" action="pro.htm">
    UserName : <input type="text" name="name" />
    password : <input type="password" name="password" />
        <input type="submit" />
</form>
```

User.java

```
package p1;
public class User
{
    private String name, password;
}   // getXxx(), setXxx() methods
```

UserControllerClass :

Note: @RequestMapping annotation can be used to link specific http method based request with specific java method of command-controller 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("/ms.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("satya") && 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 method name as additional request parameter value.

To configure this additional req parameter name use parameterMethodNameResolver Bean.

→ for ex: of this refer ~~app~~ Handout of 13/12/14.

The Java class that contains persistance logic and seperate 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 recomended process bcz 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

24/12/11

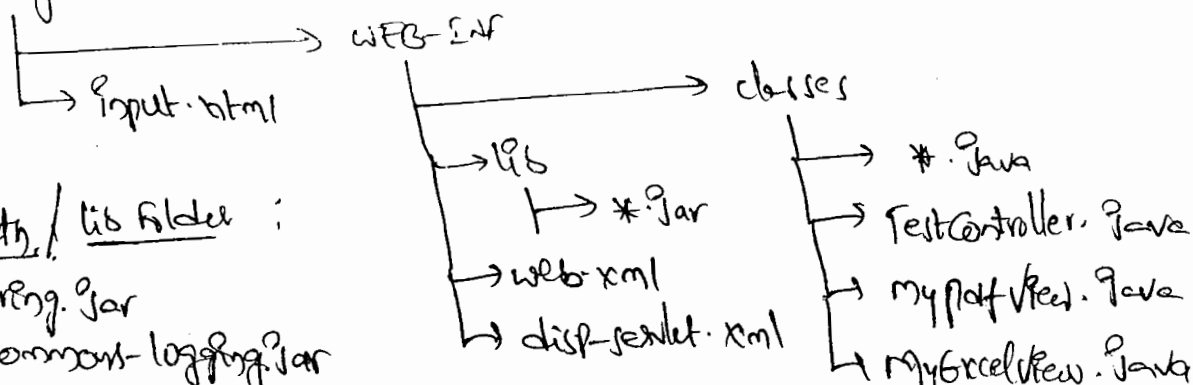
Spring API supplies AbstractXxxView classes to generate excel document based view and spring web mvc applications. These classes are like org.sf.web.servlet.view.document.AbstractExcelView, AbstractJExcelView, AbstractPdfView and etc...

AbstractExcelView internally uses POI mechanism to generate excel doc based views.

AbstractPdfView internally uses IText mechanism to generate pdf doc based views.

Ex: app on writing spring web mvc application generating pdf, excel document based results.

springpdfapp:



classpath / lib folder:

1. spring.jar
2. commons-logging.jar
3. poi-3.2-FINAL-20081019.jar → springframe/lib/poi folder
4. iText-2.1.7.jar → springframe/lib/iText folder
5. spring-webmvc.jar

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.

~~dispatcher~~ - TestController.java

```
import javax.servlet.http.*;
import org.springframework.web.servlet.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("Sandy");
      l.add("Narayana");
      l.add("Reddy");
      return new ModelAndView("viewbean", "result", l);
   }
}
```

Bean id of springbean that contains view generation logics

Note: org.springframework.web.servlet.view.BeanNameViewResolver makes spring web mvc application to use springbean 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.servlet.handler.SimpleUrlHandlerMapping">
      <property name="mappings">
         <props>
            <prop key="springdb">Controller</prop>
         </props>
      </property>
   </bean>
   <bean id="Controller" class="TestController"/>
   <bean id="viewResolver" class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
</beans>
```

myExcelView.java :

```
import java.util.*;
import javax.servlet.http.*;
import org.sf.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 names"));
    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 blocked to generate next request until given request related response comes from server then that communication comes under synchronous communication.

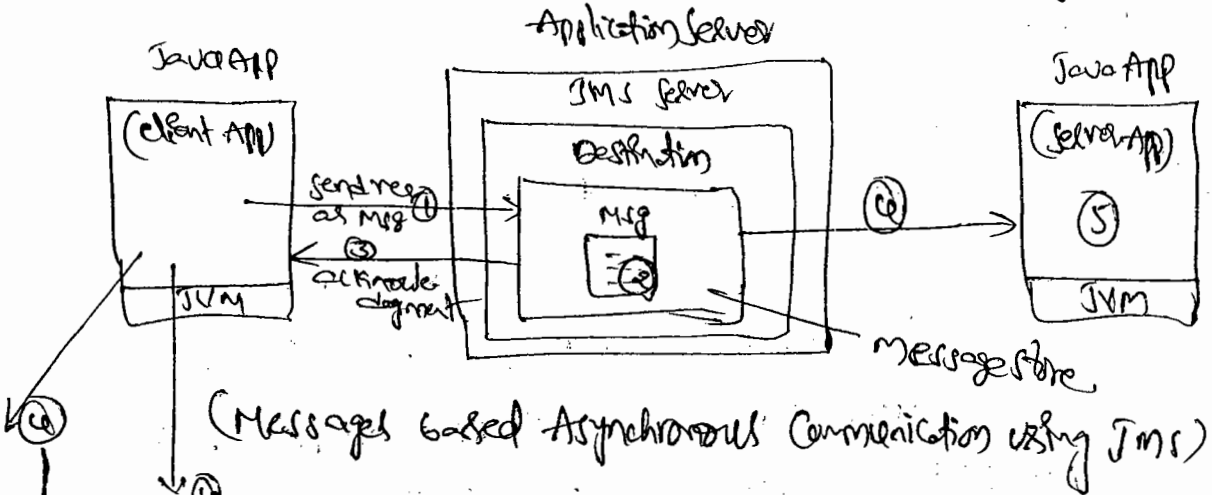
In client/server communication if client is free to generate next request or to perform client side operations without waiting for given request related response from the server then that is called as Asynchronous communication.

Generally request-response model communication is synchronous and messages 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 messages based Asynchronous communication between two distributed applications use JMS (Java messaging service)



In the diagram client is sending request as message and that message is stored in the destination, client is getting acknowledgement from destination.

After receiving acknowledgement client is free to generate next request asynchronously.

when server is free it gathers new msg from destination and process the request. server app sends result to client app as msg through destination.

In synchronous communication client and server applications are tightly coupled. In Asynchronous communication the client and server applications are loosly coupled.

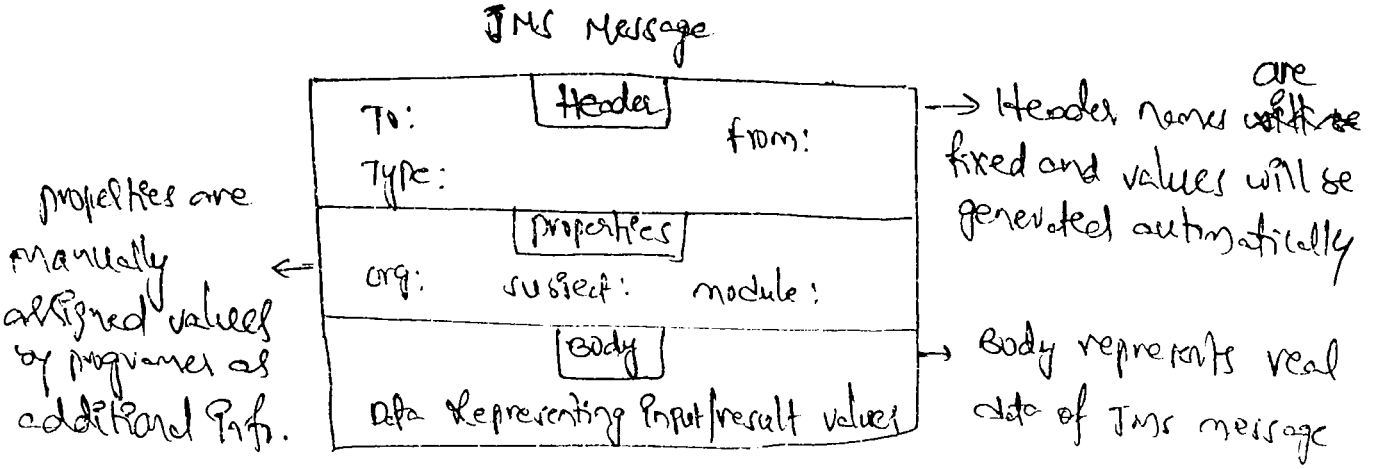
Jms is a sun microsystem supplied specification having set of rules and guidelines in the form of Jms API. Vendor companies use Jms API to develop Jms servers and destinations. programers use Jms API to develop client and server applications having the ability to create messages, send and recieve messages.

Vendor company supplied Jms server, destination s/w together is called as Jms provider. IBM supplied Jms provider s/w name is mq series where as microsoft supplied s/w name is ms mq. But every appserver s/w gives 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).



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 AppServer 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 Connectionfactory, Queue Con. factory, Topic destination, Queue destination in domain1 server of Glassfish ex:-

Step①: Start domain1 server of Glassfish and open its Administration console.

Step②: Create JMS Topic Con. factory

Admin console → resources → JMS resources → Connection factories → new → Jndi Name: [Confact Jndi] ResourceType: [TopicConnectionFactory] → ok

Step③: Create Queue Connection Factory.

Admin Console → resources → JMS resources → Connfactories → new → Jndi Name: [QConfact Jndi] → ResourceType: [QueueConfactory] → ok

Step④: Create Topic Destination.

AdminConsole → resources → jms resources → Destination Resources

→ new → JndiName: TopicJndi

PhyficalDestinationName: TConfactJndi } → OK

Resource Type: ~~TopicConfactory~~ → javax.jms.Topic

Step(6): Create Queue Destination

→ new → JndiName: QueueJndi

Phyfical dest. Name: QConfactJndi } → OK

Resource Type: javax.jms.Queue

Note: All the above four resources will be registered with Registry o/w automatically.

JMS API means javax.jms package.

Every server supplied jar file that represents javaee api'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: 5.6 of Handout.

We can develop the Receiver/subscriber application to read the msgs from destination either in synchronous & asynchronous mode.

In Asynchronous mode the msglistener should be registed with destination whenever ~~the~~ a msg arives to the destination the JMS provider delivers that msg by calling onmessage() of

MessageListener which is implemented by (Receiver) client application.

The receiver/subscriber application can use message selectors as small queries to filter messages that are required to be received from destination.

Spring JMS

In spring JMS programming JmsTemplate class is there providing abstraction layer on plain JMS programming.

To create this both obj connection factory obj, destination object (Topic or Queue) are the base objects.

for Spring JMS based P2P messaging domain application refer page: 6 to 8 of handout.

The Receiver/Subscriber 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 msgs from destination synchronously through receive() method call. And we can use the MessageListener's onMessage() method to receive asynchronously.

while running pubsub model programs first start subscriber program (one or more) then run publisher program.

In pubsub model there is a timing dependency b/w publisher and subscriber, but it still gives Asynchronous Communication

17/11/11

WEB SERVICES

Web services is a distributed technology, it allows us to develop interoperatable distributed applications.

- language independent
- Architecture independent
- Platform independent etc

Generally the server app of web services is web app and client app is standalone or web application.

The server app of webservice contains b.methods and client app calls them from local or remote locations.

Imp Terminologies,

WSDL → webservice discription Language

UDDI → Universal discovery and discription and Integration

SOAP/REST → (Simple Obj 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 the client app.

webservice client interacts with webservice server app by using SOAP.

The b-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 builtin api of JDK 1.6,

Spring webservices provide abstraction layer on Jax-ws based webservices programing.

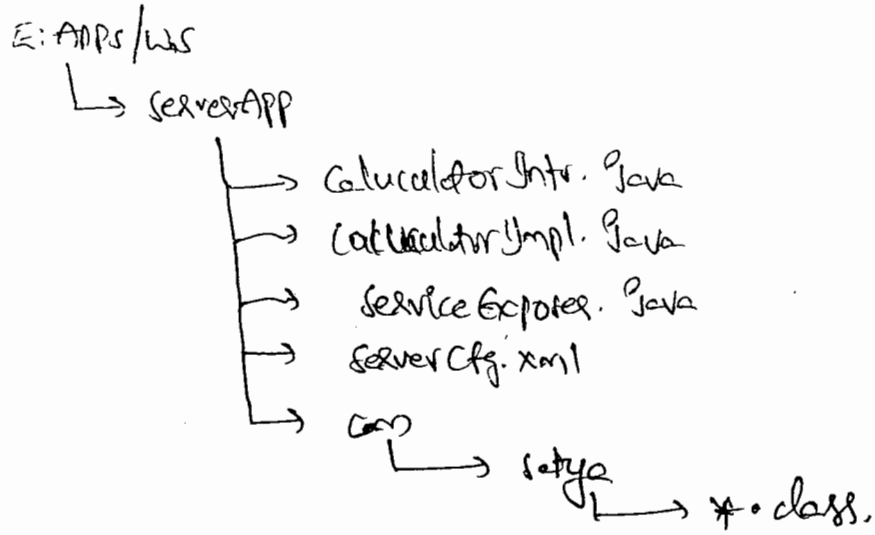
- spring supplies built in web server / container in spring s/w itself to manage server app of web services.

- supplies built-in UDDI Registry

- Allows to export web services related b components through dependency Injections.

- Allows to get web services related b components reference in client app through Dependency Injections.

* The Server app of web services in spring environment :-



CalculatorIntr.java

```
package com.satya;
  public Interface CalculatorIntr
      {  public String sum(Int x, Int y);
      }
```

CalculatorImpl.java

```
package com.satya;
  Import javax.jws.WebMethod;
                WebService;
                .soap.SoapBinding;
  Import org.st.stereotype.service;
```

```
@SoapBinding(style = SOAPBinding.Style.RPC, use = SOAPBinding.
                Use.LITERAL, parameterStyle = SOAPBinding.ParameterStyle.
                                                WRAPPED)
@WebService(serviceName = "calcyws")
@Service("calcy")
public class CalculatorImpl implements CalculatorIntr
{
    @WebMethod
    public String sum(int x, int y)
    {
        return "" + (x+y);
    }
}
```

Here:

@SoapBinding → Maps the given webservice details with SOAP msg protocol

@WebService → Makes current class as webservice related component.

@Service → Makes current class as spring bean.

@WebMethod → Makes current method as operation of webservice.

springCfg.xml

```
<beans ..... spring-jee-3.0.xsd spring-context-3.0.xsd>
    <context:component-scan base-package="com.satya"/>
    <bean id = "org.sf.remoting.jaxws.SimpleJaxWsServiceExporter">
        <property name="baseAddress"
                  value="http://localhost:5678"/>
    </bean>
</beans>
```

↑

starts @WebService as spring built-in webserver at 5678 port number and exposes @WebService based Java class to Registry as webservice (CalculatorImpl).

ServiceExposer.java

```
package com.satya;
import org.sf.context.support.*;
public class ServiceExposer
{
    psvm(String[] args)
    {
        FileSystemApplicationContext ctx = new
                FileSystemApplicationContext("springCfg.xml");
    }   S.O.P("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

→ Collects from spring 3.x

4. org.sf.expression-3.x.jar
5. org.sf.web-3.x.jar
6. org.sf.asm ~~expression~~-3.x.jar

7. Commons-logging.jar → Collect from spring 2.x.

* To Run the application :> java com.satya.ServiceExposer ↵

* To see the WSDL document of above web service:

→ http://localhost:5678/CalcWs

Client Application:

E:\APPS\WS

└→ clientApp

- → CalculatorStr.java
- → ServiceClient.java
- → clientCfg.xml
- → P1

CalculatorIntr.java (Develop Based on server app)

```
package p1;
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 CalculatorIntr
{
    public String sum(int x, int y);
}
```

client cfg-xml

(Gathers WSDL reference of WebService from UDDI Registry)

```
<beans ..... (same as server app) ..... >
  <bean id="gs" class="org.sf.remoting.jaxws.JaxWsPortProxy
                                    FactoryBean">
    <property name="wsdlDocumentUrl"
        value="http://localhost:8084/Calcws?wsdl"/>
    <property name="serviceName" value="Calcws"/>  w.s. logicalname
    <property name="namespaceUri" value="http://satya.com"/>  collect from WSDL doc
    <property name="serviceInterface" value="p1.CalculatorIntr"/>
    <property name="portName" value="CalculatorImplPort"/>
  </bean>
</beans>
```

Jar files (same as before)

⊕ a) org.sf.aop-3.x.jar
b) aopalliance-1.0.jar (from Internet)

serviceclient.java

```
package p1;
public class serviceclient
{
    p.s.v.m(String args[])
    {
        FileSystemXmlApplicationContext ctx = new FSXC("clientCfg.xml");
        CalculatorIntr obj = (CalculatorIntr) ctx.getBean("gs");
        S.O.println(obj.sum(10,20));
    }
}
```

15/12/11

spring security

programmers are not responsible to take care of nw security They are just 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: 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.st.web.context.ContextLoaderListener is a servlet-Context listener, which activates spring's web app Context Container either during server startup or deployment of webapp, when servletContext-obj is created.

org.st.web.filter.DelegatingFilterProxy is the predefined servlet filter that takes all the req coming to web app and passes to the springbeans that are configured in spring config file.

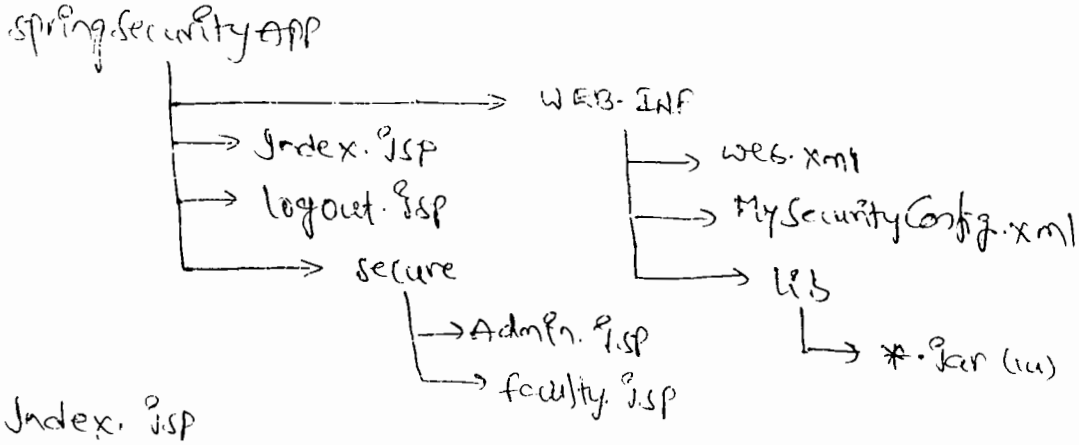
In this spring config file, we specify entries related to Authentication and authorization by defining users and their roles.

Ex App:

step ①: Keep Spring 2.5 or 3.x slw Ready

step ②: Keep spring-security-3.0.5.release zip file ready (download from Internet).

step ③: Develop spring Based web app 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 req will be traped by servlet filter.

web.xml:

```
<web-app>
  <listener>
    <listener-class> org.sf.web.context.ContextLoaderListener </listener-class>
  </listener>
  <context-param>
     <param-name> ContextConfigLocation </param-name>
     <param-value> /WEB-INF/mySecurityConfig.xml </param-value>
  </context-param>
  <filter>
     <filter-name> springSecurityFilterChain </filter-name>
     <filter-class> org.sf.web.filter.DelegatingFilterProxy </filter-class>
  </filter>
```

```
<filter-mapping>
    <filter-name> springSecurityFilterChain </filter-name>
    <url-pattern> /* </url-pattern>
</filter-mapping>
</web-app>
```

Specify url-pattern to make security filter trapping and taking all the requests, and responses.

myfacilitySecurityConfig.xml :

```
<beans:beans ... ... ... spring-security-3.0.3.xsd">
  <http use-expressions = "true" />
    <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/>
    <logout/>
    <remember-me/>
    <session-management invalid-session-url="/logout.jsp">
      <concurrency-control max-sessions="3"
                           error-if-maximum-exceeded="true"/>
    </session-management>
  </http>
  <authentication-manager>
    <authentication-provider>
      <user-services>
        <user name="satya" password="tech" authorities=
                                    "admin, faculty" />
        <user-name="nataraj" pwd="nataraj" authority="faculty"/>
      </user-services></a-p></a-m>
</beans:beans>
```

Specifying Access permissions To users on Resources

defining users and roles

-Admin.jsp

```
<h1> Admin page </h1>
welcome Mr. <%= request.getUserPrincipal().getName() %> <br>
wanna go to faculty page <a href="faculty.jsp">Click here </p>
<p> <a href="../j_spring_security_logout"> Logout </a>
```

faculty.jsp

```
<h1> faculty page </h1>
welcome Mr. <%= request.getUserPrincipal().getName() %> </br>
wanna 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 logedout <br>
<a href="index.jsp"> start again </a>
```

Jarfiles in classpath:

1. commons-logging.jar
2. spring-aop-3.0.3.RELEASE.jar
3. spring-asm-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

→ from these security related jar file

12. spring-web-3.x.jar
13. spring-webmvc-3.x.jar

lib folder

save as classpath
total (13).

* 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 sw → select Java development and select web and Jeatee development → Install. → close.

* procedure to create web project in Eclipse IDE:-

step①: Create web project

file menu → new → other → web → Dynamic web proj → next → Nani proj → next → finish.

step②: Add a JSP program to web proj (Nani proj)

Right click on project → new → other → web → JSP → next → Index → next → finish.

step③: Configure Tomcat server with Eclipse IDE.

window menu → preferences → server → Runtime environment → 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 spring

Example on annotations based autowiring

①
```
public interface Test
{ public String sayHello();
}
```

② // TestBean.java
```
import org.springframework.stereotype.*;
import org.springframework.beans.factory.annotations.*;
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 "good night" + d.toString();
  }
}
```

③ Democfg.xml
```
<beans ---- >
<context:annotation-config /> <!-- Makes spring container to recognise annotations based configs -->
<!--<context:component-scan base-package="." /> -->
<bean id="tb" class="TestBean" />
<bean id="dt" class="java.util.Date" />
</beans>
```

④ TestClient.java

*) Activate Spring container (normal manner) → get TestBean class obj & call sayHello on that obj

Jars in classpath: Spring.jar, commons-logging.jar

Note: for annotations based appcfg user defined advisors creation using annotations refer app given in page # 10 of 5th dec [illegible] 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 jdk1.1 version onwards). We use these annotations in documentation comments while generating api documentation (/** ... */)

Ex: @Author, @Param, @Return, @See..

2. Programming annotations: Available from jdk1.5 onwards. We use these annotations in stand alone apps to make jvm or JRE recognizing code easily. Ex: @Override

We use these annotations in high end apps to config resources as alternate for xml files

* All the java technologies that are given based on jdk 1.5 gives support for annotations for resources configurations like servlet 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. Because the xml parser is heavy weight c/co.
* Annotations based resources config gives good performance but bad flexibility of modifications Because 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. where 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)

@ @component / @service annotation can be used to config java class as spring Bean.

@ @autowired annotation can be used to configure bean property for autowiring.

Ex:-

6/12/11

In spring AOP while developing advices we need to make our classes implementing the spring api supplied interfaces. In aspectj programming advices can be developed as 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 prog gather the following two additional jar files from location "www.springframework.org" (aspectjrt-1.6.0.jar, aspectjweaver.jar).
* For example app on distributed transaction management using 'aspectj' on hibernate persistant logic refer app given in handout of 6/12/11
* Jar files in class path →
  - 8 → HB jar files
  - 2 → spring jar files
  - 1 → ojdbc14.jar
  - 1 → mysql-connector-java-3.0.8-stable-bin.jar
  - 2 → aspectjrt-1.6.8.jar, aspectjweaver.jar

DB table in oracle:

Account_tab1

(acno → number	— 101
acname → varchar2(20)	— raju
balance → number	— 10000.

DB table in mysql → (logical db

account_tab2

(acno → ~~number~~ int	— 101
acname → varchar2(20)	— sreeni
balance → ~~number~~ int	— 50000.

* For spring AOP aspectj based ~~adv~~ userdefined advisors 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, aspectjweaver.jar.

Attach in the handout

MyAspect.java:

```
import org.aspectj.lang.annotation.*;
@Aspect
public class MyAspect
{ @Pointcut ("execution(* TestInter.*(..))")
  private void testInterOk() {} // dummy method as advice which will be used as a reference
                                   to convey other advices.
  @Before ("testInterOk()")
  public void beforeMethod()
  {
    s.o.p ("am before advice");
  }                                  } acts as before advice
  @AfterReturning ("testInterOk()")
  public void afterReturningMethod()
  {
    s.o.p ("am after advice");
  }                                  } acts as after advice
}
```

(*) Other annotations to develop other types of advices

@AfterThrowing ⟶ to make a java method as throws advice

@Around ⟶ to make java method as around advice

Spring.xml:

```
<beans ..... xsi:schemaLocation="... spring-aop-2.5.xsd" >
<bean id="id1" class="TestBean" />
<bean id="id2" class="MyAspect" />
<aop:aspect-autoproxy /> // to convert normal spring bean class obj as aspect & based proxy obj
</beans>
```

(*) Client.java — Same as client.java of page no. 10 belongs to 5th Dec hand out

Jars in class path: spring.jar
commons-logging.jar
aspectjrt-1.6.0.jar
aspectjweaver.jar

*2 This applies throws method based throws advice of myAdvices class on demo() method of TestBean class.

```
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

<bean id="id1" class="TestBean"/>
<bean id="id2" class="MyAdvices"/>                    advice class
<aop:config>                          ← pointing to
 <aop:aspect ref="id2">  //refer bean id
   <aop:pointcut id="pt1" expression="execution(* TestInter.demo(..))"/>  // this pointcut points
   <aop:after-throwing pointcut-ref="pt1" method=                          demo method of testInter
                       "throwsMethod" throwing="nfe"/>
   <aop:pointcut id="pt2" expression="execution(long TestInter.*(..))"/>  // points to findfact()
   <aop:around pointcut-ref="pt2" method="aroundMethod"/>                  method of testInter class
 </aop:aspect>                                                             whose return type is long
</aop:config>
</beans>                                  client app
-----------client.java-------------------------
import org.springframework.beans.factory.*;
import org.springframework.context.*;
import org.springframework.context.support.*;
public class client {
  public static void main(final String[] args) {
    BeanFactory ctx = new ClassPathXmlApplicationContext("spring.xml");
    TestInter ti=(TestInter)ctx.getBean("id1");
    try
    {
    ti.demo("sathya");
    }
    catch(Exception e)
    {}
    System.out.println("---------");
    long m1 = ti.findfact(20);
    System.out.println(m1);
    System.out.println("----------");
    long m2 = ti.findfact(58);
    System.out.println(m2);

  }
}
```

*In this application on demo() method of testBean class throws advice is applied whose implementation is doen in throwsMethod() method of myAdvices class.

*By aroundAdvice is applied on findfact() method of TestBean class whose implementation is doen in aroundmethod() method of myAdvices class.

*The following Jar files in the classpath:
Spring.jar, Commons-logging.jar, Aspectjrt-1.6.0.jar, Aspectjweaver.jar

```
-----------------------------------------------------------
  App --> AspectJ annotations
-----------------------------------------------------------
-----------TestInter.java-------------  Spring interface
public interface TestInter
{
     void  demo(String name);
     long findfact(int k);
}
-----------TestBean.java-------------  Springbean class
public class TestBean implements TestInter
{
     public void demo(String name)  // B.method 1
     {
          int k =Integer.parseInt(name);
     }
     public long findfact(int k)    // B.method 2
     {
          long f=1;
          for(int i=1; i<=k; i++)
          {
               f = f * i;
          }
          return f;
     }
}
```

*3. This lines applies around method based around advice of myAdvices class on findfact() method of testBean class.

```
   FooService foo = (FooService) ctx.getBean("fooService");
   foo.getFoo("raja", 12);
   foo.getAfter();
   foo.getBefore("raja");
   }
}
-------------------------------------------------------------------
App --> Spring AOP schema support for throws and around advices
-------------------------------------------------------------------
------------TestInter.java-------------------
public interface TestInter
{
    void  demo(String name);
    long findfact(int k);
}
------------TestBean.java-------------------
public class TestBean implements TestInter
{
    public void demo(String name)
    {
        int k =Integer.parseInt(name);
    }
    public long findfact(int k)
    {
        long f=1;
        for(int i=1; i<=k; i++)
        {
            f = f * i;
        }
        return f;

    }
}
------------MyAdvices.java-------------------
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.JoinPoint;
public class MyAdvices {
 public void throwsMethod(JoinPoint jp,NumberFormatException nfe)
                                throws Throwable
 {
 System.out.println("this advice is applied for "
                    +jp.getSignature().getName());
 System.out.println("The exception occured is : "+nfe.getMessage());
 System.out.println("advice from throws method");

 }
 public Object aroundMethod(ProceedingJoinPoint pjp) throws Throwable
 {
  System.out.println("this advice is applied for "
                        +pjp.getSignature().getName());
    long x = System.currentTimeMillis();
    Object retval=pjp.proceed();
    long y = System.currentTimeMillis();
    System.out.println("The method execution taken "
                                +(y-x)+" milliseconds");
    System.out.println("The above service is from around advice");
     return retval;
 }
}
------------spring:xml-------------------
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:aop="http://www.springframework.org/schema/aop"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
1  ------------------------------------------------------------
2  App--> Spring AOP with schema support for before and after advices
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.xsdhttp://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 Transfered");
195     else
196         System.out.println("Money not Transfered");
197   }
198   }
199
200
```

E:|Apps → SpringTx → App1 → *.java (1)
*.xml (5)
*.class (5)

C:|Apps|SpringTx|App1 > javac *.java
> java clientApp

DB table :-

HB-Account

acid	number
holdername	varchar2(20)
balance	number(8)

Insert two records into HB-Account.

* Jar files in classpath :- 2- Spring Jars, 1- ojdbc14.jar and 8- HB jar 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()                 value of transferMoney() method.
143
144     }//class
145 ----------Spring.cfg.xml----------
146 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
147 "http://www.springframework.org/dtd/spring-beans.dtd">
148 <beans>
149  <bean id="myds" class="org.springframework.jdbc.datasource  // Spring bean giving JDBC
150                         .DriverManagerDataSource">            DataSource object.
151   <property name="driverClassName"> <value>
152         oracle.jdbc.driver.OracleDriver</value></property>
153   <property name="url">  <value>
154    jdbc:oracle:thin:@localhost:1521:satya</value> </property>
155   <property name="username"><value>scott</value></property>
156   <property name="password"> <value>tiger</value></property>
157  </bean>
158 <bean id="mySessionFactory" class="org.springframework
159                 .orm.hibernate3.LocalSessionFactoryBean">
160     <property name="dataSource" ref="myds"/> // Refers LOC:149
161      <property name="configLocation"><value>
162                 classpath:mycfg.xml</value></property> // Spring bean giving HBSession
163 </bean>                                  HB cfg file:    factory object.
164  <bean id="template" class="org.springframework.orm
165                     .hibernate3.HibernateTemplate">
166    <constructor-arg><ref bean=
167                 "mySessionFactory"/></constructor-arg> // Refers LOC:158
168  </bean>
169  <bean id="hbt" class="org.springframework
170           .orm.hibernate3.HibernateTransactionManager"> // Transaction manager that can
171       <property name="sessionFactory" ref="mySessionFactory"/> perform tx mgmt on HB
172  </bean>                                  Refers LOC:158    persistance logic.
173
174   <bean id="tt1" class="org.springframework.transaction
175                         .support.TransactionTemplate">
176     <property name="transactionManager"><ref  bean="hbt"/></property>
177   </bean>                                          Refers LOC:169
178   <bean id="db" class="DemoBean">
179*    <property name="ht"><ref bean="template"/></property> // Refers LOC:169
180*         <property name="tt"><ref bean="tt1"/></property> // Refers LOC:174
181   </bean> * Here two objects are injected to our Spring Bean class properties.
182 </beans>
183 ----------ClientApp.java----------
184 import org.springframework.context.support.*;
185 public class ClientApp
186 {
187   public static void main(String args[ ])
188   {
189     FileSystemXmlApplicationContext ctx=
190      new FileSystemXmlApplicationContext("SpringCfg.xml");
191          Demo  bean =(Demo)ctx.getBean("db");
192          boolean res=bean.transferMoney(101,102,3000);
```

```
        <property name="holdername" column="HOLDERNAME"/>
        <property name="balance" column="BALANCE"/>
    </class>
</hibernate-mapping>
----------Demo.java---------- Spring interface
//Demo.java
public interface Demo
{
  public boolean transferMoney(int srcid,int destid,float amt); // Declaration of the B. method.
}
----------DemoBean.java---------- Spring bean
//DemoBean.java
import org.springframework.transaction.support.*;
import org.springframework.transaction.*;
import org.springframework.orm.hibernate3.*;

public class DemoBean implements Demo
{
    TransactionTemplate tt;
    HibernateTemplate ht;
    public void setHt(HibernateTemplate ht) // setXxx() methods supporting setter injection.
    {
        this.ht=ht;
    }
    public void setTt(TransactionTemplate tt)
    {
        this.tt=tt;
    }
    public float fetchBalance(int acid)
    {
     Account ac1=(Account)ht.get(Account.class,new Integer(acid));
        return ac1.getBalance();
    }
  public boolean transferMoney(final int srcid,final int destid,final float amt)
    {

     Boolean result=(Boolean)tt.execute(new TransactionCallback()
        {
          public Object doInTransaction(TransactionStatus ts)
          {
            boolean status=false;

          try
          {
            //transferMoney Logic
          int r1=ht.bulkUpdate("update Account
          a1 set a1.balance=a1.balance-? where a1.acid=?",   // HQL update query performing withdraw amt on source a/c. (operation1)
          new Object[]{new Float(amt),new Integer(srcid)});

          int r2=ht.bulkUpdate("update Account a1 set
            a1.balance=a1.balance+? where a1.acid=?",   // (operation2) depositing amount in destination a/c.
            new Object[]{new Float(amt),new Integer(destid)});

              if(r1==0 || r2==0)
              {
               status=false;
               ts.setRollbackOnly();
               System.out.println("Tx is rolledback");
              }
              else
              {
                System.out.println("Tx is committed");
                            status=true;
              }
```

* To make outer class parameters visible to innerclass defined in that class the parameters must be taken as final.

App1 (programatic Local transaction management of HB persistance logic

```
<!--mycfg.xml-->
<!DOCTYPE hibernate-configuration PUBLIC
 "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">
            oracle.jdbc.driver.OracleDriver</property>
          <property name="hibernate.connection.url">
       jdbc:oracle:thin:@localhost:1521:satya</property>
          <property name="hibernate.connection.username">
                        scott</property>
          <property name="hibernate.connection.password">
                        tiger</property>
          <property name="hibernate.dialect">
          org.hibernate.dialect.Oracle9Dialect</property>
          <property name="show_sql">true</property>
    <mapping resource="Account.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
----------Account.java----------
public class Account          HB POJO class
{
     int acid;
     String holdername;
     float balance;

     public void setAcid(int acid)
     {
          this.acid=acid;
     }
     public int getAcid()
     {
          return acid;
     }
     public void setHoldername(String holdername)
     {
          this.holdername=holdername;
     }
     public String getHoldername()
     {
          return holdername;
     }
     public void setBalance(float balance)
     {
          this.balance=balance;
     }
     public float getBalance()
     {
          return balance;
     }
}
----------Account.hbm.xml---------- mapping file.
<!DOCTYPE hibernate-mapping PUBLIC
     "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
     "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
     <class name="Account"  table="HB_Account">
          <id name="acid" column="ACID" >
            <generator class="increment"/>
     </id>
```

PROPAGATION_REQUIRED PROPAGATION_NOTSUPPORTED
PROPAGATION_REQUIRESNEW PROPAGATION_NEVER
PROPAGATION_SUPPORTS PROPAGATION_MANDIATORY

```
  <property name="url"> <value>
    jdbc:oracle:thin:@localhost:1521:satya</value> </property>
  <property name="username"><value>scott</value></property>
  <property name="password"> <value>tiger</value></property>
 </bean>

<bean id="mySessionFactory" class="org.springframework.orm
                .hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myds"/>
    <property name="configLocation"><value>
                classpath:mycfg.xml</value></property>
</bean>

<bean id="template" class="org.springframework
                .orm.hibernate3.HibernateTemplate">
   <constructor-arg><ref bean="mySessionFactory"/></constructor-arg>
</bean>

<bean id="hbt" class="org.springframework.orm.hibernate3
                .HibernateTransactionManager">
     <property name="sessionFactory" ref="mySessionFactory"/>
 </bean>
  <bean id="db" class="DemoBean">
    <property name="ht"><ref bean="template"/></property>
  </bean>
    <bean id="tas" class="org.springframework.transaction
          .interceptor.NameMatchTransactionAttributeSource">
        <property name="properties">
          <props>
            <prop key="transferMoney">PROPAGATION_REQUIRED</prop>
          </props>
        </property>
  </bean>
  <bean id="tfb" class="org.springframework.transaction
                .interceptor.TransactionProxyFactoryBean">
    <property name="target"><ref bean="db"/></property>
    <property name="transactionManager"><ref bean="hbt"/></property>
    <property name="transactionAttributeSource">
                <ref bean="tas"/></property>
  </bean>
</beans>
```

gives DemoBean class object as proxy object by enabling tx service on it.

----------ClientApp.java----------

```
import org.springframework.context.support.*;
public class ClientApp
{
 public static void main(String args[ ])
 {
        FileSystemXmlApplicationContext ctx=new
             FileSystemXmlApplicationContext("SpringCfg.xml");
          Demo  bean =(Demo)ctx.getBean("tfb");
          try
          {
          boolean res=bean.transferMoney(101,102,3000);
          if(res)
               System.out.println("Money Transfered");
          else
               System.out.println("Money not Transfered");
          }
          catch(Exception e)
          {
               System.out.println("Money not Transfered");
     }
     }
}
```

* 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 transfers money always runs with new transaction because the transaction attribute "required"

(46) ←

```
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                                        throws Exception;
79  }                                      Runtime
80  ----------------------------DemoBean.java--------------------------------
81  //DemoBean.java
82
83  import org.springframework.transaction.support.*;
84  import org.springframework.transaction.*;
85  import org.springframework.orm.hibernate3.*;
86
87  public class DemoBean implements Demo
88  {
89    HibernateTemplate ht;
90    public void setHt(HibernateTemplate ht) // setxxx() methods supporting setter injec
91    {
92       this.ht=ht;
93    }
94    public boolean transferMoney( int srcid, int destid, float amt)
95                                        throws Exception
96     {                                  Runtime
97      boolean status=false;
98      int r1=ht.bulkUpdate("update Account a1 set a1.balance  // operation 1 (to withdraw am
99              =a1.balance-? where a1.acid=?",new Object[]      from source account)
100                    {new Float(amt),new Integer(srcid)});
101     int r2=ht.bulkUpdate("update Account a1 set a1.balance  // operation 2 (deposite the amo
102                    =a1.balance+? where a1.acid=?",           into destination a/c)
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 new Exception();
115                  }          Runtime
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  // Spring Bean giving jdbc datasou
126              .datasource.DriverManagerDataSource">   object.
127    <property name="driverClassName"> <value>
128      oracle.jdbc.driver.OracleDriver</value></property>
```

04/12/11

K. Ravi +91-8019875108

Spring With Hibernate Apps With Transaction Mgmt.

App2 (Local declarative Tx management by using HB Tx manager):

```
----------mycfg.xml----------
<!DOCTYPE hibernate-configuration PUBLIC
 "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
 "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
 <session-factory>
     <property name="hibernate.connection.driver_class">
           oracle.jdbc.driver.OracleDriver</property>
     <property name="hibernate.connection.url">
        jdbc:oracle:thin:@localhost:1521:satya</property>
     <property name="hibernate.connection.username">
                        scott</property>
     <property name="hibernate.connection.password">
                        tiger</property>
     <property name="hibernate.dialect">
         org.hibernate.dialect.Oracle9Dialect</property>
     <property name="show_sql">true</property>
     <mapping resource="Account.hbm.xml"/>
     </session-factory>
</hibernate-configuration>

----------Account.java----------
public class Account
{
     int acid;
     String holdername;
     float balance;

     public void setAcid(int acid)
     {
          this.acid=acid;
     }

     public int getAcid()
     {
          return acid;
     }
     public void setHoldername(String holdername)
     {
          this.holdername=holdername;
     }
     public String getHoldername()
     {
          return holdername;
     }
     public void setBalance(float balance)
     {
          this.balance=balance;
     }
     public float getBalance()
     {
          return balance;
     }
}
----------Account.hbm.xml----------
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
     <class name="Account"  table="HB_Account">
```

Jar files in classpath:- 8 - HS jar file
2 - Spring jar files
1 → ojdbc14.jar
1 → mysql-connector-java-3.0.8-stable-bin.jar
2 → aspectjrt-1.6.8.jar, aspectjweaver.jar

DB table in oracle:-

Account-tab1

acno (n)	acname (vc2)	balance (n)
101	RAJA	90000

DB table in mysql:-

Account-tab2 (Logical DB name dbj)

acno (Integer)	acname -vc	balance (m)
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 Lno: 92
146      <property name="mappingResources">  // gives HB SessionFactory object pointing to
147          <list>                                  mySQL
148              <value>account2.hbm.xml</value>
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"> // HBTemplate class object
159    <property name="sessionFactory"  ref="aliaSessionFactory"/> // refer Lno.125  pointing to oracle.
160   </bean>
161
162   <bean id="temp2"  class="org.springframework.orm.hibernate3.HibernateTemplate">  // HBTemplate class object
163    <property name="sessionFactory"  ref="betaSessionFactory"/> //refer Lno: 144      pointing to mySQL
164   </bean>
165
166
167   <bean id="tb"  class="testbean">  // Injecting two hibernate template class objects to our
168    <property name="ht"> <ref  bean="temp1"/> </property>          Spring bean class.
169    <property name="ht1"> <ref  bean="temp2"/> </property>
170   </bean>
171
172    <tx:advice id="txAdvice" transaction-manager="txmgr"> // refer Lno: 127
173       <tx:attributes>
174          <tx:method name="*" propagation="REQUIRED"/> // The transaction attribute required is applied on
175       </tx:attributes>                                    B methods of Spring bean class.
176     </tx:advice>
177
178     <aop:config>
179       <aop:pointcut id="dtxops" expression="execution(* testinter.*(..))"/> // pointcut Configuration pointing to
180       <aop:advisor advice-ref="txAdvice" pointcut-ref="dtxops"/>              the spring bean (testBean) that
181     </aop:config>                refer Lno: 172      179                    implements testinter interface.
182
183  </beans>
184  ----------client.java----------  client App
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 Lno: 167
196          ob.transferMoney(101,102,3000); // calls the B method from client application without Tx,so,
197
198      }   the transferMoney() method always runs with new tx because the transaction
199                                                                    attribute that is Configured is required.
200  }
201  ----------
```

* talks about Aspect style of Configuring transaction Service on Spring Bean class B methods.

* At Lno: 180 the <aop:Advisor> tag is applying transaction Service Lno: 172 on pointcut based Spring Specified at Lno: 179.

```
70      {
71          int r1=ht.bulkUpdate("update Account ac set ac.balance=ac.balance-? where ac.acno=?",  // operation1 withdraw
72                                  new Object[]{new Double(amt),new Integer(srcid)});
73          int r2=ht.bulkUpdate("update Account ac set ac.balance=ac.balance+? where ac.acno=?",
74                                  new Object[]{new Double(amt),new Integer(destid)}); // operation2
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----------
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>
100   </property>
101   <property name="url">
102   <value>jdbc:oracle:thin:@localhost:1521:satva</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>
115  </property>
116  <property name="url">
117  <value>jdbc:mysql:///mydb1</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"/>
128
129  <bean id="alfaSessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
130      <property name="dataSource" ref="drds1"/>
131      <property name="mappingResources">
132    <list>
133        <value>account1.hbm.xml</value>
134
135      </list>
136    </property>
137    <property name="hibernateProperties">
138        <value>
```

Runtime

deposit operation on dest a/c of mysql DB s/w

Spring cfg file

// schema related xml Entries.

// bean giving jdbc DataSource object pointing to oracle.

// bean giving jdbc DataSource object pointing to mysql

// Special tx manager to perf distributed tx mgmt.

// refers LNO: 97

// gives HB SessionFactory object pointing to oracle.

05/12/11 K. Ravi +91-8019875108

Application on Distributed transactions.

```
1  ----------Account.java----------
2  public class Account {                    HB pojo class
3      private int acno;
4      private String acname;
5      private double balance;
6      public int getAcno() {
7          return acno;
8      }
9      public void setAcno(int accno) {
10         this.acno = accno;
11     }
12     public String getAcname() {
13         return acname;
14     }
15     public void setAcname(String acname) {
16         this.acname = acname;
17     }
18     public double getBalance() {
19         return balance;
20     }
21     public void setBalance(double balance) {
22         this.balance = balance;
23     }
24
25 }
```

* Declarative distributed tx mgmt on mysql, oracle DB s/w using AspectJ.

```
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"/>
33    <property name="acname"/>
34    <property name="balance"/>
35  </class>
36 </hibernate-mapping>
```

↳ source table name hold source A/c details.

```
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>
```

destination table holding dest A/c details.

```
48 ----------testinter.java---- Spring interface.
49
50 public interface testinter {
51     public void transferMoney(int srcid,int destid,double amt)throws Exception;
52
53 }
```

Runtime

```
54 ----------testbean.java---- Spring Bean
55 import org.springframework.orm.hibernate3.*;
56 public class testbean implements testinter
57 {
58     HibernateTemplate ht;
59     HibernateTemplate ht1;
60     public void setHt(HibernateTemplate ht)
61     {
62         this.ht=ht;
63     }
64     public void setHt1(HibernateTemplate ht1)
65     {
66         this.ht1=ht1;
67     }
68
69     public void transferMoney(int srcid,int destid,double amt)throws Exception
```

Runtime

// two HB template class objects will be injected pointing to oracle, mysql db s/w.

For example application local transaction management or HB persistance 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 environment?

Ans: The B.method catch block catch that checked Exception and rethrow as unchecked Exception.

In Distributed transaction management multiple DBs r 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 implementing the Spring api supplied interfaces.

In AspectJ programming Advices can be developed as 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 web site.

* aspectjrt-1.6.0.jar
* aspectjweaver.jar

For Example application on distributed transaction management using AspectJ or Hibernate persistance logic refer application given in handout of 05/12/11

Summary table on Transaction attributes:-

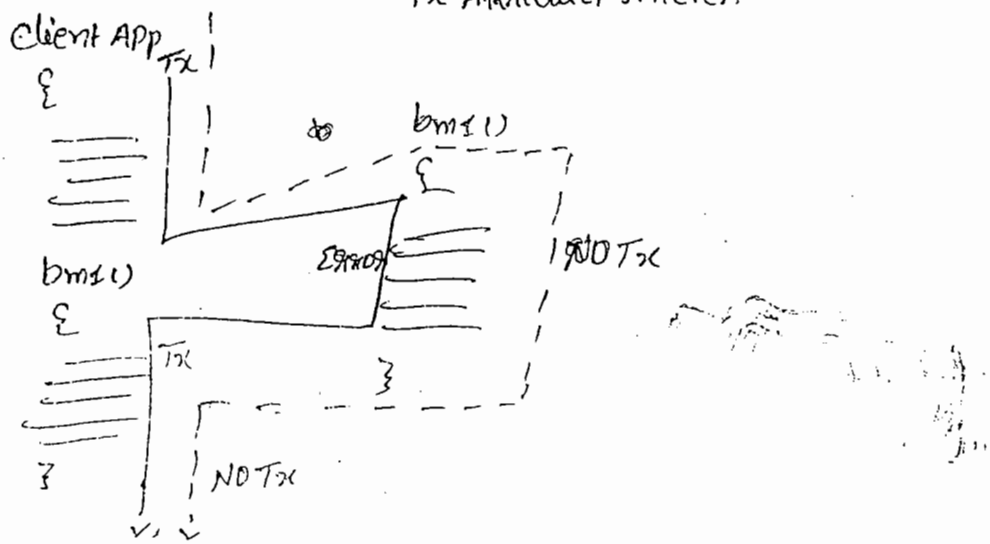
Transaction attribute	clients Transaction	Bean's Transaction
Required	none	T_2
	T_1	T_1
Requiresnew	none	T_2
	T_1	T_2
Supports	none	none
	T_1	T_1
mandatory	none	error
	T_1	T_1
NotSupported	none	none
	T_1	none
Never	none	none
	T_1	error

* T_2 is Bean's transaction indicates that our B. method runs new T_x.
* T_1 in Bean's transaction indicates that our B. method runs in the client started T_x.
* For Example app or Local declarative T_x mgmt on spring DAO persistance logic refer Application(1) of the page no's 130 to 131
* In declarative transaction management the transaction manager don't rollback transaction with B. method raises checked Exception. but it rollbacks the transaction when B. method raises unchecked Exception. (RuntimeException & it's Subclasses)

Ravi

* 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:

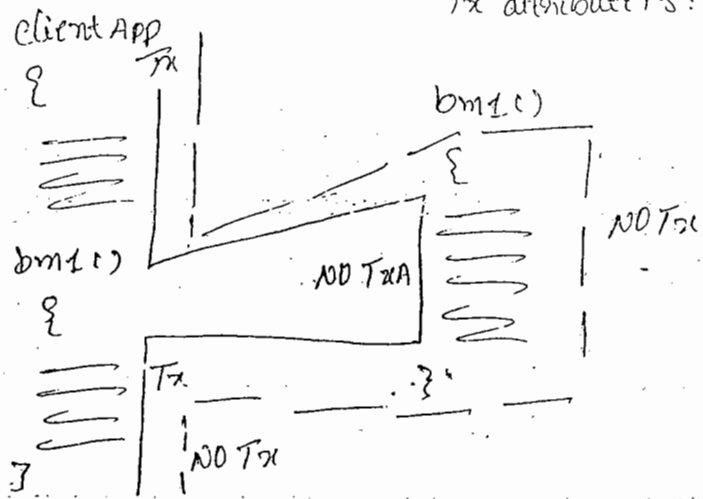
Tx Attribute is: never



This indicates client must not call the B. method of spring Bean with Tx. Other wise B. method raises the Exception.

NOT Supported

Tx attribute is: Not Supported.

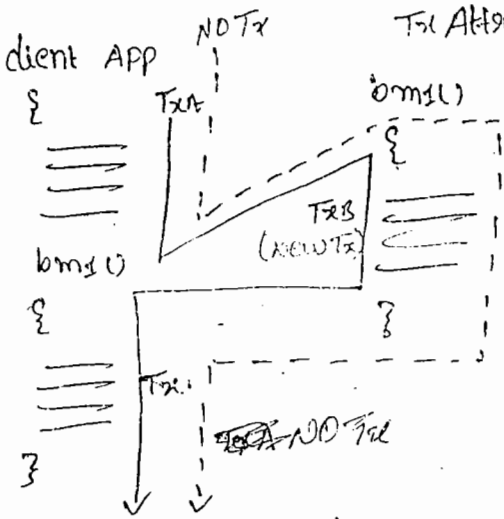


This indicates B. method runs without Tx irrespective of weither client is calling B. method with or without Tx.

In Declarative Tx mgmt the tx manager rollbacks the transaction with B. method through runtime Exception. Otherwise it Commits the transaction.

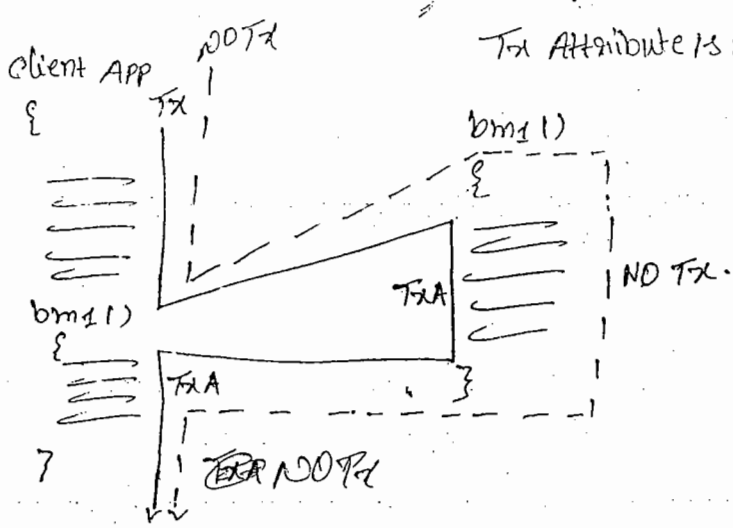
Requires New;-

* makes B.method always running with new Tx irrespective weither that B.method is called with or without Tx by client App.

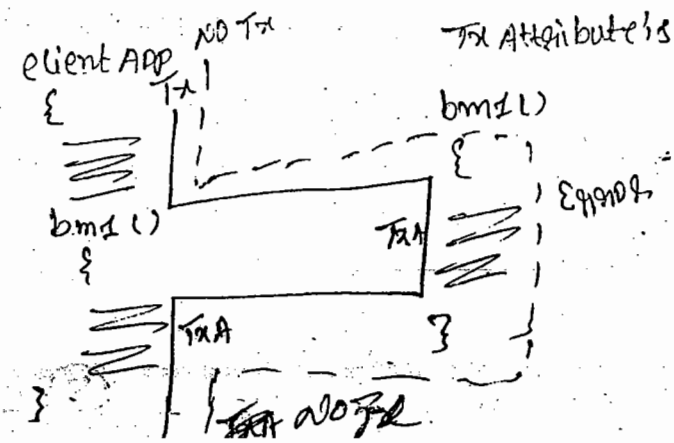


Supports;- If client calls the B.method with Tx then B.method also runs in the same Tx.

* If client calls B.method without Tx then B.method also runs without Tx.



mandiatory;- This indicates the client must calls the B.method of Spring Bean with Tx otherwise Error will be raised during the B.method execution.



(22) ←

never The transaction attribute must configure the B. method.

Supports (default)

NotSupported.

In declarative transaction management use org.sf.transaction.interceptor.

TransactionProxyFactoryBean that gives proxy object by application Tx service on B. methods.

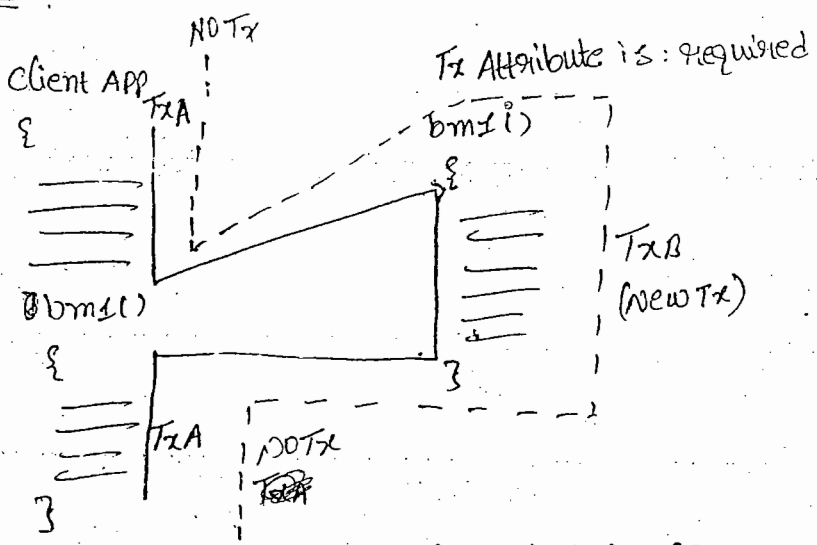
To link cfg transaction attributes on B.methods we can use a pointcutadvisor called org.sf.transaction.interceptor.NameMatchTransactionAttributeSource.

Eg/11

Understanding Tx attribute behaviour of declarative Tx mgmt:-

Note:- In programatic tx mgmt there is no need of working with tx attributes.

required:-



bm1() is the B.method of Spring Bean.

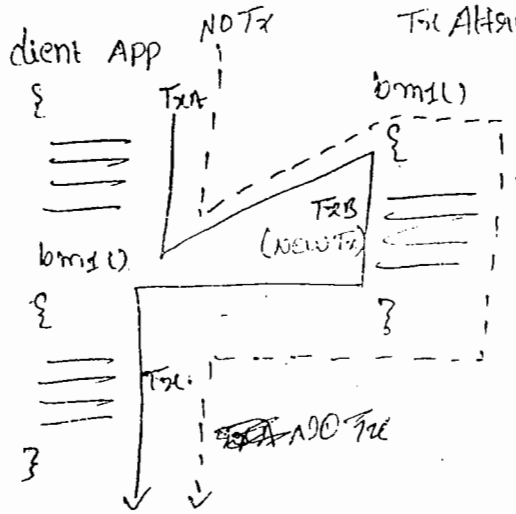
— Client APP is calling B.method with Tx.

--- 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 weither that B.meth is called with or without Tx by client App.



Tx Attribute is: Requires New

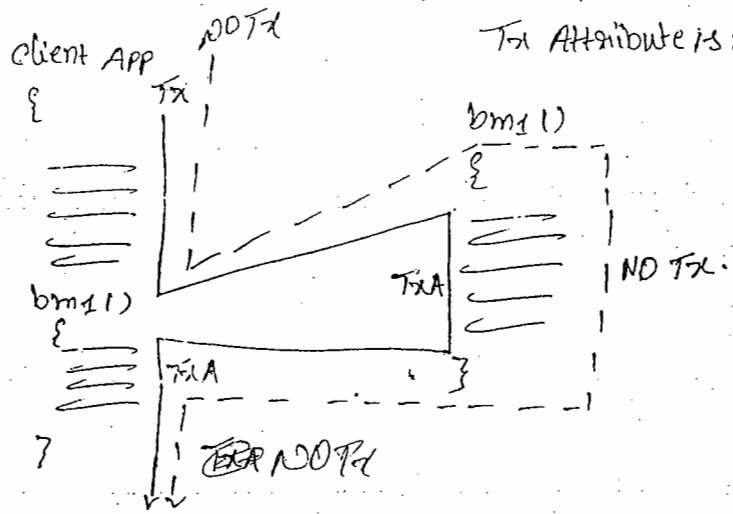
— client App is calling B.method with Tx

--- client app is calling B.method without Tx

bm1() is the B.method of Spring Bean

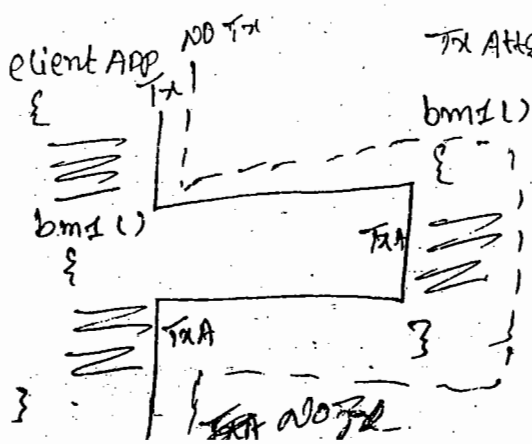
Supports ;- If client calls the B.method with Tx then B.method also runs in that same Tx.

* If client calls B.method without Tx then B.method also runs without Tx.



Tx Attribute is: Support.

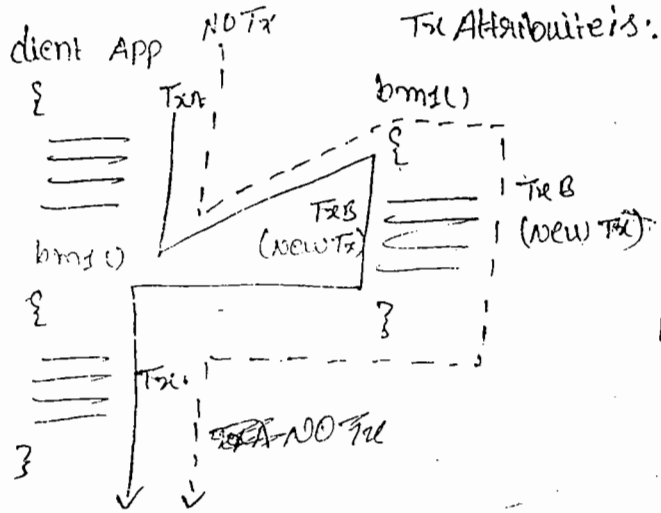
mandiatory ;- This indicates the client must calls the B.method of Spring Bean with Tx otherwise Error will be raised during the B.method execution.



Tx Attribute is : mandiatory

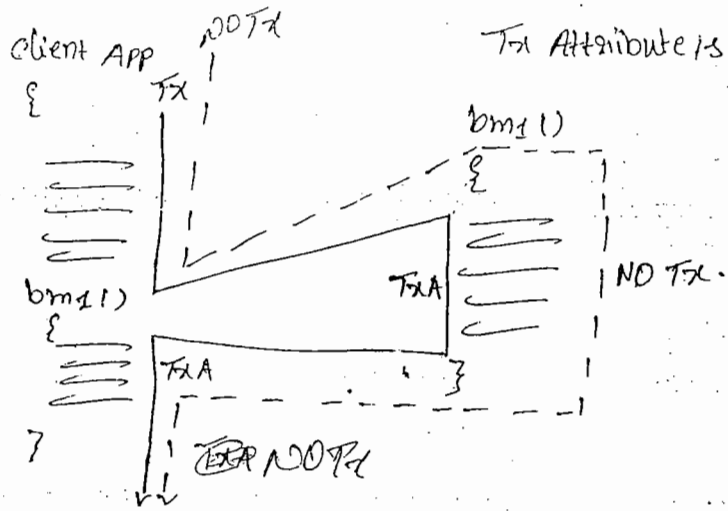
Requires New:

* makes B.method always running with new Tx irrespective weither that B.meth is called with or without Tx by client App.

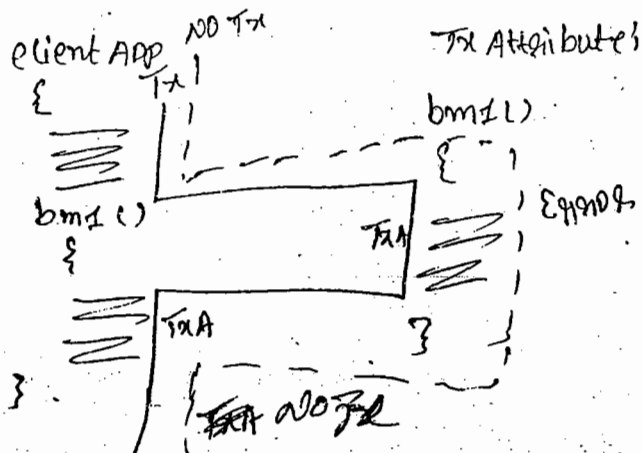


Supports:- If client calls the B.method with Tx then B.method also runs in that same Tx.

* If client calls B.method without Tx then B.method also runs without Tx.



mandiatory:- This indicates the client must calls the B.method of Spring Bean with Tx otherwise Exception will be raised during the B.method execution.



* EJB, Hibernate — e transaction attribute that configured the B.method.
nested Tx's. — III)

23/12/11

* Tx. — sation management use org.sf.transaction.interceptes.
T. — Spring Bean — ybean that gives proxy object by application Tx
*

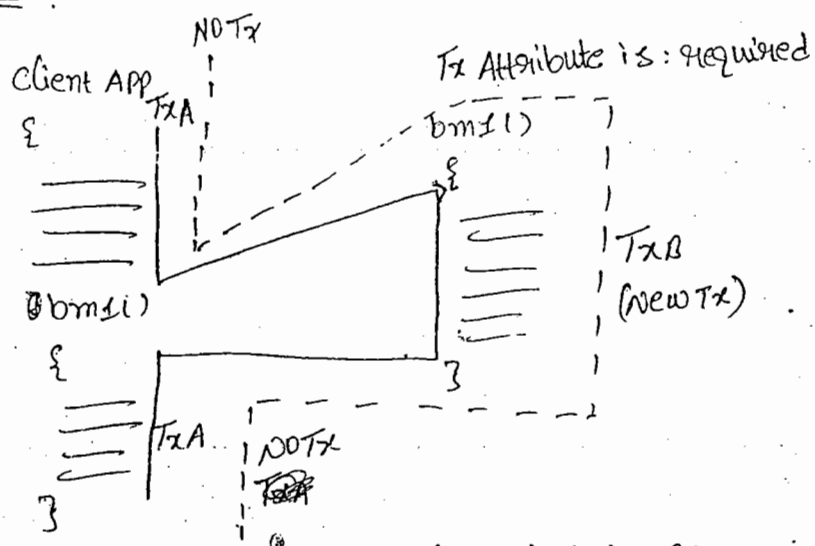
: To link cfg transaction attributes on B.methods we can use a pointcutadvisor called org.sf.transaction.interceptes.NameMatchTransactionAttributeSource.

24/12/11

* Understanding Tx attribute behaviour of declarative Tx mgnt:-

Note:- In programatic tx mgnt there is no need of working with tx attributes

Required:-



bm1() 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

Compile & execute the client app.

client > javac *.java
> java HelloClient

Note: Make sure that spring batch domain server of weblogic is in running mode while executing the above client app.

* While developing plain EJB client app we must gather following details from service provider (the EJB component developers)
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.sf.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 at client 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

2: Develop spring based EJB client app as shown below.

C:\app1\ses4\SpringEjbClient
- Demo.class
- DemoBean.class
- ejb-cfg.xml
- DemoClient.class
- myHello.class (copy from ses4\servers EJB component)

3: 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 stateful session bean components, entity bean components & mdb components

* To execute certain task of app at specific time (or) repeatedly at regular intervals we need scheduling. this scheduling allows us to enable timers on our apps

* Using spring gee module we cannot develop EJB component in spring style - Bcoz to execute EJB component EJB container is required & spring containers cannot act as EJB containers. But we can develop the client app of EJB component in spring style by getting abstraction on plain EJB based client app development process

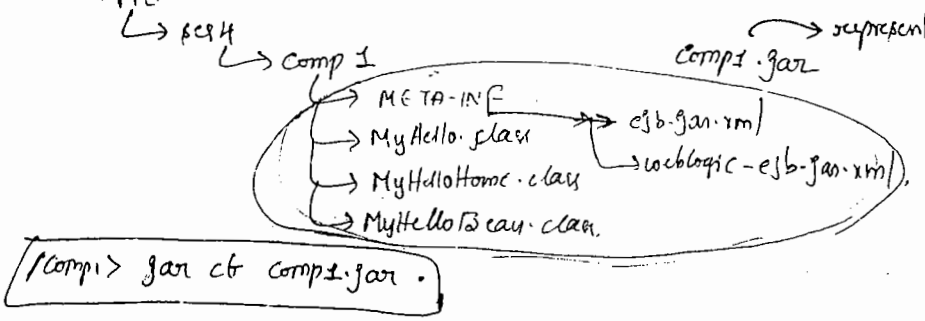
* for example app on plain EJB component and plain EJB client app refer page # 88 to 89 app (88)

Procedure to deploy plain EJB 2.x component of page # 88 & 89 in weblogic 10.3 ver

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:\Apps



13/11/11

step 3: start Spring Batch domain server of weblogic

start → programs → oracle w.l → user projects → SpringBatch domain → start admin server

step 4: Deploy the above developed EJB Component in springbatch domain server.

step open browser window → http://localhost:2020/console → deployments → install

step → upload ur files → 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 bookleft

→ Procedure to execute plain EJB client app i.e, given in 89 & 90 (HelloClient.java)

step 1: Develop the client app

E:\apps\ses4\client → HelloClient.java

step 2: Copy remote Interface file (MyHello.class), Home Interface file (MyHelloHome.class) to client side. (Copy to E:\apps\ses4\client folder from comp1 directory of Ses4 Server

step 3: Add weblogic.jar to class path

12/11/11

For spring RMI app that uses spring DAO module based persistance logic refer app 16 of the booklet page no # 64 & 65

+ while developing spring gee module based distributed apps we can use any other spring module (or) any other technology support to develop b.logic & persistance logic in server 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 apps cannot use Internet n/w as communication channel b/w client & server apps (Because of it internally O.S sockets 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 registry s/w
* Suitable for developing large scale distributed apps
* Allows to use Internet n/w as communication channel.

Q: What is the difference b/w JavaBean & EJB?

Ans:

JavaBean	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 weigh B.component
* Useful as helper resource in model layer (DTO, HIBPOJO & etc.....)	* Can be used in model layer to develope B & persistance logics

* The org.sf.remoting.rmi.RmiProxyFactoryBean can inject Rmi B.Object reference to specified bean property of specified bean class by gathering that B.Object reference from Rmi registry.

* The important 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.

1/11/11

* For Spring Rmi based distributed application refer the app(15) of the page NO's 62 to 64.

* The Spring Rmi application can utilized the already started Rmi registry of at certain port number ~~and~~ or can start new Rmi register Explicitly at specified port number.

* while developing Spring Rmi applications ~~we~~ the service client needs to gather following details from service provides.

  ⟶ 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 ~~App~~ 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 copyed to the spring rmi client application.

(21) ←

(8) 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 Benifits 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 & POJI classes.
  - ⇒ No need of generating Stub file Explicitly.
  - ⇒ Starts / Locates Rmi Registry internally (No need of working with Rmi registry Explicitly)
  - ⇒ ~~No need of testing~~ 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 ~~through~~ through IOC or dependency Injection process.

* The org.sf.remoting.rmi.RmiServiceExporter can registers given B.object with Rmi registry having nick name by starting rmi registry &, generating Stub file.

* The important Bean properties are
  - * Service ⟶ B.object bean id.
  - * ServiceInterface ⟶ name of the Service interface / B.interface.
  - ~~* Serviceport ⟶ port number of Rmi registry.~~
  - * ServiceName ⟶ nick name B.object.
  - * registryport ⟶ port number 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/n client & Server apps.

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 it's 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 by the Stubfile 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 ie Rmi client & Server applications are interacting with ~~each~~ eachother.

~~keep change~~ we can change the port number of Rmi Registry as shown bellow

> start rmiregistry 2222 new port number.

As a Service provider who develops Rmi client application from Remote place. u must gather following details from Service provider who develops Server application.

D A copy of stub file.

D A copy of Service interface file.

D Ip address/ host name and port number details Rmi Registry.

The nick name Assigned to B.object reference.

10/11/11

// FactClientApp.java :-

```
import java.rmi.*;
public class FactClientApp
 {
  public static void main(String args[]) throws Exception
  {
    // get B.Object from Rmi Registry through lookup operation
   Fact bobjref = Naming.lookup ("rmi://localhost:1099/india");
   // call B.method on B.Object reference.
   Long result = bobjref.findFactorial(5);
   System.out.println("The Factorial value is:"+result);
  }
}
```

Steps for Executions :-

Step ① :- Compile resources of Server side.

E:\APPS\Ses1\Server> javac *.java

Step ② :- generate stub file based on implementation class.

E:\APPS\Ses1\Server> rmic FactImpl

—> gives FactImpl_Stub.class file.

Step ③ :- Start rmi registry.

E:\APPS\Ses1\Server> start ~~rmiregistry~~ rmiregistry

↳ to start rmiregistry service in a new window.

Step ④ :- Run the Server application

E:\APPS\Ses1\Server> java FactServerApp

Step ⑤ :- copy the service interface(Fact.class) Stubfile (FactImpl_Stub.class) files to client folder from Server folder.

Step ⑥ :- Compile the resources of client side

E:\APPS\Ses1\client> javac *.java

Step ⑦ :- Run the client application.

```
    // 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;
    }
  }
```

// FactServerApp.java :-

```
  public class FactServerApp
  {
   P.S.V. main (String args[]) throws Exception
    {
    // create obj for Impl class (B.obj)
    FactImpl bobj = new FactImpl();

  // bind b.obj reference with Rmi registry.
  Naming.bind ("rmi://localhost:1099/india", bobj);
```

(protocol; hostname & port no of rmi registry; nickname for; obj to be bound.)

```
  S.o.pln ("Bobj is bound and Server B.obj 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 where as the methods of Remote object can be invoked from both Local & remote clients.

* when Java class implements any 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 to it's implementation class is called as marker interface.

Eg:- Java.io.Serializable.
Java.rmi.Remote.
Java.lang.Cloneable
Java.lang.Runnable and etc.

* A marker interface can be there with or without methods.

Fact.Java

```
import java.rmi.*;
public interface Fact extends Remote
{
  public Long findFactorial(int val) throws RemoteException;
}
```
declaration of a method.

// FactServerApp.java :-

```
import java.rmi.*;
import java.rmi.server.*;
class FactImpl extends UnicastRemoteObject implements Fact
{
public FactImpl() throws RemoteException
 {
 S.o.pln("0-arg Constructor of FactImpl class");
}
```

allows the objects of its subclass registerable with RmiRegistry.

UnicastRemoteObject

/* If Superclass Constructor throws Exception the Sub class Constructor can not catch and ~~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 s/w's of distributed application multiple B.Component references will be placed and that registry s/w will reside in fixed location.
* Rmi, EJB, CORBA can be used to develop non web based distributed application.
* http invokes and webservices can be used to develop web based distributed applications.
* In distributed application development foure important parties will be involved.

Service provider ——→ develops Server application having B.Comp/B.obj.
——→ Binds B.obj ref with registry s/w.

Service clients ——→ develops client application calling B.methods.
——→ gets B.obj ref from registry to lookup operation

Service interface ——→ understanding document b/n Service provider & Service client having declaration of B.methods.
——→ It is an Java interface.

Registry s/w ——→ manages B.object ref having global visibility.
——→ makes distributed applications as Location transperancy ap

plain Rmi :-

* Built-in technology in Jdk s/w.
* Uses Rmi registry as registry s/w. (default port No: 1099)
* The core Rmi api is Java.rmi package & Java.rmi.server package.
* Basic Jdk s/w is enough to develop Rmi applications.

plain Rmi application :-

E:\Apps\ses1
- Server
  - Part.java (Service interface)
  - PartAppServer.java (Server app)
- Client
  - PartClientApp.java (Client app)

* Rmi application the B.objects of Server 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 Transperancy is called as distributed application, Rmi, EJB, CORBA, webservices are the distributed technologies to develop distributed applications.

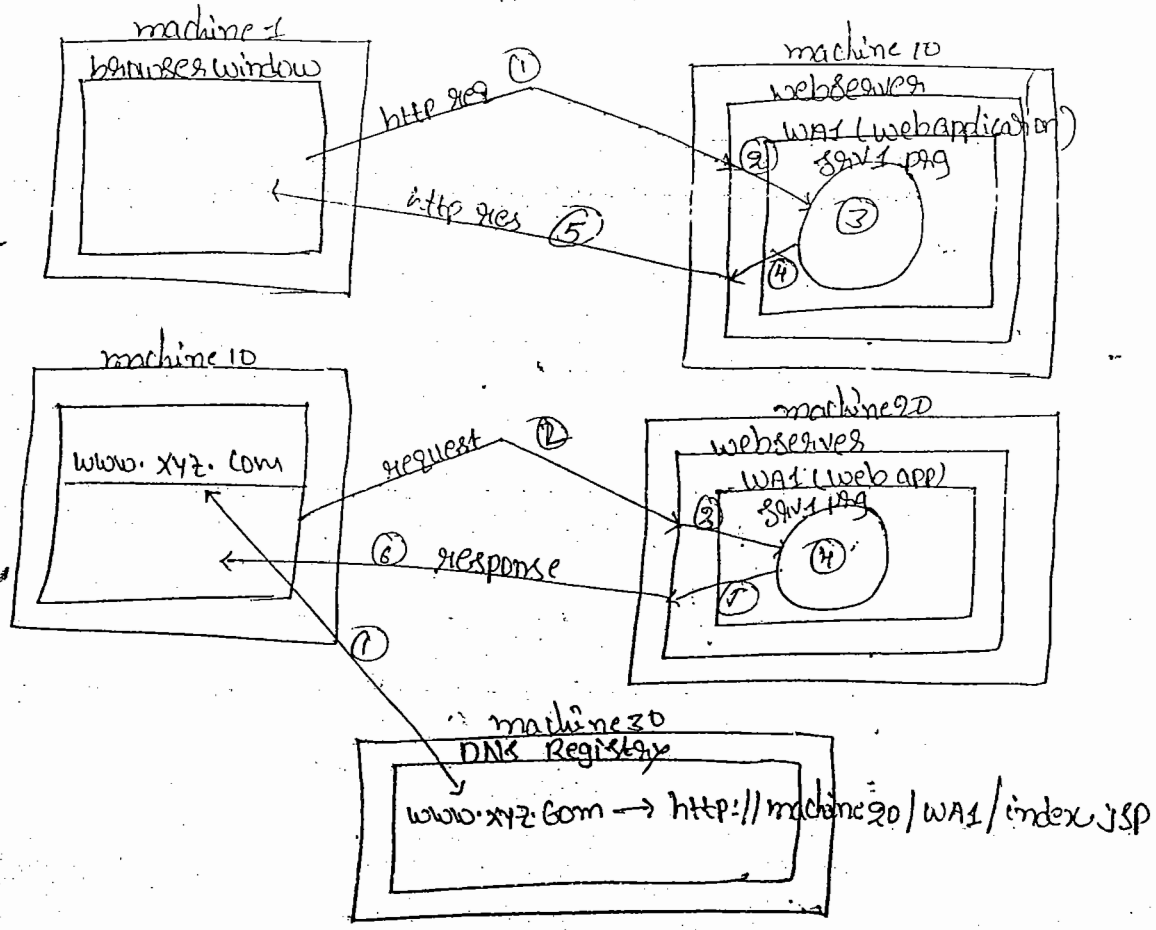
* A Distributed application can be developed either as web application or Non-webapp

* The web application with DNS registry s/w 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 classroomlevel (NO DNS registry support) is called as client Server application.



web application as Distributed application

```
//write b.logic
   intramt = (pamt*rate*time)/100;
// write the persistance logic DAO class
   HbCustomer hc = New HbCustomer();
      ~~hc.SetCname~~
      hc.SetCname(cname);
      hc.Setpamt(new Integer(pamt));
      hc.SetIntramt(new Integer(intramt));
       dao.save(hc);
   // return total amt
      return (pamt+intramt);
   } //method.
}
```

Step (10) :- Configure the above created SpringBean class in SpringHB.xml file.

```
<bean id="mb" class="modelBean">
   <property name="dao" ref="HbCustomerDAO" />
</bean>
```

* injects the IOE generated DAO class object to our SpringBean class project.

Step (11) :- Since increment algorithm can not generate object type identifier 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 ② 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.

Re on webRoot folder → new → html : input.html

```
<form action="Controller.1" method="get" >
   <b> Name : <input type="text" name="tname" /> <br>
   <b> pamt : <input type="text" name="tpamt" /> <br>
```

use pattern of Controller Servlet program

Step (13)

Step④:- perform HB reverse Engineering on HB-Customer table of the above created oracle DB profile.

go to DB BROWSER window → RC on oracle profile → open Connection → Expand oracle → Expand Connected to oracle → scott → table → Hb-Customer RC on Hb-Customer table → hibernate reverse Engineering → java Source folder: [/SpringHBAPP3/src]

- ☑ create POJO ----
- ☑ Java data object ----
- ☐ Deselect create abstract class
- ☑ Java Data Access Object

DAO type : ⊙ Spring DAO

Spring cfg file : src/springHB.xml

Session Factory ID : SessFact

→ Next

Type mapping : ⊙ Hb Types

ID generator : [Increment] → Next → finish

* the above step generates mapping file [HbCustomer.hbm.xml) POJO class (HbCustomer.java) and DAO class (HbCustomerDAO.java)

Note:- Add Connection.autoCommit property in hibernate.cfg.xml file.

<property name="connection.autocommit"> true </P>

Step⑤:- Add Spring resources to the project (model.java, modelbean.java)

model.java:- RC src → new → Interface → [model.java]

```
public interface model
 {
   public int CalcIntrAmt (String cname, int pamt, int time, int rate);
 }
```

RC on src → new → Class → [modelBean.java] → implements → [model]

modelBean.java:-

```
public class modelBean implements model
{
  HbCustomerDAO dao;
  public void SetHbCustomerDAO(HbCustomerDAO dao)
    { this.dao = dao;
    }
public int CalcIntrAmt(String cname, int pamt, int time, int rate)
  {
```

ep① create webproject in myEclipse IDE having name SpringHBAPP3.

File menu → New → webproject → project name: SpringHBAPP3 → finish.

ep② :- Add HB capabilities to the project.

R.c on project → myEclipse → Add HB capabilities → hibernate 3.2.0 → next → next → DataSource ⊙ use jdbc drivers → DB Driver: Orap → next → ☐ Deselect checkbox → finish → Add → property: show-sql value: true → save.

ep③ :- Add Spring capabilities to the project.

Rc on project → myEclipse → add Spring capabilities → ~~finish~~

☑ Spring 2.5 core libraries ☑ Spring 2.5 persistance core libraries

☑ Hibernate 3.2 core libraries.

☐ Annotations } → next → ☐ deselect Enable AOP bldes →

☐ Advanced

file: SpringHB.xml → next → testdat → finish.

becomes bean id of localSessionFactory bean Configuration.

ep④ :- Add Apache DBCP s/w related jar file to classpath / buildpath of the project. (tomcat-dbcp.jar)

R.C on project → buildpath → add External Archives → Browse and select tomcat-home/lib → tomcat-dbcp.jar → OK.

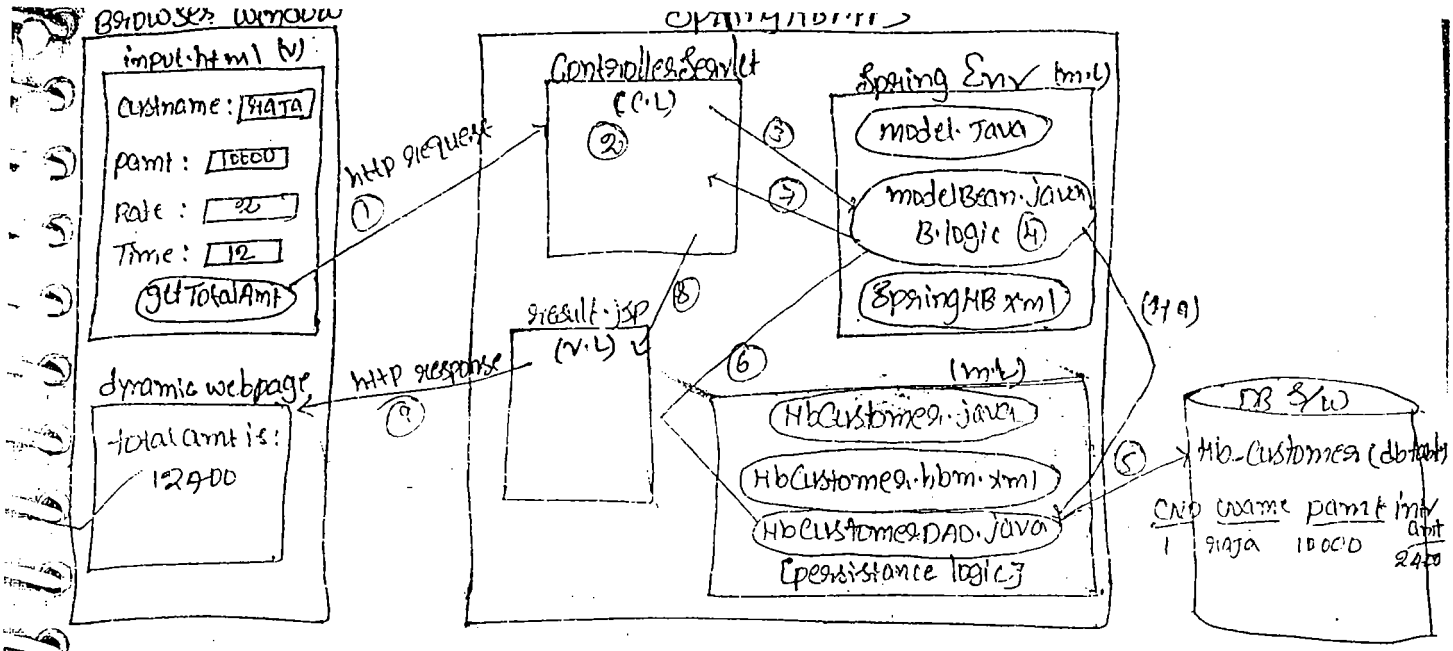
ep⑤ :- Configure the apache dbcp related BasicDataSource class in Spring HB.xml file.

In SpringHB.xml :-

```
<bean id="dbcp" class="org.apache.dbcp.dbc.BasicDataSource"
  <property name="driverclassname" value="oracle.jdbc.drivers.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~~ intrest amount and the persistance logic is inserting customer details.

07/11/11

Licence key of myEclipse is:-

Subscriber: Satya Technologies

Subscription code: FLR8Z6-855-55-6966505931894727

* procedure to develop the above diagram based application by using myEclipse IDE;-

Step①:- Create DB profile for oracle in myeclipse IDE:

Window menu → open persispetive → myeclipse DB Explorer → DB browse window → Right click → new → driver template: oracle thin drivers,

Driver name: orap → url: jdbc:oracle:thin@localhost:1521:orcl

username: scott password tiger → Add JARS → Browse & select ojdbc14.jar → testDriver → Save password → Next → finish.

Step②:- Create Hb-Customer table in oracle DB S/W having primary key constraint column.

SQL > create table Hb-Customer (CNO number(5) primary key, cname varchar2(15) pamt number(7,2), intramt number(7));

⑩ ←

Executing native SQL query directly using HBTemplate @ hibernateCallback (I) callback (C)

Step①:- In getData() method of demo2mpl.java.

```
public Iterator getData()
 {   List
 List l = ht.execute(new HibernateCallback()
  {
  public Object doInHibernate(Session ses) throws HibernateException,
                                                   SQLException;
   {
    // executing native SQL query directly
*  SQLQuery q1 = ses.createSQLQuery("select * from users");
          q1.addEntity(User.class);
          List l = q1.list();
           return l;
       } // method
 });
 Iterator it = l.iterator();
     return it;
} // getData()
```

* Here ht.execute(-) method is called having object of one annomous inner class. that annomous inner class is implementing ~~a class~~ callback interface called org.sf.orm.hibernate3.HibernateCallback(I)

* myEclipse IDE is capable of generating hibernate based Spring orm module style persistance 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 HB we need to perform transaction management while executing non-select SQL-queries.

* while working with HB Template class there is no need of performing Transaction management towards non-select queries execution.

In getData() :-

```
// to insert record.
User u1 = new User();
u1.setUid(154);
u1.setUname("Ravi");
u1.setRole("tl");
ht.save(u1);
```

```
// to delete the record
User u1 = new User();
u1.setUid(154);
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.uid>?",
                        new Object[] {"pl", new Integer(200)});
S.o.println("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 underlaying hibernate related hibernate Session object using that object we can develop our choice code.

11/11

Executing named native SQL select query having named parameters :-

P① :- In User.hbm.xml :-

```
<sql-query name="myq2">
    <return class="User"/>
    <![CDATA[ select * from users where userid >=:p1 and
                                        userid<=:p2 ]]>
</sql-query>
```

① :- In getData() method of demo2impl.java .

```
List l= ht.findByNamedQueryAndNamedparam ("myq2", new String[]{
    "p1", "p2"}, new object[]{new Integer(100), new Integer(200)});
```

Using pl/sql procedure of oracle using HibernateTemplate class obj :-

P① :- write pl/sql procedure in oracle as required for HB. (follow rules)

```
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 sqlprompt of oracle .

P② :- call the above pl/sql procedure from HB mapping file as named native sql query.

1 User.hbm.xml :-

```
<sql-query name="myq2" callable="true">
    <return class="User"/>
      { call getEmpDetails (?,?)}
</sql-query>
```

P③ :- write following code in getData() method of demoImpl class-

```
List l= ht.findByNamedQuery("myq2", new object[]{"b1"});
```

Q) Explain:

* HibernateTemplate class provides abstraction layer on plain hibernate and also converts the underlaying hibernate s/w generated checked Exceptions into un-checked Exception.

* executing HQL Select query with position parameters.

In getData() method of demoImpl class:-

```
public Iterator getData()
{
    List l = ht.find("from User u where u.uid >= ? and u.uid <= ?",
        new Object[] {new Integer(100), new Integer(200)});
    Iterator it = l.iterate();
    return it;
}
```

HQL query with positional parameters.

Java.lang.Object class obj[] supplying argument values.

executing HQL query with namedparameters:-

* List l = ht.findByNamedParam("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.iterate();
  return it;

* Executing named native SQL query with positional parameters (select):-

Step①:- prepare named nativeSql query in hibernate mapping file.

In User.hbm.xml:-

~~<sql-query class="User">~~

* mapping Sql query selects with HB Pojo class (User)

```
<sql-query name="myp2">
<return class="User"/> tablename. columnname.
<![CDATA[ Select * from users where userid >= ? and userid <= ? ]]>
</sql-query>
```

Step②:- write the following code in the getData() method of demoImpl class

```
List l = ht.findByNamedQuery("myp2", 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="demoImpl">
   <property name="ht" ref="template" />
</bean>
```

the bean id of HibernateTemplate class which is configured above.

Step 5: Write spring style HB persistance logic in our Spring Bean class by using the injected HibernateTemplate class obj (ht).

Ex:
```
public class DemoImpl implements demoInter
{ HibernateTemplate ht;
  public void setHt(HibernateTemplate ht)
  { this.ht = ht; }
  public void bm1()
  { ==== } use the methods of HT class & write HB persistance logic
}
```

Step 6: Develop the remaining resources of app in regular manner.

Understanding Various methods of the predefined HibernateTemplate (c) for persistance

save(-), persist(-) for insertion of records
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 of select
iterate(<->) → to execute hql select queries
findByXxx(--) → to execute native sql queries
findByCriteria(<->) → to execute logics of criteria api.
bulkUpdate() → to execute non select hql queries
the call back interface in HBTemplate class environment is "org.springframework.orm.hibernate3.HibernateCallback"

(*) 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 setFactory(SessionFactory factory)
    { this.factory = factory;
    }
    public void bm1()  // B.method
    { try {
        Session ses = factory.openSession();
        ==========  // use HB session obj to write HB persistance logic.
      }
      catch (Exception e)
      { ---- }
    } // bm1()
}
```

Steps: Develop the remaining resources of the app in regular manner as required for the app.

For above steps & diagram based Spring with HB app refer page #70 app# 21 (Approach-1)

Drawbacks:

* If B business component containg persistance logic as main logic of the app then the persistance logic itself acts as the business logic of the app. (refer getData() business method of page#71)

The limitation with approach #1 based Spring with HB app is we still need to use HB api in spring bean class to develop hibernate persistance logic.

Approach no #2

* Procedure to develop Spring with HB app based on approach 2 (by injecting HibernateTemplate class obj)

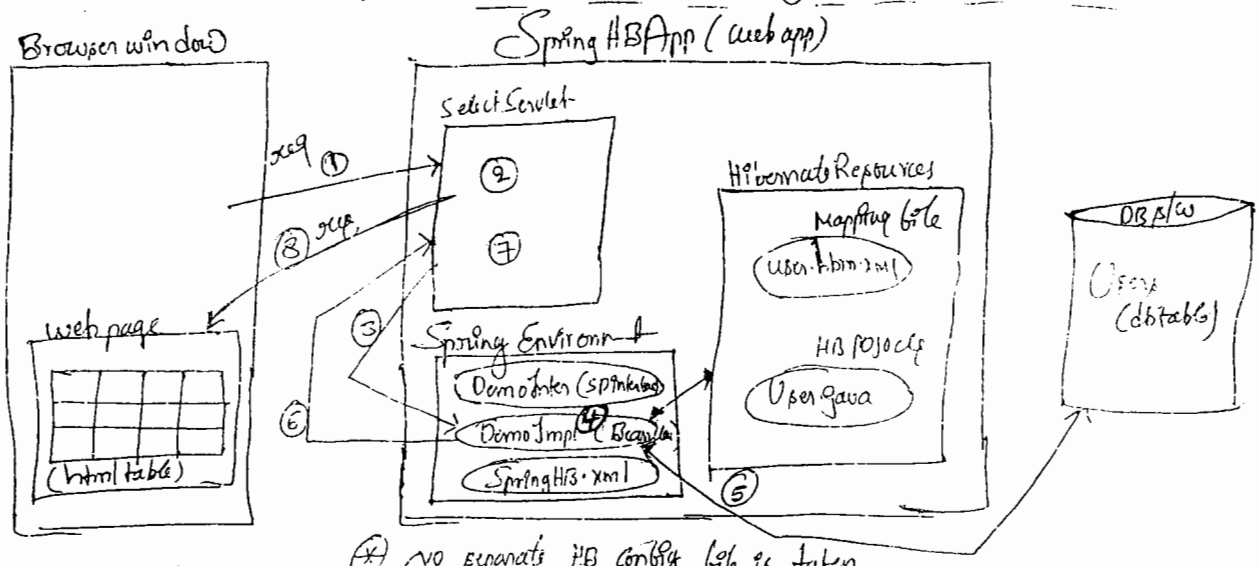
Step 1 & 2 are same as approach no 1

Step 3- config: Use the local SessionFactoryBean generated HB SessionFactory obj as base obj and configure org.sf.orm.hibernate3.HibernateTemplate class in spring config file.

* Sample code: In Spring config file:

```
==========
<bean id="template" class="org.sf.orm.hibernate3.HibernateTemplate">
   <property name="sessionFactory" ref="sesfact"/> </bean id of LocalSessionFactoryBean
</bean>                                            // to inject HB SessionFactory obj.
```

* If you are developing spring orm module app based on the existing resources of hibernate then go for example 1 style of Spring config file



W.r.to the diagram

① Browser window gives blank request to servlet prog.
② Servlet prog takes the request, activates the spring container, gets Spring bean class obj
③ Servlet prog calls B.method of Spring bean class.
④ The Spring Bean class uses either approach 1 or 2 to develop HB persistance logic based on HB resources.
⑤ This persistance logic of spring bean class B.method will interacts with db s/w & gets records from the table.
⑥ The B.method of Spring bean class sends the results to the Servlet prog.
7 & 8 The servlet prog formats the result by using presentation logic and sends the results to browser window as web page containing html table.

Procedure to develop the above diagram based app (Spring with HB) by using approach 1

Step1: Config bean that gives Jdb DataSource obj
Step2: Config org.sf.orm.hibernate3.LocalSessionFactoryBean that gives hb SessionFactory obj
Step3: Config our Spring Bean class (DemoImpl) injecting HB SessionFactory obj given by the above LocalSessionFactory Bean

(for sample code refer example 2 Spring config in the previous page once)

Step4: Create HB Session obj in the B.method of Spring bean based on the injected HB Session factory obj.

pad

```
<bean id="sessfact" class="org.sf.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="ds" /> <!--
    <property name="configLocation" value="classpath:hibernate.cfg.xml" />
</bean>                                          HB config file is specified here.
<bean id="db" class="DemoBean">
    <property name="factory" ref="sessfact" /> // our Bean class obj will be injected with
</bean>                                          the local SessionFactory bean generated
</beans>                                         HB SessionFactory obj.
```

Note: Here Spring config & HB config files are two different xml files.

~~Approach 5~~ - Example 2

```
<beans>
  <bean id="ds" class="org.apache.tomcat.dbcp.dbcp.BasicDataSource">
  ==
  </bean>
  <bean id="sessfact" class="org.sf.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="ds" />
    <property name="mappingResources">
      <list>
        <value>Employee.hbm.xml</value> // hb mapping file
      </list> </property>
      <property name="hibernateProperties">
        <props>
hb config properties. ---> { <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="sessfact" />
  </bean>
</beans>
```

* Here separate hb config file is not required bcoz the spring config file itself containing hibernate config details.
* HB persistance class (pojo), hb mapping file, hb config file are called as hb resources
* If Spring orm module app is developed from scratch level then go for example 2 style Spring config file

→ This pages will read from this page to back

These are the gaps of

03/11 → 07/11
09/11 → 13/11
03/12 → 07/12

Best sites for Java Code

RoseIndia.net
OnJava.net
JavaBeat.org
devx.com
preciseJava.com
JavaBasic.Internet.com

03/11/11

Spring with Hibernate App

* The two approaches of developping Spring with HB app

1. Approach 1
   [Inject HB SessionFactory obj to Spring Bean]
   * Here we need to use plain hb api in Spring Bean class to develop persistance logic

2. Approach II
   [Inject HibernateTemplate class obj in Spring Bean class]
   * Here there is no need of using plain HB api in Spring Bean class to develop persistance logic

⊛ Approach II is recommended to use

* To work with approach no I we use "org.sf.orm.hibernate3.LocalSessionFactoryBean" class which is capable of generating and injecting HB SessionFactory obj to the specified property of Spring bean class.
* While configuring this class DataSource obj is mandatory and hb config location property is specified then we can work with Separate hibernate configuration file. Otherwise the details of HB config file must be specified in Spring config file it self while configuring this class.

Example 1 (In Spring config file)

```
<beans>
   <bean id="dmds" class="org.apache.tomcat.dbcp.dbcp.BasicDataSource">
   </bean>
```

gives JDBC DataSource obj