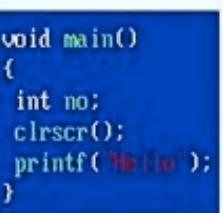





**College
Projects**


**Basic
Program**

**C
Tutorial**


**JAVA
Programming**



Tutorial4Us

Tutorials for Beginners



Tutorial4Us

Tutorials for Beginners



Tutorial4Us

Tutorials for Beginners

tutorial4us.com

A Perfect Place for All **Tutorials** Resources

Core Java | Servlet | JSP | JDBC | Struts | Hibernate | Spring

Java Projects | C | C++ | DS | Interview Questions | JavaScript

College Projects | eBooks | Interview Tips | Forums | Java Discussions

For More Tutorials Stuff Visit

www.tutorial4us.com

A Perfect Place for All **Tutorials** Resources

Spring

(Natraj Notes)

www.tutorial4us.com

→ Using web Frameworks we can just develops web applications.

Using ORM Frame works we can just develop persistence Logics.

Using Spring we can develop all kinds of logics like Business Logic, Persistence Logic, Integration Logic, presentation Logic and etc.

→ Using Spring we can develop all kinds of applications like standalone appns, two-tier applications, web applications, Distributed applications, enterprise appn & etc

Spring

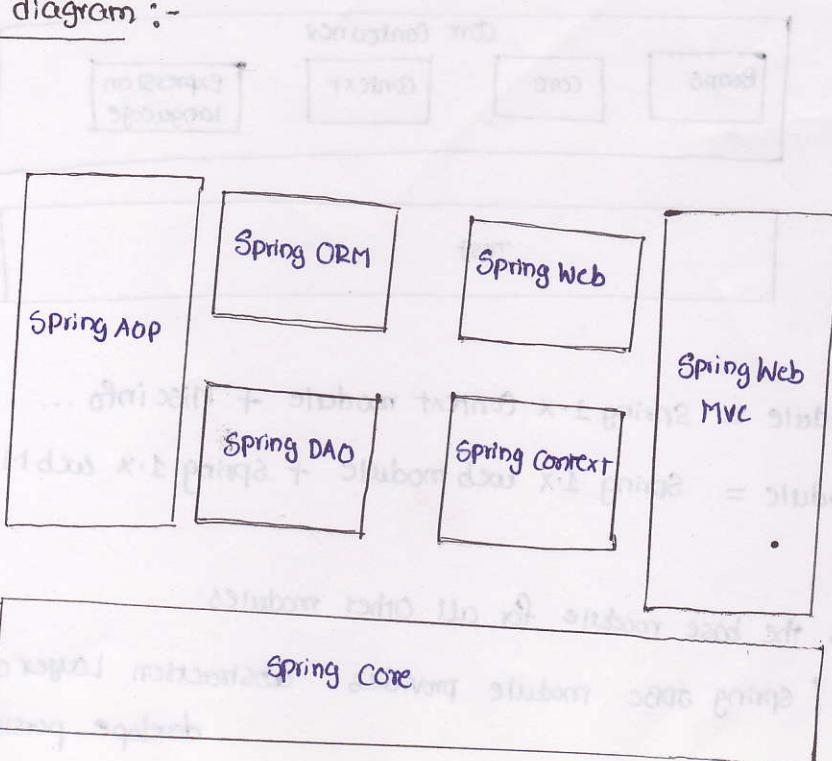
- Type: Java-jEE Framework / Application Framework
- Vendor: Interface 21
- Creator: Mr. Rod Johnson
- OpenSource s/w
- To Download s/w: Download as zip file from www.springframework.org
- Version: 2.5, 3.x 2.5 (compatible with jdk1.5+)
 3.x (compatible with Jdk1.5+)
- For docs and FAQs: www.springframework.org
- For online Tutorial: www.roseindia.net
- For good articles: coco.java4s.com & coco.mkyong.com
coco.nataraj.in

Books: Spring in Action From manning publisher

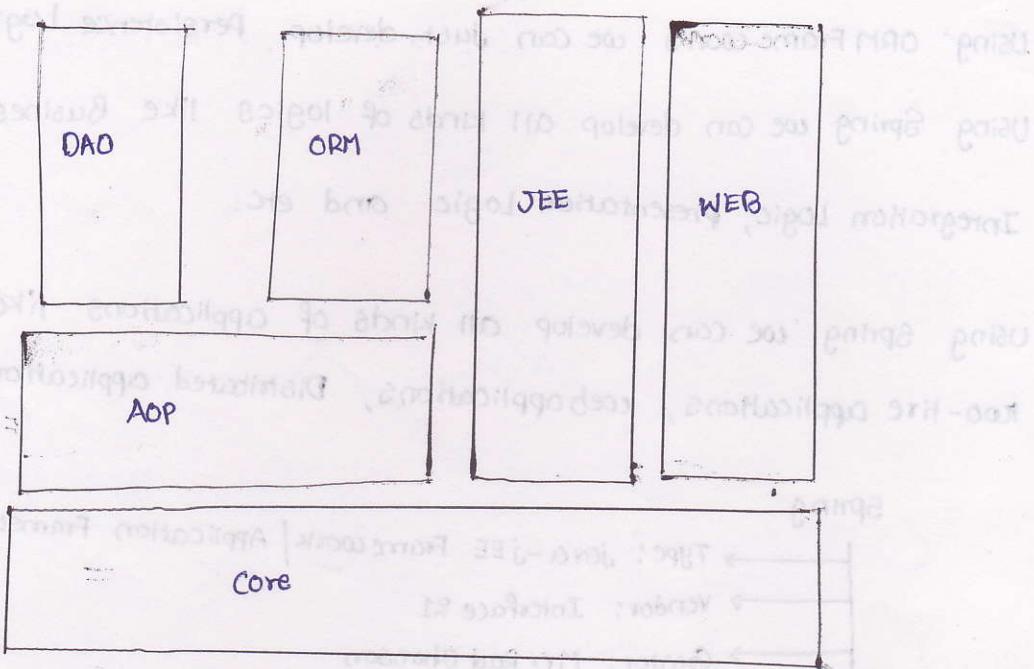
→ Spring 1.x contains 7 modules, where as spring 2.x contains 6 modules.

Spring 3.x contains 20 modules but they are grouped into 6 major categories

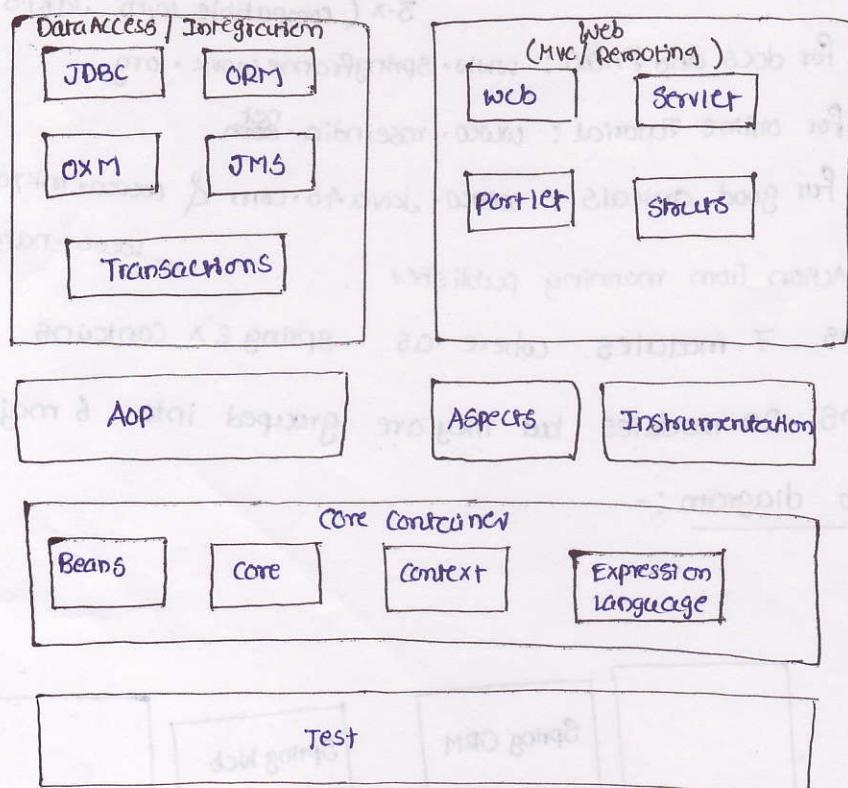
→ Spring 1.x overview diagram :-



Spring 2.x Over view Diagram :-



Spring 3.x Over view Diagram :-



Spring 2.x JEE module = Spring 1.x Context module + Misc info ...

Spring 2.x web module = Spring 1.x web module + Spring 1.x web MVC module

- Spring core is the base module for all other modules
- Spring DAO / Spring JDBC module provides abstraction layer on plain JDBC to develop persistence logic

- Spring ORM module provides abstraction layer on other ORM Flws and allows to develop O-R mapping persistence Logic. like Hibernate
- Spring AOP module is given to apply Aspects (Middleware services) on Spring Apps.
- Spring Web module is given to make spring applications communicable from Other web frame work s/w applications. like struts appns, JSF appns etc.
- Spring Web-MVC is given spring's own web framework develop MVC architecture based web application.
- Spring Context module / JEE module provides abstraction layers on multiple Core technologies like JNDI, EJB, RMI, JMS, JavaMail and etc.... develop enterprise appns, Distributed appns and other applications.
- Middleware services are additional, optional and configurable services / logics on our applications to make them more perfect and accurate.
Eg:- Security, Transaction mgmt, Logging

To install spring & its s/w

- `<Spring 2.5-home>\dist\spring.jar` represents the whole spring api and its dependent jar file is `<Spring 2.5-home>\lib\jakarta-commons\commons-logging.jar`
- The classes and interfaces of `spring.jar` uses the classes and interfaces of `commons-logging.jar`
- `<Spring 2.5-home>\changelog.rtf` use details about various versions of Spring s/w.

Spring definition :-

- Spring is a Java based open source, light-weight, loosely coupled, aspect oriented and dependency injection based application / Java-JEE framework to develop all kinds of applications.
- Open source software means It is not only free software but its source code will be exposed to programmers along with installation.

<Spring home>/src folder represents source code of Spring software.

- Spring is light-weight software and Spring Applications are light-weight apps. reasons,

① Spring containers are light-weight containers. Because they can be activated anywhere by just creating objects for certain pre-defined classes.
NOTE:- Servlet Container, JSP Container are heavy weight container.

- ② most of the Spring apps can be executed without the heavy weight application server and webserver software's.
- ③ resources of Spring application can be developed without using Spring API.
- ④ while executing Spring applications much memory and CPU resources are not required

- ⑤ Spring Technology is easy to learn and use
- If the Degree of dependency is less b/w two components then we can say the components are loosely coupled.

If the Degree of dependency is more b/w two components then the components are tightly-coupled components.

Spring is loosely coupled software. reasons,

- ① Spring software while working with Spring core can use either individual modules or multiple modules to gather.
- ② core can integrate Spring with other Java Technologies in application

development. like, servlets, JSP, struts, hibernate and etc...

③ Every spring module provides abstraction layer on providing multiple core technologies and makes programmer free from core technologies based programming while developing applications.

→ It is always recommended to develop large scale ^{spring} applications by enabling middleware services like security, Transaction mgmt and etc..
Spring provides a new methodology called Spring AOP to apply these middleware services on Spring applications.

This makes the Spring as Aspect oriented.

→ What is the difference b/w dependency lookup and dependency injection?

→ dependency lookup:- Resource explicitly ^{searches and gathers dependent values} dependency

from others. (like other resources)

→ in dependency lookup resource pulls the values from others

Eg(1):- if student gets course material only after passing request for it. It is called ^{then} dependency

Dependency lookup.

Eg(2):- The way we collect Data source object reference from JNDI registry is called dependency

Dependency lookup.

Advantage:- resource can gather only required dependent values.

Disadvantage:- resource should spend some time to gather its dependent values before utilizing them.

→ Dependency Injection:- The underlying server (or) Container (or) Flw (or) Runtime

(or) special resource assigns values to the resource automatically and dynamically.

→ in Dependency Injection (also called as IOC (inversion of control)) the underlying Container (or) Server (or) Runtime (or) Flw and etc.. pushes the values to the resource.

Eg(1): If course material is assigned to student the moment student registers for a course is called Dependency Injection.

Eg(2): The way ActionServlet writes formdata to FormBean class properties

Eg(3): The way JVM calls constructor automatically to assign ~~formdata~~ initial values to object.

Eg(4): The way Servlet Container calls lifecycle method service (-,-) to expose request, response objects to our servlet program.

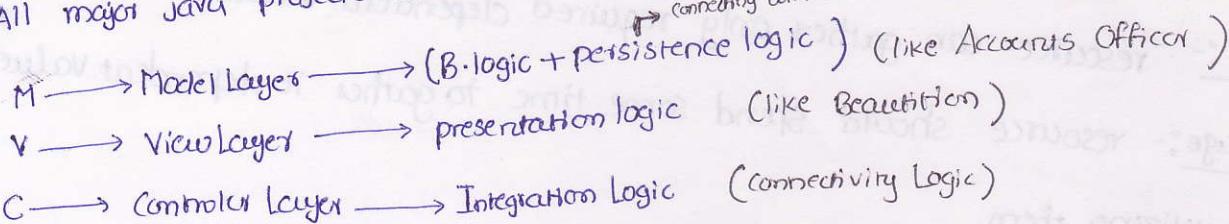
Advantages:-
resource can use the injected dependent values directly without spending time to get them.

Disadvantage:-
The underlying environment like Server / Framework may inject both necessary & unnecessary values.

necessary & unnecessary values.
→ Spring supports both dependency lookup & dependency injection. Spring Containers are designed to perform dependency injection on the resources of Spring application.

→ A file (or) program of an application is called resource. (generally it is class or interface)

All major Java projects are based on MVC2 / MVC Architecture



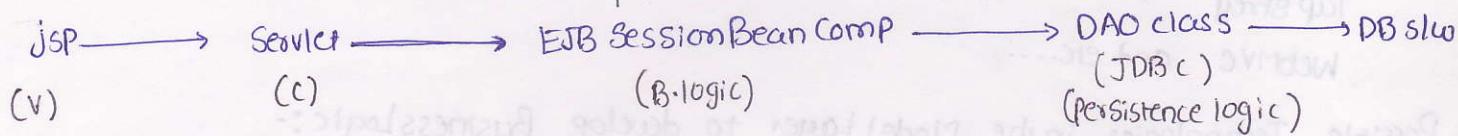
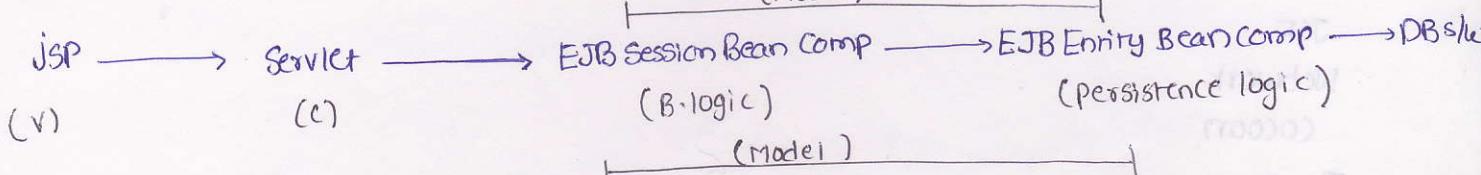
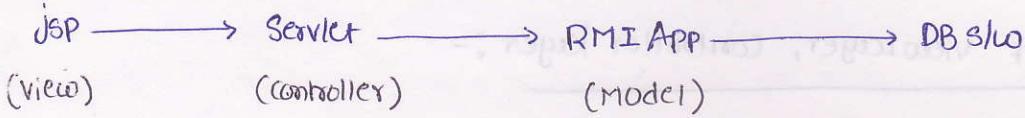
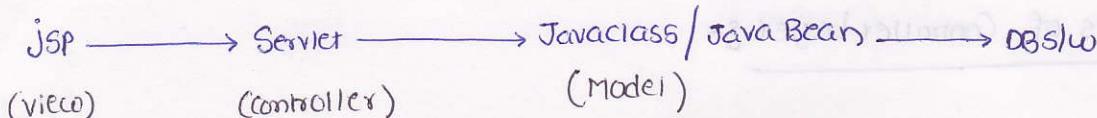
→ The main logic of the application is called Business Logic

→ Persistence logic is useful to perform CURD operations on Database like's

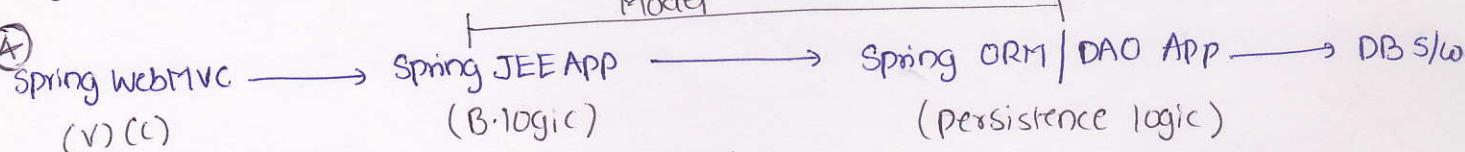
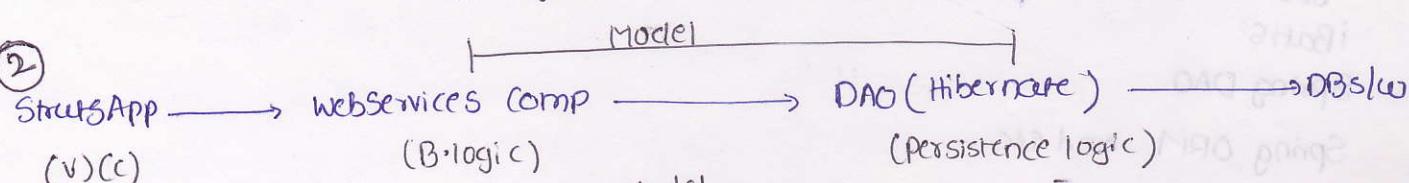
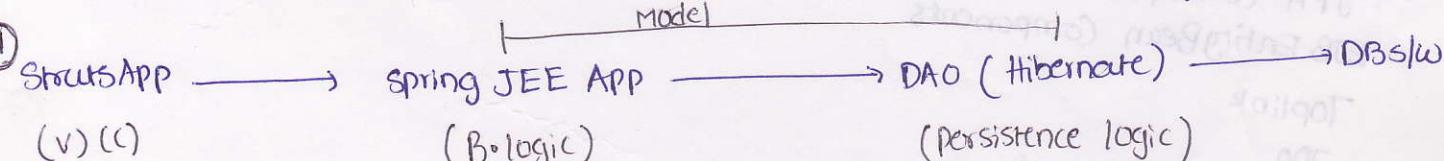
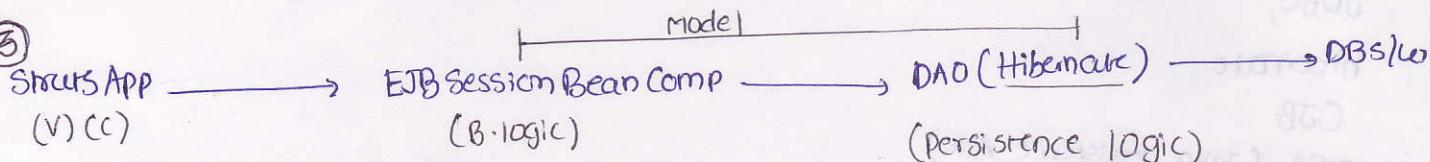
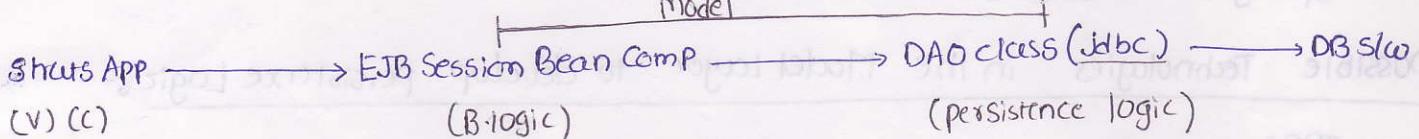
C → Create U → Update R → Read D → Delete
(Insert) (select) (remove)

→ presentation logic gives user interface for end user to supply inputs to application and to view results of the application

→ Integration logic controls and monitors all the operations of the application.



⁶⁶ The Java class that separates persistence Logic from other logics of the application to make the persistence Logic as the flexible logic to modify and reusable logics is called DAO (Data Access Object) "



The possible Technologies of view layer

jsp

velocity

free marker

html

XSLT (XML Stylesheet Language Transformation)

Jasper Reports / iReports

Possible Technologies of Controller Layer :-

Servlet
ServletFilter

Possible Frameworks of viewLayer, controller layer :-

Struts
JSF
Webwork
Cocoon
Tapestry
WebMVC and etc....

Possible Technologies in the Model Layer to develop Business Logic:-

EJBSession Bean, Webservices, javaclass | JavaBean class

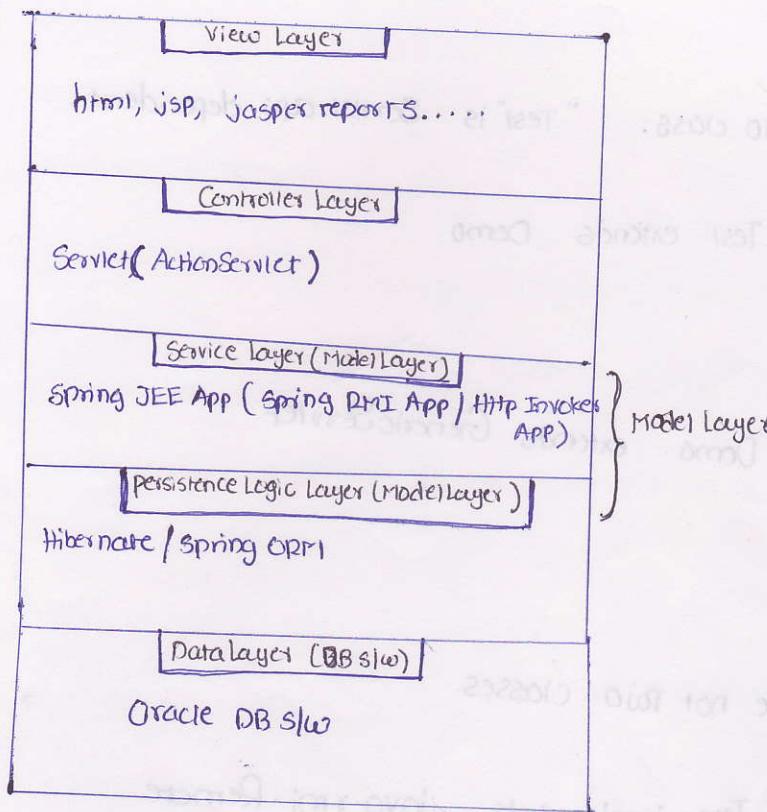
RMI, HttpInvoker (springs on Distributed Technology), CORBA,

Spring JEE and etc....

Possible Technologies in the Model Layer to develop persistence Logic:-

JDBC,
hibernate
OJB
JPA (Java Persistence API)
EJB EntityBean Components
Toplink
JDO
iBATIS
Spring DAO
Spring ORM and etc...

High Level project Architecture of Struts-Spring-Hibernate Integration



→ All latest technologies of Java and their latest versions (released after 2003) are supporting POJO/POJI model programming to make the technologies and applications as light-weight.

POJO class :- If a Java class that is acting as resource of certain technology based software application is not extending, implementing predefined classes, interfaces of that technology specific API then it is called POJO class.

Eg:- If Java class is taken as resource of Spring application without extending, implementing the predefined class, interfaces of Spring API then it is called POJO class.

Examples :-

(1) public class Test
{
 }
}
"Test" is POJO class.

(2) public class Test extends Demo
{
 }
}
"Demo" is not extending from any other class.
Test is POJO class.

Eg3:- public class Test extends HttpServlet

```
{  
    =  
}
```

"Test" is not POJO class. "Test" is Servlet API dependent.

(4) public class Test extends Demo

```
{  
    =  
}
```

public class Demo extends GenericServlet

```
{  
    =  
}
```

"Test", "Demo" are not POJO classes

(5) public class Test implements java.rmi.Remote

```
{  
    =  
}
```

"Test" is not POJO. bcz Remote is a Technology

(6) ~~public class Test implements java.sql.Driver~~

```
{  
    =  
}
```

"Java.sql.Driver" (I) is part of JDBC technology. so "Test" is not POJO class.

(7) public class Test implements java.lang.Runnable

```
{  
    =  
}
```

"java.lang.Runnable" is the core Java. so, Test is POJO

(8) public class Test extends Thread

```
{  
    =  
}
```

"Test" is POJO class bcz java.lang.Thread is part of basic Java APIs

④ public class Test
{
 public void m1()
 {
 ~~uses spring API~~
 }
}

Test is POJO class.

Spring supports POJO / POJI model programming due to this Spring apps are light weight apps

POJI :-

When Java interface is taken as the resource of certain technology based application

And if that interface is not extending from predefined interface of the technology

Specific API's then it is called POJI

Examples :-

① interface Test
{
 ~~// method declarations~~
 ~~-----~~
}
"Test" is POJI

② interface Test extends Demo
{
 ~~"Demo" is user-defined interface not extending other interfaces~~
 ~~-----~~
}
So, "Test" is POJI

③ interface Test extends java.rmi.Remote
{
 ~~"Test" is not a POJI. (because java.rmi.Remote is RMI API interface)~~
 ~~-----~~
}

④ interface Test extends java.io.Serializable
{
 ~~"Test is POJI" (because java.io.Serializable interface is very much part of basic Java API's).~~
}

Features of Spring:-

- ① Supports POJO / POJI model programming.
- ② Spring is light-weight software. (Most of the Spring applications can be executed without the heavy weight Webserver / Application Server)
- ③ Spring gives built-in middleware services and allows to work with server managed and third party supplied middleware services.
- ④ Spring gives new methodologies called AOP / Aspect / to apply middleware services...
- ⑤ Supports both Dependency injection and Dependency lookup.
- ⑥ provides abstraction Layer on multiple core Technologies and simplifies the application development.
- ⑦ provides environment to perform Unit Testing on Applications.

NOTE:- programmers testing on his own piece of code is called Unit Testing -

- ⑧ provides Built-in plugins to make Spring applications integratable with other Java technology based apps.
- ⑨ Allows to develop all kinds of applications like standalone, two-tier, three-tier, n-tier distributed applications, web applications and etc...
- ⑩ Allows to develop all kinds of logics like presentation logic, Business logic, Integration logic, persistence logic and etc...
- ⑪ Gives its own web framework Spring WebMVC to develop MVC2 architecture based web apps.
- ⑫ provides abstraction Layer on other frameworks like Hibernate, OJB and etc...
- ⑬ supports ~~EJB~~ Java / J2EE technologies and Java based Frameworks.
- ⑭ Easy to learn and use.

Module - I

Spring Core Module

- Base module for other modules of Spring.
- Gives a light-weight Spring Container called BeanFactory "BeanFactory".
- Using this module we can develop only stand alone applications.
- We can use this module to understand SpringBean lifecycle management and dependency injection.

The Concrete Java class that is configured in Spring configuration file and through annotations is called SpringBean. JavaBean can act as SpringBean but every SpringBean need not to be a JavaBean.

→ Spring gives two built-in containers and lightweight containers

① BeanFactory Container (part of core module)

② Application Context Container (part of spring JEE/Context module)

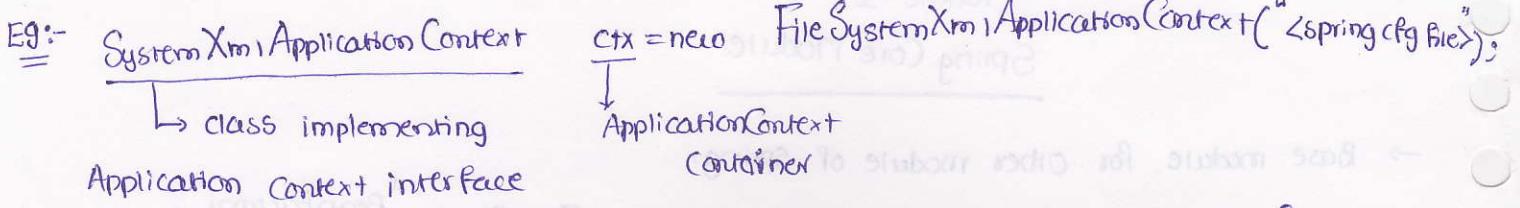
→ Container is a Software Appn (or) Java class that can take care of the whole life cycle of given resource.
↳ like aquarium
↳ like fish

→ Spring Containers are given perform SpringBean Lifecycle management and to perform Dependency Injection on SpringBean.

→ To activate BeanFactory container in our applications create object for java class that implements org.springframework.beans.factory.BeanFactory(I)

XmIBeanFactory → class implementing BeanFactory
factory = new XmIBeanFactory ("<obj pointing to spring config file>");
represents BeanFactory Container

→ To activate ApplicationContext container in our Apps Create obj for java class that implements org.springframework.context.ApplicationContext(I)



→ we can't activate Server Container, JspContainer by creating objects for certain classes. So, they are heavy weight containers. Bcz we need to start heavy weight servers to activate these containers.

But Spring Containers are light-weight Containers

→ ApplicationContext container is enhancement of BeanFactory container.

Spring CoreModule Application contains the following resources.

- 1) Spring Interface (JavaInterface)
- 2) Spring Bean class (java class)
- 3) Spring Cfg file (xml file)
- 4) Client Application (java class with main method)

Spring Interface

- it can be a POJO
- it is optional resource. But recommended to take
- contains declaration of B-methods/utility methods
- acts as a common understanding document between client application & SpringBeans
- it like course brochure having declaration of topics [which acts as agreement document b/w faculty and student].

Spring Bean

- A concrete Java class. configured in Spring Configuration file
- can be a JavaBean
- implements Spring Interface and provides implementation to Spring interface methods.
- Contains methods, constructors supporting Dependency Injections.

- Contains utility methods
- Contains lifecycle methods.
- It can be POJO class (or) non-POJO class.
- It can be predefined / user defined / Third party supplied concrete Java class / Java Bean.
- Contains Business methods implementation having Business Logic.

Note:- All Spring Beans must be configured in Spring Configuration file either by XML entries (or) using `@Bean` annotation

→ Specifying the details of resource through XML entries (or) Annotations to make underlying container (or) server (or) Frameworks recognizing that resource is called resource configuration

14/02/2013

Spring Configuration File :-

- any <filename>.xml can be taken as spring cfg file.
- We can one or more spring cfg files in our spring apps
- Spring beans must be cfg in spring cfg file.
- Contains Dependency Injection / IOC related instructions
- Contains life cycle methods configuration
- Can be developed either based on DTD / schema rules.

Client Application :-

- Activate Spring Container by specifying spring cfg file
- Get Spring Bean class Obj from Spring Container
- call Business methods on Spring Bean class object.

→ Spring Supports 3 types of Dependency Injection

1) Setter Injection

(by using setXXX(-) of Spring Bean class)

2) Constructor Injection

(by using parameterized constructors of Spring Bean class)

3) Interface Injection

(by implementing special interfaces on SpringBean class)

→ ① Setter Injection:-

If Spring container calls setXXX(-) methods automatically and dynamically

to assign values to the properties (member variables) of SpringBean class. Then

it is called Setter injection.

Example Scenario

// TestBean.java (Spring Bean)

public class TestBean

{

// Bean Property

String msg;

// Write setXXX (-) supporting Setter Injection

public void setMsg (String msg)

{

this.msg = msg;

}

// Write B-methods

public void bmt()

{

....

.... // use msg here

}

}

Demo.xml (Spring cfg file)

<beans >

① msg property configuration for Setter Injection

```

<bean id="tb" class="TestBean">
    <!--> id/objname          <--> Bean class
        <property name="msg" value="hello"/>
            <!--> Bean property           <--> Value to be injected
    </bean>
</beans>

```

} spring Bean configuration

- When Spring Container is activated by giving this Spring cfg file the Spring Container creates TestBean class object having "tb" as the Object name and calls tb.setMsg("hello") to inject the value "hello" to the bean property "msg" through setterInjection.

- Spring Container manages the whole lifecycle of SpringBean in that process it performs dependency injection on SpringBean class properties the moment it creates SpringBean class object.
- In real time projects we make Spring Container injecting special objects to SpringBean like JdbcConnection obj, Jdbc Stmt obj, Rmi Business obj and etc...

(2) Constructor Injection :-

- If Spring Container uses parameterized constructors to create SpringBean class objs and to assign values to Bean properties then it is called constructor injection.
- Setter injection takes place immediately after SpringBean class object creation where as constructor injection takes place along with SpringBean class object creation.

Example Scenario :-

```
// TestBean.java (Spring Bean)
```

```
public class TestBean
```

```
{
```

```
    // Bean Property
```

```
    String msg;
```

```
// Parameterized Constructor Supporting Constructor injection
```

```
Public TestBean (String msg)
```

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

//Write B-methods

```
Public void bml()
```

```
{  
    ...  
    use msg here  
    ...  
}
```

```
} //class
```

Demo.xml (spring cfg file)

```
<beans>
```

```
<bean id="tb" class="TestBean">  
    <constructor-arg value="hai"/>
```

```
</bean>
```

```
</beans>
```

↳ since `<constructor-arg>` tag is placed only for one time the

Spring Container uses one param constructor to create

Spring Bean class object and to assign value "hai" to "msg" property.

When Spring Container is activated by giving the above Demo.xml spring cfg file

the Spring Container creates TestBean class ^{Object} having name "tb" by using

1-param Constructor (`TestBean tb=new TestBean("hai");`)

NOTE:- assigning values directly to Bean properties like `name="hello"` gives tight coupling between Bean properties and its dependent values. Assigning

values to Bean properties by specifying the values in Spring Configuration File

and by enabling dependency injection shows loose coupling b/w Bean properties

and its dependent values.

Naming Conventions

X → Spring interface name

XBean → Spring Bean class name

XCfg.xml → Spring cfg file name

XClient → Client App name

Eg:- Test → Spring interface name

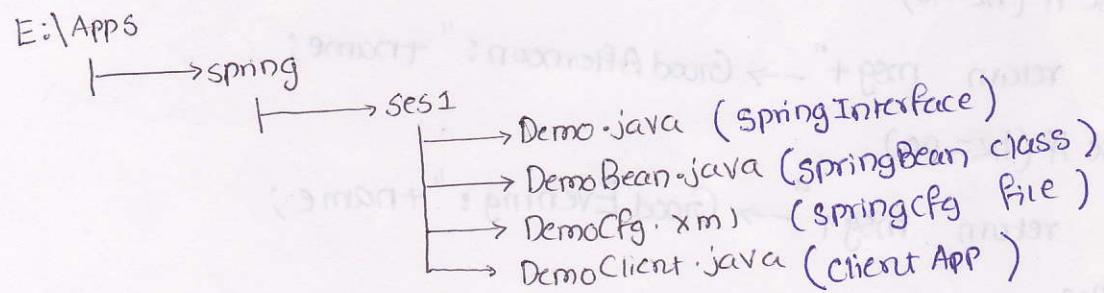
TestBean → SpringBean class name

TestCfg.xml → Spring cfg file name

TestClient → Client App name

1st Example Application of Spring Core module to demonstrate Setter Injection by using BeanFactory Container.

Step-I:- Develop the source code as shown below



{
 // B.method Declaration
 public String generateWishMsg(String name);
}

// DemoBean.java (POJO class) (Spring Bean)

import java.util.*;

public class DemoBean implements Demo

{
 // Bean Property to hold injected value
 private String msg;

// Setter method supporting setter injection

```
public void setMsg(String msg) {  
    this.msg = msg;  
}  
  
// implement B的方法  
public String generateWishMsg(String name)  
{  
    // get current date and time  
    Calendar cl = Calendar.getInstance();  
    // get current hours of the day  
    int h = cl.get(Calendar.HOUR_OF_DAY);  
    // generate wish msg  
    if (h <= 12)  
        return msg + "Good morning : " + name;  
    else if (h <= 16)  
        return msg + "Good Afternoon : " + name;  
    else if (h <= 20)  
        return msg + "Good Evening : " + name;  
    else  
        return msg + "Good Night : " + name;  
}  
// method  
// class
```

```
<!-- DemoCfg.xml -->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEANS//EN" "http://www.springframework.org/dtd/spring-beans.dtd"> → indicates that we are developing this XML document based
<beans> → collect this content from <spring-home>\dist\resources\

<bean id="db" class="DemoBean"> → ① Spring-beans-2.0-DTD file.
    <property name="msg" value="hello"/> → ②

</bean> → DemoBean db=new DemoBean();
          db.setMsg("hello");

</beans>
```

```

// DemoClient.java

import org.springframework.beans.factory.BeanFactory; // (I)
import org.springframework.beans.factory.xml.XmlBeanFactory; // (C)
import org.springframework.core.io.FileSystemResource; // (C)

public class DemoClient
{
    public static void main(String[] args) → (a)

    {
        // Locate Spring Configuration file
        FileSystemResource res = new FileSystemResource("DemoCfg.xml"); } (c)

        // Activate BeanFactory container
        XmiBeanFactory factory = new XmiBeanFactory(res);

        // Get SpringBean class obj from SpringContainer
        Object obj = factory.getBean("db"); → (d)

        // type casting
        Demo bobj = (Demo) obj; → (e)

        // call B.method
        String result = bobj.generateWishMsg("raja"); } (f)

        System.out.println("Result is " + result); } (g)
    }
}

```

Step-II :- Add the following 2 jar files to CLASSPATH

<spring_home>\dist\spring.jar (Mainjar file)

<spring_home>\lib\jakarta-commons\commons-logging.jar (Dependent jar
file to main jar file)

↳ spring.jar

Step-III :- Compile all the javaresources and execute the client application

> javac *.java

> java DemoClient

(a)

→ When BeanFactory container is activated the entries of Spring Configuration

file will be verified by using SAX XML Parser
↳ Simple API for XML

Object obj = factory.getBean("db");

- ① BeanFactory container to load "DemoBean" class based on the given Bean id "db" using configurations done in Spring Cfg file
- ② makes BeanFactory container to create "DemoBean" class object using 0-param constructor.
- ③ Since "msg" property is configured using <property> tag the BeanFactory Container calls setMsg("hello") method on DemoBean class Object perform setterInjection
- ④ factory.getBean("db"); method returns "DemoBean" class object and we are referring that object using java.lang.Object class ref "obj"

NOTE:- getBean() method is the predefined method in predefined class Object. if we want to get the object (Eg: "db") from BeanFactory container we use this getBean() method. If we call B-method generateWishMsg() it is not part of the predefined class "Object". it is part of the implement class of SpringInterface. so we must typecast the ^{Object class to Demo interface} Demo bobj = (Demo) obj;

→ When Superclass reference variable refers its subclass Object then we can't call direct methods of subclass using that reference Variable. To make this happening typecast that reference Variable either with subclass or

with interface implemented by subclass having that Direct method declaration
Demo obj = (Demo) obj; is placed to satisfy above statement in our
Spring appns.

→ Spring Container performs Dependency Injection only on those Spring Bean objects
that are created by the Spring container itself.
That means Spring Container can't perform Dependency Injection on Spring Bean
class objects that are created by programmer manually.

→ Keeping DTD related or schema related header statements in SpringCfg file
is mandatory. But this work is optional in web.xml.

→ Spring application class start with main ends with main. control goes to
one resource to another resource.

In the client appn we donot take Spring Bean class manually as Typecast. The interface of
implementation class Demo will be taken.

→ When BeanFactory container is activated ~~it does~~ it doesn't creates any
Spring Bean class objects. It creates Spring Bean class objects and
Completes dependency injection only when getBean(-) method is called.

Example application on Constructor injection :-

Demo.java → same as previous appn.

DemoBean.java → same as previous appn. But add the following 1-param constructor

```
public DemoBean (String msg)
{
    this.msg = msg;
    System.out.println ("DemoBean 1-param constructor");
}
```

DemoCfg.xml

```
<DOCTYPE ..... >
<beans>
    <bean id="db" class="DemoBean">
        <constructor-arg value="hello" />
    </bean>
</beans>
```

DemoClient.java → same as previous appn.

→ While configuring SpringBean to place only `<property>` tags under `<bean>` tag

then SpringContainer uses 0-param constructor for springBean class instantiation
and also uses `setterXxx(-)` methods for `setterInjection` on bean properties

→ While configuring SpringBean if you place only `<constructor-arg>` tags under
`<bean>` tag then SpringContainer uses parameterized constructor to
Create SpringBean class object & to perform Constructor Injection on
Bean Properties.

Q:- What happens if both `setterInjection`, `ConstructorInjections` are enabled on
same Bean properties of springBean class?

Ans: Spring Container creates SpringBean class object by using parameterized
Constructor. but the values given by `setterInjection` become final values of
Bean property. Because `setter` method executes after `Constructor`.
So, Values injected by `ConstructorInjection` will be overridden because of
`setter` injection.

→ To configure dependent values to Simple bean properties (like String, simple data types)

use <value> or value attribute in Spring cfg file.

→ To configure one Spring Bean class obj as dependent value Other Spring Bean class properties use <ref> or ref attribute.

→ if one spring Bean class is holding another SpringBean class obj then it called BeanCollaboration.

Example Application on <ref> tag / ref attribute (Bean Collaboration)

E:\APPS



SES2

Demo.java

DemoBean.java

TestBean.java

DemoCfg.xml

DemoClient.java

//Demo.java (POJO)

```
Public interface Demo
{
    public String sayHello(); //BusinessMethod
}
```

//DemoBean.java (POJO class)

```
import java.util.*;
public class DemoBean implements Demo
{
```

//bean properties

String name;

int age; //simple property

← TestBean tb; } reference type properties to hold objects

← Date d;

//Write setXxx(-) methods supporting Setter Injection

```
public void setAge(int age)
{
```

this.age = age;

S.O.P("DemoBean: setAge(-) method");

}

```

public void setTb(TestBean tb)
{
    this.tb = tb;
    S.O.P("DemoBean: setTb(-) method");
}

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

// implement BusinessMethod

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

```

NOTE:- When Object is passed in S.O.P() statement `toString()` will be called on that object automatically.

In most of the classes `toString()` will be overridden to display object data. `toString()` originally belongs to `java.lang.Object(c)`

// TestBean.java (Dependent Bean to DemoBean)

public class TestBean

{ // Bean property

String msg;

// Write setter method of Setter Injection

public TestBean {~~str~~}

public void setMsg(String msg)

{

this.msg = msg;

S.O.P("TestBean: setMsg(-) method");

}

public String toString()

{

return "TestBean.msg=" + msg;

}

}

NOTE:- Here DemoBean, TestBean classes are Taken as Collaborator beans because the "tb" property of DemoBean class can hold TestBean class object.

```
<!-- DemoCfg.xml -->
<beans>
    <bean id="dt" class="java.util.Date">
        <property name="year" value="113"/> Takes it as 1900 + 113
        <property name="month" value="10"/> Takes it as 0 to 11
        <property name="date" value="29"/>
    </bean>
    <!-- Boy and girl f/girl nominees -->
    <bean id="ti" class="TestBean">
        <property name="msg" value="hello"/>
    </bean>
    <bean id="db" class="DemoBean">
        <property name="age" value="30"/>
        <property name="tb" ref="ti"/>
        <property name="d" ref="dt"/> <!-- or <property name="d" />
        <!-- ref bean="dt" />
    </bean>
</beans>
```

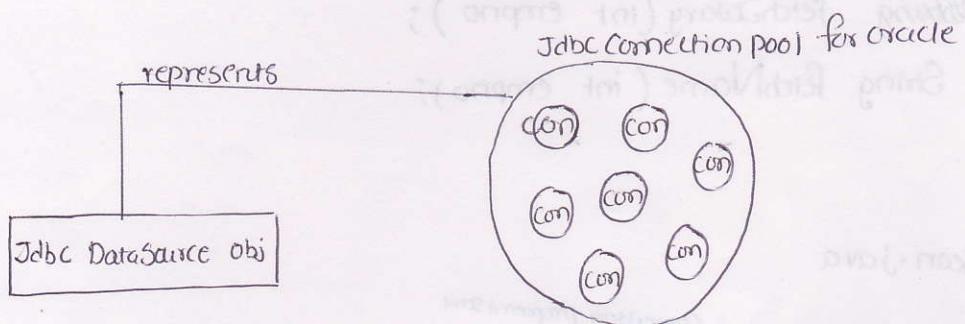
// DemoClient.java

Same as 1st application but call sayHello() B.method instead of generateWishMsg() B.method.

```
String result = bobj.sayHello();
S.O.P("Result is: "+result);
}
```

Jdbc Connection pool :-

- Jdbc Connectionpool is factory that contains set of readyly available Jdbc Connection Objects. All Connection objects in the Connection pool represents Connectivity with Same DataBase sw.
- DataSource object represents Jdbc Connection pool and this can be used to access Jdbc Connection Objects from Connection pool.



Jdbc DataSource obj is nothing but it is the object of underlying Jdbc driver

Supplied java class that implements java.sql.DataSource(I)

→ org.springframework.jdbc.datasource.DriverManagerDataSource class gives Jdbc DataSource object pointing to jdbc connection pool. This class creates jdbc connection pool based on the jdbc driver details supplied to bean properties.

The important bean properties are

- 1) driverClassName
- 2) url
- 3) username
- 4) password

If these properties are filled up with oracle thin driver details then jdbc connection pool for oracle will be created.

Eg:- Example application to inject jdbc DataSource object that points to jdbc connection pool to our SpringBean class using Dependency Injection.

E:\ APPS

→ Spring

→ ses3

→ DBOperation.java (Spring Interface)

DBOperationBean.java (Spring Bean)

SpringCfg.xml (Spring Configuration File)

DBOperationClient.java (Client Appn)

//DBOperation.java (POJO)

public interface DBOperation

{

 public int fetchSalary(int empno);

 public String fetchName(int empno);

}

//DBOperationBean.java

import java.sql.*; → Connection preparedstmt

import javax.sql.*; → JDBC Datasource obj

public class DBOperationBean implements DBOperation

{

 //Bean property

 DataSource ds;

 //Write setXxx(-) method to support setter injection

 public void setDs(DataSource ds)

 {

 this.ds = ds;

 }

} (h)

 //implement B methods

 public String fetchName(int empno)

 {

 try

 {

 //get Connection obj from connection pool

 Connection con = ds.getConnection();

 // get emp name

 PreparedStatement ps = con.prepareStatement ("Select ename from emp where empno=?");

```
ps.setInt(1, empno);
ResultSet res = ps.executeQuery();
String name = null;
if(rs.next())
{
    name = rs.getString(1);
}
else
{
    name = "record not found";
}
return name;
}

Catch (Exception e)
{
    return "DB problems";
}

} // method
```

```
public int fetchSalary(int empno)
{
    int sal = 0;
    try
    {
        // get one connection obj from jdbc com pool
        Connection con = ds.getConnection();
        // get emp salary
        PreparedStatement ps = con.prepareStatement ("Select sal from emp where empno = ? ");
        ps.setInt(1, empno);
        // execute the query
        ResultSet rs = ps.executeQuery();
        // process the results
        if(rs.next())
            sal = rs.getInt(1);
    }
    catch (Exception e)
    {
        return "DB problems";
    }
}
```

```
    catch (Exception e)
```

```
    {
```

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

```
}
```

```
    return scl;
```

```
} //method
```

```
} //class
```

→ <!-- Spring Cfg.xml -->

```
<!DOCTYPE ----- >
```

```
<beans>
```

```
    <bean id="drds" class="org.sf.jdbc.datasource.DriverManagerDataSource">
```

gives
datasource
obj

① *L gives Datasource obj representing
the spring container managed
built-in connection pool*

```
        <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="dob" class="DBOperationBean"> ②
```

③ *Injects Datasource obj to our
BeanProperty ds.*

```
</bean>
```

```
</beans>
```

→ //DBOperation Client.java

```
import org.sf.beans.factory.*;
```

```
import org.sf.core.io.*; → Filesystem Resource class
```

```
public class DBOperationClient
```

```
{
```

④ *throws Exceptions* ⑤

```
{
```

⑥ *//Locate Spring Configuration file*

FileSystemResource res = new FileSystemResource("springCfg.xml") } ⑦

⑧ *//Activate BeanFactory Container*

```
XmlBeanFactory factory = new XmlBeanFactory(res);
```

// get SpringBean obj from container

```
DBOperation bobj = (DBOperation) factory.getBean("db"); (d)
```

// Call B.methods

```
System.out.println("Emp salary is " + bobj.fetchSalary(7499)); (j)
```

```
S.O.P ("Emp name is " + bobj.fetchName(7499)); (l)
```

3 //main

3 // class

//> javac *.java

//> java DBOperationClient (a)

Jar files in CLASSPATH:-

(1) Spring.jar (2) Commons-logging.jar (3) OJdbc14.jar

NOTE:- in the above application Spring Container is injecting DataSource object to our Spring Bean class property "ds".

*) <Property name="ds" ref="drds" /> can also be written as

```
<property name="ds"> <ref bean="drds" /> </property>
```

*) <Property name="password" value="tiger" /> can also be written as

```
<property name="password"> <value> tiger </value> </property>
```

→ we can locate Spring Configuration file for BeanFactory container by using

the following classes.

1) FileSystemResource Class (org.springframework.core.io package)

→ locates the Spring Configuration file from the specified path of Filesystem.

→ Here we can specify Absolute path (or) relative path of Spring cfg File

Note:- all files folder of our Computer together is called Filesystem.

2) ClassPathResource Class (org.springframework.core.io package)

→ Locates the Spring cfg file from the directory locations, jar files added to CLASSPATH environment Variable.

Example on FileSystemResource to locate Spring Cfg file :- When springcfg file is there in E:\Apps\temp folder and when clientapp is there in E:\APPS\spring\ses3 folder.

In Client APP:-

FileSystemResource

```
res = new FileSystemResource ("E:/apps/temp/springcfg.xml");  
(or)
```

FileSystemResource

```
res = new FileSystemResource ("E:\\apps\\temp\\springcfg.xml");  
(or)  
↳ absolute path
```

FileSystemResource

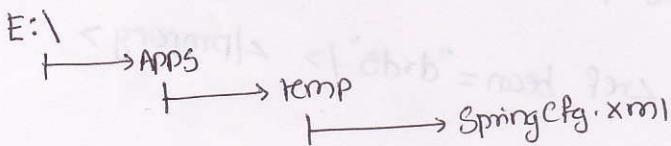
```
res = new FileSystemResource ("..||..||temp||SpringCfg.xml")  
↓ ↓ ↓  
Spring Apps relative path  
Folder Folder (Parent folder)
```

Example on Utilizing CLASSPATH resource :-

Step-I:- keep spring application ready

Step-II:- get the location on Spring Configuration file and add the location to the

CLASSPATH env... variable,



my Computer → properties → advanced tab → env variables → user / system

Variables → Variable name: CLASSPATH

→ OK → OK → OK

Value: E:\Apps\Temp ; <existingvalues>;

Step-III:- use CLASSPATH resource in client application

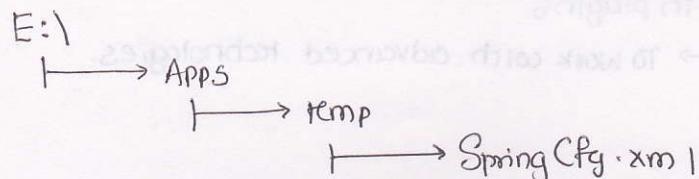
in clientApp

ClassPathResource res = new ClassPathResource ("SpringCfg.xml");

Example on CLASSPATH resource :-

Step-I:- keep spring application ready

Step-II:- prepare jar file representing Spring Configuration file and add that jar file to CLASSPATH.



E:\ Apps\ temp > jar cf test.jar .

my computer → properties → advanced tab → env variable → user/system

Variables → Variable name : CLASSPATH

Value : E:\ APPS\ temp\ test.jar ; <existing values> ; → OK → OK

Step-III :- use ClassPathResource in client appn

in clientappn

```
ClassPathResource res=new ClassPathResource ("SpringCfg.xml");
```

While working with Spring Environment always make sure that Current directory

is added to Classpath Env... Variable.

To cfg Dependent values to diff types of bean properties we need to use diff tags
and attributes

property type

Simple data type

java.lang.String

Array

java.util.List

java.util.Set ^{like} (HashSet, TreeSet ... etc)

java.util.Map

Bean type

java.util.Properties

tag | attribute to use

<value> or "value" attribute

<value> or "Value" attribute

<list>

<list>

<set>

<map>

<ref> or "ref" attribute

<props>

note:- To assign null value to Bean property use <null/> tag

MyEclipse = Eclipse + Built-in plugins

→ To work with advanced technologies.

MyEclipse

type : IDE SW for java

Vendor : Eclipse org

Commercial IDE

gives Tomcat as Built-in server and also to cfg other external servers.

Version : 10.x (compatible with jdk 1.6+)

To download SW : www.myeclipseide.com (30 days trial)

Procedure to keep MyEclipse IDE ready in our computer :-

Step I :- install MyEclipse 10.x. (from www.myeclipseide.com)

Step II :- Get the Crack related to MyEclipse 10.x cheat code (in the form of zip file) [www.naturaz.in blog](http://www.naturaz.in/blog)

Step III :- Run .Batch file. which comes bcz of Crack extraction

Step IV :- follow the instructions given by application launched by run batch file

and get licence name & licence key

Step V :- Submit the above license details ~~to~~ through MyEclipse IDE

MyEclipse menu → Subscription Information → Subscriber :
subscriptioncode :
→ Ok.

procedure to develop Spring Core module application in MyEclipse 10.x IDE that performs Setter injection on all types of Bean properties.

Step - I :- launch MyEclipse IDE specifying WorkSpace Folder

→ The folder where the projects will be saved.

Step - II :- Create Java project in MyEclipse IDE.

File → new → JavaProject → project name : SpringProj → next → finish

Step - III :- Add Spring Capabilities to the project.

Right click on project → MyEclipse → Add Spring Capabilities → Spring 2.5 →

Spring 2.5 core libraries → next → App Builder → File : DemoCfg.xml → finish

Step-IV:- Add SpringInterface, SpringBean class to the src folder of the project

right click on src folder → New → Interface → Filename: Demo Demo

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

Step-V:- Add SpringBean class

right click on src folder → New → Class → Filename: DemoBean

```
public class DemoBean implements Demo
```

```
{ // Bean Properties  
    String name;
```

```
    int age;
```

```
    Date d;
```

```
    List fruits;
```

```
    String colors[];
```

```
    Set phones;
```

```
    Map faculties;
```

```
    Properties capitals;
```

Select all Bean Properties → right click → Source → generate setters and

getters → Setters → insertion point After capitals

after setters methods

@Override

```
public String sayHello()
```

```
{  
    return "good morning" + "name=" + name + "age=" + age + "d=" + d +  
           "fruits=" + fruits + "colors=" + colors[0] + "..." + colors[1] + "..."  
           + colors[2] + "phones=" + phones + "faculties=" + faculties + "capitals=" + capitals;  
}
```

} // method

} // class

Step VI:- add following content in spring configuration file.

<beans> ----- spring-bean-2.5.xsd </beans>

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

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

<Property name="name" value="rajai"/>

<Property name="age" value="30"/>

<Property name="d" ref="dt"/>

<Property name="fruits">

<list>

<value> apple </value>

<value> banana </value>

<ref bean="dt"/>

</list>

</property>

<property name="colors">

<list>

<value> red </value>

<value> yellow </value>

<value> blue </value>

</property>

<property name="phones">

<set>

<value> 9247285621 </value>

<value> 7702349715 </value>

<ref bean="dt"/>

</set>

</property>

<property name="faculties">

<map>

<entry>

{ <key> <value> ramesh </value> </key> } → represents key of the element

<value> Java Faculty </value> } → represents value of the element

represents
one
element

<entries>

<key> <value> ramana </value> </key>

<value> .NET Faculty </value>

<entry>

<entry>

<key> <value> today </value> </key>

<ref bean="dt"/>

<entry>

</map>

</property>

<property name="capitals">

<props>

represents one element ← <prop key="AP"> hyd </prop>
<prop key="MH"> Mumbai </prop>
<prop key="KR"> Bangalore </prop>

</props>

</property>

</bean>

</beans>

→ In java.util.Properties Map datastructure
element key must be string and
value must be string.

→ In Other Map datastructure element
key can be any Java object and
value can be any Java object.

Step-VII :- add Client appn to Src

right click on Src → new → class → name: DemoClient

// DemoClient.java

public class DemoClient

{

 public void main (String[] args)

{

 // Activate BeanFactory Container

 XmIBeanFactory factory = new XmIBeanFactory (new ClassPathResource ("Springfg.xml"));

 // get SpringBean obj

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

 // call B.methods

 System.out.println (bobj.sayHello ());

y

y

System.out.println (bobj.sayHello ());

Step-VII:- Run the appn.

Rightclick in the DemoClient.java → Run as → Java application.

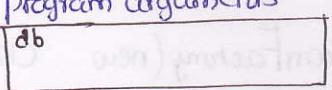
→ While working with DriverManagerDataSource class we can use `java.util.Properties` type bean property called ConnectionProperties. to specified username and Password.

Wrt to SpringCfg.xml of ses3 appn on 18/02/2013 appn.

```
<!-- SpringCfg.xml -->  
<!DOCTYPE >.....>  
<beans>  
    <bean id="dbs" class="org.springframework.jdbc.datasource.DriverManagerDataSource">  
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>  
        <property name="url" value="jdbc:oracle:oci8:@orcl"/>  
        <property name="connectionProperties">  
            <props>  
                <prop key="user"> scott </prop>  
                <prop key="password"> tiger </prop>  
            </props>  
        </property>  
    </bean>
```

→ To Pass Command line argument Values to MyEclipse IDE appn

Rightclick in the SourceCode of appn → Run as → Run Configurations

→ Arguments tab → 
val1 val2 val3.... valn
args[0] args[1] args[2] ... args[n-1]

Constructor Injection

- If Spring Container is using Parameterized Constructor to create Bean class object & to assign Dependent Values to Bean properties then it is called Constructor Injection.
- Based on <constructor-arg> tag appearance count under <bean> tag the spring container picks up appropriate Parameterized constructor to perform Constructor injection.

Example cппн Demo.java → same as previous cппн

DemoBean.java

public class DemoBean implements Demo

{

//DemoBean properties

int age;

String name;

float salary;

//parameterized Constructors Supporting

====} place 3-param constructor

Select Bean properties → source → generate Constructor using field → select All

Select Bean properties → source → generate Constructor using field → select All

→ OK.

// implement Business method

public String sayHello()

{

return "age=" + age + "name=" + name + "salary=" + salary;

}

}

<springcfg.xml-->

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

<constructor-arg value="30"/>

<constructor-arg value="rcya"/>

<constructor-arg value="34563.69"/>

</bean>

</beans>

→ Since <constructor-arg> tag is placed for 3 times the Spring Container uses three param constructor for

} These values must be specified in the order the parameters are available in Constructor.

SpringBean instantiation and for Constructor Injection,

// DemoClient.java

Same as previous appn.

NOTE:-

While Working with Constructed injection we need to Configure dependent values in the order the parameters are placed in the constructor.
→ If not interested we can resolve/ identify parameters based on their type of Index.

→ If Each Parameter type is unique then we can identify that parameter based on "type" in Spring Configuration file during constructor injection configurations.

Eg:- in SpringBean

```
public DemoBean(int age, String name, float salary)
{
    this.age = age;
    this.name = name;
    this.salary = salary;
}
```

in SpringConfiguration file

```
<beans>
<bean id="db" class="DemoBean">
    <constructor-arg type="int" value="30" />
    <constructor-arg type="float" value="3456.127" />
    <constructor-arg type="java.lang.String" value="rajat" />
</bean>
</beans>
```

→ If multiple parameters of Constructor Contains Same type then we can resolve/ identify them based on their ~~Antix~~ indexes.

Eg:- in SpringBean

```
int a,b,c;
public DemoBean(int a,int b,int c)
{
    this.a=a;
    this.b=b;
    this.c=c;
}
```

in Spring Configuration file

```
<bean id="db" class="DemoBean">
    <constructor-arg index="0" value="10"/> goes to "a"
    <constructor-arg index="1" value="30"/> goes to "c"
    <constructor-arg index="2" value="20"/> goes to "b"
```

</bean>

</beans>

- There is no provision to specify BeanProperty name (or) Constructor parameter name in <constructor-arg> tag. for this reason we need to identify constructor parameters either through their ^{types} (or) indexes.

V

→ There is a provision to activate BeanFactory container based on the already activated BeanFactory container. In this process the 2nd BeanFactory container acts as the child container, and the first BeanFactory container acts as parent container.

Example application:-

Demo.java → Same as previous appn

// 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 "d=" + d;

}

}

// DemoCfg1.xml (Parent file)

<beans>

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

</beans>

// DemoCfg2.xml (Child file)

<beans>

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

<property name="d"><ref bean="dt"/></property>

</bean>

</beans>

// DemoClient.java

imports

public class DemoClient {

psvm (String[] args) {

```
// Parent BeanFactory contained  
XmiBeanFactory factory1 = new XmiBeanFactory(new ClassPathResource("DemoCfg1.xmi"));  
// child BeanFactory contained  
XmiBeanFactory factory2 = new XmiBeanFactory(new ClassPathResource("DemoCfg2.xmi"));  
factory1);  
// get SpringBean class obj  
Demo bobj = (Demo)factory2.getBean("db");  
System.out.println("Result is " + bobj.sayHello());  
}
```

Q:- What is the diff b/w Bean and local attribute of <ref> tag?

Ans:- "bean" attribute searches for the dependent bean in current Spring Configuration file and also in parent Spring Configuration File.

Eg:- <property name="d"> <ref bean="dt"/> </property>

"local" attribute searches for the dependent bean in current Spring Cfg file

Eg:- <property name="d"> <ref local="dt"/> </property>

<property name="d" ref="dt"/> is same as <property name="d"><ref bean="dt"/> </property>

- We can make Spring Container to keep Spring Bean class objects in the following scopes.
- (1) Singleton
 - (2) Prototype
 - (3) request
 - (4) session

(1) Singleton:- Container creates only one object of Spring Bean class even though factory.getBean() method is called for multiple times with same Bean id.

Eg:- `<bean id="db" class="DemoBean" scope="singleton">`

`</bean>`

(2) Prototype:- Container creates multiple objects for Spring Bean class for multiple factory. factory.getBean() method calls having same Bean id.

Eg:- `<bean id="db" class="DemoBean" scope="prototype">`

`</bean>`

(3) request:- Container keeps Spring Bean class object as request attribute value in webappn

Eg:- `<bean id="db" class="DemoBean" scope="request">`

`</bean>`

(4) session:- Container keeps Spring Bean class object as session attribute value in webappn

Eg:- `<bean id="db" class="DemoBean" scope="session">`

`</bean>`

NOTE:- request, session scopes are applicable only in Web environment ...

The Java class that allows us to create only one Java Object per JVM is called Singleton Java class. Even though java class allows multiple objects creation if we are creating only one object for it then it is not called Singleton java class.

When Spring Bean Scope is Singleton our springBean class does not become Singleton java class.

→ Singleton scope Spring Beans can be configured as dependents to prototype scope SpringBean properties and vice versa... In this whole process Container does not change scopes of any Bean.

NOTE:- most recommended scope for SpringBean is Singleton scope.

Q:- Can be placed only parameterized constructors in SpringBean when all properties of SpringBean are configured for SetterInjection?

Ans:- Not Possible,

spring
Bcz Container looks to use 0-param constructor to create the above Spring Bean class object.

ApplicationContext Container :-

It is enhancement of BeanFactory container.

This Container is available in spring context / JEE module.

It is also light-weight container to activate this Container create Object for a class that implements org.sf.context.ApplicationContext (I).

This Interface is Subinterface of org.sf.beans.factory.BeanFactory (I).

There are 3 important classes implementing ApplicationContext (I). we can use this classes to activate ApplicationContext Container

① FileSystemXmlApplicationContext (org.sf.context.support)

→ Activates Spring Container by locating Springcfg file from the specified path of FileSystem. ↳ ApplicationContext

FileSystemXmlApplicationContext ctx = new FileSystemXmlApplicationContext ("Democfg.xml")

② ClassPathXmlApplicationContext (org.springframework.context.support)

→ Activates Spring Container by locating Spring cfg file from the values added to the Classpath env... Variable.

ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("DemoCfg.xml");

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

→ Activates ApplicationContext container in Webapplications

X*

25/02/2013

→ Bean property can be taken as static or non-static but the setter method taken for setter injection must be taken as non-static method.

Using ApplicationContext Container we can perform all the activities with BeanFactory Container but it can be also used to perform additional functionalities like

1) Pre-instantiation of Singleton beans

2) ability to work with multiple Spring Configuration files in Single instance of Container activation.

3) Ability to read values from properties file

4) Support for i18n [internationalization]

5) Support for EventHandling and etc...

Note:- all the above features are not there in BeanFactory container.

All these features are exclusive features of Application Context Container.

* In Environment multiple SpringBeans developed by multiple programmers will be configured in different Spring Configuration files. we can activate Application Context Container by specifying this multiple Spring Configuration files.

//Activate ApplicationContext Container

String cfg[] = {"DemoCfg1.xml", "DemoCfg2.xml"};

ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext(cfg);

NOTE:- in BeanFactory env... to cook with multiple configuration files we need

to activate multiple instances of BeanFactory container.

→ If SpringBean objects are created by container in the moment SpringContainer is activated then it is called pre-instantiation of SpringBean.

ApplicationContext Container performs pre-instantiation on "Singleton scope" SpringBeans.

Q:- What happens if "prototype" scope SpringBean is taken as DependentValue to "Singleton" scope SpringBean class property?

Ans:- Container creates "Prototype" scope SpringBean class object and Singleton Scope SpringBean class object through pre-instantiation process if the Container is ApplicationContext Container. But it does not change the scope of any Bean.

Q:- When Spring Beans are configured in Spring Configuration file how can be enable pre-instantiation (~~sing~~ singleton creating objects when container is Activated) only on four SpringBeans.

Ans:- Take four SpringBeans having Singleton scope and take other 6-BEANS having prototype scope and also make sure that prototype scope SpringBeans are not dependent to Singleton scope SpringBeans

factory.getBean(-) is called?
↳ BeanFactory.

Q:- What happens when factory.getBean(-) is called?

Ans:- Creates and returns SpringBean class object if SpringBean scope is "prototype".

Creates and returns SpringBean class object if SpringBean scope is "Singleton".

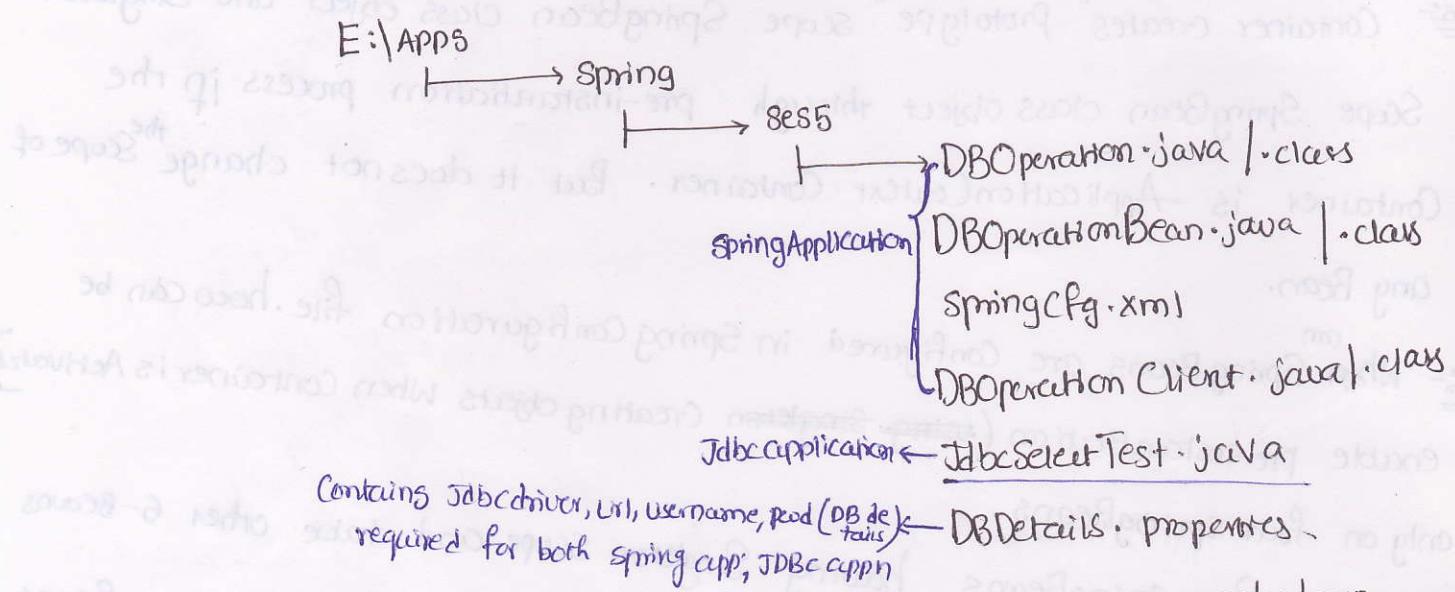
Q:- What happens when ctx.getBean(-) is called?

Ans:- Creates and returns SpringBean class object if SpringBean scope is "prototype".

Creates and returns SpringBean class object if SpringBean scope is "Singleton".

→ The text file that maintains the entries in the form of key = value pairs is called "TextProperties file".
The standard principle of software industry is don't hard code any values in our application that are changeable in the feature. It is recommended to pass these values to application from outside the application by taking the support of Text Properties file / XML file.

→ If Spring Application placed along with other Technology based apps, it is recommended to make Spring Application to use same properties file that other applications are using to centralized common data.



→ To make Spring Configuration file getting certain values from properties file we place placeholders specifying the keys of the properties file (like \${key?})
To recognize these placeholders and to make container gathering values from properties file based on the placeholders that are specified we need to configure org.springframework.beans.factory.config.PropertyPlaceholderConfigurer as spring bean is SpringFg file.

Source code of the above application :-

jdbc driver and DB details

// DBDetails.properties

my.driver = Oracle.jdbc.driver.OracleDriver

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

my.dbuser = Scott

my.dbPswd = tiger

Spring application

DBOperation.java → Same as 18/02/2013 appn

DBOperationBean.java → Same as "

makes container to recognize
placeholders & properties file.

<!-- Spring Cfg.xml -->

<!DOCTYPE >

<beans>
<bean id="propconfig" class="org.sf.beans.factory.config.PropertyPlaceholderConfigurer">
 <property name="location" value="DBDetails.properties"/>
 ↳ properties file in current directory.

</bean>

<bean id="drds" class="org.sf.jdbc.datasource.DriverManagerDataSource">
 <property name="driverClassName" value="\${my.driver}"/>
 ↳ placeholder
 <property name="url" value="\${my.uri}"/>
 <property name="username" value="\${my.dbuser}"/>
 <property name="password" value="\${my.dbPswd}"/>

</bean>

<bean id="dob" class="DBOperationBean">

 <property name="ds" ref="drds"/>

</bean>

</beans>

// DBOperationClient.java

import org.sf.context.support.*;

public class DBOperationClient

{

 public void main(String[] args) throws Exception

{

 // Activate Application Context Container

 FileSystemXmlApplicationContext ctx = new FileSystemXmlApplicationContext("springcfg.xml")

```
// get Spring Bean obj from container  
DBOperation bobj = (DBOperation) ctx.getBean("dob");  
// call b. methods  
S.O.P("Emp salary is "+bobj.fetchSalary(7499));  
S.O.P("Emp Name is "+bobj.fetchName(7499));  
} // main  
} // class  
1> javac *.java  
1> java DBoperationClient
```

JDBC application :

```
// JDBCSelectTest.java  
import java.sql.*;  
import java.util.*;  
import java.io.*;  
  
public class JdbcSelectTest  
{  
    public static void main(String[] args) throws Exception  
    {  
        // locate properties file  
        FileInputStream fis=new FileInputStream("DBDetails.properties");  
        // load the content of properties file to java.util.Properties class Obj  
        Properties p=new Properties();  
        p.load(fis);  
  
        // read content from Properties class Obj  
        String s1=p.getProperty("my.driver");  
        String s2=p.getProperty("my.un");  
        String s3=p.getProperty("my.dbuser");  
        String s4=p.getProperty("my.dbpwd");  
  
        // write JDBC code  
        Class.forName(s1);  
        Connection con=DriverManager.getConnection(s2,s3,s4);  
        Statement st=con.createStatement();
```

```

ResultSet rs = st.executeQuery("select * from Student");
while(rs.next())
{
    System.out.println(rs.getInt(1) + " " + rs.getString(2) + " " + rs.getString(3));
}
//close jdbc objs
rs.close();
st.close();
con.close();
}
} //main
} //class
1>javac JdbcSelectTest.java

```

→ BeanFactory Container cannot recognize and use place holders and properties file even though the Special Bean "property placeholderConfigurer" is configured.

→ We can configure multiple properties file in Spring Environment. If this multiple properties file contain same key with different values then The values collected from lastly configured properties file will be taken and used

In <!-- spring.cfg.xml -->

Eg:- <beans>
<bean id="propconfig" class="org.sf.beans.factory.config.PropertyPlaceholderConfigurer">
<property name="locations">
 ↳ Any type property.
<list>
 <value>DBDetails1.properties </value>
 <value>DBDetails.properties </value>
</list>
<property>
</beans>

* Internationalization

→ Country + Language = locale.

java.util.Locale class is given for working with locales.

Eg:- en-US, fr-FR, fr-CA, hi-IN etc....
↳ French as it's speak in CANADA.

→ Making our application working for different locales is nothing but enabling i18n on the application. For this we need to take care of the presentation labels, numberformats, Dateformats, currency symbols and etc....

→ To enable i18n on our applications we need multiple locales specific properties files including Base properties file. If no matching properties file is found then our application uses labels collected from the Baseproperties file. We can use google translator & unicode editor tools to get presentation labels in different languages. ~~use~~ to get their Unicode numbers.

→ If label can't be typed using English alphabets get their Unicode numbers and supplied to the applications. To get Unicode numbers for the content of regional languages like Telugu, Hindi and etc... we can use Unicode editor tool downloaded from "higopi.com"

27/02/2013

Procedure to get Unicode numbers for certain Hindi words and by using Unicode editor and ~~native2ASCII tool~~ native2ascii tool

Step 1: Download unicode editor tool from www.higopi.com.

Step 2: Launch "unicode editor tool" through index.html and type the following 4 Hindi words निकाली, संरक्षित, फ़रवी, रघु

Step 3: Copy above 4 Hindi words to Notepad text file.

P1.txt

????

??????

???

????

NOTE:- While saving this file choose the encoding mode as unicode.

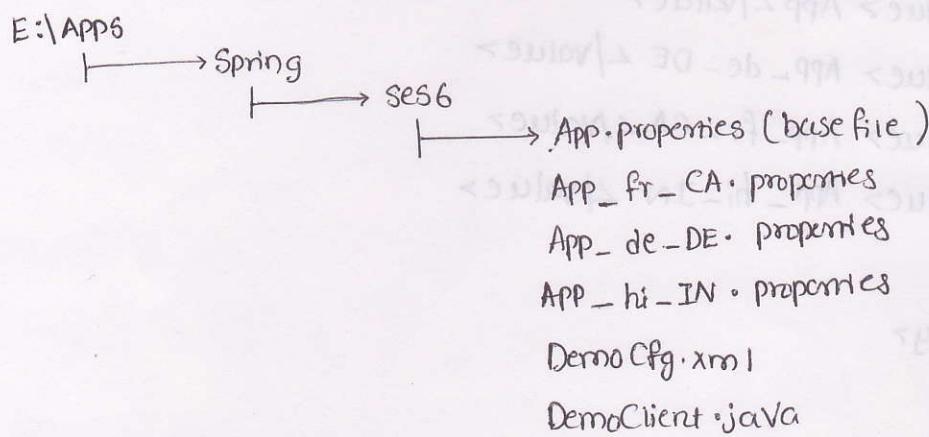
Step 4:- Use native2ascii tool to get Unicode numbers

↳ native2ascii - encoding unicode p1.txt p2.txt ↳ contains the generated unicode numbers
↳ built-in tool of jdk

You can make ApplicationContext container ready for make i18n by Configuring the following Spring Bean having properties file names.

org.springframework.context.support.ResourceBundleMessageSource

Spring based i18n App Example appn



// App.properties (BaseFile)

Str1 = delete

Str2 = save

Str3 = stop

Str4 = cancel

// APP_fr_CA.properties (French)

Str1 = EFFACER

Str2 = SAUF

Str3 = ARRÊT

Str4 = ANNULER

// APP_de_DE.properties (German)

Str1 = LÖSCHEN

Str2 = DENN

Str3 = ZUG

Str4 = ABSAGEN

// APP_hi_IN.properties (Telugu)

Str1 = \u0928\u093f\u0915\u093e\u0932\u094a Unicode representation for Architect

Str2 = ...

Str3 = ...

Str4 = ...

} copy from P2.txt file

NOTE:- The filename of Base File is must be there in Locale specific Filenames.
→ all properties file must contain same keys having different values

<!-- DemoCfg.xml -->

```
<beans>
  <bean id="messageSource" class="org.sf.context.support.ResourceBundleMessageSource">
    <!-- Fixed Bean ID -->
    <property name="basename">
      <list>
        <value>App</value>
        <value>APP_de_DE</value>
        <value>APP_fr_CA</value>
        <value>App_hi_IN</value>
      </list>
    </property>
  </bean>
</beans>
```

//DemoClient.java

```
import org.sf.context.support.*;
import java.util.*;
import javax.swing.*;
import java.awt.*;
```

```
public class DemoClient {
```

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

```
    // Locale Object
    Locale l = new Locale(args[0], args[1]);
    // language ↑ country ↑
```

```
    // Activate Application Context Container
```

```
    ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("DemoCfg.xml");
```

```
    // Get Msgs from properties file
```

```
    String msg1 = ctx.getMessage("str1", null, "default msg1", l);
```

```
    String msg2 = ctx.getMessage("str2", null, "default msg2", l);
```

```
    String msg3 = ctx.getMessage("str3", null, "default msg3", l);
```

```
    String msg4 = ctx.getMessage("str4", null, "default msg4", l);
```

↳ key in the property file.

// Develop Swing Frame

```
JFrame jf = new JFrame();
Container cp = jf.getContentPane();
```

// Create buttons by getting labels from activated

```
 JButton b1 = new JButton(msg1);
```

```
 JButton b2 = new JButton(msg2);
```

```
 JButton b3 = new JButton(msg3);
```

```
 JButton b4 = new JButton(msg4);
```

```
 cp.setLayout(new FlowLayout());
```

```
 cp.add(b1);
```

```
 cp.add(b2);
```

```
 cp.add(b3);
```

```
 cp.add(b4);
```

```
 jf.pack();
```

```
 jf.show();
```

```
} // main
```

```
} // class
```

//> javac DemoClient.java

//> java DemoClient fr FR

//> java DemoClient de DE

//> java DemoClient x y

(uses APP_Fr_FR.properties)

(uses App.properties)

→ ctx.getBean(-) gives SpringBean class object Similierly ctx.getMessage (-,-,-) gives value of the properties file based on the given key.

Can you Compare BeanFactory Container and Application Context Container.

Feature	BeanFactory	Application Context
Bean Instantiation / Setting wiring Dependency Injection	Yes	Yes
Ability to work with multiple spring config files with single instance of container.	No	Yes
Spring pre-instantiation of Beans	No	Yes
Recognition of placeholders and properties files	No	Yes
I18N Support	No	Yes
Application Event handling	No	Yes
Automatic Bean Post Processor registration	No	Yes
Automatic BeanFactory Post Processor registration	No	Yes

Conclusion:- while using Spring in memory sensitive applications like Applets prefer using BeanFactory Container (uses less memory). In all other applications prefer

Application Context Container.

→ Action performed on the Component / Object is called "Event". The process of executing logic when Event is raised is called "Event Handling". To perform this Event Handling we need Event Listeners. When the Application Context Container is started or stopped the Application Event will be raised. We can handle this Application Event by using Application Listener. In this process we can notice the amount of time that the Application Context Container is there in running mode.

Example Application:-

Step-I:- keep 1st appn ready. (ses 1)

Step-II:- Develop separate Listener class implementing org.springframework.context.ApplicationListener Interface.

```
// MyListener.java
import org.springframework.context.*;
import java.util.*;
```

```

public class MyListener implements ApplicationListener
{
    long stime, endtime;
    public void onApplicationEvent(ApplicationEvent ae)
    {
        System.out.println("onApplicationEvent of MyListener class");
        if(ae.toString().indexOf("ContextRefreshed Event") != -1) // when Container is started
        {
            stime = System.currentTimeMillis();
            System.out.println("The Container is started at " + new Date());
        }
        else if(ae.toString().indexOf("ContextClosed Event") != -1) // when Container is stopped
        {
            endtime = System.currentTimeMillis();
            System.out.println("The container is stopped at " + new Date());
            System.out.println("The current container is in running mode for " + (endtime - stime) + "msecs");
        }
    } // method
} // onApplicationEvent()
} // class

```

Step-III:- configure the above Listener class in spring.cfg file.

```

<!-- DemoCfg.xml -->
<!DOCTYPE >

<beans>
    <bean id="mb" class="MyListener" />
    <bean id="db" class="DemoBean" />

```

Step-IV:- Activate Application Context Container in the Client appn and stop that Container at the end by calling ctx.close();

→ SpringBean can have two user-defined methods as lifecycle methods.

① init lifecycle method

- Container executes this method right after Bean Instantiation and Dependency injection
- This method is useful to check whether bean properties are assigned with Valid Values or not. It is also useful to check whether mandatory bean properties are assigned with values or not.

② destroy lifecycle method

- Container executes this method when it is about to destroy our SpringBean class
- This method is useful to nullify bean properties and release non-Java resources associated with Spring beans (like closing JDBC connection)

These two life cycle method must be specified in SpringBean during configuration

Spring cfg file using init-method , destroy-method attributes of <bean> tag.

Example Application :-

// Demo.java

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

// DemoBean.java

```
public class DemoBean implements Demo
```

{ // Bean Properties

```
    private String msg;  
    private int age;
```

~~Has~~ setters & getters

// user-defined init life cycle Method

```
public void myInit() throws Exception
```

{
 System.out.println("DemoBean : myInit()");

Validating
Bean
property Value.

```
    if (age <= 0 || msg == null)  
    {  
        throw new Exception ("set proper values to Bean property");  
    }
```

Re-project → Build & run →
Add libraries → My Eclipse library

Run → Search if we want
to copy information

// user-defined destroy() life cycle method

```
public void destroy()
```

```
{
```

S.O.P ("DemoBean: myDestroy()");
 ~~Setting Bean properties Value~~
 { age = 0; } nullifying
 { msg = null; } Bean properties.
}

// B-method implementation

```
public String sayHello()
```

```
{  
    return "msg=" + msg + "age=" + age;  
}
```

```
}
```

<!-- SpringCfg.xml -->

<!DOCTYPE >

```
<beans>  
    <bean id="db" class="DemoBean" init-method="myInit" destroy-method="myDestroy">  
        <property name="age" value="30"/>  
        <property name="msg" value="hello"/>  
    </bean>  
</beans>
```

// DemoClient.java

Same as first application (can use any container)

```
public class DemoClient
```

```
{ p s v m (String[] args)
```

```
{
```

// Activate Spring container

```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("SpringCfg.xml")
```

// get SpringBean class object

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

// call B-method

```
S.O.P (bobj.sayHello());  
    // class contained  
        ctx.close();
```

* if container is Application Context Container call ctx.close(); at the end.

* if the container is BeanFactory container then call factory.destroySingletons();

→ While Configuring pre-defined, third party supplied java classes as SpringBeans in spring configuration file Identification of lifecycle methods from huge no. of methods that are available in class is practically not feasible. more ever programmer may forget to configure these lifecycle methods.

To overcome this problem it is recommended to make SpringBean class implementing two special interfaces (InitializingBean, DisposableBean).

and we can use methods of these two interfaces as alternate for user-defined lifecycle methods.

org.springframework.beans.factory.InitializingBean interface gives

afterPropertiesSet() method
→ can be used as alternate to custom-init() method.

org.springframework.beans.factory.DisposableBean interface gives

destroy() method

→ can be used as alternate to custom-destroy() method.

→ while working with these two methods there is no need of configuring method names in Spring Cfg file.

→ POJO class can act as SpringBean but it is not mandatory that every SpringBean must be a POJO class. When SpringBean class implements the above two interfaces then it acts as non-POJO class.

→ If SpringBean class implements above two interfaces and also contains custom lifecycle methods then both will be executed. (custom-init method executes after afterPropertiesSet(), custom-destroy method executes after destroy()).

Example appn:-

// Demo.java same as previous appn

// DemoBean.java

public class DemoBean implements Demo, InitializingBean, DisposableBean

{ // Bean properties

private String msg;

private int age;

Setters & Getters

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

public void destroy() throws Exception {
    System.out.println("DemoBean: destroy()");
    age = 0;
    msg = null;
}

public void afterPropertiesSet() throws Exception {
    System.out.println("DemoBean: afterPropertiesSet()");
    if (age <= 0 || msg == null)
        throw new Exception("set proper values to Bean property");
}
```

<!-- DemoCfg.xml --> Same as previous appn. but don't specify init-method, destroy-method attributes.

//DemoClient.java Same as previous appn.

- If Spring Bean class is not implementing InitializingBean, DisposableBean Interfaces then only think about Configuring certain methods as Custom-lifecycle methods. It contains only Spring supplied Java classes
- Since third party supplied classes, Jdk Supplied classes cannot implements Initializing Bean, DisposableBean interfaces. so configure custom lifecycle methods while working with ~~sp~~ those Spring Beans.

Spring Supports 3 modes of Injection:-

01/03/2013

- 1) Setter Injection
- 2) Constructor Injection
- 3) Interface Injection

XxxAware interfaces we can go for Interface Injection.

using interface injection we can inject only special obj to our spring Bean class properties

XxxAware interfaces are

- ① org.springframework.BeanFactoryAware
→ useful to inject BeanFactory Container to our SpringBean class
- ② org.springframework.ApplicationContextAware
→ useful to inject ApplicationContext Container to our SpringBean class
- ③ org.springframework.BeanNameAware
→ useful to inject beanid spring bean to Springbean class

while performing Interface injection our Spring Bean class acts as non-POJO class.

Example Application performing Interface Injection :-

Hi Demo Java ~~before as previous app~~

NOTE:- Through Interface Injection we can inject Underlaying Containers to SpringBean and we can use these containers to know about other beans Beings from Current SpringBean class.

For example appn on Interface Injection refer Supplementary handout given on 01/03/2013

→ If you want to write BeanProperties Validation logic and Initialization logic outside the SpringBean classes without disturbing the source code of existing SpringBeans then we can use Special java class acting as

③ BeanPostProcessor. for this the class must implement org.springframework.config.BeanPostProcessor (2)

→ This Interface contains 2 methods

① public Object postProcessBeforeInitialization (Object bean, String bname)
represents Bean logical name
represents SpringBean obj

→ executes after all Dependency Injection

→ useful to modify Bean properties after dependency Injection

② public Object postProcessAfterInitialization (Object bean, String bname)
represents SpringBean obj
represents Bean logical name

→ executes after custom-init method

→ useful to modify Bean properties after complete Initialization.

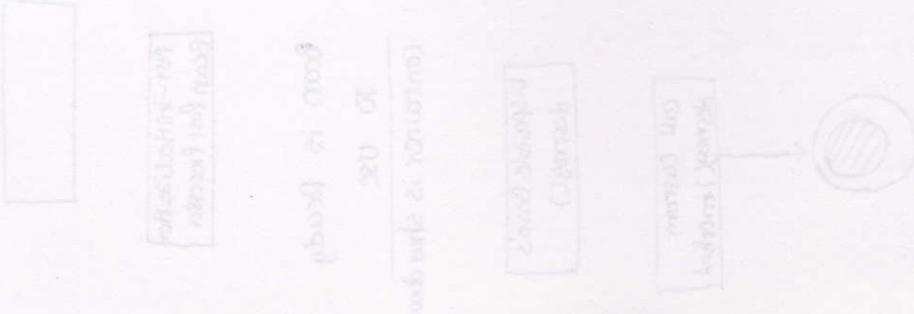
NOTE:- Custom init() method and afterPropertiesSet() related logics are specified to one Spring bean whereas the Bean post processor (the above two methods related logics are common for multiple Spring Beans).

For Example application on BeanPostProcessor implementation refer the supplementary handout given on 01/03/2013

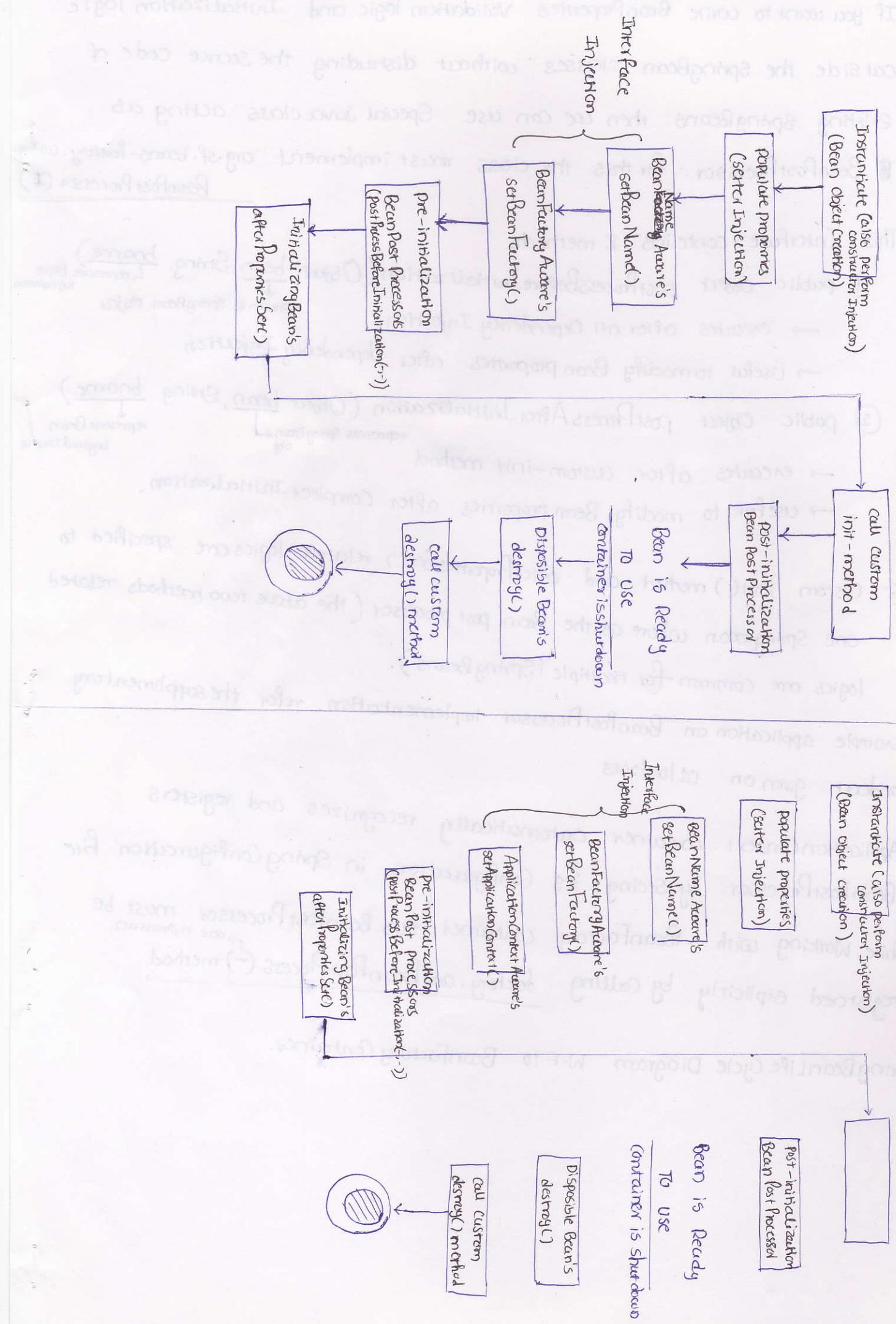
→ Application Context container automatically recognizes and registers BeanPostProcessor by seeing its Configuration in Spring Configuration file.

While Working with BeanFactory contained this BeanPost Processor must be registered explicitly by calling factory.addBeanPostProcess(→) method.

Spring Bean Life Cycle Diagram w.r.t. to BeanFactory Container.



Bean life cycle Diagram wrt Application Context Container



```

1 Spring App on Interface Injection and Bean Post Processor Impl
2 =====
3 -----Demo.java-----
4 public interface Demo {
5     public String sayHello();
6 }
7 -----DemoBean.java-----
8 import org.springframework.beans.BeansException;
9 import org.springframework.beans.factory.BeanFactory;
10 import org.springframework.beans.factory.BeanFactoryAware;
11 import org.springframework.beans.factory.BeanNameAware;
12 import org.springframework.context.ApplicationContext;
13 import org.springframework.context.ApplicationContextAware;
14
15 public class DemoBean implements Demo, BeanNameAware, BeanFactoryAware, ApplicationContextAware
16 {
    //Special Bean properties
    private String bname;
    private BeanFactory factory;
    private ApplicationContext ctx;
17
18    String msg;
19
20    //setXxx() for setter Injection
21    public void setMsg(String msg) {
22        System.out.println("DemoBean:setMsg(-)");
23        this.msg = msg;
24    }
25
26    //method of ApplicationContextAware()
27    public void setApplicationContext(ApplicationContext ctx)
28        throws BeansException {
29        System.out.println("DemoBean:setApplicationContext(-)");
30        this.ctx=ctx;
31    }
32
33    //method of BeanFactoryAware()
34    public void setBeanFactory(BeanFactory factory) throws BeansException {
35        System.out.println("DemoBean:setBeanFactory(-)");
36        this.factory=factory;
37    }
38
39    //method of BeanNameAware()
40    public void setBeanName(String name) {
41        System.out.println("DemoBean:setBeanName(-)");
42        this.bname=name;
43    }
44
45    public String sayHello() {
46
47        System.out.println("current bean id"+bname);
48        System.out.println("current bean is singleton?"+ctx.isSingleton(bname));
49        System.out.println("current bean is prototype?"+ctx.isPrototype(bname));
50        System.out.println("all beans are");
51        String id[] = ctx.getBeanDefinitionNames();
52        for(int i=0;i<id.length;++i)
53        {
54            System.out.println(id[i]+"....");
55        }
56    }
57
58    return "Good morning:msg="+msg;
59    }//method
60
61
62 }//class
63 -----DemoCfg.xml-----
64 <?xml version="1.0" encoding="UTF-8"?>
65 <beans
66     xmlns="http://www.springframework.org/schema/beans"
67     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
68     xmlns:p="http://www.springframework.org/schema/p"

```

Interfaces for Interface injection

methods supporting Interface Injection

```

69 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-bean.xsd">
70 <bean id="mp" class="MyProcessor"/> → BeanPost Processor Configuration
71
72 <bean id="db" class="DemoBean" >
73   <property name="msg" value="hello"/>
74 </bean>
75
76 <bean id="dt" class="java.util.Date"/>
77 </beans>
78 -----DemoClient.java-----
79 import org.springframework.context.support.FileSystemXmlApplicationContext;
80
81 public class DemoClient {
82
83   public static void main(String[] args) {
84     //Activate Spring container
85     FileSystemXmlApplicationContext ctx=new FileSystemXmlApplicationContext("./src\\DemoCfg.xml");
86     // get Spring Bean class obj
87     Demo bobj=(Demo)ctx.getBean("db");
88     // call B.method
89     System.out.println(bobj.sayHello());
90   } //method
91 } //class
92 -----MyProcessor.java----- → BeanPost Processor
93 import java.util.Date;
94 import org.springframework.beans.BeansException;
95 import org.springframework.beans.factory.config.BeanPostProcessor;
96 public class MyProcessor implements BeanPostProcessor {
97   @Override
98   public Object postProcessBeforeInitialization(Object bean, String bname) throws BeansException {
99     System.out.println("MyProcessor: postProcessBeforeInitialization()");
100    if(bean instanceof DemoBean )
101    {
102      DemoBean obj1=(DemoBean)bean;
103      obj1.msg="hello1";
104      return obj1;
105    }
106    if(bean instanceof java.util.Date)
107    {
108      Date obj2=(Date)bean;
109      obj2.setYear(130);
110      return obj2;
111    }
112    return bean;
113  }
114  public Object postProcessAfterInitialization(Object bean, String bname) throws BeansException {
115    System.out.println("MyProcessor: postProcessAfterInitialization()");
116    return bean;
117  }
118 }
119 }
```

Factory Bean :-

- If Normal Bean is Configured as DependentBean to Bean property then
 - Normal Bean class Object will be injected directly to Bean Property.
- Factory Bean is a bean that generates resultant Object always. If Factory Bean is configured as DependentBean then FactoryBean class object will not be injected to property but the resultant Bean generated by FactoryBean will be injected.

Eg:-

```

<bean id="tb" class="TestBean"/>
<bean id="db" class="DemoBean">
  <property name="t1" ref="tb"/>
</bean>
```

if "TestBean" normal spring bean then the "t1" property of DemoBean class will be injected with "TestBean" class obj.

if "TestBean" is Factory Bean then the "t1" property of DemoBean class will be injected with the resultant object given by TestBean class.

Spring Bean becomes Factory Bean when it implements

This interface contains 3 methods

- ① getObjec() → returns the resultant obj
- ② getObjectType() → returns the type of resultant obj
- ③ isSingleton() → returns whether resultant object is singleton or not

NetBeans

type: IDE SW for Java

Vendor: SunMS (Oracle Corp)

version: 6.7.1

OpenSource

gives GlassFish 2.x as built-in server

to download SW: www.netbeans.org

for docs: www.netbeans.org

Resources of the application

- 1) Demo.java (spring Interface)
- 2) DemoBean.java (spring Bean)
- 3) Demofg.xml (spring file)
- 4) DemoBean.java (client App)
- 5) TestBean.java (FactoryBean)

*) procedure to deal with Spring appn that deals with userdefined FactoryBean development by using netbeans 6.7.1 :-

Step-I:- Create Java project in NetBeans IDE

File Menu → New → Project → Java → Java Application → Project Name [SpringApp] → Next
→ create Main Class → Finish

Step-II:- Add spring Libraries to the project

Libraries folder of
Right Click on Project → Add Library → Spring Framework 2.5

Step-III:- Add the above .java resources to the Source Pkgs folder of the project

Rightclick on Source Pkg → New → Java Interface → className: [Demo]

```
//Demo.java
public interface Demo {
    public String sayHello();
}
```

Step-IV:- Rightclick on Source Pkg → New → Java Class → className: [DemoBean]

```
//DemoBean.java
public class DemoBean implements Demo {
    Object obj;
}
```

Right click → Insert Code → setters → OK

// Setters method supporting setter injection

```
public void setObj(Object obj) {
    this.obj = obj;
}
```

```
public String sayHello() {
    return "Good Morning " + obj.toString();
    System.out.println("obj class name " + obj.getClass());
}
```

Step-V:- R.C. on New Java Package → New → Java Class → className: [TestBean]

//TestBean.java

```
public class TestBean implements FactoryBean {
    
```

```
    public Object getObject() throws Exception {
        return new java.util.Date();
    }
}
```

→ returns Dateclass obj as the resultant obj

```
public Class getObjectType () {
```

return Date.class; → specifies the type of the Resultant Object

```
}
```

```
public boolean isSingleton() {
```

return true; → specifies the resultant object scope is Singleton

```
}
```

Step-VI:

R.C. on Source package → New → Java class → Name: DemoClient → Finish.

```
//DemoClient.java
```

```
public class DemoClient
```

```
{
```

```
    public (String[] args)
```

```
{
```

```
    //Activate Spring Container
```

Class Path

File System XML Application Context

ctx = new

Class Path

File System XML Application Context

(DemoConfig.xml)

```
//get Spring Bean Objects
```

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

args[0] ↗ bean id of DemoBean

```
Object obj= ctx.getBean("tb");
```

args[1] ↗ bean id of TestBean

```
System.out.println("Obj data is "+obj);
```

```
S.O.P("obj class name is "+obj.getClass());
```

```
//Call B. methods
```

```
String result=bobj.sayHello();
```

```
S.O.P("result is: "+result);
```

```
}
```

R.C on Main → properties → Run

→ main class [DemoClient].

Argument [db tb] → ok.

Right click on springmos

Step-VII: Add Configuration file to the project

R.C. on Source Pkg's → New → Other → XML → XML Document → Filename: DemoCfg

→ next →

① DTD-Constraint Document → next →

DTD public id : [-//SPRING/DTD//Bean 2.0/EN]

DTD System id : [http://www.springframework.org/dtd/spring-beans-2.0.dtd] → Finish

Document Root : beans

```
<beans>
```

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

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

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

```
</beans>
```

since TestBean is FactoryBean generating Date class object as ResultantObject. so, the Date class object will be injected with Date class object.

Step-VIII:- run the client application

Right click in DemoClient.java → Run File.

The predefined SpringBean classes that are developed to gather objects from registries, pools and etc.. are designed as FactoryBeans.

Eg:-

① org.springframework.jndi.JndiObjectFactoryBean

TimerFactoryBean

RMI registry FactoryBean and etc...

To set Command line arguments to the Application of netbean IDE

Right Click on project → properties → Run → Main class name: DemoClient,
arguments:

db tb
vals vals vals... vals
args\$0 args\$1 args\$2 args\$3

Auto wiring :-

→ performing Dependency Injection on BeanProperties is called wiring.

There are two modes to performing this wiring.

① Explicit wiring :-

→ here bean properties will be configured explicitly for coining using `<property>`, `<constructor-arg>` tags.

② Auto wiring :-

→ here bean properties will not be configured explicitly. But Container automatically detects and injects values to bean properties.
→ To enable Autowiring we need to use "autowire" attribute of `<bean>`.

limitation with Auto wiring :-

- ① Autowiring is not possible on Simple, String properties. Autowiring is possible only on reference type bean properties.
- ② Autowiring may rise ambiguous Situation towards dependency Injections.
- ③ Autowiring kills the readability of Configuration file and Spring application.

We can make Spring Container performing autowiring in 4 modes.

① byName (autowire = "byName")

→ performs setter injection based on bean properties and dependent bean ids names matching.

② byType (autowire = "byType")

→ performs setter injection based on bean properties and dependent bean ids type matching

③ constructor (autowire = "constructor")

→ performs constructor injection by using parameterized constructors.

④ autodetect (autowire = "autodetect")

→ performs either constructor injection or setter injection

→ If Bean class contains ^{compiler generated (default constructor)} 0-param constructor then it performs

otherwise it performs constructor injection

byType mode
setter injection

Example application :-

Demo.java → Same as previous appn (Spring interface)

DemoBean.java (Spring Bean)

TestBean.java (Dependent Bean)

DemoCfg.xml (Spring Cfg file)

DemoClient.java (Client appn)

// Demo.java

Same as previous

// DemoBean.java

public class DemoBean implements Demo

{

// bean property

TestBean tb;

// 0-param constructor

DemoBean()

{

S.O.P ("DemoBean: 0-param constructor");

}

// 1-Param Constructor

Shift+Tab+Space
Ctrl+Shift+i

DemoBean (TestBean tb)

{ this.tb = tb;

S.O.P ("DemoBean: 1-Param constructor");

}

// method for Setter Injection

public void setTb (TestBean tb)

{

S.O.P ("DemoBean: setTb(-) method");

this.tb = tb;

}

public String sayHello ()

{

return "Good morning TestBean - msg = " + tb;

} }

// TestBean.java

public class TestBean

{

// Bean property

String msg;

// Setter method for Setter Injection

public void setMsg (String msg)

{

this.msg = msg;

}

// Implement toString() method

public String toString()

{

return msg;

} }

// DemoConfig.xml

<beans>

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

<property name="msg" value="hello"/> → Explicit wiring

<bean>

<bean id="db" class="DemoBean" autowire="byName" />

</beans>

↳ for this the DemoBean class Beanproperty

(tb)

and TestBean class Beanid (tb) must match

//DemoClient.java

public class DemoClient {

 public void main(String[] args) {

 //

 //Activate Spring Container

 ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("DemoClient.xml");

 //get Spring Bean obj

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

 //call b.method

 S.O.P(bob.sayHello());

}

}

→ The Beanid in spring configuration file must be unique id. otherwise SpringContainer may raise ambiguity problem.

Autowire = "byType" coding SpringCfg File :-

//in DemoCfg.xml

<!DOCTYPE ... >

<beans>

<bean id="t1" class="TestBean">

 <property name="msg" value="hello" />

</bean>

<bean id="db" class="DemoBean" autowire="byType" />

↳ here the Beanproperty "tb" type TestBean and the DependentBean obj "t," type (TestBean) must match

</beans>

Note:- When you Configure Same TestBean class for multiple times with different

Beanid's then byType mode of autowiring may raise ambiguity problems

autowire = "constructor" Coding SpringCfg File :-

//in DemoCfg.xml

<!DOCTYPE ... >

<beans>

```

<bean id="t1" class="TestBean">
    <property name="msg" value="hello"/>
</bean>

```

```

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

```

</beans>

↳ for this springBean class must have 1-param
Constructor having TestBean has the param type.

autowire = "autodetect" coding Spring Configuration file :-

<beans>

```

<bean id="t1" class="TestBean">

```

```

    <property name="msg" value="hello"/>

```

</bean>

```

<bean id="db" class="DemoBean" autowire="autodetect" />

```

</beans>

- (*) If Demo Bean does not contain any explicit constructor then Spring Container performs "byType" mode setterInjection otherwise SpringContainer performs "Constructor Injection".

We can configure certain properties for explicit wiring and certain other properties for autowiring in one SpringBean class.

Q:- What happens if we configured both autowiring & explicit wiring on one Bean property.

Ans:- If both autowiring & explicit wiring performs setter injection then the explicit wiring will effect (applied).

If both autowiring & explicit wiring performs constructor injection then the explicit wiring will effect.

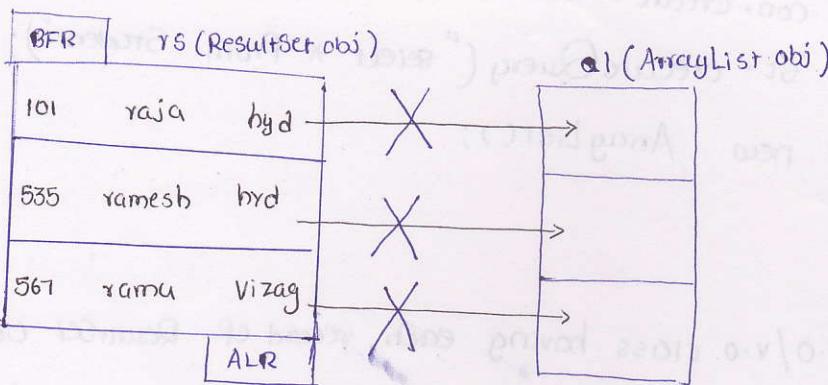
If ~~both~~ autowiring performs setter injection and explicit wiring performs constructor injection ~~then~~ or vice versa then the wiring that performs Setter Injection will be effected.

Both containers (BeanFactory Container & Application Context Container) supports autowiring.

- while using Spring with other Java technologies it is recommended to place the logic of activating Spring container and getting Spring Bean class object in one time execution block called constructor (or) static block
- ~~initially~~ overboast It is recommended to call Business method (or) Spring Bean class object from respectively executing block.

<u>Client App</u>	Place to activate Spring Container and to get Spring Bean class obj	Place to call B-methods of spring Bean
AWT/ swing Frame	Constructor / static block	Event handling methods
Applet/ JApplet	Constructor / static block / init()	Same
Servlet prg	Constructor / static block / init()	service(-,-) / doXxx(-)
JSP prg	<%! public void jspInit() { -----%> ----- %>	<% ----- ----- %>
Struts App	constructor / static block of Struts Action class	execute(-,-,-) of Struts Action class
JSF APP	Constructor / static block of Managed Bean	b-method of Managed Bean

- If Java webapplication uses springAPI (Third Party API) in the webresource programs then the springAPI related main jar file should be added to CLASSPATH and SpringAPI related main and dependent jar files should be added to WEB-INF/lib folder
 - jar files added to CLASSPATH will be used by java compiler during the compilation of webresource programs (server pgs) to recognize spring api. jar files added WEB-INF\lib folder will be used by servletcontainer/jsp container to recognize and uses spring api
 - ResultSet object is not a Serializable object. so, we can't send ResultSet object over the network. To solve this problem there are two solutions
- Solution(1) :- use Rowsets instead of ResultSet
- Rowsets are Serializable objects and we can send them over the network.
 - Every few JDBC drivers supports Rowsets
- Solution(2) :- copy the records of ResultSet obj to CollectionFramework DataStructure
- All collection framework DataStructures (like ArrayList and etc...) are Serializable objects. so we can send them over the network.
- Solution(2) is good
- If you are looking to perform only Read operation then prefer working with Non-Synchronized Data Structures for Better performance. (Eg:- ArrayList, HashMap and etc...)
 - If you are looking to perform both Read and write operation Simultaneously then prefer to working with Synchronized Datastructure for ThreadSafety. (Eg:- Vector, Hashtable and etc...)
 - Even though network is not there it is not recommended to send ResultSet object from DestinationLayer to SourceLayer directly. It is recommended to keep the records of ResultSet in CollectionFramework Datastructure and send that DataStructure to Destination Layer.



Each element of ArrayList allows only one object at a time. Each record of

ResultSet contains multiple values including multiple objects. So, we can't copy each record of ResultSet to each element of ArrayList directly.

To solve the above problem keep each record values in UserDefined Java class object and add that object to each element of ArrayList as shown below

here, this user defined Java class is generally a JavaBean calling

and it is called as V.O.class / D.T.O class
(Value Object class) (DataTransfer Object class)

Logic to Transfer ResultSet obj records to ArrayList elements using V.O.class/D.T.O class

StudentBean (D.T.O class/V.O class)

```
public class StudentBean implements java.io.Serializable
{
    //bean properties
    int sno;
    String sname;
    String scadd;

    // write setters & getters
}
```

NOTE:- Objects added to the elements of Collection Framework DataStructure must be taken as Serializable objects in order to send Collection Framework DataStructure over the network.

Logic to copy the records of ResultSet obj to ArrayList

Statement st = con.createStatement();
ResultSet rs = st.executeQuery("select * from Student");
ArrayList al = new ArrayList();
while (rs.next())
{

// create obj for D.T.O / V.O class having each record of ResultSet obj

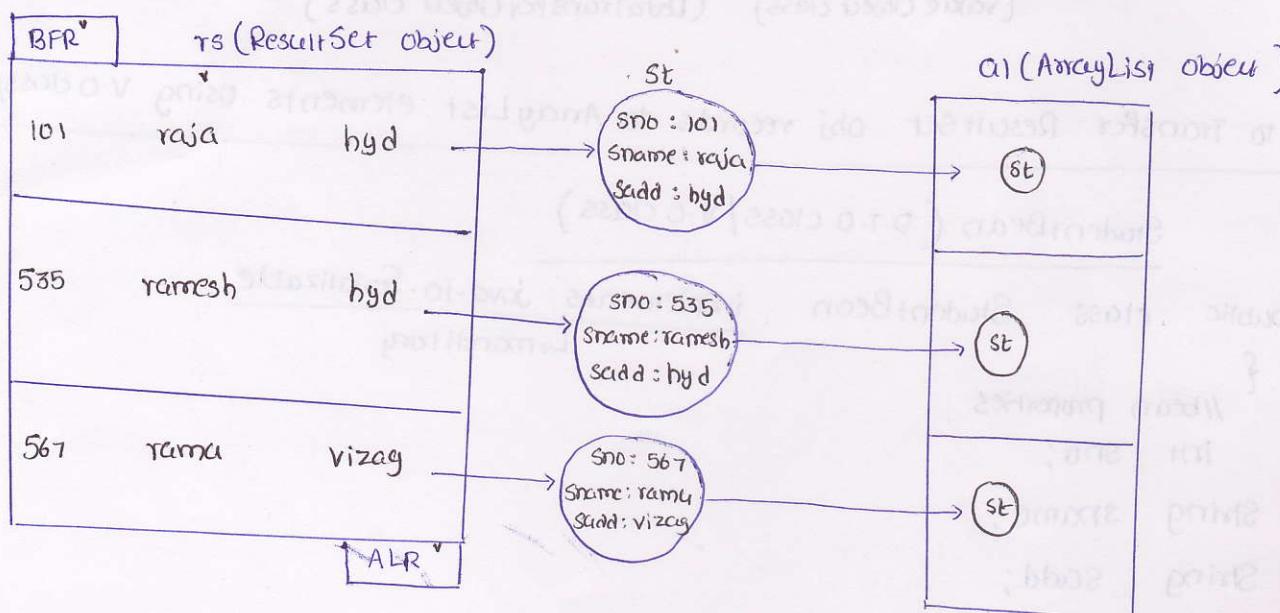
StudentBean st = new StudentBean();

st.setSno(rs.getInt(1));
st.setSname(rs.getString(2));
st.setScadd(rs.getString(3));

// add each D.T.O class / V.O class obj to the elements of ArrayList

al.add(st);

} // while



→ Since, 'st' objects are representing single big value by combining multiple individual values. So, it is called Value Object and its class is called V.O. class.

Each record of Resultset Data is transferring from Resultset to ArrayList in blo^{1st} this record is storing one single big value object. This object is transfer the Data into ArrayList. This "st" object is called Data Transfer Object and its class is called D.T.O. class.

Q:- Where did you use CollectionFW DataStructures in your project?

- Ans:-
- ① To send ResultSet object records over the network as discussed above
 - ② while gathering JDBC properties from properties file we need `java.util.Properties`
 - ③ To maintain JNDI properties
 - ④ To maintain Mail properties
 - ⑤ In SessionTracking to maintain huge amount of Data as SingleSession attribute value.
 - ⑥ While developing gaming appns to maintain huge amount of Temporay data.
 - ⑦ To maintain huge amount of data with out DB SLOS
 - ⑧ To construct buffer and cache.

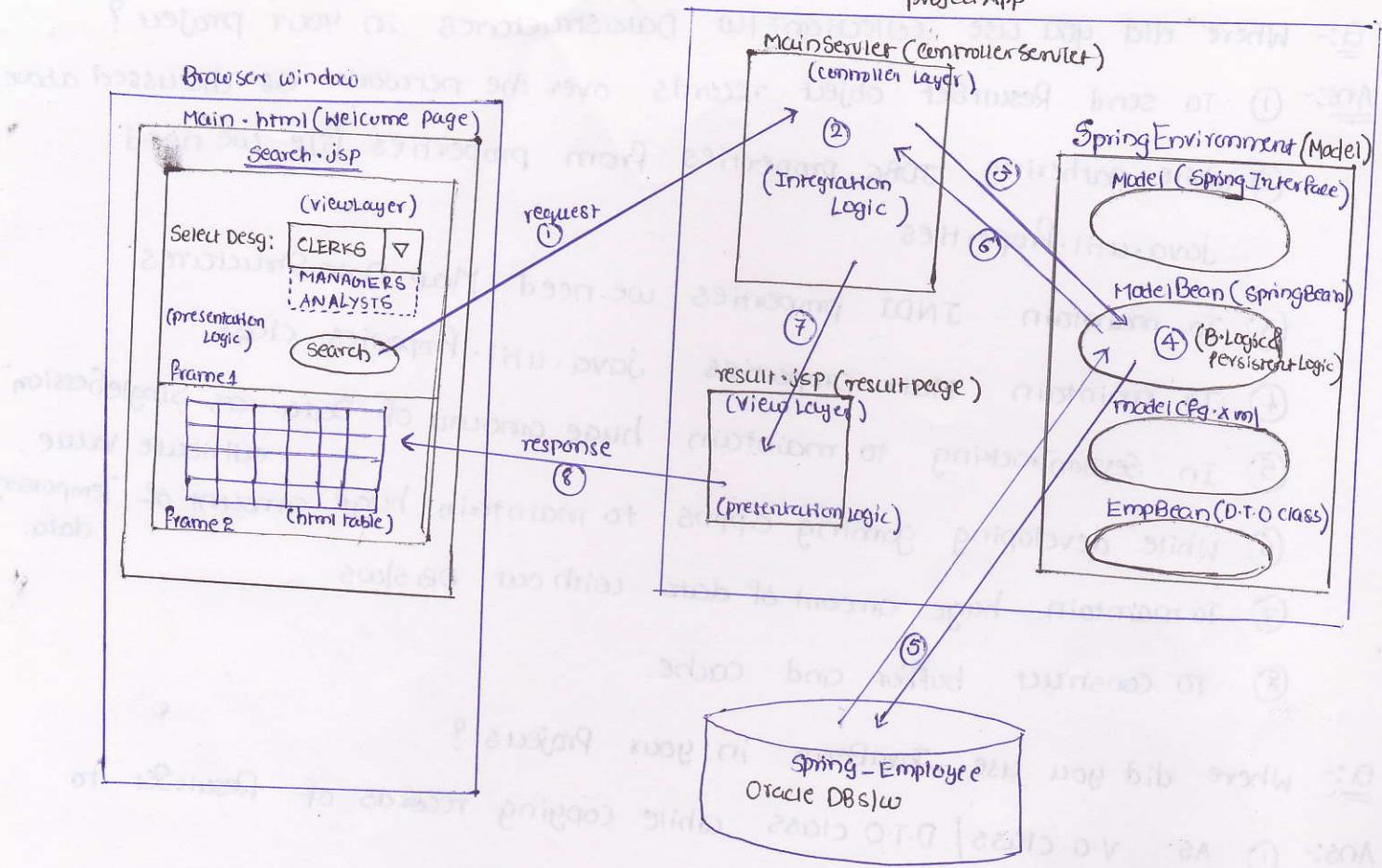
Q:- Where did you use JavaBean in your Projects?

Ans:- ① As V.O.class / D.T.O class while copying records of ResultSet to Collection Framework DataStructure

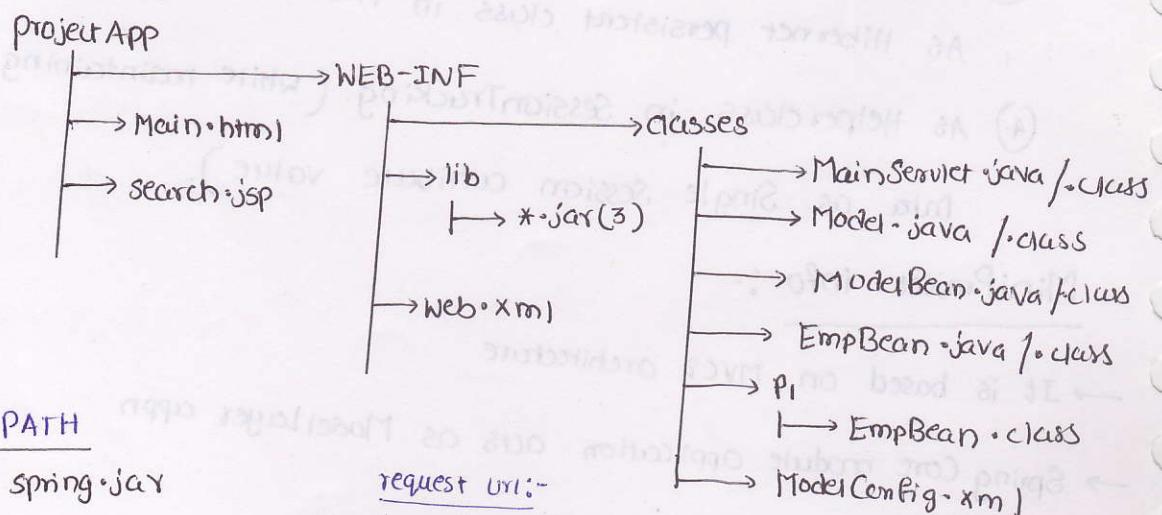
- ② As Model Layer Resource while Developing MVC1 & MVC2 architecture based projects
- ③ As springBean in Spring APPNs
- ④ As Helper class in SessionTracking. (While maintaining huge amount of data as Single session attribute value).

MiniProject info :-

- It is based on MVC2 architecture
- Spring Core module application acts as Model Layer APPN
- Jsp programs acts as View Layer resources
- Servlet program acts as Controller Layer resource



- Servlet program passes data to Result.jsp using request attributes.
- The SpringBean class (ModelBean) copies the records ResultSet object to ArrayList elements by using EmpBean class objects.



- To use Java class in JSP^{program}, it must be placed in the package created in the same folder of jsp (or) in the user-defined package of WEB-INF / classes folder. That means JSP can't use Java class that is there directly in WEB-INF / classes folder.
- For the source code of above diagram refer appn (4) of the page no: (43) to (47)

Understanding Annotations :-

Annotations are java statements and they are alternate for the XML file based

Metadata, Resource Configuration Operations.

Syntax:- $\text{@} \langle \text{Annotation_name} \rangle (\text{param1=val1, param2=val2, ...})$

| It is like XML tag name

| Like XML tag attributes

two types of annotations

① Annotations for Documentation

→ useful while generating Java API documentation

 @Params @Since @Version @Author @Inheritance and etc...

→ introduced from JDK 1.1

② Annotations for Programming

→ useful for metadata, resource config operations

 @Override @Autowired @Resource @Component and etc...

→ introduced from JDK 1.5

Metadata means data about data.

Resource config is also some sort of meta data operation.

→ XML files give good flexibility of modification but Bad performance. bcz

XML parsers are heavy weight slows to process XML documents.

→ Annotations give Bad flexibility of modification but good performance.

→ Annotations based programming. Jdk 1.5+ are supporting

Annotations based programming.

Eg:- Spring 2.5+, Hibernate 3.2+, Struts 2.x, EJB 3.x, Servlet 3.x and etc...

→ we can apply Annotations at 3 level

① Resource Level (on class/interface)

② Method Level (on java method)

③ Field Level (on Member Variable)

To Configure Spring Bean we can use the following annotations :-

07/03/2013

a) @Component

→ makes java class as Spring Bean and auto detectable from classpath
for dependency injection

b) @Service

→ same as @Component but recommended to use for Service Layer Spring beans.
(containing Business Logic)

c) @Repository

→ same as @Component but recommended to use for DAO's type Spring Beans
(containing persistence logic)

To perform dependency injection cfgs

@Value → to inject Simple values, array values, List values

@Resource → to inject values to reference type bean properties

@Autowired → to enable autowiring on bean properties

Note:- Still annotations not ready to cfg dependent values for Map, set, Properties
type bean properties.

* In spring environment we generally use both XML files and Annotations

together to perform Dependency Injections.

→ For Spring 2.5 based Autowiring refer appn(1) of the Supplementary handout
given on 07/03/2013

→ If @Autowired annotation is placed on the top of setXX(-) then it is
go for setter Injection based autowiring

If @Autowired annotation is placed on the top of param Constructor then it is
go for Constructor Injection based autowiring

If @Autowired annotation is placed on the top of Field then it goes for
setter Injection based autowiring.

```

1 Spring 2.5 Application on Annotations (IOC) (Application1)
2 =====
3 -----Test.java-----
4 // Test.java (Spring Interface)
5 public interface Test
6 {
7     public String sayHello();
8 }
9 -----TestBean.java-----
10 //TestBean.java
11 import org.springframework.stereotype.*; → @Service annotation
12 import org.springframework.beans.factory.annotation.*; → @Autowired annotation
13 import java.util.*; → for Date
14
15 @Service // We can also place @Component (makes the Java class as Spring Bean)
16 public class TestBean implements Test
17 {
18     // Bean property
19     Date d; → reference type Bean property.
20
21     @Autowired //to perform auto wiring(by type) on bean property
22     public void setD(Date d)
23     {
24         this.d=d;
25     }
26
27     public String sayHello()
28     {
29         return "good morning "+d.toString();
30     }
31 }
32 -----DemoCfg.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     xmlns:context="http://www.springframework.org/schema/context"
37     xsi:schemaLocation="http://www.springframework.org/schema/beans
38         http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
39         http://www.springframework.org/schema/aop
40         http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
41         http://www.springframework.org/schema/context
42         http://www.springframework.org/schema/context/spring-context-2.5.xsd">
43
44     <context:annotation-config /> → makes the spring Container to recognize and use annotations
45     <!--<context:component-scan base-package=". "/>-->
46
47     <bean id="tb" class="TestBean"/>
48     <bean id="dt" class="java.util.Date"/>
49
50 </beans>
51 -----TestClient.java-----
52 // TestClient.java
53
54 import org.springframework.context.support.*;
55
56 public class TestClient
57 {
58     public static void main(String[] args)
59     {
60         FileSystemXmlApplicationContext ctx=new FileSystemXmlApplicationContext("DemoCfg.xml");
61         Test beanobj=(Test)ctx.getBean("tb");
62         System.out.println("result is "+beanobj.sayHello());
63
64     } //main
65 } //class
66 =====
67 Spring 3.x Application on Annotations(IOC)
68 -----TestInter.java-----
    (Spring Interface)

```

Since we can't add @Service, @Component annotation on the top of predefined Java.util.Date class. So, we must configure that class in springcfg file as Spring Bean

Jar files in CLASSPATH :- spring.jar, commons-logging.jar
(Spring 2.5)

```

69 package p1;
70
71 public interface TestInter
72 {
73     public String sayHello();
74 }
75 -----TestBean.java-----
76 package p1;
77 import java.util.Date;
78 import java.util.List;
79 import javax.annotation.Resource;
80 import javax.annotation.Resources;
81 import org.springframework.beans.factory.annotation.Value;
82 import org.springframework.stereotype.Component;
83 import org.springframework.stereotype.Service;
84
85 @Component("tb") → Bean id
86 public class TestBean implements TestInter
87 {
88     //Bean Property
89     UserBean u1; → reference Type Bean property
90
91     @Value("10") → 91-92 injects Value '10' to property "no".
92     int no; → Simple Bean property
93
94     Date d1; → performs Dependency Injection reference type Bean property
95     → Spring Expression Language gives List Datastructure from array [This List Datastructure will be inject to nickname]
96     @Value("#{T(java.util.Arrays).asList('India','Bharat','Hindustan')}")
97     String nicknames[]; → Array type property
98     → Template Contra → gives List Datastructure from array [This List Datastructure will be injected to colors]
99     @Value("#{T(java.util.Arrays).asList('Red','Blue','Green')}")
100    List colors; → java.util.List type
101
102    → performs Dependency Injection reference type Bean property
103    @Resource(name="dfb") → refer line no: 151
104    public void setD1(Date d)
105    {
106        d1=d;
107    }
108
109    @Resource(name="ub") → refer line no: 135
110    public void setU1(UserBean u1)
111    {
112        this.u1=u1;
113    }
114
115
116    public String sayHello()
117    {
118        System.out.println("no"+no);
119        System.out.println("d1"+d1.toString());
120        System.out.println("nicknames");
121
122        for(int i=0;i<nicknames.length;i++)
123            System.out.println(nicknames[i]);
124
125        System.out.println("Colors="+colors.toString());
126        System.out.println("u1="+u1);
127        return "Good Morning";
128    }
129 }
130 -----UserBean.java-----
131 package p1;
132
133 import org.springframework.stereotype.*;
134
135 @Service("ub") → We can also use @Component annotation.
136 public class UserBean

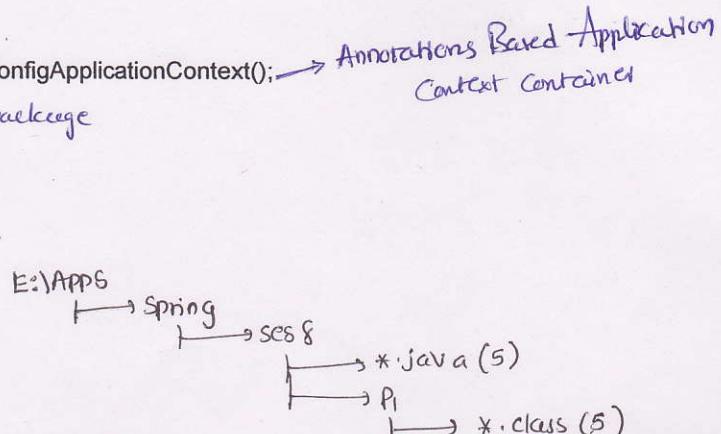
```

```

137 {
138     String msg="Hello World";
139
140     public String toString()
141     {
142         return "DemoBean.msg"+msg;
143     }
144 }
145 -----DateFactoryBean.java-----
146 package p1;
147
148 import org.springframework.beans.factory.FactoryBean;
149 import org.springframework.stereotype.Component;
150
151 @Component("dfb")
152 public class DateFactoryBean implements FactoryBean
153 {
154     public Object getObject() throws Exception {
155         return new java.util.Date(); → resultant object
156     }
157     public Class getObjectType() {
158         return java.util.Date.class;
159     }
160     public boolean isSingleton() {
161         return false;
162     }
163 }
164 }
165 -----TestClient.java-----
166 package p1;
167
168 import org.springframework.context.annotation.AnnotationConfigApplicationContext;
169 import org.springframework.context.support.*;
170
171 public class TestClient
172 {
173     public static void main(String s[])
174     {
175         AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(); → Annotations Based Application
176             ctx.scan("p1"); //p1 is the package name
177             ctx.refresh(); → scans the every class of P1 package
178             TestInter test=(TestInter)ctx.getBean("tb");
179             System.out.println("result is"+test.sayHello());
180
181     } } → To make container loading the classes.
182 }
```

UserDefined Factory Bean class returning
java.util.Date class object.

E:\Apps\Spring\ses8>javac -d . *.java
java P1.TestClient



Jar files in CLASSPATH

- 1) commons-logging.jar (Collect from Spring 2.5)
- 2) org.sf.asm-3.1.0.RC1.jar
- 3) org.sf.beans-3.1.0.RC1.jar
- 4) org.sf.context-3.1.0.RC1.jar
- 5) org.sf.context.support-3.1.0.RC1.jar
- 6) org.sf.core-3.1.0.RC1.jar
- 7) org.sf.expression → 3.1.0.RC1.jar

import
General
Existing Project in WorkB

Folder:

→ For Spring 3.x annotations based Dependency Injection application based without using any XML files refer appn(2) of march 07th handout

→ Opening already creating project in MyECLIPSE IDE

File menu → import → General → Existing projects in workspace

→ Select directory → Finish

NOTE:-

While working with AnnotationBased Dependency Injection, if Dependency Injection Configurations are done directly on BeanProperties then there is no necessity of writing setter (-) methods, Constructors related to Injections.

08/03/2013

More annotations of core module:-

@Inject → to configure explicit dependency Injection

@PostConstruct → to configure custom init method

@PreDestroy → to configure custom destroy method

@Scope → to specify SpringBean scope (single, prototype)

@ComponentScan → to recognize SpringBeans and resources that are there in packages Spring Configuration file.

*) for Spring 3.0 based Explicit Constructor injection through Annotations refer appn(1) of supplementary handout given on 08/03/13

handout given on 08/03/13 it uses properties file refer appn(2) of supplementary

*) for spring 3.0 annotations based example application refer appn(3) of the 08/03/13 handout there is

If you want to know what jar files are added to the project goto .CLASSPATH. there is the jar files.

*) for spring 3.0 annotations based example appn custom-init, custom-destroy methods configuration refer appn(3) of the 08/03/13 handout (@PostConstruct, @PreDestroy)

*) for spring 3.0 annotations based FactoryBeanDevelopment and injects it resultant object to another BeanClass property refer appn(4) of the 08/03/13 handout

We can also use `@Inject` annotation to also perform explicit `Setter Injection` in the similar fashion.

3/8/2013 9:46 AM

```

1 >>>>>>>>> Example Applications on Spring Annotations(IOC)>>>>>>>>>
2 App1 (on constructor Injection)
3 -----Demo.java-----
4 public interface Demo
5 {
6     public String sayHello();
7 }
8 -----DemoBean.java-----
9 import javax.inject.Inject; → for @Inject
10 import org.springframework.beans.factory.annotation.Value; → for @Value
11 import org.springframework.stereotype.Component; → for @Component
12 @Component("db") → bean id
13 public class DemoBean implements Demo
14 { int age;
15     String name; } BeanProperties
16     float avg;
17     @Inject
18     public DemoBean(@Value("30")int age, @Value("Raja")String name, @Value("50.67")float avg) {
19         this.age = age;
20         this.name = name;
21         this.avg = avg;
22     }
23     public String sayHello()
24     {
25         return "Welcome to Spring age= "+age+" name= "+name+" avg= "+avg;
26     }
27 }

```

here `@Value` annotation injects the values to parameters and these parameter values are assigned to Bean properties.

```

28 -----DemoClient.java-----
29 import org.springframework.context.ApplicationContext;
30 import org.springframework.context.annotation.AnnotationConfigApplicationContext;
31 import org.springframework.context.support.ClassPathXmlApplicationContext;
32
33 public class DemoClient {
34     public static void main(String s[]) throws Exception {
35         ApplicationContext ctx = new AnnotationConfigApplicationContext(DemoBean.class); → Activates the annotations based
36         // ask spring container to give springbean class object
37         Demo d1=(Demo)ctx.getBean("db"); → line no : 12
38         // calls B.method of SpringBean class
39         String result = d1.sayHello();
40         System.out.println(result);
41     }
42 }
43

```

spring container.

44 App2 (Reading values from properties file) (3.x environment)

45 -----App.properties-----

JAR files in CLASSPATH :- Same as appn①

46 jdbc.driver=oracle.jdbc.driver.OracleDriver

47 jdbc.url=jdbc:oracle:thin:@localhost:1521:xe

48 db.user=noc1

49 db.pwd=noc1

50 -----select.java-----

51 package p1;

52 public interface Select

53 {

54 public void readDBDetails();

55 }

56 -----SelectBean.java-----

57 package p1;

58 import org.springframework.beans.factory.annotation.Value;

59 import org.springframework.context.annotation.ImportResource;

60 import org.springframework.stereotype.Component;

61 → To also use the configurations done in SpringCfg file (Bcz properties file is configured there)

62 @ImportResource("classpath:/SpringCfg.xml")

63 @Component("sb") → Bean id

64 public class SelectBean implements Select

65 { private @Value("\${db.user}")

66 String uname; → key in the properties file (refer line no: 50)

67 - 68

Page 1 of 4

Injects value collected from "db.user" key of properties file to "uname" property.

```

69     private @Value("${db.pwd}")
70     String pwd;           ↳ refer lineno: 51
71     @Override
72     public void readDBDetails() {
73         System.out.println("userName="+uname); } B.method
74         System.out.println("Password="+pwd); implemented
75     }
76 }
77 -----Springcfg.xml-----
78 <?xml version="1.0" encoding="UTF-8"?>
79 <beans xmlns="http://www.springframework.org/schema/beans"
80   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
81   xmlns:context="http://www.springframework.org/schema/context"
82   xsi:schemaLocation="http://cxf.apache.org/core http://cxf.apache.org/schemas/core.xsd
83   http://www.springframework.org/schema/beans
84   http://www.springframework.org/schema/beans/spring-beans.xsd
85   http://www.springframework.org/schema/context
86   http://www.springframework.org/schema/context/spring-context.xsd">
87   <context:annotation-config/> → instructs to recognize annotations
88   <context:component-scan base-package="p1"/> → makes the container to look for the resources in 'p1' package
89   <context:property-placeholder location=".//App.properties"/> → recognizes the properties file of current directory (.)
90   <!-- <context:property-placeholder location="classpath://App.properties"/> -->
91 </beans>
92 -----SelectClient.java-----
93 package p1;
94 import org.springframework.context.support.ClassPathXmlApplicationContext;
95 public class SelectClient {
96 {
97     public static void main(String[] args) {
98     {
99         System.out.println("main():SelectClient");
100        ClassPathXmlApplicationContext ctx=new ClassPathXmlApplicationContext("SpringCfg.xml");
101        Select bobj=(Select)ctx.getBean("sb");
102        bobj.readDBDetails();
103    }
104 }
105 -----
106 App3 (custom init-method and custom-destroy methods cfg)
107 -----Demo.java-----
108 package p1;
109 public interface Demo
110 {
111     public String sayHello();
112 }
113 -----DemoBean.java-----
114 package p1;
115 import javax.annotation.PostConstruct;
116 import javax.annotation.PreDestroy;
117 import org.springframework.beans.factory.annotation.Value;
118 import org.springframework.stereotype.Component;
119
120 @Component("db") ↳ BeanId
121 public class DemoBean implements Demo {
122     @Value("satya")
123     String msg;
124
125     @Value("10")
126     int age;
127     DemoBean()
128 {
129     System.out.println("DemoBean():constructor");
130 }
131 @PostConstruct → makes myInit() method as custom-init() method
132     public void myInit()
133     {
134         System.out.println("Init-Method");
135         if(age<0)
136     }

```

```

137     age=18;
138     System.out.println("Age cannot be -ve,it is set to 18");
139 }
140 } make the myDestroy() method as custom-destroy()
141 @PreDestroy
142 public void myDestroy()
143 {
144     System.out.println("Destroy-method");
145     age=0;
146 }
147
148 public String sayHello() {
149     return "Hello :" + msg + "\n Your Age :" + age;
150 }
151
152 }
-----DemoCfg.xml-----
153 <beans xmlns="http://www.springframework.org/schema/beans"
154   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
155   xmlns:context="http://www.springframework.org/schema/context"
156   xsi:schemaLocation="http://www.springframework.org/schema/beans
157   http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
158   http://www.springframework.org/schema/context
159   http://www.springframework.org/schema/context/spring-context-2.5.xsd">
160   <context:component-scan base-package="p1" />
161 </beans>
-----DemoClient.java-----
163 package p1;
164 import org.springframework.context.support.ClassPathXmlApplicationContext;
165 public class DemoClient {
166     public static void main(String s[]) throws Exception {
167         ClassPathXmlApplicationContext ctx=new ClassPathXmlApplicationContext("DemoCfg.xml");
168         // ask spring container to give springbean class object
169         Demo d1 = (Demo) ctx.getBean("db");
170         // calls B.method of SpringBean class
171         String result = d1.sayHello();
172         System.out.println(result);
173
174         ctx.close();
175     }
176 }
177 }

-----TestBean.java----- Factory Bean generating Date class obj representing the given Date Value)
179 App4 (on Factory Bean )
180 package p1;
181 import java.text.SimpleDateFormat;
182 import java.util.Date;
183 import org.springframework.beans.factory.FactoryBean;
184 import org.springframework.beans.factory.annotation.Value;
185 import org.springframework.context.annotation.Scope;
186 import org.springframework.stereotype.Component;
187
188 @Component("tb") → Bean Id
189 @Scope("prototype") → for every Bean on separate Bean Obj will be created
190
191 public class TestBean implements FactoryBean { mandatory
192
193     @Value("21/12/2012")
194     private String dt;
195
196     public Object getObject()
197     {
198         Date d=null;
199         try{
200             System.out.println(" getObject():TestBean");
201             SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
202             d = sdf.parse(dt);
203             }catch(Exception e)
204             {
}

```

} we can avoid this XML file by using AnnotationConfigApplicationContext Container and `ctx.scan("p1");`
`ctx.refresh();`
 (refer march 7th/08/13 2nd appn)

logic to convert String Date value to Java.util.Date class object

```

205     System.out.println("Exception while Parsing Dt"+e);
206 }
207 return d; → gives java.util.Date class object
208 }
209 public Class getObjectType()
210 {
211     System.out.println("getObjectType():TestBean");
212     return java.util.Date.class;
213 }
214 public boolean isSingleton()
215 {
216     System.out.println("isSingleton():TestBean");
217     return false;
218 }
219 }
220 -----Demo.java-----
221 package p1;
222
223 public interface Demo
224 {
225     public String sayHello();
226 }
227 -----DemoBean.java-----
228 package p1;
229 import java.util.Date;
230 import javax.annotation.Resource;
231 import org.springframework.stereotype.Component;
232
233 @Component("db")
234 public class DemoBean implements Demo
235 {
236     @Resource(name="tb") refer line no : 189
237     Date dt;
238
239     public String sayHello()
240     {
241         return "The time is"+dt;
242     }
243 }
244 -----DemoCfg.xml-----
245 <?xml version="1.0" encoding="UTF-8"?>
246 <beans xmlns="http://www.springframework.org/schema/beans"
247   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
248   xmlns:context="http://www.springframework.org/schema/context"
249   xsi:schemaLocation="http://www.springframework.org/schema/beans
250   http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
251   http://www.springframework.org/schema/context
252   http://www.springframework.org/schema/context/spring-context-2.5.xsd">
253     <context:component-scan base-package="p1" />
254 </beans>
255 -----DemoClient.java-----
256 package p1;
257 import org.springframework.context.support.ClassPathXmlApplicationContext;
258 public class DemoClient
259 {
260     public static void main(String[] args)
261     {
262         System.out.println("Hello");
263         ClassPathXmlApplicationContext ctx=new ClassPathXmlApplicationContext("DemoCfg.xml");
264         Demo bobj=(Demo)ctx.getBean("db");
265         System.out.println(bobj.sayHello());
266     }
267 }

```

JNDI :-

→ To provide global visibility to java objects and their references we keep them in registry slo. Every web server / Application Server gives an registry slo

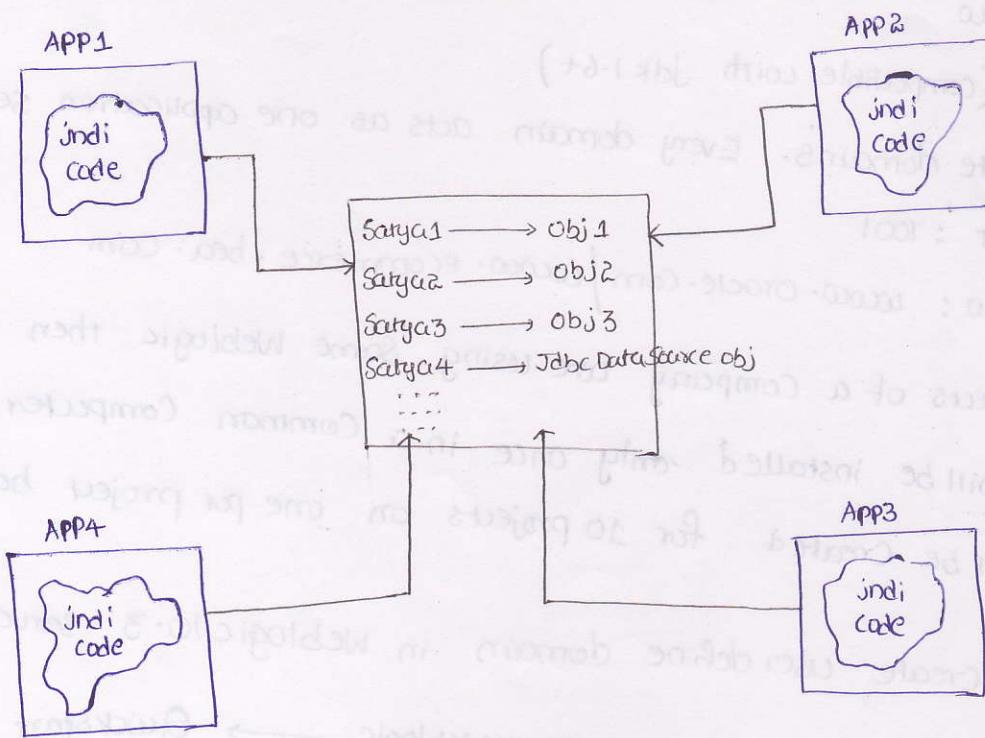
Weblogic gives Weblogic registry

Tomcat gives Tomcat Registry

GlassFish gives GlassFish registry

JBOSS gives Jnp registry and etc...

Java applications use jdbc api (javax.sql, java.sql pkgs) to interact with DBs
 Java applications use jndi api (javax.naming and its sub pkgs) to interact with Registry
 JDBC, JNDI API are part of JSE module (jdk slo)



→ There are two types of registry slo's

i) Naming Registry slo

→ Maintains info as key values pairs

→ we must know keys to get the values

Eg:- Telephone book (real life)

Rmi registry

Cos registry

and etc...

2) directory registry

- Maintains info as key-value pairs and values can extra properties/attribute
- we can use either keys or extra properties to get the values

Eg:- Yellow pages (category wise it give) }
Windows filesystem (real life)

Weblogic registry

GlassFish registry

DNS registry

and etc...

Weblogic

Type : Application Server software

Vendor : BEA Systems (Oracle Corp)

Commercial software

Version : 10.3 (compatible with jdk 1.6+)

Allows to create domains. Every domain acts as one application server

default port : 7001

To download software : www.oracle.com/cacoco/e-commerce.bea.com

If multiple projects of a company are using same Weblogic then the

Weblogic software will be installed only once in a common computer but

10 domains will be created for 10 projects on one per project basis.

10 domains will be created for 10 projects on one per project basis.

* Procedure to create user define domain in Weblogic 10.3 server.

Step-I:- Start → Programs → OracleWeblogic → Quickstart →

① Getting started with Weblogic server → ② Create a new Weblogic domain

→ ③ Generate a domain configuration... → Next →

Domain Name [MyDomain1] → Next →

User Name javaboss

User password javaboss1

Conform Password javaboss1

→ Next → Administration Services

→ Next → Listen port: 7171 → Next → Create

procedure to see the weblogic registry of Weblogic Server

Step I:- Start the above MyDomain1 server

Start → programs → OracleWeblogic → user projects → My Domain1

→ start admin server for weblogic

Step II:- Open admin Console of the above Domain Server

open Browser window → http://localhost:7171/console

Username: javaboss

Password: javaboss1 → Login

Step III:- View the weblogic registry

Admin Console → Environment → servers → AdminServer → View Jndi tree

→ We can perform insert, update, delete, ~~update~~ and select jdbc operations on DB table by using jdbc code.

→ We can perform bind, rebind, unbind, lookup and list operations on Jndi registry by using jndi code.

bind → keeping obj with nickname / alias name / jndi name in registry

unbind → removing object from registry

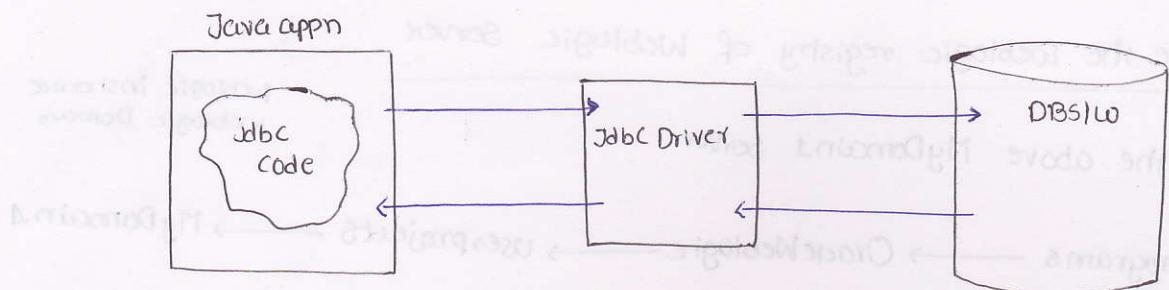
lookup → gathering object from registry with nickname / alias name

list → gathering all bindings from registry (nicknames, objs)

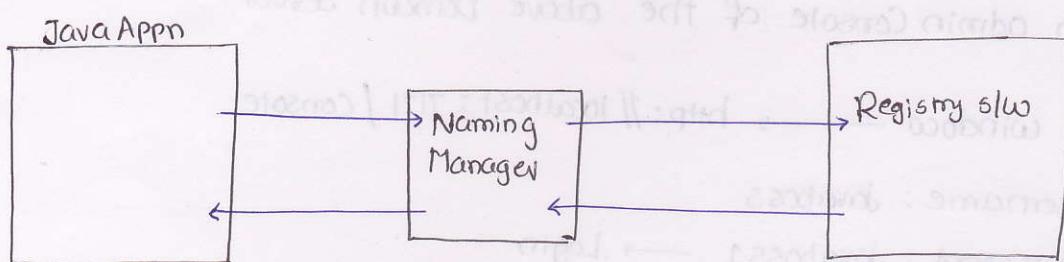
rebind → replacing existing object with new object

→ jdbc Connection object represents Connectivity b/w java appn and DB s/o.

To create this Con object we need jdbc properties (Driver class name, url, dbuser, db pwd) and these jdbc properties will change based on the jdbc driver, db s/o we use



JDBC Driver is the bridge b/w Java appn and DB s/w.



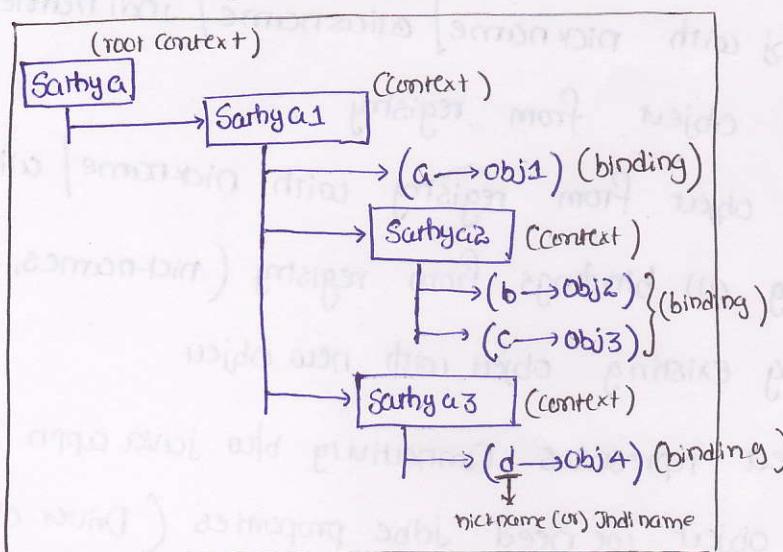
Naming Manager is the bridge b/w Java appn and registry s/w.

→ InitialContext Obj represents connectivity b/w java appn and registry s/w
To create this InitialContext Obj we need jndi properties they are

- 1) InitialContextFactory class
- 2) Provider url

these jndi properties will change based on the registry s/w we use.

Understanding Jndi Tree



Sathyas1 is subcontext of Sathyas context and it is parent context of Sathyas2,

Sathyas3

NOTE:- In real time projects the Datasource object that represents JDBC Connection pool will be placed in Registry slo having nickname (or) alias name for global visibility.

Jndi properties of Weblogic Registry

InitialContext class name: weblogic.jndi.WLInitialContextFactory (Weblogic.jar)
provider url : t3://<hostname/ IPaddress>:<port no>

Eg:- t3://localhost:7171

* Write a Jndi appn to establish connection b/w Java appn and Weblogic registry

//JndiConnTest.java

```
import javax.naming.*; //jndi api  
import java.util.*;  
public JndiConnTest
```

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

```
{
```

//Prepare Jndi properties

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

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

↳ Fixed key

↓
InitialContextFactory class name

```
ht.put(Context.PROVIDER_URL, "t3://localhost:7171");
```

// Create Initial Context object

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

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

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

```
else
```

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

```
} //main
```

```
} //class
```

```
//add Weblogic.jar (<Weblogic_home>/middleware/wlserver_10.3/server/lib/weblogic.jar)  
in class path
```

javax.naming.InitialContext class gives the following method to perform jndi operations

bind (-,-)

unbind (-)

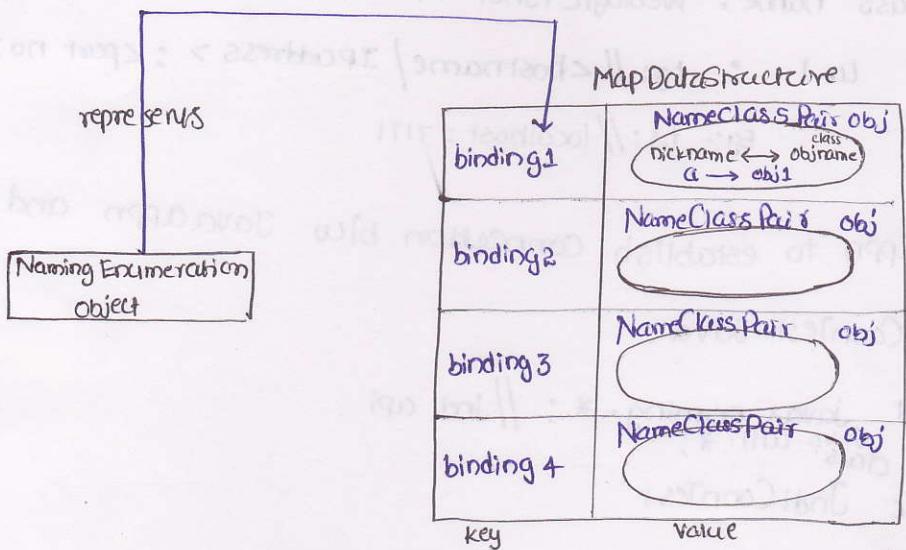
rebind (-,-)

list (-)

lookup(-)

11/03/2013

jndi API is
related to JSE module
API.



Each NameClassPair class obj represents one binding (nickname, obj).
javax.naming.NamingEnumeration(I) is sub interface of java.util.Enumeration(I).

(*) Example application on JndiList operation.

//JndiList.java

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

public class JndiList
{
    public static void main (String[] args) throws Exception
    {
        //Prepare Jndi properties
        Hashtable<String, String> ht = new Hashtable<String, String>();
        ht.put (context. INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFactory");
        ht.put (context. PROVIDER_URL, "t3://localhost:7171");
        ht.put (context. SECURITY_PRINCIPAL, "javaboss");
        ht.put (context. SECURITY_CREDENTIALS, "javaboss1"); } }
```

Optional
while working
with root context

Initial Context ic = new InitialContext (ht); //points root context by default

```

    // performing list operation
    // NamingEnumeration ne = ic.list(""); // gives all the binding of root context
    NamingEnumeration ne = ic.list("javax/jms"); // gives all bindings of javax/jms context

    while (ne.hasMore())
    {
        NameClassPair ncp = (NameClassPair) ne.nextElement(); // gives one binding at a time
        S.O.P(ncp.getName() + " " + ncp.getClassName());
    }
}
}

```

↑ represents root context
↑ represents the binding of root context
↑ gives nickname
↑ gives class name of the object

* For example application on Jndi operations refer TestJndi.java, TestJndi1.java of page no's 57 to 59

GlassFish

type: Application Server s/w

vendor: SunMS (Oracle Corp)

Version: 2.x (compatible with jdk1.5+)

openSource

allows to create domains. default domain is domain1

default port no's : To access admin console 4343

To access webapplication: 8080

GlassFish s/w that comes with NetBeans IDE s/w installation can be used with or without NetBeans IDE.

Procedure to create User Defined Domain in GlassFish.

D:\sun\appserver\bin> asadmin create-domain --adminport=4845 --user=testuser mydomain2

D:\sun\appserver\bin> asadmin create-domain --adminport=4846 --user=testuser mydomain2

Procedure to see GlassFish registry of mydomain2 server

Step-I:- Start mydomain2 server

D:\sun\AppServer\bin > asadmin start-domain mydomain2

Step-II:- Open the admin console of mydomain2 server.

Open Browser window → Type the following url

http://localhost:4846

Username: testuser

Password: testuser → Login

Step-III:- give the glassFish registry.

Admin Console → Application Server → Jndi Broosing →

* GlassFish server gives GlassFish registry as the built-in registry S10.

Jndi properties of GlassFish registry

InitialContextFactory class name : com.sun.enterprise.naming.SerialInitContextFactory
provider url : iiop://<host>:<port no>
eg: iiop://localhost:4845

(*) This class is available in Appserv-rt.jar and this jar file having multiple dependent jar files. They are

appserv-admin.jar
appserv-ee.jar
appserv-ext.jar
javaee.jar

available in D:\sun\AppServer\lib folder

imqjmsra.jar → D:\sun\AppServer\lib\install\application\jmsra

(*) By changing the JNDI properties of TestJndi.java, TestJndi1.java files

we can perform JNDI operations on other registry S10's (refer page no: 57 to 59)

iiop → inter internet ORB protocol

↳ Object Request Broker

JBOSS

- Type : Application server SW
- Version : 5.x (Compatible with jdk 1.5)
- Vendor : Apache (Redhat)
- gives 5 default domains
 - default (default)
 - Web
 - Standard
 - All
 - minimal

→ to download SW : As zip file from www.apache.org website

Jboss - ~~5.1.0.GA~~ - Jdk 1.5
5.1.0

→ to install SW : extract zip file to a folder

→ default port no : 8080 (to access the webapplications)
1099 (to access the jnp registry)

→ to start the server : use <Jboss-home>\bin\run.bat file

(if problems are raised then remove findstr word from
run.bat file content)

* procedure to see the JNP registry of JBOSS 5.x server.

Step-I :- Start JBOSS server

Step-II :- Open admin console of JBOSS

Open browser window → write following url

http://localhost:8080/jmx-console

Step-III :- give the jnp registered related Jndi tree

admin console → Service = Jndiview → select "list" operation → invoke

→ look at Global Jndi Name Space

the jndi properties of JNP registry :

InitialContextFactory class name : org.jnp.interfaces.NamingContextFactory

Provider url : jnp://<host>:<portno>

Eg : jnp://localhost:1099

available in <jboss-home>/client/jnp-client.jar file

its dependent jar file is <jboss-home>/client/jboss-logging spi.jar

jar files required in classpath to interact with Weblogic registry : Weblogic.jar

jar files required in classpath to interact with GlassFish registry :

appserv-rt.jar

javaee.jar

appserv-admin.jar

appserv-ee.jar

appserv-ext.jar

imq,jmsra.jar

jar files required in classpath to interact with JBOSS registry :

jnp-client.jar

jboss-logging-spi.jar

Spring Jndi :-

- Spring jndi provides abstraction layer on plain Jndi programming by giving `org.springframework.jndi.JndiTemplate` class.
- This `JndiTemplate` class internally uses plain Jndi and Simplify the process of performing Jndi operations on Registry side.

plain jndi programming

- Prepare Jndi properties
- Create Initial Context obj
- perform Jndi operations
- take care of Exception handling
- close Initial Context object

Spring jndi programing

- get `JndiTemplate` class obj

- perform Jndi operations

Note:- the remaining Common operations will be taken care by `JndiTemplate` class.

Note:- Spring Jndi allows to pass Jndi properties from Spring Configuration File gives flexibility of modification.

- Spring Jndi internally uses plainJndi but it never makes programmer to bother about plainJndi (This is nothing but getting abstractionLayer).
- Client application of Spring application is java class. so that it can be taken as Spring Bean and its properties can be configured for Dependency Injection.
- JndiTemplate class gives bind(-,-), unbind(-,-), lookup(-) methods but it is not giving list().

⑧ for example appn on SpringJndi that uses JndiTemplate class refer page No : (59) & (60) appn no : (12)

- Spring Technologies provides abstraction Layer on plain Technologies.
- In this process if Spring API fails to support certain operation then we can use Spring supplied Callback Interfaces to implement that operation by using plain Technology API.
- The interface whose methods implemented in implementation class called by underlying container automatically is called Callback interface.

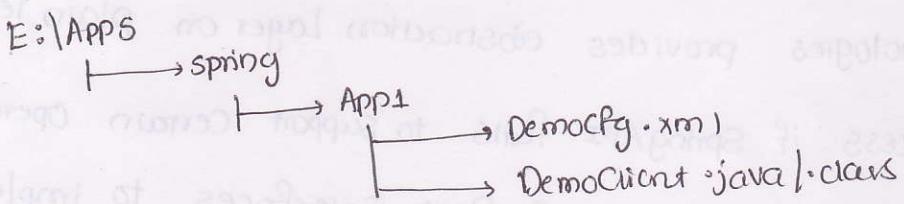
Example Scenario :-

The SpringJndi supplied JndiTemplate class does provide methods to perform list operation on registry. This can be achieved by using JndiCallback interface. JndiCallback (I) gives doInContext (-) method. In this method we can write plain jndi code to perform list operation.

```
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent we)
    {
        System.exit(0)
    } // Method
}); // Method call
```

In the above code:

- a) AddWindowListener() method is called having one object as the argument value.
 - b) → That object is the object of anonymous inner class that extends from WindowAdapter class.
 - c) → This Anonymous Inner class is overriding WindowClosing() method having some logic.
- * Example Application on to perform JndiCallback Interface to implement the list operation functionality.



DemoCfg.xml same as app1 (page no: 59)

// DemoClient.java

```
import org.springframework.stereotype.*;  
import org.springframework.jndi.*;  
import javax.naming.*; ————— plainJndi to work with callback (I)  
  
public class DemoClient  
{  
    static JndiTemplate template;  
  
    // Setter method for setter injection  
    public void setTemplate(JndiTemplate template)  
    {  
        this.template = template;  
    }  
  
    public static void main(String[] args) throws Exception  
    {  
        FileSystemXmiApplicationContext ctx = new FileSystemXmiApplicationContext(  
            "DemoCfg.xml");  
        // Implement list operation... logic
```

```
template.execute(new JndiCallback() {  
    public Object doInContext(Context ctx) throws NamingException  
    {  
        // use plain Jndi api to implement list operation logic  
    }  
});
```

NamingEnumeration ne = ctx.list("");

```
while(ne.hasMore())  
{
```

NameClassPair np = (NameClassPair) ne.next();

S.o.p(np.getName() + " ----- " + np.getClassName());

```
} // while
```

return null;

} // doInContext()

} // inner class

); // execute()

} // main

} // class

//> javac *.java

//> java DemoClient

make sure that weblogic Server is there in running mode.

In spring Environment if ~~the~~ & spring doesn't contain any method

manually then we use Spring JndiCallback Interface to achieve
that method

(*) template.execute() method is called
by keeping Object of anonymous inner
class implementing JndiCallback Interface
This anonymous inner class is defining
doInContext() method having logic to
perform List operation.

In Spring Application we can use 3 types of JDBC Con pools

① Spring's Built-in Connection (Working with Driver Manager DataSource class)

→ It is not recommended to use because it does pool the con objs

② Third party JDBC Con pool

→ Apache DBCP

→ C3P0

(Recommended to use in Standalone Spring Apps (Applications that run outside the server)).

③ Webserver / Application Server Managed JDBC Con pool

(Recommended to use the Spring apps are deployable in the server)

* For example App spring's Built-in Con pool utilization refer Feb 18th appn.

→ Apache DBCP

- Type : Third Party
- jar file : <spring-home>/lib/jakarta-commons/commons-dbcx.jar
- class name : org.apache.commons.dbcp.BasicDataSource

Important properties of BasicDataSource class

driverClassName

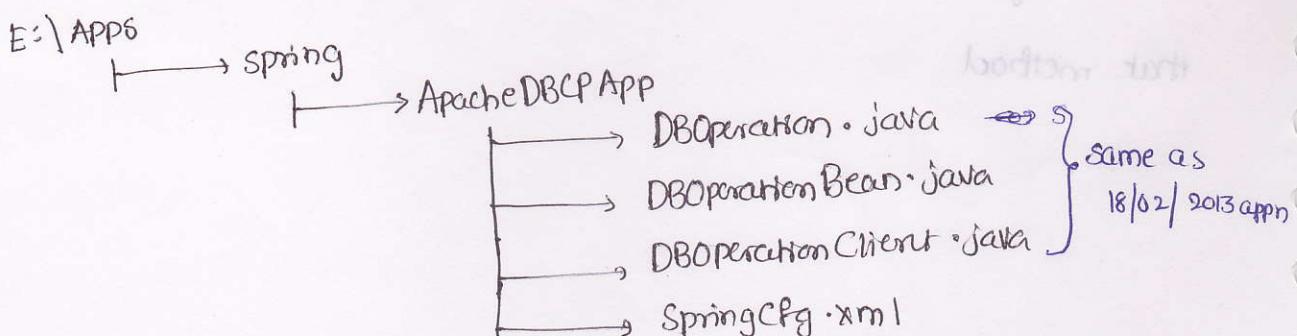
initialSize

password

url

username

(*) Example application



SpringCfg.xml

```
<!DOCTYPE >
<beans>
    <bean id="dbcp" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
        <property name="url" value="jdbc:oracle:thin:@localhost:1521:ORCL"/>
        <property name="username" value="scott"/>
        <property name="password" value="tiger"/>
        <property name="initialSize" value="5"/>
    </bean>
    <bean id="dob" class="DBOperationBean">
        <property name="ds" ref="dbcp"/>
    </bean>
</beans>
```

Jar files in CLASSPATH :- jar files in classpath

- ① commons-dbcP.jar → <spring-home>\lib\jackson-commons folder
- ② spring.jar
- ③ commons-logging.jar

```
> javac *.java
> java DBOperationClient
```

C3P0

type : third party jdbc Con pool slw

Jar file : <spring-home>\lib\C3P0\C3P0-0.9.1.2.jar

Class name that gives

important properties

password

User

minPoolSize

maxPoolSize

jdbc Uri

driverClass

Example application :-

Same as previous appn but perform the following operations.

- ① Add following entries in Spring Cfg.xml file.

"com.mchange.V2c3P0.ComboPooledDataSource"

```
<beans>
    <bean id="dbcp" class="com.mchange.V2c3P0.ComboPooledDataSource">
        <property name="driverClass" value="Oracle.jdbc.driver.OracleDriver"/>
        <property name="jdbcUrl" value="jdbc:oracle:thin:@localhost:1521:ORCL"/>
        <property name="user" value="scott"/>
        <property name="password" value="tiger"/>
        <property name="minPoolSize" value="5"/>
        <property name="maxPoolSize" value="15"/>
    </bean>
    <bean id="dob" class="DBOperation Bean">
        <property name="ds" ref="dbcp"/>
    </bean>
</beans>
```

Add <Spring-home>\lib\c3p0\c3p0-0.9.1.2.jar file to CLASSPATH environment

Variable as additional jar file.

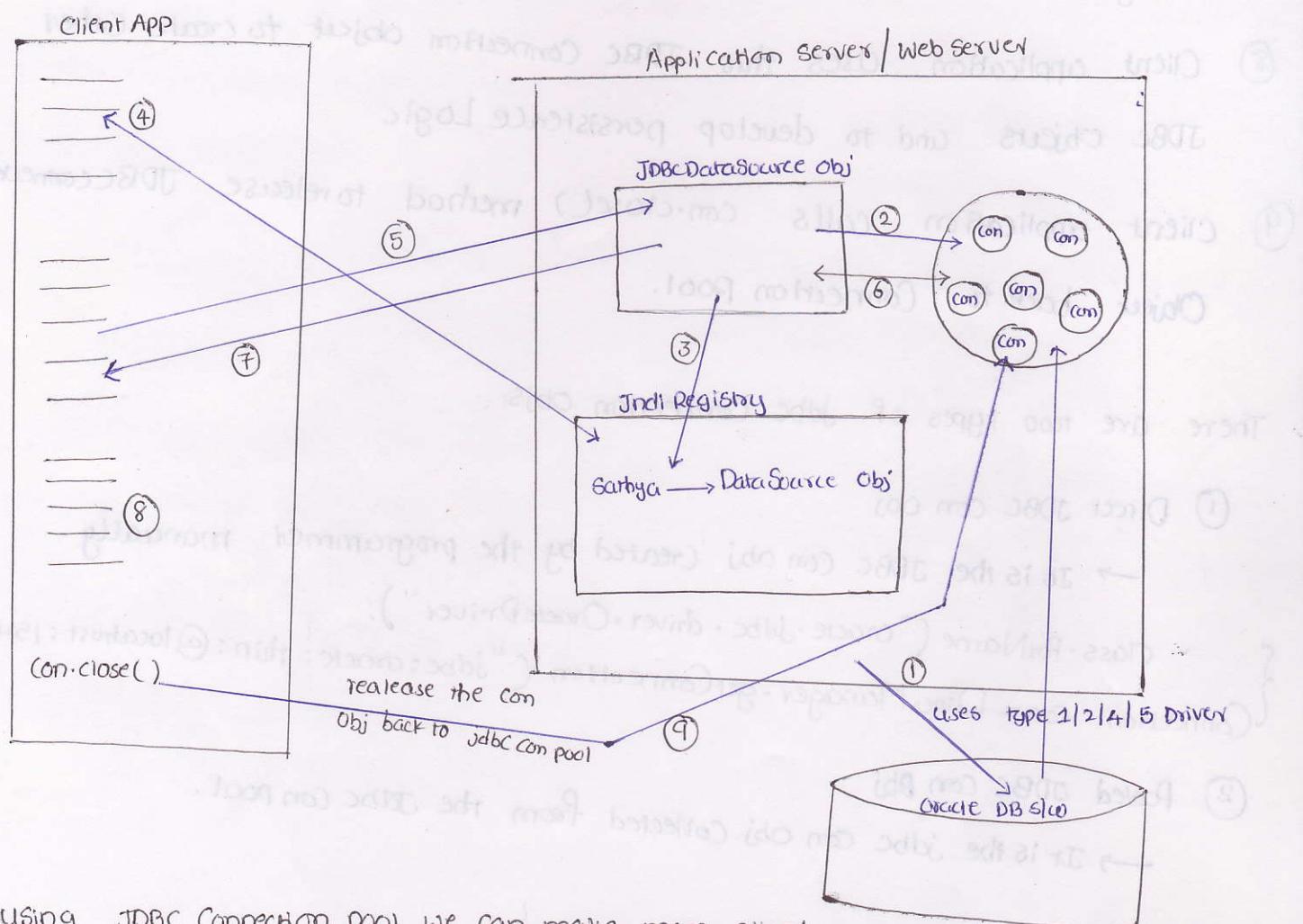
Q:- What is the JDBC connection pool that it have used in your project?

Ans:- If your spring project is standalone (or) Desktop project then use third party managed Connectionpool. Apache DBcp & c3p0 and etc..

If spring project is Deployable webapplication then use Server managed JDBC Connectionpool

Understanding the process of working with Server Managed JDBC Connection Pool

DataSource Object represents jdbc con pool and this DataSource obj will be placed in Registry s/w for global visibility having nickname or alias name.



Using JDBC Connection pool we can make more client applications interacting with DB s/w by using minimum no.of JDBC con obj's.

All Connection Obj's in Connection pool represents connectivity with same DB s/w. Eg:- JDBC Connection pool for Oracle means all Connection obj's in that Connection pool represents connectivity with same Oracle DB s/w.

W.r.t. the Diagram,

- ① Server uses type 1/2/4/5 driver to interact with DB s/w and to create JDBC Connection pool having JDBC Connection Obj's.
- ② & ③ Server creates JDBCDataSource Object representing JDBC Connection pool and keeps that Object in Jndi registry having nickname (or) alias name.

- (4) Client application uses Jndi code to get DataSource object from registry.
- (5)(6)(7) Client application gets JDBC Connection Object from Connection Pool through Data Source object.
- (8) Client application uses this JDBC Connection object to create Other JDBC objects and to develop persistence Logic
- (9) Client application calls con.close() method to release JDBC connection Object back to Connection pool.

There are two types of jdbc Connection Obj's

① Direct JDBC Con Obj

→ It is the JDBC Con Obj created by the programmer manually

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl");
```

② Pooled JDBC Con Obj

→ It is the jdbc Con Obj collected from the JDBC Con pool.

* procedure to Create JDBC Connection pool for oracle in mydomain1 server of Weblogic.

Step-I:- Start mydomain1 server and Open its adminConsole.

Step-II:- Create JdbcDataSource representing JDBC Connection pool for oracle.

Admin Console Screen → Services → JDBC → Datasources →

new → Name: → Jndiname: → DatabaseType:

DB Driver: → next → next →

DataBase Name: Hostname: port: Database Username:

Password: Confirm Password: → next → Test Configuration

→ next → AdminServer → Finish.

Specify the JDBC Conn pool parameters

Admin Console → services → JDBC → DataSources → myds1 →

Connection Pool tab → Initial capacity → Max capacity seconds

Capacity increment: → Advanced → Shrink frequency seconds
↳ no. of connect objects that should be created when there is a need of creating new JDBC connection objects.
→ save

destroys idle JDBC connection objects after every 900 seconds.

NOTE:- in the above steps when Finish button is clicked the JDBC Data Source object reference will be placed automatically in Registry s/w.

* Org.springframework.JndiObjectFactoryBean is the FactoryBean that gathers object from registry s/w through Jndi lookup operation and injects that Object to specific Bean property. This class is very useful to gather DataSource object from Registry s/w and to inject that Object to our BeanClass property.

Example Spring application to use the above weblogic server managed JDBC Connection pool

Steps:- Keep mydomain1 server of Weblogic in running mode.

Step2:- keep Feb 18th cuppn ready. and perform the following modifications in

SpringCfg.xml file.

//SpringCfg.xml

<!DOCTYPE ... >

<beans>

<bean id="jofb" class="org.springframework.jndi.JndiObjectFactoryBean">

<property name="jndiName" value="sarkya"/>

↳ jndiname

Jndi
Properties to establish the Connection with Registry.

```

<property name="jndiEnvironment">
  <props>
    <prop key="java.naming.Factory.initial">weblogic.jndi.WLInitialContextFactory</prop>
    <prop key="java.naming.provider.url">t3://localhost:7071</prop>
  </props>
</property>
</bean>

<bean id="dobj" class="DBOperationBean">
  <property name="ds" ref="jofb"/>
</bean>
</beans>

```

↓
Injects JDBC DataSource Object gathered from Jndi registry by using JndiObjectFactoryBean. its resultant object will be injected.

Step-3:- Run the client application

> java DBOperationClient

Jar files in classpath:

spring.jar

Commons-logging.jar

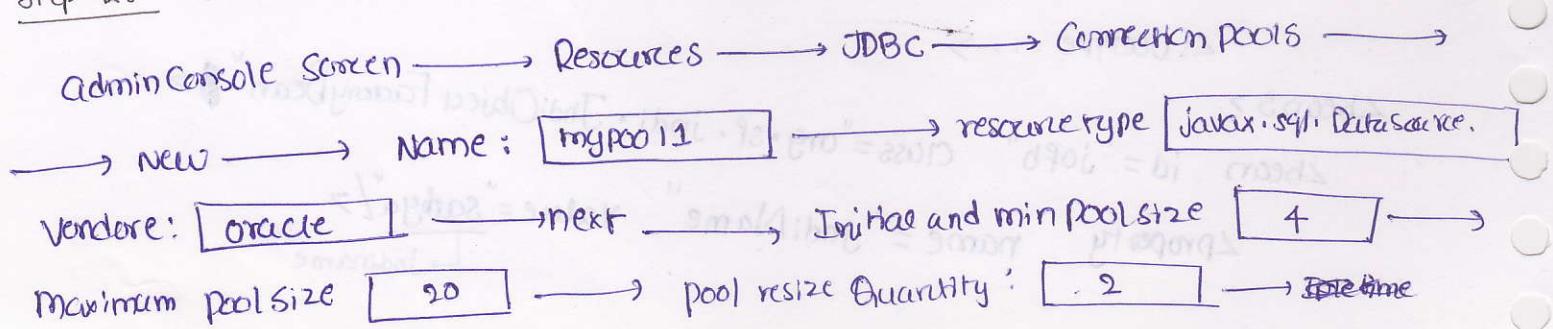
weblogic.jar

(*) procedure to create JDBC Connection pool for oracle in mydomain2

Step-0:- Servers of GlassFish 2.x.
keep Oracle thin driver rekited ojdbc14.jar file <glassfish-home>/domains/mydomain2/lib/ext folder.

Step-1:- Start mydomain2 server & open its adminConsole.

Step-2:- Create JDBC Connection Pool for oracle.



Database name orcl
 Driver Type thin
 Password tiger
 Port Number 1521
 Server Name localhost
 Service Name localhost
 URL jdbc:oracle:thin:@localhost:1521:orcl
 User scott

→ Finish → Select mypool1 → Ping → save

Step-III:- Create JDBC DataSource pointing to the above JDBC Connection pool.

Admin Console Screen → resources → JDBC → ~~Create~~ JDBC Resources

New → JNDI Name: Sathyas MyDsJndi → poolname: mypool1 → OK

Keeps DataSource obj in Jndi Registry
AppServer → Jndi Brow
→ see Jndi tree

* Step 2:- Refer appn on 18/02/2013 modify Spring.cfg.xml

to use GlassFish Server managed Connection pool in Spring appn

Step(1):- keep mydomain2 ^{server of} GlassFish in Running mode

Step(2):- keep feb 18th appn ready as shown in previous application but
change the ~~the~~ following Values in Springcfg.xml file.

(a) Change Jndiname to MyDsJndi from Sathyas

(b) change Weblogic Registry properties with GlassFish Registry properties.
refer 11th march discussions.

Step(3):- execute the client appn by adding following jar files

Spring.jar, commons-logging.jar

6 → GlassFish registry jar file refer march 12th

Spring DAO (also called as Spring JDBC)

15/03/2013

- The Java class that separates persistence logic from other logics of the application and makes that logics as reusable logics is called DAO
- DAO Design pattern is not only related with SpringDAO module.
- SpringDAO module gives JDBC Template class to provide abstraction layer on plain JDBC programming that means Simplifies the process of developing JDBC persistence Logic.

Plain JDBC programming

- ① register jdbc driver with DriverManager Service
- ② Establish the connection with DB SQL
- ③ Create JDBC Statement obj
- ④ send and execute SQL query in DB SQL } Application specific logic
- ⑤ Gather results and process the results } (changes in every appn)
- ⑥ Close JDBC obj's } Common logics of the appn (same in all JDBC apps)
- ⑦ performs Exception Handling } Common logics of the appn (same in all JDBC apps)

In the above application programmer should take care of Common and application specific logics.

Spring JDBC programming

- ① get JdbcTemplate class obj by supplying jdbc DataSource obj
- ② send and execute SQL Queries in DB SQL } Application Specific logics
- ③ Gather results and process the results } Common logics of the appn (same in all JDBC apps)

In the above appn JdbcTemplate class will take care of Common logics so programmer just need to take care of only Application specific logics

→ to create JdbcTemplate class Obj JdbcDataSource obj is dependent obj.

→ dataSource is the important property of JdbcTemplate class (org.springframework.jdbc.core.pkg)

The important methods of JdbcTemplate class to send and execute the

SQL Queries :-

TO execute non-select Queries

update(String qry)

TO execute select SQL Queries

queryForInt (String qry) } When select query return numeric values

queryForLong (String qry) } Eg:- select count(*) from student.

queryForMap (String qry) } when select query returns single record.

Eg:- Select * from student where sno=101

queryForList (String qry) } when select query returns multiple records.

Eg:- Select * from student.

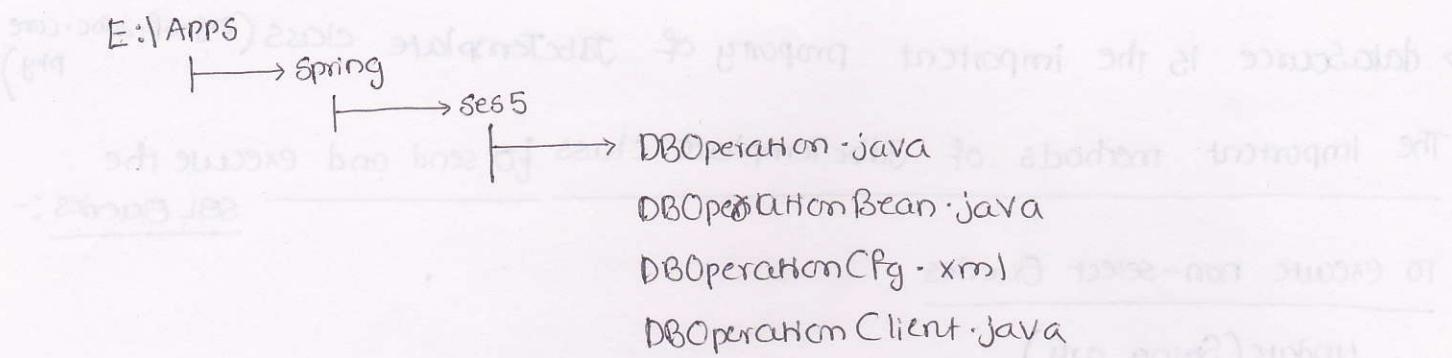
To perform Batch Processing :-

batchUpdate (String[] queries)

→ When all these methods are having single argument then Simple Statement obj will be used internally.

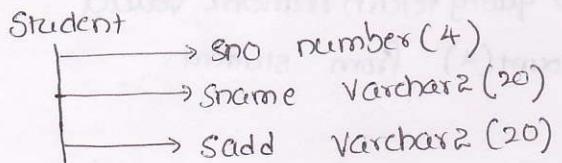
→ When all these methods are called having two arguments or three arguments then PreparedStatement obj will be used internally.

Example Application :-



Prefer using Third party jdbc com pool like Apache DBCP.

DB table in Oracle



```

// DBOperation - Java
import java.util.*;
public interface DBOperation
{
    public int insertInfo(int no, String name, String add);
    public int updateDetails(int no, String newname, String newadd);
    public int deleteDetails(int no);
    public int countStudents();
    public Map getStudDetails(int no); → BMap
                                                → to execute select query for
                                                    single record.
    public List getStudDetails(String city); → List
                                                → to execute select query
                                                    for multiple records.
}
  
```

// DBOperationBean.java

```

import org.springframework.jdbc.core.*;
import java.util.*;
public class DBOperationBean implements DBOperation
{
    JdbcTemplate jt;
    // method for constructor injection
    public DBOperationBean(JdbcTemplate template)
    {
        this.jt = template;
    }
  
```

```

    //implement B-methods
    public int insertInfo (int no, String name, String addrs)
    {
        int result = jt.update ("insert into Student values (?, ?, ?)",
                               new Object[]{no, name, addrs});
        Supplies the query parameter values
        return result;
    }

    public int updateDetails (int no, String newname, String newaddrs)
    {
        int result = jt.update ("update student set sname=? , saddr=? where",
                               "sno=? ", new Object[]{newname, newaddrs, no});
        ↳ using PreparedStatement objects.
        return result;
    }

    public int deleteDetails (int no)
    {
        int result = jt.update ("delete from student where sno=? ", new Object[]{no});
        return result;
    }

    public int countStudents ()
    {
        int result = jt.queryForInt ("select count(*) from student", new Object[]{});
        return result;
    }

    public Map getStudDetails (int no)
    {
        Map m = jt.queryForMap ("select * from student where sno=? ", new
        Object[]{no});
        return m;
    }

    public List getStudDetails (String city)
    {

```

JDBC template through
 DataAccessException.
 This is an unchecked
exception. So, exception
 handling in DAO module
 is optional.

AutoUnboxing
 concept

Map Data Structure
Map Based

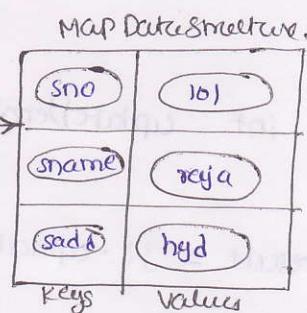
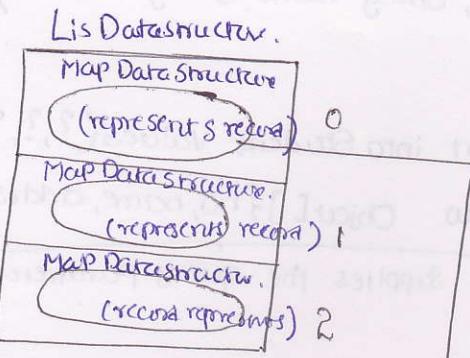
sno	101	0
sname	Raja	1
saddr	Hyd	2

Column Names Column Values
Keys Values

```

List l=jt.queryForList ("select * from Student where Suid = ? ", new
Object[]{ctry});
return l;
}
}

```



// DBOperation Cfg.xml.

<beans>

```

<bean id="dbcp" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName"

```

</bean>

```

<bean id="template" class="org.sf.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dbcp" />

```

</bean>

```

<bean id="dob" class="DBOperation Bean">
    <constructor-arg ref="template" />

```

</bean>

</beans>

// DBOperation Client.java

```

import org.sf.context.support.*;

```

public class DBOperationClient

```

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

```

// Activate Container

FileSystemXmlApplicationContext ctx = new FileSystemXmlApplicationContext("DBOperation.cfg.xml");
// getBean class obj
DBOperation bobj = (DBOperation) ctx.getBean("dob");
// call Methods

S.o.p("no.of Records that are inserted "+bobj.insertInfo(101,"rcya","hyd"));

S.o.p("no.of Records that are Updated "+bobj.updateDetails(101,"newrcya","newhyd"));

S.o.p("no.of Records that are deleted "+bobj.deleteDetails(101));

S.o.p("no.of students Count "+bobj.~~student~~.countStudents()));

S.o.p("101 student details "+bobj.getStudDetails(101));

S.o.p("101 hyd student details "+bobj.getStadeDetails("hyd2"));

} // main

} // class

jar files in CLASS PATH :-

2 Spring jars → spring.jar, Commons-logging.jar

Commons-dbcop.jar

ojdbc14.jar

Q:- What is the diff b/w executing selectquery in plainJDBC and in Spring JDBC?

Ans:- plain JDBC gives ResultSet object. And it is not Serializable object to send over the network.

Spring JDBC select query gives Results in List, Map Datastructures and

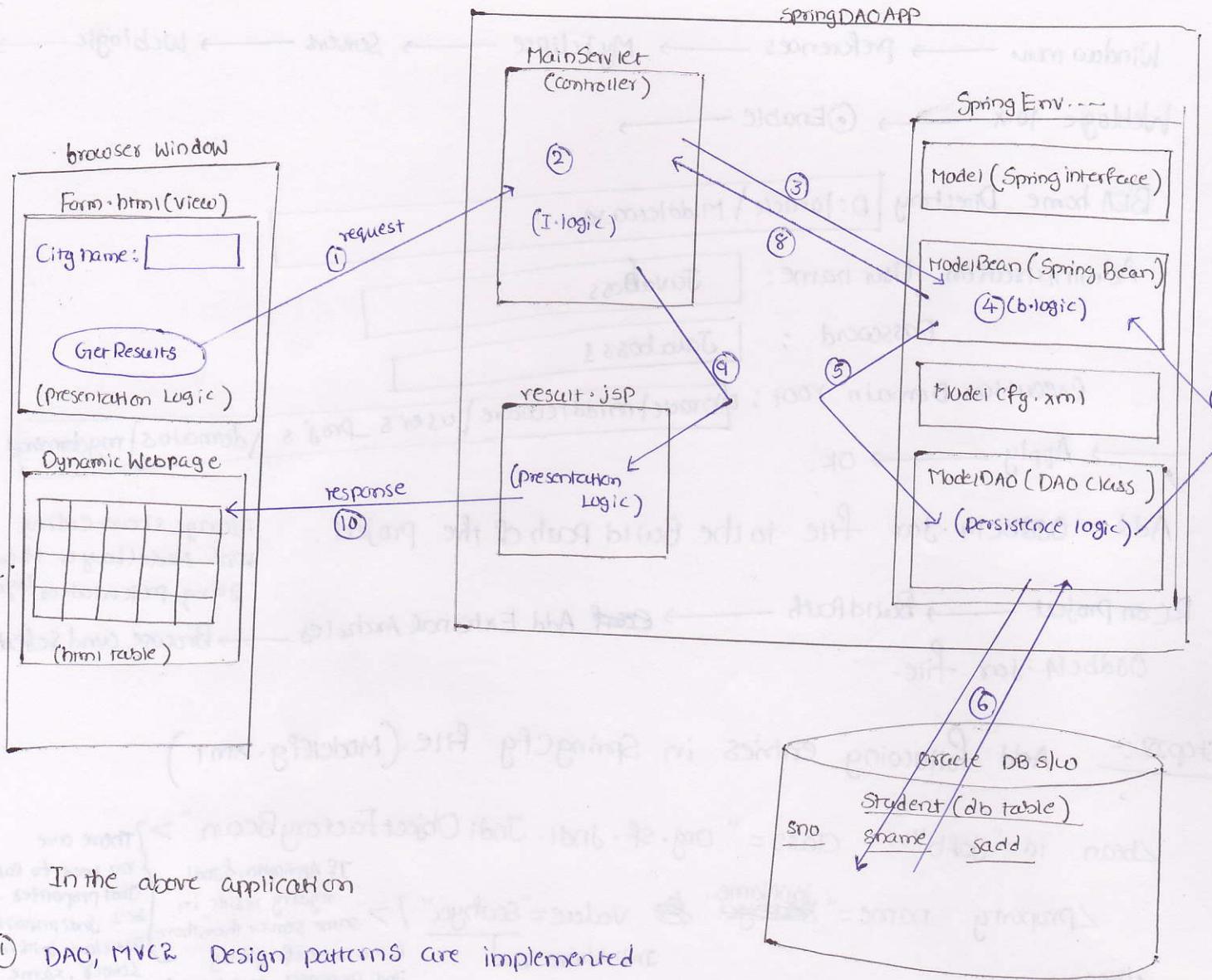
we can send this Datastructures directly over the network (All collection framework Datastructures are Serializable objects by default)

Spring JDBC internally uses plain JDBC even though plain JDBC throws checked exception called `SQLException`. The Spring JDBC converts that one to unchecked exception called `DataAccessException`.

```
Ex:- public int queryForInt(String qry) throws DataAccessException  
{  
    try {  
        // plain JDBC code  
    }  
    catch (SQLException e) {  
        throw new DataAccessException();  
    }  
}
```

Exception Rethrowing

The Java class that separates persistence logic from other logics of the application is called DAO. DAO makes the persistence logic as reusable logic and the flexible logic to modify.



In the above application

- ① DAO, MVC2 Design patterns are implemented.
- ② DAO class contain Spring JDBC based persistence Logic.
- ③ USE Server managed Connection pool in the above application.

Design Pattern is a Best solution for recurring problems of application Development

Design Patterns are the Best practices to develop s/w applications by using s/w technologies

~~Ex~~ ① procedure to develop the above webapplication by using MyEclipse 10.x Server.

Step-I:- Create web project

File → New → WebProject

proj Name: → Finish

Step-II:- Add spring capabilities to the project

Right click on project → MyEclipse → Add spring capabilities → ① Spring 3.0
 → Spring 3.0 Core, persistence core, persistence JDBC Libraries → Spring Cfg

Modelcfg.xml → File → Finish

Step III:- Configure MyDomain1 server of weblogic server with My Eclipse IDE.

Window menu → Preferences → My Eclipse → Servers → Weblogic → Weblogic 10.x → Enable →

BEA home Directory [D:\oracle\middleware]
Administration User name: JavaBoss
Password: JavaBoss
Execution Domain root: D:\oracle\middleware\users_Proj's\domains\mydomain1
→ Apply → OK.

Add JDBC14.jar file to the Build Path of the project.
Always start coding with Model layer then going to presentation layer.
Right Click on Project → Build Path → ~~Export~~ Add External Archives → Browse and Select JDBC14.jar file.

Step IV:- Add following entries in SpringCfg file (Modelcfg.xml)

```
<bean id="jofb" class="org.sf.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="Sathya" />
    </bean>

<bean id="template" class="org.sf.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="jofb" />
</beans>

<bean id="mydao" class="ModelDAO">
    <property name="jt" ref="template" />
</bean>

<bean id="mb" class="ModelBean">
    <property name="mdao" ref="mydao" />
</bean>
```

If Application & Jndi registry reside in same server then there is no need of giving Jndi properties to talk with registry.
There are no need to supply Jndi properties bcz Jndi properties and Jndi class both in the service. Same location.

Step-V: Develop the DAO class as shown below
RC on src → Class → classname: ModelDAO → Finish.

```
public class ModelDAO {
```

// bean property

^{ctrl+shift+t0} JdbcTemplate jt;

// Write setXXX(-) method for setter injection

```
public void setJt(JdbcTemplate jt)
```

{

this.jt = jt;

}

```
public Map<String, List> getDetails(String city)
```

{

List<Map<String, List<Student>>> list = jt.queryForList("select * from student where saddr = ?;", new Object[] {city});

return list;

} // method

} // class

Step-VI: Add Spring Interface, Spring Bean class to the project

L Model M ModelBean

RC on Src → New → Interface → Name: Model → finish.

// Model.java

```
public interface Model {
```

```
    public Map<String, List<Student>> searchStudents(String city);
```

}

RC on Src → New → class → Name: ModelBean → add: Model → finish.

```
public class ModelBean implements Model {
```

// bean property

ModelDAO mdao;

// setter method for Setter Injection

```
public void setMdao(ModelDAO mdao)
```

{

this.mdao = mdao;

}

```
public Map<String, List<Student>> searchStudents(String city) {
```

// b. logic

```

if (city == null || City.equals(""))
{
    City = "hyd";
}

//use persistence logic of DAO class

List Map = mdao.getDetails(city)
return Map;
} //method

} //class

```

Step-VII :- add Form.html to the webroot folder of the project

R.C ~~to~~ Webroot → new → html → Name : Form.html

```

<form action="Controller" method="get">
    city name: <input type="text" name="city" />
    <input type="submit" value="search details" />
</form>

```

Step - VIII :- add servlet program to the project

R.C ~~on src folder~~ → new → servlet → Name : MainServlet → init & destroy, doGet() doPost()

→ next → Servlet / JSP mapping (Or) : /Controller → Finish.

```

public class MainServlet extends HttpServlet {
    Model bobj;
    public void init(){
        //Activate SpringContainer
    }
}

```

ClassPathXmiApplicationContext ctx = ClassPathXmiApplicationContext("ModelConfig.xmi")

// get Spring Bean class object from Container

Model bobj = (Model) ctx.getBean("mb");

} // init()

Public void doGet(HttpServletRequest request, HttpServletResponse) throws ServletException {

// read form data

String s1 = request.getParameter("city");

// call B.method

List Map $m = \text{obj}.\text{searchStudents}(s_1);$

// keep result in Request attribute

request.setAttribute("attr1", m);

// forward the Request to result page

RequestDispatcher rd = request.getRequestDispatcher("result.jsp");

rd.forward(request, response)

} // doGet(-,-)

public void doPost(HttpServletRequest request, HttpServletResponse response) throws SE, IOException

{

doGet(request, response);

}

public void destroy()

obj = null;

}

Step-8: add result.jsp to Webroot folder

R.C on Webroot \rightarrow new \rightarrow JSP \rightarrow name: result.jsp

<%@page import="java.util.*" %>

<% List l = (List) request.getAttribute("attr1"); %>

<table>

<tr>

<th> sno </th>

<th> sname </th>

<th> sadd </th>

</tr>

<% for (int i=0; i < l.size(); ++i) { %>

&

Map m = (Map) l.get(i); %>

<tr>

<td> <%= m.get("SNO") %> </td>

<td> <%= m.get("SNAME") %> </td>

<td> <%= m.get("SADD") %> </td>

</tr>

<%>

%> </table>

Step 3:- Run the project

Rc on project → Run as → My Eclipse server application →
Weblogic10.x

Test the project.

Open Browser window → http://localhost:7171/TestProj1/Form.htm

Procedure to configure glassFish 2.X Domain Server with MyEclipse IDE.

Window menu → preferences → My Eclipse → Servers → GlassFish →

GlassFish 2.X → Enable → Home directory : D:\sun\Appserver →

Configuration directory : D:\sun\Appserver\config → Domain name : mydomain2

→ Apply → OK

→ In the configuration file to change Jndiname to Sathya to myDSJndi

If underlying server does not provide server managed connection pool then
programmer can use third party connection pools in that Spring based
Web application.

plain JDBC supports only positional parameters (?) in the query

Spring JDBC supports both position(?) and named parameters (<name>) in the query.

Name Parameters are self descriptive, to work positional parameters we need
JDBC Template class, to work with Named Parameter we need NamedParameterTemplate class

* For example application on Named Parameters refer page No: 54/55 Application ⑧

if certain persistent operations are not directly possible with the methods of JdbcTemplate class then we can implement those operations with callback interfaces support.

RowMapper → To process one record of the ResultSet Obj

ResultSetExtractor → To process the multiple records of ResultSet Obj

PreparedStatementCreator → gives com object to create and use JDBC Prepared Statement Obj

PreparedStatementSetter → gives PreparedStatement obj to use it.

* For example application on Spring DAO Base Select operations and working with

Various callback interfaces refer application: (7) of the page no's (51) to (54)

19/03/2013

→ In order to make certain Business Logic as reusable logic and centralized logic we can work with PL/SQL procedures and functions of DataBase s

→ PL/SQL procedures and Functions are visible in multiple applications of a module and in multiple modules of a project.

→ Using Spring JDBC we can call PL/SQL procedure or Function. For this we need to take a Java class extending from org.springframework.jdbc.object.StoredProcedure class and we use execute() to call the PL/SQL procedure (or) Function.

* For example application on calling PL/SQL procedure of Oracle DB s/w refer page No: 56 & 57 Appn No: 10

→ plain JDBC and Spring JDBC persistence logic will be developed by using SQL Queries and these SQL Queries are DataBase s/w dependent queries so, these persistence logic are DataBase s/w dependent persistence logic

→ To develop Objects based DataBase s/w independent persistence logic without using SQL Queries then we need to work with ORM s/w's like Hibernate, JBoss Seam, TopLink and etc...

→ Spring ORM module does not supply its own ORM s/w. but it's provide abstraction layer on existing ORM s/w's and simplifies the process of working with ORM s/w's.

for this it supplies multiple Template classes.

ORM slow

Hibernate
ibatis
Jdo
Jpa
Toplink

its Template class

org.sf.orm.hibernate3.HibernateTemplate
org.sf.orm.ibatis.SqlMapClientTemplate
org.sf.orm.JdoTemplate
org.sf.orm.JpaTemplate
org.sf.orm.toplink.TopLinkTemplate

Org.sf.orm.hibernate3.LocalSessionFactoryBean gives SessionFactory obj. For this JdbcDataSource obj is required.

SessionFactory obj is required to create HibernateTemplate class obj.

Configuration in Spring cfg file

- Bean that gives JDBC DataSource obj
- Bean that gives HibernateSession Factory obj (LocalSessionFactory)
- Bean ~~that~~ ^{of} gives HibernateTemplate class

important methods of HibernateTemplate class

for Single row operations:

- Save(-) / persist(-) → for record insertion
- update(-) → for record update
- load(-) / get(-) → for selecting a record
- delete(-) → for deleting record
- saveOrUpdate(-) / merge(-) → for inserting / updating record.

for bulk operations:

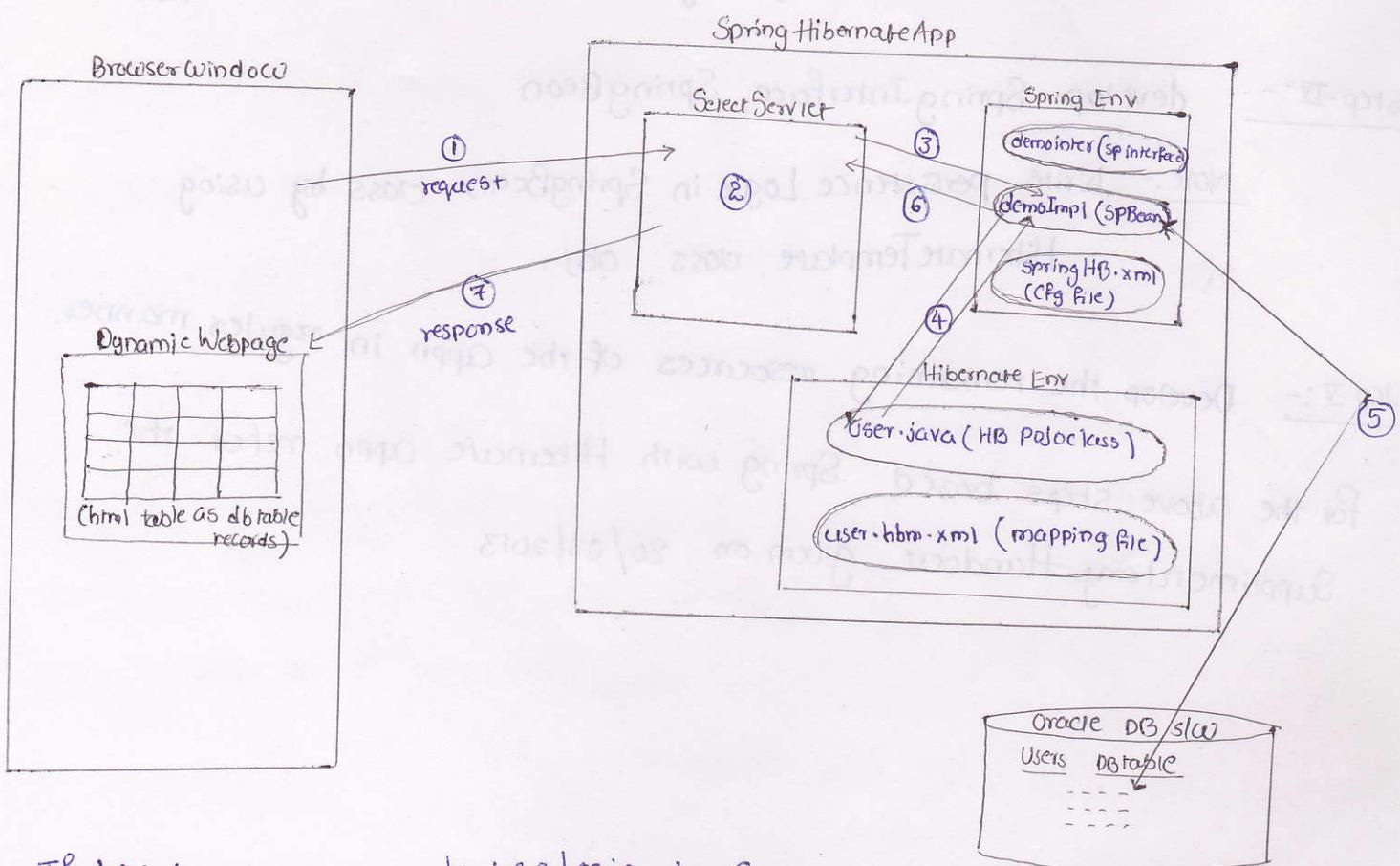
- bulkUpdate(-) → for HQL Based non-select operation
- find(-), findXxx(-) → for HQL Based select operation
- iterate(), ~~list()~~ → for HQL based select operations.

Module - III

Spring ORM Module

- In Spring with Hibernate Application the Spring Appn represents B-Logic and Hibernate Appn represents persistence Logic.
- Spring and Hibernate Integration is nothing but adding hibernate persistence logic in Spring appn.
- To write Hibernate ~~appn~~ persistence logic in Spring Bean class by using Spring CRM module we need HibernateTemplate class obj.

Spring with Hibernate Appn (using Hibernate Template class obj)



If there is no separate business logic in Spring Bean then persistence logic itself acts as B.Logic of the Spring Bean

In Spring with Hibernate application we can take separate Spring, Hibernate configuration files (or) we can take only Spring configuration file also maintaining the entries of Hibernate configuration.

Procedure to develop the above Spring with Hibernate application

Step-I:- Develop Hibernate POJO class, mapping file

Step-II:- keep Spring and Hibernate related jar files in CLASSPATH and WEB-INF/lib folder
jar files in CLASSPATH: spring.jar, hibernate3.jar

jar files in WEB-INF/lib folder:
2 → spring jar files
8 → hibernate jar files

Step-III:- Cfg the following beans in Spring cfg file

- Bean giving JdbcDataSource object
- Local SessionFactory Bean giving Hibernate SessionFactory obj.
- HibernateTemplate class as Spring bean
- Our Spring Bean injecting HibernateTemplate class obj.

Step-IV:- develop Spring Interface, Spring Bean

NOTE:- Write persistence Logic in SpringBean class by using
HibernateTemplate class obj.

Step-V:- Develop the remaining resources of the appn in regular manner.

for the above steps based Spring with Hibernate appn refer the

Supplimentary Handout given on 20/03/2013

The HAL Query that is placed in mapping file having logical name is called "Named HAL query".

and by Named param.

* Executing Native SQL queries with Positional Parameters

Step-I :- Write Native SQL query with Positional Parameters

<h-m>

<class> --->

</class>

<sql-query name="test2">

<return class="Users"/>

<!CDATA[select * from users where userid >= ? and userid <=?]>

</sql-query>

</h-m>

Step-II :- execute the above NativeSQL query.

public Iterator getData()

{

List l = ht.findByNameQuery("test2", new Object[] { 100, 300 });

Iterator it = l.iterator();

return it;

}

* Executing Native SQL query with Named parameters.

Step-I :- prepare Native SQL query with Named parameters in mapping file

<h-m>

<class> --->

</class>

<sql-query name="test2">

<return class="User"/>

select * from users where role = :P1

</sql-query>

</h-m>

Step-II :- execute the above query

public Iterator getData()

{

List l = ht.findByNameQueryAndNamedParam("test2", new String[] { "P1" }, new Object[] { 100, 300 });

Iterator it = l.iterator();

// to insert Single record

```
User u1=new User();
u1.setUId(1001);
u1.setUname("Rajesh");
u1.setRole("P.I");
ht.save(u1);
return it;
```

While Working with plain Hibernate all non-select persistent operations must be executed as Transactional statements.
But in Spring Hibernate that is not required.

*)

```
public Iterator getData()
List l=ht.findByNameQueryAndNamedParam("test2",new String[]{"P.I"},new Object[])
Iterator it=l.iterator();
// to perform bulk modification using HQL
int res=ht.bulkUpdate("update User u1 set u1.role=? where u1.role=?",
new Object[]{"P.I","T.I"});
S.o.p("no.of records that are updated "+res);
return it;
```

→ HibernateTemplate class provides abstraction Layer on plain hibernate and takes care the following internal operations.

- 1) Session obj creation
- 2) Transaction mgmt
- 3) closing of SessionFactory, Session Obj's and etc....

*) The following operations are not possible directly with HibernateTemplate class.

- ① Non-select HQL Queries with NamedParameters
- ② Non-select Native SQL Queries with Named (or) positional parameters.
- ③ HQL select query with Named parameters Criteria API based Logics execution
- ④ Native SQL select query with Both Named, positional parameters.

To implement all these facilities we need to work with HibernateCallback interface
org.sf.orm.hibernate3.*;
Supplied by Spring ORM module. This interface gives doInHibernate(-) exposing
the Session object.

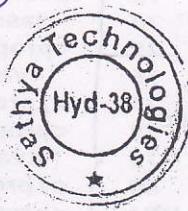
1 =====
 2 Title: Spring with Hibernate App (approached)
 3 =====
 4 -----User.java-----
 5 public class User
 6 {
 7 private int uid;
 8 private String uname,role;
 9 public void setUid(int n)
 10 {
 11 uid=n;
 12 }
 13 public int getUid(){ return uid; }
 14 public void setUname(String s){ uname=s; }
 15 public String getUname(){ return uname; }
 16 public void setRole(String r){ role=r; }
 17 public String getRole(){ return role; }
 18 }
 19
 20 /*
 21 create table users
 22 (userid number(5) primary key,
 23 uname varchar2(20),
 24 role varchar2(20)
 25);
 26 */

27 -----User.hbm.xml-----
 28 <?xml version="1.0"?>
 29 <!DOCTYPE hibernate-mapping PUBLIC
 30 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
 31 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
 32 <hibernate-mapping> *POJO class*
 33 <class name="User" table="users">
 34 <id name="uid" column="userid" />
 35 <property name="uname" />
 36 <property name="role" />
 37 </class>
 38
 39 </hibernate-mapping>

40 -----demointer.java-----
 41 import java.util.*;
 42 public interface demointer
 43 {
 44 public Iterator getData();
 45 } *Business method Declaration*

46 -----demoimpl.java-----
 47 import java.util.*;
 48 import org.springframework.orm.hibernate3.*;
 49
 50 public class demoimpl implements demointer

51 {
 52 private HibernateTemplate ht;
 53 {
 54 public void setHt(HibernateTemplate ht)
 55 {
 56 this.ht=ht;
 57 }
 58 }
 59 public Iterator getData()
 60 {
 61 List l=ht.find("from User"); *HQL query*
 62 Iterator it=l.iterator();
 63
 64 return it;
 65 }
 66 }
 67 -----SpringHB.xml-----
 68 <?xml version="1.0" encoding="UTF-8"?>
 69 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"



```

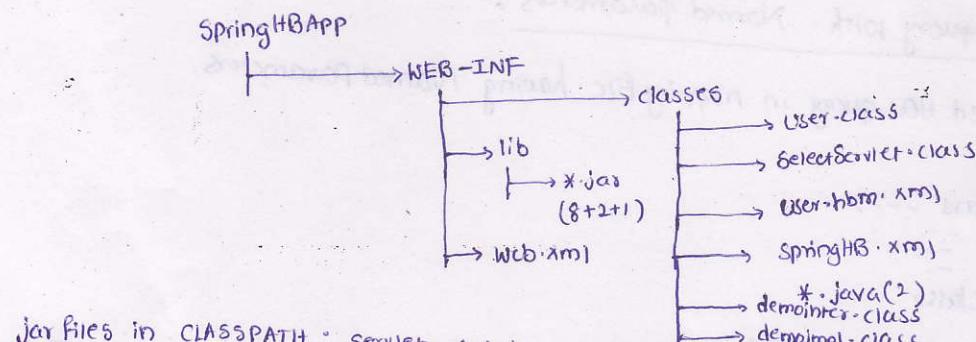
70  "http://www.springframework.org/dtd/spring-beans.dtd">
71  <beans>
72
73  <bean id="myds"
74    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
75    <property name="driverClassName">
76      <value>oracle.jdbc.driver.OracleDriver</value>
77    </property>
78    <property name="url">
79      <value>jdbc:oracle:thin:@localhost:1521:satya</value>
80    </property>
81    <property name="username">
82      <value>scott</value>
83    </property>
84    <property name="password">
85      <value>tiger</value>
86    </property>
87  </bean>
88  {Connection pool}
89  <bean id="mySessionFactory" class=
90    "org.springframework.orm.hibernate3.LocalSessionFactoryBean">
91    <property name="dataSource" ref="myds"/>
92    <property name="mappingResources">
93      <list>
94        <value>User.hbm.xml</value>
95      </list>
96    </property>
97    <property name="hibernateProperties">
98      <props>
99        <prop key="hibernate.dialect">org.hibernate.dialect.OracleDialect</prop>
100       <prop key="show_sql">true</prop>
101     </props>
102   </property>
103 </bean>
104 <bean id="template" class="org.springframework.orm.hibernate3.HibernateTemplate">
105   <property name="sessionFactory"><ref bean="mySessionFactory"/></property>
106   </bean>
107   {ref line no: 89}
108
109 <bean id="d1" class="demoimpl">
110   <property name="ht"><ref bean="template"/></property>
111 </bean>
112 {ref line no: 105}
113 </beans>
114 -----SelectServlet.java-----
115 import javax.servlet.*;
116 import javax.servlet.http.*;
117 import java.io.*;
118 import org.hibernate.*;
119 import java.util.*;
120 import org.springframework.beans.factory.*;
121 import org.springframework.context.*;
122 import org.springframework.context.support.*;
123 public class selectservlet extends HttpServlet
124 {
125   public void service(HttpServletRequest request, HttpServletResponse response)
126     throws ServletException, IOException
127   {
128     PrintWriter out=response.getWriter();
129     System.out.println("In service method of Servlet");
130     ApplicationContext ctx=new ClassPathXmlApplicationContext("SpringHB.xml");
131     BeanFactory factory=(BeanFactory)ctx;
132     demointer d1=(demointer)factory.getBean("d1"); gives spring Bean class object
133     Iterator i1=d1.getData();
134     out.println("<body bgcolor=#ffffcc text=red>");
135     out.println("<h1><center>all users</h1><hr><br><h3>");
```

137-144 logic to display records as HTML table by gathering results from the
Business methods of SpringBean.

```

139         User u1=(User)i1.next();
140         out.println("<tr><td>"+u1.getId()+"<td>"+u1.getUsername()+""
141                         "<td>"+u1.getRole()+"</tr>");
142     }
143     out.println("</table>");
144 }
145 catch(HibernateException e)
146 {
147     out.println(e);
148     e.printStackTrace();
149 }
150 }
151 }
-----web.xml-----
153 <?xml version="1.0" encoding="ISO-8859-1"?>
154 <web-app xmlns="http://java.sun.com/xml/ns/j2ee"
155   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
156   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
157   http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
158   version="2.4">
159 <servlet>
160   <servlet-name>select</servlet-name>
161   <servlet-class>selectservlet</servlet-class>
162 </servlet>
163 <servlet-mapping>
164   <servlet-name>select</servlet-name>
165   <url-pattern>/selectaction</url-pattern>
166 </servlet-mapping>
167 </web-app>
168

```



Jar files in CLASSPATH : servlet-api.jar, spring.jar, hibernate3.jar

Jar files in WEB-INF/lib folder :

- 2 → spring jarfiles
- 8 → Hibernate jar files
- 1 → DJDBC14.jar file

More persistence operations by using HibernateTemplate class.

*) Executing HQL select query with positional parameters

```

Public Iterator getData()
{
    List l=ht.find("From User where uid=? and uid=? ", new Object[]{100,200});
    Iterator it=l.iterator();
    return it;
}

```

↳ parameter values

↳ Positional parameters

*) Executing HQL select query with Named parameters

```

Public Iterator getData()
{
    List l=ht.findByNameParam("From User where uid=:P1 and uid=:P2 ", new String[]{"P1","P2"}, new Object[]{100,200});
    Iterator it=l.iterator();
    return it;
}

```

↳ named parameter

↳ new Object[]{100,200}

Executing Named HQL query with Positional parameters :-

Step-I:- Write Named HQL query in hibernate Mapping file

```
<h-m>
<class -->
  ...
</class>
<query name="test1">
  from User ub where ub.role like ?
</query>
</h-m>
```

Step-II: execute the above HQL query.

```
public Iterator getData()
{
  List l = ht.findByNameQuery("test1", new Object[] {"%P1"});
  Iterator it = l.iterator();
  return it;
}
```

Executing Named HQL query with Named parameters :-

Step-I:- Prepare Named HQL query in mapping file having Named Parameters.

```
<h-m>
<class -->
  ...
</class>
<query name="test1">
<!CDATA[ from User ub where ub.uid >= :P1 and ub.uid <= :P2 ]>
</query>
</h-m>
```

Step-II:- execute the above HQL query.

```
public Iterator getData()
{
  List l = ht.findByNameQueryAndNamedParam ("test1", new String[] {"P1", "P2"}, new Object[] {100, 300});
  Iterator it = l.iterator();
  return it;
}
```

Executing Hibernate Persistence Logic of Criteria API by using HibernateCallback (I) of Spring ORM module.

```
public Iterator getData()
{
    List l = (List) ht.execute(new HibernateCallback() {
        public Object doInHibernate(Session ses) {
            // Criteria API based persistence logic
            Criteria ct = ses.createCriteria(User.class);
            // add Condition
            Criterion cond = Restrictions.like("role", "%-P-1");
            ct.add(cond);
            // execute logic
            List l = ct.list();
            return l;
        }
    });
}

Iterator it = l.iterator();
return it;
}
```

NOTE: In the above code ht.execute() method is called having anonymous inner class object that implements HibernateCallback (I).

To execute this code place the following jar files in the classpath.

```
{ Spring.jar
  { hibernate3.jar
    Servlet-api.jar }
```

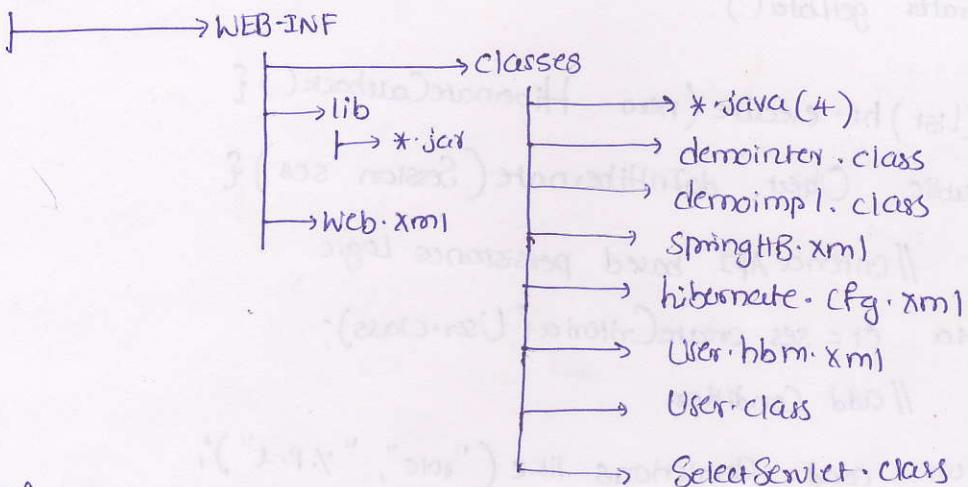
→ In Spring with Hibernate applications we can take Hibernate Configuration file and

Spring Configuration file separately, then Hibernate Configuration file must be linked with spring config file by using Configure Location property of LocalSessionFactoryBean.

In this situation there is no need of configuring bean that gives dataSource object in Spring Configuration file.

Revised Spring With Hibernate App of previous class

SpringHBApp



hibernate.cfg.xml :-

Same old hibernate configuration file.

SpringHB.xml

<!DOCTYPE ----- >

<beans>

<bean id="mySessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

<property name="configLocation" value="classpath:hibernate.cfg.xml"/>

</beans>

= // Same old HibernateTemplate, Demoimpl classes configuration
= refer previous appn.
=

</beans>

based spring, hibernate integration

* For example application on annotations (Spring 3, hibernate 4).

refer the supplementary Handout given on 23/03/2013

Spring JEE Module

- 1) gives ApplicationContext Container
 - 2) provides abstraction Layer on javamail api for mailing operations.
 - 3) provides abstraction Layer on JMS for messaging operations.
 - 4) provides abstraction Layer on rmi, webservices to develop distributed applications
 - 5) gives direct distributed technology called HttpInvokers.
 - 6) Supports Timer based scheduling operations.
 - 7) provides abstraction Layer on JNDI to interact with Registry softwares.
 - and etc...
- Making our application executing on Certain Date and time either only for one time or for multiple times is nothing but enabling scheduling on the applications. While dealing with Time critical jobs this scheduling is very important.

Examples:-

Sending reminders on the 1st of every month.

Cleaning temporary files every day at 12:0' clock.

We can use TimerTask, Timer class of jdk environment for these operations.

→ The Java class that extends from TimerTask class defines the job on which we are planning to enable service.

→ The scheduled method of Timer class can enable scheduling on the job

for jdk level Timer Service application refer application (8) of the page no's 67 and 68.

1975 - 1982 The logic on which we are looking to enable Timer Service

1992 - 1996 Initial delaying milli seconds

$5000^{(x)}$ Specifies initial delay.

$3000^{(y)}$ specifies the time gap b/w two successive executions.

1994 - 1996 execute the job on specified Date and time

Advantages of Spring Scheduling abstraction Layer on Jdk level scheduling and gives the following benefits.

- ① allows to perform scheduling on multiple jobs.
- ② allows to specify scheduling related parameters declaratively using xml files.
- ③ allows to perform scheduling on the user defined methods of user defined classes. that means classes need not to be extended from TimerTask class.

Three important beans of Spring Based Timer Operations.

- ① org. sf. scheduling. timer. TimerFactoryBean
(Starts ~~factory~~ Timer service When Container is started)
- ② org. sf. scheduling. timer. SchedulingTimerTask
→ represents the job on which we want to enable Timer Service
→ This job must be there in a java class that extends from TimerTask class.
- ③ org. sf. scheduling. timer. MethodInvokingTimerTaskFactoryBean
→ represents the job on which we want to enable Timer Service
→ This job can be there in a user-defined method of user-defined java class.

For Spring based scheduling example applications refer application 19, 20 of the page no's 68 to 70

sprint.txt

3/23/2013 9:22 AM

```

1 SpringHBAppAnno
2   |----->WEB-INF
3   |   |----->classes
4   |   |   |----->p1
5   |   |   |   |----->Demolnter.java/.class
6   |   |   |   |----->Demolmple.java/.class
7   |   |   |   |----->SelectServlet.java/.class
8   |   |   |   |----->User.java/.class
9   |   |----->SpringHB.xml
10  |----->lib
11  |   |----->(*.jar)
12  |----->web.xml

13 jars in lib
14 -----
15 antlr-2.7.7.jar      15-23 collect from hibernate 4-x
16 commons-logging.jar  24-25 collect from Jboss JBoss
17 dom4j-1.6.1.jar      27-36 collect from spring 3.1
18 hibernate-commons-annotations-4.0.1.Final.jar
19 hibernate-core-4.1.8.Final.jar
20 hibernate-ehcache-4.1.8.Final.jar
21 hibernate-jpa-2.0-api-1.0.1.Final.jar
22 javassist-3.15.0-GA.jar
23 javax.persistence.jar
24 jboss-logging-3.1.0.GA.jar
25 jboss-transaction-api_1.1_spec-1.0.0.Final.jar
26 ojdbc14.jar
27 org.springframework.asm-3.1.1.RELEASE.jar
28 org.springframework.aspects-3.1.1.RELEASE.jar
29 org.springframework.beans-3.1.1.RELEASE.jar
30 org.springframework.context.support-3.1.1.RELEASE.jar
31 org.springframework.context-3.1.1.RELEASE.jar
32 org.springframework.core-3.1.1.RELEASE.jar
33 org.springframework.expression-3.1.1.RELEASE.jar
34 org.springframework.jdbc-3.1.1.RELEASE.jar
35 org.springframework.orm-3.1.1.RELEASE.jar
36 org.springframework.transaction-3.1.1.RELEASE.jar
37 -----
38 =====
39 Demolnter.java
40 -----
41 package p1;
42 import java.util.*;
43 public interface Demolnter
44 {
45     public Iterator getData()throws Exception;
46 }
47 -----
48 Demolmpl.java
49 -----
50 package p1;
51 import org.hibernate.*;
52 import java.util.*;
53 import org.springframework.stereotype.*;
54 import org.springframework.beans.factory.annotation.Autowired;
55
56 @Component("d1")
57 public class Demolmpl implements Demolnter
58 {
59     @Autowired
60     private SessionFactory sesfact;
61
62     public void setSesfact(SessionFactory f1)
63     {
64         sesfact=f1;
65     }
66
67     public Iterator getData()throws Exception
68     {

```

JAR files in CLASSPATH

javax.persistence.jar
 org.sf.context.support-3.1.1.RELEASE.jar
 org.sf.context-3.1.1.RELEASE.jar
 org.sf.beans-3.1.1.RELEASE.jar
 org.sf.core-3.1.1.RELEASE.jar
 hibernate-core-4.1.8.Final.jar

```
69         Session ses=sesfact.openSession();
70         Query query=ses.createQuery("from User");
71         Iterator i1=query.iterate();
72         return i1;
73     }
74 }
75 =====
76 User.java
77 -----
78 package p1;
79 import javax.persistence.*;
80
81 @Entity
82 @Table (name= "users")
83 public class User
84 {
85     @Id
86     @GeneratedValue
87     @Column(name = "userid")
88     private int uid;
89
90     @Column(name= "uname")
91     private String uname;
92
93     @Column (name= "role")
94     private String role;
95
96     public void setUid(int n)
97     {
98         uid=n;
99     }
100
101    public int getUid(){ return uid; }
102    public void setUserName(String s){ uname=s; }
103    public String getUserName(){ return uname; }
104    public void setRole(String r){ role=r; }
105    public String getRole(){ return role; }
106 }
107 =====
108 DBTable
109 -----
110 create table users
111 ( userid number(5) primary key,
112   uname varchar2(20),
113   role varchar2(20)
114 );
115 =====
116 SpringHB.xml
117 -----
118 <beans xmlns="http://www.springframework.org/schema/beans"
119   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
120   xmlns:context="http://www.springframework.org/schema/context"
121   xsi:schemaLocation="http://www.springframework.org/schema/beans
122   http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
123   http://www.springframework.org/schema/context
124   http://www.springframework.org/schema/context/spring-context-2.5.xsd">
125
126 <context:component-scan base-package="p1" />
127
128 <bean id="myds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
129   <property name="driverClassName"><value>oracle.jdbc.driver.OracleDriver</value> </property>
130   <property name="url"> <value>jdbc:oracle:thin:@localhost:1521:orcl</value> </property>
131   <property name="username"> <value>scott</value> </property>
132   <property name="password"> <value>tiger</value>
133   </property>
134 </bean>
135
136 <bean id="sesfact" class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
```

recommended to
configure either
Apache DBCP (or)
C3P0 (or) server
managed Connection pool

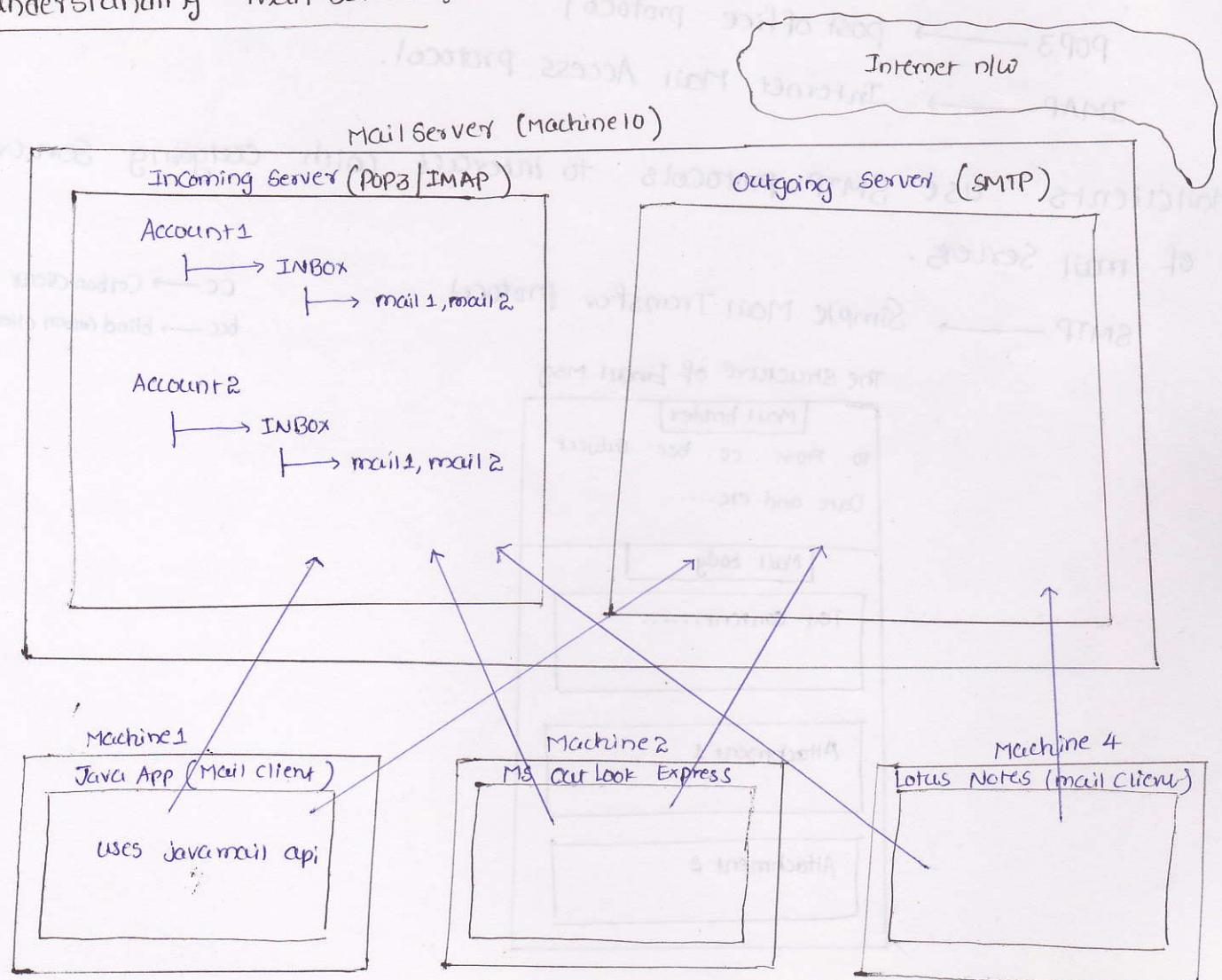
```
137     <property name="dataSource" ref="myds"/>
138         <!-- use this to avoid configuring mapping file-->
139     <property name="packagesToScan" value="p1"></property>
140 </bean>           makes the Spring container recognizing all the classes that are there in 'P' Pac
141                                     kages
142 </beans>
143 =====
144 SelectServlet.java
145 -----
146 package p1;
147
148 import javax.servlet.*;
149 import javax.servlet.http.*;
150 import java.io.*;
151 import org.hibernate.*;
152 import java.util.*;
153 import org.springframework.beans.factory.*;
154 import org.springframework.context.*;
155 import org.springframework.context.support.*;
156 public class SelectServlet extends HttpServlet
157 {
158     public void service(HttpServletRequest request,HttpServletResponse response)
159         throws ServletException,IOException
160     {
161
162         PrintWriter out=response.getWriter();
163         System.out.println("In service method of Servlet");
164         try{
165             ApplicationContext ctx=new ClassPathXmlApplicationContext("SpringHB.xml");
166             DemoInt d1=(DemoInt)ctx.getBean("d1");
167             Iterator i1=d1.getData();
168
169             out.println("<body bgcolor=#ffffcc text=red>");
170             out.println("<h1><center>all users</h1><hr><br><h3>");;
171             out.println("<table width=80% border=2>");;
172             while(i1.hasNext())
173             {
174                 User u1=(User)i1.next();
175                 out.println("<tr><td>"+u1.getId()+" <td>"+u1.getName()+" <td>"+u1.getRole()+"</tr>");;
176             }
177             out.println("</table>");;
178         }
179         catch(Exception e)
180         {
181             out.println(e);
182             e.printStackTrace();
183         }
184     }
185 }
186 =====
187 web.xml
188 -----
189 <web-app>
190     <servlet>
191         <servlet-name>select</servlet-name>
192         <servlet-class>p1.SelectServlet</servlet-class>
193     </servlet>
194
195     <servlet-mapping>
196         <servlet-name>select</servlet-name>
197         <url-pattern>/selectaction</url-pattern>
198     </servlet-mapping>
199
200 </web-app>
201 =====
```

In spring 3.1 `HibernateTemplate` class has been deprecated and removed. So, inject `HibernateSessionFactory` object to `SpringBean` while working with `Spring 3 & hibernate 4 combination`.

Java Mail

- DataBase s/w maintains records
- registry s/w maintains Objects
- mail server maintains email accounts and messages.
- java application uses uses jdbc api → DBs/w (oracle, sybase and etc...)
- java appn. Jndi api → Registry s/w (Weblogic, jboss registry and etc...)
- java appn. Java mail api → Mail server s/w (JRun, MailServer, Exchange Server, Lotus server and etc...)
- java mail api means javax.mail, javax.mail.internet pkgs (part of JEE module)
- DB s/w, Web Server s/w are not responsible to email accounts and email
- they are two types mail clients.
 - 1) Ready Made mail clients (MS outlook, MS outlook express, Lotus notes)
 - 2) Programmable Mail clients (App with Java mail api utilization)

Understanding mail Server :-



- Incoming Server is responsible to receive and manage email messages in the inboxes of email accounts.
- Outgoing Server is responsible to send email messages from one email account to another email account.
- POP3 incoming Server deletes the email messages from inbox once they are read by mail clients. So, it is useful for intranet mailing system.
- The IMAP incoming Server maintains the email messages in the inbox given after ^{they are} read from inbox, so, it is suitable for Internet based mailing operations.

Eg:- like www.GMail.com

- Mail clients use POP3 / IMAP protocols to interact with incoming Servers of mail servers.

POP3 → post office protocol

IMAP → Internet Mail Access protocol.

- Mail clients use SMTP protocols to interact with outgoing servers of mail servers.

SMTP → Simple Mail Transfer Protocol.

CC → Carbon Client
BCC → blind Carbon Client.

The structure of Email msg



James

→ Type : Mail Server

Version: 2.x

Vendor: Apache

default port no's

For admin console: 4555

For POP3 incoming server: 110

For SMTP outgoing server: 25

→ Installation: Extract the zip for installation of SW.

To start the mail server: <James-home>\bin\run.bat file.

Procedure to create email accounts manually in James mail server:-

Step 1:- Start the James server

Step 2:- use telnet ^{tool} to connect to the admin console of James server.

Start → run → telnet → open localhost 4555

Login id:

root

Password:

root

Welcome root.

help

adduser	Srikanth	→ username
		→ password
adduser	mani	mani

listusers

Countusers

deluser ravi

User ravi deleted

→ Session object of Java mail API represents the connectivity between java appn and mail server. To create this session obj we need mail properties and these property names are fixed but values will vary based on the mail server we use.

* for example application on Java mail API 60 Pg 61 page no's appn no:

→ In real weblevel mail applications the JavaMail API code will be return in Webresource programs like Servlet, JSP, to perform mail operations on email accounts.

Details of Gmail mail Server :-

Outgoing Server name : SMTP.Gmail.com

Port number of outgoing server : 465

Incoming Server name : POP.Gmail.com

Incoming Server port number : 995

for plain Java mail API based apps to interact with mail server of Gmail refer application (2) to appn (4) of the Supplementary Handout given on 26/08/2013

```

1 Java mail Apps
2 -App1-----SendAttachment.java----- (using plain java mail API )
3 /*
4  Java Application to send an E-mail with attachment
5 */
6 import javax.mail.*;
7 import javax.mail.internet.*;
8 import javax.activation.*;
9 import java.util.*;
10 public class SendAttachment
11 {
12     public static void main(String []args) throws Exception
13     {
14         //Creating Properties Objct
15         Properties properties=new Properties();
16
17         //adding protocol,mailserver address & the port number of the mailserver
18         properties.put("mail.transport.protocol","smtp");
19         properties.put("mail.smtp.host","localhost");
20         properties.put("mail.smtp.port","25");
21
22         //Creating the Session Object
23         Session session=Session.getInstance(properties);
24
25         //Creating and configuring the Message object
26         Message message=new MimeMessage(session);
27
28         message.setFrom(new InternetAddress("sachin1"));
29         InternetAddress address[]={new InternetAddress("dhoni1")};
30         message.setRecipients(Message.RecipientType.TO,address);
31         message.setSentDate(new Date());
32         message.setSubject("MAIL WITH ATTACHMENT");
33
34         // mail body obj
35         Multipart mailbody=new MimeMultipart(); → represents the empty body of the Email message
36
37         // mail body part1 obj (text content)
38         MimeBodyPart part1=new MimeBodyPart(); } adding part1 to message body.
39         part1.setText("hello see my resume");
40         mailbody.addBodyPart(part1);
41
42
43         //mail body part2(attachment)
44         FileDataSource fds=new FileDataSource(args[0]); //represents attachment file
45         MimeBodyPart part2=new MimeBodyPart();
46         part2.setDataHandler(new DataHandler(fds));
47         part2.setFileName(fds.getName());
48         mailbody.addBodyPart(part2); } adding part2 to message body
49
50         message.setContent(mailbody); → Setting body to message content
51
52         //Sending the mail.
53         Transport.send(message);
54     }//main
55 } //class
56 //>javac SendAttachment.java
57 //>java SendAttachment one.txt
58
59 // Prgs on plain java mail api (with Gmail Env.....)
60 -----App2-----SendMail.java-----
61 //Mail.java
62 import java.util.*;
63 import javax.mail.*;
64 import javax.mail.internet.*;
65
66 public class SendMail
67 {
68     public SendMail() {

```

add mail.jar file to the CLASSPATH.

```

69 // mail properties outgoing server (gmail.com)
70 Properties props = new Properties();
71 props.put("mail.smtp.host", "smtp.gmail.com"); → hostname
72 props.put("mail.smtp.port", "465"); → port number
73 props.put("mail.smtp.auth", "true");
74 props.put("mail.smtp.socketFactory.class", "javax.net.ssl.SSLSocketFactory");
75
76 try {
77 // create Session obj
78 Authenticator auth = new SMTPAuthenticator(); → Object creation for inner class
79
80 Session session = Session.getInstance(props, auth);
81
82 //prepare mail msg
83 MimeMessage msg = new MimeMessage(session);
84 //set header values
85 msg.setSubject("open to it know it"); testzava@gmail.com
86 msg.setFrom(new InternetAddress("satyaemail123@gmail.com"));
87 msg.addRecipient(Message.RecipientType.TO, new InternetAddress("nataraz@gmail.com"));
88 //msg text
89 msg.setText("mail from satya");
90
91 Transport.send(msg);
92 } //try
93 catch (Exception ex) {
94 ex.printStackTrace();
95 } //catch
96 } //constructor
97
98 private class SMTPAuthenticator extends javax.mail.Authenticator {
99     public PasswordAuthentication getPasswordAuthentication() {
100         return new PasswordAuthentication("satyaemail123@gmail.com", "javaj2ee"); testzava test.java
101     } //method
102 } //inner class
103 public static void main(String[] args){
104     SendMail mail=new SendMail();
105     System.out.println("mail has been delivered"); add mail.jar in the CLASSPATH.
106 } //main
107 } // Mail class
108 -----App3----- GetMail.java ----->java SendMail.java
109 import java.util.*;
110 import javax.mail.*;
111
112 public class GetMail {
113
114     String host="pop.gmail.com"; testzava@gmail.com
115     String emailfrom="satyaemail123@gmail.com";
116     String password="javaj2ee";
117     String port="995"; test.java
118
119     GetMail()
120     {
121         Properties props = new Properties();
122
123         props.put("mail.pop3.host", host);
124         props.put("mail.pop3.port", port);
125         props.put("mail.pop3.auth", "true");
126         props.put("mail.pop3.socketFactory.class", "javax.net.ssl.SSLSocketFactory");
127
128     try {
129         Authenticator auth = new SMTPAuthenticator();
130         Session session = Session.getInstance(props, auth);
131
132         Store store = session.getStore("pop3s"); → Connecting to pop3 incoming server
133         store.connect(host,emailfrom,password); → Connects to email ID
134
135         Folder folder = store.getFolder("inbox");
136

```

} inner class having authentication logic
(when u talk with mail servers this logic is must)


```

205 private class SMTPAuthenticator extends javax.mail.Authenticator {
206     public PasswordAuthentication getPasswordAuthentication() {
207         return new PasswordAuthentication(emailfrom, password);
208     }
209 } //private class
210
211 public static void main(String args[])
212     {
213         new GetMail();
214     }//main
215 }//class
216
217 Application sending sms.....
218 -----App5-----SendSMS.java
219 import java.io.*; → work with I/O stream
220 import java.net.InetAddress;
221 import java.util.Properties;
222 import java.util.Date;
223 import javax.mail.*;
224 import javax.mail.internet.*; } javamail API
225 import javax.activation.*;
226
227 public class SendSMS {
228
229     public SendSMS() {
230     }
231     //create an account on ipipi.com with the given username and password
232     public void msgsend() {
233         String username = "sathyasMS"; //Your Credentials
234         String password = "sathyas123"; zavaboss1
235         String smtphost = "ipipi.com";//Ip/Name of Server
236         String compression = "None"; //I dont want any compression
237         String from = "sathyasMS@ipipi.com";//u r userid@ipipi.com
238         //This mobile number need not be registered with ipipi.com
239         String to = "919861098610@sms.ipipi.com";//mobile number where u want to send sms
240         String body = "Hi This Msg is sent through Java Code";
241         Transport tr = null;
242
243         try {
244             Properties props = System.getProperties(); } adding new mail properties
245             props.put("mail.smtp.auth", "true"); } to the existing mail properties
246
247             // Get a Session object
248             Session mailSession = Session.getDefaultInstance(props, null);
249
250             // construct the message
251             Message msg = new MimeMessage(mailSession); → empty email msg
252
253             //Set message attributes
254             msg.setFrom(new InternetAddress(from)); → ipipi.com
255             InternetAddress[] address = {new InternetAddress(to)}; → mobile number
256             msg.setRecipients(Message.RecipientType.TO, address);
257             msg.setSubject(compression);
258             msg.setText(body); preparing email msg
259             msg.setSentDate(new Date());
260
261             tr = mailSession.getTransport("smtp"); → this transport object points to the outgoing server of ipipi
262             //try to connect
263             tr.connect(smtphost, username, password); → connects to outgoing server of ipipi
264             msg.saveChanges();
265             //send msg to all recipients
266             tr.sendMessage(msg, msg.getAllRecipients()); → sends the above created msg to the specified
267             tr.close(); recipients
268         } catch (Exception e) {
269             e.printStackTrace();
270         }
271     }
272     public static void main(String[] args) {

```

```

273     SendSMS sms = new SendSMS();
274     sms.msgSend();
275     System.out.println("Successfull");
276   }
277 }
278
279 App6>>>>>> Spring Mail with Attachments <<<<<<<<<
280 ----- Student.java (Spring Interface)
281 public interface Student {
282     public void sendEmailMsg();
283 }
284 ----- StudentImpl.java (Spring Bean)
285 import org.springframework.mail.*;
286 import org.springframework.mail.javamail.*;
287 import javax.mail.*; Using plain Java mail API core Base Interface so, we use plain Java mail API
288 import javax.mail.internet.*;
289 import org.springframework.core.io.*;
290
291 public class StudentImpl implements Student
292 {
293     private JavaMailSenderImpl mailSender;
294
295     public void setMailSender(JavaMailSenderImpl mailSender) { } To inject JavaMailSenderImpl object through Setter Injection
296         this.mailSender = mailSender;
297 }
298
299     public void sendEmailMsg()
300     {
301         try{
302             mailSender.send(new MimeMessagePreparator() {
303
304             Here send() is called . public void prepare(MimeMessage mimeMessage) throws MessagingException {
305             having Anonymous Inner class obj that implements MimeMessageHelper message=new MimeMessageHelper(mimeMessage, true, "UTF-8");
306             message.setFrom("advani"); sachin
307             message.setTo("kalami"); dhoni
308             message.setSubject("my subject");
309             message.setText("hello This mail Content/body");
310             message.addAttachment("abc.txt", new ClassPathResource("abc.txt"));
311             message.setPriority(Priority); 1
312             message.setRecipients(RecipientType.ALL, Address[]); multiple parts in the message
313             message.setSentDate(Date); when prepared
314             prepare()); });
315         } //try
316         catch(MailException me) {
317             me.printStackTrace();
318         } //sendEmailMsg
319
320     } //class
321 ----- mailcfg.xml (Spring Cfg file)
322 <?xml version="1.0" encoding="UTF-8"?>
323 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
324 "http://www.springframework.org/dtd/spring-beans.dtd">
325 <beans>
326     represent <bean id="ms" class="org.springframework.mail.javamail.JavaMailSenderImpl">
327         bean <property name="host"><value>localhost</value></property>
328         property <property name="port"><value>25</value></property>
329         <property name="protocol"><value>smtp</value></property>
330     </bean>
331
332     <bean id="sti" class="StudentImpl">
333         <property name="mailSender"><ref bean="ms"/></property>
334     </bean>
335 </beans>
336 ----- MailClient.java (Client appn)
337 import org.springframework.beans.factory.*;
338 import org.springframework.context.*;
339 import org.springframework.context.support.*;
340 public class MailClient

```

add mail.jar file to the classpath.

> Javac SendSMS.java

> java SendSMS

Using plain Java mail API core Base Interface so, we use plain Java mail API

to allocate multiple parts in the message

refer line no: 326

1) To run this application keep James mail Server in Running mode.

2) Add following jar files in the CLASSPATH

spring.jar, mail.jar, commons-logging.jar

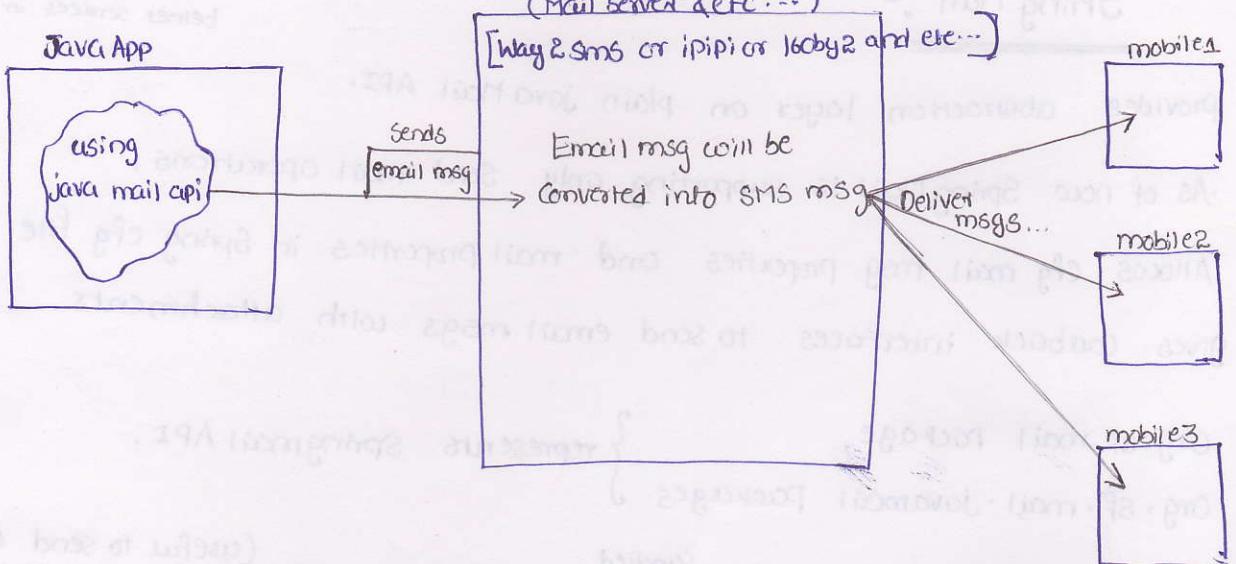
```
341 {  
342     public static void main(String args[])  
343     {  
344         ApplicationContext ctx=new FileSystemXmlApplicationContext("mailcfg.xml");  
345         Student beanref=(Student)ctx.getBean("sti");  
346         beanref.sendEmailMsg();  
347         System.out.println("mail has been delivered");  
348     }  
349 }  
350  
351 }
```

Spring Mail :-

telnet services in Windows 7

- provides abstraction layer on plain javaMail API.
 - As of now SpringMail is supporting only send mail operations.
 - Allows cfg-mail msg properties and mail properties in Spring cfg file declaratively.
 - gives callback interfaces to send email msgs with attachments.
 - org.sf.mail package, org.sf.mail.javamail packages } represents SpringMail API.
 - org.sf.mail represents Spring's direct facilities to perform mail operations
 - org.sf.mail.javamail represents Spring's abstraction layer on plain javaMail API
 - (useful to send email messages)
 - * For example application on sending email message by using SpringMail
refer appn no: (14) of the page no's 61 & 62.
 - To send email msgs with attachment by using SpringMail API we need to work with the callback interface MimeMessagePreparator() & one concrete class called MimeMessage Helper
 - * For example application on SpringMail based send mail operation with attachment refer appn (6) of the given handout 26/03/2013

Way2SMS, IPIPI, 160by2 and etc... are the SMS Gateways. These Gateways can receive messages as email messages and delivers these messages to specific phone numbers as SMS messages.



(*) procedure to Send SMS msgs to mobile numbers by using JavaMail API appn

Step1:- Create one Account in IPIPI.com.

Step2:- Username : zavarboss

password : zavarboss1

Step3:- Run the Send SMS application (refer : supplementary handout given on 26/03/13)

procedure to access the webapplication of Tomcat Server with domain name.

Step1:- specify domain name in built-in DNS registry of windows Xp/7

goto C:\Windows\system32\drivers\etc\hosts file and replace 127.0.0.1 localhost
to 127.0.0.1 www.nataraz.in

Step2:- change the Tomcat Server port number to "80"

If port number is not specified in Http request (e.g.) then the protocol http tries to send the request to a Service whose port number is "80".

Add following code in server.xml file under <Engine> tag specifying the domain name and webapplication name

103line

```

<Host name="www.naturaz.in" domainName="www.naturaz.in" appBase="webapps"
    unpackWARs="true" autoDeploy="true">
    <Context path="" docBase="VoterApp" />
</Host>
```

Step 3:- Deploy the above specified VoterApp webapp in tomcatServer.

make sure that welcomefile is configured in that webapp.

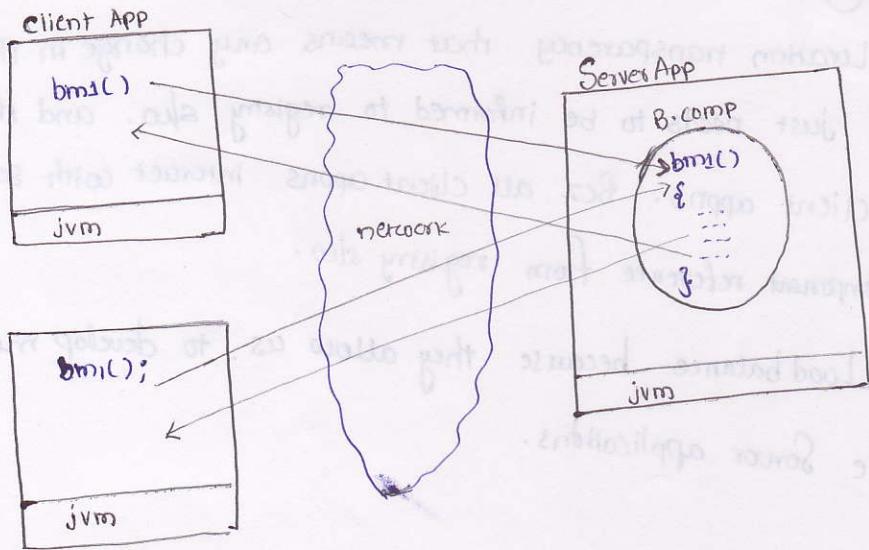
Step 4:- Restart the Server once.

Step 5:- Test the application

Open browser window and type this url <http://www.naturaz.in>

Understanding the need of Distributed Application Development :-

Traditional client-server appn



Traditional client-server applications (Socket Programming Based) are location dependent.

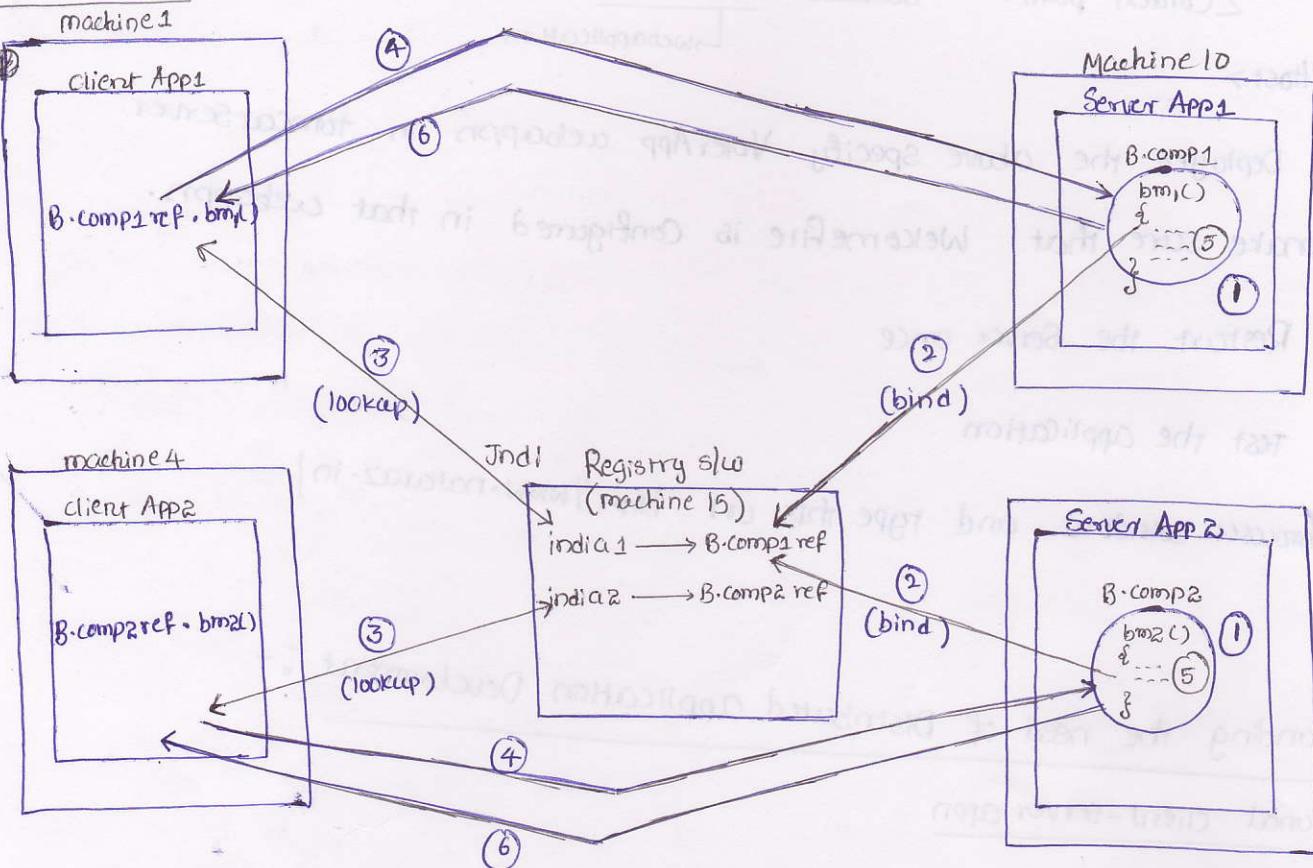
that means any change in the location of server application we must modify code in the client appns. This indicates client and server appns are tightly-coupled appns (hard-coded).

If degree of dependency is more b/w two appns then they are called

tightly coupled appns. Otherwise they are called loosely-coupled appns.

To develop the ~~apps~~ as Client-Server Apps as Location transparent Apps then develop them as distributed applications.

Distributed Appn :-



Distributed apps support Location transparency that means any change in the location of Server application just needs to be informed to registry ser. and there is no need of change in coding client apps. Bcz all client apps interact with server apps by getting Business Component reference from registry ser.

Distributed apps support Load balance because they allow us to develop multiple Business Components in multiple Server applications.

W.r.t. the diagram :-

- (1) multiple Server applications will be developed having multiple B.components.
- (2) these multiple B.components references will be registered with registry ser.s for global visibility.
- (3) Client apps gather B.component references from registry ser.
- (4), (5), (6) client app calls B.methods of B.component and these B.methods of Server app will execute & final result goes back to client app.

To develop Distributed appns we need the Support of distributed technologies.

Eg:- RMI, EJB, web services, HttpInvoker, CORBA and etc....

→ Spring JEE provides abstraction layer on existing distributed technologies like RMI, EJB, WebServices and also provides its own distributed technology called HttpInvoker.

→ In Distributed App Development 4 parties are involved.

① Service Interface / Business Interface → (as a common understand document b/w Server provider Apps and Server Client appn)
→ Declaration of b.methods

② Server Provider App
(The Server App that contains b.methods in B.Component)

③ Registry Slw
(makes b.comp references as globally visible objs and also makes Distributed Apps as location transparent appns)

④ Service Client App
(calls the b.methods of B.Components.)

plain RMI :-

→ It is built-in Distributed Technology of JDK

→ uses Rmi registry as Registry Slw

→ Core pkgs of rmi api are: java.rmi, java.rmi.server Pkgs

Q:- What is the Difference b/w Local client and Remote client?

Ans:- If the appn and its client resides on same JVM then it is called Local client to appn.

If appn and its client resides in two different JVMs of same or different Computers then it is called Remote client to appn.

Q:- What is the Diff b/w Local object and Remote Object?

Ans:- If the methods of the object can be called only from Local clients then it is called Local Object.

If the methods of the object can be called from both Local, Remote clients then it is called Remote Object. Object becomes RemoteObject only when its

class implements `java.rmi.Remote` (I).

→ While developing distributed apps all the business objects must be developed as `Remote object`.

⑥ Example application on plain rmi

`Fact.java` (Service Interface)

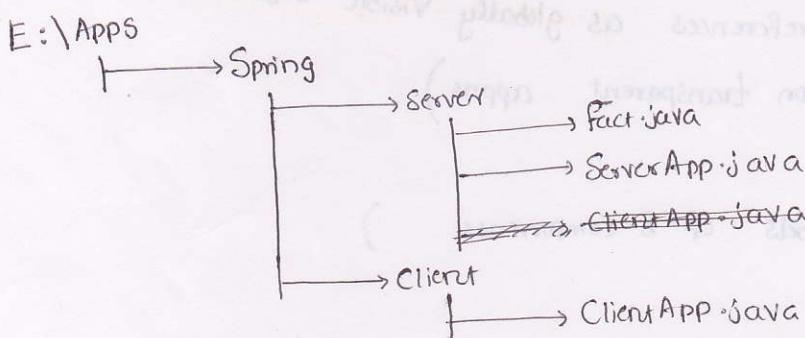
`ServerApp.java` (Service Provider App)

`ClientApp.java` (Service Client App)

Rmi registry (Registry s/w).

In one computer multiple JVMs can be there to run multiple java apps

Simultaneously (or) Concurrently



//`Fact.java` (Service Interface)

```
import java.rmi.*;  
public interface Fact extends Remote  
{  
    //declare B.methods  
    public long findFactorial(int val) throws RemoteException;
```

}

}

//`ServerApp.java`

```
import java.rmi.*;  
import java.rmi.server.*;  
class FactImpl extends UnicastRemoteObject implements Fact  
{  
    //impl gives B.complobj  
    class ServerApp extends Server {  
        FactImpl factImpl = new FactImpl();  
        public FactImpl() throws RemoteException {  
            S.O.P("factImpl");  
        }  
        public long findFactorial(int val)  
        {  
            return factImpl.findFactorial(val);  
        }  
    }  
}
```

extends `UnicastRemoteObject` implements `Fact`
↳ To make this Business Object registerable in Registry s/w

```

        long res = 1;
        for (int i=1; i<=val; ++i)
        {
            res = res * i;
        }
        return res;
    } // findFactorial
} // class

public class ServerApp
{
    public void (String args[]) throws Exception
    {
        FactImpl obj = new FactImpl(); // Business obj
        // bind B.obj with registry slot
        Naming.bind("india", obj); // impossible also Naming.bind("rmi://localhost:1099/india", obj)
        System.out.println("Server App is ready....");
    }
} // ClientApp.java

import java.rmi.*;

public class ClientApp
{
    public void (String[] args) throws Exception
    {
        // get B.obj ref from registry through lookup operation
        Fact bobj = (Fact) Naming.lookup("india"); // fact bobj=(Fact) Naming.lookup("rmi://localhost:1099/india")
        // call B.methods
        System.out.println("Client App: " + bobj.findFactorial(5));
    }
} // main
} // class

```

Compile .java files.

E:\Apps\Spring\Server> javac *.java

Copy Fact.class file to Client folder.

E:\Apps\Spring\Client> javac *.java

Generate Stub file Built-in Jdk tool

E:\Apps\Spring\Server> rmic FactImpl classname of B.obj

gives FactImpl-Stub.class

Copy Stub file to client folder

Start rmi registry from Server folder

E:\Apps\Spring\Server> rmiregistry

The default port number of
rmi registry is 1099

Run the Server application

E:\Apps\Spring\Server> java ServerApp

Run the Client application

E:\Apps\Spring\Client> java ClientApp

To change Rmi registry port number :-

Syntax :-

>rmiregistry <newportno>

Eg:- >rmiregistry 2000

- The process of converting Java notation Data to Network notation data is called marshalling. and reverse is called Unmarshalling.
- The stub at client side & at server side performs these marshalling & unmarshalling operations.
- Stub acts as Client Side proxy for client appn & Server Side proxy for server appn. Bcz, these two proxies will communicate with each other on behalf of on rmi client & server side appns.

: Spring rmi :-

- provides abstraction layer on plain rmi
- allows to develop server & client appns as Spring appns.
- There is no need of working with plain rmi api.
- org.springframework.rmi pkg represents Spring rmi api.
- There is no need of performing exception handling in Spring rmi appns.
- There is no need of starting rmi registry explicitly.
- can work with Spring Container Started rmi registry or can work with explicitly started rmi registry.

→ There is no need of generating stub files explicitly

→ in Spring rmi appn the Obj that is registered with rmi registry is identified through two items
① nickname/alias name ② ServiceInterface name

→ allows to keep business Object reference in rmi registry through dependency injection process and allows to get B.obj ref from rmi registry through dependency injection process.

→ allows to develop the resources as pojo's & pojo's

→ allows to specify rmi registry related cfgs in xml file declaratively.

org. sf. remoting. rmi. RmiServiceExporter can expose given Business object to Rmi having Specified name

org. sf. remoting. rmi. RmiProxyFactoryBean → gives Business obj ref gathered from rmi registry through look operation

* For example application on SpringRmi based Distributed appn refer pgno's: 62-66 app (15)

→ While developing plain RMI client appn we need to gather following details from Server application.

a) nickname given to rmi B.object.

b) a copy of Service interface, stub files

c) Url to perform Lookup operation on rmi registry like (host name, port number)

d) Documentation about Business methods

→ While developing SpringRMI client appn we need following details from Server appn.

→ Same as above details but Stub file is not required.

Plain rmi Server Appn ← (Yes) Plain rmi Client Appn

Spring rmi Server Appn ← (Yes) Spring rmi Client Appn

Plain rmi Server Appn ← (Yes) → Spring rmi Client Appn

[Stub file should copied to Client appn side]

Spring rmi Server Appn ← (No) → Plain rmi Client Appn

[Plain rmi Client Appn need stub file but Spring rmi Server Appn does not generate this Stub file explicitly]

→ In real world all Credit card and Debit Card component processing will developed as Distributed Components / Applications.

for example application on SpringRMI using Spring ~~DA~~^{DAO} module refer appn 16 of the page no's 64 & 65

While developing logics in SpringRMI Server application we can take the Support of Spring CRM, DAO modules while developing Business Logics.

Drawbacks of RMI :-

- ① RMI is Language dependent (That means both client and Server appn must be developed in Java Language.)
- ② RMI appns can't use Internet network as the Communication channel.
- ③ Applying middleware Services on the BusinessLogics of Rmi appns is very difficult.
- ④ RMI registry is a weak registry to handle huge no.of bind & lookup operations.
To overcome these problems use Other Distributed Technologies like EJB, HttpInvoker, Webservices and etc... EJB, HttpInvoker are language dependent. Webservices is language independent.

- Spring EJB provides abstraction Layer on plain EJB. but abstraction layer can not be used in the development of EJB component that is just useful in the development of EJB client App development (so spring developers never thinks about Spring EJB).
- Spring Webservices provides abstraction Layer on both server and client App development.

HTTP INVOKER

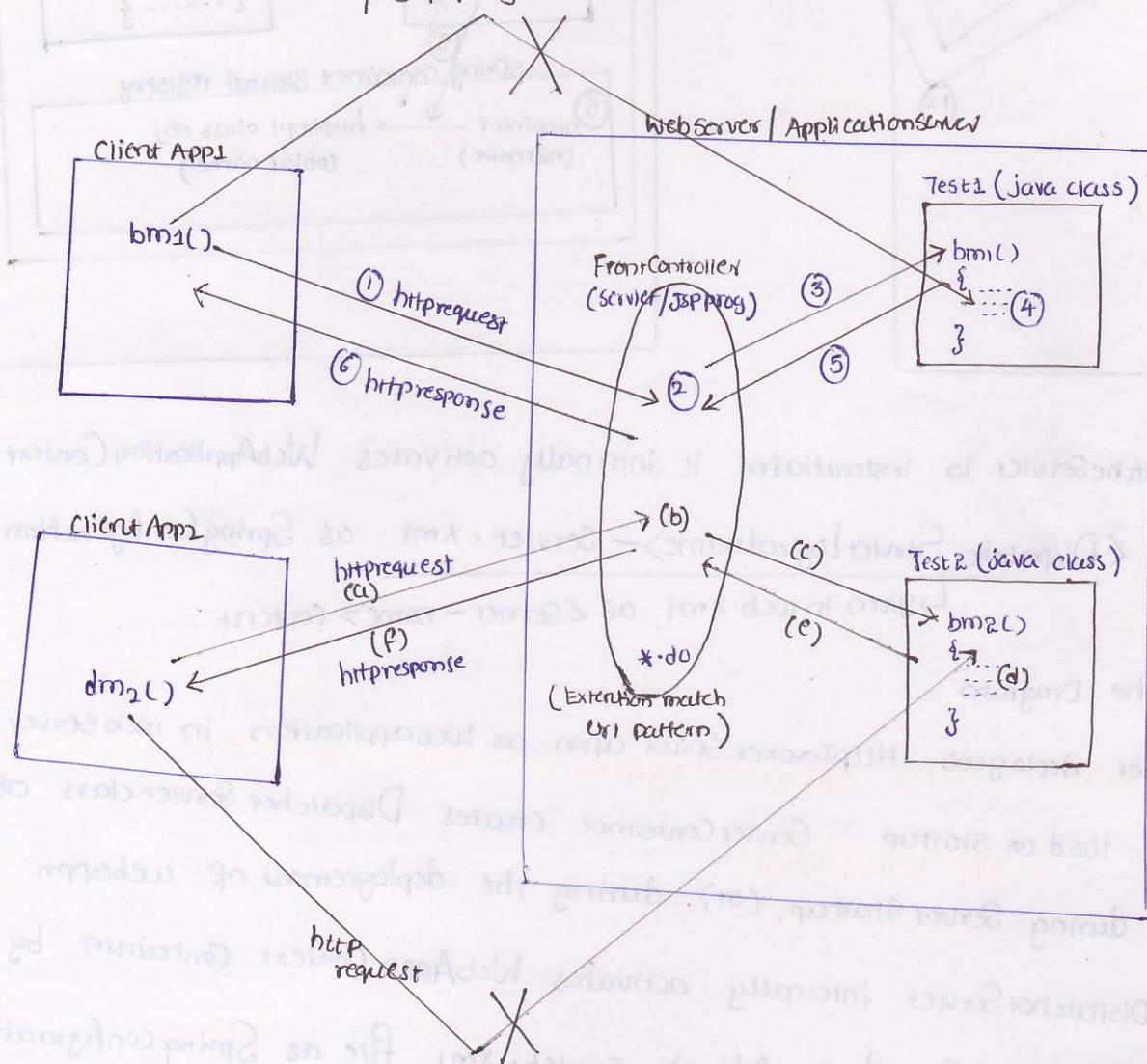
- It is spring's own web based distributed technology
- Spring Container acts as Registry Svc
- Here Server appn is nothing but web application deployed in the Server
- Here Client appn can be stand alone appn or webapp
- org. sf. remoting. httpinvoker. HttpInvokerServiceExporter keeps B.Objs in the registry (Spring Container) having Service interface name as nick name (or) alias name.
- org. sf. remoting. httpinvoker. HttpInvokerProxyFactory Bean gets B.Objs reference from registry

- This technology is also language dependent, both server and applications must be written in Java Language.

→ The Java classes of Webapplication can't take Http request direct from client and they can't generate Http response direct to client.

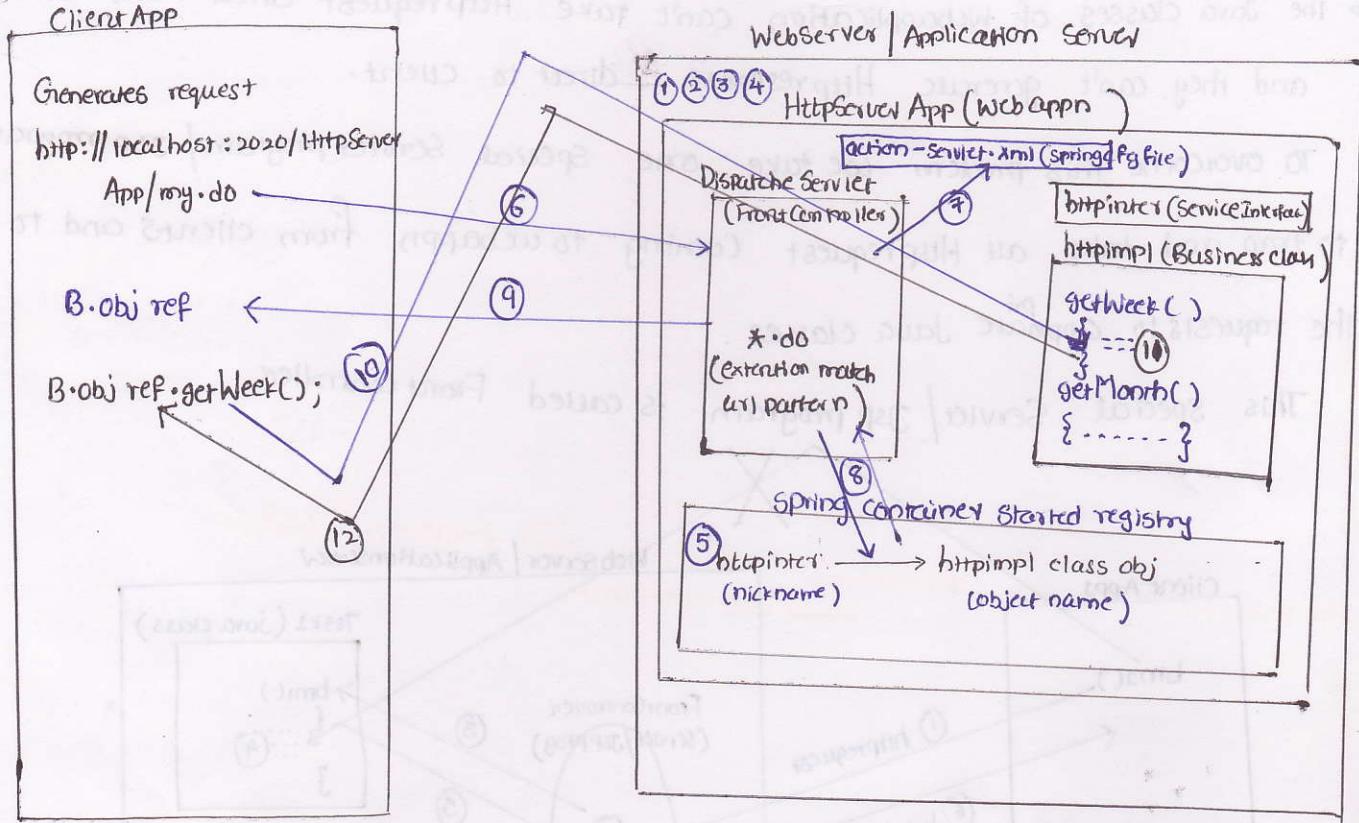
To overcome this problem we take one special Servlet program / JSP program to trap and take all Httprequest coming to webapp from clients and to pass the requests to appropriate Java classes.

This special Servlet / JSP program is called Front Controller.



- FrontController is a special webresource program of a web server that acts as entry & exit point handling all the HttpServletRequest & HttpServletResponse of ordinary java classes of web appn (refer above diagram)
- In Struts appn ActionServlet is Front Controller Servlet. In JSF application FacesServlet is FrontControllerServlet. In spring appns org.springframework.web.servlet.DispatcherServlet is Front Controller Servlet.

→ a Front Controller Servlet must be configured in web.xml file having extension match or Directory match url pattern to trap and take multiple request.



When DispatcherServlet is instantiated it internally activates WebApplicationContext Container

by taking `<DispatcherServletLogicalName>-servlet.xml` as Spring Configuration file name
↳ given in `web.xml` as `<Servlet-name>` content

W.r.t to the Diagram

- ① programmer deploys `HttpInvoker Server Appn` as Webapplication in webServer
- ② Bcz of load on startup ServerContainer creates DispatcherServlet class object either during Server Startup (or) during the deployment of webappn.
- ③ This DispatcherServlet internally activates `WebAppn Context Container` by taking `<DispatcherServletLogicalName>-servlet.xml` file as Spring Configuration file.
- ④ This Spring Container performs pre-instantiation on all the Spring Beans of `SpringFg` file.
- ⑤ In the process of preInstantiation the `HttpInvokerServiceExporter` keeps Business Class object in registry also having ServiceInterface name as nickname/alias name.
- ⑥ Client appn uses some request Url and `HttpInvoker` Bean to send request to Server application.

- ⑦ As a FrontController the DispatcherServlet traps and takes the request and passes the request to SpringContainer. and this Container uses SimpleUriHandlerMapping class to bind the request with B-object.
- ⑧ Based on the above activities the DispatcherServlet gathers B-object reference from Registry also.
- ⑨ Client appn gets B-object reference from Server appn.

- ⑩ Client appn calls business method on Business Object reference.

- ⑪ Logic of B-method executes in Server appn

- ⑫ The results generated by B.method comes to client application.

- * For example appn on HttpInvoker appn no: 17 pageno 65, 66 & 67 based Distributionappn

→ A Distributed application can be a Webapp or Non-webapp.

→ Webapp that uses registry supporting LocationTransparency and allows programmable client appns to call the B-methods. is called Distributed appn.

→ HttpInvoker, WebServices are the Webbased distributed appns.

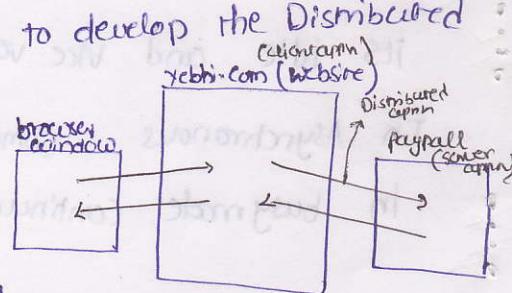
→ Normal Servlet, JSP webappns are not at all Distributed appns.

The client appn developer of HttpInvoker should gather following details from the Server appn of HttpInvoker.

- a copy of ServiceInterface / BusinessInterface
- request url to get Business Object reference
- Documentation about Business method.

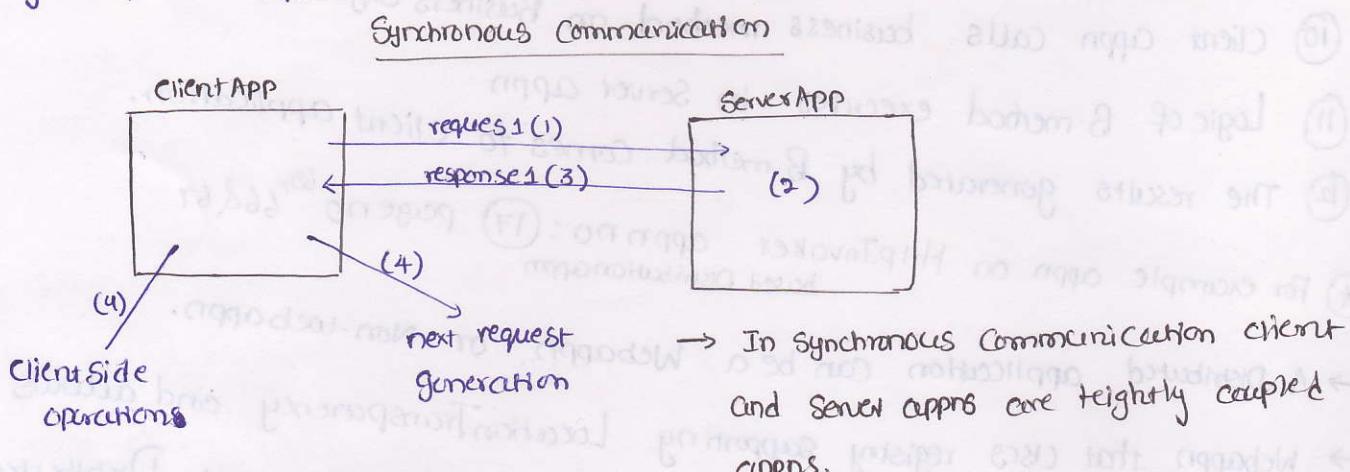
Server appn developer can send this details to client APP Developers through mail or as a content to download.

→ Industry prefers WebServices (or) SpringWebServices to develop the Distributed appn. Otherwise they prefer using HttpInvoker to develop Spring based Distributed appns. Spring RMI is not industry standard Distributed Technology to develop the Distributed appns.



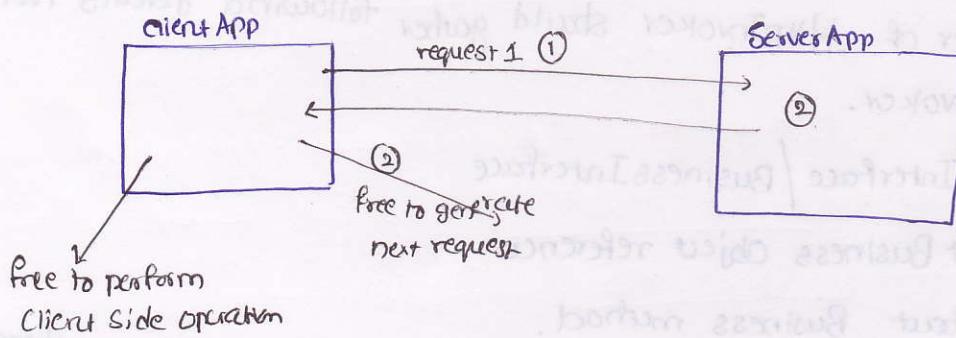
JMS (Java Message Service) :-

- If client is blocked or not free to generate next request (or) to perform client side operations until given request related response comes to client from Server, then it is called Synchronous Communication.
- In Synchronous Communication Client apps and Server apps are tightly coupled apps



→ In Synchronous Communication Client and Server apps are tightly coupled apps.

- If client app needs to perform client side operations and to generate next request without waiting for given request related response from Server app then it is called "Asynchronous Communication."



- In Asynchronous Communication Client and Server apps are loosely coupled apps.
- In Synchronous Communication if client app is busy the Server app is idle and vice versa.

In Asynchronous Communication both Client and Server apps reside in busy mode continuously.

What Is Messaging?

Messaging is a method of communication between software components or applications. A messaging system is a peer-to-peer facility: A messaging client can send messages to, and receive messages from, any other client. Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.

Messaging enables distributed communication that is *loosely coupled*. A component sends a message to a destination, and the recipient can retrieve the message from the destination. However, the sender and the receiver do not have to be available at the same time in order to communicate. In fact, the sender does not need to know anything about the receiver; nor does the receiver need to know anything about the sender. The sender and the receiver need to know only what message format and what destination to use. In this respect, messaging differs from tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application's methods.

Messaging also differs from electronic mail (e-mail), which is a method of communication between people or between software applications and people. Messaging is used for communication between software applications or software components.

What Is the JMS API?

The Java Message Service is a Java API that allows applications to create, send, receive, and read messages. Designed by Sun and several partner companies, the JMS API defines a common set of interfaces and associated semantics that allow programs written in the Java programming language to communicate with other messaging implementations.

The JMS API minimizes the set of concepts a programmer must learn to use messaging products but provides enough features to support sophisticated messaging applications. It also strives to maximize the portability of JMS applications across JMS providers in the same messaging domain. The JMS API enables communication that is not only loosely coupled but also

Asynchronous. A JMS provider can deliver messages to a client as they arrive; a client does not have to request messages in order to receive them.

Reliable. The JMS API can ensure that a message is delivered once and only once. Lower levels of reliability are available for applications that can afford to miss messages or to receive duplicate messages.

The JMS Specification was first published in August 1998. The latest version of the JMS Specification is Version 1.0.2b, which was released in August 2001. You can download a copy of the Specification from the JMS Web site, <http://java.sun.com/products/jms/>.

The JMS API in the J2EE platform has the following features.

Application clients, Enterprise JavaBeans (EJB) components, and web components can send or synchronously receive a JMS message. Application clients can in addition receive JMS messages asynchronously. (Applets, however, are not required to support the JMS API.) Message-driven beans, which are a kind of enterprise bean, enable the asynchronous consumption of messages. A JMS provider can optionally implement concurrent processing of messages by message-driven beans.

Message send and receive operations can participate in distributed transactions, which allow JMS operations and database accesses to take place within a single transaction.

Messaging Domains

Before the JMS API existed, most messaging products supported either the *point-to-point* or the *publish/subscribe* approach to messaging. The JMS specification provides a separate domain for each approach and defines compliance for each domain. A stand-alone JMS provider can implement one or both domains. A J2EE provider must implement both domains.

In fact, most implementations of the JMS API support both the point-to-point and the publish/subscribe domains, and some JMS clients combine the use of both domains in a single application. In this way, the JMS API has extended the power and flexibility of messaging products.

Point-to-Point Messaging Domain

A point-to-point (PTP) product or application is built on the concept of message queues, senders, and receivers. Each message is addressed to a specific queue, and receiving clients extract messages from the queues established to hold their messages. Queues retain all messages sent to them until the messages are consumed or until the messages expire.

Telephone connection with Answering machine
Setup is the best example to understand
PTP messaging Domain behaviour

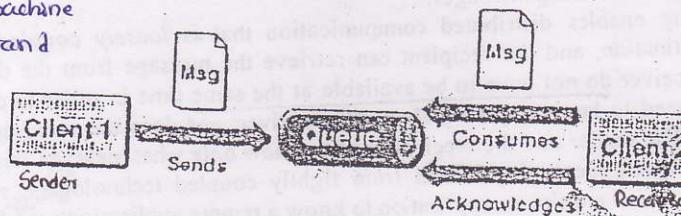


Figure 33-3 Point-to-Point Messaging

PTP messaging has the following characteristics

1. Each message has only one consumer.
2. A sender and a receiver of a message have no timing dependencies. The receiver can fetch the message whether or not it was running when the client sent the message.
3. The receiver acknowledges the successful processing of a message.

Use PTP messaging when every message you send must be processed successfully by one consumer.

Publish/Subscribe Messaging Domain

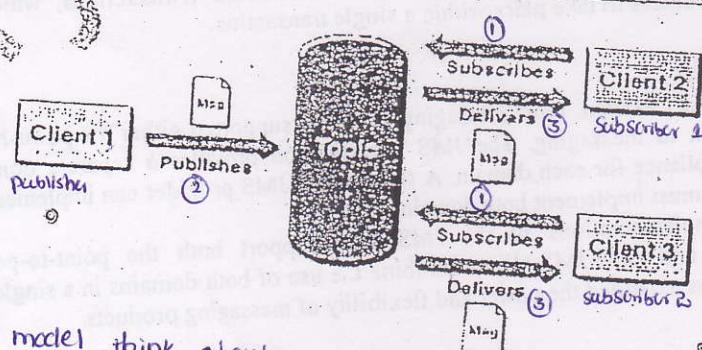
In a publish/subscribe (pub/sub) product or application, clients address messages to a *topic*, which functions somewhat like a bulletin board. Publishers and subscribers are generally anonymous and can dynamically publish or subscribe to the content hierarchy. The system takes care of distributing the messages arriving from a topic's multiple publishers to its multiple subscribers. Topics retain messages only as long as it takes to distribute them to current subscribers.

Pub/sub messaging has the following characteristics.

1. Each message can have multiple consumers.
2. Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.

The JMS API relaxes this timing dependency to some extent by allowing subscribers to create *durable subscriptions*, which receive messages sent while the subscribers are not active. Durable subscriptions provide the flexibility and reliability of queues but still allow clients to send messages to many recipients.

Use pub/sub messaging when each message can be processed by zero, one, or many consumers.



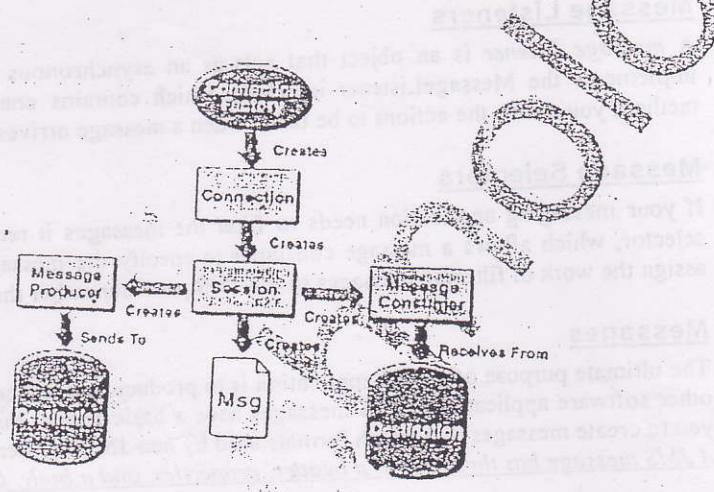
To understand pub/sub model think about Satellite TV channel broadcasting System.

Message Consumption

Messaging products are inherently asynchronous: There is no fundamental timing dependency between the production and the consumption of a message. However, the JMS specification uses this term in a more precise sense. Messages can be consumed in either of two ways:

Synchronously: A subscriber or a receiver explicitly fetches the message from the destination by calling the receive method. The receive method can block until a message arrives or can time out if a message does not arrive within a specified time limit.

Asynchronously: A client can register a *message listener* with a consumer. A message listener is similar to an event listener. Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's *onMessage* method, which acts on the contents of the message.



Administered Objects

Two parts of a JMS application—destinations and connection factories—are best maintained administratively rather than programmatically. The technology underlying these objects is likely to be very different from one implementation of the JMS API to another. Therefore, the management of these objects belongs with other administrative tasks that vary from provider to provider.

Connection Factories

A *connection factory* is the object a client uses to create a connection to a provider. A connection factory encapsulates a set of connection configuration parameters that has been defined by an administrator. Each connection factory is an instance of the *ConnectionFactory*, *QueueConnectionFactory*, or *TopicConnectionFactory* interface.

Destinations

A *destination* is the object a client uses to specify the target of messages it produces and the source of messages it consumes. In the PTP messaging domain, destinations are called queues. In the pub/sub messaging domain, destinations are called topics.

Connections

A *connection* encapsulates a virtual connection with a JMS provider. A connection could represent an open TCP/IP socket between a client and a provider service daemon. You use a connection to create one or more sessions.

Sessions

A *session* is a single-threaded context for producing and consuming messages. You use sessions to create message producers, message consumers, and messages.

Message Producers

A *message producer* is an object that is created by a session and used for sending messages to a destination. It implements the MessageProducer interface.

Message Consumers

A *message consumer* is an object that is created by a session and used for receiving messages sent to a destination. It implements the MessageConsumer interface.

A message consumer allows a JMS client to register interest in a destination with a JMS provider. The JMS provider manages the delivery of messages from a destination to the registered consumers of the destination.

Message Listeners

A *message listener* is an object that acts as an asynchronous event handler for messages. This object implements the MessageListener interface, which contains one method, onMessage. In the onMessage method, you define the actions to be taken when a message arrives.

Message Selectors

If your messaging application needs to filter the messages it receives, you can use a JMS API message selector, which allows a message consumer to specify the messages it is interested in. Message selectors assign the work of filtering messages to the JMS provider rather than to the application.

Messages

The ultimate purpose of a JMS application is to produce and to consume messages that can then be used by other software applications. JMS messages have a basic format that is simple but highly flexible, allowing you to create messages that match formats used by non-JMS applications on heterogeneous platforms.

A JMS message has three parts: a header, properties, and a body. Only the header is required.

Message Headers

A JMS message header contains a number of predefined fields that contain values that both clients and providers use to identify and to route messages.

Message Properties

You can create and set properties for messages if you need values in addition to those provided by the header fields. You can use properties to provide compatibility with other messaging systems, or you can use them to create message selectors.

Message Bodies

The JMS API defines five message body formats, also called message types, which allow you to send and to receive data in many different forms and provide compatibility with existing messaging formats.

Message Type	Body Contains
TextMessage	A java.lang.String object (for example, the contents of an Extensible Markup Language file).
MapMessage	A set of name-value pairs, with names as String objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
BytesMessage	A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.
StreamMessage	A stream of primitive values in the Java programming language, filled and read sequentially.
ObjectMessage	A Serializable object in the Java programming language.
Message	Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.

```

1-----Plain JMS applications-----
2 (on Queue) (App1)
3 -----MyQueueSender.java (sender appn)
4 import javax.jms.*;
5 import javax.naming.*;
6 import java.util.Properties;
7
8
9 public class MyQueueSender
10 {
11     public static void main(String[] args) throws Exception
12     {
13         Properties prop=new Properties(); "com.sun.enterprise.naming.SerialInitContextFactory"
14         prop.put(Context.INITIAL_CONTEXT_FACTORY,"weblogic.jndi.WLInitialContextFactory");
15         prop.put(Context.PROVIDER_URL,"t3://localhost:7001");
16         iop://localhost:4646
17         InitialContext ic=new InitialContext(prop);
18         QueueConnectionFactory qcf=(QueueConnectionFactory)ic.lookup("ConFactJndi");
19         QueueConnection qconn=qcf.createQueueConnection(); → disableS → Jndi name given to JMS Connection
20         QueueSession qsession=qconn.createQueueSession(false,Session.AUTO_ACKNOWLEDGE); → Transaction
21         ↳ creates SessionObject as base to perform any JMS operations.
22         Queue queue=(Queue)ctx.lookup("QueueJndi"); → points to the Queue destination of GlassFish server
23         QueueSender qsender=qsession.createSender(queue); → creates Sender pointing to Queue destination
24
25         TextMessage message; 25-50:- creates 20 text messages having 20 different text contents
26         String text; as bodies. but every 5 messages contains same value
27         int k=1; in name property.
28
29         for(int i=1;i<=4;i++)
30         {
31             if(i==1)
32                 text="Hotmail";
33             else if(i==2)
34                 text="Amazon";
35             else if(i==3)
36                 text="JGuru";
37             else
38                 text="Bazee";
39
40             for(int j=1;j<=5;j++)
41             {
42                 message=qsession.createTextMessage();
43                 message.setStringProperty("Name",text);
44                 message.setText("This is Item "+k+"....."+text);
45                 System.out.println("Sending "+k+"\t"+text);
46                 qsenter.send(message);
47                 k++;
48             } //for
49         } //for
50
51         qsession.close();
52         qconn.close();
53     } >javac MyQueueSender.java
54 } >java MyQueue Sender
55 -----MyQueueReceiver.java
56
57 import javax.jms.*;
58 import javax.naming.*;
59 import java.util.Properties;
60
61 public class MyQueueReceiver implements MessageListener
62 {
63
64     public static void main(String args[]) throws Exception
65     {
66         Properties prop=new Properties(); → receive messages from Queue destination based on
67         prop.put(Context.INITIAL_CONTEXT_FACTORY,"weblogic.jndi.T3InitialContextFactory"); Event Handling
68         prop.put(Context.PROVIDER_URL,"t3://localhost:7001"); (receiving the messages Asynchronously)
69     } (x1) change these Jndi properties to GlassFish properties
70
71     InitialContext ic=new InitialContext(prop);
72     QueueConnectionFactory qcf=(QueueConnectionFactory)ic.lookup("ConFactJndi");
73
74     QueueConnection qconn=qcf.createQueueConnection();

```

```

76     QueueSession qsession=qconn.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
77
78     Queue queue=(Queue)ctx.lookup("QueueJndi");
79
80     String selector="Name IN('Bazee','Hotmail')"; → Message Selector to filter the messages
81     QueueReceiver qreceiver=qsession.createReceiver(queue,selector);
82
83     MyQueueReceiver mq=new MyQueueReceiver(); → current class obj
84     qreceiver.setMessageListener(mq); → creating receiver pointing to Queue destination
85
86     qconn.start(); → registering message Listener with our current class object
87
88     } → starts listening to message store to receive the messages.
89
90     public void onMessage(Message msg) → represents received msg each time
91     {
92     try{ → EventHandling method that executes when each message comes to
93     ? prints the message store
94     System.out.println("Received Message is:"+tmsg.getText());
95     }catch(Exception e){ → S-o-f("message property "+tmsg.getStringProperty("uname"));
96     received { → "uname"
97     e.printStackTrace();
98     }
99     }
100    }

```

102 (App2) On Topic

```

103 ----- TopicDemo.java (publisher appn)
104 import javax.naming.*;
105 import java.util.Properties;
106 import javax.jms.*; (JMS API)
107
108 public class TopicDemo
109 {
110     public static final String TOPIC_FACTORY ="ConFactJndi";
111     public static final String TOPIC="TopicJndi";
112     public static void main(String args[]) throws NamingException, JMSException
113     {
114         Properties prop=new Properties(); → JNDI properties of GlassFish registry
115         prop.put(Context.INITIAL_CONTEXT_FACTORY,"com.sun.enterprise.naming.SerialInitContextFactory");
116         prop.put(Context.PROVIDER_URL,"iiop://localhost:4848");
117         Context ctx=new InitialContext(prop);
118         TopicDemo tt=new TopicDemo(); → current class object
119         tt.doPub(ctx); → UserDefined method is called
120         ctx.close();
121     }
122
123     public void doPub(Context ctx) throws JMSException, NamingException
124     {
125         Topic topic=(Topic)ctx.lookup(TOPIC); → get access to the Topic destination of GlassFish registry
126
127         TopicConnectionFactory tcf = (TopicConnectionFactory)ctx.lookup(TOPIC_FACTORY); → X2
128         TopicConnection tc = tcf.createTopicConnection(); → gets JMSConnection from ConnectionFactory
129         TopicSession ts=tc.createTopicSession(false,TopicSession.AUTO_ACKNOWLEDGE);
130
131         TopicPublisher tp=ts.createPublisher(topic); → Publisher is created pointing to Topic
132         tp.start();
133
134     }
135
136     { → TextMessage msg;
137     msg=ts.createTextMessage("This is from Sathya Techs112..I "); → creating one Text msg.
138     System.out.println("Publishing msg on the Destination(Topic)");
139
140     tp.publish(msg); → Sends one Text msg to The Topic destination
141     tc.close();
142     }
143
144 ----- SubscriberDemo.java
145 import javax.naming.*;
146 import java.util.Properties;
147 import javax.jms.*;
148 public class SubscriberDemo{
149     public static void main(String args[]) throws NamingException, JMSException {
150         Properties prop=new Properties();

```

JAR files in classpath same as previous appn()

1st run subscriber appn for multiple times from different windows then run the publisher appn for one time.

```

151 prop.put(Context.INITIAL_CONTEXT_FACTORY,"com.sun.enterprise.naming.SerialInitContextFactory");
152 prop.put(Context.PROVIDER_URL,"iiop://localhost:4848");
153
154 Context ctx=new InitialContext(prop);
155
156 SubscriberDemo tt=new SubscriberDemo(); → current class object
157 tt.doSub(ctx); → User Defined method call refer line no: 160-177
158 ctx.close();
159 }
160 public void doSub(Context ctx) throws JMSEException, NamingException{
161
162 Topic topic=(Topic)ctx.lookup("TopicJndi"); → getting access to the topic destination of GlassFish registry
163
164 TopicConnectionFactory tcf = (TopicConnectionFactory)ctx.lookup("ConFactJndi");
165 TopicConnection tc = tcf.createTopicConnection(); → gets JMS Connection from Connection Factory
166
167 TopicSession ts=tc.createTopicSession(false,TopicSession.AUTO_ACKNOWLEDGE);
168
169 TopicSubscriber tsub=ts.createSubscriber(topic); → TopicSubscriber is created pointing to
170 tc.start(); Topic Destination.
171
172 TextMessage msg;
173 msg=(TextMessage)tsub.receive(); → receives msg from Topic destination
174
175 System.out.println("Received message is:"+msg.getText()); → Displays the content of the msgs.
176 tc.close();
177 }
178 }

```

180 Applications on Spring JMS 181 (App4) on queue

183 Sender App (on Queue)

```

184 -----MessageSender.java (SpringBean) having logic to send Text message
185 import javax.jms.*;
186 import org.springframework.jms.core.*; → code to inject JMSTemplate class object through Setter Injection
187 public class MessageSender {
188
189     private JmsTemplate jt;
190
191     public MessageSender() { } →
192
193     public void setJt(JmsTemplate jt) { } →
194         this.jt = jt;
195
196     }
197
198
199     → userDefined B-method
200     public void sendMessage() {
201
202         MessageCreator creator = new MessageCreator() →
203             {
204                 public Message createMessage(Session session) →
205                     {
206                         TextMessage message = null;
207                         try {
208                             {
209                                 TextMessage msg ← creating the
210                                     message = session.createTextMessage();
211                                     message.setStringProperty("text", "Hello World");
212                                     message.setJndiName("from Sahyadri 112");
213                                     catch (JMSEException e) {
214                                         e.printStackTrace();
215                                     }
216                                     return message;
217                             }
218                         }; → here object for Anonymous
219                         inner class that implements
220                         jt.send(creator); → messageCreator(Interface) is created
221                         representing one JMS message.
222                     }
223     } → sends the msg
224 -----spring.xml (SpringCfg file) having the logic to inject
225 <?xml version="1.0" encoding="UTF-8"?>

```

(SpringBean) having logic to send Text message

code to inject JMSTemplate class object through Setter Injection

jarfiles in classpath

spring jar files : ②

Same 6 Jar files of Appn1 in handout

GlassFish server should be in running mode

Sender

→ Java(2)

→ Spring.xml

Sender > javac *.java

sender > java SendClient

(*) Jndi properties of glassfish registry

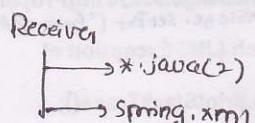
```

226 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
227 "http://www.springframework.org/dtd/spring-beans.dtd">
228 <beans>
229   <bean id="cf" class="org.springframework.jndi.JndiObjectFactoryBean">
230     <property name="jndiEnvironment">
231       <props>
232         <prop key="java.naming.factory.initial">com.sun.enterprise.naming.SerialInitContextFactory</prop>
233         <prop key="PROVIDER_URL">iop://localhost:4848</prop>
234       </props>          java.naming.provider.url    4646
235     </property>
236     <property name="jndiName" value="ConFactJndi"/>
237   </bean>   (*) Gathers ConnectionFactoryObject from glassfish registry
238
239
240   <bean id="q" class="org.springframework.jndi.JndiObjectFactoryBean">
241     <property name="jndiEnvironment">
242       <props>
243         <prop key="java.naming.factory.initial">com.sun.enterprise.naming.SerialInitContextFactory</prop>
244         <prop key="PROVIDER_URL">iop://localhost:4848</prop>
245       </props>          java.naming.provider.url    4646
246     </property>
247     <property name="jndiName" value="QueueJndi"/>
248   </bean>   (*) gives QueueObj pointing to QueueDestination of Glassfish Server.
249
250
251   <bean id="template" class="org.springframework.jms.core.JmsTemplate">
252     <property name="connectionFactory" ref="cf"/> refer line: 229
253     <property name="defaultDestination" ref="q"/> refer line: 240
254   </bean>
255
256   [ our springBean configuration having logic to inject JMSTemplate class object. ]
257   <bean id="ms" class="MessageSender">
258     <property name="jt" ref="template"/> refer line no: 261
259   </bean>
260 </beans>
261
-----SendClient.java (client appn)
262 import org.springframework.context.support.*;
263 public class sendclient
264 {
265   public static void main(String args[])
266   {
267     FileSystemXmlApplicationContet ctx=new FileSystemXmlApplicationContext("spring.xml");
268     MessageSender ms =(MessageSender)factory.getBean("ms");
269     ms.sendMessage();           Gx
270     System.out.println("message sent");
271   }
272
273 }
274
-----Receiver App
275
-----MessageReceiver.java (Spring Bean)
276 import javax.jms.*;
277 import org.springframework.jms.core.*;
278
279 public class MessageReceiver {
280   private JmsTemplate jt;
281
282   public MessageReceiver() {}
283
284   public void setJmsTemplate(JmsTemplate jt)
285   {
286     this.jt = jt;
287   }
288
289
290   public void receiveMessage()
291   {
292     Message message = jt.receive(); receives the msg from
293     TextMessage tmsg = null; Queue Destination
294     if (message instanceof TextMessage)
295     {
296       tmsg=(TextMessage)message;
297
298
299
300

```

Jars files in class path :- same as

SendAppn.



Receiver > javac * .java

> java ReceiverClient

```

301     try
302     {
303         System.out.println(tmsg.getText());
304         System.out.println(tmsg.getStringProperty("text"));
305     }
306     catch (JMSEException e)
307     {
308         e.printStackTrace();
309     }
310 } //if
311 } //method
312 } //class
-----spring.xml-----
313 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
314 "http://www.springframework.org/dtd/spring-beans.dtd">
315 <beans>
316
317     <bean id="cf" class="org.springframework.jndi.JndiObjectFactoryBean">
318         <property name="jndiEnvironment">
319             <props>
320                 <prop key="java.naming.factory.initial">com.sun.enterprise.naming.SerialInitContextFactory</prop>
321                 <prop key="PROVIDER_URL">iop://localhost:4448</prop>
322             </props>          java.naming.provider.url      4446
323         </property>
324         <property name="jndiName" value="ConFactJndi"/>
325     </bean>
326
327     <bean id="q" class="org.springframework.jndi.JndiObjectFactoryBean">
328         <property name="jndiEnvironment">
329             <props>
330                 <prop key="java.naming.factory.initial">com.sun.enterprise.naming.SerialInitContextFactory</prop>
331                 <prop key="PROVIDER_URL">iop://localhost:4448</prop>
332             </props>          java.naming.provider.url      4446
333         </property>
334         <property name="jndiName" value="QueueJndi"/>
335     </bean>
336
337     <bean id="template" class="org.springframework.jms.core.JmsTemplate">
338         <property name="connectionFactory" ref="cf"/>
339         <property name="defaultDestination" ref="q"/>
340     </bean>
341
342     <bean id="ms" class="MessageReceiver">
343         <property name="jt" ref="template"/>
344     </bean>
345 </beans>
-----ReceiveClient.java-----
346 import org.springframework.context.support.*;
347 public class ReceiveClient
348 {
349     public static void main(String args[])
350     {
351         FileSystemXmlApplicationContet ctx=new FileSystemXmlApplicationContext("spring.xml");
352         MessageReceiver mr =(MessageReceiver)factory.getBean("ms");
353         mr.receiveMessage();           ctx
354     }
355
356 }
357
358 }
359
360 App5 (on Topic)
361 Sender App (publisher appn)
-----MessageSender.java (Spring Bean)-----
362 import javax.jms.*;
363 import org.springframework.jms.core.*;
364 public class MessageSender {
365
366     private JmsTemplate jt
367
368     public MessageSender() {}
369
370     public void setJt(JmsTemplate jt)
371     {
372         this.jt = jt;
373     }
374
375 }

```

```

376
377     public void sendMessage()
378     {
379         MessageCreator creator = new MessageCreator()
380         {
381             public Message createMessage(Session session)
382             {
383                 Inner class TextMessage message = null;
384                 try
385                 {
386                     Implementing MessageCreator Interface
387                     message = session.createTextMessage();
388                     message.setStringProperty("text", "Hello World");
389                     message.setText("this is test msg");
390                 }
391                 catch (JMSException e)
392                 {
393                     e.printStackTrace();
394                 }
395                 return message;
396             }
397         };
398         jt.send(creator); → Sends the message to Topic destination.
399     }
400 }
401 }

402 -----spring.xml----- (spring.cfg file)
403 <?xml version="1.0" encoding="UTF-8"?>
404 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
405 "http://www.springframework.org/dtd/spring-beans.dtd">
406
407 <beans>
408     <bean id="cf" class="org.springframework.jndi.JndiObjectFactoryBean">
409         <property name="jndiEnvironment">
410             <props>
411                 <prop key="java.naming.factory.initial">com.sun.enterprise.naming.SerialInitContextFactory</prop>
412                 <prop key="PROVIDER_URL">iop://localhost:4848</prop>
413                 </props> Java-naming-provider-url 4646
414             </property>
415             <property name="jndiName" value="ConFactJndi"/>
416         </bean>

417
418     <bean id="q" class="org.springframework.jndi.JndiObjectFactoryBean">
419         <property name="jndiEnvironment">
420             <props>
421                 <prop key="java.naming.factory.initial">com.sun.enterprise.naming.SerialInitContextFactory</prop>
422                 <prop key="PROVIDER_URL">iop://localhost:4848</prop>
423                 </props> Java-naming-provider-url 4646
424             </property>
425             <property name="jndiName" value="TopicJndi"/>
426         </bean>

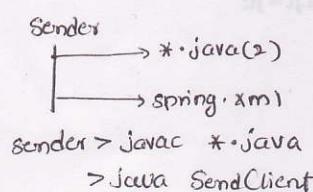
427     <bean id="template" class="org.springframework.jms.core.JmsTemplate">
428         <property name="connectionFactory" ref="cf"/> refer : 408
429         <property name="defaultDestination" ref="q"/> refer : 419
430     </bean>

431     <!-- our springBean injected with JmsTemplate class object -->
432     <bean id="ms" class="MessageSender">
433         <property name="jt" ref="template"/> refer : 430
434     </bean>
435 </beans>

436 -----SendClient.java-----
437 import org.springframework.context.support.*;
438 public class sendclient
439 {
440     public static void main(String args[])
441     {
442         FileSystemXmlApplicationContext ctx=new FileSystemXmlApplicationContext("spring.xml");
443         MessageSender ms =(MessageSender)factory.getBean("ms");
444         ms.sendMessage(); Ctx
445         System.out.println("message sent");
446     }
447 }

```

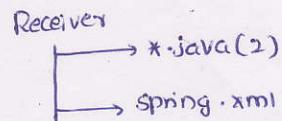
Jar files in CLASSPATH : Same as previous apps.



```

451
452 }
453
454 Receiver App (Subscriber)
455
456 MessageReceiver.java
457 import javax.jms.*;
458 import org.springframework.jms.core.*;
459
460 public class MessageReceiver {
461     private JmsTemplate jt;
462
463
464     public MessageReceiver() {}
465
466     public void setJmsTemplate(JmsTemplate jt)
467     {
468         this.jt = jt;
469     }
470
471
472     public void receiveMessage()
473     {
474         Message message = jt.receive();
475         TextMessage tmsg = null;
476         if (message instanceof TextMessage)
477         {
478             tmsg=(TextMessage)message;
479             try
480             {
481                 System.out.println(tmsg.getText());
482                 System.out.println(tmsg.getStringProperty("text"));
483             }
484             catch (JMSException e)
485             {
486                 e.printStackTrace();
487             }
488         }
489     }
490 }
491
492 spring.xml
493 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
494 "http://www.springframework.org/dtd/spring-beans.dtd">
495 <beans>
496
497     <bean id="cf" class="org.springframework.jndi.JndiObjectFactoryBean">
498         <property name="jndiEnvironment">
499             <props>
500                 <prop key="java.naming.factory.initial">com.sun.enterprise.naming.SerialInitContextFactory</prop>
501                 <prop key="PROVIDER_URL">iijop://localhost:4848</prop>
502             <props>          Java-naming-provider-conf      4646
503         </property>
504         <property name="jndiName" value="CenFactJndi"/>
505     </bean>
506
507     <bean id="q" class="org.springframework.jndi.JndiObjectFactoryBean">
508         <property name="jndiEnvironment">
509             <props>
510                 <prop key="java.naming.factory.initial">com.sun.enterprise.naming.SerialInitContextFactory</prop>
511                 <prop key="PROVIDER_URL">iijop://localhost:4848</prop>
512             <props>          Java-naming-provider-url      4646
513         </property>
514         <property name="jndiName" value="QueueJndi"/>
515     </bean>          TopicJndi
516
517     <bean id="template" class="org.springframework.jms.core.JmsTemplate">
518         <property name="connectionFactory" ref="cf"/> refer : 497
519         <property name="defaultDestination" ref="q"/> refer : 507
520     </bean>
521
522     <bean id="ms" class="MessageReceiver">
523         <property name="jt" ref="template"/> refer : 517
524     </bean>
525 </beans>

```

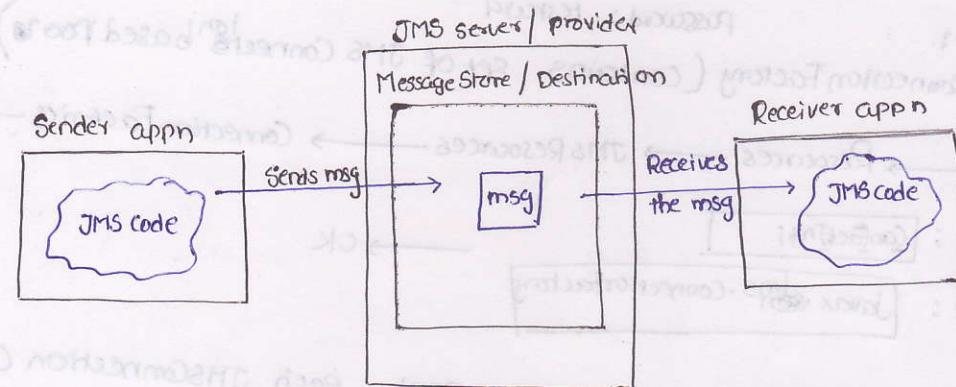


>javac *.java
>java ReceiveClient

First start multiple Receiver / subscriber
apps then start only one sender appn
(publisher appn)

```
526 -----ReceiveClient.java-----  
527 import org.springframework.context.support.*;  
528 public class ReceiveClient  
529 {  
530     public static void main(String args[]){  
531         FileSystemXmlApplicationContet ctx=new FileSystemXmlApplicationContext("spring.xml");  
532         MessageReceiver mr =(MessageReceiver)factory.getBean("ms");  
533         mr.receiveMessage();  
534         ctx  
535     }  
536 }  
537 }  
538 }
```

- To achieve asynchronous communication b/w browser window and webapp use either Ajax (client side technology) or Java Portlets (serverside Technology)
- To achieve messages based asynchronous communication between two standalone Java apps use JMS based messaging.



- JMS is given to perform Messages based Asynchronous Communication between two Java appns.
- JMS server / provider supplies destinations having capability to store message temporarily
 - Eg:- JMS Server / providers
 - MASeries from IBM
 - MSMQ from Microsoft
 - Weblogic Message Server from Weblogic
 - Glassfish Message Server from GlassFish
- Every Application Server supplies one built-in JMS Server.
- For Basics for Messaging and JMS refer page no's 1 to 4 of the supplementary handout given on 02/04/2013
- JMS supports two messaging domains
 - 1) PTP (Point to Point) Domain
 - uses Queue as Destination / Message Store
 - 2) pub/sub Domain
 - uses Topic as the Destination / Message Store

For related info on these messages domain refer page no's 1 & 2 of the 02/04/2013.

(*) Procedure to create in GlassFish 2.x server

Step(1):- Start mydomain2 server of GlassFish. and open its adminConsole.

Sun\appserver\bin>asadmin start-domain mydomain2

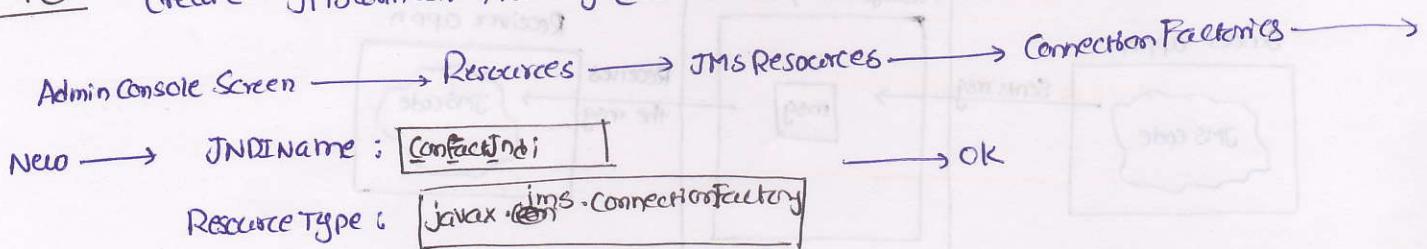
Step(2):-

http://localhost:4646

Login : testUser

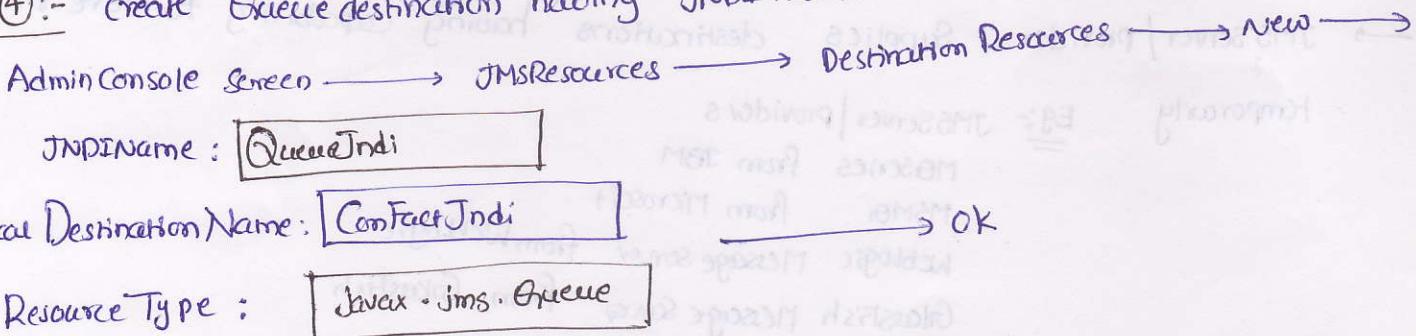
Password : testUser

Step(3):- Create JMS Connection Factory (contains set of JMS Connectors based pool)

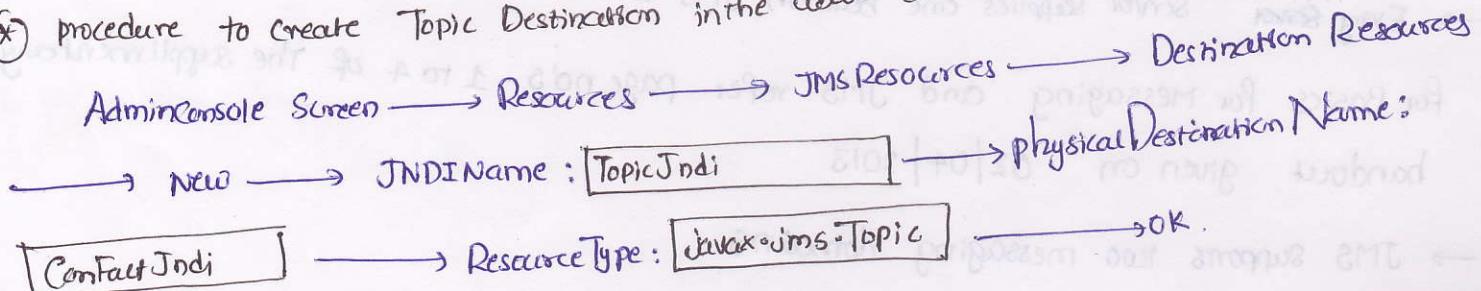


JMS Connection Factory represents JMS Connection pool. each JMS Connection Object represents the connection b/w client appn and JMS server.

Step(4):- Create Queue destination having JNDI name



(*) procedure to create Topic Destination in the above environment



→ JMS is part of JEE module JMS API means working with javax.jms pkg
the jar files the JEE APIs contains JMS API

in Weblogic Server → Weblogic.jar

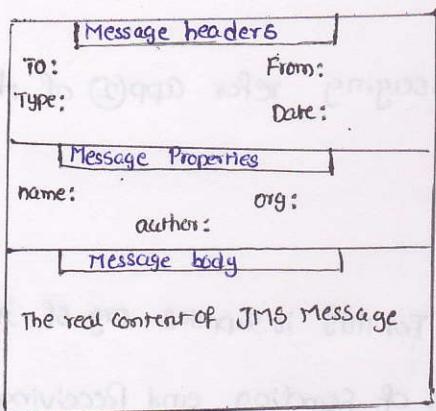
in glassfish Server → javamail.jar

in JBoss Server → Jboss-javamail.jar

Sender/Publisher App uses JMS API to send ^{msgs} to the Message Store

Received App/Subscriber App uses JMS API to receive msgs from the Message

The Structure of JMS message



- header names are fixed
- property names and values are programmer choice
- names and values
- To send additional info along with Message we need to work with Message properties

→ There is a possibility of developing 5 types of messages for related info on these messages refer page no: (4) of 02/04/2013 handout.

→ Important resources of JMS API

related Queue

QueueConnectionFactory (I)
QueueConnection (I)
Queue (I)
QueueSession (I)

QueueSender

QueueReceiver
and etc....

related Topic

TopicConnectionFactory (I)
TopicConnection (I)
Topic (I)
TopicSession (I)

Topic Publisher

Topic Subscriber
and etc....

related to Messages

Text Message
Stream Message
Map Message
Object Message
and etc....

*) For plain JMS API based Sender, receiver application dealing with Queue destination based PTON Messaging Domain refer appn(1) of the 02/04/2013 handout

→ The real time example applications of JMS are

- ① Complaint management
- ② Bugs management
- ③ Offline order booking and etc..

* For plain JMS based pub/sub model of messaging refer app(2) of the page no's

(6) & (7) of 02/13

Spring JMS :-

- provides abstraction layer of plain JMS. For this it supplies org.springframework.jms.core.JmsTemplate class.
- This JMS Template class simplifies the process of sending and receiving msgs.
- No need of working with plain JMS API for JMS operations
- Gives callback interfaces to ~~develop~~ ^{create} diff types of messages.
- To create JMS template class obj JMSConnector obj, Topic/Queue destination objs are required.
- Spring JMS gives the callback interface called messageCreator to create diff types of message creators by using the exposed JMSsession object.

* For Spring JMS based p-to-p example application refer appn(4) of the

page no's (7) to (9)

* For example appn on Spring JMS based pub-sub messaging domain appn(5)
of the page no's (9) to (12)

Module - V

Spring Web Module

Spring Web module

- Part 1 (Gives plugins and facilities to make Spring apps communicateable from other webflow slcs based apps like Struts apps, JSF app etc.)
- Part 2 (Spring Web MVC / Spring MVC / Spring Web Flow)
 - It is Spring's own web slc to develop MVC2 architecture based webapps.

Struts and Spring Integration :-

- In this appn Struts appn represents View layer and Controller layer logics.
(Presentation logic) (Integration logic)
- and Spring appn represents Model Layer logics.
(Business Logic & Persistence logic)

The two ways of Struts and Spring Integration.

- ① Struts with Spring → Struts and Spring appns reside and execute on the same JVM

- Struts appn is local client appn to Spring appn
- here Spring appn must not be a Distributed appn

- ② Struts to Spring

- here Struts appn and Spring appn resides on two different Jvms of same or different computer
- Struts appn is Remote client appn to Spring appn.
- here Spring appn must be a Distributed appn.

- The plugin required for Struts 1.x and Spring integration is

org.springframework.web.struts.ContextLoaderPlugIn and this plugin is supplied by

Spring slc to configure in Struts Configuration file by using <plugin> tag,

ActionServlet recognizes all the plugins configured in Struts Configuration file

the moment it is instantiated (Object creation). ActionServlet instantiation

takes place either during Server Startup or during the deployment of Webappn. because we generally enable load-on-startup on ActionServlet.

- The above Context Loader PlugIn activates Webappn Context Spring Container by taking specified file name (or) ActionServlet logicalname - servlet.xml as Spring Configuration file name.
- In this process all the beans that are configured will be pre instantiated

In struts cfg file

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn"/>
```

here the activated spring container takes <ActionServlet Logicalname> - servlet.xml

In struts cfg file

```
<plug-in className="org.sf.web.struts.ContextLoaderPlugIn">
```

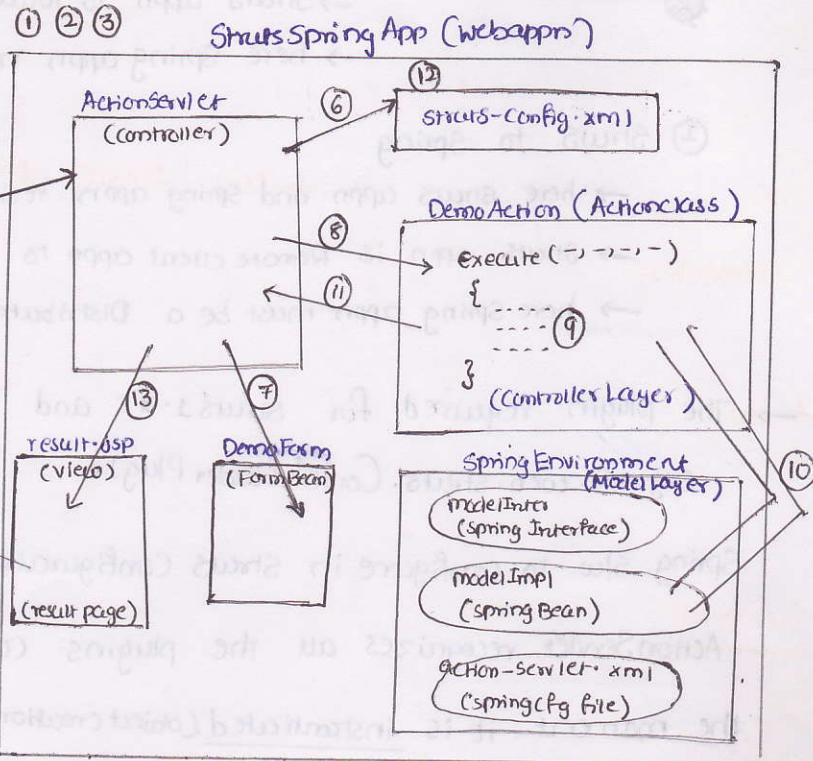
```
  <set-property property="contextConfigLocation" value="/WEB-INF/mycfg1.xml/WEB-INF/mycfg2.xml"/>
```

```
</plug-in>
```

→ here the activated Spring container takes mycfg1.xml, mycfg2.xml files as Spring Configuration files.

→ In struts with Spring appn development the Struts Action class should contain Client code and should call the B. methods of Spring Bean available in Spring appn.

Example appn on "Struts with Spring"



W.r.t to the diagram

- ① programmers deploys struts spring application in webserver or ApplicationServer.
- ② because of LoadOnStartup servlet container creates ActionServlet class obj either during Server startup (or) during the deployment of webappn.
- ③ ActionServlet activates ContextLoadPlugIn. this plugin activates the webApplicationContext Container by taking <ActionServlet Logicalname> - servlet.xml as Spring Configuration file.

In this process all the beans of Spring Configuration file will be pre-instantiated

④ & ⑤ end user launch the Form page and submit the request to webappn.

⑥ ActionServlet traps and takes the request and uses Struts Configuration file entries to decide Formbean, and Action class that are required to process the request

⑦ ActionServlet writes the Received Form data to FormBean class object

⑧ ActionServlet calls execute(-,-,-,-) method of Actionclass.

⑨ This execute(-,-,-,-) method somehow manages to get SpringBean class object

⑩ execute(-,-,-,-) method calls Business method of Spring Bean and gets the result. keeps this result in request attribute to make it visible to result page.

⑪ This execute(-,-,-,-) method returns the control back to ActionServlet.

⑫ ActionServlet uses Struts Configuration file entries to decide the result page of Action class

⑬ ActionServlet passes the Control to resultpage

⑭ Result page formats the results and sends the results to browser window as the dynamic webpage Content

There are two approaches to develop Struts with Spring appn.

Approach 1) By making Struts Action class as Spring Aware class with the support of

XxxSupport classes (like ActionSupport, DispatchActionSupport,)

Approach 2) By Injecting Spring Bean class object to Struts Action class property (for this we need to cfg Struts Action class as Spring Bean in Spring cfg file)

Approach 1) discussions :-

When struts Action class extends from XxxSupport class of Spring api. The

Struts Action class can get access to the ContextLoader PlugIn activated Spring Container.

and we can use this Spring Container to get access to Spring Bean class object and

to call B. methods on it.

→ in struts and spring integration always prefer working with ContextLoaderPlugIn rather

Spring Container. because this container can be used in multiple Action classes and

this container will be activated either during Server startup (or) during the

deployment of webappn. This process improves the performance when compared to

activating Spring Container in every Action class Separately.

→ Sample Struts Action class Code of Approach (1) :-

```
public class TestAction extends ActionSupport
{
    public ActionForward execute(-, -, -, -)
    {
        // get access to the Spring Container activated by
        // Context Loader Listener
        WebApplicationContext ctx = getWebApplicationContext();
        // get access to Spring Bean class Obj
        modelinter bobj = (modelinter) ctx.getBean("mt");
        // call the B.method of Spring Bean
        bobj.bm1(-);
    }
}
```

(*) For the above diagram based and the above approach (1) based Struts 1.x

With Spring appn refer appn (23) page no: 75-77

for different approaches of Struts & Spring appn based development related documentation
refer page no's (26) to (34) of the material.

Approach (2) Discussions:-

- here we configured Struts Action class as Spring Bean class in Spring Configuration file. so that we can inject the model layer Spring Bean class object to Struts Action class to call the Business methods.
- In this process we need to configure one Dummy Action class as proxy class for original Struts Action class in Struts Configuration file to receive the request from Action Servlet and to pass the request to Original Struts Action class that is configured in Spring Configuration file.
- Spring API Supplies org.sfg.web.struts.DeliveringActionProxy class for this purpose

→ In struts Configuration file

<action path="/xyz" type="org.apache.struts.DegatewayingActionProxy">

↳ Proxy class for original Action class

<action>

In Spring Configuration file

<beans>

<bean name="/xyz" class="TestAction">
↳ must match with above xyz (actionpath)

<property name="bobj" ref="d1"/>
↳ logic to inject SpringBean class object to a property of Struts Action class

</bean>

<bean id="d1" class="ModelImpl"/>

</beans> ↳ springBean

Sample Struts Action class of the above scenario :-

public class TestAction extends Action

{

 ModelInter bobj; // Bean property to hold Spring Bean class obj

 public void setBobj(ModelInter bobj)

{

 this.bobj = bobj;

}

 public ActionForward execute(..., ...) throws Exception

{

 String result = bobj.getWish(); // calling b.method on springBean class object.

 ...
}

}

→ Approach ② gives better performance in Struts Integration because it injects Spring Bean

class object to Struts Action class property either during Server Startup or

during the deployment of the webapp.

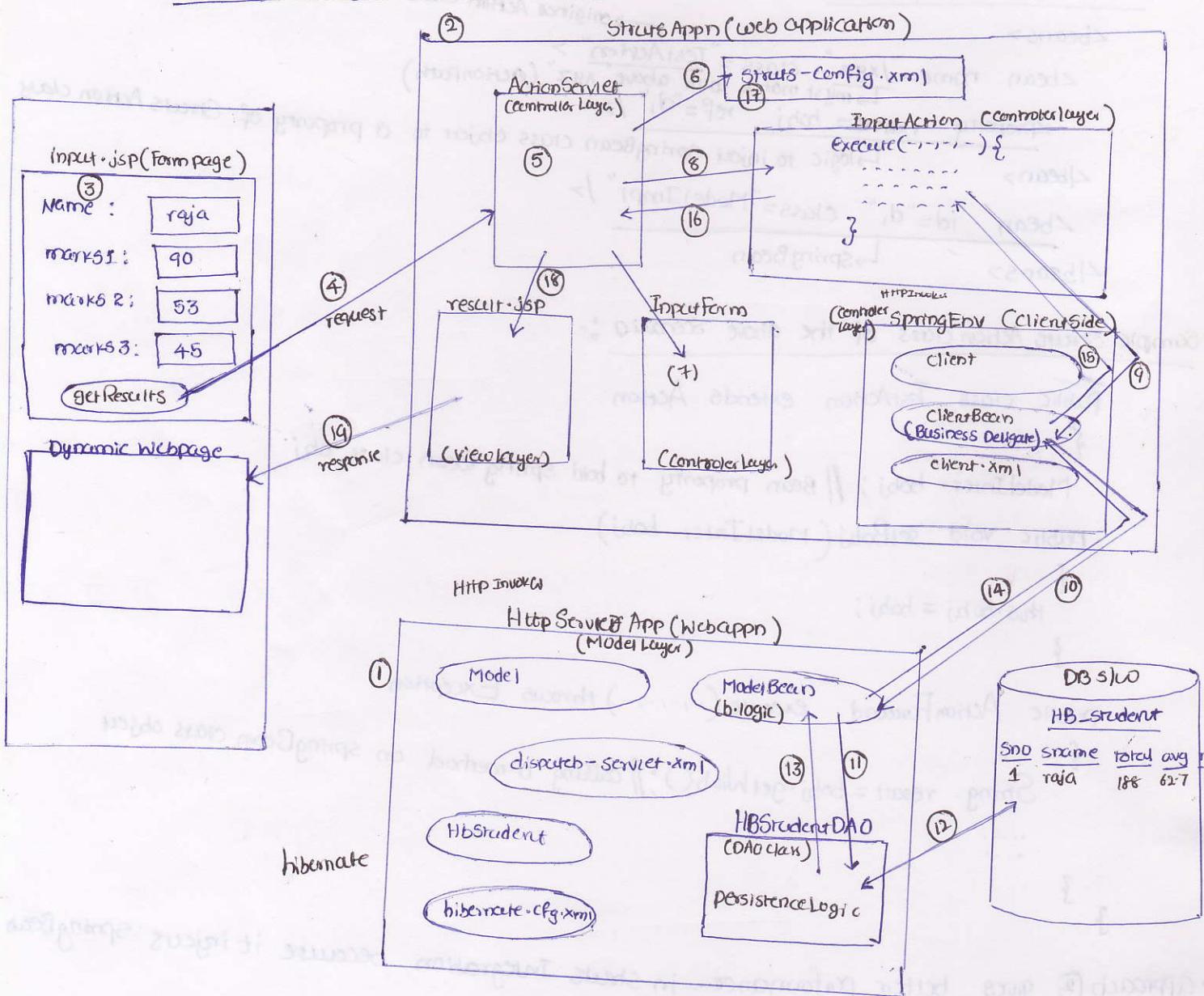
for approach no ② based Struts with Spring app development refer appn no: 22

Page no: 73 to 75

*) MiniProject showing Struts, Spring, hibernate Integration and also implementing MVC2, DAO, Business Delegate Design patterns.

→ The helper java class of client appn that separates Business method calls to make them flexible to modify is called Business Delegate class.

Struts to spring with Hibernate appn



In the above diagram Server appn is HttpInvoker Server appn, which uses spring JEE, Spring ORM modules to develop the business, persistence Logics.

→ spring Environment placed in Struts appn is HttpInvoker Client appn to help struts Action class interacting with HttpInvoker Server appn.

→ In the above diagram two webapps core there (1) Struts Application.

(2) The HttpInvokerServer appn.

→ In the above appn Business Logic is calculating total Avg and results based on Students marks and persistence Logic is inserting record.

procedure to develop the above diagram based mini project by using MyEclipse IDE
for server application:-

Step-I:- Create Web project.

File → New → Web project → Project Name: [HttpServerApp] → Finish.

Step-II:- Configure Dispatcher servlet in web.xml file having *.do url pattern and the logical name "dispatcher".

refer web.xml of page no: (66)

Step-III:- Create HB_Student table in Oracle DBMS.

Create table HB_Student (sno number(4) primary key, sname varchar(20), total number(6), Avg number(8,2), result varchar(5));

Step-IV:- Create DataBase profile for oracle in MyEclipse IDE

Windows → open perspective → My Eclipse DataB base Explorer → DB Browser →

rightclick → NEW →

Driver Template : [Oracle (Thin Driver)]

Driver Name : [orap]

Connection URL : [jdbc:oracle:thin:@localhost:1521:ORCL]

Username : [scott]

Password : [tiger]

add jars [ojdbc14.jar]

→ next → finish

Step-V:- Add Hibernate Capabilities to the project

Rightclick on project → MyEclipse → Add Hibernate Capabilities → [Hibernate 3.2]

Enable Hibernate Annotation support → Next → Next →

DBdriver : [orap]

Password : [tiger]

→ next → create SessionFactory class? → Finish

Step-VI:- Perform Hibernate Reverse Engineering on HB_Student table

window menu → openperspective → my Eclipse DBExplorer → DB Browser →

→ Right click on orap → OpenConnection → password : [tiger] → OK

→ Expand orap → Expand Connected to orap → Expand scott → Expand table

Right click on HB_Student → Hibernate Reverse Engineering →

java scr folder | /Httpscr App /src

Create POJO <=> DB table → Java Data Access Object (DAO)

① Add Hibernate mapping Annotations.

update hibernate configuration

generate precise find By methods.

DAO type : SpringDAO

→ next → ① Hibernate types

IDGenerator : increment

Hb_student → IDencrator: [increment] → Finish.

Step-III:- Add spring Capabilities to the project

Rightclick on project → my Eclipse → Add Spring capabilities → spring 2.5
→ Using misc libraries → next

→ Select Core, Persistence Core, JDBC, J2EE, Remoting, Misc Libraries → next

→ Enable App Builder → File: dispatcher-servlet.xml → next → Add Hibernate 3.2 annotation support.

SessionFactory Bean id : sesfact → finish.

Step-VII :- This step generates Hbstudent.java $\xrightarrow{\text{POJO class}}$ HbStudent DAO.java $\xrightarrow{\text{DAO class}}$ files

Step-VIII :- Add springInterface, springBeans^{class} to the project.

src → new → Interface : [model]

model.java

```
public interface Model {  
    public String generateResult(String name, int m1, int m2, int m3)  
}
```

~~Step 1~~ ~~Step 2~~ Src → new → class: ModelBean implements Model

public class ModelBean implements Model {

implements model

```
    public class ModelBean implements Model {  
        HibStudentDAO dao;
```

```
public void setDao (HbStudent DAO dao) {
```

```
public void setDao (HbStudent DAO dao) {  
    this.dao = dao ;  
}  
public String generateResult (String name, int m1, int m2, int m3) {
```

//Write blogic

$$\text{int_total} = m_1 + m_2 + m_3;$$

float avg = (float) total / 3.0f;

// generate result

~~Keypress~~ String res = " ";

```

if (avg < 35)
    res = "fail";
else
    res = "pass";
// insert record by using DAO class
HbStudent st = new HbStudent();
st.setName(name);
st.setTotal(total);
st.setAvg((double) avg);
st.setResult(res);
dao.save(st);
return res;
}

```

// class

Configure the following additional beans in Spring Configuration file.

in dispatcher-servlet.xml

```

<bean id=""
      ...
      ...
</bean>
<bean id="HbStudentDAO"
      ...
      ...
</bean>
<bean id="mb" class="ModelBean">
    <property name="dao" ref="HbStudentDAO"/>
</bean>
<bean name="/service" class="org.sf.remoting.httpinvoker.HttpInvokerServiceEx">
    <property name="serviceInterface" value="model"/>
    <property name="service" ref="mb"/>
</bean>
<bean id="surl" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="my.do">/service</prop>
        </props>
    </property>
</bean>
</beans>

```

Configure Tomcat Server with MyEclipse IDE

Step - IX :- Run the project

Right click on project → Run as → My Eclipse Service Application → Tomcat 6.0

→ OK

Changes:-

Change 1:- Copy dispatcher-servlet.xml file to WEB-INF folder of Webroot

Change 2:- Add spring-home\lib\asm\asm-2.2.3.jar to the Build Path of the project

Change 3:- If ClassVisitor problem is raised

a) remove Hibernate, Spring Libraries from the Build path of the project generated by IDE

b) add spring, Hibernate related jar files manually into the libraries of the project

Spring jars

Spring.jar

Commons-logging.jar

Spring-Webmvc.jar

Hibernate Jar files

The Regular 8 jar files
hibernate

(hibernate 3.jar)
(JTA-1.1.jar)
(javassist-3.14.0.GA.jar)
(hibernate-JPA-2.0-api-1.0.0.Final.jar)
(commons-collection-3.1.jar)

antlr-2.7.6.jar
slf4j-api-1.6.1.jar
dom4j-1.6.1.jar

Struts appn

Step - I :- Create web project having name StrutsAPP

File menu → new → web project → projectName : [Struts App] → Finish

Step - II :- add spring capabilities to the project.

Right click on project → MyEclipse → Add spring Capabilities → Spring 2.5

→ Select Spring 2.5 Core Libraries, JEE Libraries, Remoting Libraries, Misc Libraries

→ next → Enable Aop Builder → File : [client.xml] → Folder : [WebRoot/WEB-INF]

→ Finish

NOTE :- copy Model.java from Server appn (HttpServerApp) to current appn (StrutsApp)

Step - III :- Develop Spring Interface, Spring Bean having Logic to call the Business methods of Server appn.

↳ client ↳ ClientBean

File → New → Interface → Name : [Client] → Finish

public interface Client {

 public String callMethod(String name, int m1, int m2, int m3);

Src → new → class → name: ClientBean → Finish.

```
public class ClientBean implements Client {
    Model bobj;
    public void setBobj(Model bobj) {
        this.bobj = bobj;
    }
}
```

```
public Model getBobjRef() {
    return bobj;
}
```

return Bobj reference to Struts Action class.

```
public String callBmethod(String name, int m1, int m2,
                           int m3) {
    // call b.method of Server App
    String result = bobj.generateResult(name, m1, m2, m3);
    return result;
}
```

in client.xml

```
<bean id="Pfb" class="org.sf.remoting.HttpInvokerProxyFactoryBean">
    <property name="serviceInterface" value="Model" />
    <property name="serviceUri" value="http://localhost:2020/HttpServerApp/my.do" />
</bean>
```

```
<bean id="cb" class="ClientBean">
    <property name="bobj" ref="Pfb" />
</bean>
```

NOTE:-

```
<bean name="/demo" class="InputAction">
```

```
    <property name="mo" ref="cb" />
</bean>
```

```
</beans>
```

Step-IV:- add Struts Capabilities to the project

Right click on project → myEclipse → Add Struts Capabilities → Struts 1.3
→ Base packages for [] → Finish.

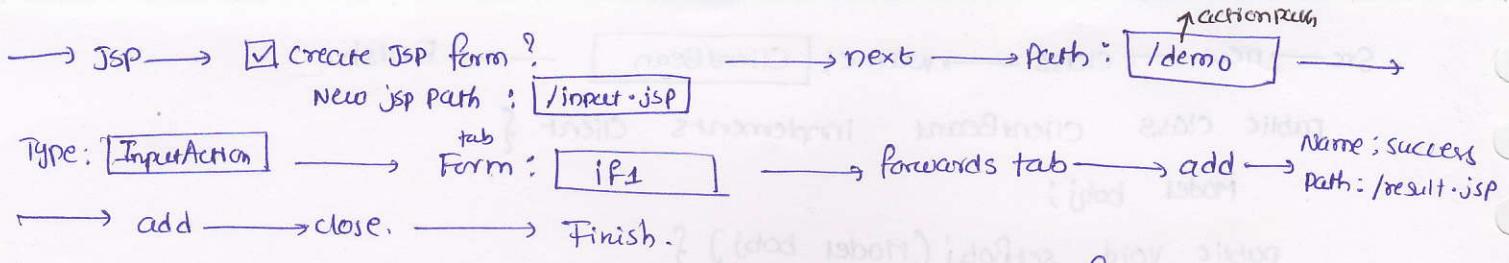
Step-V:- add Form page, FormBean class, Action class to project.

R.C. on project → new → Others → myEclipse → WebStruts → Struts 1.3
→ FormBean Log: custom → FormBean Log: custom

→ Struts 1.3 Form, Action & jsp → next → Name: if1 →

Super class: org.apache.struts.action.ActionForm → Form Type: InputForm

Form Properties: [name m1, m2, m3] → add → methods [] → deselect all the check boxes



Step-III:- Do following modifications in struts configuration file.

- ① remove existing Action class configuration and Configure Delegating Action Proxy (Under <action-mappings> tag)

<action-mappings>

<action path="/demo" type="org.sf.web.struts.DelegatingActionProxy" name="if1">
 <forward name="success" path="/result.jsp"/>

- ② Configure the ContextLoaderPlugIn as shown below.

(under <action-mappings> tag)

<message-resources parameter="ApplicationResources"/>

<plug-in className="org.sf.web.struts.ContextLoaderPlugIn">

<set-property property="contextConfigLocation" value="/WEB-INF/context.xml"/>

</plug-in>

Step-IV:- Write following code in Action Class

//InputAction.java

public class InputAction extends Action {

client mo;

public void setMo(Client mo) {

this.mo = mo;

}

public ActionForward execute (...) {

//read form data
 InputForm if1 = (InputForm) form;

String name = if1.getName();

int m1 = Integer.parseInt(if1.getM1());

int m2 = Integer.parseInt(if1.getM2());

int m3 = Integer.parseInt(if1.getM3());

//call B.method of client Bean

String result = mo.callBmethod(name, m1, m2, m3);

```
//keep results in request attribute  
request.setAttribute("msg", result);  
return mapping.findForward("success");
```

} //method

} //class

Step VII:- Add Result.jsp to Webroot folder of the project.

R.c. on Webroot folder → new → JSP → filename: result.jsp → finish.

Result is = $\text{ly} = \text{request.getAttribute("msg")}$

Step VIII:- Deployee this project in tomcat Server and also make sure that the Server appn is also in running mode.

R.c on project (struts App) → Runas → my Eclipse Server appn → Tomcat 6.

→ OK

Step IX:- Test the application.

Open browser coin down → type this Uri

<http://localhost:2020/StrutsApp/input.jsp>

Q:- What is the difference b/w programming Language, Technology and Frameworks?

→ Languages are ^{raw material} given to programmers having basic and direct facilities to develop applications.

These Languages are useful while creating Technologies, Frameworks.

Eg:- C, C++, Java.

→ Software Technology is a SW specification having set of rules and guidelines to develop softwares. While developing this SWs ^{the} programming language will be used.

Eg:- JDBC, Servlet, JSP, EJB and etc..

→ Frameworks are special SW technologies which provide abstraction layer on multiple ^{other} core technologies

Eg:- Struts, JSF, Spring & etc..



```

1 -----register.jsp (Formpage) Struts 2
2 <%@ taglib prefix="html" uri="/struts-tags" %>
3 <html>
4 <head></head>
5 <body>
6 <h1>User Registration Page</h1>
7 <html:form action="reg">
8     <html:textfield name="username" label="Username"/>
9     <html:password name="password" label="Password"/>
10    <html:submit value="Register"/>
11 </html:form>
12 </body>
13 </html>
14 -----success.jsp (result page) Struts 2
15 <%@ taglib prefix="myHtml" uri="/struts-tags" %>
16 <html>
17 <head></head>
18 <body>
19 <h1>Success Page</h1>
20 <h4>Hello <myHtml:property value="username"/></h4>
21 </body>   ↗ retrieves gives username from Value Stack obj
22 </html>
23 -----web.xml (DDF file) Struts 2
24 <web-app>
25     <filter>
26         <filter-name>struts2</filter-name>
27         <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>
28     </filter>
29     <filter-mapping>
30         <filter-name>struts2</filter-name>
31         <url-pattern>*</url-pattern>
32     </filter-mapping> } Struts 2 Supplied Filter Configuration which
33     <welcome-file-list> actually acts as Controller.
34         <welcome-file>register.jsp</welcome-file> to make Service Filter to trap and take care the request and responses of
35     </welcome-file-list> all the web resource programs.
36 <listener>
37     <listener-class>
38         org.springframework.web.context.ContextLoaderListener
39     </listener-class>
40 </listener>
41 </web-app>
42 -----struts.xml (Struts Configuration file) Struts 2
43 <?xml version="1.0" encoding="UTF-8" ?>
44 <!DOCTYPE struts PUBLIC
45     "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
46     "http://struts.apache.org/dtds/struts-2.0.dtd">
47 <struts>
48     <!-- This tells the struts, that Injection(for Action Class) will be done by spring-->
49     <constant name="struts.objectFactory" value="spring" />
50     <!-- <constant name="struts.devMode" value="true" />-->
51     <package name="myPack" extends="struts-default">
52         <action name="reg" class="regAction" > } !-- does not look for class, but looks for bean in Spring cfg file-->
53             <result name="SUCCESS">success.jsp</result> real Register Action class (refer line no : 79)
54         </action>
55     </package>
56 </struts>
57 -----applicationContext.xml (Spring configuration file) Spring
58 <beans xmlns="http://www.springframework.org/schema/beans"
59     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
60     xmlns:context="http://www.springframework.org/schema/context"
61     xsi:schemaLocation="http://www.springframework.org/schema/beans
62     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
63     http://www.springframework.org/schema/context
64     http://www.springframework.org/schema/context/spring-context-2.5.xsd">
65
66     <!-- Configuring datasource-->
67     <bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
68         <property name="username" value="scott"/>

```

- *1 hear the Struts Action class name is not specified but its beanId is specified as spring class name. That means the struts environment should looks for StrutsAction class as Spring Bean in Spring Configuration file.

```

69 <property name="password" value="tiger"/>
70 <property name="url" value="jdbc:oracle:thin:@localhost:1521:orcl"/>
71 <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
72 </bean>
73 <context:component-scan base-package="p1" /> to recognizes the Spring Beans of 'p1' package
74 </beans>
75 ----- RegisterAction.java (struts2 Action class Configured as the Spring Bean)
76 package p1;
77 import javax.annotation.Resource;
78 import org.springframework.stereotype.Component;
79 @Component("regAction") here struts2 Action class is configured as SpringBean
80 public class RegisterAction { in springConfiguration file.
81     @Resource(name="dbs") here SpringBean class obj is injected to a property of
82     private DBStore dbStore; struts Action class.
83
84     public void setDbStore(DBStore dbs) { setter method Supporting SetterInjection
85         dbStore=dbs;
86     }
87     private String username,password; properties to hold Form data.
88
89 //getters and setters
90     public String getUsername() {
91         return username;
92     }
93
94     public void setUsername(String username) { getter () & setters (-) methods on
95         this.username = username;
96     }
97
98     public String getPassword() {
99         return password;
100    }
101
102    public void setPassword(String password) {
103        this.password = password;
104    }
105
106 // all struts logic here
107    public String execute() {
108        System.out.println(" Execute Method called");
109        dbStore.storeUser(username,password); the calling B-method of Spring Bean class
110        return "SUCCESS";
111    }
112 }
113 -----DBStore.java (Spring Interface) spring
114 package p1;
115 public interface DBStore{
116     public void storeUser(String uname,String password); Declaration of B-method
117 }
118 -----DBStoreImpl.java (Spring Bean) spring
119 package p1;
120 import javax.sql.*;
121 import java.sql.*;
122 import org.springframework.beans.factory.annotation.Autowired;
123 import org.springframework.stereotype.Component;
124
125 @Component("dbs") current bean id
126 public class DBStoreImpl implements DBStore{ springInterface
127     Connection con;
128     @Autowired Injects DataSource object to 'ds' property through Autowiring
129     DataSource ds;
130
131     public void storeUser(String uname,String password)
132     {
133         try { B-method
134             con=ds.getConnection();
135             gets JDBC Connection object from the Connection pool
136         }
137     }
138 }

```

we are configure

In this application Struts 2 Action class as SpringBean class. The real SpringBean class object is injected to struts2 Action class. so that B-method of SpringBean can be called from Struts2 Action class.

```

137     PreparedStatement ps=con.prepareStatement("insert into userdetails values(?,?)");
138     ps.setString(1, uname);
139     ps.setString(2,password);
140     int count=ps.executeUpdate();
141
142     if(count!=0)
143         System.out.println("Successful");
144     else
145         System.out.println("Not Successful");
146     con.close();
147 }
148 catch(Exception e)
149 {
150     e.printStackTrace();
151 //con.close();
152 }
153 }
154 */
155 SQL> create table.userdetails
156 (
157     username varchar2(10),
158     password varchar2(10)
159 );
160 */

```

JAR files in WEB-INF/lib Folder

asm - 3.3
 asm-commons - 3.3
 asm-tree - 3.3
 commons-fileupload - 1.2.2
 commons-io - 2.0.1
 commons-lang3 - 3.1

} collects from Struts 2.3 S/W

commons-logging → from spring 2.5

freemarker - 2.3.19 } from Struts 2.3 S/W

javassist - 3.11.0.GA }

ognl - 3.0.5

ojdbc14 → from oracle

struts2-core - 2.3.4.1

***struts-spring-plugin - 2.1.8

xwork-core - 2.3.4.1

} from struts 2.3 S/W

org.sf.asm - 3.1.1 - RELEASE }

 " beans - 3.1.1 " "

 " context - 3.1.1 " "

 " core - 3.1.1 " "

 " expression - 3.1.1 " "

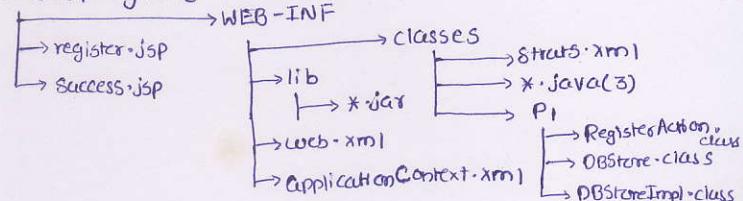
 " jdbc - 3.1.1 " "

 " web - 3.1.1 " "

 " web-servlet - 3.1.1 " "

 " web-struts - 3.1.1 " "

} from Spring 3.1

Deployment Directory StructureStruts2Spring AnnoJAR files in class path :

Struts 2 - Core - 2.3.4.1.jar, org.sf.context - 3.1.1.RELEASE.jar,
~~spring jar~~ org.springframework.context-3.1.1.RELEASE.jar org.springframework.web-3.1.1.RELEASE.jar
~~commons-logging jar~~ org.springframework.beans-3.1.1.RELEASE.jar

In Struts 2.x

- NO Form beans
- Action classes are POJO classes and they also hold form data.
- Struts configuration file name is struts.xml
- Controller is Servlet Filter
 - Struts 2.0.x → FilterDispatcher
 - Struts 2.1.x → StrutsPrepareAndExecuteFilter
- main jar files representing Struts 2.x API are : struts2-core-<version>.jar
xwork-<version>.jar
- Struts 2 gives built-in JSP tag libraries whose taglib uri is : /struts-tags
- Servlet Listeners are given to perform Event Handling on request, response, Session, Servlet Context
- Spring 3.5 gives org.springframework.context.ContextLoaderListener as ServletContextListener which will be activated when Servlet Context object is created/destroy. This Listener activates the webapp context container by taking applicationContext.xml as the Spring configuration file. In this process all the Spring beans will be instantiated.

* For example Application On Struts 2.x and Spring 3.5 Integration refer the supplementary handout given on 10/04/2013

- with spring 3 appn handout of 10/04/2013
- Flow of execution related to Struts 2.x
 - a) Deployment
 - b) Instantiation of Servlet Filter either during server startup or during deployment of webapps.
 - c) → Context Loader Listener activation (37-39)
 - web App Context Container activation by taking applicationContext.xml as the Spring Configuration File
 - pre-Instantiation of Spring beans belonging to
 - SpringBean class (DBstoneImpl obj) will be injection to StrutsAction^{class} (Register Action)
 - end user gives request to Formpage (register.jsp) this form page submits the request (ref: 10)
 - d) end user gives request to Formpage (register.jsp) this form page submits the request (ref: 10)
 - e) Servlet Filter traps and takes the request (26-32)
 - f) Servlet Filter passes the request to Struts Action class Configuration (52-54)
 - g) Servlet Filter calls execute() method of Struts Action class. (108-112)

- b) this execute() calls storeDetails() of Spring Bean class (182-183). This method gives the results back to execute() method of Action class
- i) execute() returns "SUCCESS" string to ServletFilter and ServletFilter uses this string to get the result page (ref 53)
- j) ServletFilter passes the control to Success.jsp page.

Module - VI

Spring AOP

→ Spring AOP is no way related with OOP. OOP is the methodology to create programming Languages. Spring AOP is the methodology to apply middleware services on Spring applications.

→ Middleware services are the additional services that are applied on the applications to make our apps more perfect & accurate in all situations.

Eg:-

Transaction management (applies do everything or nothing principle on the given code)

Security service (protects the application from unauthorized users)

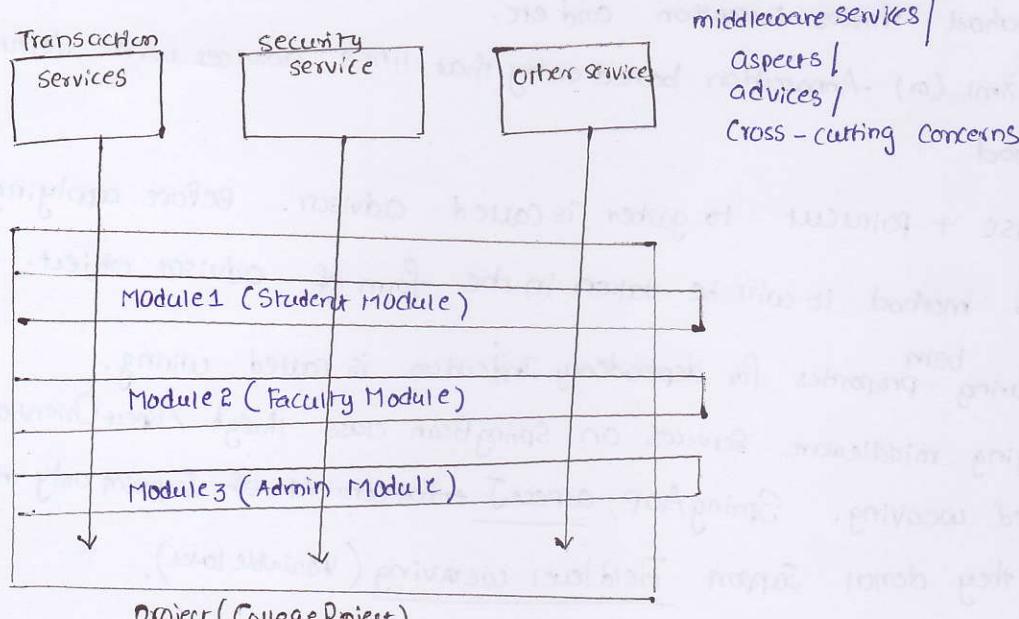
Logging service (keeps track of Application execution)

Jdbc Conn pooling (keeps set of readyly available JDBC Connection objs).

→ If we write code of middleware services directly in the application, middleware services code becomes specific to one application (no reusability) moreover the code of Middleware services may dominate application code.

→ Spring AOP allows us to write Application code and middleware services separately and allows to link middleware services code with application through xml files or Annotations. Due to this the code of middleware services becomes reusable.

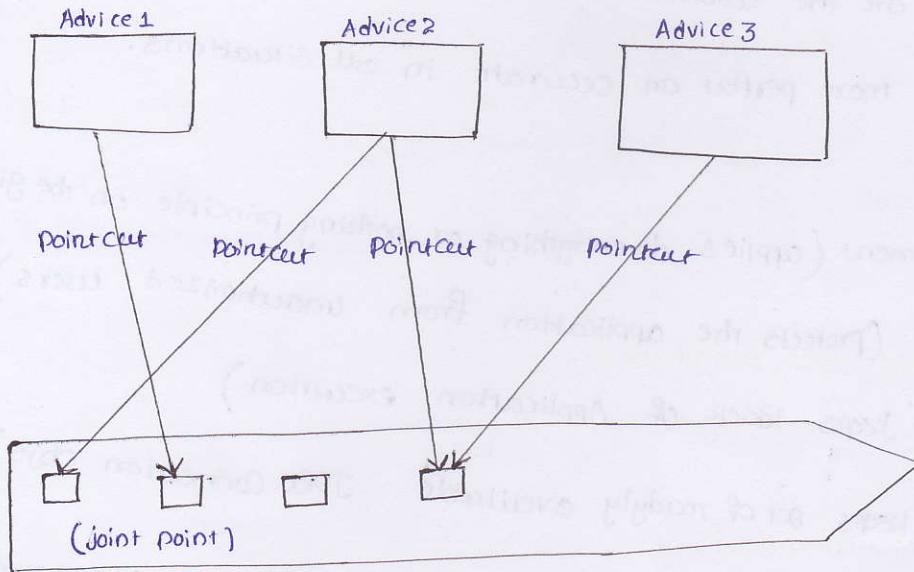
→ To understand the need of Spring AOP and understanding that refer page no's 100 to 102



Spring AOP style Middleware Services enabling

Spring AOP Terminologies

Aspect	Wiring
Advice	Weaving
Joint point	target bean / target obj
Pointcut	proxy bean / proxy obj
Advisor	



Jointpoint

Aspect :- It is the specification containing rules and guidelines. To implement the middleware services practically.

Advice :- Real middleware service which is the implementation of aspect

JointPoint :- Possible positions in our Business method where advices can be applied

EG :- beginning of B.method, end of B.method, B.method returns a value when B.method throws Exception and etc.

Pointcut :- The XML (or) Annotation based entry that links advices with joint points of B.method.

Advisor :- advise + pointcut together is called advisor. Before applying advise on Business method it will be taken in the form of advisor object.

Wiring :- Configuring bean properties for dependency injection is called wiring.

Weaving :- Applying middleware services on Spring Bean class through Aspect Oriented Programming is called weaving. Spring AOP, aspectJ enhancement of support only method level weaving.

That means they do not support Field level weaving (Variable level).

targetbean / target object :-

SpringBean class object on which we are planning to applying middleware services is called targetbean / target object.

Proxybean / proxy object :-

SpringBean class object on which middleware services are already applied is called proxy object.

→ Business methods called on target object just execute the Business method.

→ Business methods called on proxy object execute the Business method logic along with middleware services.

org.springframework.ProxyFactoryBean generates proxy object based on given target object.

There are 4 types of advices

① Before Advice (Executes at the beginning of the b.method execution)

② After Advice (Executes at the end of b.method execution)

③ Throws Advice (Executes when exception is raised in the b.method)

④ AroundAdvice (Executes at beginning of b.method and at the end of B.method)

→ using Spring AOP, AspectJ not only we can use Spring (or) servers supplied Middleware Services we can also use them to develop Userdefined Middleware Services and advices.

for the above 4 types of advices refer page no's 104 to 106.

* procedure to develop Spring AOP based Spring application.

Step 1:- Develop Java class as the advice class

Step 2:- Develop Spring Interface, Spring Bean class having Business method declarations, definitions.

Step 3:- Develop Spring Configuration File

- cfg Spring Bean class
- cfg Advice class
- cfg Advisor class pointing to Advice class
- cfg ProxyFactoryBean specifying adviser, SpringBean

NOTE:- This proxy Factory Bean gives SpringBean class object as proxy Object.

Step 4:- Client Application

- Activate SpringContainer (BeanFactory or ApplicationContext)
- gather proxy object from Spring Container
- call B.methods on proxy object

NOTE:- Business method called on proxy object will be executed along with middleware Services (Advices).

- Based on the advice we develop the joint point ~~Advice~~ will be decided automatically.
- Advisor is given to Map certain Advices with certain Business methods of SpringBean class. There are two types of Advices.

(1) NameMatchMethodPointcut Advisor

- here the advices will be linked with b.methods based on the names of the B.methods.

(2) RegexpMethodPointcut Advisor

- here the advices will be linked with b.methods based on regular expression for related information on Advisors refer page no's 108 & 109

First Spring AOP application :-

E:\ APPS

spring

ses1

Demo.java (spring Interface)

DemoBean.java (spring Bean)

MyAdvice.java (Before Advice)

SpringCfg.xml (spring cfg file)

clientApp.java (client Appn)

// MyAdvice.java (Before Advice)

```
import org.springframework.*;  
import java.lang.reflect.*; // reflection API  
import java.util.*;  
public class MyAdvice implements MethodBeforeAdvice  
{  
    public void before(Method method, Object[] args, Object target)  
    {  
        // gathering details about b.method  
        System.out.println("MyAdvice: B.method name is " + method.getName());  
        System.out.println("MyAdvice: Name of Bean class is " + target.getClass());  
        System.out.println("method.getName() + execution is started at " + new Date());  
  
        // changing B.method arg value if it is invalid  
        String s1 = (String)args[0]; // gathers b.method's 1st argument value  
        if(s1 == null || s1.length() <= 3) // set new value to b.method  
            args[0] = "satya"; // if it is invalid  
    } // method  
} // class
```

// Demo.java (Spring Interface)

```
public interface Demo  
{  
    public void sayHello(String name); // Decl of B.method  
}
```

// DemoBean.java (Spring Bean class)

```
public class DemoBean implements Demo  
{  
    // implementation of B.method  
    public void sayHello(String name)  
    {
```

S.O.P ("DemoBean : execution of sayHello() → Name: " + name);
S.O.P("In In In");

}

<!-- spring.cfg.xml -->

<!DOCTYPE >

<beans>

<bean id="db" class="DemoBean"/> <!-- BeanClass Cfg -->

<bean id="adv" class="MyAdvice"/> <!-- Advice Cfg -->

<!-- Advisor Cfg -->

<bean id="advr" class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">

<property name="advice" ref="adv"/>

<property name="mappedName" value="sayHello"/>

↳ B.method name

</bean>

<!-- This cfg gives proxy obj based on target object by applying advice -->

<bean id="pfb" class="org.springframework.framework.ProxyFactoryBean">

<property name="proxyInterfaces" value="Demo"/>

↳ Spring Interface

<property name="target" ref="db"/>

↳ Bean Id

<property name="interceptorNames">

<list>

<value> advr </value>

</list>

↳ Advisor object

</property>

</bean>

</beans>

// ClientApp.java

import org.springframework.context.*;

public class ClientApp {

public void main(String[] args)

{

FileSystemXmlApplicationContext

ctx = new

FileSystemXmlApplicationContext("springcfg.xml")

// get proxy object

Demo proxyObj = (Demo) ctx.getBean("Pfb");

// call B-methods ... on proxy obj

proxyObj.sayHello("Raja");

proxyObj.sayHello("aa");

}

/> javac *.java

/> java ClientApp

/> jar files in classpath: spring.jar, commons-logging.jar

→ to use Regular Expression PointCut Advisor place the following code in the Spring Cfg file

<!-- Advisorcfg -->

<bean id="advr" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">

<property name="advice" ref="adv" />

<property name="pattern" value=".*" />

Sample code of Around Advice

public class MyAdvice implements MethodInterceptor

{

 public Object invoke(MethodInvocation i)

i represents the executing environment of the Business method.

{

..... // executes at the beginning of the B-method

.....

i.proceed(); // calls the B-method

.....

..... // executes at the end of the B-method

.....

}

}

* for example application of the Around Advice appn(1) to appn(4) of the booklet

page no's 114 to 118

Q:- what are the spring modules that are used in your project?

Ans:- If Spring is used only in model layer then go for Spring JEE, ORM, AOP modules.

If Spring is used only in view, controller layer then go for Spring Web module.

If project is based on Struts, Spring & Hibernate Integration then go for Spring AOP, JEE, ORM, web modules.

NOTE :- Core module is the Base module for every module.

13/04/2013

* Keeping the track of flow of execution is called logging operation and messages used for this are called log messages.

* Writing log messages using `System.out.println()` is not industry standard. The reasons are

- ① → `S.O.P()` can write messages only to Console monitor. That means can't be used to write msgs to Database s/w, files, mail servers and etc...
- ② → `S.O.P()` msgs writing process is Single Threaded operation. So, multiple messages can be written at a time.

③ → `S.O.P()` does not allow to categorize the log messages.

④ → `S.O.P()` does not allow to format the log message.

To overcome these problems use Log4j api/tool to place log messages.

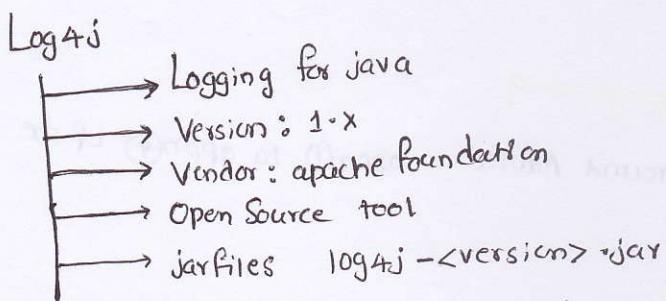
① → Log4j allows to write Log messages to different destinations like file, db, mail server, Console and etc. ...

② → Log4j can write msgs Simultaneously or parallelly.

③ → Log4j allows categorize log msgs
debug < info < warn < error < fatal

④ → Log4j allows to filter Log messages while retrieving them.

⑤ → Log4j allows to format the log messages.



`<weblogic_home>/Middleware/WLServer_10.3/server/lib/ConsoleApp/app-INF/lib/log4j-1.2.8.jar`
file represents log4j api.

→ use "debug" level confirmation stmts/ debugging stmts

· logger.debug ("I am in main()");

→ use "Info" level when certain important work is completed.

Eg:- Connection con = DriverManager.getConnection ("...");

logger.info ("JDBC Con Obj is created");

→ use "warn" level to display warning messages.

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

```
{
```

```
  ...
```

```
}
```

```
else
```

```
{
```

logger.warn ("ResultSet obj must not be empty");

```
}
```

→ use "error" level in the catch blocks of known Exceptions.

use "fatal" level in the catch blocks that handles unknown Exceptions.

```
try
```

```
{
```

```
  ...
```

```
  ...
```

```
} catch (SQLException se) //known exception handling
```

```
{
```

logger.error ("DB problem");

```
}
```

```
catch (Exception e) //unknown exception handling.
```

```
{
```

logger.fatal ("unknow problem");

```
}
```

There are 3 important objs in log4j programming

① Logger obj

② Appender obj

③ Layout obj

Logger obj :- (i) enables logging operation on given java class. (ii) allows to create log messages having different categories. (iii) It Base obj for Log4j based logging operations.

```
Logger logger = Logger.getLogger (TestApp.class);
```

Static Factory method

obj of java.lang.class pointing to TestApp class

```
logger.debug("hello1");
logger.debug("hello2");
```

Appender Obj:-

- specifies the destination object for which we can write log messages (like dbappending, fileappending, etc..)
- will be linked with logger object to take and write log messages.
- The configurable appenders are
 - File Appender (to write to files)
 - Console Appender (to write to Console)
 - JDBC-Appender (to write to DB like)
 - IM Appender (to write to mailserver and etc..)

Layout Obj:-

- allows to specify the format and pattern of the log messages.
- It can be used to customize the look and feel of Log messages.
- The available Layout classes are.

HtmlLayout

SimpleLayout

PatternLayout

XmlLayout and etc...

NOTE:- If the loggerLevel retrieve level log messages info we will get only those log messages whose logger level is \geq Info.

- (*) for Spring AOP example appn that uses all the 4 types of advices refer appn(5) of the page no's (119) to (121)

Transaction management :-

executing them

- The process of combining the related operations into Single Unit by applying do everything or nothing principle is called Transaction management.

- All sensitive logics like transfer money, debit card processing must be developed by enabling Transaction mgmt Service.

- transferMoney (-, -)

- Transfer money operation is the combination of two operations.
- ① Withdraw amount from Source account.
 - ② deposit amount into destination account.

It is recommended to execute these logics by enabling do everything or nothing principle

- we can bring the effect of Transaction Mgmt using try/catch blocks manually, but it makes code unmanageable. To Overcome this problem always prefer using Server Managed (or) Technology

Supplied Transaction mgmt Service.

- (i) In hibernate apps we use Technology supplied Transaction Service.

- (ii) In Spring apps we can use either Technology supplied Transaction Service (or) Server Managed Transaction Service.

- (iii) In EJB compns we use Server managed Transaction Service.

The typical code of Transaction management :-

```

public void bmn()
{
    try {
        Begin Transaction
        Operation 1.....
        Operation 2.....
        Operation 3.....
        Commit the Transaction
    } catch (Exception e) {
        {
            rollback Transaction
        }
    }
}

```

Transaction mgmt gives Support for ACID properties implementation

- A → Atomocity
- C → Consistency
- I → Isolation
- D → Durability

→ The Process of Combining multiple & related individual Operations into Single Unit is called "Atomocity"

The word "Atomocity" come from the word "Atom"

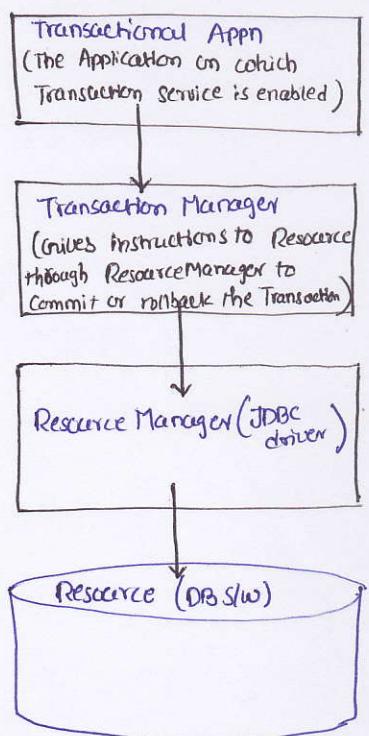
→ Consistency :- Even though rules kept on DataBase data are Violated during the course of Transaction in there is a guarantee for no rule Violation at the end of

Transaction mgmt code. then that code is called as Consistent Code.

→ Isolation :- The process of applying lots of amount of data and allowing only one user on DataBase to manipulate the data is called enabling Isolation. This Isolation prevents Concurrency and Simultaneous Operations on DataBase data.

→ Durability :- The ability of bringing DataBase s/w back to normal state through logfiles and backup files when database is corrupted. is called making DB as the Durable database.

The Architecture of Transaction Management



Transaction Manager commits or rollbacks application data in a DataBase s/w through Resource Manager to based on the Commit (or) rollback operations that comes from Transactional appn.

→ based on the no. of resources (Database Slices) that are involved to perform various operations of Transactional code we can say there are 2 types of Transactional managements.

① Local Transaction (Tx) Management.

→ all the operations of Transactional code will act on Single DB slice

Eg:- transfer Money operation b/w two diff accounts of same bank.

② Distributed Transaction (Tx) Management.

→ The operations of Transactional code will act on Multiple DB slice

Eg:- Transfer Money operation b/w two accounts of two diff banks.

→ Hibernate, Spring, EJB Technologies support Local Transaction mgmt.
Hibernate does not support Distributed Transactions. whereas Spring, EJB supports Distributed Transactions.

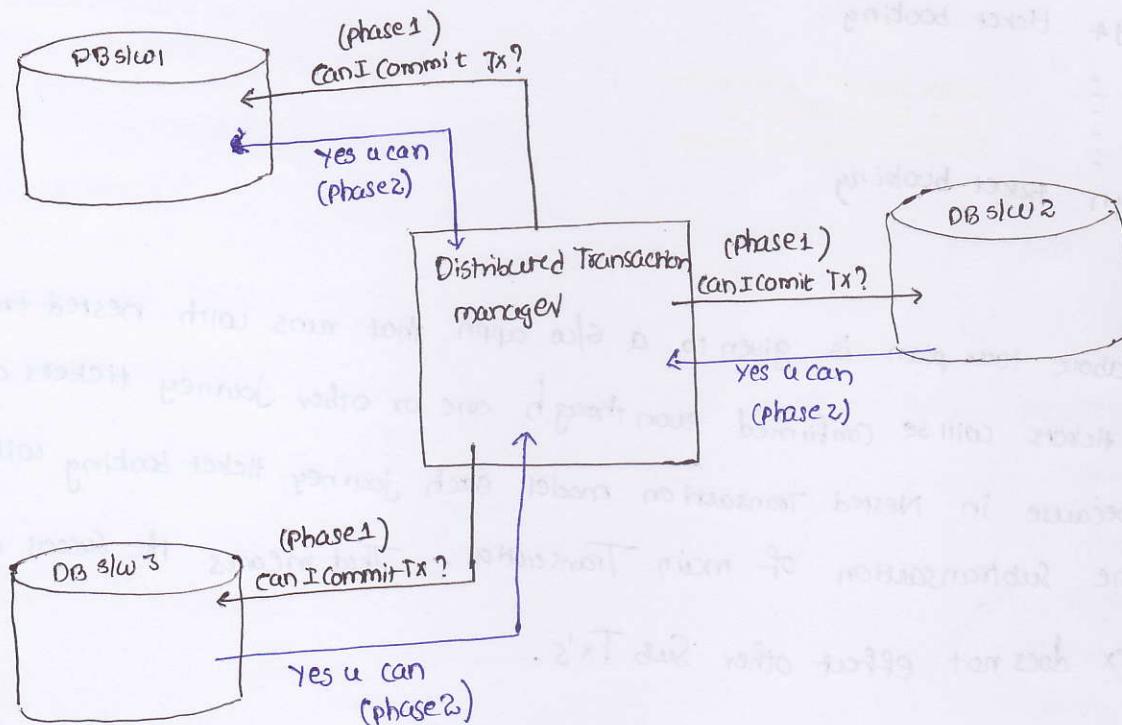
15/04/2013

→ Distributed Tx's run on RPC (2 phase commit) protocol.

In phase 1, The Distributed Tx manager will seek permission from DBS/Ws to Commit the Tx.

In phase 2, if all DBS/Ws replay yes then the Distributed Tx will be committed

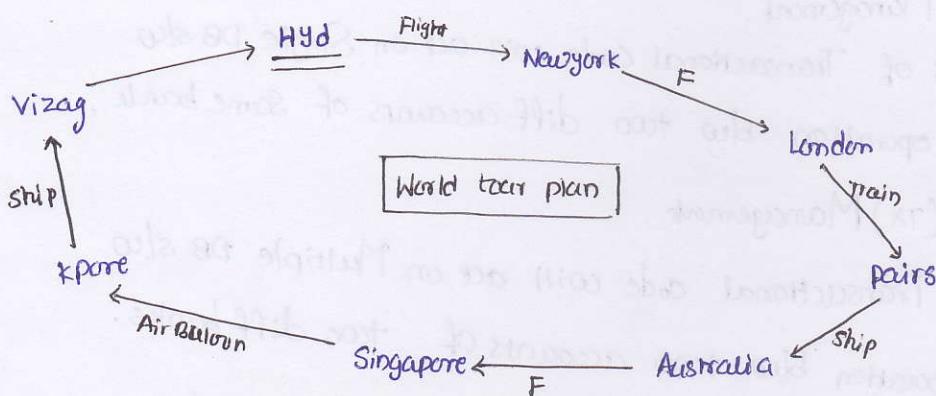
Otherwise the Distributed Tx will be ~~not~~ rollback.



Transaction models

Flat Transaction Model

Nested Transaction Model



The transaction will be rolled back if one or other journey tickets are not available because, in Flat Transaction model all journey ticket booking operations will be placed as direct operations of main transaction.

Flat Transaction Model

Main Tx {

journey1 ticket booking

journey2 ticket booking

journey3 ticket booking

journey4 ticket booking

⋮

journey n ticket booking

}

When the above tour plan is given to a s/o appn that runs with nested transaction the journey tickets will be confirmed even though one or other journey tickets are not available. because in Nested Transaction model each journey ticket booking will be taken as the subtransaction of main Transaction. That means the success or failure of One SubTx does not effect other SubTx's.

Nested Transaction model :-

Main Tx {

→ EJB, hibernate, spring supports Flat Tx model

Sub Tx 1

→ hibernate, EJB do not support Nested Tx Model

{
journey1 Ticket booking

→ Spring supports Nested Transaction.

Sub Tx 2

{
journey 2 Ticket booking
}

=
=

Sub Tx n

{
journey n Ticket booking
}

}

→ We can apply Transaction mgmt Service on Spring Applications in two ways

1) programmatic Approach (write code directly in Spring Bean class b. methods)

(no need of working with Spring AOP module)
(we can use TransactionTemplate class to simplify this Tx mgmt.)

2) Declarative Approach (cfg Transaction Service by using Xml or annotations)

(we need to work Spring AOP module or AspectJ concepts)

→ Declarative Approach is always recommended to use... it keeps our spring bean class b. methods clean and neat.

→ To enable Transaction mgmt on diff kinds of persistence logics we need to work with diff kinds of Transaction managers.

① DataSourceTransactionManager

→ To enable Tx mgmt on jdbc persistence logic.

② HibernateTransactionManager

→ To enable Tx mgmt on plain hibernate or Spring ORM hibernate persistence logic.

③ ToplinkTransactionManager

→ To enable Tx mgmt on Toplink persistence logic.

④ JDOTransactionManager

→ To enable Tx mgmt on JDO persistence logic

⑤ JtaTransactionManager

→ To enable Tx mgmt on Spring appn by using Service Manager Tx service.

and etc...

* To create TransactionTemplate class obj TransactionManager obj is the base object

* for Spring DAO module based local, Flat Transaction management in programmatic approach
refer appn(2) of the page no's 131 & 132

TransactionTemplate class provides the abstraction layer on Programmatic Transaction management by automatically begining the Transaction . Committing (or) roll back the Transaction.

- Avoid programmatic Transaction mgmt in Spring appn because it makes the programmer to write more code of Transaction mgmt along with the Business Logic.
- To overcome this problem use Spring AOP / AspectJ base Declarative Tx mgmt.
- Passing the Transaction started in one resource to another resource is called propagation of Transaction. In programmatic Tx mgmt ~~the client appns~~ ~~related to~~ the Business method always execute with new Transaction because it can't work ~~with~~ Client Supplied Transaction. This indicates Programmatic Tx mgmt does not Support propagation of Transaction.
- Declarative Tx mgmt Supports Transaction propagation. That means B-methods can run
 - ① In client started/^{supplied} Tx service
 - ② In new Tx that is started in b.method itself
 - ③ with no Tx.
- In Declarative Tx mgmt, Business method contains only ^{Business Logic and the logic of} annotations. Transaction Service will be Configure through XML files ^(or) annotations.
- Weather the b.method on which Declarative Tx mgmt enabled runs with no Transaction (or) new Transaction (or) Client Supplied Transaction will be decided base on the Transaction attribute that is Configured on the business method.
- The 6 Tx attributes are

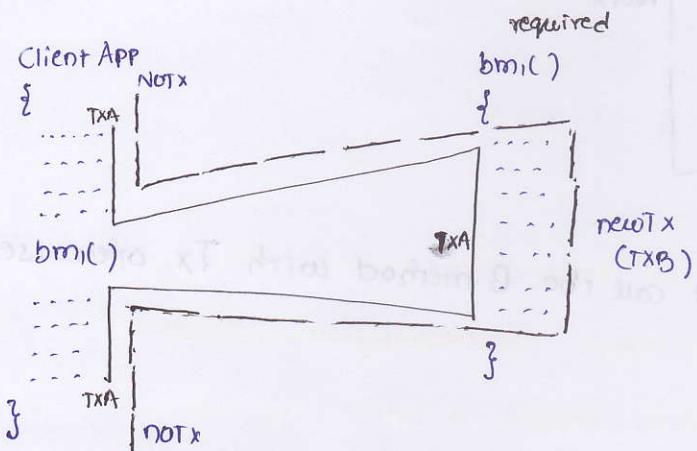
- 1) required
- 2) required new
- 3) supports (default)
- 4) Not Supported
- 5) Mandatory
- 6) Never

16/04/2013

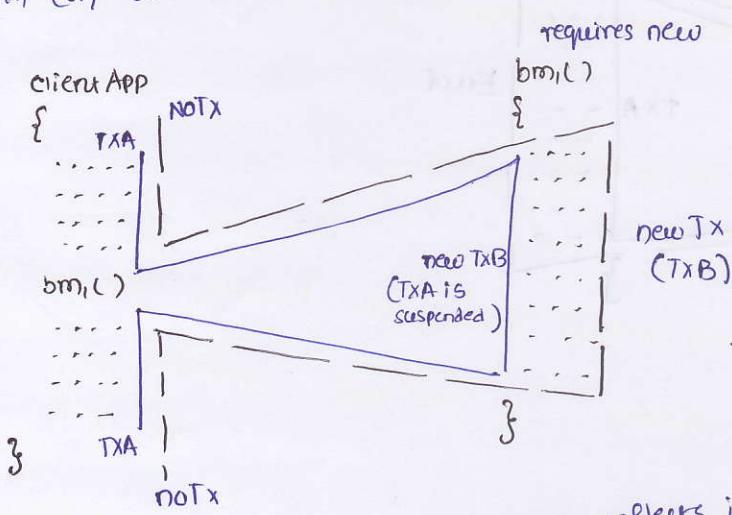
Note:- Tx attributes are not required while dealing with programmatic Tx mgmt.

Note:- TransactionTemplate class not required while dealing with Declarative Tx mgmt.

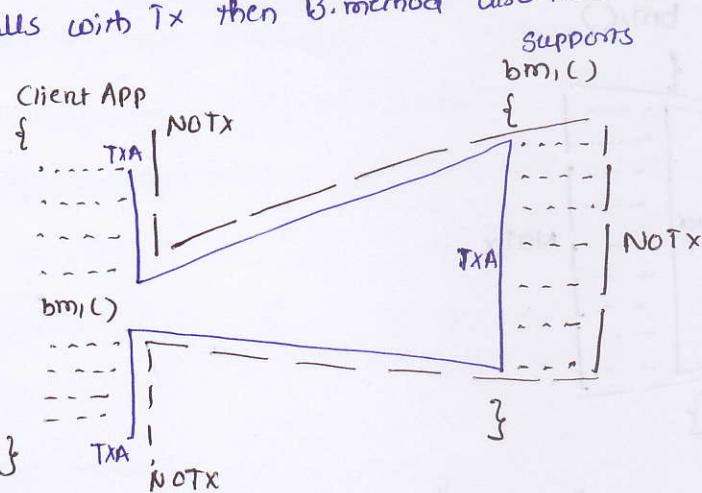
Required :- B-method uses client supplied Tx if client appn calls the B-method with Tx. Otherwise new Tx will be started in the B-method.



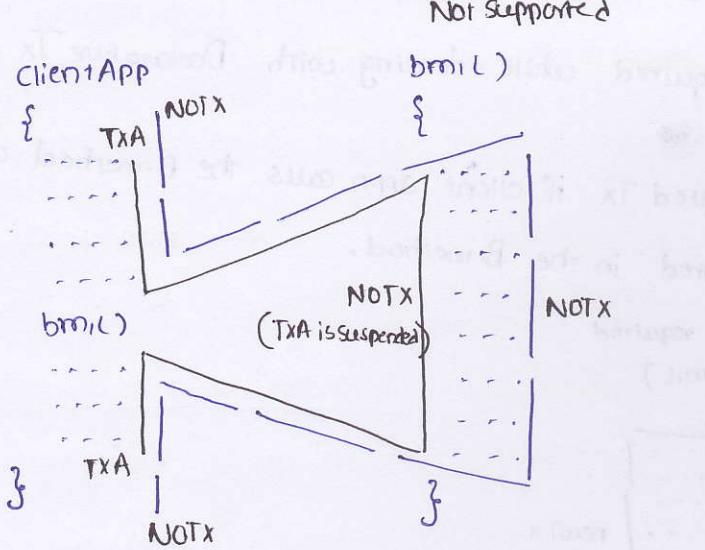
Requires new:- This B-method runs with new Tx irrespective of whether client calls B-method with (or) without Tx



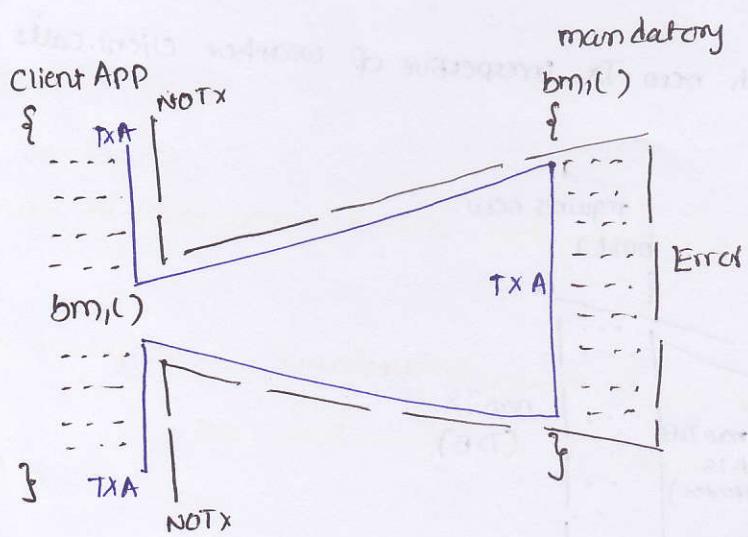
Supports:- The state of Tx in client appn also reflects in B-method. That means if client appn calls with Tx then B.method also runs in the Tx. Otherwise B.method runs without Tx.



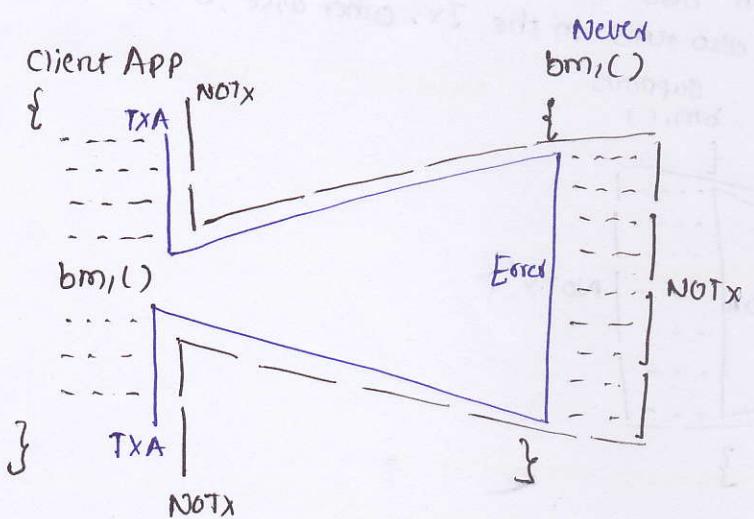
Not supported:- B.method always runs without Tx irrespective of whether client appn is calling B.method with or without Tx.



- ⑤ Mandatory :- The client appn must call the B.method with Tx otherwise B.method throws an Exception.



- ⑥ Never :-



most popularly used Tx Attribute is "required".

The Default Tx attribute is "support".

The Effect of Tx attributes :-

Transaction Attribute	Client's Transaction	Bean's Transaction
Required	none T ₁	T ₂ T ₁
RequiredNew	none T ₁	T ₂ T ₂
Supports	none T ₁	none T ₁
Mandatory	none T ₁	error T ₁
NotSupported	none T ₁	none none
Never	none T ₁	none error

→ org.springframework.transaction.interceptor.TransactionProxyFactoryBean returns proxy object on which Tx service is enabled.

→ org.springframework.transaction.interceptor.NameMatchTransactionAttributeSource is like pointcut advisor to apply Transaction attributes on the b-methods of Spring Bean class.

→ In Declarative Transaction Management the Transaction will be committed when no exception is raised in the Business method. otherwise the Transaction will be rolled back.

* For example upon on Declarative Flat, Local Transaction mgmt refer appn① of the page no's 130 & 131

→ Declarative Tx mgmt by using annotations :-

//Demo.java Same as page no: 130 Demo.java

//DemoBean.java Same as 130 DemoBean.java but modify following things.

- a) add import org.springframework.transaction.annotation.*;
- b) add @Transactional (propagation = Propagation.REQUIRED) on b.m() b.method
 - ↳ Tx attribute

//Demo.xml
a) must be schema base including "spring-beans.xsd, spring-tx.xsd, spring-aop.xsd"

- b) place line no 42 to 63 of page no: 130 as it is here.
- c) add following code at the end of (b) step
- ```
<tx:annotation-driven transaction-manager="dts" />
```

ClientApp.java :-

same as the client appn. java of page no: 131

JAR files in CLASSPATH :-

spring.jar, commons-logging.jar (spring 2.5 envi--)

>javac \*java

>java ClientApp

Other Tx attributes while working with @Transactional  
Propagation.REQUIRED, Propagation.REQUIRES\_NEW,  
Propagation.SUPPORTS, Propagation.NOT\_SUPPORTED, Propagation.MANDATORY,  
Propagation.NEVER, Propagation.NESTED (to enable nested Tx's)

17/04/2013

- To apply Spring based Transactions on Spring ORM module based hibernate persistence logic use org.sf.orm.hibernate3.HibernateTransactionManager
- For example application on programmatic, local, Flat Tx management on SpringORM module based Hibernate Persistence Logic refer appn(1) of the page no's ④ to ⑦ of Supplementary handout given on 17/04/2013
- For example appn on Declarative, local, Flat Tx's on Spring ORM module based persistence Logic refer appn(2) of the page no's ① to ③ Supplementary handout given on 17/04/2013

## \* Global / Distributed Transactions \*

18/04/2013

- If multiple Databases are involved in Transaction management then it is called Global (or) Distributed Tx's. for this we need Jta Transaction manager and its implementation.

Atomikos supplies special jdbc drivers and JtaTransactionManager implementation to add Distributed Tx mgmt support on Spring appns.

U can download related resources from [www.atomikos.com](http://www.atomikos.com) website.

Transfer money operation b/w two diff a/cs of two diff banks need Global Transaction Support.

→ mysql DB s/w

Account2 (db table)

| accno | holdername | bal  |
|-------|------------|------|
| 102   | ravi       | 9000 |

→ Oracle DB s/w

Account1 (db table)

| accno | holdername | bal  |
|-------|------------|------|
| 101   | raja       | 8000 |

→ we need Distributed Tx mgmt support to perform transfer money operations on the above two accounts....

\* for example appn on Declarative and Distributed Tx mgmt on Transfer money operation refer appn (given on) available in 18/04/2013 handout

## \* Security \*

### Security

1. Network Level Security

2. Application Level Security

#### Network Level

→ To protect the data that is travelling over the network

→ Application developers are not responsible for this, Network admins will take care of this through Firewalls and others

#### Application Level

→ Authentication + Authorization

→ programmers are responsible for this operation

- Checking the Identity of a User is called Authentication.
- Checking the Access Permissions of a User on resources is called Authorization.
- It is always recommended to enable Security Services on application in Declarative mode. ~~so~~ we can use Acegi framework (or) Spring Security fw to enable the declarative mode of security on Spring apps.

(\*) For example appn on aspectJ programming refer the appns given in ~~handout~~ 17/04/2013

20/04/2013

## → AspectJ Annotations :-

- ① `@Aspect` → To make the java class as the Advice class.
- ② `@PointCut` → To point to the methods of Spring Bean class.
- ③ `@Before` → To make the java methods the Before advice.
- ④ `@AfterReturning` → To make the java method as the After advice.
- ⑤ `@AfterThrowing` → To make the java method as Throwing Advice.
- ⑥ `@Around` → To make the java method as the around advice.

\* for above annotations based AspectJ programming refer the following example.

TestInter.java → same as page no: 10 TestInter.java of 17/04/2013 handout.

TestBean.java → " " TestBean.java of "

// MyAspect.java

```
import org.aspectj.lang.annotation.*;

@Aspect
public class MyAspect
{
 @Pointcut ("execution(* TestInter.*(..))") }
 {
 public void testinterOK()
 {
 System.out.println("iam before advice");
 }
 @Before("testinterOK()")
 public void beforeMethod()
 {
 System.out.println("iam before advice");
 }
 @AfterReturning ("testinterOK()")
 public void afterReturningMethod()
 {
 System.out.println("iam after returning advice");
 }
 }
}
```

pointcut pointing to all the methods of Spring Bean

Spring.xml  
<beans>  
 <!-- include spring-beans-2.5.xsd  
 spring-aop-2.5.xsd -->

```
<bean id="id1" class="TestBean"/>
<bean id="id2" class="MyAspect"/>
<aop:aspectj-autoproxy />
```

</beans>

Client.java Same as Client.java of  
page no: 10 belonging to April 17th handout.

JAR files in CLASSPATH: aspectjrt-1.6.0.jar, spring.jar  
javac & java aspectjweaver.jar, commons-logging.jar  
java client

## Spring WebModule part-II

Web framework like provide abstraction layer on core servlet, jsp technologies and simplifies the process of developing MVC architecture based webapps.

Struts → From apache foundation

JSF → From Sun/MG (Oracle Corp)

Webwork → From open symphony

Spring Web mvc / spring mvc / spring web flow → From interface21

What is the diff b/w Struts 1.x and Spring mvc?

### Struts 1.x

- ① resources core API dependent
- ② does not support Annotation based programming
- ③ does not allow to use other than html, jsp technologies in view layer
- ④ supports dependency injection only on Formbeans
- ⑤ Validator plugin is given to perform FormValidations

### Spring mvc

- ① resources can be developed as POJO classes
- ② supports annotations base programming
- ③ allows to use Velocity, freemarker, JSP, HTML technologies in view layer.
- ④ supports dependency injection on every Java resource.
- ⑤ uses spring's built-in facilities to perform FormValidation

Terminology matching b/w Struts 1.x and Spring MVC

### Terminology

Controller Servlet

### Struts 1.x

ActionServlet

FormBean

Action class

Struts cfg file

<Any filename>.xml

### Spring mvc

DispatcherServlet

command class

Command Controller class

Spring cfg file

<DispatcherServlet>logicalname>-> <sevlet.xml>

ActionMapping

uri handler

ActionForwards

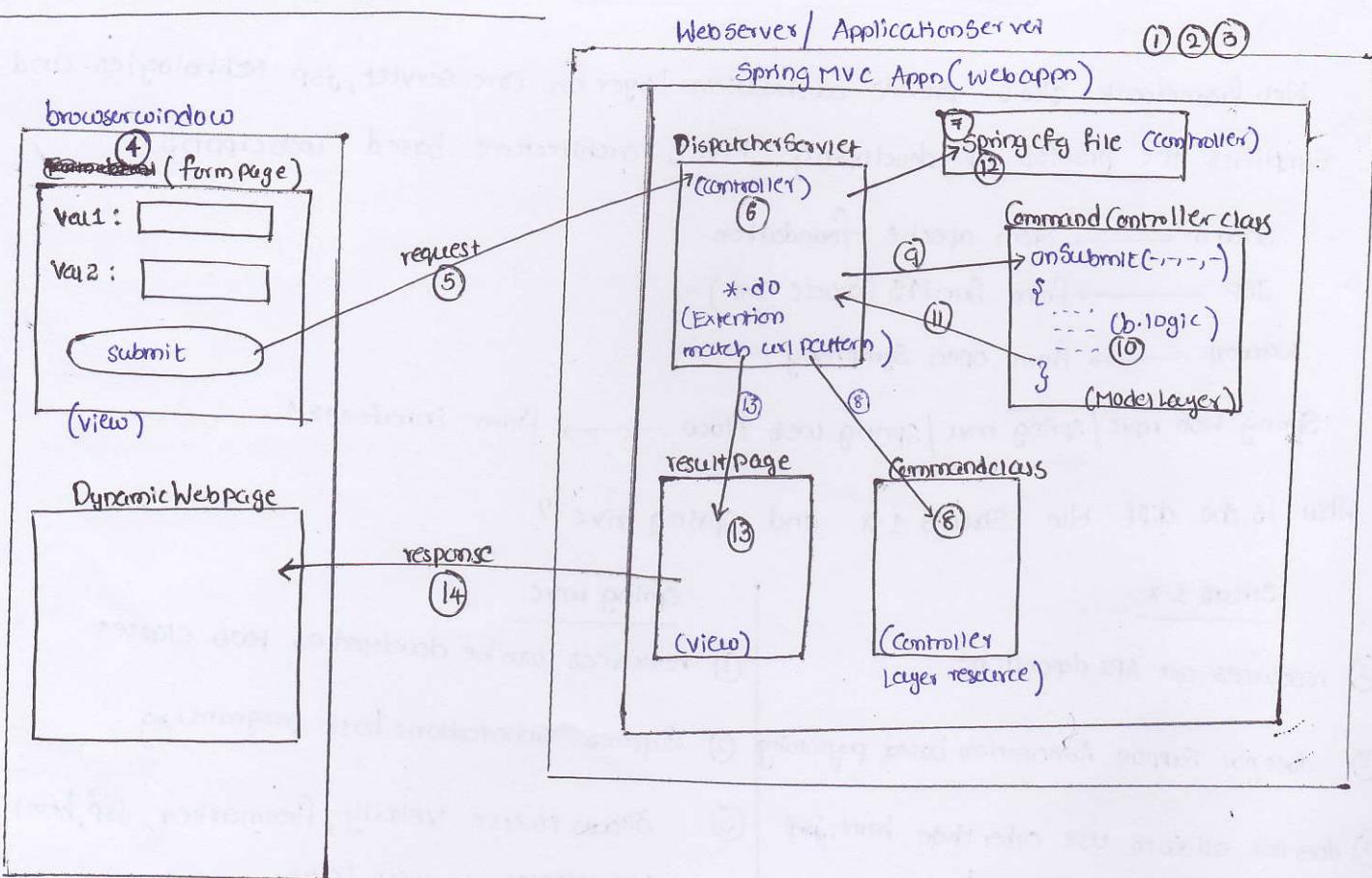
(to configure result pages)

View Resolvers and Views

Model And View object

Same as Struts 1.x flow

## Flow of execution in Spring MVC appn :-



w.r.t to the diagram :-

- ① programmer deploys Spring MVC web appn in Webserver (or) appn server.
- ② because of load-on-startup enabled on the DispatcherServlet it will be instantiated either during Server startup or during the deployment of webappn.
- ③ This DispatcherServlet activates web appn Context container by taking ~~DispatcherServlet logical name~~ <DispatcherServlet logical name> - `servlet.xml` as the Spring Configuration file. In this process all the beans Configured in Spring Configuration file will be pre-instantiated.
- ④ End user launches browser and submits the request to webappn.
- ⑤ End user Form page on the browserwindow and submits the request to webappn.
- ⑥ DispatcherServlet traps and takes the request
- ⑦ DispatcherServlet uses Uri-handlers Configuration to know Command Controller class, Command class that are required to process the request.
- ⑧ DispatcherServlet writes the received Form data to Command class object.
- ⑨ DispatcherServlet calls certain B-method of Command Controller class
- ⑩ & ⑪ This B-method will process the request and returns Model And View class object to Dispatcher Servlet.
- ⑫ DispatcherServlet uses Views and ViewResolvers configured in Spring Cfg file to decide the result pages.
- ⑬ DispatcherServlet passes the Control to ResultPage.
- ⑭ The presentation Logic of ResultPage formats the results and sends response to browser window.

- Command class is POJO class, ..., and java bean
- Command class properties need not to match with FormComponent names.
- Command class properties must be bound with FormComponents by using `<spring:bind>` tag.
- Command Controller class is Spring API dependent. This class must implement `org.springframework.web.servlet.mvc.Controller` (Interface)

the Spring Configuration file of Spring WebMVC appn contains the following cpgs.

### ① Uri handlers

(to link incoming requests with Command controller classes)

### ② Command controller classes & Command classes.

### ③ View Resolvers.

View Resolver :- This configuration lets the Spring file to specify the technology that is

required to develop the resources of View Layer. This ViewResolver class name varies based on the ViewLayer technology we use.

#### Technology

#### viewResolver class name

JSP

`org.springframework.web.servlet.view.InternalResourceViewResolver` (default)

velocity

`org.springframework.web.servlet.view.velocity.VelocityViewResolver`

freemarker

`org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver`

tiles

`org.springframework.web.servlet.view.tiles.TilesConfigurer`

XSLT (XmL Stylesheet Language  
for Transformation)

`org.springframework.web.servlet.view.xslt.XsltViewResolver`

jasper reports

`org.springframework.web.servlet.view.jasperreports.JasperReportsViewResolver`

#### UriHandlers :-

→ these are the classes configured in Spring Cfg file to link the client generated incoming request with Command Controller classes based on their Request Uri's. There are two important Uri

handlers. `SimpleUrlHandlerMapping`

(recommended to use)

① `org.springframework.web.handler.SimpleUrlHandlerMapping` :- This class allows to use bean ids, bean names and even regular expressions to link from uris (request uris) with Command Controller classes.

② `org.springframework.web.handler.BeanNameUrlHandlerMapping` :- This class allows to use bean names (that starts with "/" symbol) to link from uris (request uris) with Command Controller classes.

Command Controllers :- There is a possibility to develop different types of Command Controller classes.

- ① org.sf.web.servlet.mvc.SimpleFormController :- useful to handle single submit button form page, here the Formpage must be launch under the controller of ControllerServlet. gives onSubmit(..) method to process the request
- ② org.sf.web.servlet.mvc.AbstractFormController :- useful to handle single submit button form page, here the form page can be launched directly. gives handleRequest() method to process the request.
- ③ org.sf.web.servlet.mvc.AbstractWizardFormController :-
- ④ org.sf.web.servlet.mvc.CancellableFormController :- same as SimpleFormController but also provides onCancel(..) to process the request coming from Cancel button (reset btn).
- ⑤ org.sf.web.servlet.mvc.mutiaction.MultiActionController :-  
Useful to handle the multiple request coming from the multiple Submit buttons of Formpage

22/04/2013

Some points about SimpleFormController :-

- When this class based CommandController class gets "GET" method based request then the form page will be launched.
- When this class based CommandController class gets "POST" method based request then the request will be processed and result will be generated.

The important properties are :-

- a) SessionForm → to specify the scope of Command class Obj
- b) formView → to specify the form page.
- c) successView → to specify the result page.
- d) commandName → to specify the Command class logical name.
- e) commandClass → to specify the Command class name  
and etc...

Above diagram W-I-O diagram

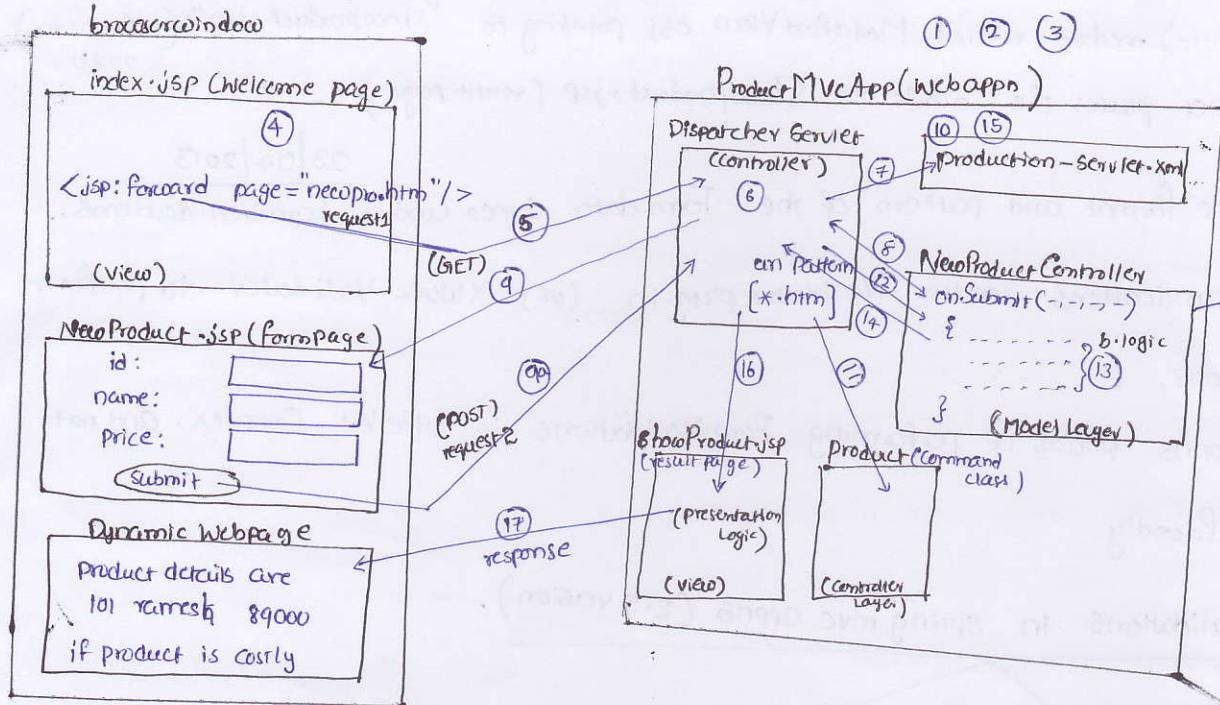
⑥ Welcome page generates "GET" method based implicit request

⑦ The Dispatcher Servlet Traps & takes the request.

⑧ ⑨ Dispatcher Servlet links the request with Command Controller class. Since the request method is "GET" the Command Controller class makes DispatcherServlet to launch the Formpage.

⑩ Dispatcher Servlet Traps the request and links the request with Command Controller class. Since the request method is "POST" the Command Controller class becomes ready to process the request.

## Example Application



- (11) Dispatcher Servlet writes the received formdata to Command class object.
- (12) Dispatcher Servlet calls onSubmit (-,-,-) method on CommandController class obj.
- \* For the above code base example appn reffer apn (24) of the page no's (78) (79)

The Flowof execution of Appn (24) page noo: (78) & (79) :-

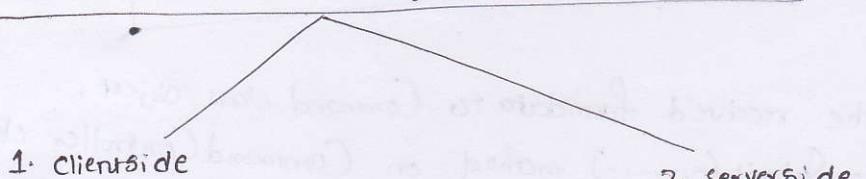
- (A) Same as step1 21/04/2013. Diagram
- (B) same as step2 of 21/04/2013 Diagram
- (C) same as step3 of 21/04/2013 Diagram
- (D) Enduser gives request to Webappn (SpringMVC). in this process the welcomepage Index.jsp will execute and this welcome page generates implicit request to Webappn having "newpro.htm" in the request URL (2658 reffer line no) (GET method)
- (E) DispatcherServlet traps and takes the request (2695-2703).
- (F) The SimpleUriHandlerMapping class links the request with CommandController class (NewProductController) (2720, 2725)
- (G) Since CommandController class type is SimpleForm Controller and the request method is GET so the form page gathered from "formView" property ("newproduct.jsp") will be displayed on the browser window through DispatcherServlet. (ref: 2729 & 2659)
- (H) EndUser fills up the form page and Submits the request (ref: 2684)
- (I) DispatcherServlet traps and takes the request (2695-2705)
- (J) Same as (F)
- (K) The CommandController class (NewProductController) becomes ready to process the request because the request method is "POST"
- (L) DispatcherServlet writes the received form data of form page to Command class obj. (use setXXX(-) methods (ref: 2747, 49, 51))

- (M) DispatcherServlet calls onSubmit(----) method of CommandController class obj. (2761-2772)
- (N) onSubmit(----) method returns ModelAndView obj pointing to "showproduct.jsp" (ref: 2770).  
2730
- (O) DispatcherServlet parses the Control to Showproduct.jsp (result page).

23/04/2013

- Verifying the Format and pattern of the Form data comes under FormValidations.
- In struts applications we use Validator plug-in (or) Xwork Validator to perform Form Validations.
- SpringMVC appns process of performing FormValidations is little bit Complex. And not programmer friendly.

### FormValidations in Spring mvc appns (2.5 version)



1. ClientSide  
only programmatic FormValidations are possible using javascript

2. ServerSide  
→ only programmatic form Validations are possible using java code

→ we can use Various methods of ValidatorUtils class for simplifying the process of form Validation.

→ Spring allows to develop FormValidation logics in separate java class and this java class can be used in all kinds of Spring appns including SpringMVC appns. This class must implements org.springframework.validation.Validator(I) and must provide implementation for supports, validate() methods. While writing logics in validate() method we can take the support of rejectXXX() methods of ValidationUtils class.

\* Procedure to develop the SpringMVC appn having the Support of ServerSide FormValidations

Step-I:- Take a java class implementing org.springframework.validation.Validator(I) and implement the FormValidation logic.

refer ProductValidator.java file of page no: (81) appn no: (25).

Step-II:- Specify the above class in Command controller class Configuration of Spring Configuration file.  
refer 2868 entries of page no: (81)

Step-III:- Write following logic in the Form page by replacing the code of 2814 to 2818 of page no (80)

```

<spring:hasBindErrors name="logical name of Command Controller classP">

 <c:forEach var="e" items="represents all Errors ${errors.allErrors}">
 it contains all the Error messages (represents all FormValidation errors)
 ^{at contains only one validation error in each iteration.}

 </c:forEach>


```

<spring: message text="\${e.defaultMessage}" /> → It is foreach loop so, one by one Validation errors displayed.

</li>  
</c:forEach>  
</ul>  
<spring:hasBindErrors>

24/04/2013

## Important annotations of SpringMVC (related to Spring 3.x version)

- ① `@Controller` → makes the java class as Command Controller
  - ② `@RequestMapping` → maps the input request url with CommandController class.
  - ③ `@ModelAttribute` → keeps the parameter of given b.method in the ModelAttribute to use in View layer.
  - ④ `@SessionAttributes` → creates Session attribute in web resource prgs.
- \* for Annotations based SpringWebMVC example appn refer the appn(1) of the page no's (1) & (2). 24/04/2013
- If you want to use special Springbeans as ViewLayer resources and to make SpringMVC appns recognizing them we need to work with Special View Resolver called BeanNameViewResolver `org.springframework.web.servlet.view.BeanNameViewResolver`. To generate Results in the form of PDF document, X-CEL sheets we don't have ViewResolvers. so, we write those logics in special Springbeans and we make them as ViewLayer resources.
- \* for the example appn on the above discussion the appn(3) of the page no's (5)(6)(7) 24/04/2013
- In SpringMVC to handle the situation of multiple submit button of 25/04/2013 a formpage we must use the CommandController class of type "MultiActionController" and we should also use `org.springframework.web.mvc.mutiaction.ParameterMethodNameResolver` to link the Formpage generated request with methods of CommandController class.
- \* for example appn on MultiActionController refer the application(2) of 24/04/2013

## Spring - webservices

- It provides the abstraction layer on JAX-RPC, JAX-WS webservices. In this process the Spring Container itself provides environment to execute webservice Component. that means there is no need of Separate webserver.
- Spring 3.x Supplies two classes to expose java class as webservice Component and to Consume webservice being from client appn.  
org.springframework.remoting.jaxws.SimpleJaxWsServiceExporter can export given Java class Object as webservice Component.

org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean can be used to Consume webservice from Client application.

\* for example appn on Spring 3.x based Spring Webservice ref for appn ① on 24/04/2013

I would try to update our site [JavaEra.com](http://JavaEra.com) everyday with various interesting facts, scenarios and interview questions. Keep visiting regularly.....

**Thanks and I wish all the readers all the best in the interviews.**

**www.JavaEra.com**

A Perfect Place for All **Java Resources**