www.tutorial4us.com

Tutorial4Us

College Projects

```
void main()
{
 int no;
 clrscr();
 printf("Hello");
}
```

Basic Program

C Tutorial

JAVA Programming

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

08/02/2013

→ Using WebFrameworks we can just develops Web applications.

Using ORM Frameworks we can just develop Persistence Logics.

Using Spring we can develop all kinds of logics like Business Logic, Persistence Logi

Integration Logic, presentation Logic and etc.

→ Using Spring we can develop all kinds of applications like Standalone appns, 100-tire applications, web applications, Distributed applications, enterprise appn & etc

Spring

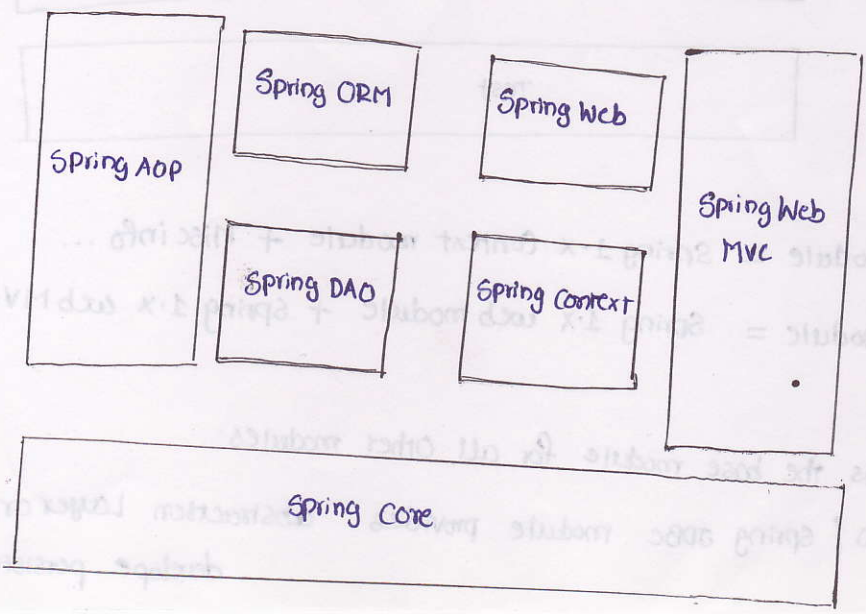
- Type: Java-JEE Framework | Application Framework
- Vendor: Interface 21
- Creator: Mr. Rad Jhonson
- OpenSource s/w
- To Download s/w: Download as Zipfile from www.springframework.org
- Version: 2.5, 3.x — 2.5 (compatible with jdk1.6+); 3.x (compatible with jdk1.5+)
- for docs and FAQs: www.springframework.org
- for online Tutorial: wwww.roseindia.net
- for good articals: wwww.java4s.com & wwww.mkyoung.com & wwww.nataraz.in

Books: Spring in Action From manning publisher

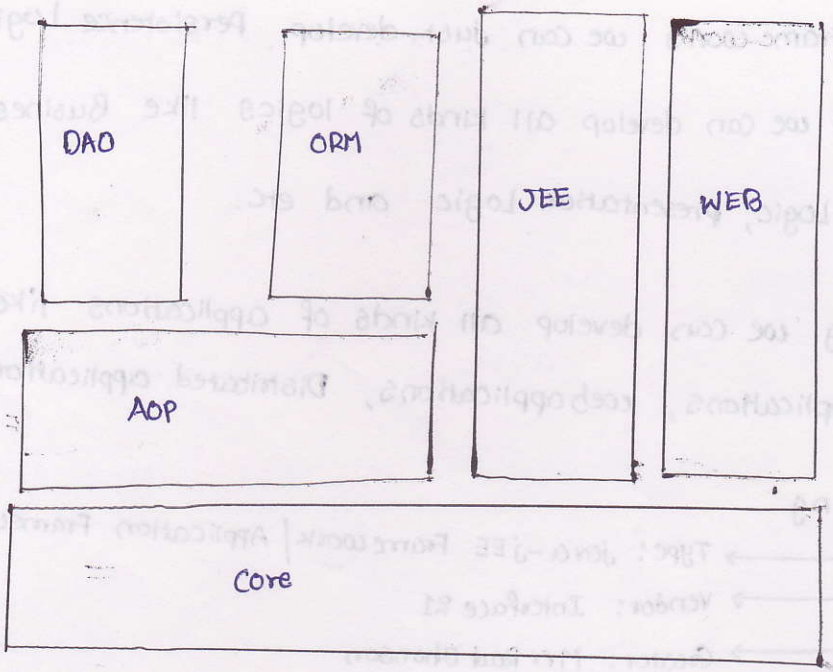
→ Spring 1.x contains 7 modules. where as spring 2.x contains 6 modules.

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

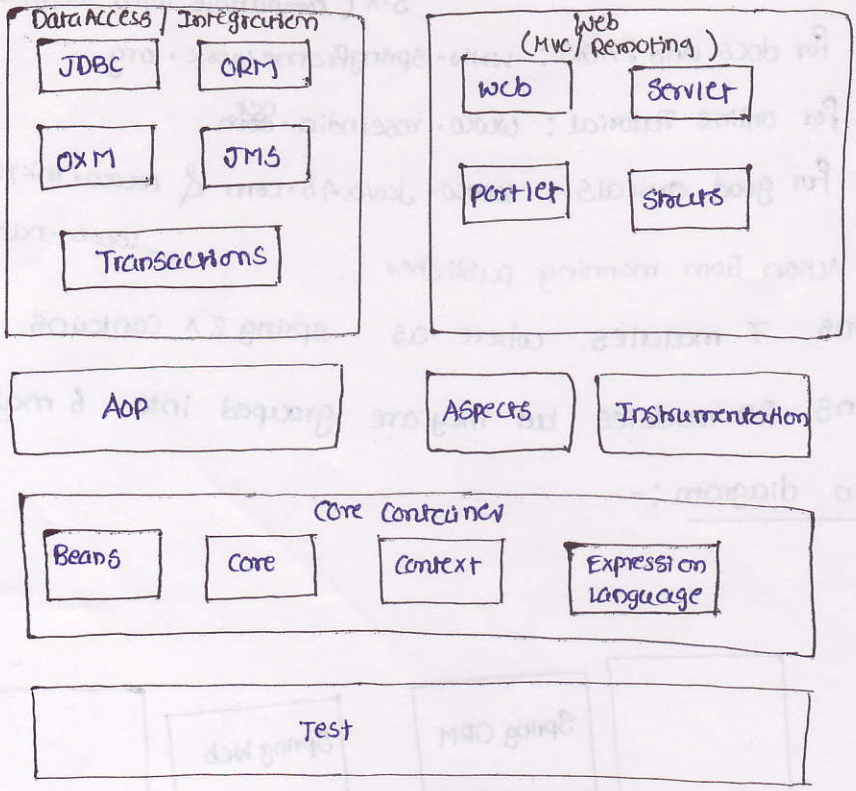
→ Spring 1.x overview diagram :-



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 develope persistence logic

→ Spring ORM module provides abstraction layer on other ORM Fws 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 framework s/w applications. like Struts appn, JSF appns etc.

→ Spring Web-mvc is given Spring's owns web Framework develop MVC2 architecture based web application.

→ Spring context module/ JEE module provides abstraction Layer 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 occurate.

Eg:- Security, Transaction mgmt, Loging

~~To install spring 2.5 s/w~~

→ <Spring 2.5_home>\dist\spring.jar represents the cobole spring api and its dependent jar file is <Spring 2.5_home>\lib\jakarrha-commons\commons-logging.jar

→ The classes and interfaces of spring.jar uses the classes and interfaces of Commons-logging ja

→ <Spring 2.5_home>\changelog.txt use details about various versions of Spring s/w.

11/02/2013

Spring defination :-

→ Spring is a Javabased open source, light-weight, loosely coupled, aspect oriented and dependency injection based application \ Java-JEE framework to develop all kinds of applications.

→ Opensource s/w means It is not only free s/w but its source code will be exposed to programmers along with installation.

<Spring home>\src folder represents source code of Spring s/w.

→ Spring is light-weight s/w and Spring Applications are light-weight appns. reasons,

① Spring Containers are light-weight Containers. Because they can be activated any where by just creating objects for certain pre-defined classes.

NOTE:- Servlet Container, Jsp Container are heavy weight Container.

② most of the Spring appns can be executed without the heavy weight application Server and webserver s/w'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 teightly-coupled Components.

Spring is loosely coupled s/w. reasons,

① while working with Spring we can use either individual modules or multiple modules to gather.

② we can integrate Spring with Other Java Technologies in application

development. like, servlets, JSP, Structs, Hibernate and etc...

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

→ It is always recomended to develop ~~spring~~ large scale applications by enabling middlewar services like security, Transaction mgmt and etc...

Spring provides a new methodology called Spring AOP to apply these middlewcare services on spring applications.

This makes the Spring as Aspect oriented.

→ What is the difference blw 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

Eg①:- if student (resource) gets course metrical (like dependent value) only after passing request for it. It is called then Dependency lookup.

Eg②:- The way we collect Data source object reference from JNDI registry is called Dependency lookup.

Advantage:- resource can gather only required dependent values.

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

→ Dependency Injection:- The underlaying server (or) Container (or) F/w (or) Runtime (or) special resource assigns values to the resource automatically and dynamically.

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

Eg①:- If course meterial is assigned to Student the moment Student registers for a course is called Dependency Injection.

Eg②:- The way ActionServlet writes formdata to FormBeanclass properties

Eg③:- The way JVM calls constructor automatically to assign ~~formdata~~ Initial values to object.

Eg④:- The way ServletContainer 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 underlaying environment like Server/Framework may inject both necessary & unnessary values.

→ Spring supports both dependency lookup & dependency Injection. Spring Containers are design to perform dependency Injection on the resources of Spring application.

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

All major java projects are based on MVC2/MVC Architecture

M ——→ Model Layer ——→ (B.logic + persistence logic (connecting with DB)) (like Accounts Officer)

V ——→ View Layer ——→ presentation logic (like Beautition)

C ——→ Controler Layer ——→ Integration Logic (connectivity Logic)

→ The main logic of the application is called Business Logic

→ persistence logic is useful perform CURD operations on DataBase tbl's

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

→ presentation Logic is gives userinterface for enduser. to supply to inputs to application and to view results of the application

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

12/02/2013

JSP → Servlet → Javaclass / Java Bean → DB s/w

(view) (controller) (Model)

JSP → Servlet → RMI App → DB s/w

(view) (controller) (Model)

(Model)

JSP → Servlet → EJB Session Bean Comp → EJB Entity Bean comp → DB s/w

(V) (C) (B.logic) (persistence logic)

(Model)

JSP → Servlet → EJB Session Bean Comp → DAO class → DB s/w

(V) (C) (B.logic) (JDBC) (persistence logic)

"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)"

Model

Struts App → EJB Session Bean Comp → DAO class (jdbc) → DB s/w

(V) (C) (B.logic) (persistence logic)

③ Struts App → EJB Session Bean Comp → DAO (Hibernate) → DB s/w

(V) (C) (B.logic) (persistence logic)

Model

① Struts App → Spring JEE App → DAO (Hibernate) → DB s/w

(V) (C) (B.logic) (persistence logic)

Model

② Struts App → webservices comp → DAO (Hibernate) → DB s/w

(V) (C) (B.logic) (persistence logic)

Model

④ Spring WebMVC → Spring JEE App → Spring ORM / DAO App → DB s/w

(V) (C) (B.logic) (persistence logic)

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 view Layer, controller layer :-

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

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

EJB Session Bean, Webservices, Javaclass / JavaBean class

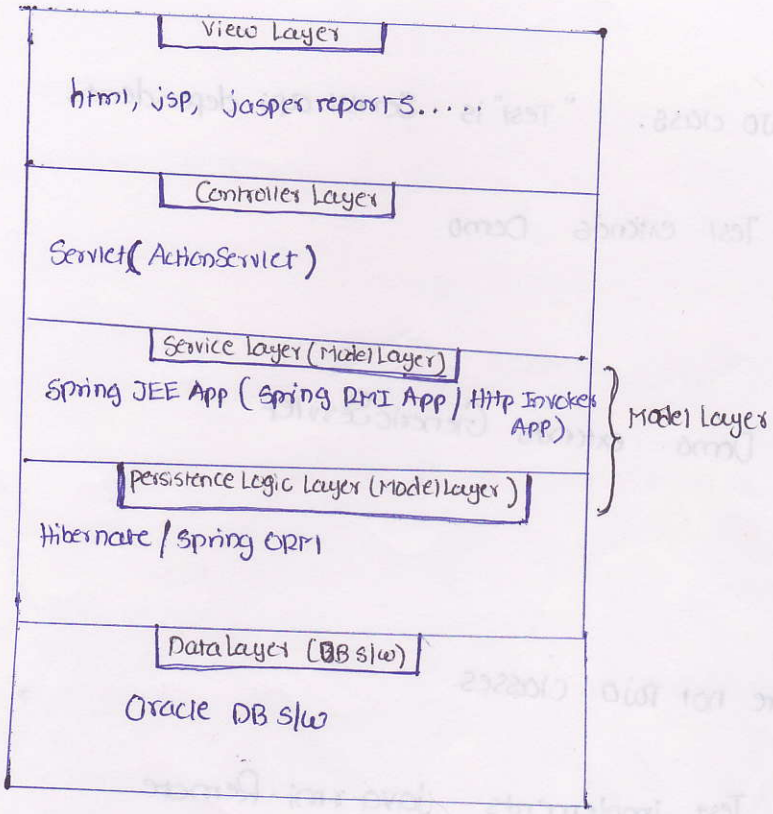
RMI, HttpInvoker (springs own Distributed Technology), CORBA,

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 2005) are supporting POJO/POJI model programing 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 part of the corejava. 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 programing due to this Spring appns are light weight appns

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 programing.

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 supplid 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 peace of code is called Unit Testing.

⑧ provides Built-in plugins to make Spring applications integratable with other Java technology based appns.

⑨ Allows to develop all kinds of applications like standalone, two-tire, three-tire, 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 appns.

⑫ provides abstraction Layer on other frameworks like Hibernate, OJB and etc..

⑬ supports Integration with Java/J2EE technologies and Java based Frameworks.

⑭ Easy to learn and use.

Module - I

Spring Core Module

→ Base module for other module 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 Concreate 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 container and lightweight containers

① BeanFactory Container (Part of Core module)

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

→ Container (like acquirium) is a Software Appn (or) Java class that can take care of the complete life (Birth to death) cycle of given resource. (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 Objects for java class that implements org.springframework.beans.factory.BeanFactory(I)

```
XmlBeanFactory factory = new XmlBeanFactory("<obj pointing to spring cfg file>");
```

(XmlBeanFactory: class implementing BeanFactory; factory: represents BeanFactory container)

→ To activate ApplicationContext Container in our Apps create obj for java class that implements Org.SpringFramework.Context.ApplicationContext(I)

Eg:- `SystemXmlApplicationContext ctx = new FileSystemXmlApplicationContext("<spring cfg file>");`

- `SystemXmlApplicationContext` → class implementing ApplicationContext interface
- `ctx` → ApplicationContext container

→ we can't Activate ServletContainer, 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 (Java Interface)
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 POJI

→ it is optional resource. But recommended to take

→ contains declaration of B.methods/utility methods

→ acts as a common understanding document between client application & SpringBean

→ it like course brocchet having declaration of topics [which acts as agreement document b/w faculty and student].

Spring Bean

→ A concreate 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 concreate Java class / JavaBean.

→ Contains Business methods implementation having Business Logic.

NOTE:- All Spring Beans must be configured in SpringConfiguration file. either by using xml entries (or) annotation

→ Specifying the details of resource through xml entries (or) Annotations to make underlaying 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 DependencyInjection / 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 SpringBean 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 bm1()
    {
      .....
      .... //use msg here
    }
}
```

Demo.xml (Spring cfg file)

(*1) msg property configuration for setter Injection

```
<beans>
```

```
<bean id="tb" class="TestBean" >
        <property name="msg" value="hello" />
</bean>
</beans>
```

(*1)

id/objname → "tb"
Bean class → "TestBean"
Bean property → "msg"
value to be injected → "hello"

} 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, JdbcStmt obj, RmiBusiness 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 bm1()
 {
   ....
   .... 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 SpringBean class object and to assign value "hai" to "msg" property.

When SpringContainer is activated by giving the above Demo.xml spring cfg file the SpringContainer 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 Beanproperties 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 Beanproperties and its dependent values.

Naming Conventions

X ⟶ Spring interface name

XBean ⟶ SpringBean class name

XCfg.xml ⟶ Spring cfg file name

XClient ⟶ Client App name

Eg:- Test ⟶ Spring interface name

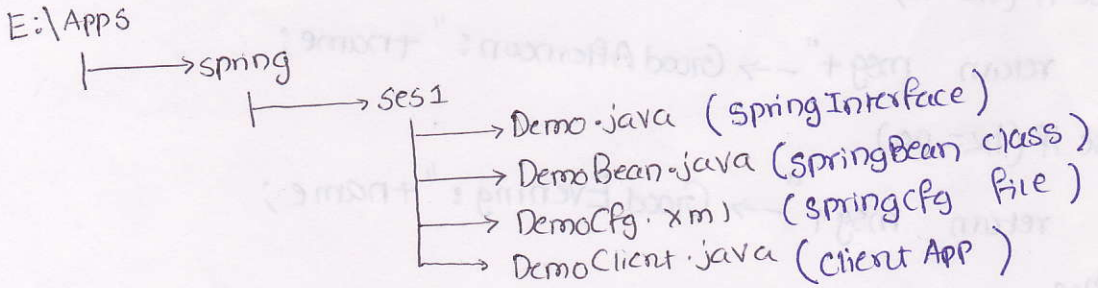
TestBean ⟶ SpringBean class name

TestCfg.xml ⟶ Spring cfg file name

TestClient ⟶ Client App name

1st Example Application of SpringCore module to demonstrate SetterInjection by using BeanFactory Container.

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




```
//Demo.java (POJI) (SpringInterface)
public interface Demo
{
  //B.method Declaration
  public String generateWishMsg(String name);
}
//DemoBean.java (POJO class) (SpringBean)
  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;
}
```
(g)

```
//implement B-methods
public String generateWishMsg(String name)
{
  //get current date and time
  Calender cl=Calender.getInstance();
  //get current hour of the day
  int h=cl.get(Calender.HOUR_OF_DAY);
  //generate wish msg
  if(h<=12)
    return msg+"-->Good moring:"+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
```
(J)

```
<!-- DemoCfg.xml -->
<!DOCTYPE ---------->
```
--> indicates that we are developing this xml document based on DTD rules given in spring-beans-2.0.DTD file.
collect this content from <spring-home>\dist\resources\spring-beans-2.0.DTD file.

```
<beans>
  <bean id="db" class="DemoBean">
    <property name="msg" value="hello"/>
  </bean>
</beans>
```
--> (e)

--> (f)

```
DemoBean db=new DemoBean();
db.setMsg("hello");
```

```
// 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) ---> (b)
    {
        // Locate Spring configuration file
        FileSystemResource res = new FileSystemResource("DemoCfg.xml");
        // Activate BeanFactory container
        XmlBeanFactory factory = new XmlBeanFactory(res);
        // get SpringBean class obj from SpringContainer
        Object obj = factory.getBean("db"); ---> (d)
        // type casting
        Demo bobj = (Demo) obj; ---> (h)
        // call B.method
        String result = bobj.generateWishMsg("raja");
                (k)              (I)
        System.out.println("Result is" + result);
                                          (k)
    }
}
```

(c)

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

<Spring_home>\dist\spring.jar (Main jar file)

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

Step-III :- Compile all the java resources 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 XmlParser
↳ SimpleApi 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 refering that Object using java.lang.Object class ref "obj"

NOTE:- getBean() method is the predefined method in predifined 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 impliment 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 bobj = (Demo) obj; is placed to satisfy above statement in our Spring appns.

→ Spring Container performs DependencyInjection only on those SpringBean objects that are created by the Spring container itself.

That means SpringContainer can't perform DependencyInjection on SpringBean class objects that are created by programmer manually.

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

→ Spring application Flow start with main ends with main. controle goes to one resource to another resource.

In the client appn we donot take SpringBean class manually as Typecast. The interface of implemention class Demo will be taken.

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

16/02/2013

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:- SpringContainer 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, the values injected by ConstructorInjection will be overridden because of setterInjection.

→ 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 BeanCollabration.

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

E:\APPS
→ Spring
→ ses2
→ Demo.java
DemoBean.java
TestBean.java
Democfg.xml
DemoClient.java

```
//Demo.java (POJI)
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
user defined class ← TestBean tb;
predefined class ← Date d;   } reference type properties to hold objects

        //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;
 }
}//class
```

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 (DependentBean to DemoBean)
public class TestBean
 { //Bean property
   String msg;
   //Write setter method of SetterInjection
   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 Collaborater 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" />
    <property name="month" value="10" />
    <property name="date" value="29" />
  </bean>
  <bean id="t1" class="TestBean">
    <property name="msg" value="hello" />
  </bean>
  <bean id="db" class="DemoBean">
    <property name="age" value="30" />
    <property name="tb" ref="t1" />
    <property name="d" ref="dt" /> <!-- or <property name="d" >
                                       <ref bean="dt" />
                                       </property> -->
  </bean>
</beans>
```

→ Takes it as 1900 + 113

→ Takes it as 0 to 11

Eg:- Boy and girl / girl roommates

// 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);
}
}
```

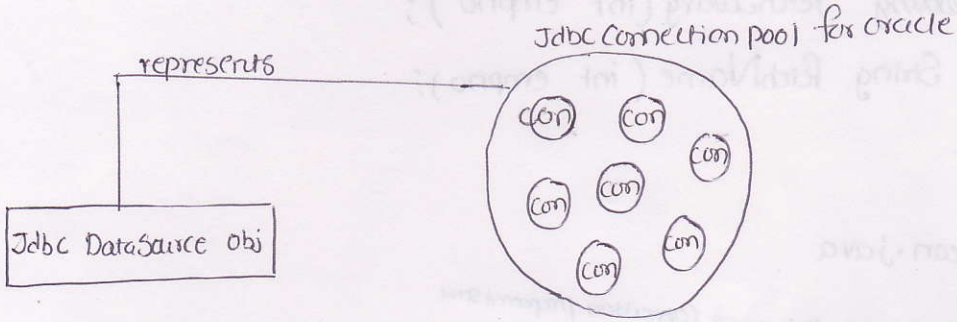
18/02/2013

Jdbc Connection Pool :-

Jdbc ConnectionPool is factory that contains set of readyly available Jdbc Connection Objects. All the Connection objects in the Connection pool represents Connectivity with same DataBase s/w.

→ 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 javax.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 coith oracle thin driver details then jdbc Connection pool for oracle coill 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 (POJI)
public Interface DBOperation
{
   public int fetchSalary(int empno);
   public String fetchName(int empno);
}

//DBOperationBean.java
import java.sql.*;  → Connection prepared Stmt
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;
   }
   //implement B. methods
   public String fetchName(int empno)
   {
      try
      {
         //get one 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 rs = 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 conpool
      Connection con = ds.getConnection();
          //get emp salary
  PreparedStatement ps = con.prepareStatement("select sal from emp cohere
                                                empno=?");
        ps.setInt(1, empno);
          //execute the query
  ResultSet rs = ps.executeQuery();
          // process the Results
        if(rs.next())
          sal = rs.getInt(1);
        }//try
```

```
        catch(Exception e)
        {
        e.printStackTrace();
        }
        return scal;
    }//method
}//class
```

→ <!-- Spring cfg.xml -->

```
<!DOCTYPE ..... >
<beans>
  <bean id="drds" class="org.sf.jdbc.datasource.DriverManagerDataSource" >
    <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" > (e)
    <property name="ds" ref="drds" /> (g)
  </bean>
</beans>
```

(f) gives datasource obj

↳ gives DataSource obj representing the spring container managed builtin connection pool

(g) → Injects DataSource obj to accr BeanProperty ds.

→ //DBOperation Client.java

```
import org.sf.beans.factory.xml.*;
import org.sf.core.io.*; ——→ FileSystem Resource class

public class DBOperationClient
{
    p s v m(String args[]) throws Exceptions (b)
    {
        //Locate Spring Configuration file
    FileSystemResource res = new FileSystemResource("springcfg.xml"); } (c)
        //Activate BeanFactory Container
```

```
XmlBeanFactory factory = new XmlBeanFactory(res);
// get SpringBean obj from container
DBOperation bobj = (DBOperation) factory.getBean("dbb"); (d)
   (i)
// call B.methods
System.out.println("Emp salary is" +bobj.fetchSalary(7499));
                                        (j)
S.O.P("Emp name is" +bobj.fetchName(7499));
                                    (L)
}//main
}//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 SpringContainer is injecting DataSource object to our SpringBean class property "ds".

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

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

*) <property name="passcoord" value="tiger" /> can also be written as

<property name="passcoord" > <value> tiger </value> </property>

→ we can locate SpringConfiguration file for BeanFactory container by using the following classes.

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

→ locates the SpringConfiguration 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.sf.core.io pkg)

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

19/02/2013

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 ClientApp:-

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

(or)

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

↳ absolute path

(or)

```
FileSystemResource res = new FileSystemResource("..\\..\\temp\\SpringCfg.xml");
```

↳ relative path

".." → Spring folder (child folder); ".." → Apps folder (Parent folder)

Example on Utilizing CLASSPATH resource :-

Step-I :- keep spring application ready

Step-II :- get the location on SpringConfiguration file and add the location to the CLASSPATH env... Variable.

E:\
⟶ Apps
⟶ temp
⟶ SpringCfg.xml

myComputer ⟶ properties ⟶ advanced tab ⟶ env Variables ⟶ user/system Variables ⟶ Variable name: CLASSPATH

Value: E:\Apps\temp; <existingValues>;.

⟶ OK ⟶ OK ⟶ OK

Step-III :- use CLASSPATH resource in client application

in ClientApp

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

Example ② on CLASSPATH resource :-

Step-I :- keep spring application ready

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

E:\
⟶ Apps
⟶ temp
⟶ SpringCfg.xml

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

myComputer ⟶ properties ⟶ advanced tab ⟶ env variable ⟶ user/system variables ⟶ Variable name : CLASSPATH

Value : E:\Apps\temp\test.jar ; <existing values> ; . ⟶ OK ⟶ OK ⟶ O

Step-III :- use ClassPathResource in client appn

in Clientappn

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

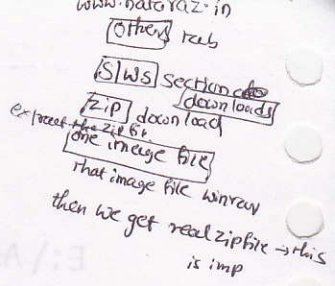
While working with SpringEnvironment always make sure that Current directory (.) is added to ClasspathEnv... Variable.

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

property type	tag/attribute to use
Simple data type	<value> or "value" attribute
java.lang.String	<value> or "value" attribute
array	<list>
java.util.List	<list>
java.util.Set (like HashSet, TreeSet ...etc)	<set>
java.util.Map	<map>
Bean type	<ref> or "ref" attribute
java.util.Properties	<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 slw 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 jdk1.6+)

To doconload slw: www.myeclipseide.com (30 days trail)

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.nataraz.in blog)

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

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

and get licency name & licence key

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

MyEclipse menu ⟶ Subscription Information ⟶ Scrscriber: [ ]

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 ⟶ ☐ Aop Builder ⟶ file: [DemoCfg-xml] ⟶ finish

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

right click on src folder ⟶ New ⟶ Interface ⟶ Filename: 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;
```

Ctrl+shift+O package import stmts.

Select all Beapn 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.

22-02-2013

```
<!-- DemoCfg.xml -->
<beans ---------- spring-bean-2.5.xsd >
  <bean id="dt" class="java.util.Date" />
  <bean id="db" class="DemoBean" >
    <property name="name" value="raja" />
    <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>
      </list>
    </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
        </entry>
```

represents one element

```
<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>
        <prop key="AP"> hyd </prop>
        <prop key="MH"> Mumbai </prop>
        <prop key="KR"> Banglore </prop>
    </props>
</property>
</bean>
</beans>
```

represents one element ← `<prop key="AP"> hyd </prop>` (key = "AP", value = hyd)

→ 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

rightclick Src → new → class → name: DemoClient

```
//DemoClient.java
public class DemoClient
{
    P S V m (String[] args)
    {
//Activate BeanFactory Container
XmlBeanFactory factory = new XmlBeanFactory(new ClassPathResource("SpringCfg.xml"));
//get SpringBean obj
Demo bobj = (Demo) factory.getBean("db");
//call B.methods
System.out.println(bobj.sayHello());
    }
}
```

Sysout + ctrl + space.

Step-VIII:- 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.

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

```
<!-- SpringCfg.xml -->
<!DOCTYPE ..... >
<beans>
  <bean id="ds" class="org.sf.jdbc.datasource.DriverMangerDataSource">
    <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>
```

FileSystemResource(e||src||)

→ To pass command line argument values to MyEclipse IDE appn

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

⟶ Arguments tab ⟶ Program arguments

db

val1 val2 val3 ..... valn

args[0] args[1] args[2] args[n-1]

Constructor Injection

→ If SpringContainer is using Parameterized Constructor to create Bean class object & to assign Dependent Values to Bean properties then it is called ConstructorInjection

→ Based on <constructor-arg> tag appearence count under <bean> tag the spring container picks up appropriate parameterized constructor to perform Constructor injection.

Example appn Demo.java → same as previous appn

DemoBean.java

```
public class DemoBean implements Demo
 {
    //DemoBean properties
     int age;
   String name;
    float salary;
//parameterized Constructors Supporting
                                  } place 3-Param constructor
SelectBean 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="raja" />
    <constructor-arg value="34563.69" />
</bean>
</beans>
```

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

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

SpringBean instantiation and for Constructor Injection,

// DemoClient.java

Same as previous appn

NOTE:-

While Working with Constructor 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 Parameters type in the Constructor 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="raja" />
 </bean>
</beans>
```

→ If multiple parameters of Constructor Contains Same type then we can resolve/identify them based on their 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="2" value="30" /> goes to "c"
    <constructor-arg index="1" 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 datatypes (or) indexes.

→ 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>
```

```
// Demo Client.java
imports
public class DemoClient {
  P S V m (String[] args) {
```

```
// Parent Bean Factory Container
XmlBeanFactory factory1 = new XmlBeanFactory(new ClassPathResource("DemoCfg1.xml");
// child BeanFactory Container
XmlBeanFactory factory2 = new XmlBeanFactory(new ClassPathResource("DemoCfg2.xml",
                                                                   factory1);
// get SpringBean class obj
Demo bobj = (Demo) factory2.getBean("db");
S.O.P("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 SpringBean class objects in the following scopes. ① Singleton
② Prototype
③ request
④ session

① Singleton:- Container creates only one object of SpringBean 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 SpringBean class for multiple factory.getBean() method calls having same Bean id.

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

</bean>

③ request:- Container keeps SpringBean class object as request attribute value in webapp

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

</bean>

④ session:- Container keeps SpringBean class object as session attribute value in webapp

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 SpringBean scope is Singleton our springBean class doesnot 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 doesnot change scopes of any Bean.

NOTE:- most recomended 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,

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

Application Context 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 implement org.sf.context.ApplicationContext (I).

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

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

① FileSystemXmlApplicationContext (org.sf.context.support)

→ Activates Spring Container (→ ApplicationContext) by locating Spring cfg file from the specified path of FileSystem.

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

② ClassPathXmlApplicationContext (org.sf.context.support)

→ Activates Spring Container by locating Spring cfg file from the values added to the Classpath env.... variable.
(Spring Container ↳ ApplicationContext container)

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

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

→ Activates ApplicationContext container in Webapplications

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 (that are possible 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 ApplicationContext container.

*) In Bean Environment multiple SpringBeans developed by multiple programmers will be configured in different Spring Configuration files. we can activate ApplicationContext 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 can SpringBeans are Configured in SpringConfigurationfile.hbco can be enable pre-instantiation (~~using 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

Q:- What happen when factory.getBean(-) is called?
↳ BeanFactory.

Ans:- Creates and returns SpringBean class object if SpringBean scope is "prototype"
Creates/locates and returns SpringBean class object if SpringBean scope is "Singleton"

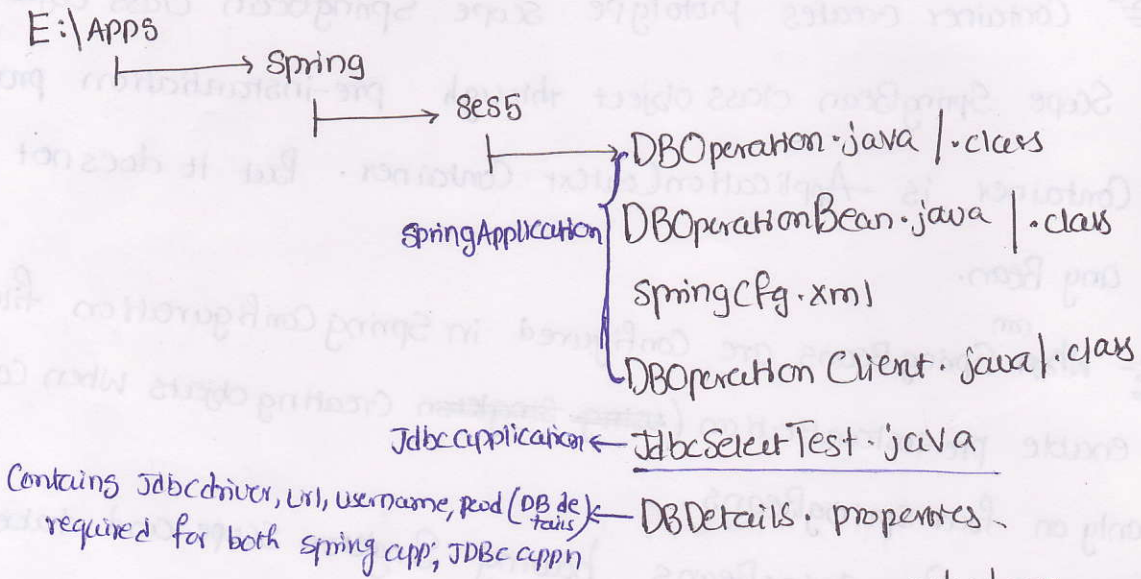
Q:- What happen when ctx.getBean(-) is called?
↳ ApplicationContext

Ans:- Creates and returns SpringBean class object if SpringBean scope is "prototype"
locates and return 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 s/w industry is don't hard code any values in ur application that are changable in the feature. It is recommended to pass these values to application from out side the application by taking the support of TextProperties file / xml file.

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



26/02/2013

→ To make SpringConfiguration file ~~access~~ getting certain values from properties file we place placeholders specifying the keys of the properties file (like ${<key>}) ← represents placeholder

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.sf.beans.factory.config.PropertyPlaceholderConfigurer as spring bean is SpringCfg file.

Source code of the above application :-

// DBDetails.properties

```
#jdbc driver and DB details
my.driver = oracle.jdbc.driver.OracleDriver
my.url = jdbc:oracle:thin:@localhost:1521:orcl
my.dbuser = scott
my.dbpwd = tiger
```

Spring application

DBOperation.java ⟶ Same as 18/02/2013 appln

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" />
  </bean>
  <bean id="drds" class="org.sf.jdbc.dataSource.DriverManagerDataSource" >
    <property name="driverClassName" value="${my.driver}" />
    <property name="url" value="${my.url}" />
    <property name="username" value="${my.dbuser}" />
    <property name="password" value="${my.dbpwd}" />
  </bean>
  <bean id="dob" class="DBOperationBean" >
    <property name="ds" ref="drds" />
  </bean>
</beans>
```

properties file in current directory.

placeholder

```
// DBOperationClient.java
import org.sf.context.support.*;
public class DBOperationClient
{
  p s v m(String[] args) throws Exception
  {
    //Activate Application Context Container
FileSystemXmlApplicationContext ctx = new FileSystemXmlApplicationContext("springcfg.xml");
```

```
//get SpringBean 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
```

/> javac *.java

/> java DBOperationClient

JDBC application :

```
//JDBCSelectTest.java
import java.sql.*; → query operations
import java.util.*;
import java.io.*; → stream
public class JdbcSelectTest
{
p s v m(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.url");
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())
    {
S.o.p (rs.getInt(1) + " " +rs.getString(2)+" "+rs.getString(3));
    }
    //close jdbc objs
      rs.close();
      st.close();
       con.close();
  } //main
} // class
```

/>javac JdbcSelectTest.java

→ BeanFactory Container cannot recognize and use placeholders and properties file even though the special Bean "propertyplaceholderConfigurer" is configured.

→ We can configure multiple properties file in SpringEnvironment. 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 <!-- springcfg.xml -->

Eg:-

```
<beans>
  <bean id="propconfig" class="org.sf.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
      ↳ Array type property.
      <list>
        <value> DBDetails1.properties </value>
        <value> DBDetails.properties </value>
      </list>
    </property>
  </bean>
  =
</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 its 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. are to get they 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 Unicodeeditor tool downloaded from "higopi.com"

27/02/2013

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

Step1 Download unicode editor tool from cocoa.higopi.com.

Step2 Launch "unicode editor tool" through index.html and type the following 4 hindi words निकालो, सुरक्षित, रुको, रहा

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

i18nConverter> nativeRascii -encoding unicode P_1.txt P_2.txt

↳ built in tool of jdk ↳ contains the generated Unicode numbers

You can make ApplicationContext container ready for make i18n by configuring the following SpringBean having properties file names.

Org.sf.context.support.ResourceBundleMessageSource.

Spring based i18n App Example appn

E:\Apps
- Spring
  - ses6
    - App.properties (base file)
    - App_fr_CA.properties
    - App_de_DE.properties
    - APP_hi_IN.properties
    - DemoCfg.xml
    - DemoClient.java

//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 (germcan)

```
Str1 = LÖSCHEN
Str2 = DENN
Str3 = ZUG
Str4 = ABSAGEN
```

//APP_hi_IN.properties (telugu) for Unicode representation for

```
Str1 = \u0928\u093f\u0915\u093e\u0932\u094a
Str2 = ....
Str3 = ....
Str4 = ....
```

} Copy from P_2.txt file

NOTE:- The Filename of BaseFile is must be there in Localespecific Filenames.

→ all properties file must contain same keys having different values

```
<!-- DemoCfg.xml -->

<beans>
  <bean id="messageSource" class="org.sf.context.support.ResourceBundle
                                  MessageSource">
    <property name="basenames">
    <list>
      <value> App </value>
      <value> App_de_DE </value>
      <value> App_fr_CA </value>
      <value> App_hi_IN </value>
    </list>
    </property>
  </bean>
</beans>
```

↳ fixed BeanId

```
//DemoClient.java

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

public class DemoClient {
  p s v m (String[] args) throws Exception {
    // Locale Object        language ↑   country ↑
  Locale l = new Locale(args[0], args[1]);
    // 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);
```

new Object{"hello"}

↳ This msg come when the properties file is not located.

↳ supplies for values

↳ locale object

↳ key in the property file.

```
// Develop swing Frame
JFrame jf = new JFrame();
Container cp = jf.getContentPane();
// create btns 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      (uses App_fr_FR.properties)
//> java DemoClient de DE
//> java DemoClient x y        (uses App.properties)
```

docs/reference/pdf/spring-reference.pdf IoC container Application context container.

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

Can you compare BeanFactory Container and ApplicationContext Container.

Feature	BeanFactory	ApplicationContext
BeanInstantiation / ~~Wiring~~ / ~~Autowiring~~ → dependency Injection	yes	yes
Ability to work with multiple spring cfg files with single Instance of container.	NO	Yes
pre-instantiation of spring Beans	NO	yes
Recognization of placeholders and properties files	NO	yes
i18N Support	NO	yes
ApplicationEvent handling	NO	yes
Automatic BeanPostProcessor registration	NO	Yes
Automatic BeanFactoryPostProcessor 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 ApplicationContext Container.

→ ActionPerformed on the component / object is called "Event". The process of executing logics when Event is raised is called "EventHandling". To perform this EventHandling we need EventListeners. When the ApplicationContext Container is started or stopped the ApplicationEvent will be raised. we can handle this ApplicationEvent we can use ApplicationListener. In this process we can notice the amount of time that the ApplicationContext Container is there in running mode.

Example Application:-

Step-I:- keep 1st appn ready. (ses1)

Step-II:- Develop seperate Listener class implementing org.sf.context.ApplicationListener Interface.

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

```
public class MyListener implements ApplicationListener
{
   long sttime, endtime;
   public void onApplicationEvent(ApplicationEvent ae)
   {
     S.o.p("onApplicationEvent of MyListener class");
     if(ae.toString().indexOf("ContextRefreshedEvent")!=-1) // When Container
                                                              is started
      {
        sttime = System.currentTimeMillis();
        S.o.p("The Container is started at" + new Date());
      }
   else if(ae.toString().indexOf("ContextClosedEvent")!=-1) // When Container
                                                              is stopped.
     {
     endtime = System.ccurentTimeMillis();
     S.o.p("The container is stoped at" + new Date());
     S.o.p("The ~~current~~ Container is in running mode for" + (endtime-sttime)
                                                    + "msecs");
     } //method
   } //onApplicationEvent(-)
} //class
```

Step-III :- Configure the above Listener class in Springcfg file.

```
<!-- DemoCfg.xml -->
<!DOCTYPE ----->
<beans>
  <bean id="mb" class="MyListener" />
  <bean id="db" class="DemoBean" />
```

Step-IV :- Activate ApplicationContext 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 execute this method right after Bean Instantiation and Dependency injection

→ This method is useful to check weather bean properties are assigned with valied values or not. It is also useful to check weather manditory 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 Object

→ This method is useful to nullify bean properties and release non-java resources assossicated with Spring beans (like closing jdbc con obj)

These two life cycle method must be specified in SpringBean during configure in Spring cfg file using init-method, destroy-method attributes of <bean> tag.

Rt proj → Build path → Add Libraries → MyEclipse Library

Run → search if we want to any information

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;

  setters & getters

  //user-defined init life cycle method
  public void myInit() throws Exception
  {
    S.O.P("DemoBean : myInit()");
    if(age<=0 || msg==null)
    throw new Exception("set proper values to Bean property");
  }
```

Validating Bean property value.

```
// user-defined destroy() life cycle method
    public void destroy()
    {
      S.o.p("DemoBean: myDestroy()");
      {age = 0;   } nullifying
      {msg = null;} Bean properties.
    }
// B.method implementation
  public String sayHello()
  {
  return "msg="+msg+"age="+age;
  }
}
```

Validating Bean Property Value

```
<!-- 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());
// close container
ctx.close();
  }
}
```

*) if container is ApplicationContext 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 over programmer may forget to configure these lifecycle methods.

TO Overcome this problem it is recomended to make SpringBean class implementing two special interfaces (InitializingBean, DisposibleBean). and we can use methods of these two interfaces as aultermate for user-defined lifecycle methods.

Org.sf.beans.factory.InitializingBean interface gives afterPropertiesSet() method
↳ can be used as aultermate to custom-init() method.

Org.sf.beans.factory.DisposableBean interface gives destroy() method
↳ can be used as aultermate to Custom-destroy() method.

→ While Working with these two methods there is no need of Configuring method names in SpringCfg 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 Customlifecycle 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, DisposibleBean
{
    //Bean properties
    private String msg;
    private int age;
```

setters & getters

```
public String sayHello() {
    return "msg="+msg+"age="+age;
}
public void destroy() throws Exception
{
    S.o.p("DemoBean: destroy()");
    age=0;
    msg=null;
}
public void afterPropertiesSet() throws Exception
{
    S.o.p("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 SpringBean class is not implementing InitialingBean, DisposibleBean 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 InitializingBean, DisposibleBean interfaces. so Configure Customlifecycle methods while working with those SpringBeans.

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 springBean class properties

XxxAware interfaces are

① org.sf.beans.factory.BeanFactoryAware
→ useful to inject BeanFactory container to our SpringBean class

② org.sf.context.ApplicationContextAware
→ useful to inject ApplicationContext container to our SpringBean class

③ org.sf.beans.factory.BeanNameAware
→ useful to inject beanid spring bean to Springbean class

While performing Interface injection our SpringBean class acts as non-POJO class.

Example Application performing Interface Injection :-

//Demo.java ~~Same as previous appn~~

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

For example appn on Interface Injection refer Supplimentary 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 BeanPostProcessor. for this the class must implement org.sf.beans.factory.config.BeanPostProcessor (I)

→ This Interface contains 2 methods

① public Object postProcessBeforeInitialization(Object bean, String bname)
- bean → represents SpringBean Object
- bname → represents Bean logical name
- → executes after all Dependency Injection
- → useful to modify Bean properties after dependency Injection

② public Object postProcessAfterInitialization(Object bean, String bname)
- bean → represents SpringBean obj
- bname → represent Bean Logical name
- → executes after custom-init method
- → useful to modify Bean properties after Compleet Initialization.

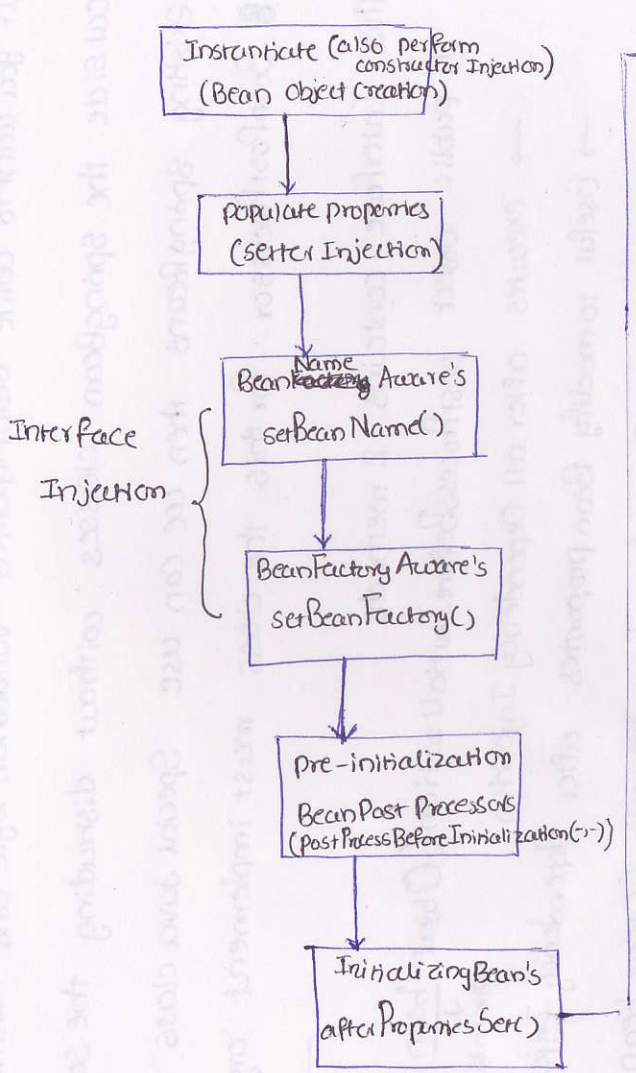
NOTE:- Custom init() method and afterPropertiesSet() related logics are specified to one Spring bean where as the Bean post processor (the above two methods related logics are common for multiple SpringBeans).

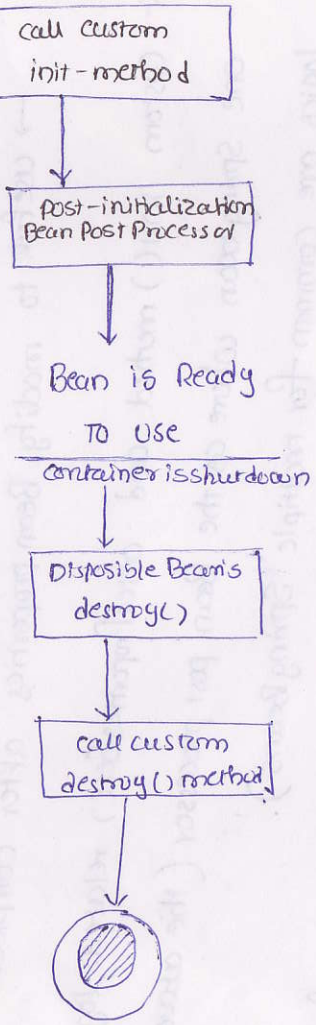
For Example application on BeanPostProcessor implementation refer the supplimentory handout given on 01/03/2013

→ ApplicationContext container automatically recognizes and registers BeanPostProcessor by seeing its Configuration in SpringConfiguration file.

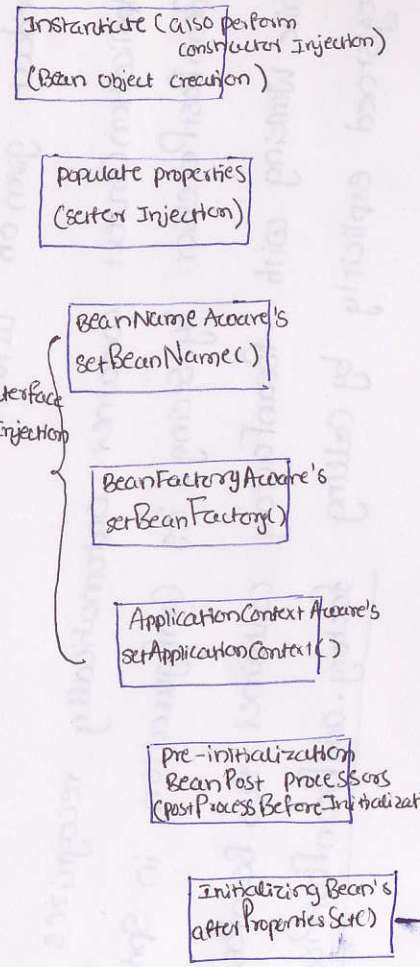
While Working with BeanFactory container this BeanPostProcessor must be registered explicitly by calling factory.addBeanPostProcess(new MyProcessor()); method.

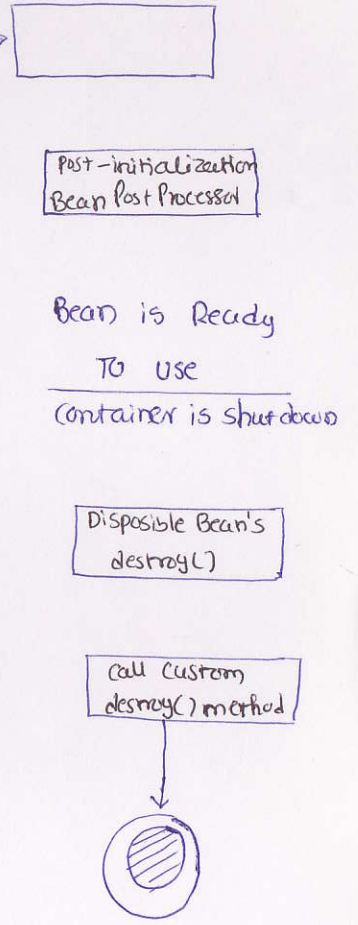
SpringBeanLife cycle Diagram W.r.to BeanFactory Container.





Bean life cycle Diagram w.r.to Application Context Container.





Spring App on Interface Injection and Bean Post Proccessor Impl

```
==========================================================
----------------------Demo.java----------------------
public interface Demo {
        public String sayHello();
}
-------------------DemoBean.java-------------------------
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.BeanFactoryAware;
import org.springframework.beans.factory.BeanNameAware;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;

public class DemoBean implements Demo,BeanNameAware,BeanFactoryAware,ApplicationContextAware
{
        //Special Bean properties
     private String bname;
     private BeanFactory factory;
     private ApplicationContext ctx;

        String msg;

        //setXxx(-) for setter Injection
         public void setMsg(String msg) {
               System.out.println("DemoBean:setMsg(-)");
                    this.msg = msg;
                }

     //method of ApplicationContextAware(I)
     public void setApplicationContext(ApplicationContext ctx)
                    throws BeansException {
               System.out.println("DemoBean:setApplicationContext(-)");
               this.ctx=ctx;
     }
     //method of BeanFactoryAware(I)
     public void setBeanFactory(BeanFactory factory) throws BeansException {
               System.out.println("DemoBean:setBeanFactory(-)");
               this.factory=factory;
     }

     //method of BeanNameAware(I)
     public void setBeanName(String name) {
               System.out.println("DemoBean:setBeanName(-)");
               this.bname=name;
     }

   public String sayHello() {

        System.out.println("current bean id"+bname);
        System.out.println("current bean is singleton?"+ctx.isSingleton(bname));
        System.out.println("current beab is prototype?"+ctx.isPrototype(bname));
        System.out.println("all beans are");
        String id[]=ctx.getBeanDefinitionNames();
        for(int i=0;i<id.length;++i)
        {
              System.out.println(id[i]+"....");
        }
        return "Good morning:msg="+msg;
        }//method

}//class
--------------------------------DemoCfg.xml--------------------------------
<?xml version="1.0" encoding="UTF-8"?>
<beans
        xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:p="http://www.springframework.org/schema/p"
```

Interfaces for Interface injection

methods supporting Interface Injection

```
xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-bean

   <bean id="mp" class="MyProcessor"/>

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

   <bean id="dt" class="java.util.Date"/>
  </beans>
  ----------DemoClient.java----------
  import org.springframework.context.support.FileSystemXmlApplicationContext;

public class DemoClient {

      public static void main(String[] args) {
          //Activate Spring container
      FileSystemXmlApplicationContext ctx=new FileSystemXmlApplicationContext(".\\src\\DemoCfg.xml");
          // get Spring Bean class obj
          Demo bobj=(Demo)ctx.getBean("db");
          // call B.method
          System.out.println(bobj.sayHello());
      }//method
}//class
----------MyProcessor.java----------
import java.util.Date;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;
public class MyProcessor implements BeanPostProcessor {
      @Override
      public Object postProcessBeforeInitialization(Object bean, String bname)   throws BeansException {
          System.out.println("MyProcessor: postProcessBeforeInitialization()");
          if(bean instanceof DemoBean )
          {
              DemoBean obj1=(DemoBean)bean;
              obj1.msg="hello1";
              return obj1;
          }
          if(bean instanceof java.util.Date)
          {
              Date obj2=(Date)bean;
             obj2.setYear(130);
             return obj2;
          }
          return bean;
      }
      public Object postProcessAfterInitialization(Object bean, String bname)     throws BeansException {
          System.out.println("MyProcessor: postProcessAfterInitialization()");
          return bean;
      }
}
```

Factory Bean :- 02/03/2013

→ If NormalBean is configured as DependentBean to Bean property then Normal Bean class Object will be injected directly to Bean Property.

→ FactoryBean is a bean that generates resultent object always. If FactoryBean is configured as DependentBean then FactoryBean class object will not be injected to property but the resultentBean 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 a FactoryBean then the "t1" property of DemoBean class will be injected with the resultent object given by TestBean class.

SpringBean becomes FactoryBean when it implements org.sf.beans.factory.FactoryBean (I)

This interface contains 3 methods

① getObject() → returns the resultent obj

② getObjectType() → returns the type of resultent obj

③ isSingleton() → returns weather resultent object is singleton or not

NetBeans

type: IDE s/w for java

vendor: SunMS (oracle corp)

version: 6.7.1

opensource

given GlassFish 2.x as built-in server

to download s/w: www.netbeans.org

for docs: www.netbeans.org

Resources of the application

1) Demo.java (spring Interface)
2) DemoBean.java (spring Bean class)
3) Democfg.xml (spring cfg file)
4) DemoClient.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

FileMenu → New → Project → java → javaapplication → next → projectName SpringPrj

→ ☐ create Main class → Finish

Step-II :- Add spring Libraries to the project

Right Click on Libraries folder of project → add Library → SpringFramework 2.5

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

Rightclick on Source pkg → New → JavaInterface → ClassName : Demo

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

Step-IV :- Rightclick on Source pkg → New → Javaclass → className : DemoBean

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

Rightclick → insert code → setters → ok

```
// Select Setter method supporting setter injection
public void setObj(Object obj)
{ this.obj=obj;
}
public String sayHello() {
return "Good morning -->obj=" + obj.toString();
} }
```

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

Step-V :- R.C. on Source Package → New → Javaclass → className : TestBean

```
//TestBean.java
Public class TestBean implements FactoryBean
{
public Object getObject() throws Exception {
return new java.util.Date(); → returns Dateclass obj as the Resultent obj
}
```

```
public Class getObjectType() {
    return Date.class;  → specifies the type of the Resultent Object
}

public boolean isSingleton() {
    return true;  → specifies the resultent object scope is Singleton
}
}
```

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

```
//DemoClient.java
public class DemoClient
{
    psvm(String[] args)
    {
        //Activate Spring Container
ClassPath FileSystem Xml ApplicationContext ctx = new ClassPath FileSystem Xml ApplicationContext("Democfg.xml");

        //get Spring Bean objects
Demo bobj = (Demo) ctx.getBean("db");   (args[0] → bean id of DemoBean)
Called to know the Resultent Obj of Factory Bean:
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 class → properties → Run → main class: DemoClient
Argument: db tb → Ok
Rightclick on springpros

Step-VII :- Add Configuration file to the project

R.C on Source pkgs → New → Other → XML → XML Document → FileName: Democfg
→ next → ⊙ DTD-constraint Document → next →
DTD public id : -//SPRING DTD//Bean 2.0//EN
DTD System id : http://www.sf.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" />   → since TestBean is FactoryBean generating Date class object as ResultentObject. so, the obj will be injected with Date class object
    </bean>
</beans>
```

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.sf.jndi.~~FRD~~ JndiObjectFactoryBean.

TimerFactoryBean

RMI registryFactoryBean and etc...

To set Command line arguments to the application of netBean IDE

RightClick on project ⟶ properties ⟶ Run ⟶ Mainclass name: DemoClient

arguments : db tb

val1 val2 val3 ... valn

↓ ↓ ↓ ↓

args[0] args[1] args[2] args[n]

Auto Wiring :-

→ performing Dependendency 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 wiring 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 Auto wiring we need to use "autowire" attribute of <bean> tag.

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

04/03/2012

~~Date~~ We can make spring container performing autowiring in 4 modes.

① byName (autowire = "byName")

→ performs setter Injection based on bean properties and dependent beans 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~~ byType

→ If Bean class contains 0-param (compiler generated (default constructor)) constructor then it performs byType mode ~~setter~~ injection

Otherwise it performs constructor 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                                  Sout + 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 "goodmorning TestBean.msg =" + tb;
} }
```

// TestBean.java

```
public class TestBean
{
   //Bean property
     String msg;
   //setter method for setterInjection
    public void setMsg(String msg)
    {
      this.msg = msg;
    }
   //implement toString() method
     public String toString()
     {
       return msg;
     }
}
```

// Democfg-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 Bean property (db) and TestBean class Beanid (tb) must match

```
//DemoClient.java
public class DemoClient {
    psvm(String[] args)
    {
    //Activate Spring Container
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("DemoCfg.xml");
    //get Spring Bean obj
    Demo bobj = (Demo) ctx.getBean("db");
    //call b.method
    S.O.P(bobj.sayHello());
    }
}
```

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

Autowire = "byType" Coding Spring cfg file :-

```
//in DemoCfg.xml
<!Doctype ....>
<beans>
  <bean id="t1" class="TestBean">
    <property name="msg" value="hello" />
  </bean>
  <bean id="db" class="DemoBean" autowire="byType" />
</beans>
```

↳ here the Beanproperty "tb" type TestBean and the DependentBean obj "t1" type (TestBean) must match

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 Spring cfg file :-

```
//in DemoCfg.xml
<!DOCtype ....>
```

<Spring-home>\resources\spring-injects.cfg.xml

```
<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-params Constructor having TestBean has the param type.

autowire="autodetect" coding SpringConfiguration file :-

```
<beans>
    <bean id="t1" class="TestBean">
        <property name="msg" value="hello" />
    </bean>
    <bean id="db" class="DemoBean" autowire="autodetect" />
</beans>
```

(*1)

(*1) If DemoBean does not contain any explicit Constructor then SpringContainer 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 autowiring performs setter injection and explicit wiring performs Constructor injection or vise versa then the wiring that perform Setter Injection will be effected.

Both Containers (BeanFactoryContainer & ApplicationContextContainer) supports autowiring.

→ While using Spring with other Java technologies it is recomended to place the logic of activating Spring Container and getting SpringBean class object in one time execution block called Constructor (or) static block

→ ~~init() method~~ It is recomended to call Business method (or) SpringBean class object from respectively executing block.

Client App	Place to activate Spring Container and to get SpringBean 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 (servlet prgs) to recognize spring api. jar files added WEB-INF\lib folder will be used by servlet container/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

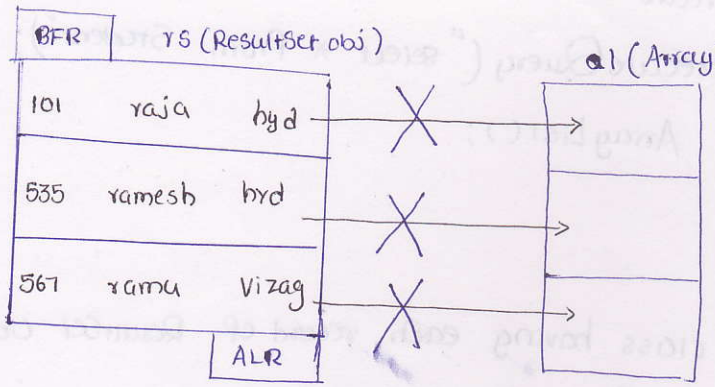
→ If you are looking to perform only Read operation then prefer working with Non-Synchronized Data structures for Better performance. (Eg:- ArrayList, HashMap and etc...)

→ 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....)

→ Eventhough network is not there it is not recomended to send ResultSet object from DestinationLayer to SourceLayer directly. It is recomended to keep the records of ResultSet in CollectionFramework Datastructure and send that DataStructure to Destination Layer.

Understanding problem to copy the records of ResultSet obj of ArrayList DataStructure



Each element of ArrayList allows only one object at a time. Each record of ResultSet contains multiple values including multible 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 Javaclass object and add that object to each element of ArrayList as shocon 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

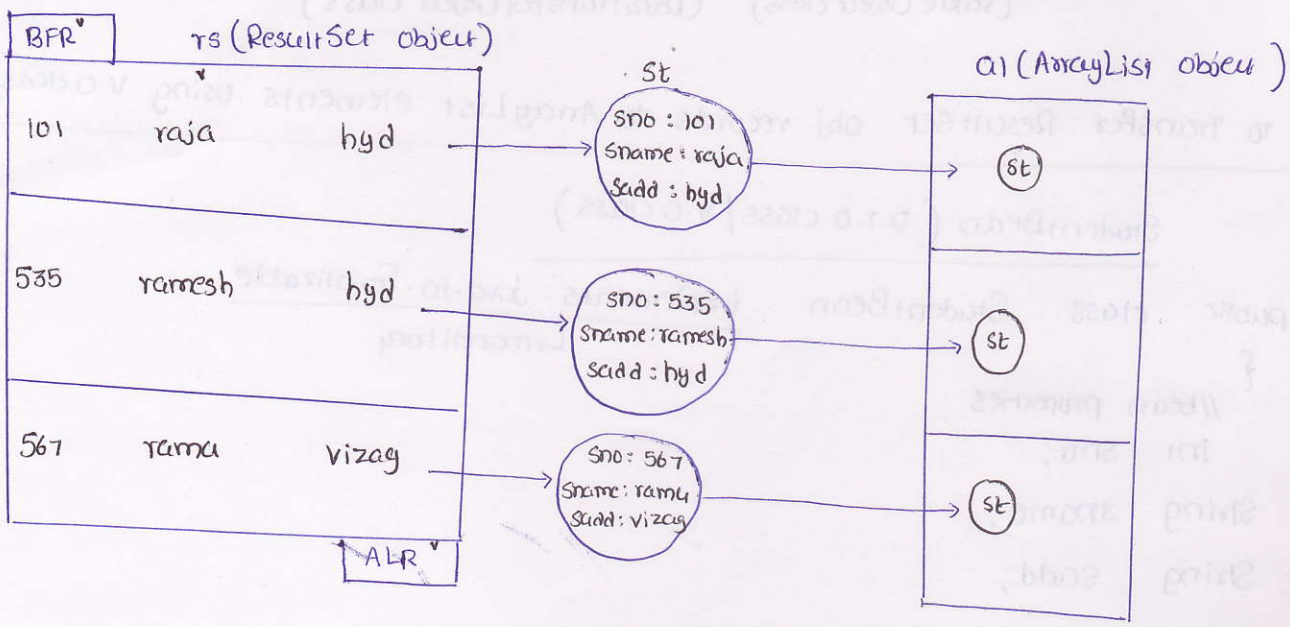
StudentBean (D.T.O class/V.O class)

```
public class StudentBean implements java.io.Serializable
                                    ↳ mandatory
{
  //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 inorder 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.setSadd(rs.getString(3));
// add each D.T.O class / V.O class obj to the elements of ArrayList
        al.add(st);
 } // while
```



→ Since, 'st' object are representing single big value by combining multiple individual values (multiple simple values & multiple objs). So, it is called value object and its class is called V.O. class.

→ Each record of ResultSet Data is transfering from ResultSet to ArrayList in b/w 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 CollectionFlw 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 we need Map DataStructures

(4) To maintain Mail properties java.util.Properties class

(5) In SessionTracking to maintain huge amount of Data as SingleSession attribute value.

(6) While developing gaming appns to maintain huge amount of Temporary 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 CollectionFramework DataStructure

(2) AS ModelLayer Resource while Developing MVC1 & MVC2 architecture based projects

(3) AS springBean in spring Appns

AS Hibernet persistent class in Hibernate appns

(4) AS Helperclass in SessionTracking. (While maintaining huge amount of Data as Single session attribute value).

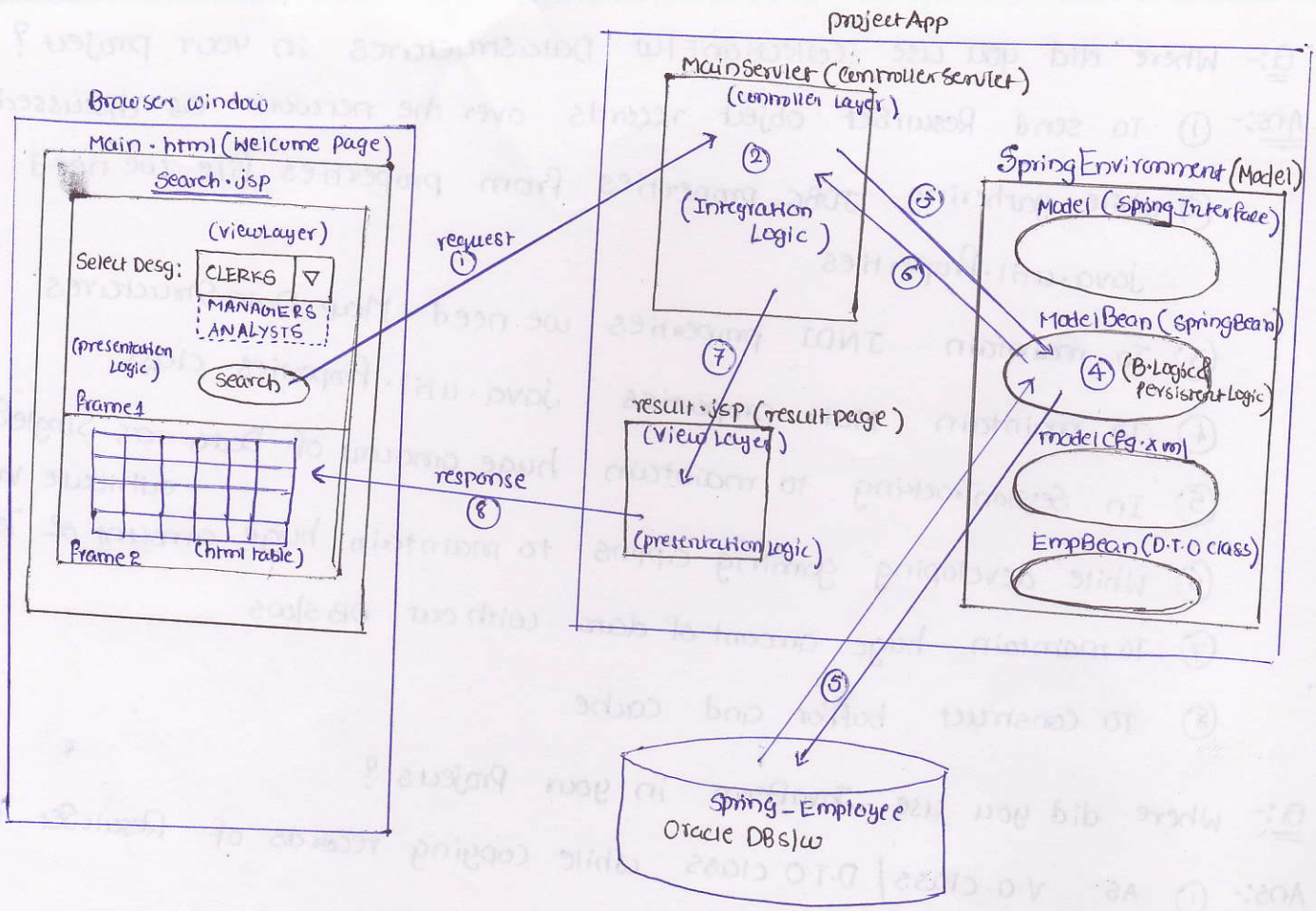
MiniProject info :-

→ It is based on MVC2 architecture

→ Spring Core module application acts as ModelLayer appn

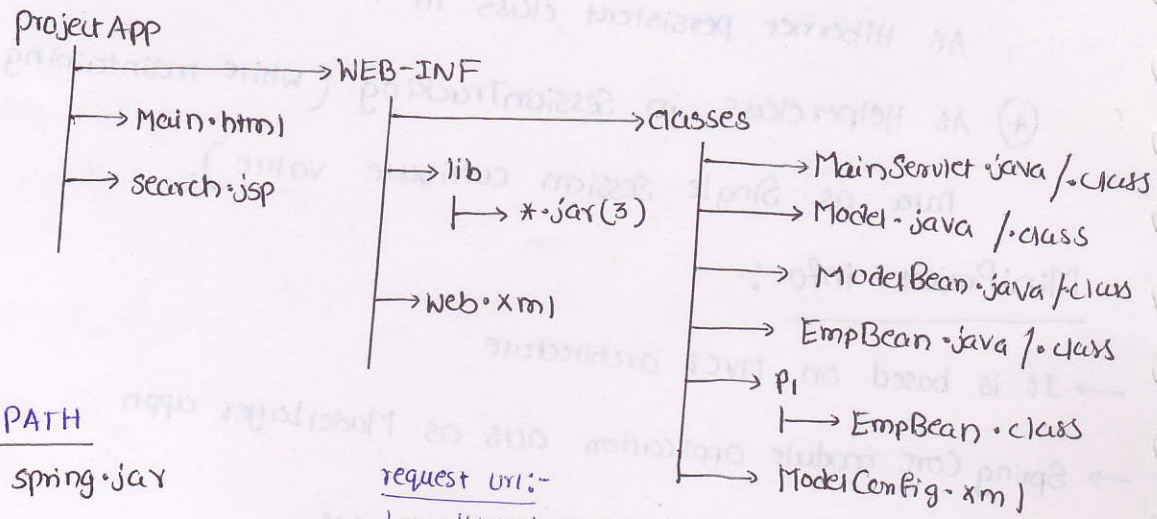
→ Jsp programs acts as viewLayer resources

→ Servlet program acts as ControllerLayer 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.



jar files in CLASSPATH

Servlet-api.jar, spring.jar

jar files in WEB-INF/lib folder

Spring.jar, commons-logging.jar, ojdbc14.jar

request url:-

http://localhost:2020/ProjectApp/Main.html

→ To use Java class in Jsp program it must be placed in the package created in the same folder of jsp (or) in the userdefined 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 statement and they are alternate for the xml file based MetaData, Resource configuration operations.

Syntax :- @<Annotation_name>(param1=val1, param2=val2, ...)

- <Annotation_name> → It is like xml tag name
- (param1=val1, param2=val2, ...) → Like xml tag attributes

two types of annotations

① Annotations for Documentation
- → useful while generating java api documentation
  - @Params @since @version @author @throws and etc...
- → introduced from jdk1.1

② Annotations for programming
- → useful for metadata, resource cfg operations
  - @Override @AutoWired @Resource @Component and etc...
- → introduced from jdk1.5

MetaData means data about data.

Resource cfg is also some sort of meta data operation.

→ XML files give good flexibility of modification but Bad performance. bcz XML parsers are heavy weight slws to process xml documents.

→ Annotations give Bad flexibility of modification but good performance.

→ All Java Technologies that are designed based on Jdk1.5+ are supporting Annotations based programing.

Eg:- Spring 2.5+, Hibernate 3.2+, Struts 2.x, EJB3.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)

07/03/2013

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

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 recomended 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 togather to perform Dependency Injections.

→ For Spring 2.5 based AutoWiring refer appn ① of the Supplimentary handout given on 07/03/2013

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

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

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

Spring 2.5 Applicaiton on Annotations (IOC) (Application 1)

```
==========================================
----------Test.java----------
// Test.java (Spring Interface)
public interface Test
{
      public String sayHello();
}
----------TestBean.java---------
//TestBean.java
import org.springframework.stereotype.*; --> for @Service annotation
import org.springframework.beans.factory.annotation.*; --> for @Autowired annotation
import java.util.*; --> for Date

@Service // we can also place @Component (makes the Java class as SpringBean)
public class TestBean implements  Test
{
  // Bean property
     Date d; --> reference type Bean property.

   @Autowired  //to perform auto wiring(by type)  on bean property
     public void setD(Date d)
     {
          this.d=d;
     }

     public String sayHello()
     {
          return "good morning  "+d.toString();
     }
}
----------DemoCfg.xml----------
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xmlns:context="http://www.springframework.org/schema/context"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd">

          <context:annotation-config /> --> makes the springContainer to recognize and use annotations
          <!--<context:component-scan base-package="." /> -->

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

     </beans>
----------TestClient.java----------
// TestClient.java

import org.springframework.context.support.*;

public class  TestClient
{
  public static void main(String[] args)
  {
FileSystemXmlApplicationContext ctx=new FileSystemXmlApplicationContext("DemoCfg.xml");
     Test beanobj=(Test)ctx.getBean("tb");
     System.out.println("result is "+beanobj.sayHello());

  }//main
}//class
==========================================
Spring 3.x Application on Annotations(IOC)
----------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 SpringBean

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

#{.....} → spring Expression Language


```
package p1;

public interface TestInter
{
    public String sayHello();
}
----------TestBean.java----------
package p1;
 import java.util.Date;
import java.util.List;
import javax.annotation.Resource;
import javax.annotation.Resources;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

@Component("tb") → Bean id
public class TestBean implements TestInter
{
    //Bean Property
    UserBean u1; → reference Type Bean property

    @Value("10")                                    91-92 injects value '10' to property "no".
    int no; → Simple Bean property

    Date d1;

    @Value("#{T(java.util.Arrays).asList('India','Bharat','Hindustan')}")
    String nicknames[]; → Array type property
    @Value("#{T(java.util.Arrays).asList('Red','Blue','Green')}")
    List colors; → java.util.List type

    @Resource(name="dfb") → refer line no: 151
    public void setD1(Date d)
    {
        d1=d;
    }

    @Resource(name="ub") → refer line no: 135
    public void setU1(UserBean u1)
    {
        this.u1=u1;
    }


    public String sayHello()
    {
    System.out.println("no"+no);
    System.out.println("d1"+d1.toString());
    System.out.println("nicknames");

    for(int i=0;i<nicknames.length;i++)
        System.out.println(nicknames[i]);

    System.out.println("Colors="+colors.toString());
    System.out.println("u1="+u1);
    return "Good Morning";
    }
}
----------UserBean.java----------
package p1;

import org.springframework.stereotype.*;

@Service("ub") → we can also use @Component annotation.
public class UserBean
```

```
{
    String msg="Hello World";

    public String toString()
    {
        return "DemoBean.msg"+msg;
    }
}
----------DateFactoryBean.java----------
package p1;

import org.springframework.beans.factory.FactoryBean;
import org.springframework.stereotype.Component;

@Component("dfb")
public class DateFactoryBean implements FactoryBean
{
    public Object getObject() throws Exception {
        return new java.util.Date();
    }
    public Class getObjectType() {
        return java.util.Date.class;
    }
    public boolean isSingleton() {
        return false;
    }

}
----------TestClient.java----------
package p1;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.*;

public class TestClient
{
    public static void main(String s[])
    {
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
        ctx.scan("p1"); //p1 is the package name
    ctx.refresh();
        TestInter test=(TestInter)ctx.getBean("tb");
        System.out.println("result is"+test.sayHello());

    }
}
```

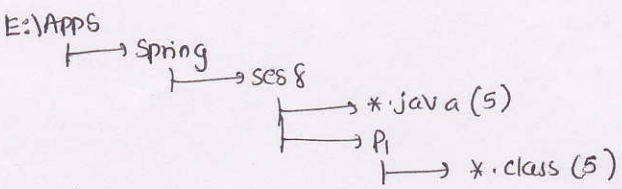
UserDefined FactoryBean class returning java.util.Date class object.

→ resultent object

→ Annotations Based Application Context Container

↳ scans the Every class of P1 package

↳ To make container loading the classes.



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 Workspace 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 DependencyInjection, if Dependency Injection Configurations are done directly on BeanProperties then there is no necessity of writting 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 (singleton, 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 supplimentary handout given on 08/03/13

*) for spring 3.0 annotations based example application that uses properties file refer appn (2) of Supplimentary handout given on 08/03/13

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

*) 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 resultent Object to another Beanclass property refer appn (4) of th 08/03/13 handout

08/03/2013

We can also use @Inject annotation to also perform explicit SetterInjection in the similler fashion.

3/8/2013 9:46 AM

```
>>>>>>>>>>>>Example Applications on Spring Annotations(IOC)>>>>>>>>>>>>
App1 (on constructor Injection)
----------Demo.java----------
public interface Demo
{
public String sayHello();
}
----------DemoBean.java----------
import javax.inject.Inject; ---> for @Inject
import org.springframework.beans.factory.annotation.Value; ---> for @value
import org.springframework.stereotype.Component; ---> for @component
@Component("db") -> bean id
public class DemoBean implements Demo
{  int age;
   String name;   } BeanProperties
   float avg;
      @Inject
public DemoBean(@Value("30")int age, @Value("Raja")String name, @Value("50.67")float avg) {
  this.age = age;
  this.name = name;
  this.avg = avg;
}
 public String sayHello()
 {
 return "Welcome to Spring age= "+age+" name= "+name+" avg= "+avg;
 }
}
----------DemoClient.java----------
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class DemoClient {
public static void main(String s[]) throws Exception {
 ApplicationContext ctx = new AnnotationConfigApplicationContext(DemoBean.class); -> Activates the annotations based springcontainer.
      //ask spring container to give springbean class object
      Demo d1=(Demo)ctx.getBean("db"); -> line no: 12
      // calls B.method of SpringBean class
      String result = d1.sayHello();
      System.out.println(result);
 }
}
-----------------------------------------------------
App2 ( Reading values from properties file) (3.x environment)
-----------------------------------------------------
App.properties
--------------
jdbc.driver=oracle.jdbc.driver.OracleDriver
jdbc.url=jdbc:oracle:thin:@localhost:1521:xe
db.user=noc1
db.pwd=noc1
----------select.java----------
package p1;
public interface Select
{
  public void readDBDetails();
}
----------SelectBean.java----------
package p1;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.ImportResource;
import org.springframework.stereotype.Component;
      -> to also use the configurations done in springcfg file(Bcz properties file is configured there)
@ImportResource("classpath:/SpringCfg.xml")
@Component("sb") -> Bean id
public class SelectBean implements Select
{   private @Value("${db.user}")
    String uname;   -> key in the properties file (refer line no: 50)
```

Same as appn ② of previous ⑦ jar files and

jar files in CLASSPATH :- /D/jar/Spring3latest/javax.inject.jar" />

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

jarfiles in CLASSPATH :- Same as appn ①

67-68
Injects value collected from "db.user" key of properties file to "uname" property.

```
69          private @Value("${db.pwd}")
70      String pwd;
71      @Override
72      public void readDBDetails() {
73          System.out.println("userName="+uname);
74          System.out.println("Password="+pwd);
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/>
88      <context:component-scan base-package="p1"/>
89      <context:property-placeholder location="./App.properties"/>
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")
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
132 @PostConstruct
133 public void myInit()
134 {
135       System.out.println("Init-Method");
136       if(age<0){
```

↳ refer lineno: 51

} B. method implemented

→ instructs to recognize annotations

→ makes the container to looks for the resources in 'p1' package

→ recognizes the properties file of current directory (.)

→ BeanId

→ makes the myInit() method as custom-init() method

```
137          age=18;
138          System.out.println("Age cannot be -ve,it is set to 18");
139        }
140  }
141  @PreDestroy ──→ make the myDestroy() method as Custom-destroy()
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  }
153  ----------DemoCfg.xml----------
154  <beans xmlns="http://www.springframework.org/schema/beans"
155        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
156        xmlns:context="http://www.springframework.org/schema/context"
157        xsi:schemaLocation="http://www.springframework.org/schema/beans
158        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
159        http://www.springframework.org/schema/context
160        http://www.springframework.org/schema/context/spring-context-2.5.xsd">
161         <context:component-scan base-package="p1" />
162  </beans>
163  ----------DemoClient.java----------
164  package p1;
165  import org.springframework.context.support.ClassPathXmlApplicationContext;
166  public class DemoClient {
167        public static void main(String s[]) throws Exception {
168        ClassPathXmlApplicationContext ctx=new ClassPathXmlApplicationContext("DemoCfg.xml");
169              // ask spring container to give springbean class object
170        Demo d1 = (Demo) ctx.getBean("db");
171        // calls B.method of SpringBean class
172        String result = d1.sayHello();
173        System.out.println(result);
174
175        ctx.close();
176        }
177  }
178  ----------
179  App4 (on Factory Bean )
180  ----------TestBean.java (FactoryBean generating Date class obj representing the given Date value)
181  package p1;
182  import java.text.SimpleDateFormat;
183  import java.util.Date;
184  import org.springframework.beans.factory.FactoryBean;
185  import org.springframework.beans.factory.annotation.Value;
186  import org.springframework.context.annotation.Scope;
187  import org.springframework.stereotype.Component;
188
189  @Component("tb") ──→ Bean Id
190  @Scope("prototype") ──→ for every Bean on seperate Bean obj will be created
191  public class TestBean implements FactoryBean
192  {                                  └→ manditory
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"); method (refer march 7th ctx.refresh(); 03/13 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     }
```

Module - II

DAO Module

09/03/2013

JNDI :-

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

Weblogic gives Weblogic registry

Tomcat gives Tomcat registry

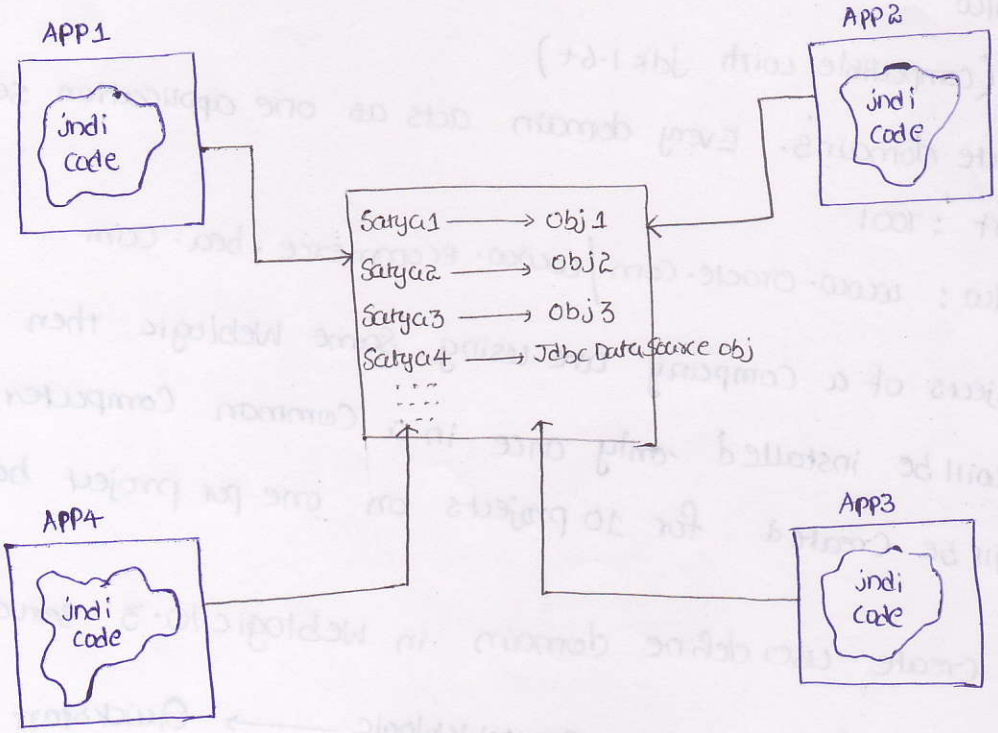
GlassFish gives GlassFish registry

JBOSS gives Jnp registry and etc...

Java applications use jdbc api (javax.sql, java.sql pkgs) to interact with DB s/w

Java applications use jndi api (javax.naming and its sub pkgs) to interact with Registry

jdbc, jndi api are part of JSE module (jdk s/w)



→ There are two types of registry s/ws

1) Naming Registry s/w

→ 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 coise it give) } (real life)
Windows filesystem

Weblogic registry
GlassFish registry
DNS registry
and etc...

Weblogic

type : Application Server slw

Vendor : BEA Systems (Oracle Corp)

Commercial slw

Version : 10.3 (compatible with jdk 1.6+)

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

default port : 7001

to doconload slw : www.oracle.com / www.ecommerce.bea.com

If multiple projects of a company are using same Weblogic then the weblogic slw will be installed only once in a common computer but 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 →

⊙ Getting started with Weblogic Server → ⊙ create a new weblogic domain

→ ⊙ Generate a domain Configured.... → Next →

Domain Name [MyDomain1] → Next →

User Name javaboss
User password javaboss1
Conform Userpass javaboss1
→ Next → ☑ Administration Server

→ Next → Listen port: 7171 → Next → Create

procedure to see the weblogic registry of Weblogic Server

- weblogic instance
weblogic Domain

Step I:- Start the above MyDomain1 server

Start → programs → OracleWeblogic → user projects → MyDomain1

→ start Admin Server for weblogic

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

Open Brocosr windoco → http://localhost:7171/console

Username : javaboss

Password : javaboss1 → Login

Step-III:- View the weblogic registry

Admin Console → Environment → servers → Admin Server → 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/aliasname/jndiname in registry

unbind → removing object from registry

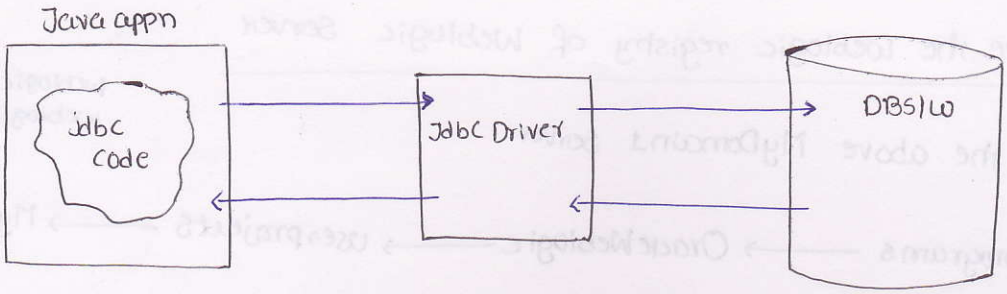
lookup → gathering object from registry with nickname/alias name

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

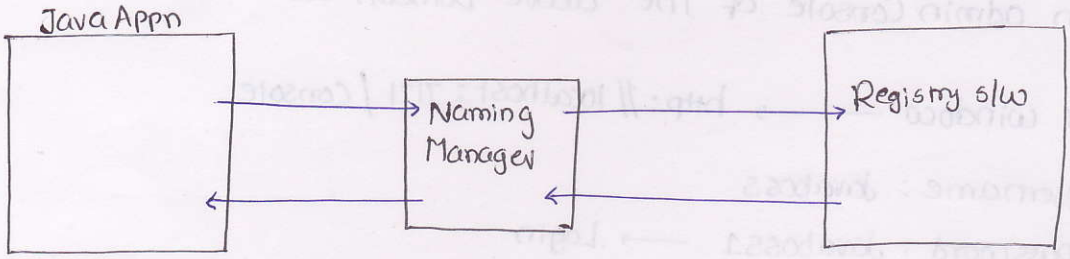
rebind → replacing existing object with new object

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

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/w we use



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



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

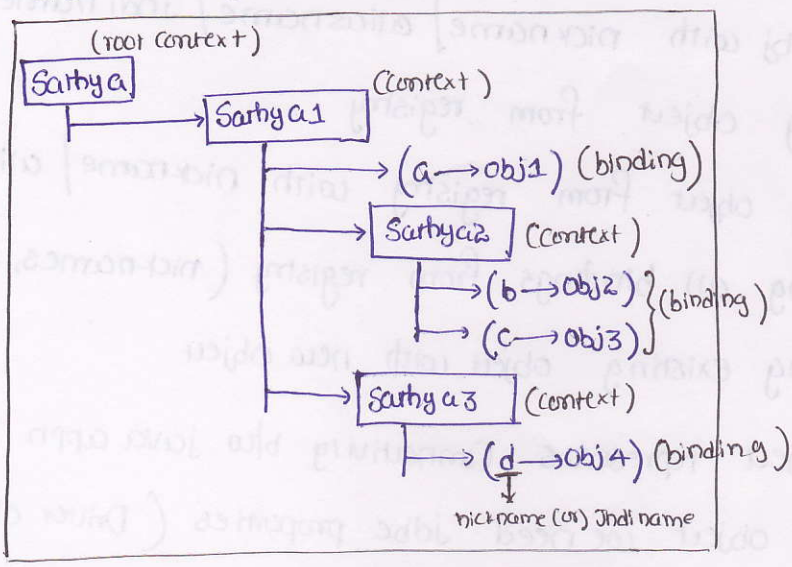
→ InitialContext obj represents connectivity b/w java appn and registry s/w

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



Sathya1 is subcontext of Sathya context and it is parent context of Sathya2, Sathya3

NOTE:- In real time projects the DataSource object that represents JDBC Connection pool will be placed in Registry s/w 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 class JndiConnTest
{
  ps v m(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");
  // Create Initial Context object
  InitialContext ic = new InitialContext(ht);
   if(ic==null)
     S.o.p("Connection is not established");
   else
     S.o.p("Connection is established");
  }//main
}//class
//add Weblogic.jar (<Weblogic_home>\Middleware\Wlserver_10.3\server\lib\weblogic.jar in class path
```

Fixed key

InitialContextFactory class name

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

bind(-,-)

unbind(-)

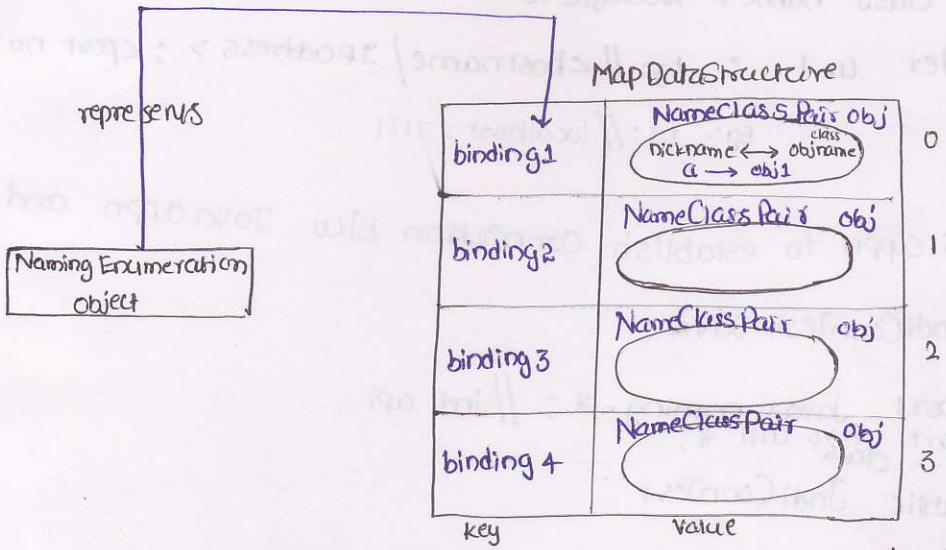
rebind(-,-)

list(-)

lookup(-)

11/03/2013

JndiAPI is related to JS Emodule 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
{
    p s v m (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

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

```
//performing list operation
//NamingEnumeration ne = ic.list("");  //gives all the binding of root context
                                       (represents root context)
NamingEnumeration ne = ic.list("java:/jms");  //gives all bindings of
                                                 java:/jms context
While (ne.hasMore())
{
NameClassPair ncp = (NameClassPair) ne.next();  //gives one binding at a time.
S.o.p (ncp.getName() + "<-->" + ncp.getClassName());
}
                    gives nickname        gives class name of the object
} //main
} //class
```

(*) 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=4545 --user=testuser mydomain2~~

D:\sun\Appserver\bin> asadmin create-domain --adminport=4646 --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:4646

username: testuser

password: testuser ⟶ Login

Step-III:- give the glassFish registry.

Admin Console ⟶ Application Server ⟶ Jndi Browsing ⟶

*) GlassFish server gives GlassFish registry as the built-in registry s/w.

Jndi properties of GlassFish registry

Initial Context Factory class name : com.sun.enterprise.naming.SerialInitContextFactory (*1)

provider url : iiop://<host>:<port no>

eg:- iiop://localhost:4545

(*1) This class is available in Appserv-rt.jar and this jar file bewing multiple dependent jar files. They are ↳ D:\sun\Appserver\lib

- 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 s/w's (refer pageNO: 57 to 59)

iiop ⟶ inter internet ORB protocol

↳ Object Request Broker

12/03/2013

JBoss

- Type : Application server s/w
- version : 5.x (Compatible with jdk 1.5)
- Vendor : Apache (RedHat)
- gives 5 default domains
  - default (default)
  - web
  - Standard
  - all
  - minimal
- to download s/w : As zip file from www.apache.org website
  Jboss - 5.1.0.GA - jdk1.5
- to install s/w : extract zip file to a folder
- default port no : 8080 (to access the webapplications)
  1009 (to access the jnp registry)
- to start the server : use <Jboss_hom>\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
(package name; available in <jboss_home>\client\ jnp-client.jar file)

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

Eg: jnp://localhost:1099

its dependent jars 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 s/w.

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 plain Jndi but it never makes programmer to bother about plain Jndi (This is nothing but getting abstraction Layer).

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(-,-), rebind(-,-), unbind(-) and lookup(-) methods but it is not giving list().

(*) 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 underlaying 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 acheived 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
    } //innerclass
); // method call
```

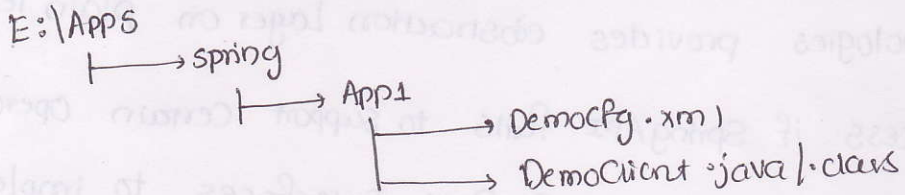
In the above code:

a) addWindowListener( ) method is called having one object as the argument value.

b) → That object is the object of anonomamous Inner class that extends from WindowAdapter class.

c) → This Anonomamous Inner class is Overridding WindowClosing(-) method having same logic

(*) Example Application on to perform JndiCallback Interface to implement the list operation functionality.



DemoCfg.xml same as app(12) pageno: 59

// DemoClient.java

```
import org.sf.context.support.*;
import org.sf.jndi.*;
import javax.naming.*; ——→ plainJndi to coork with Callback (I)

public class DemoClient
{
  static JndiTemplate template;
    //setter method for setter Injection
    public void serTemplate(JndiTemplate template)
    {
      this.template = template;
    }
    public static void main(String[] args) throws Exception
    {
      FileSystemXmlApplicationContext ctx = new FileSystemXmlApplicationContext
                                            ("DemoCfg.xml");
      //implement list operation... logic
```

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

NamingEnumeration ne = ctx.list("");
(*1)
  while(ne.hasMore())
  {
NameClassPair np = (NameClassPair)ne.next();
S.O.P(np.getName()+"  ⟶  "+np.getClassName());
  }//While
  return null;
 }// doInContext(-)
}// inner class
); //execute(-)
}//main
}//class
```

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

```
//> javac *.java
//> java DemoClient
//make sure that Weblogic Server is there in running mode.
```

→ In spring Environment if spring doesn't contain any method manually then we use Spring JndiCallback Interface to acheive that method

In Spring Application we can use 3 types of Jdbc con pools

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

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

② Third party JDBC con pool

→ Apache DBCP

→ C3P0

(Recomended to use in Standalone spring Apps (Applications that run outside the server).

③ Webserver / Application Server Managed JDBC con pool

(Recomended to use the Spring appns 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\jakartha-commons\commons-dbcp.jar
- class name : org.apache.commons.dbcp.BasicDataSource

imperent properties of BasicDataSource class

driverClassName
initialSize
password
url
username

(*) Example application

E:\Apps
→ spring
→ ApacheDBCPApp
- DBOperation.java
- DBOperationBean.java
- DBOperationClient.java
- SpringCfg.xml

(5) same as 18/02/2013 appn

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\jakarta-commons folder
② spring.jar
③ commons-logging.jar

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

C3P0

type : third party jdbc con pool s/w

jar file : <spring_home>\lib\C3P0\C3P0-0.9.1.2.jar

class name that gives

important properties

password
User
minPoolSize
maxPoolSize
jdbcUrl
driverClass

Example application :-

Same as previous appn but perform the following operations.

① Add following entries in SpringCfg.xml file.

"com.mchange.V2.c3p0.ComboPooledDataSource

```
<beans>
  <bean id="dbcp" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="driverClass" value="oracle.jdbc.driver.OracleDriver"/>
    <property name="jdbcUrl" value="jdbc:oracle:thin:@localhost:1521:ora"/>
    <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="DBOperationBean">
    <property name="ds" ref="dbcp"/>
  </bean>
</beans>
```

add <Sprin_home>\lib\c3p0\c3p0-0.9.1.2.jar file to CLASSPATH Environment Variable as additional jar file.

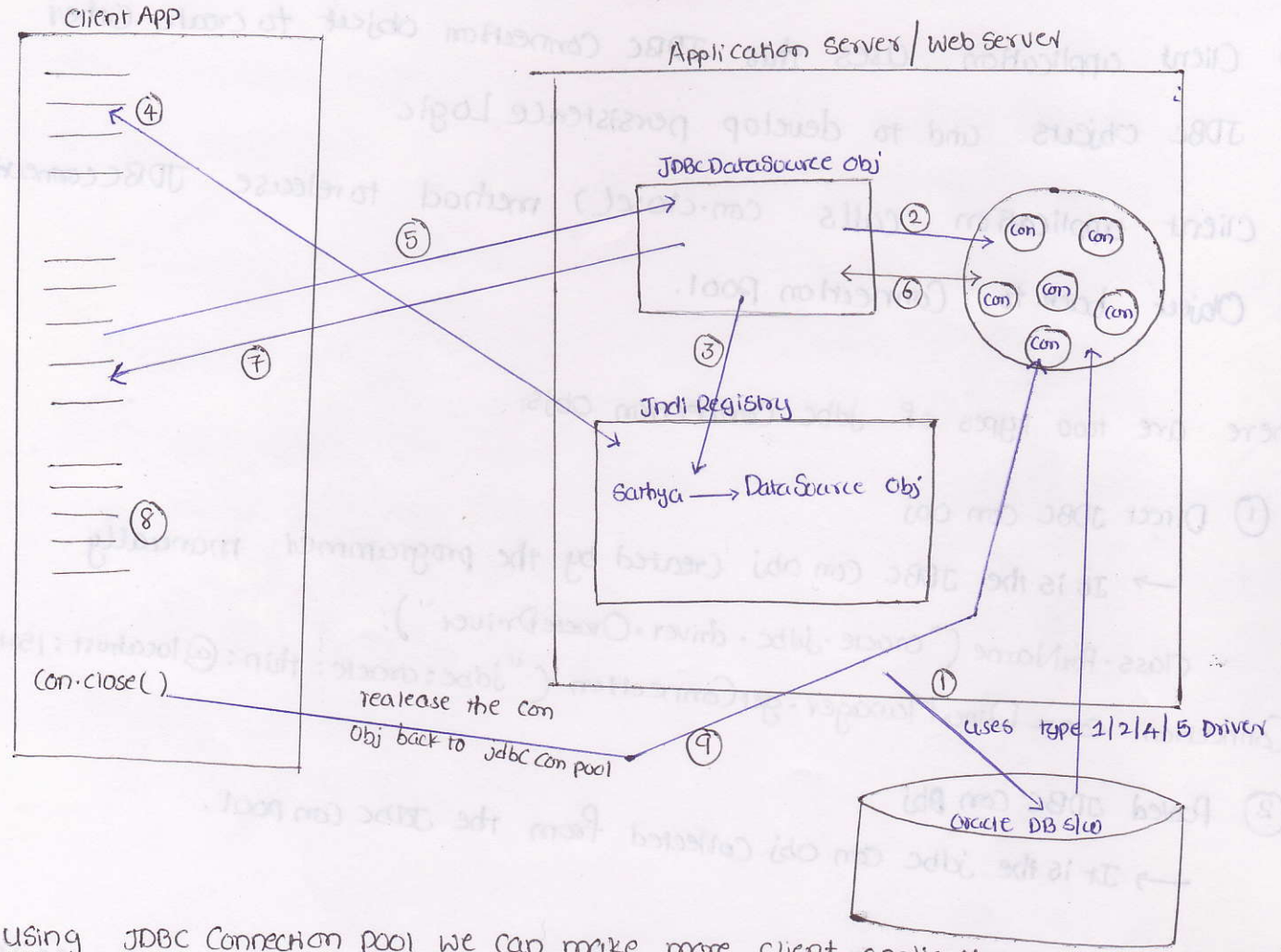
Q:- What is the JDBC connection pool that it have used in your project?

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 Deployeeable 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 Objs 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.to 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 objs.

(2) & (3) Server Creates JDBC DataSource 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.

⑤⑥⑦ Client application gets JDBC Connection object from ConnectionPool through DataSource 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 objs

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: [myds1] → JndiName: [Sathya] → DataBase type: [Oracle]

DB driver [Oracle Driver (Thin) for Service Connection] → next → next →

DataBase Name: [orcl] Host name: [localhost] port: [1521] DataBase Username: [scott]

Password [tiger] Confirm Password [tiger] → next → TestConfiguration

→ next → ☑ AdminServer → Finish.

Specify the JDBC Conpool parameters

Admin Console → Services → JDBC → DataSources → mydS 1 →

Connection Pool tab → Intial capcity [3] → Max capacity [30]

capcity increment: [2] → advanced → Shrink frequency [900] seconds

→ save

↳ no. of connect objects that should be created when there is a need of creating new Jdbc Connection objects.

destroys IDle JDBC connection objects after every "900" seconds.

Server → Admin → Server → JNDI tree

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

*) Org.sf.jndi.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

Step1:- Keep mydomain1 Server of weblogic in running mode.

Step2:- Keep Feb 18th appn ready. and perform the following modifications in Springcfg.xml file.

```
//SpringCfg.xml
<!DOCTYPE ... >
<beans>
  <bean id="jofb" class="org.sf.jndi.JndiObjectFactoryBean" >
    <property name="jndiName" value="sankya"/>
                                    ↳ 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:7171
                                          </prop>
  </props>
</property>
</bean>
<bean id="dob" class="DBOperationBean">
  <property name="ds" ref="jofb" />
</bean>
</beans>
```

Injects JDBC DataSource Object gathered from Jndi registry by using JndiObjectFactoryBean. its resultent 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 Server of GlassFish 2.x.

Step-0:- keep oracle thin driver related ojdbc14.jar file <Glassfish_home>\appserver\domains\mydomain2\lib\ext folder

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

asadmin start-domain mydomain2

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

AdminConsole Screen → Resources → JDBC → Connection Pools →

→ New → Name: [mypool1] → resource type [javax.sql.DataSource.]

Vendore: [oracle] → next → Initial and min Pool size [4] →

Maximum Pool size [20] → pool resize Quantity: [2] → ~~Idle time~~

→ ☑ 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 → JDBC Resources →

new → JNDI Name: ~~Sathya~~ MyDsJndi → pool name: mypool1 → OK

Keeps DataSource obj in Jndi Regis
AppnServer → Jndi Brows
→ see Jndi tree

(*) ~~Step-IV~~ Refer appn on 18/02/2013 modify SpringCfg.xml to use GlassFish server manged 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 Sathya

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

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

2 → 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 persistance logic from other logics of the application and makes that logics as reusable logics is called DAO

→ DAO Design pattern is no way related with SpringDAO module.

→ SpringDAO module gives JDBC Template class to provide abstraction layer on plain JDBC programing that means simplifies the process of developing JDBC persistance Logic.

Plain JDBC programing

(1) register jdbc driver with DriverManager service
(2) Establish the connection with DB s/w
(3) create JDBC statement obj
} Commonlogics of the application (same in all JDBC appns)

(4) send and execute SQL Query in DB s/w
(5) Gather results and process the results
} Application specific logic (changes in every appn)

(6) close JDBC objs
(7) performs Exception Handling
} Common logics of the appn (same in all JDBC appns)

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 s/w
(3) Gather results and process the results
} Application specific logics

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

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

→ dataSource is the important property of JdbcTemplate class (org.sf.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 BatchProcessing :-

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 :-

E:\APPS
- Spring
  - ses5
    - DBOperation.java
    - DBOperationBean.java
    - DBOperationCfg.xml
    - DBOperationClient.java

Prefer using Third party jdbc con pool like Apache DBCP.

DB table in Oracle

Student
- sno number(4)
- sname varchar2(20)
- sadd varchar2(20)

```
// 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 newaddr);
    public int deleteDetails(int no);
    public int countStudents();
    public Map getStudDetails(int no);      --> Map : to execute select query for single record.
    public List getStudDetails(String city); --> List : to execute selecting query for multiple records.
}
```

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

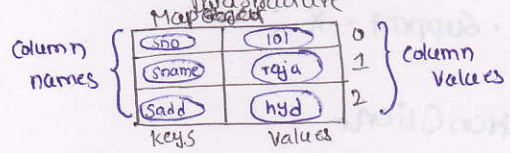
```
//implement B.methods
public int insertInfo (int no, String name, String addrs)
 {
int result = jt.update("insert into Student values(?,?,?)",
                        new Object[]{no, name, addrs});
  return result;
 }
public int updateDetails (int no, String newname, String newaddrs)
 {
int result = jt.update("update student set sname=?, sadd=? where
          sno=?", new Object[]{newname, newaddrs, no});
  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 getStudentDetails(String city)
{
```

JDBC Template through DataAccess Exception. This is an unchecked exception. so, exception handling in DAO module is optional.

Supplies the query parameter values

AutoUnboxing concept

using PreparedStatement objects.

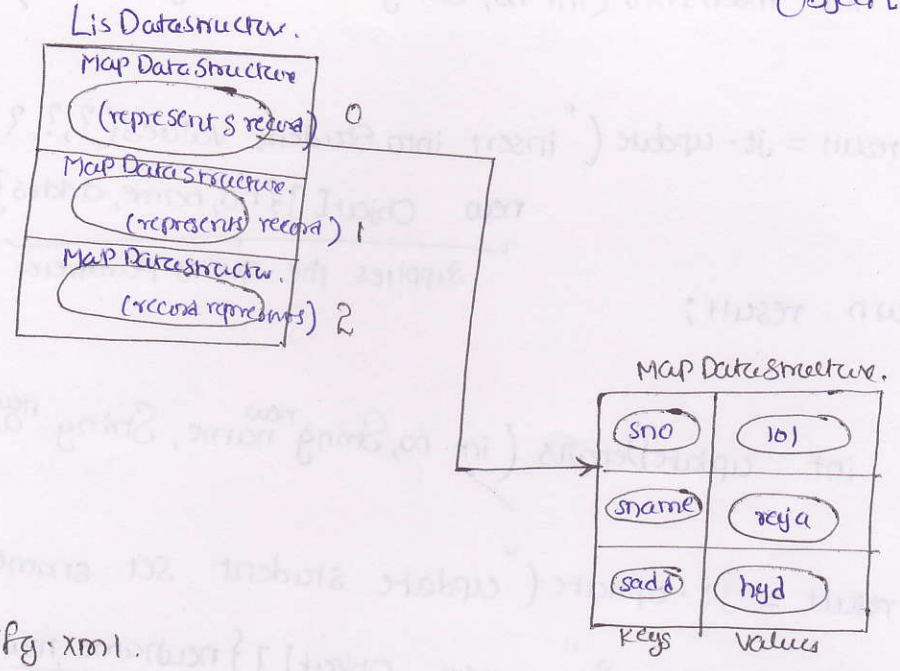


List l = jt.queryForList("select * from Student where scadd = ?", new Object[]{city});

return l;

}

}



// DBOperationCfg.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="DBOperationBean">
    <constructor-arg ref="template" />
  </bean>
</beans>
```

//DBOperationClient.java

```
import org.sf.context.support.*;

public class DBOperationClient
{
  psvm(String[] args)
  {
```

```
//Activate Container
FileSystemXmlApplicationContext ctx = new FileSystemXmlApplicationContext("DBOperationCfg.xml");
//getBean class obj
DBOperation bobj = (DBOperation) ctx.getBean("dob");
//callMethods
S.o.p("no.of Records that are inserted" +bobj.insertInfo(101,"raja","byd"));
S.o.p("no.of Records that are Updated" + bobj.updateDetails(101,"newraja","newhyd"));
S.o.p("no.of Records that are deleted" +bobj.deleteDetails(101));
S.o.p("no.of students Count" +bobj.countStudents());
S.o.p("101 student details" +bobj.getStudDetails(101));
S.o.p("hyd student details" +bobj.getStudDetails("hyd"));
}//main
}//class
```

jar files in CLASSPATH :-

2 Spring jars ⟶ spring.jar, Commons-logging.jar
Commons-dbcp.jar
OJdbc14.jar

Q:- What is the diff blw executing selectquery in plainJDBC and in SpringJDBC?

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 CheckedException called SQLException. The Spring JDBC converts that one to Unchecked Exception called DataAccessException.

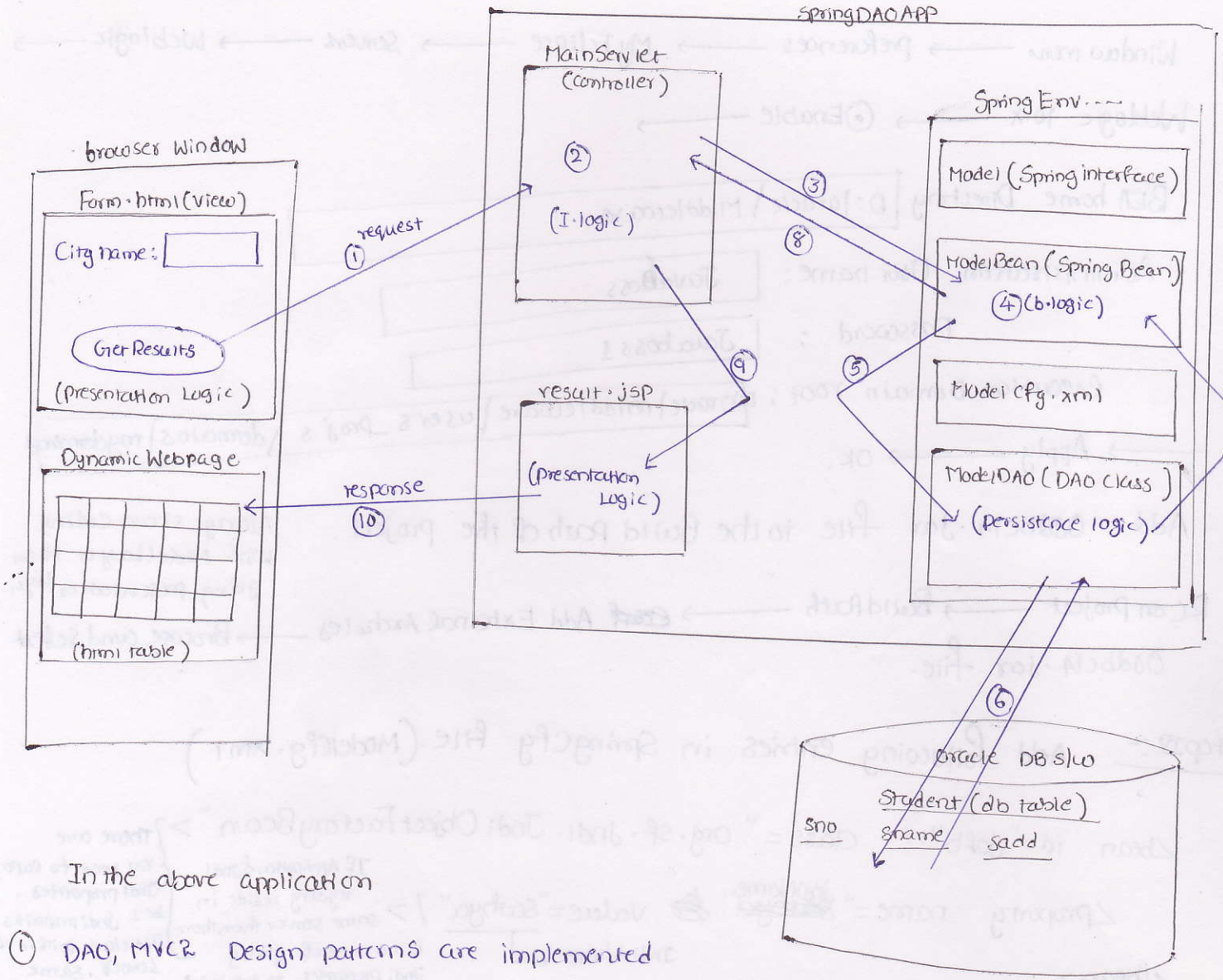
Eg:-

```
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 others logics of the application is called DAO. DAO makes the persistence logic as reusable logic and the flexible logic to modify.

Example Webapplication based on mvc2 architecture and implementing DAO Design pattern



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 reoccurring problems of application Development

Design Patterns are the Best practices to develop s/w applications by using s/w Technologies

Eg ⓧ procedure to develop the above webapplication by using MyEclipse 10.x Server.

Step-I:- Create web project

File → New → WebProject

Proj Name: TestProj1 → Finish

Step-II:- Add spring capabilities to the project

Right click on project → MyEclipse → Add spring capabilities → ⊙ 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 MyEclipse IDE.

Window menu → Preferences → MyEclipse → Servers → Weblogic →

Weblogic 10.x → (•) Enable →

BEA home Directory | D:\oracle\Middleware

Administration User name : | Javaboss

Password : | Javaboss1

Execution Domain root : | D:\oracle\middleware\users_proj's\domains\mydomain1

→ Apply → Ok.

Add OJdbc14.jar file to the Build Path of the project.

Always start coding with Model layer then going presentation layer.

RC on project → BuildPath → ~~Conf~~ Add External Archives → Browse and Select Ojdbc14.jar file.

Step-IV :- Add following entries in SpringCfg file (ModelCfg.xml)

```
<bean id="jofb" class="org.sf.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="scathya" />
</bean>
```

Jndi Name

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 with in the server. same location.

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

<bean id="mydao" class="ModelDAO">
    <property name="jt" ref="template" />
</bean>

<bean id="mb" class="ModelBean">
    <property name="mdao" ref="mydao" />
</bean>
```

Step-V:- Develop the DAO class as shown below → this having only persistence logic

RC on src → class → className: ModelDAO → Finish.

```
public class ModelDAO {
    // bean property
    JdbcTemplate jt;
    // Write setXxx(-) method for setter Injection
    public void setJt(JdbcTemplate jt)
    {
        this.jt = jt;
    }
    public List<Map> getDetails(String city)
    {
        List<Map> l = jt.queryForList("select * from student where sadd=?",
                                      new Object[] {city});
        return l;
    } //method
} //class
```

Step-VI:- Add SpringInterface, SpringBean class to the project
(SpringInterface → Model; SpringBean → ModelBean)

RC on src → New → Interface → Name: Model → finish.

```
//Model.java
public interface Model {
    public List<Map> 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 List<Map> searchStudents(String city) {
        //b.logic
```

```
if (city == null || city.equals("") )
{
    city = "hyd";
}
//use persistance logic of DAO class
List l = mdao.getDetails(city)
    return l;
} //method
} //class
```

Step-VII :- add Form.html to the webroot folder of the project

R.C → 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 → new → Servlet → Name : MainServlet → ☑ init & ☑ destroy, ☑ doGet(), doPost()

→ next → Servlet/JSP mapping URL : /Controller → Finish.

```
public class MainServlet extends HttpServlet {
  Model bobj;
  public void init(){
    //Activate SpringContainer
ClasiPathXmlApplicationContext ctx = classPathXmlApplicationContext("model1cfg.xml");
    //get SpringBean class object from Container
    bobj = (Model) ctx.getBean("mb");
  } //init()
Public Void doGet(HSR request, HSR response) throws SE,IOE {
    //read form data
String s1 = request.getParameter("city");
    // call B.method
```

```
List l = bobj.searchStudents(s1);
    //keep result in Request attribute
    request.setAttribute("attr1", l);
    //forward the Request to result page
RequestDispatcher rd = request.getRequestDispatcher("result.jsp");
    rd.forward(request, response)
} //doGet(--)

public void doPost(HSR request, HSR response) throws SE, IOE
{
    doGet(request, response);
}
public void destroy(){
    bobj = null;
}
}
```

Step-XI: add result.jsp to Webroot folder

R.C on webroot → new → JSP → 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 I:- Run the project

RC on project → Run as → MyEclipse Server application →

Weblogic10.x

Test the project.

Open Browser window → http://localhost:7171/TestProj1/Form.html

Procedure to configure glassFish 2.X Domain Server with MyEclipse IDE.

Window menu → preferences → MyEclipse → 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 underlaying 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 JDBCTemplate class, to work with Named Parameter we need NamedParameterJdbcTemplate class

*) For example application on NamedParameters 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 con object to create and use jdbc PreparedStatement obj

PreparedStatementSetter ⟶ gives PreparedStatement obj to use it.

(*) For example application on SpringDAO 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/w

→ 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.sf.jdbc.Object.StoredProcedure 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) AppnNO: (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 Logics

→ 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, IBatis, toplink and etc...

→ Spring ORM module does not supply its own ORM s/w. but its provide abstraction Layer on existing ORM s/w and Simplifies the process of working with ORM s/w's.

for this it supplies multiple Template classes.

ORM s/w	its Template class
Hibernate	org.sf.orm.hibernate3.HibernateTemplate
ibatis	org.sf.orm.ibatis.SqlMapClientTemplate
Jdo	org.sf.orm.jdo.JdoTemplate
Jpa	org.sf.orm.jpa.JpaTemplate
Toplink	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 HibernateSessionFactory obj (LocalSessionFactory)
→ Bean ~~that~~ of ~~gives~~ HibernateTemplate class

imperent methods of HibernateTemplate class

for Single row operations:

save(-) / persist(-) → for record insertion
update(-) → for record updation
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(), ~~find()~~ → for HQL based select operations.

Module - III

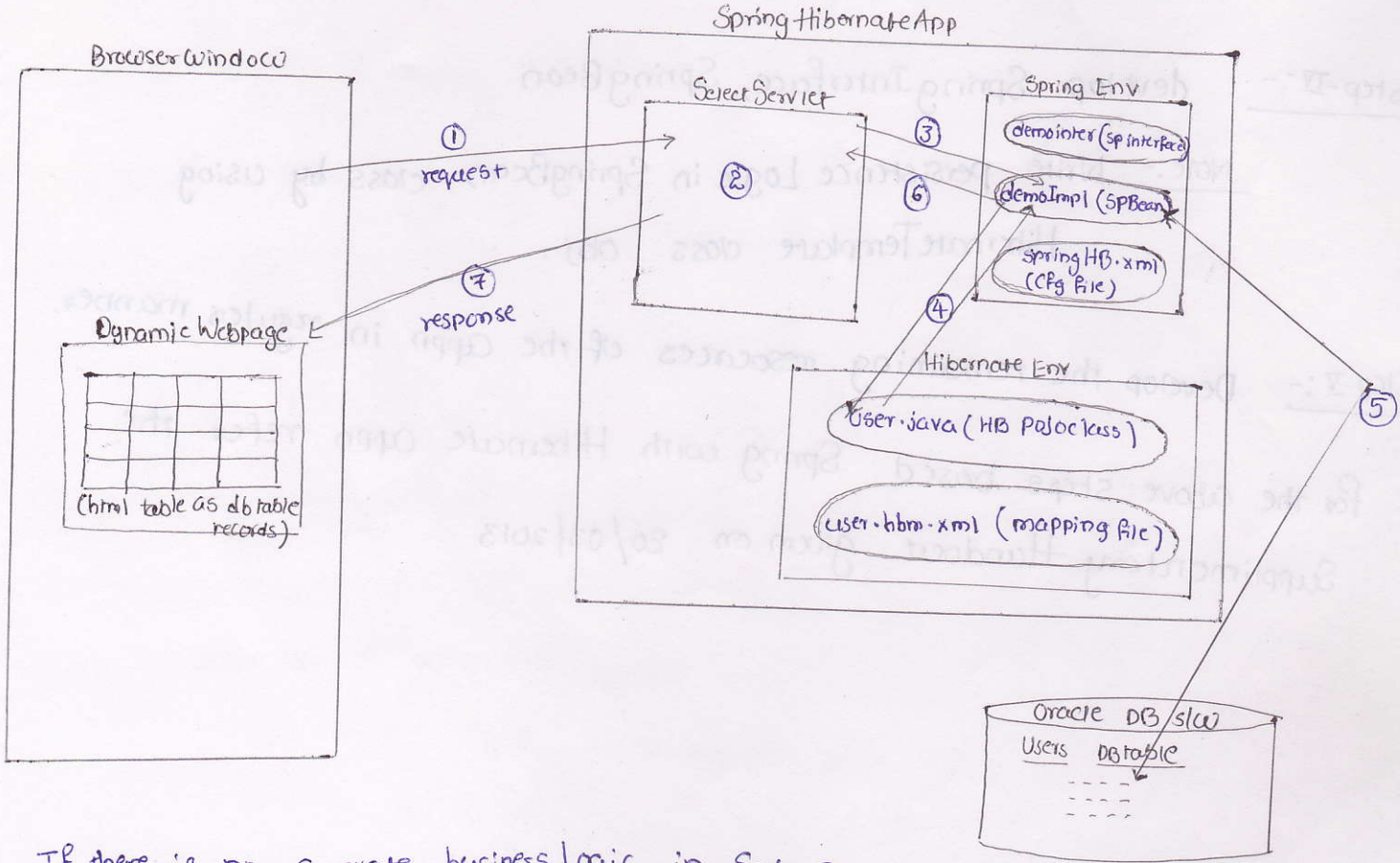
Spring ORM Module

→ 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 persistance logic in Spring appn.

→ To write Hibernate ~~appn~~ persistence logic in SpringBean class by using Spring ORM module we need HibernateTemplate class obj.

Spring with Hibernate Appn (using HibernateTemplate class obj)



If there is no separate business Logic in SpringBean then persistence logic itself acts as B.Logic of the SpringBean

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 Springbean

→ Our Spring Bean injecting HibernateTemplate class obj.

Step-IV :- develop SpringInterface, SpringBean

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 HQL Query that is placed in mapping file having logical name is called "Named HQL query" and by Named param.

*) Executing Named 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="User" />   (HB POJO class)
      <![CDATA[ select * from users where userid >=? and userid<=? ]]>
   </sql-query>
</h-m>
```

Step-II:- execute the above Native SQL query.

```
public Iterator getData()
{
List l=ht.findByNamedQuery("test2", new Object[]{100,300});
   Iterator it=l.iterator();
      return it;
}
```

*) Executing Named Native SQL query with Named parameters.

Step-I:- prepare Named 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.findByNamedQueryAndNamedParam("test2", new String[]{"p1"}, new Object[]{"b.l
   Iterator it=l.iterator();
```

```
// to insert Single record
User u1=new User();
u1.setUid(1001);
u1.setUname("Rajesh");
u1.setRole("P.L");
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.findByNamedQueryAndNamedParam("test2",new String[]{"p1"},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.l","t.l"});
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 objs

and etc....

*) The following operations are not possible directly with HibernateTemplate class.

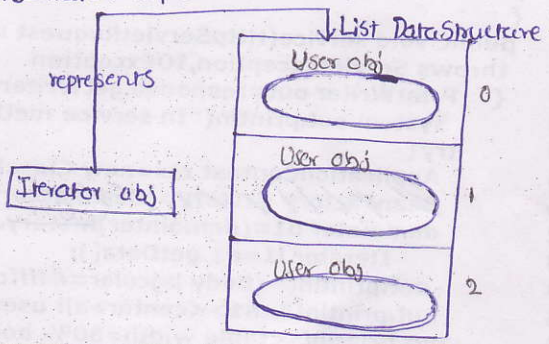
(1) Non-select HQL Queries with NamedParameters
(2) Non-select Native SQL Queries with Named (or) positional parameters.
(3) ~~HQL select query with Named parameters~~ Criteria API based Logics execution
(4) Native SQL select query with Both Named, positional parameters.

To implement all these facilities we need to work with HibernateCallback interface (org.sf.orm.hibernate3.*) supplied by SpringORM module. This interface gives doInHibernate(-) exposing the Session object.

20/03/2013

```
1  ==========================================================================
2  Title:Spring with Hibernate App (approach2)
3  ==========================================================================
4  ----------------------------------User.java-(Hibernate persistent class (or) Entity class)
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---------- Hibernate mapping file
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>
33       <class   name="User"  table="users" >   → POJO class
34            <id  name="uid"  column="userid"  />
35            <property  name="uname" />
36            <property  name="role" />
37     </class>
38
39 </hibernate-mapping>
40 ----------------------------------demointer.java---------- Spring Interface
41 import java.util.*;
42 public interface  demointer
43 {
44    public Iterator getData();
45 }                          → Business method Declaration
46 ----------------------------------demoimpl.java---------- Spring Bean class
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"
```

logic to inject Hibernate Template class object.



```
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
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
105  <bean id="template" class="org.springframework.orm.hibernate3.HibernateTemplate">
106    <property name="sessionFactory"><ref bean="mySessionFactory"/></property>
107  </bean>
108
109  <bean  id="d1"  class="demoimpl">
110    <property name="ht"> <ref bean="template"/> </property>
111  </bean>
112
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  {   PrintWriter out=response.getWriter();
128       System.out.println("In service method of Servlet");
129       try{
130        ApplicationContext ctx=new ClassPathXmlApplicationContext("SpringHB.xml");
131        BeanFactory factory=(BeanFactory)ctx;
132        demointer d1=(demointer)factory.getBean("d1");
133            Iterator i1=d1.getData();
134        out.println("<body bgcolor=#ffffcc text=red>");
135        out.println("<h1><center>all users</h1><hr><br><h3>");
136        out.println("<table width=80% border=2>");
137        while(i1.hasNext())
138        {
```

gives JDBC DataSource object (recomended to use either ServerManaged Connectionpool (or) Third party connection pool.

refer line no:173

gives HibernateSessionFactory obj by using the following entries

log

Hibernate related misillenceous properties configuration

ref line no:89

ref line no:105

Our spring Bean configuration injecting HibernateTemplate class obj

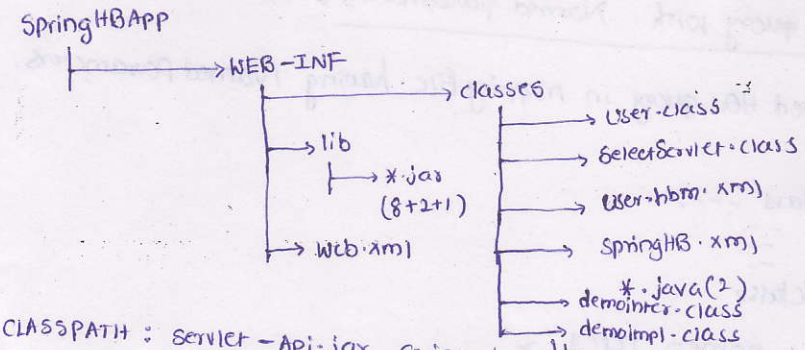
gives spring Bean class object

137-144 logic to display records as HTML table by gathering results from the Business methods of SpringBean.

```
            User u1=(User)i1.next();
            out.println("<tr><td>"+u1.getUid()+" <td>"+u1.getUname()+"
                        <td>"+u1.getRole()+"</tr>");
        }
        out.println("</table>");
      }
        catch(HibernateException e)
        {
            out.println(e);
            e.printStackTrace();
        }
  }
}
```

---web.xml---

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
         http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
                              version="2.4">
  <servlet>
     <servlet-name>select</servlet-name>
     <servlet-class>selectservlet</servlet-class>
   </servlet>
   <servlet-mapping>
     <servlet-name>select</servlet-name>
     <url-pattern>/selectaction</url-pattern>
   </servlet-mapping>
</web-app>
```



jar files in CLASSPATH : servlet-api.jar, Spring.jar, Hibernate3.jar

jar files in WEB-INF\lib folder : 2 ⟶ spring jar files
8 ⟶ Hibernate jar files
1 ⟶ OJDBC14.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});
                                  positional parameters      parameter values
      Iterator  it=l.iterator();
      return  it;
}
```

*) Executing HQL select query with Named parameters

```
public  Iterator  getData()
{
   List l=ht.findByNamedParam("from User where uid>=:P1 and uid<=:P2", new String[]{"P1","P2"},
                                  named parameters                    new Object[]{100,200});
      Iterator  it=l.iterator();
      return  it;
}
```

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.findByNamedQuery("test1", new Object[]{"%.P.1"});
    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.findByNamedQueryAndNamedParam("test1", new String[]{"P1","P2"},
                                          new Object[]{100,300});
    Iterator it = l.iterator();
    return it;
}
```

Executing Hibernate Persistance 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 persistance Logic
       Criteria ct=ses.createCriteria(User.class);
            //add Condition
       Criterion cond=Restrictions.like("role","%P.L");
        ct.add(cond);
            //execute logic
        List l=ct.list();
        return l;
          }//method
        }//inner class
       );//method
    Iterator it=l.iterator();
      return it;
   }
}
```

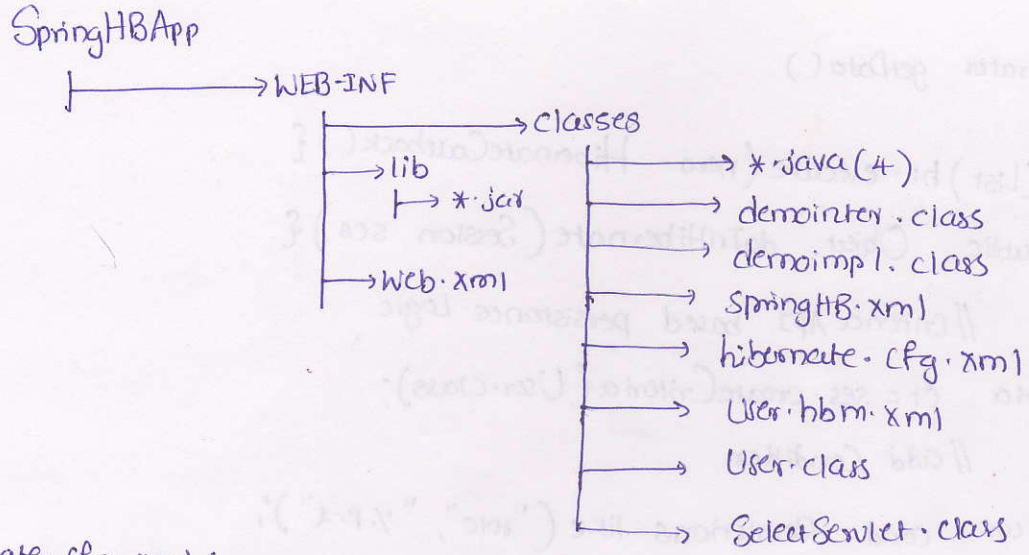
NOTE:- In the above code ht.execute() method is called having anonomus inner class object that implements HibernateCallback (I)

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 separetly. then Hibernate configuration file must be linked with Spring cnfg file by using configure Location property of LocalSessionFactoryBe 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



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"/>
  </bean>
    ||  Same old HibernateTemplate, DemoImpl classes Configuration
        refer previous appn.
</beans>
```

(*) For example application on annotations based spring, hibernate integration (spring 3, hibernate 4).
refer the supplimentary Handout given on 23/03/2013

Module - IV

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 TimeCritical Jobs this scheduling is very important.

Examples:-

Sending remainders 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 TimeTask 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 pageno's (67) and (68)

lines:

1975-1982 The logic on which we are looking to enable Timer Service

1992 ~~20000~~ * Initial delaying milli seconds.

5000 (*1) Specifies initial delay. 3000 (*2) 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 benfits.

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 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

23/03/2013


```
SpringHBAppAnno
     |-------------------->WEB-INF
                           |--------->classes
                           |           |--------------->p1
                           |           |              |-------->DemoInter.java/.class
                           |           |              |-------->DemoImple.java/.class
                           |           |              |--------->SelectServlet.java/.class
                           |           |              |---------->User.java/.class
                           |           |--->SpringHB.xml
                           |-------->lib
                           |         |---->(*.jar)
                           |-->web.xml
jars in lib
------------
antlr-2.7.7.jar
commons-logging.jar
dom4j-1.6.1.jar
hibernate-commons-annotations-4.0.1.Final.jar
hibernate-core-4.1.8.Final.jar
hibernate-ehcache-4.1.8.Final.jar
hibernate-jpa-2.0-api-1.0.1.Final.jar
javassist-3.15.0-GA.jar
javax.persistence.jar
jboss-logging-3.1.0.GA.jar
jboss-transaction-api_1.1_spec-1.0.0.Final.jar
ojdbc14.jar
org.springframework.asm-3.1.1.RELEASE.jar
org.springframework.aspects-3.1.1.RELEASE.jar
org.springframework.beans-3.1.1.RELEASE.jar
org.springframework.context.support-3.1.1.RELEASE.jar
org.springframework.context-3.1.1.RELEASE.jar
org.springframework.core-3.1.1.RELEASE.jar
org.springframework.expression-3.1.1.RELEASE.jar
org.springframework.jdbc-3.1.1.RELEASE.jar
org.springframework.orm-3.1.1.RELEASE.jar
org.springframework.transaction-3.1.1.RELEASE.jar

================================================================================
DemoInter.java
----------------
package p1;
import java.util.*;
public interface  DemoInter
{
    public Iterator getData()throws Exception;
}
================================================================================
DemoImpl.java
----------------
package p1;
import org.hibernate.*;
import java.util.*;
import org.springframework.stereotype.*;
import org.springframework.beans.factory.annotation.Autowired;

@Component("d1")
public class DemoImpl implements DemoInter
{
@Autowired
 private SessionFactory  sesfact;

       public void setSesfact(SessionFactory f1)
       {
        sesfact=f1;
       }

        public Iterator getData()throws Exception
        {
```

15-23 collect from hibernate 4.x
24-25 collect from jarfinder.com
27-36 collect from spring 3.1

jar files in CLASSPATH

javax.persistence.jar
org.sf.context.support-3.1.1.RELE
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

```
                Session ses=sesfact.openSession();
                Query query=ses.createQuery("from  User");
          Iterator i1=query.iterate();
          return i1;
         }
}
==========================================================================================
User.java
-----------
package p1;
import javax.persistence.*;

@Entity
@Table (name= "users")
public class User
{
   @Id
   @GeneratedValue
   @Column(name = "userid")
   private int uid;

   @Column(name= "uname")
   private String uname;

   @Column (name= "role")
    private String role;

    public void setUid(int n)
    {
      uid=n;
    }

    public int getUid(){ return uid; }
    public void setUname(String s){ uname=s; }
    public String getUname(){ return uname; }
    public void setRole(String r){ role=r; }
    public String getRole(){  return role; }
}
==========================================================================================
DBTable
-----------
  create table users
  ( userid number(5) primary key,
    uname varchar2(20),
   role varchar2(20)
   );
==========================================================================================
SpringHB.xml
---------------
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:component-scan base-package="p1" />

  <bean id="myds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"><value>oracle.jdbc.driver.OracleDriver</value> </property>
    <property name="url"> <value>jdbc:oracle:thin:@localhost:1521:orcl</value> </property>
    <property name="username"> <value>scott</value> </property>
    <property name="password"> <value>tiger</value>
    </property>
  </bean>

<bean id="sesfact" class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
```

recommended to configure either Apache DBCP (or) C3P0 (or) server managed connection(pool)

```
    <property name="dataSource" ref="myds"/>
      <!-- use this to avoid configuring mapping file-->
    <property name="packagesToScan" value="p1"></property>
</bean>
```

makes the Spring container recognizing all the classes that are there in 'P1' Pac

```
</beans>
==========
SelectServlet.java
------------------
package p1;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import org.hibernate.*;
import java.util.*;
import org.springframework.beans.factory.*;
import org.springframework.context.*;
import org.springframework.context.support.*;
public class  SelectServlet   extends HttpServlet
{
 public void service(HttpServletRequest request,HttpServletResponse response)
  throws ServletException,IOException
  {

        PrintWriter out=response.getWriter();
        System.out.println("In service method of Servlet");
       try{
        ApplicationContext ctx=new ClassPathXmlApplicationContext("SpringHB.xml");
         demointer d1=(demointer)ctx.getBean("d1");
             Iterator i1=d1.getData();

         out.println("<body bgcolor=#ffffcc text=red>");
        out.println("<h1><center>all users</h1><hr><br><h3>");
       out.println("<table width=80% border=2>");
        while(i1.hasNext())
        {
             User u1=(User)i1.next();
             out.println("<tr><td>"+u1.getUid()+" <td>"+u1.getUname()+" <td>"+u1.getRole()+"</tr>");
        }
        out.println("</table>");
      }
         catch(Exception e)
        {
             out.println(e);
             e.printStackTrace();
        }
 }
}
==========
web.xml
---------
<web-app>
  <servlet>
      <servlet-name>select</servlet-name>
      <servlet-class>p1.SelectServlet</servlet-class>
   </servlet>

   <servlet-mapping>
      <servlet-name>select</servlet-name>
      <url-pattern>/selectaction</url-pattern>
   </servlet-mapping>

</web-app>
==========
```

In Spring 3.1 HibernateTemplate class has been deprecated and removed. So, inject Hibernate SessionFactory object to SpringBean while coorking 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)→ DB s/w (oracle, sybase and etc....)

→ Java appn —(Jndi api)→ Registry s/w (Web~~application~~ logic jboss registry and etc...)

→ Java appn —(Java mail api)→ Mail server s/w (James Mailserver, Exchange server, Lotus server and etc...)

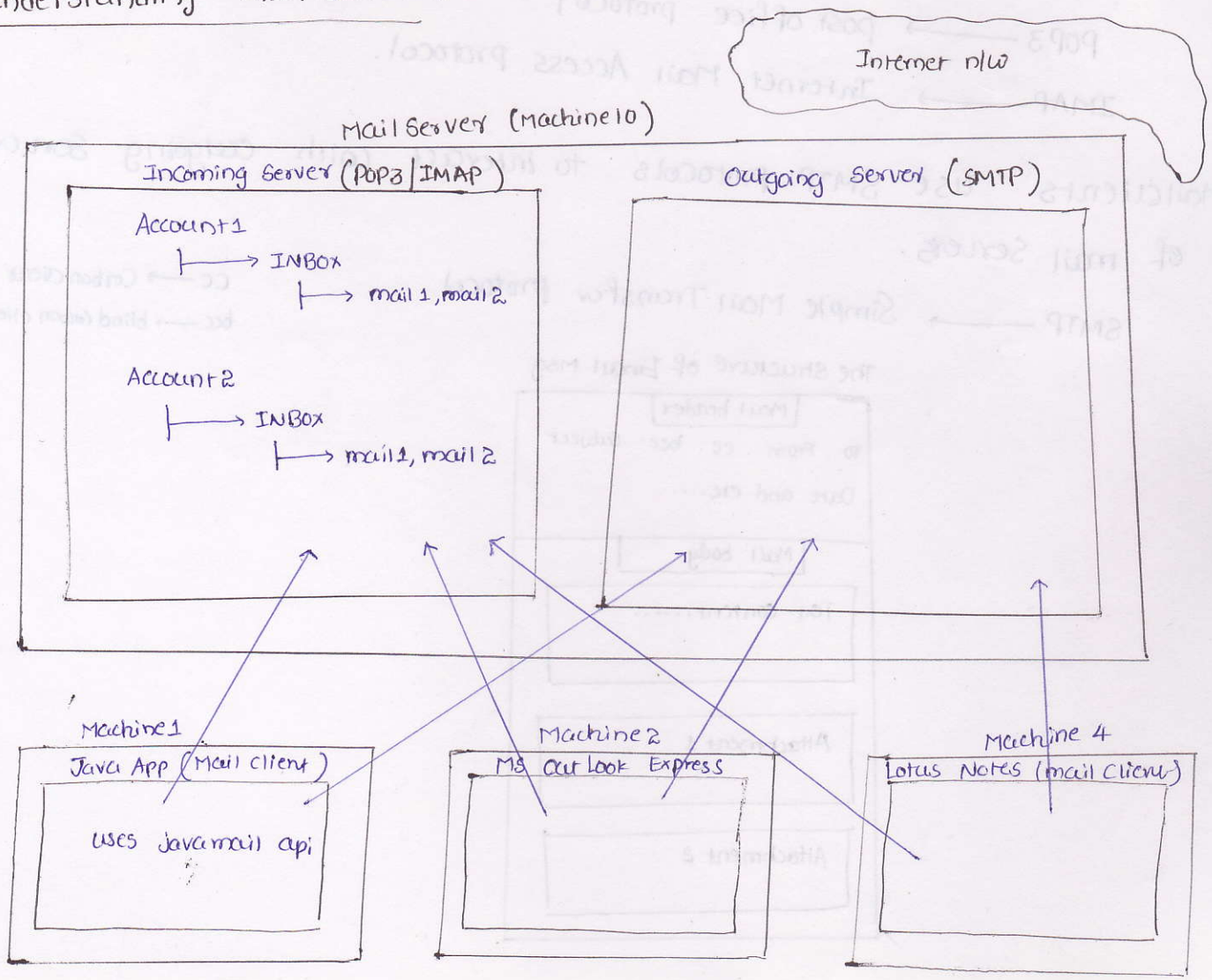
java mail api means javax.mail, javax.mail.internet pks (part of JEE module)

DB s/ws, Web server s/ws are not responsible to email accounts and email

→ they are too types mail clients

1) Ready Made mail clients (MS outlook, MS outlook express, Lotus notes)
2) Programable 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.

Mailclients 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 s/w.

To start the mail server : <James_home>\bin\ run.bat file.

Procedure to create email accounts manually in James mail server :-

Step 1 :- Star 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 (username) Srikanth (password) Srikanth

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 varying based on the mail server we use.

(*) for example application on Java mail API 60 & 61 page no's appn no:

→ In real webtevel 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

~~Incoming~~ port number of outgoing server : 465

Incoming Server name : pop.Gmail.com

Incoming Server port number : 995

for plain Java mail API based appns to interact with mail server of Gmail refer application ② to appn ④ of the Supplimentary Handout given on 26/08/2013

```
Java mail Apps
--App1----------SendAttachment.java----------
/*
  Java Application to send an E-mail with attachment
*/
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;
import java.util.*;
public class SendAttachment
{
    public static void main(String []args) throws Exception
  {
        //Creating Properties Objct
        Properties properties=new Properties();

        //adding protocol,mailserver address & the port number of the mailserver
        properties.put("mail.transport.protocol","smtp");
        properties.put("mail.smtp.host","localhost");
        properties.put("mail.smtp.port","25");

        //Creating the Session Object
        Session session=Session.getInstance(properties);

        //Creating and configuring the Message object
        Message message=new MimeMessage(session);

        message.setFrom(new InternetAddress("sachin1"));
    InternetAddress address[]={new InternetAddress("dhoni1")};
        message.setRecipients(Message.RecipientType.TO,address);
        message.setSentDate(new Date());
        message.setSubject("MAIL WITH ATTACHMENT");

        // mail body obj
        Multipart mailbody=new MimeMultipart();

        // mail body part1 obj (text content)
        MimeBodyPart part1=new MimeBodyPart();
        part1.setText("hello see my resume");
        mailbody.addBodyPart(part1);


    //mail body part2(attachment)
        FileDataSource fds=new FileDataSource(args[0]);//represents attachment file
        MimeBodyPart part2=new MimeBodyPart();
        part2.setDataHandler(new DataHandler(fds));
        part2.setFileName(fds.getName());
        mailbody.addBodyPart(part2);

        message.setContent(mailbody);

        //Sending the mail.
        Transport.send(message);
  }//main
}//class
//>javac SendAttachment.java
//>java SendAttachment one.txt
-------------------------------------------------------------------------------------------
// Prgs on plain java mail api (with Gmail Env.....)
--------App2----------SendMail.java----------
//Mail.java
import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;

public class  SendMail
 {
public SendMail() {
```

(using plain java mail API)

→ represents the empty body of the Email message

adding part1 to message body.

adding part2 to message body

email

→ Setting body to message content

add mail.jar file to the CLASSPATH.

78-74 Mandatory while working with real gmail server.


```
69  // mail properties outgoing server (gmail.com)
70        Properties props = new Properties();
71        props.put("mail.smtp.host", "smtp.gmail.com");
72        props.put("mail.smtp.port", "465");
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();
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");
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");
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");
106  }//main
107 }// Mail class
108 ----------App3--------------------GetMail.java----------
109 import java.util.*;
110 import javax.mail.*;
111
112 public class GetMail {
113
114 String host="pop.gmail.com";
115       String emailfrom="satyaemail123@gmail.com";
116       String password="javaj2ee";
117       String port="995";
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");
133
134             store.connect(host,emailfrom,password);
135
136             Folder folder = store.getFolder("inbox");
```

testzava@gmail.com
testjava

testzava@gmail.com
testjava

Object creation for innerclass

hostname
port number

inner class having authantication logic (when u talk with mail servers this logic is mucst)

add mail.jar in the CLASSPATH.

>javac SendMail.java
>java SendMail

Connecting to pop3 incoming server

Connects to email Id

```
137
138             folder.open(Folder.READ_ONLY);
139             Message[] msg = folder.getMessages();
140
141             System.out.println((msg.length-1)+" message found");
142             for (int i = 1; i <= msg.length; i++) {
143                 Message mess=folder.getMessage(i);
144                 mess.writeTo(System.out);
145             }//for
146     }//try
147     catch(Exception e)
148         {e.printStackTrace();}
149     }//constructor
150
151     private class SMTPAuthenticator extends javax.mail.Authenticator {
152             public PasswordAuthentication getPasswordAuthentication() {
153             return new PasswordAuthentication(emailfrom, password);
154         }
155     } //private class
156
157     public static void main(String args[])
158         {
159         new GetMail();
160         }//main
161     }//class
162     ---------App4-------------------DeleteMail.java-----------------------------------
163     import java.util.*;
164     import javax.mail.*;
165
166     public class GetMail {
167
168     String host="pop.gmail.com";
169         String emailfrom="satyaemail123@gmail.com";
170         String password="javaj2ee";
171         String port="995";
172
173     GetMail()
174             {
175         Properties props = new Properties();
176
177                 props.put("mail.pop3.host", host);
178                 props.put("mail.pop3.port", port);
179                 props.put("mail.pop3.auth", "true");
180                 props.put("mail.pop3.socketFactory.class", "javax.net.ssl.SSLSocketFactory");
181
182         try {
183                 Authenticator auth = new SMTPAuthenticator();
184                 Session session = Session.getInstance(props, auth);
185
186                  Store store = session.getStore("pop3s");
187
188                  store.connect(host,emailfrom,password);
189
190                  Folder folder = store.getFolder("inbox");
191
192                 folder.open(Folder.READ_ONLY);
193                 Message[] msg = folder.getMessages();
194
195                 System.out.println((msg.length+1)+" message found");
196                 for (int i = 1; i <= msg.length; i++) {
197                     Message mess=folder.getMessage(i);
198                     mess.writeTo(System.out);
199                 }//for
200     }//try
201     catch(Exception e)
202         {e.printStackTrace();}
203     }//constructor
204
```

Delete

WRITE

```
msg[0].setFlag(Flags.Flag.DELETED, true); //mark the msg for deletion
folder.close(true); //close the inbox with flag true
store.close();
}
```

```
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           {    Delete
213           new GetMail();
214           }//main
215   }//class
216   ----------------------------------------------------
217   Application sending sms.....
218   ------App5-------------------SendSMS.java----------------------
219   import java.io.*; → work with Iostream
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() {zavaboss
233          String username = "sathyaSMS"; //Your Credentials
234          String password = "sathya123"; 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[] argv) {
```

```
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.*;
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) {
296        this.mailSender = mailSender;
297     }
298
299         public void sendEmailMsg()
300         {
301        try{
302           mailSender.send(new MimeMessagePreparator() {
303
304              public void prepare(MimeMessage mimeMessage) throws MessagingException {
305
306              MimeMessageHelper message=new MimeMessageHelper(mimeMessage, true, "UTF-8");
307              message.setFrom("advani");
308              message.setTo("kalam");
309              message.setSubject("my subject");
310              message.setText("hello This mail Content/body");
311              message.addAttachment("abc.txt", new ClassPathResource("abc.txt"));
312           }
313        });
314        }//try
315        catch(MailException me) {
316          me.printStackTrace();
317        }
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     <bean id="ms" class="org.springframework.mail.javamail.JavaMailSenderImpl">
327        <property name="host"><value>localhost</value></property>
328        <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 can't Interface so, we use plane Java mail API

To inject JavaMailSenderImpl class object through setter Injection

Here send() is called having anonymous Inner class obj that implements MessagePreparator (Interface) and send() sends email msg by collecting message that comes when prepare() is called

sachin1

dhoni 1

to allow multiple parts in the message

represent bean property

refer line no:326

1) To run this application keep James mail Server is in Running mode.

2) Add following jar files in the CLASSPATH

spring.jar, mail.jar, commons-logging.jar

```
{
   public static void main(String args[])
    {
       ApplicationContext  ctx=new FileSystemXmlApplicationContext("mailcfg.xml");
       Student beanref=(Student)ctx.getBean("sti");
         beanref.sendEmailMsg();
       System.out.println("mail has been delivered");
       }
}
```

Spring Mail :-

telnet services in Windows 7

→ provides abstraction layer on plain java Mail API.

→ As of now Spring Mail 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 supplied direct facilities to perform mail operations (useful to send email messages

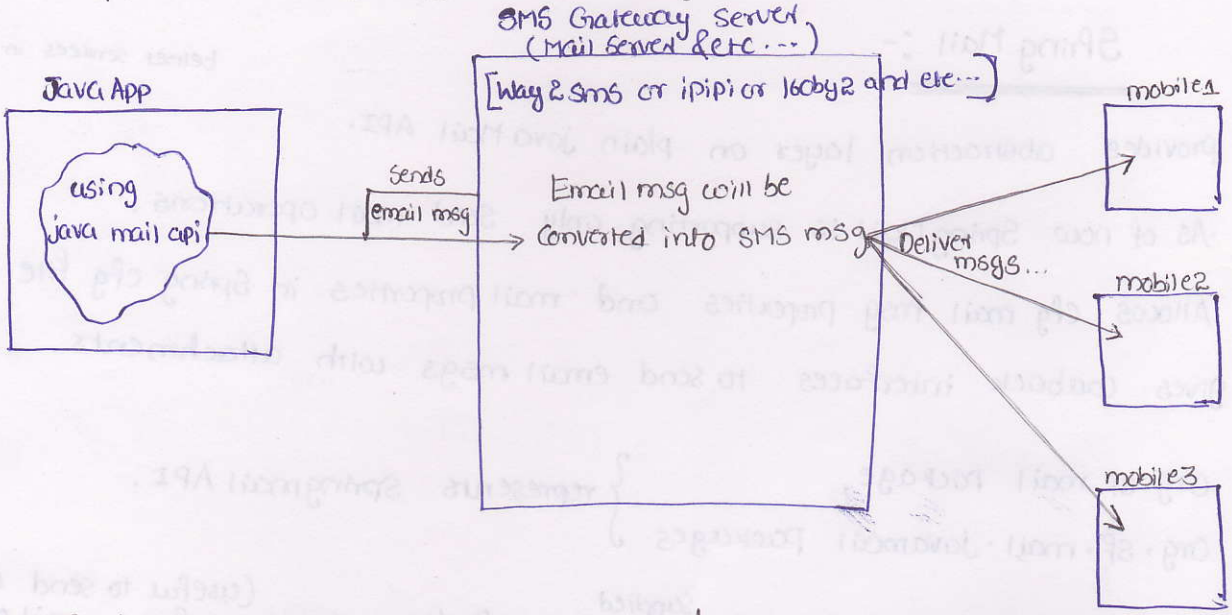
→ org.sf.mail.javamail represents Spring's abstraction Layer on plain java mail api (useful to send email messages with attachment)

(*) 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 concreate class called MimeMessageHelper

(*) For example application on Springmail based send mail operation with attachment refer appn (16) of the given handout given on 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 : zavaboss
Password : zavaboss1

Step(3):- Run the Send sms application (refer : supplimentory handout given on 26/03/13)

Procedure to access the webapplication of Tomcat Server with domain name.

Step(1):- specify domain name in built-in ~~reg~~ 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

Step(2):- change the Tomcat Server port number to "80"

If port number is not specified in Http request url 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.nataraz.in"  appBase="webapps"
      unpackWARs="true"  autoDeploy="true" >
    <Context path="" docBase="VoterApp" />
</Host>
```

(name="www.nataraz.in" → domain name; docBase="VoterApp" → webapplication)

Step 3:- Deployee the above specify VoterApp webappn in tomcatserver.
make sure that Welcomefile is configured in that webappn.

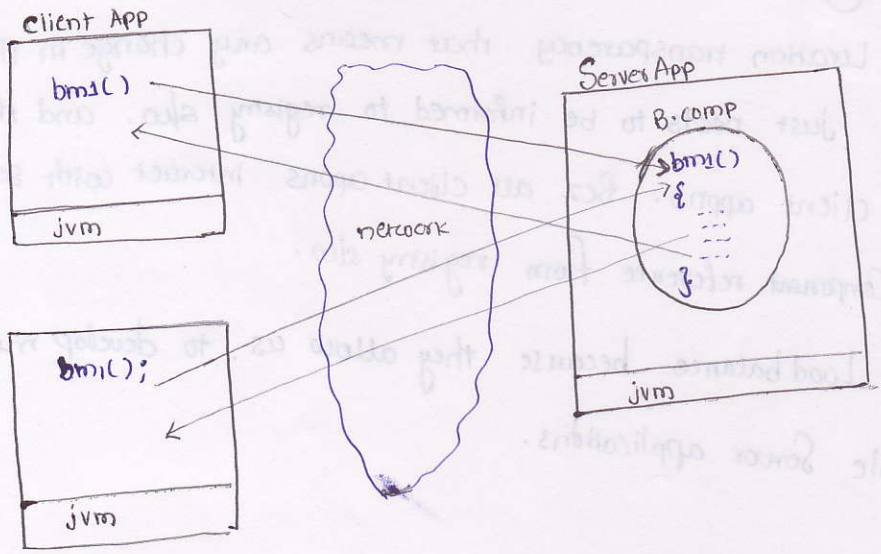
Step 4:- Restart the Server once.

Step 5:- Test the application

Open browser window and type this url http://www.nataraz.in/

Understanding the need of Distributed application Development :-

Traditional client-server appn

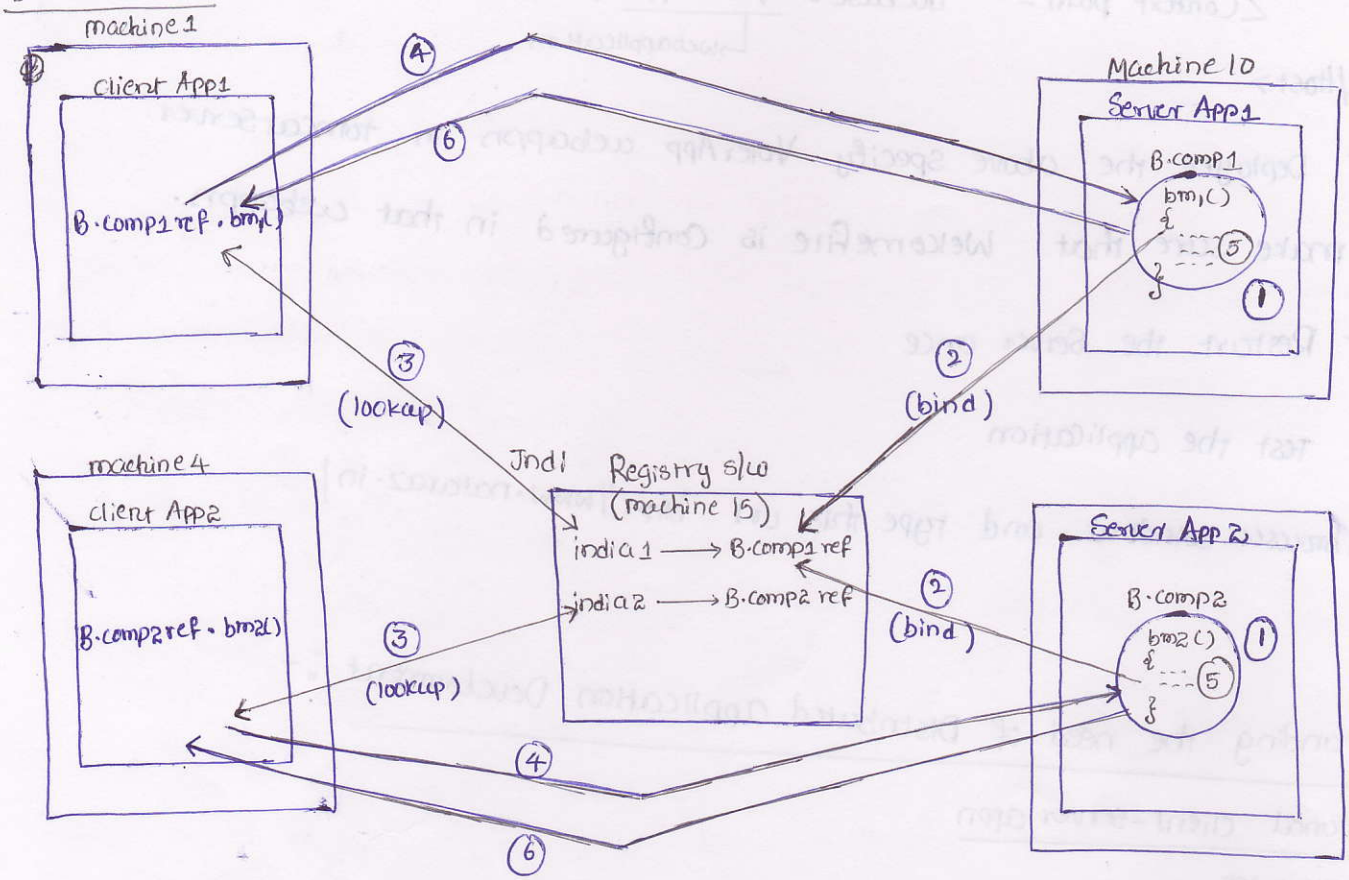


Traditional client-server applications (Socket programming Based) are location dependent that means any change in the location of Server application we must modify Code in the client appns. This indicates client and Server appns are tightly-Coupled appns(Parlswitch) here.

If degree of dependency is more b/w two appns then they are called tightly Coupled appns Other coise they are called loosely-coupled appns.

To develop the client-server Appns as Location transparent Apps then develope them as distributed applications.

Distributed Appn :-



Distributed appns support Location transparency that means any change in the Location of Server application just needs to be informed to registry s/w. and there is no need of change in coding client appns. Bcz all client appns interact with server appns by getting Business Component reference from registry s/w.

Distributed appns support Load balance because they allow us to develop multiple Business Components in multiple Server applications.

W.r.to the diagram :-

(1) multiple Server applications will be developed having multiple B.components.

(2) these multiple B.components references will be registered with registry s/w's for global visibility.

(3) Client appns gather B.component references from registry s/w

(4), (5), (6) Client appn calls B.methods of B.component and these B.methods of Server appn will executed & final result goes back to client appn.

To develop Distributed apps 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 apps) like Course brocacher understanding b/w student and faculty.
→ Declaration of b.methods

② Server Provider App
(The Server App that contains b.methods in B.component)

③ Registry s/w
(makes b.comp references as globally visible objs and also makes Distributed Apps as location transparent apps)

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 s/w

→ 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

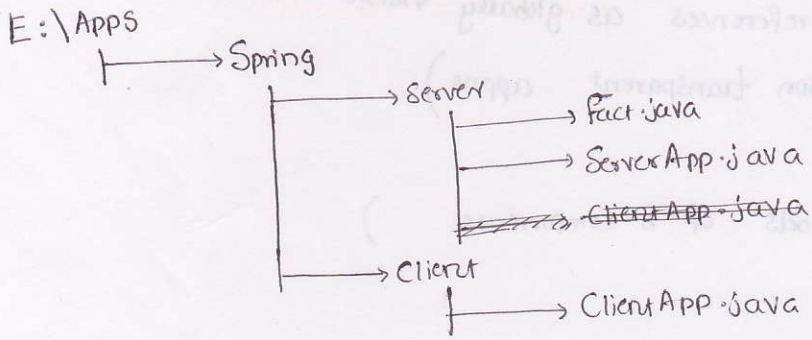
class implements java.rmi.Remote (I).

→ While developing distributed apps all the business objects must be developed as Remote object.

(*) 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.*;
//impl class that gives B.comp/obj
public class FactImpl extends UnicastRemoteObject implements Fact
{
  public FactImpl() throws RemoteException
  {
    S.o.p("FactImpl : 0-param Constructor");
  }
  //implement b.method
  public long findFactorial(int val)
  {
```

↳ 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
{
    p s v m(String args[]) throws Exception
    {
    FactImpl obj=new FactImpl(); //Business obj
      //bind B.obj with registry s/w
     Naming.bind("india", obj); // impossible also Naming.bind("rmi://localhost:1099/india", obj
      S.o.P("Server App is ready....");
    }
}

// ClientApp.java
import java.rmi.*;
public class ClientApp
{
   p s v m(Sring[] 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")
                                            ↳ it belongs to rmiapi
    // call B.methods
      S.o.p("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

E:\Apps\spring\server> rmic (Built-in JDK tool) FactImpl (→ class name of B.object)

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 <newport no>

Eg:- >rmiregistry 2000

→ The process of converting Java notation Data to Network notation data is called marshelling. 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.sf.remoting.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 B.obj that is registered with rmi registry is identified through two items ① nickname/alias name ② Service Interface name

→ allocos to keep business Object reference in rmi registry through dependency injection process and allocos to get B.obj ref from rmi registry through dependency injection process.

→ allocos to develop the resccources as POJI's & POJO's

→ allocos 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 lookoperation

(*) For example application on SpringRmi based Dismbuted appns refer pagenos: 62-6 app (15)

→ While developing plain RMI client appn we need to gather following details form 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 (rmi registry 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 require.

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 Dismbuted Components / Applications.

for example application on Springrmi using Spring 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 ORM, DAO modules while developing BusinessLogics.

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 lik EJB, HttpInvoker, Webservices and etc... EJB, HttpInvoker are language dependent. Webservices is Language independent.

→ SpringEJB provides abstraction Layer on plain EJB. but abstraction layer can no 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

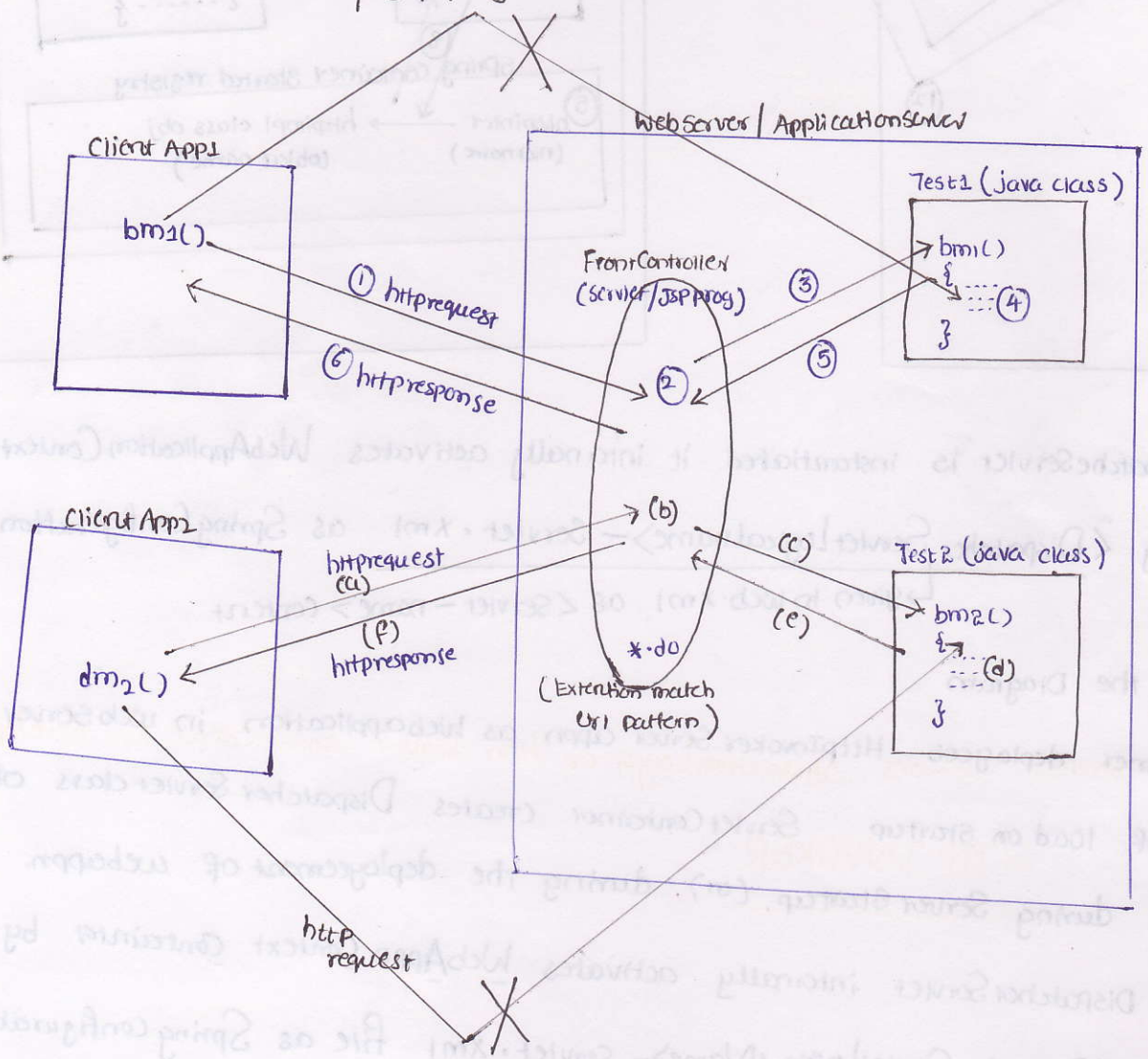
- It is spring's own web based distributed Technology
- Spring Container acts as Registry also
- Here Server appn is nothing but web application deployed in the Server
- Here client appn can be stand alone appn or webappn
- org.sf.remoting.httpinvoker.HttpInvokerServiceExporter keeps B.objs in the registry (spring Container) having Service interface name as nick name (or) alias name.
- org.sf.remoting.httpinvoker.HttpInvokerProxyFactoryBean 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 Http request coming to webappn from clients and to pass the requests to appropriate Java classes.

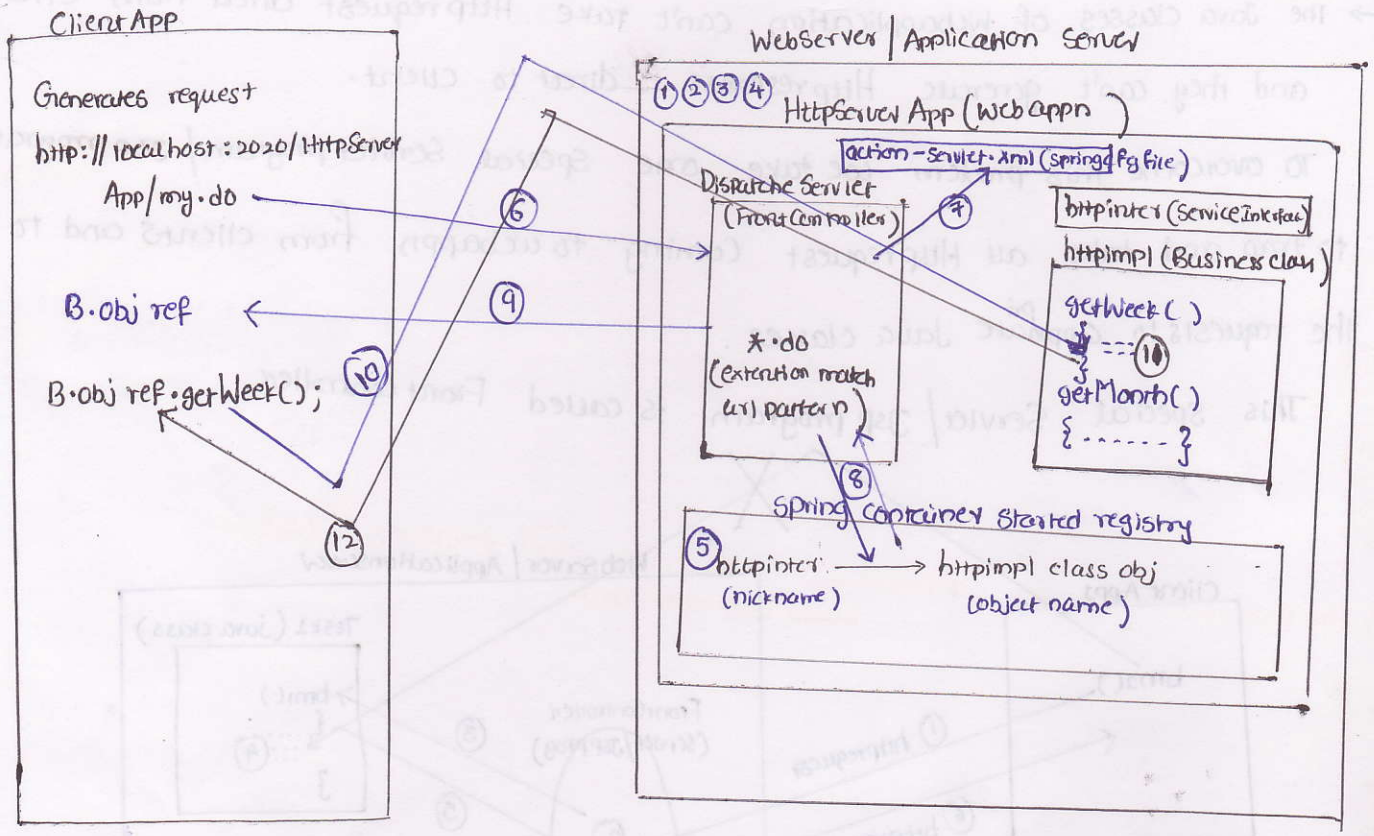
This special Servlet / Jsp program is called Front Controller.



→ FrontController is a special webresource program of a that acts as entry & exit point handling all the HttpRequest & HttpResponses 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.sf.web.servlet.DispatcherServlet is Front Controller Servlet.

→ a Front Controller Servlet must be configured in web.xml file having extention match or Directory match url pattern to trap and take multiple request.



When DispatcheServlet 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.to the Diagram

① programer deplogees HttpInvoker Server appn as Webapplication in web Server

② Bcz of load on Startup ServletContainer creates Dispatcher Servlet class object either during Server Startup (or) during the deployeement 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 Spring cfg file.

⑤ In the process of preInstantiation the HttpInvokerServiceExporter keeps Bussiness Class object in registry s/w having ServiceInterface name as nickname/ alias name.

⑥ Client appn uses some request Url and HttpInvokerProxyFactoryBean 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 SimpleUrlHandlerMapping class to bind the request with B.object.

(8) Based on the above activities the DispatcherServlet gathers B.object reference from Registry s/w.

(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) page no 65, 66&67
based Distributionappn

→ A Distributed application can be a Webappn or Non-webappn.

→ Webappn 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.

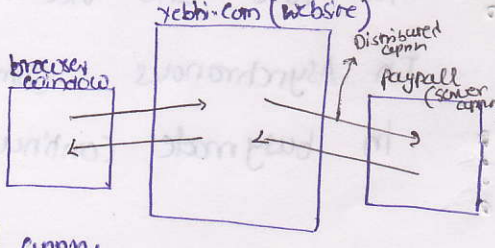
a) a copy of ServiceInterface / BusinessInterface

b) request url to get Business Object reference

c) Documentation about Business method.

Server appn developer can send this details to client App Developer 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 springbased Distributed appns. Spring RMI is not industry standard Distributed Technology to develop the Distributed appns.

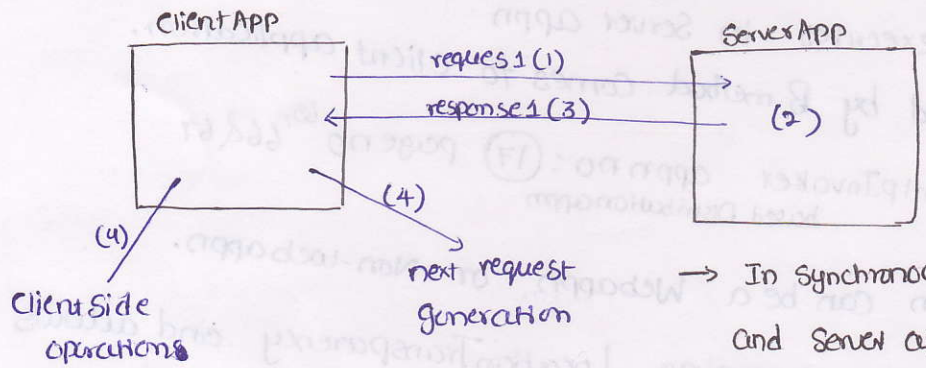


JMS (Java Message Service) :-

→ 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.

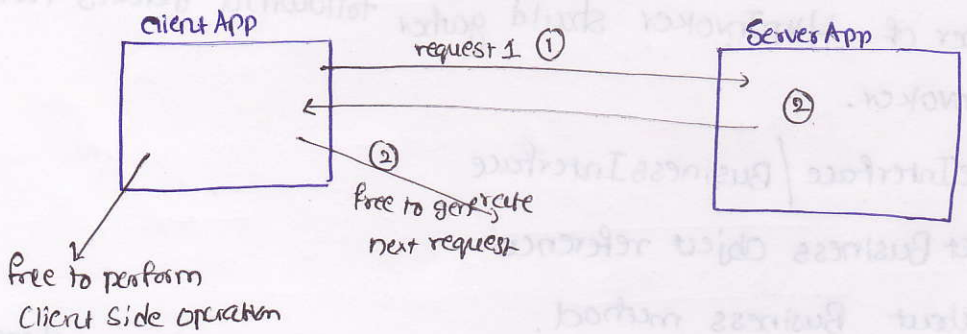
→ In Synchronous Communication Client appn and Server appns are tieghtly coupled appns

Synchronous Communication



→ In Synchronous Communication client and Server appns are teighetly coupled appns.

→ If client appn needs to perform client side operations and to generate next request coith out waiting for given request related response from Server appn then it is called "Asynchronous Communication."



→ In Asynchronous Communication Client and Server appns are loosely coupled appns.

→ In Synchronous Communication if client appn is busy the Server appns its idle and vice versa.

In Asynchronous Communication both Client and Server appns reside in busy mode continuously.

02/04/2013

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

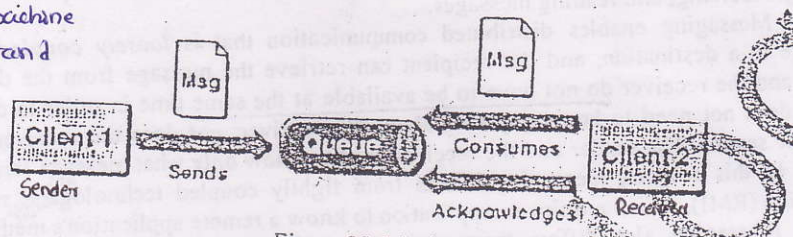


Figure 33-3 Point-to-Point Messaging

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. This makes the Queue destination to delete the messages

Use PTP messaging when every message you send must be processed successfully by one consumer.

Publish/Subscribe Messaging Domain

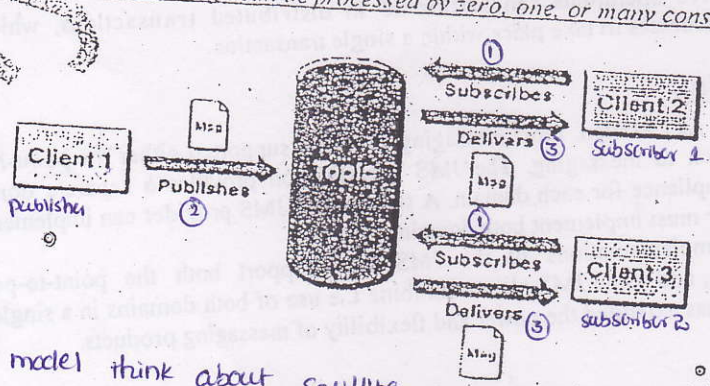
In a publish/subscribe (pub/sub) product or application, clients address messages to a *topic*, which functions somewhat like a bulletin board. Publishers and subscribers are generally anonymous and can dynamically publish or subscribe to the content hierarchy. The system takes care of distributing the 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.



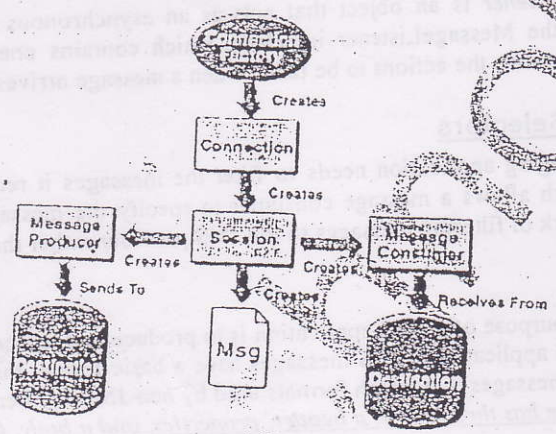
To understand pub/sub model think about satellite TV channel broadcasting System.

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.

```
----------Plain jms applications----------
(on Queue) (App1)
----------MyQueueSender.java (sender appn)----------
import javax.jms.*;
import javax.naming.*;
import java.util.Properties;

public class MyQueueSender
  {
  public static void main(String[] args) throws Exception
    {
    Properties prop=new Properties();
    prop.put(Context.INITIAL_CONTEXT_FACTORY,"weblogic.jndi.WLInitialContextFactory");
    prop.put(Context.PROVIDER_URL,"t3://localhost:7001");
    InitialContext ic=new InitialContext(prop);
    QueueConnectionFactory qcf=(QueueConnectionFactory)ic.lookup("ConFactJndi");
    QueueConnection qconn=qcf.createQueueConnection();
    QueueSession qsession=qconn.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);

    Queue queue=(Queue)ctx.lookup("QueueJndi");
    QueueSender qsender=qsession.createSender(queue);

    TextMessage message;
    String text;

    int k=1;

    for(int i=1;i<=4;i++)
      {
      if(i==1)
        text="Hotmail";
      else if(i==2)
        text="Amazon";
      else if(i==3)
        text="JGuru";
      else
        text="Bazee";

      for(int j=1;j<=5;j++)
        {
        message=qsession.createTextMessage();
        message.setStringProperty("Name",text);
        message.setText("This is Item "+k+"-------"+text);
        System.out.println("Sending "+k+"\t"+text);
        qsender.send(message);
        k++;
      }//for
    }//for

    qsession.close();
    qconn.close();
    }
}
>javac MyQueueSender.java
>java MyQueueSender
----------MyQueueReceiver.java----------
import javax.jms.*;
import javax.naming.*;
import java.util.Properties;

public class MyQueueReceiver implements MessageListener
  {
    public static void main(String args[]) throws Exception
      {
      Properties prop=new Properties();
      prop.put(Context.INITIAL_CONTEXT_FACTORY,"weblogic.jndi.T3InitialContextFactory");
      prop.put(Context.PROVIDER_URL,"t3://localhost:7001");

      InitialContext ic=new InitialContext(prop);
      QueueConnectionFactory qcf=(QueueConnectionFactory)ic.lookup("ConFactJndi");

      QueueConnection qconn=qcf.createQueueConnection();
```

(*1) Initial Context object pointing to GlassFish registry

(*2) represents connectivity blw sender appn and JMS server

(*1) "com.sun.enterprise.naming.SerialInitContextFactory"

iiop://localhost:4646

(*2) disables Transaction — JndiName given to JMS Connection Factory in GlassFish server

creates SessionObject as base to perform any JMS operations.

points to the Queue destination of GlassFish server

creates Sender pointing to Queue destination

25-50:- creates 20 text messages having 20 different text contents as bodies. but every 5 messages contains same value in name property.

To run plain Jms, Spring Jms appns

1. keep mydomain2 of server of GlassFish in running mode
2. Add Following jar files in the CLASSPATH
   (i) appserv-rt.jar
   (ii) appserv-ext.jar
   (iii) appserv-admin.jar
   (iv) appserv-ee.jar
   (v) javaee.jar
   (vi) jmsra.jar (AppServ/lib/install/applications/jmsra)

to receive messages from Queue destination based on EventHandling (receiving the messages Asynchronously)

(*1) change these Jndi properties to GlassFish properties

```
        QueueSession qsession=qconn.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);

        Queue queue=(Queue)ctx.lookup("QueueJndi");

        String selector="Name IN('Bazee','Hotmail')"; → MessageSelector to filter the messages
        QueueReceiver qreceiver=qsession.createReceiver(queue,selector);
                                  ↳ creating receiver pointing to Queue destination
        MyQueueReceiver mq=new MyQueueReceiver();
        qreceiver.setMessageListener(mq);
                  ↳ registering message Listener with our current class object
        qconn.start();
          ↳ starts listening to message store to receive the messages.
        }

        public void onMessage(Message msg)
                              ↳ represents the received msg each time
        {
          ↳ EventHandling method that executes when each message comes to message store
        try{
        TextMessage tmsg=(TextMessage)msg;
        System.out.println("Received Message is:"+tmsg.getText());
        }catch(Exception e)   S.o.p("message property" + tmsg.getStringProperty("Name"));
        {
          e.printStackTrace();
        }
        }
      }
```

? prints the body of the each received msg

(App2) On Topic

----------TopicDemo.java---------- (publisher appn)

```
import javax.naming.*;
import java.util.Properties;
import javax.jms.*; (JMS API)

public class TopicDemo
  {
  public static final String TOPIC_FACTORY ="TopicConFactJndi";
  public static final String TOPIC="TopicJndi";
  public static void main(String args[]) throws NamingException, JMSException
    {
    Properties prop=new Properties();          JNDI properties of GlassFish registry
    prop.put(Context.INITIAL_CONTEXT_FACTORY,"com.sun.enterprise.naming.SerialInitContextFactory");
    prop.put(Context.PROVIDER_URL,"iiop://localhost:4848");
    Context ctx=new InitialContext(prop);
    TopicDemo tt=new TopicDemo();→current class object
    tt.doPub(ctx); → User Defined method is called
    ctx.close();
    }

    public void doPub(Context ctx) throws JMSException, NamingException
    {
    Topic topic=(Topic)ctx.lookup(TOPIC);
                  ↳ get access to the Topic destination of GlassFish registry
(X2) TopicConnectionFactory tcf = (TopicConnectionFactory)ctx.lookup(TOPIC_FACTORY);
     TopicConnection tc = tcf.createTopicConnection();
       ↳ gets JMSConnection from ConnectionFactory
    TopicSession ts=tc.createTopicSession(false,TopicSession.AUTO_ACKNOWLEDGE);

    TopicPublisher tp=ts.createPublisher(topic); → Publisher is created pointing to Topic

    tc.start();

    TextMessage msg;
    msg=ts.createTextMessage("This is from Sathya Techs112..! ");
    System.out.println("Publishing msg on the Destination(Topic)");
                                                  } creating one Text msg.

    tp.publish(msg); → Sends one Text msg to the Topic destination
    tc.close();
    }
  }
```

(X1)

----------SubscriberDemo.java----------

```
import javax.naming.*;
import java.util.Properties;
import javax.jms.*;
public class SubscriberDemo{
  public static void main(String args[]) throws NamingException, JMSException {
    Properties prop=new Properties();
```

Jar files in classpath same as previous appn (1)

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                                                          4646
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 JMSException, NamingException{
161
162         Topic topic=(Topic)ctx.lookup("TopicJndi");
163                              -> getting access to the Topic destination of GlassFish registry
164         TopicConnectionFactory tcf = (TopicConnectionFactory)ctx.lookup("TConFactJndi");
165         TopicConnection tc = tcf.createTopicConnection();
166                              -> gets JMS Connection from Connection Factory
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 the msg from Topic destination
174
175         System.out.println("Received message is:"+msg.getText());
176         tc.close();                                  -> Displays the content of the msgs.
177         }
178   }
179 ---------------------------------------------------------------------------------------------------
180 Applications on Spring JMS
181 (App4)( on queue)
182 ---------------------------------------------------------------------------------------------------
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.*;   (*1) 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   {
195     this.jt = jt;
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 Creating the Text msg <-  message = session.createTextMessage();
210                           message.setStringProperty("text", "Hello World");
211         } message.setText("from sathyatech 112");
212         catch (JMSException e)
213         {
214           e.printStackTrace();
215         }
216         return message;
217       }
218     };      202-218 :- 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"?>
                                                    JMSTemplate class object.
```

Jar files in classpath
spring jar files : (2)
Same 6 Jar files of App(1) in handcart
GlassFish server should be in running mode

Sender
  -> *.java (2)
  -> Spring.xml

Sender > javac *.java
Sender > java SendClient

(x1) jndi properties of glassfish registry.

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
 <bean id="cf" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiEnvironment">
   <props>
     <prop key="java.naming.factory.initial">com.sun.enterprise.naming.SerialInitContextFactory</prop>
     <prop key="PROVIDER_URL">iiop://localhost:4848</prop>
   </props>
  </property>
  <property name="jndiName" value="ConFactJndi"/>
 </bean>

 <bean id="q" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiEnvironment">
   <props>
     <prop key="java.naming.factory.initial">com.sun.enterprise.naming.SerialInitContextFactory</prop>
     <prop key="PROVIDER_URL">iiop://localhost:4848</prop>
   </props>
  </property>
  <property name="jndiName" value="QueueJndi"/>
 </bean>

 <bean id="template" class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="cf"/>
  <property name="defaultDestination" ref="q"/>
 </bean>

 <bean id="ms" class="MessageSender">
  <property name="jt" ref="template"/>
 </bean>
</beans>
```

(x2) gathers ConnectionFactoryObject from glassFish registry

(x3) gives Queue obj poining to Queuedestination of GlassFish server.

refer line: 229

refer line: 240

our springBean configuration having the logic to inject JMSTemplate class object.

refer line no: 251

---SendClient.java (client appn)---

```
import org.springframework.context.support.*;
public class sendclient
{
   public static void main(String args[])
   {
      FileSystemXmlApplicationContet ctx=new FileSystemXmlApplicationContext("spring.xml");
      MessageSender ms =(MessageSender)factory.getBean("ms");
      ms.sendMessage();
      System.out.println("message sent");
   }

}
```

Receiver App

---MessageReceiver.java (SpringBean)---

```
import javax.jms.*;
import org.springframework.jms.core.*;

public class MessageReceiver {
  private JmsTemplate jt;


  public MessageReceiver() {}

  public void setJmsTemplate(JmsTemplate jt)
  {
    this.jt = jt;
  }



  public void receiveMessage()
  {
    Message message = jt.receive();
    TextMessage tmsg = null;
    if (message instanceof TextMessage)
    {
      tmsg=(TextMessage)message;
```

receives the msg from Queue Destination

Jarfiles in class path :- same as SenderApp.

Receiver
- *.java(2)
- spring.xml

Receiver> javac *.java
> java ReceiveClient

```
        try
        {
           System.out.println(tmsg.getText());
           System.out.println(tmsg.getStringProperty("text"));
        }
        catch (JMSException e)
        {
           e.printStackTrace();
        }
     }//if
   }//method
} //class
----------------------------spring.xml----------------------------------
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>

 <bean id="cf"  class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiEnvironment">
   <props>
        <prop key="java.naming.factory.initial">com.sun.enterprise.naming.SerialInitContextFactory</prop>
         <prop key="PROVIDER_URL">iiop://localhost:4848</prop>
   </props>          java.naming.provider.url          4646
  </property>
  <property name="jndiName" value="ConFactJndi"/>
 </bean>

 <bean id="q"  class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiEnvironment">
   <props>
        <prop key="java.naming.factory.initial">com.sun.enterprise.naming.SerialInitContextFactory</prop>
     <prop key="PROVIDER_URL">iiop://localhost:4848</prop>
   </props>          java.naming.provider.url          4646
  </property>
  <property name="jndiName" value="QueueJndi"/>
 </bean>

 <bean id="template" class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="cf"/>
  <property name="defaultDestination"  ref="q"/>
 </bean>

 <bean id="ms"  class="MessageReceiver">
  <property name="jt"  ref="template"/>
 </bean>
</beans>
-----------------------------------------------------ReceiveClient.java-----------------------------
import org.springframework.context.support.*;
public class ReceiveClient
{
   public static void main(String args[])
   {
      FileSystemXmlApplicationContet ctx=new FileSystemXmlApplicationContext("spring.xml");
      MessageReceiver mr =(MessageReceiver)factory.getBean("ms");
      mr.receiveMessage();                                  ctx
   }

}
-------------------------------------------------------------------------------------------------------------------
App5 (on Topic)
Sender App (publisher appn)
-------------------------------------------------------------------------------------------
-----------------------------------------MessageSender.java--(SpringBean)-----------
import javax.jms.*;
import org.springframework.jms.core.*;
public class MessageSender {

   private JmsTemplate jt

   public MessageSender() {}

   public void setJt(JmsTemplate jt)
   {
      this.jt = jt;
   }
```

```
376
377     public void sendMessage()
378     {
379        MessageCreator creator = new MessageCreator()
380        {
381           public Message createMessage(Session session)
382  Anonymous{
383  Inner class  TextMessage message = null;
384  Object       try
385  implementing {
386  MessageCreator  message = session.createTextMessage();
387                  message.setStringProperty("text", "Hello World");
388  Interface       message.setText("this is test msg");
389
390              }
391              catch (JMSException e)
392              {
393                 e.printStackTrace();
394              }
395              return message;
396        }
397     };
398
399     jt.send(creator); ---> Sends the message to Topic destination.
400     }
401  }
402
403  ----------------------------------------------spring.xml (springcfg file)----------------------------------------------
404  <?xml version="1.0" encoding="UTF-8"?>                    408-416 -> gives JMSConnectionFactory
405  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"          Object by gathering it from Jndi registry
406  "http://www.springframework.org/dtd/spring-beans.dtd">
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">iiop://localhost:4848</prop>
413     </props>      java.naming.provider.url      4646
414    </property>
415    <property name="jndiName" value="ConFactJndi"/>
416   </bean>
417
418                                                          419-427 gets Topic from
419   <bean id="q" class="org.springframework.jndi.JndiObjectFactoryBean">          Jndiregistry
420    <property name="jndiEnvironment">
421     <props>
422        <prop key="java.naming.factory.initial">com.sun.enterprise.naming.SerialInitContextFactory</prop>
423       <prop key="PROVIDER_URL">iiop://localhost:4848</prop>
424     </props>      java.naming.provider.url      4646
425    </property>
426    <property name="jndiName" value="TopicJndi"/>
427   </bean>
428
429
430   <bean id="template" class="org.springframework.jms.core.JmsTemplate">
431    <property name="connectionFactory" ref="cf"/>  -> refer: 408
432    <property name="defaultDestination" ref="q"/>  -> refer: 419
433    </bean>
434
435   our SpringBean injected with JmsTemplate class Object
436   <bean id="ms" class="MessageSender">
437    <property name="jt" ref="template"/>  -> refer: 430
438   </bean>
439  </beans>
440  ----------------------------------------------SendClient.java----------------------------------------------
441  import org.springframework.context.support.*;
442  public class sendclient
443  {
444     public static void main(String args[])
445     {
446        FileSystemXmlApplicationContet ctx=new FileSystemXmlApplicationContext("spring.xml");
447        MessageSender ms =(MessageSender)factory.getBean("ms");
448        ms.sendMessage();                    ctx
449        System.out.println("message sent");
450     }
```

Sender
- *.java(2)
- spring.xml

sender > javac *.java
> java SendClient

Jar files in CLASSPATH: same as previous apps.

```
}
-----------------------------------------------------------------------------
Receiver App (subscriber)
-----------------------------------------------------------------------------
----------------------------------MessageReceiver.java-----------------------------
import javax.jms.*;
import org.springframework.jms.core.*;

public class MessageReceiver {
   private JmsTemplate jt;


   public MessageReceiver() {}

   public void setJmsTemplate(JmsTemplate jt)
   {
      this.jt = jt;
   }


   public void receiveMessage()
   {
      Message message = jt.receive();
      TextMessage tmsg = null;
      if (message instanceof TextMessage)
      {
         tmsg=(TextMessage)message;
         try
         {
            System.out.println(tmsg.getText());
            System.out.println(tmsg.getStringProperty("text"));
         }
         catch (JMSException e)
         {
            e.printStackTrace();
         }
      }
   }
}
----------------------------------spring.xml----------------------------------------
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>

 <bean id="cf" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiEnvironment">
    <props>
        <prop key="java.naming.factory.initial">com.sun.enterprise.naming.SerialInitContextFactory</prop>
         <prop key="PROVIDER_URL">iiop://localhost:4848</prop>
    </props>
  </property>
  <property name="jndiName" value="ConFactJndi"/>
 </bean>

 <bean id="q" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiEnvironment">
    <props>
        <prop key="java.naming.factory.initial">com.sun.enterprise.naming.SerialInitContextFactory</prop>
      <prop key="PROVIDER_URL">iiop://localhost:4848</prop>
    </props>
  </property>
  <property name="jndiName" value="QueueJndi"/>
 </bean>

  <bean id="template" class="org.springframework.jms.core.JmsTemplate">
   <property name="connectionFactory" ref="cf"/>
   <property name="defaultDestination" ref="q"/>
 </bean>

 <bean id="ms" class="MessageReceiver">
  <property name="jt" ref="template"/>
 </bean>
</beans>
```

Receiver
- *.java (2)
- spring.xml

>javac *.java
>java ReceiveClient

First start multiple Receiver / subscriber apps then start only one sender appn (Publisher appn)

java.naming.provider.url 4646

4646

TopicJndi

refer : 497

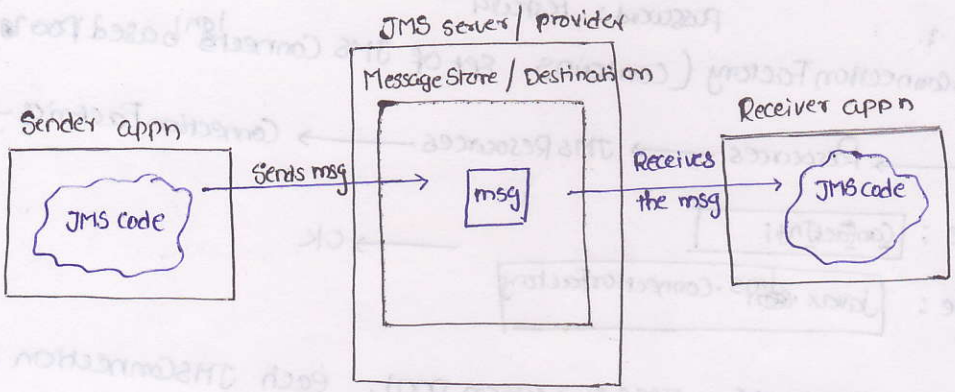
refer : 507

refer : 517

```
526  ----------------------------------------ReceiveClient.java----------------------------------
527  import org.springframework.context.support.*;
528  public class ReceiveClient
529  {
530     public static void main(String args[])
531     {
532        FileSystemXmlApplicationContet ctx=new FileSystemXmlApplicationContext("spring.xml");
533        MessageReceiver mr =(MessageReceiver)factory.getBean("ms");
534        mr.receiveMessage();                                    ctx
535     }
536
537  }
538  ------------------------------------------------------------------------------------------------
```

→ To acheive asynchronous communication blw browser window and webappn use either AJax (clientSide Technology) or JavaPortlets (ServerSide Technology)

→ To acheive messages based asynchronous Communication between two standalone Java appns 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 messages temporarly Eg:- JMS Server / providers
MQseries 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 supplimentary 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 JMSConnectionFactory (contains set of JMS Connects based pools)

Admin Console Screen → Resources → JMS Resources → Connection Factories →

New → JNDIName : ConFactJndi → OK

Resource Type : javax.jms.ConnectionFactory

JMSConnectionFactory represents JMS Connection pool. each JMSConnection Object represents the Connection b/w client appn and JMS server.

Step(4):- Create Queue destination having JNDI name

Admin Console Screen → JMSResources → Destination Resources → New →

JNDIName : QueueJndi

Physical Destination Name : ConFactJndi → OK

Resource Type : javax.jms.Queue

(*) procedure to create Topic Destination in the above environment

Admin Console Screen → Resources → JMS Resources → Destination Resources

→ New → JNDIName : TopicJndi → physical Destination Name :

ConFactJndi → Resource Type : javax.jms.Topic → OK.

→ 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 → javeee.jar

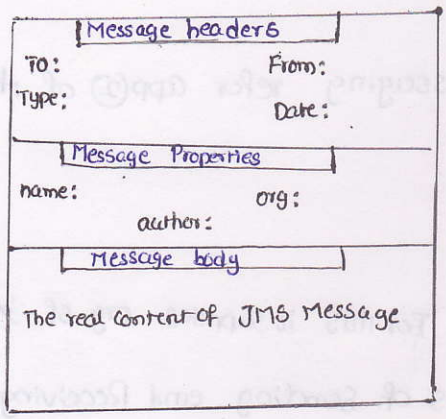
in Jboss Server → Jboss-javaee.jar

Sender/publisher App uses jms api to send msgs to the Message Store

Receiver 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)

TopicPublisher
TopicSubscriber
and etc....

related to Messages

TestMessage
StreamMessage
MapMessage
ObjectMessage
and etc....

*) For plain Jms API based Sender, receiver application dealing with Queue destination based PTOP Messaging Domain refer appn ① of the 02/04/2013 handout

→ The realtime example applications of JMS are

① Compaint management

② Bugs management

③ OFFline order booking and etc..

(*) For plain Jms based pub|sub model of messaging refer app② of the pageno's ⑥&⑦ of 02/02/13

Spring JMS :-

→ provides abstraction Layer of plain JMS. For this it supplies org.sf.jms.core.JmsTemplate class.

→ This JMSTemplate class Simplifies the process of Sending and Receiving msgs.

→ No need of working coith plainJMS API for JMS operations

→ Gives Callback Interfaces to ~~develop~~ create diff types of messceges.

→ To create JMSTemplate class obj JMSConnector obj, Topic|Queue destination objs are required.

→ Spring JMS gives the Callback Interface called messageCreator to create diff types of messageCreators by using the exposed JMSSession object.

(*) for Spring JMS based P-to-P example application refer appn④ of the page no's ⑦ to ⑨

(*) For example appn on Spring JMS based pub-sub messaging domain appn⑤ of the page no's ⑨ to ⑫

Module-V

Spring Web Module

Spring Web module
- Part1 (Gives plugins and facilities to make Spring appns communicatable from other coeb flw slcos based appns like Struts appns, JSF appn etc..)
- Part2 (Spring Web mvc / Spring mvc / Spring web flow)
  - It is spring's own coeb flw slw to develop mvc2 architecture based webappns.

Struts and Spring Integration :-

→ In this appn Struts appn represents View layer (presentation logic) and Controller layer (Integration logic) logics.
and Spring appn represents Model Layer (Business Logic & Persistence logic) logics.

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 compute
→ 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.sf.web.struts.ContextLoaderPlugIn and this plugin is supplied by Spring slw 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 deployement of webappn. because we generally enable load-on-startup on ActionServlet.

→ The above ContextLoaderPlugIn activates Webappn context Spring Container by taking specified file name (or) ActionServletlogicalname-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 <ActionServletLogicname>-servlet.xml

In struts cfg file

<plug-in className="org.sf.web.struts.ContextLoaderPlugIn">

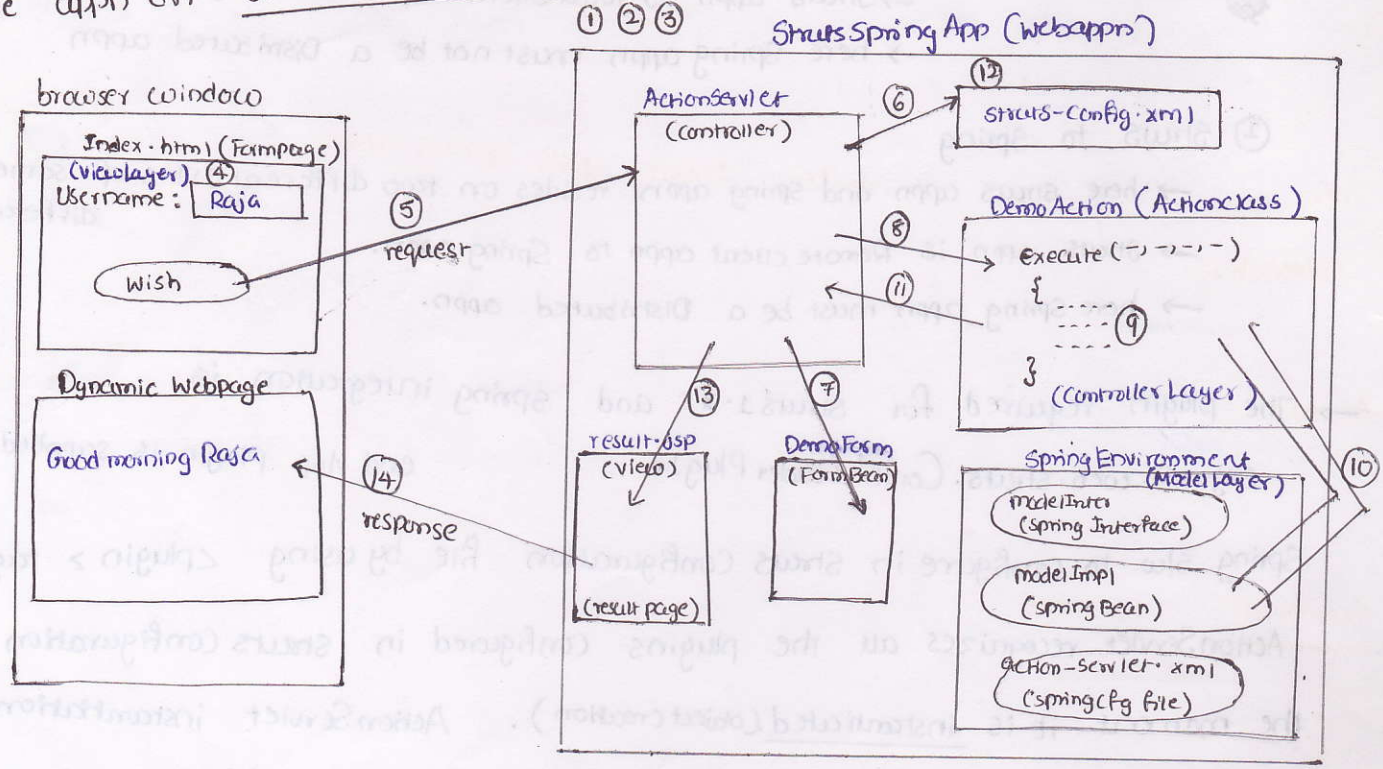
<set-property property="contextConfigLocation" value="/WEB-INF/mycf1.xml/WEB-INF/mycfg2.xml" />

</plug-in>

→ here the activated spring container takes mycfg1.xml, mycfg2.xml files as SpringConfiguration files.

→ In struts with spring appn development the Struts Action class should contain Client code and should call the B.methods of SpringBean available in Spring appn.

Example appn on "Struts with Spring"



W.r.to the diagram

(1) programer deployees struts spring application in webserver or ApplicationServer.

(2) because of Load on Startup servlet container creates ActionServlet class obj either during Server Startup (or) during the deployeement of webappn.

(3) ActionServlet activates ContextLoadPlugIn. this plugin activates the webApplicationContext Containing by taking <ActionServletLogicalname>-servlet.xml as Spring Configuration file.

In this process all the beans of Spring Configuration file will be pre-instantiated

④&⑤ enduser 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 copies 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 objec

⑩ execute(-,-,-,-) method calls Business method of SpringBean 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 Actio cla

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 SpringBean class object to Struts Action class property (for this we need to cfg Struts Action class as SpringBean 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 ContextLoaderPlugIn activated SpringContainer. and we can use this SpringContainer to get access to SpringBean class object and to call B. methods on it.

→ in Struts and Spring integration always prefer working with ContextLoaderPlugIn related 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 deployement 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 (i) :-

```
public class TestAction extends ActionSupport
{                                   (This internally extends from Action class of
                                     Struts 1.x api)
 public ActionForward execute(-,-,-,-)
 {
   // get access to the Spring Container activated by ContextLoaderPlugin
  WebApplicationContext ctx = getWebApplicationContext();
   // get access to Spring Bean class Obj
   modelInter bobj = (modelInter) ctx.getBean("mt");
                                              └→ beanid
   // call the B.method of SpringBean
    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 development related documentation refer page no's (26) to (34) of the material.

Approach (2) Discussions :-

→ here we Configured Struts Action class as SpringBean class in SpringConfiguration file. so that we can inject the model Layer SpringBean 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 ActionServlet and to pass the request to original Struts Action class that is Configured in Spring Configuration file.

→ Spring API supplies org.sf.web.struts.DeligatingActionProxy class for this purpose

→ In Struts Configuration file

```
<action path="/xyz" type="org.sf.web.struts.DelegatingActionProxy">
    ------
    ------
</action>
```

↳ Proxy class for original Action class

In Spring Configuration file

```
<beans>
  <bean name="/xyz" class="TestAction">
    <property name="bobj" ref="d1" />
  </bean>
  <bean id="d1" class="ModelImpl" />
</beans>
```

↳ originca Action class

↳ must match with above "xyz" (actionPath)

↳ logic to inject SpringBean class object to a property of Struts Action class

↳ 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 SpringBean class object to Struts Action class property either during Server Startup or during the deployment of the webappn.

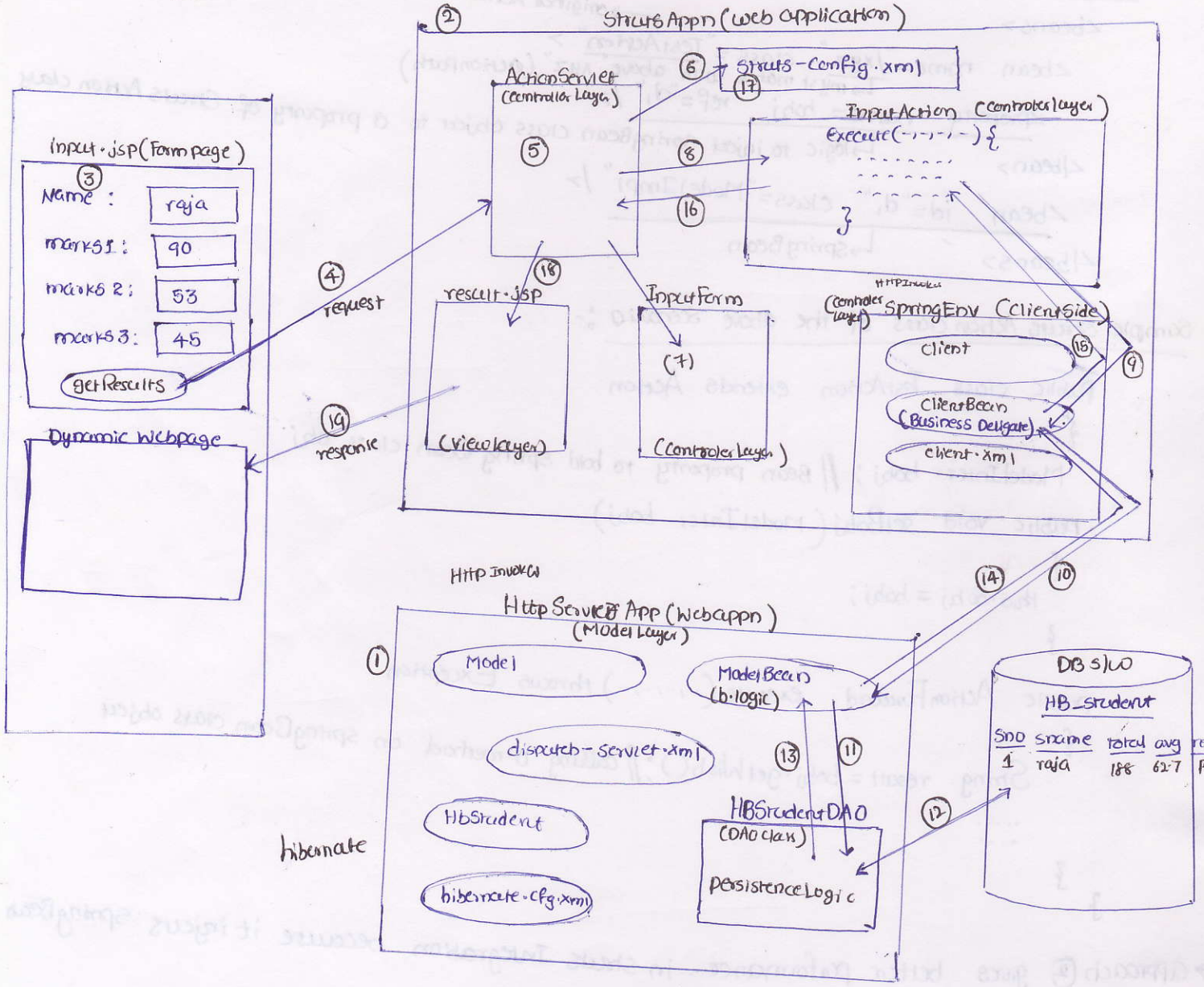
for approach no ② based Struts with Spring appn development refer appn no: (22)

page no: 73 to 75

*) Miniproject showing Struts, Spring, hibernate Integration and also implementing mvc2, DAO, Bussiness Delegate Design patterns.

→ The helper java class of client appn that separates Business method calls to make them as flexibile to modify is called Business Delegate class.

Struts to spring with Hibernate appn



In the above diagram server appn is HttpInvoker based 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 are there (1) Struts Application
(2) The HttpInvokerSever 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 miniproject by using MyEclipse IDE

for server cupplication:-

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 DBS/w.

Create table HB_Student (sno number(4) primary key, sname varchar2(20), total number(6), Avg number(8,2), result varchar2(5));

Step-IV:- Create DataBase profile for oracle in MyEclipse IDE

Windows → open perspective → My Eclipse DataBase 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

addJars: ojdbc14.jar

→ next → finish

Step-V:- Add Hibernate Capabilities to the project

Right click on project → myEclipse → Add Hibernate Capbilities → ⊙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 → myEclipse DBExplorer → DB Browser →

→ Right click on orap → openConnection → passcoord: tiger → Ok

→ Expand orap → Expand Connected to orap → Expand scott → Expand table

Right click on Hb_student → Hibernate Reverse Engineering →

java src folder | /HttpServerApp/src |

☑ create POJO <-> DB table ⟶ ☑ Java DataAccess Object (DAO)

⊙ Add Hibernate mapping Annotations ☑ Generate precise find By methods.

☑ update Hibernate configuration DAO Type : ⊙ SpringDAO

⟶ Next ⟶ ⊙ Hibernate types ⟶ IDGenerator : | increament | ⟶

Hb_student ⟶ IDGenerator : | increment | ⟶ Finish.

Step-VI :- Add spring Capabilities to the project

Rightclick on project ⟶ myEclipse ⟶ Add Spring Capabilities ⟶ ☑ spring 2.5

⟶ Select Core, persistence Core, JDBC, J2EE, Remoting, Misc Libraries ⟶ next

⟶ ☐ Enable Aop Builder ⟶ File: | dispatcher-servlet.xml | ⟶ next ⟶ ☑ Add Hibernate 3.2 annotation support

SessionFactory Bean id : | sesfact | ⟶ finish.

Step-VII :- This step generates Hbstudent.java ⟶ HbStudentDAO.java files

↳ POJO class ↳ DAO class

Step-VIII :- Add SpringInterface, SpringBean 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-IX Src ⟶ new ⟶ class : | ModelBean | implements model

```
public class ModelBean implements model {
        HbStudentDAO dao;
Public void setDao (HbStudentDAO dao) {
        this.dao = dao;
    }
public String generateResult (String name, int m1, int m2, int m3) {
    //Write b.logic
    int total = m1+m2+m3;
    float avg = (float) total/3.0f;
    // generate result
    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;
} //method
} //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.SimpleUrlHandlerMapp
  <property name="mappings">
  <props>
    <prop key="my.do">/service</prop>
  </props>
  </property>
</bean>
</beans>
```

SimpleUrlHandlerMapping

Configure Tomcat Server with myEclipse IDE

Step-II :- Run the project

Right click on project → Run as → MyEclipse Service Application → Tomcat 6.0 → OK

Changes:-

Change1 :- Copy dispatcher-servlet.xml file to WEB-INF folder of Webroot

Change2 :- Add Spring_home\lib\asm\asm-2.2.3.jar to the Build Path of the project

Change3 :- If ClassVisitor problem is raised

a) remove Hibernate, Spring Libraries from the Build path of the project ~~related to~~ 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 Reguler 8 hibernate jar files

(hibernate 3.jar
(JTA-1.1.jar
(javassist-3.4.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 → project Name : StrutsApp → Finish

Step-II :- add spring capabilities to the project.

Right click on project → myEclipse → Add spring Capabilities → ⊙spring2.5 → 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 (client), Spring Bean (clientBean) having Logic to call the Business methods of Server appn.

~~File~~ src → New → Interface → Name : client → finish

```
public interface client {
    public String callBmethod(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.httpinvoker.HttpInvokerProxyFactoryBean">
    <property name="serviceInterface" value="Model" />
    <property name="serviceUrl" value="http://localhost:2020/HttpServerApp/my.do" />
</bean>

<bean id="cb" class="ClientBean">
    <property name="bobj" ref="pfb" />
</bean>

<bean name="/demo" class="InputAction">
    <property name="mo" ref="cb" />
</bean>
</beans>
```

NOTE:-

Step-IV:- add Struts Capabilities to the project

Right click on project → myEclipse → Add Struts Capabilities → ⊙ Struts 1.3

→ Base packages for new classes [ ] → finish.

Step-V:- add Form page (↳ input.jsp), FormBean class (↳ InputForm), Action class (↳ InputAction) to project.

R.C. on project → new → Others → myEclipse → Web Struts → Struts 1.3

→ Struts 1.3 Form, Action & jsp → next → Name [if1] (↳ FormBean Logic name) →

Super class: [org.apache.struts.action.ActionForm] → Form Type: [InputForm] →

Form Properties [name, m1, m2, m3] [add] → methods ☐ ☐ deselect all the check boxes

→ Jsp → ☑ Create Jsp form ? → next → Path : /demo (action Path)

New jsp path : /input.jsp

Type : InputAction → tab Form : if1 → forwards tab → add → Name : success, Path : /result.jsp

→ add → close. → Finish.

Step-VI :- Do following modifications in struts configuration file.

① remove existing Action class configuration and Configure DeligatingActionProxy

(Under <action-mappings> tag)

```
<action-mappings>
    <action path="/demo" type="org.sf.cocb.struts.DeligatingActionProxy"
        name=if1 >
        <forward name="success" path="/result.jsp" />
    </action>
</action-mappings>
```

② Configure the ContextLoaderPlugIn as shown below.

(under </action-mappings> tag)

```
<message-resources parameter="ApplicationResources" />
<plug-in className="org.sf.cocb.struts.ContextLoaderPlugIn" >
    <set-property property="contextConfigLocation" value="/WEB-INF/client.xml"/>
</plug-in>
```

Step-VII :- 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) form;
InputForm if1 = 
    String name = if1.getName();
    int m1 = Integer.parseInt(if1.getM1());
    int m2 = Integer.parseInt
    int m3 = Integer.parseInt
    //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-VIII:- Add Result.jsp to Webroot folder of the project.

R.C. on Webroot folder → new → JSP → filename: result.jsp → finish.

Result is = <%= request.getAttribute("msg") %>

Step-IX:- Deployee this project in tomcat Server and also make sure that the Server appn is also in running mode.

R.C on project (StrutsApp) → Run as → my Eclipse Server appn → Tomcat 6.x → Ok

Step X:- Test the application.

Open browser coin dow → type this Url

http://localhost:2020/StrutsApp/input.jsp

Q:- What is the difference b/w programing 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 s/w specification having set of rules and guidlines to develop Softwares. while developing this s/w the programing Language will be used.

Eg:- JDBC, Servlet, JSP, EJB and etc..

→ Frameworks are special s/w technologies which provide abstraction Layer on multiple other core technologies

Eg:- Struts, JSF, Spring & etc...

```
----------register.jsp---------- (Formpage) Struts 2
<%@ taglib prefix="html" uri="/struts-tags" %>
<html>
<head></head>
<body>
<h1>User Registeration Page</h1>
<html:form action="reg">
    <html:textfield name="username" label="Username"/>
    <html:password name="password" label="Password"/>
    <html:submit value="Register"/>
</html:form>
</body>
</html>
----------success.jsp---------- (result page) Struts 2
<%@ taglib prefix="myHtml" uri="/struts-tags" %>
<html>
<head></head>
<body>
<h1>Success Page</h1>
<h4>Hello <myHtml:property value="username"/></h4>
</body>
</html>
----------web.xml---------- (DD file) Struts 2
<web-app>
  <filter>
    <filter-name>struts2</filter-name>
    <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <welcome-file-list>
    <welcome-file>register.jsp</welcome-file>
  </welcome-file-list>
<listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
</web-app>
----------struts.xml---------- (Struts configuration file) Struts 2
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
  "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
<!-- This tells the struts,that Injection(for Action Class) will be done by spring-->
<constant name="struts.objectFactory" value="spring" />
<!-- <constant name="struts.devMode" value="true" />-->
  <package name="myPack" extends="struts-default">
    <action name="reg" class="regAction">  <!--doesnot look for class,but looks for bean in Spring cfg file-->
      <result name="SUCCESS">success.jsp</result>
    </action>
  </package>
</struts>
----------applicationContext.xml---------- (Spring configuration file) Spring
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<!--Configuring datasource-->
<bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="username" value="scott"/>
```


↳ Fixed tag lib url.

Formpage

↳ retrives gives username from value stack obj

↳ special Url pattern

Struts 2 supplied filter configuration which actually acts as Controller. to make ServletFilter to trap and take care the request and responses of all the web resource programs.

ServletContext Listener to activate Springs web appln Context Container during Server startup.

(*1) ↳ bean id of real RegisterAction class (refer line no : 79)

(*1) Here 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 Struts Action class as SpringBean in SpringConfiguration file.

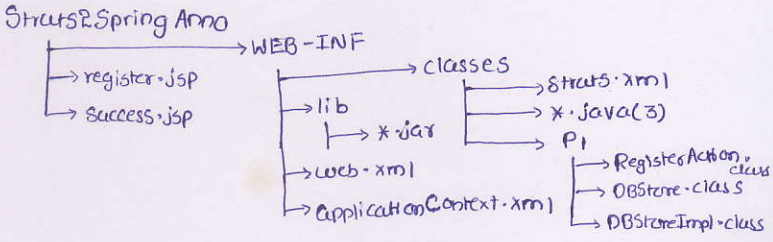
```
<property name="password" value="tiger"/>
<property name="url" value="jdbc:oracle:thin:@localhost:1521:orcl"/>
<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
</bean>
<context:component-scan base-package="p1" />   → to recognizes the SpringBeans of "p1" package
</beans>
----------RegisterAction.java---(Struts2 Action class Configured as the springBean
package p1;
import javax.annotation.Resource;
import org.springframework.stereotype.Component;
@Component("regAction")
public class RegisterAction {   → here struts2 Actionclass is configured as SpringBean
                                  in springConfiguration file.
@Resource(name="dbs")
private DBStore dbStore;   → here SpringBean class obj is injected to a property of
                             struts Action class.

public void setDbStore(DBStore dbs)
{                                   } setter method supporting setterInjection
dbStore=dbs;
}
    private String username,password;  → properties to hold Form data.

//getters and setters
    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
    // all struts logic here
    public String execute() {
    System.out.println(" Execute Method called");
    dbStore.storeUser(username,password);  → calling B-method of SpringBean class
        return "SUCCESS";
    }
}
----------DBStore.java---(Spring Interface) spring
package p1;
public interface DBStore{
    public void storeUser(String uname,String password);  → Declaration of B-method
}
----------DBStroreImpl.java---(SpringBean) spring
package p1;
import javax.sql.*;
import java.sql.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component("dbs")   → current bean id
public class  DBStoreImpl implements DBStore{
                                  → springInterface
    Connection con;
    @Autowired
    DataSource ds;  → Injects DataSource object to "ds" property through autowiring

    public void storeUser(String uname,String password)
    {                → B-method
    try
        {
        con=ds.getConnection();
                → gives JDBC Connection object from the Connection pool
```


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 Struts 2 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 /*
156 SQL> create table userdetails
157     (
158       username varchar2(10),
159       password varchar2(10)
160     );
161 */
```

Deployement Directory Structure



jarFiles in classPath :

Struts2-core-2.3.4.1.jar, org.sf.context-3.1.1.RELEASE.jar, ~~Spring.jar~~ ~~org.sf.core-3.1.1.RELEASE.jar~~ org.sf.beans-3.1.1.RELEASE.jar ~~commons-logging.jar~~

jarFiles 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 slw

- commons-logging → from spring 2.5
- freemarker-2.3.19
- javassist-3.11.0.GA
- ognl-3.0.5

} from Struts 2.3 slw

- 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 slw

- 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

http://localhost:2020/Struts2SpringAnno/

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 ServletFilter

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

→ Struts2 gives built-in jsp tag libraries whose taglib uri is: /struts-tags

→ ServletListeners are given to perform EventHandling on request, session ~~response~~, ServletContext Obj

→ Spring 2.5 gives org.sf.web.context.ContextLoaderListener as ServletContextListener which will be activated when ServletContext object is created/destroy. This Listener activates the webappn context container by taking ~~appn~~ ~~application.xml~~ applicationContext.xml as the Spring configuration file. In this process all the springbeans will be instantiated.

(*) For example application on Struts 2.x and spring 3.5 Integration refer the supplimentory handout given on 10/04/2013

Flow of execution related to Struts 2.x ~~and~~ with spring 3 appn handout of 10/04/2013

a) Deployeement

b) Instantiation of ServletFilter either during server startup or during deployment of (25-32 webappn.

c) → ContextLoaderListener activation (37-39)

→ webAppn ContextContainer activation by taking applicationContext.xml as the Spring Configuration file

→ pre-Instantiation of beans belonging to Spring

→ SpringBean class (DBStoreImpl obj) will be injection to StrutsAction class (RegisterAction)

d) enduser gives request to Formpage (register.jsp) this form page submits the request (ref:10)

e) ServletFilter traps and takes the request (26-32)

f) ServletFilter passes the request to Struts Action class Configuration (52-54)

g) ServletFilter calls execute() method of Struts Action class. (108-112)

h) this execute() calls storeDetails(-,-) of SpringBean class (132-153). 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 controle to Success.jsp page.

Module - VI
Spring AOP

→ Spring AOP is no way related with OOP. OOP is the methodology to create programing 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 appns more perfect or 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)

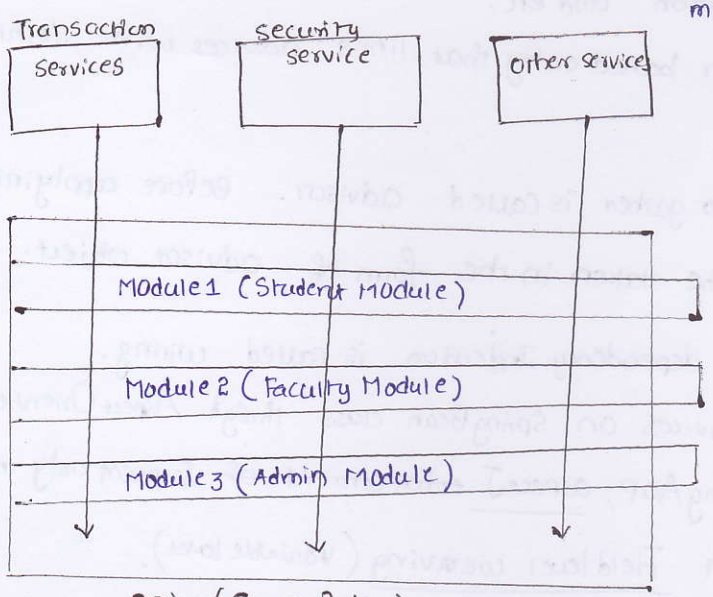
Logging service (keeps track of Application execution)

Jdbc Con 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.

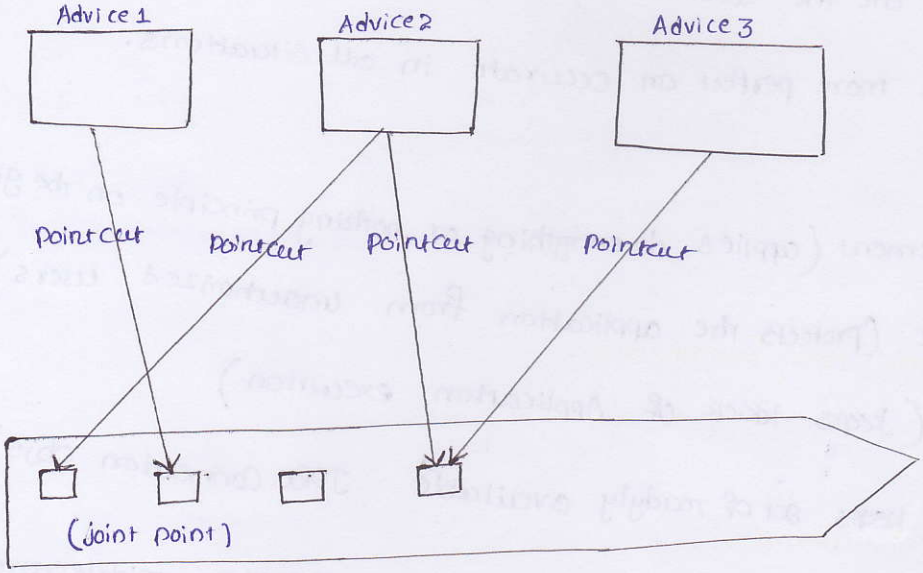
→ To understand the need of Spring AOP and understanding that refer page no's 100 to 102



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	



☐ Joint point

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

Joint Point :- Possible positions in our Business method where advices can be applied

Eg①:- begining 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 to gather 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 thrôgh Aspect Oriented Programing is called weaving. Spring AOP, aspectJ ~~enhancement of~~ Support only method level weaving. That means they donot Support Field level weaving (Variable level).

targetbean / target Object :-

SpringBean class object on which we are planning to applying middleware Service 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 Bussiness method.

→ Business methods called on proxy Object execute the Business method logic along with middleware services.

Org.sf.aop.framework.ProxyFactoryBean generates proxy object based on given target Object.

There are 4 types of advices

① Before Advice (Executes at the begining 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 begining of b.method and at the end of B.method)

→ using SpringAOP, AspectJ not only we can use Spring (or) server supplied Middleware Services we can also use them to develop User defined Middleware Services and advices.

for the above 4 types of advices refer page no's 104 to 106.

(X) procedure to develop Spring AOP based spring application.

Step①:- Develop Java class as the advice class

Step②:- Develop Spring Interface, SpringBean class having Business method declarations, definitions.

Step③:- Develop Spring Configuration file

a) cfg spring Bean class

b) cfg Advice class

c) cfg Advisor class pointing to Advice class

d) cfg ProxyFactoryBean Specifying advisor, SpringBean

NOTE:- This proxyFactoryBean gives SpringBean class object as proxy object.

Step④:- 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 position in B.method ~~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 NameMatchMethodPointCutAdvisor

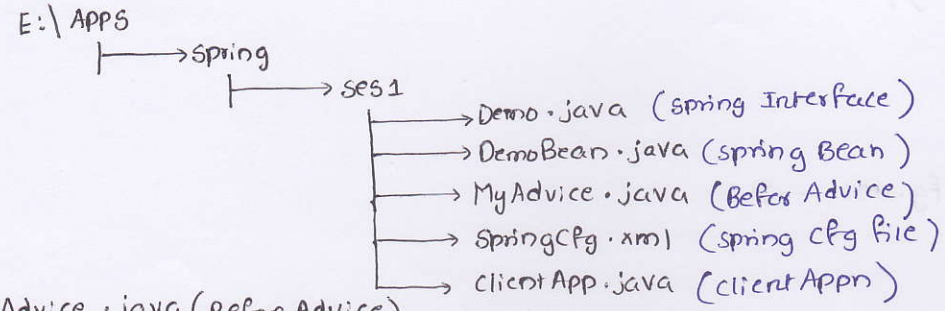
→ here the advices will be linked with b.methods based on the names of the B.methods.

② RegexpMethodPointcutAdvisor

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 :-




```
//MyAdvice.java(Before Advice)
import org.sf.aop.*;
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
S.o.p("MyAdvice: B.method name is "+method.getName());
S.o.p("MyAdvice: Name of Bean class is "+target.getClass());
S.o.p(method.getName()+" execution is started at "+new Date());
   //changing B.method arg value if it is invalied
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]="sathya";          //if it is invalied
 } //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");
    }
}
```

```
<!-- springcfg.xml -->
<!DOCTYPE ---- >
<beans>
    <bean id="db" class="DemoBean" /> <!-- BeanClass cfg -->
    <bean id="adv" class="MyAdvice" /> <!-- Advice cfg -->

    <!-- Advisor cfg -->
<bean id="advr" class="org.sf.aop.support.NameMatchMethodPointcutAdvisor">
    <property name="advise" 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.sf.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces" value="Demo" />
                                          └ spring Interface
    <property name="target" ref="db" />
                                └ Bean id
    <property name="interceptorNames">
        <list>
            <value> advr </value>
                     └ Advisor object
        </list>
    </property>
</bean>
</beans>
```

```
// ClientApp.java
import org.sf.context.support.*;
public class ClientApp {
    p s v m(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 RegulerExpressionPointCutAdvisor place the following code in the Spring cfg file

```
<!-- Advisor cfg -->
<bean id="advr" class="org.sf.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice" ref="adv" />
    <property name="pattern" value=".*" />
</bean>
```

Sample code of Around Advice

```
public class MyAdvice implements MethodInterceptor
{
    public Object invoke(MethodInvocation i1)
    {
        ..... //executes at the begining of the B.method
        .....
        .....
        i1.proceed(); //calls the B.method
        .....
        ..... //executes at the end of the B.method
        ....
    }
}
```

↳ represents the executing environment of the Business 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 Springweb 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() doesnot allow to catagorize the log messages.

④ → S.O.P() doesnot 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 parrellel.

③ → Log4j allows catagorize log msgs

debug < info < warn < error < fatal

④ → Log4j allows to filter Log messages while retreiving them.

⑤ → Log4j allows to format the log messages.

Log4j
- Logging for java
- Version: 1.x
- Vendor: apache foundation
- Open Source tool
- jar files log4j-<version>.jar

<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 Knocon Exceptions.

use "fatal" level in the catch blocks that handles Unknocon Exceptions.

```
try
{
  .....
  ....
  ....
  ....
} catch(SQLException se) //Knocon exception handling
  {
    logger.error("DB problem");
  }
 catch(Exception e) //unknocon exception handling.
 {
   logger.fatal("unknoco problem");
 }
```

There are 3 imponent objs in log4j programing

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 catagories. (iii) It Base obj for Log4j based logging operations.

```
Logger logger = Logger.getLogger(TestApp.class);
```

getLogger — Static Factory method

TestApp.class — 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, JDBC appending etc..)

→ will be linked with logger object to take and write log messages.

→ The available appenders are
- FileAppender (to write to files)
- ConsoleAppender (to write to Console)
- JDBCAppender (to write to DBs/w)
- IMAppender (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 avialable Layout classes are.
- HtmlLayout
- SimpleLayout
- PatternLayout
- XmlLayout and etc...

NOTE: If the loggerLevel retreive level log messages info we will get only those log messages whose logger level is >= Info.

(*) for SpringAop example appn that uses all the 4 types of advices refer appn (5) of the pageno's (119) to (121)

14/04/2013

Transaction management :-

→ The process of combining the related operations into single unit by applying do everything (executing them) or nothing principle is called Transaction management.

→ All sensitive logics like transfer money debit card processing must be developed by enabling Transaction mgmt service.

→ Transfer money operation (transferMoney(-,->-)) is the combination of two operations.

① Withdraw amount from source account.

② deposit amount into destination account.

It is recomended 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 ServerManaged (or) Technology supplied Transaction mgmt Service.

→ (i) In hibernate appns we use Technology supplied Transaction Service.

(ii) In Spring appns we can use either Technology supplied Transaction Service (or) Server Managed Transaction Service.

(iii) In EJB comps we use Server managed Transaction Service.

The typical code of Transaction management :-

```
public void bm1()
{ 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

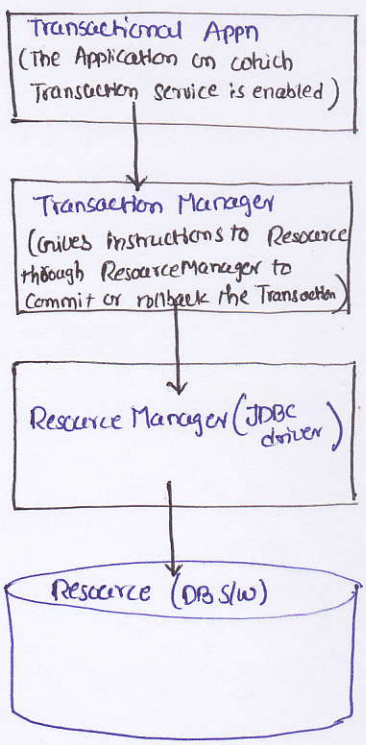
→ The Process of Combining multiple & related invidual operations into Single Unit is called "Atomocity"
The word "Atomocity" come from the word "Atom"

→ Consistency :- Even though rules kept on DataBase data are Violated during the Course of Transaction in there is a guarenty for no rule Violation at the end of Transaction mgmt code. then that code is called as Consistent Code.
↳ like balance must not be negative

→ Isolation :- The process of applying locks on DataBase of amount of data and allowing only one appn data to manipulate the data is called enabling Isolation. This Isolation prevents Concurrent and Simultaneous operations on DataBase data.

→ (long life) Durability :- The ability of bringing DataBase s/w back to normal state through logfiles and backup files cohen database is Currepted. 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 based on the Commit (or) rollback operations that Comes from Transactional appn.

→ based on the no of resources (DataBase s/ws) that are involved to perform Various Operations of Transactional code we can say there are 2 types of Transactional mangegments.

① Local Transaction (Tx) Mangegment.

→ all the operations of Transactional code will act on Single DB s/w

Eg:- transfer Money operation b/w two diff accounts of same bank.

② Distributed Transaction (Tx) Mangegemant.

→ The operations of Transactional code will act on Multiple DB s/w

Eg:- Transfer Money operation b/w two accounts of two diff banks.

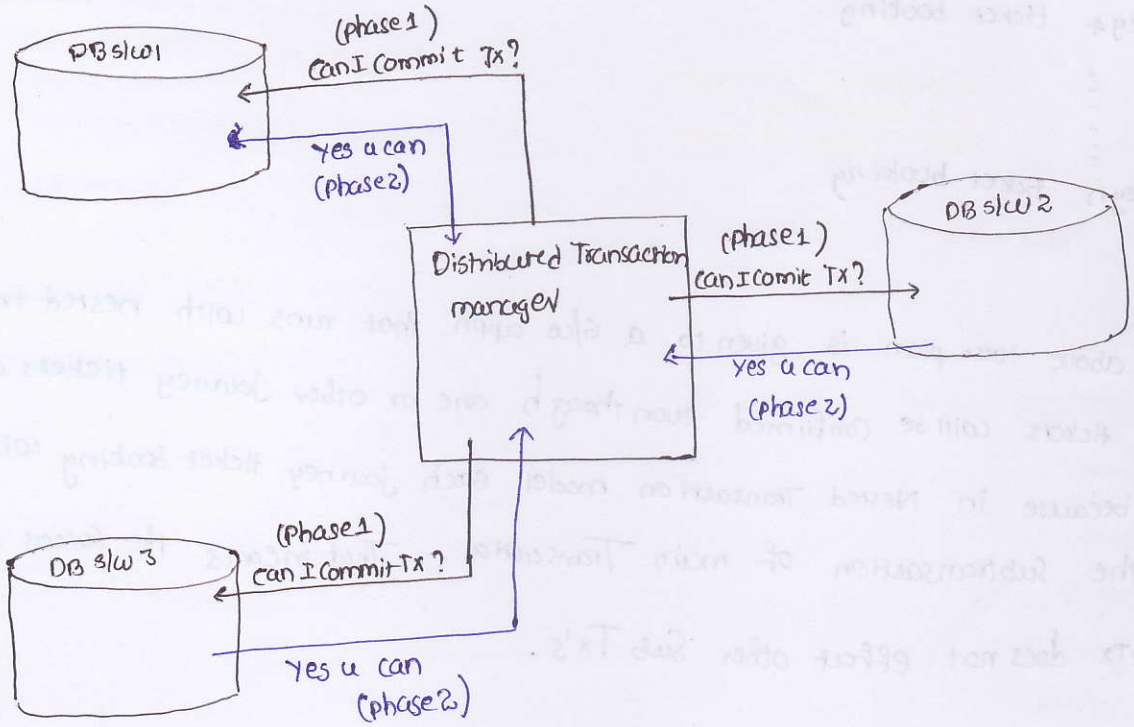
→ Hibernate, Spring, EJB Technologies support Local Transaction mgmt. Hibernate does not support Distributed Transactions. whereas Spring, EJB supports Distributed Transactions.

15/04/2013

→ Distributed Txs run on 2PC (2 phase commit) protocol.

In phase 1, The Distributed Tx manager will seek permission from DB s/w s to comm the Tx.

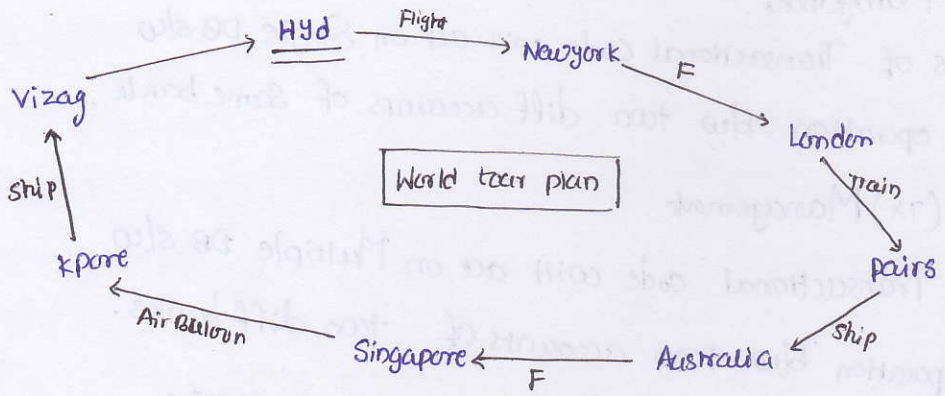
In phase 2, if all DB s/ws replay yes then the Distributed Tx will be committed Otherwise the Distributed Tx will be rollback.



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 {
  journey 1  ticket booking
  journey 2  ticket booking
  journey 3  ticket booking
  journey 4  ticket booking
        :
        :
  journey n  ticket booking
}
```

When the above tour plan is given to a s/w 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 Sub Tx's.

Nested Transaction model :-

```
main Tx {
    Sub Tx 1
    {
    journey1 Ticket booking
    }
    Sub Tx 2
    {
    journey 2 Ticket booking
    }
    =
    =
    Sub Tx n
    {
    journeyn Ticket booking
    }
}
```

→ EJB, hibernate, spring supports Flat Tx model

→ hibernate, EJB do not support Nested Tx Model

→ Spring supports Nested Transaction.

→ We can apply Transaction mgmt Service on Spring Applications in two ways

1) programatic Approach (write code directly in SpringBean 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 springbean 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 ServerManager 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 programatic approach refer appn (2) of the page no's 131 & 132

TransactionTemplate class provides the abstraction layer on Programatic Transaction management by automatically begining the Transaction. Committing (or) rollbacking the Transaction.

→ Avoid programatic Transaction mgmt in Spring appn because it makes the programer to write more code of Transcation 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 programatic Tx mgmt ~~the client appns must also~~ the Business method always execute with new Transaction because it can't coorde ~~with~~ coith Client supplied Transaction. This indicates Programatic Tx mgmt does not Suppert propegation of Transaction.

→ Declarative Tx mgmt Supports Transaction propegation. 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 TransactionService will be Configure through Xml files (or) annotations.

16/04/2013

→ weather the b.method on which Declarative Tx mgmt enabled runs coith no Transaction (or) new Transaction (or) Client Supplied Transaction coill be decided base on the Transaction attributes that is Configured on the business method.

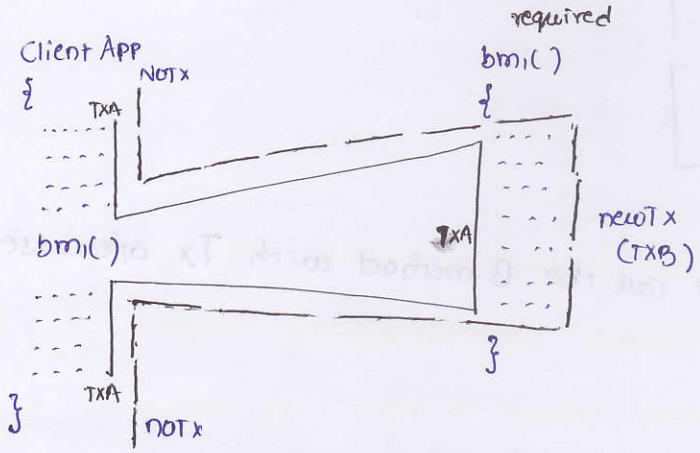
→ The 6 Tx attributes are

1) required
2) required new
3) supports (default)
4) Not supported
5) Mandatory
6) Never

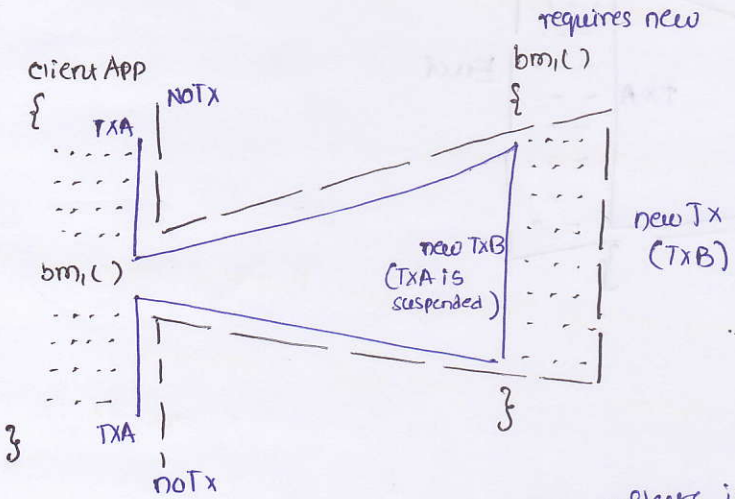
note:- Tx attributes are not required while dealing with programatic Tx mgmt.

note:- TransactionTemplate class not required while dealing with Declarative Tx mgmt.

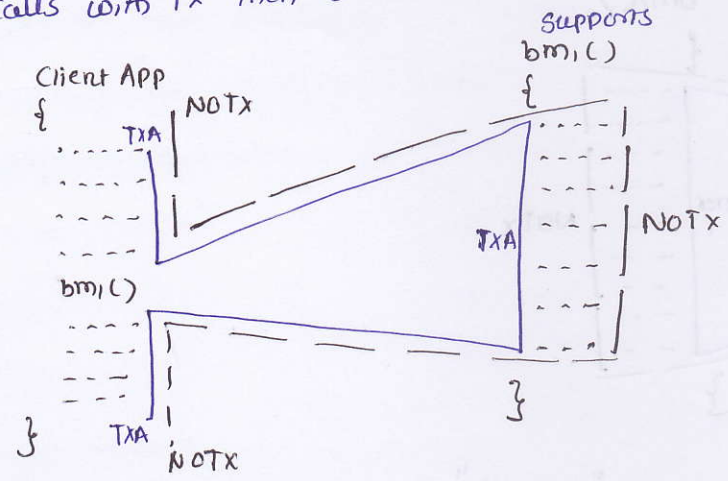
required :- B.method uses client supplied Tx if client appn calls the B.method with Tx. other wise new Tx will be started in the B.method.



requires new:- This B.method runs with new Tx irrespective of weather client calls B.method with (or) with out Tx

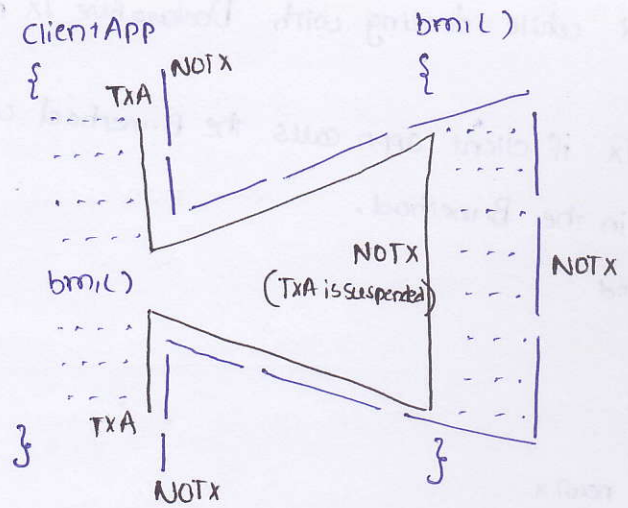


Supports:- The state of Tx in client appn also reflects in B.method. that means if client appn calls with Tx then B.method also runs in the Tx. other wise B.method runs without Tx.

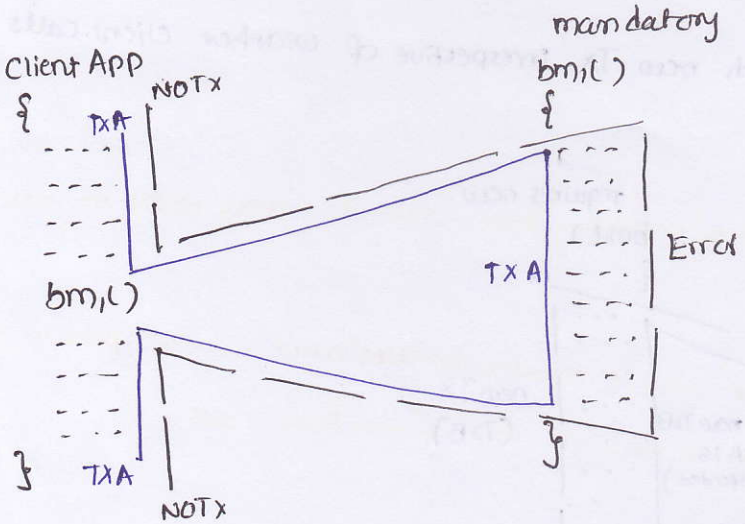


Not supported:- B.method always runs without Tx irrespective of weather client appn is calling B.method with or without Tx.

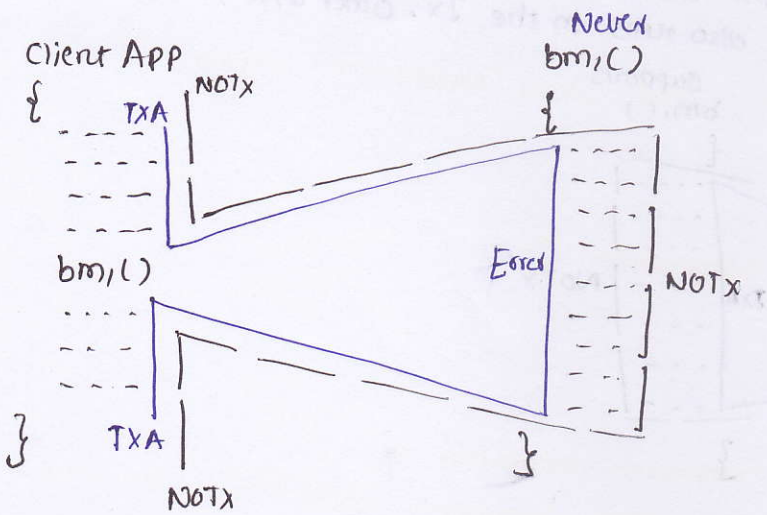
Not Supported



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 Effects of Tx attributes :-

TRansaction Attribute	CLient's Transaction	Bean's Transaction
Required	none	T_2
	T_1	T_1
RequiredNew	none	T_2
	T_1	T_2
Supports	none	none
	T_1	T_1
Mandatory	none	error
	T_1	T_1
NotSupported	none	none
	T_1	none
Never	none	none
	T_1	error

→ org.sf.transaction.interceptor.TransactionProxyFactoryBean returns proxy object on cohich Tx service is enabled.

→ org.sf.transaction.interceptor.NameMatchTransactionAttributeSource is like pointcut advisor to apply Transaction attributes on the b.methods of Spring Bean class.

→ In Declarative Transaction Mancagement the Transaction will be committed cohen no exception is raised in the Business method. otherwise the Transaction will be rolled back.
↳ RuntimeException

(*) For example appn on Declarative Flat, LOCal Transeaction mgmt refer appn (1) of the pagno'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 Pollocoing things.

a) add import org.sf.transaction.annotation.*;

b) add @Transactional(propagation = Propagation.REQUIRED) on bm1() b.method
↳ Tx attribute

//Demo.xml

a) must be schema base including "spring-beans.xsd, spring-tx.xsd, spring-aop.xs

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

jarfiles in CLASSPATH :-

Spring.jar, Commons-logging.jar (spring 2.5 envi..)

```
>javac *.java
>java ClientApp
```

Other Tx attributes while working with @Transactional Propagation.REQUIRES_NEW, Propagation.REQUIRE, Propagation.SUPPORTS, Propagation.NOT_SUPPORTED, Propagation.MANDATORY, Propagation.NEVER, Propagation.NESTED (to enable nested Txs)

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 programatic, local, Flat Tx management on SpringORM module based Hibernate Persistance Logic refer appn (1) of the page no's (4) to (7) of Supplimentary handout given on 17/04/2013

→ For example appn on Declarative, local, Flat Txs on Spring ORM module based persistence Logic refer appn (2) of the pange no's (1) to (3) Supplimentary 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 it's 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 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	~~raja~~ 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) wilable 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 other

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 recomended to enable Security Services on application in Declarative mode. we Can use Acegi framework (or) Spring Security flw to enable the declarative mode of security on Spring appns.

20/04/2013

(*) For example appn on aspectJ programing refer the appns given in 17/04/2013 handout.

→ AspectJ Annotations :-

@Aspect ⟶ To make the java class as the Advice class

@PointCut ⟶ To point to B.methods of SpringBean class.

@Before ⟶ To make the java methodas the Before advice.

@AfterReturning ⟶ To make the java method as the After advice.

@AfterThrowing ⟶ To make the java method as Throws Advice.

@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.*(..))")   → pointcut pointing to all the B.methods of Spring Bean
  public void testinterOk(){
  }
 @Before("testinterOk()")
  Public void beforeMethod()
  {
   S.o.p("iam before advice");
  }
 @AfterReturning("testinterOk()")
  public void afterReturningMethod()
  {
   S.o.p("iam after returning advice");
  }
}
```

jarfiles in CLASSPATH: aspectjrt-1.6.0.jar, spring.jar, aspectjweaver.jar, commons-logging.jar

>javac *.java

>java client

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.

Spring Web Module part-II

Web framework slws provide abstraction Layer on core servlet, jsp technologies and simplifies the process of developing MVC2 architecture based web apprns.

Struts ⟶ From apache foundation

JSF ⟶ From SunMS (oracle corp)

Webwork ⟶ From open Symphony

Spring Web mvc / spring mvc / spring web flow ⟶ From interface21

What is the diff blw Struts 1.x and spring mvc ?

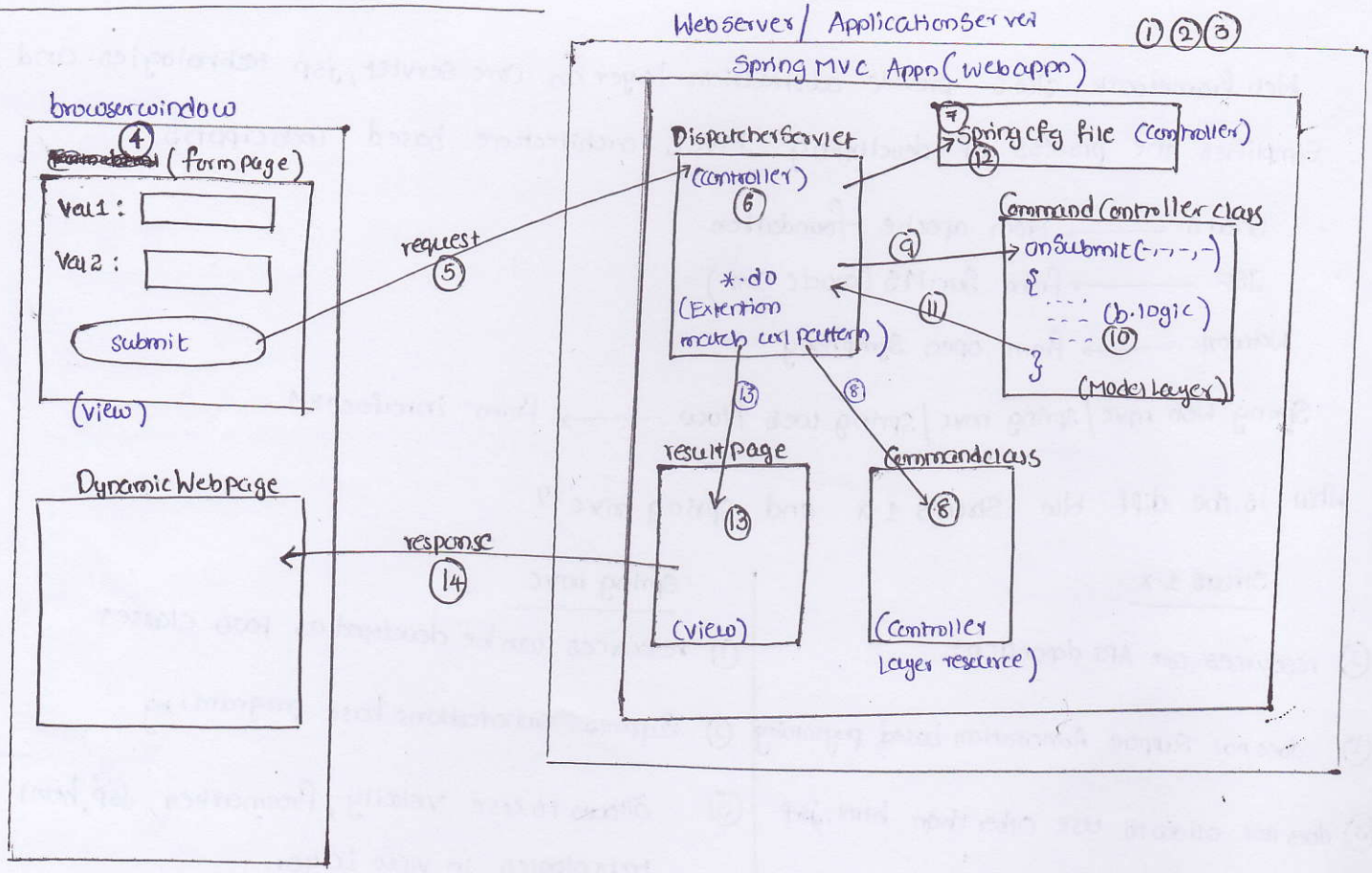
Struts 1.x	Spring mvc
(1) resources are API dependent	(1) resources can be developed as POJO classes
(2) does not support Annotation based programing	(2) Supports annotations base programing
(3) does not allow to use other than html, jsp technologies in view layer	(3) allows to use velocity, freemarker, jsp, html technologies in view Layer.
(4) Supports dependency Injection only on Formbeans	(4) Supports dependency Injection on every Java resource.
(5) Validator plugin is given to perform FormValidations	(5) Uses spring's built-in facilities to perform FormValidations

Terminology matching blw Struts 1.x and spring MVC

Terminology	Struts 1.x	Spring mvc
Controller Servlet	ActionServlet	Dispatcher Servlet
	FormBean	command class
	Action class	Command Controller class
	Struts cfg file	spring cfg file
	<Any filename>.xml	<DispatcherServlet logicalname>-servlet.xml
	ActionMapping	uri handlers
	ActionForwards (to configure Result Pages)	View Resolvers and views
ModelAndView object		

Same as Struts 1.x flow

Flow of execution in Spring MVC appn :-



W.r.to the diagram :-

① programmer deployees spring MVC web appn in webserver (or) appn server.

② because of load-on-startup enabled on the DispatcherServelet it coill be instantiated either during Server Startup or during the deployement of webappn.

⑤ This DispatcherServlet activats web appn Context Container by taking ~~DispatcherServlet.log~~ <Dispatcherservlet logicalname> - servlet.xml as the Spring Configuration file. In this process all the beans Configured in Spring Configuration file will be pre-instantiated.

④ ⑤ Enduser launces Form page on the browserwindow and submits the request to webappn.

⑥ DispatcherServlet traps and takes the request

⑦ DispatcherServlet uses Url-handler Configuration to know Command Controller class, Command class that are required to process the request.

⑧ DispatcherServlet writes the received Formdata 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 ModelAndView class object to DispatcherServlet.

⑫ DispatcherServlet uses the Views and ViewResolvers ~~cont~~ 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 cfgs.

① Uri handlers
(to link ~~incoming~~ incoming requests with Command controller classes)

② Command controller classes & Command classes.

③ View Resolvers.

View Resolver :- This configuration lets the spring fw to specify the technology that is required to develop the resources of view Layer. This ViewResolver class name varies based on the view Layer technology we use.

Technology	ViewResolver class name
Jsp	org.sf.web.servlet.view.InternalResourceViewResolver (default)
velocity	org.sf.web.servlet.view.velocity.VelocityViewResolver
Freemarker	org.sf.web.servlet.view.freemarker.FreeMarkerViewResolver
tiles	org.sf.web.servlet.view.tiles.TilesConfigurer
XSLT (XmlStylesheet Language for Transformation)	org.sf.web.servlet.view.xslt.XsltViewResolver
jasper reports	org.sf.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. ~~Simple UrlHandler~~

① org.sf.web.servlet.handler.SimpleUrlHandlerMapping :- (recommended to use) This class allows to use bean ids, bean names and even regular expressions to link ~~from~~ from urls (request urls) with Command Controller classes.

② org.sf.web.servlet.handler.BeanNameUrlHandlerMapping :- This class allows to use bean name (that starts with "/" symbol) to link from urls (request urls) 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.multiaction.MultiActionController :-
Useful to handle the multiple request coming from the multiple Submit buttons of the 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 :-

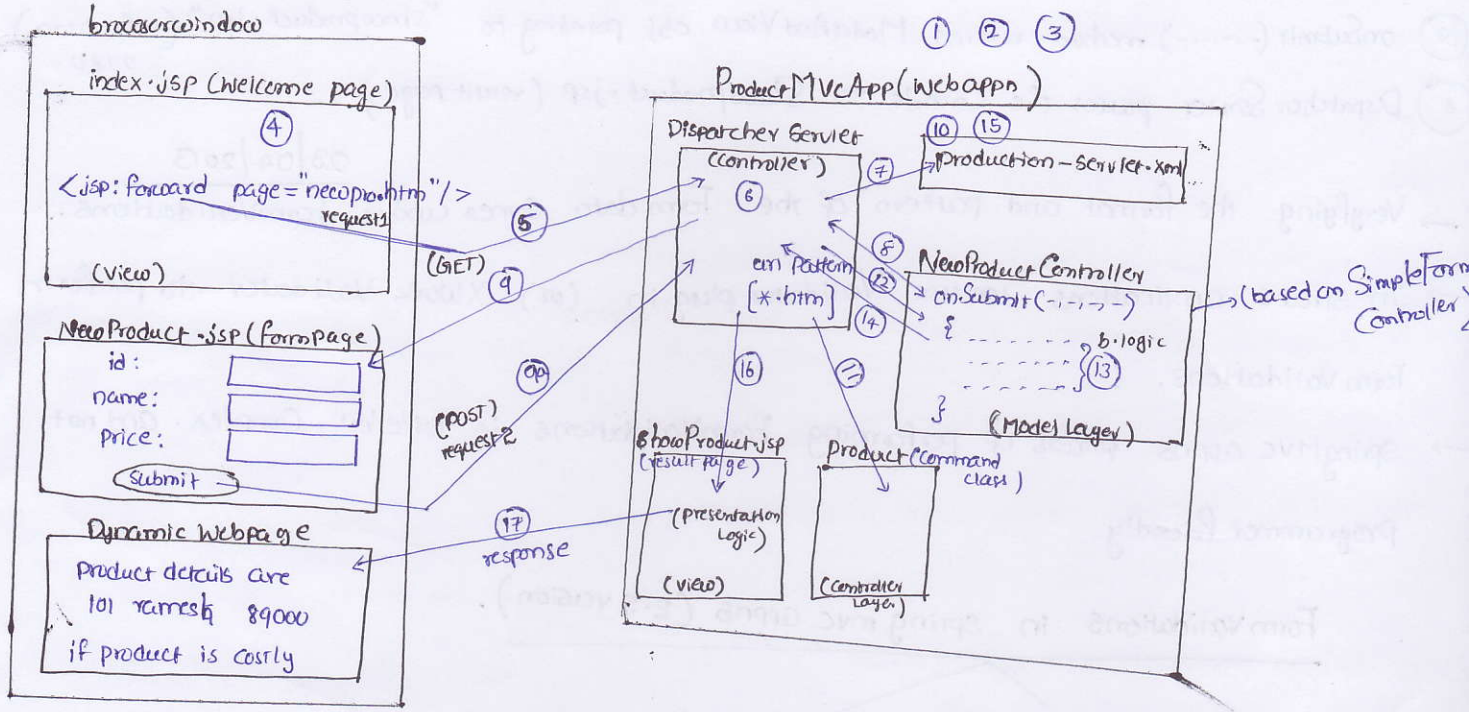
a) SessionForm ⟶ to specify the scope of Command class obj
b) formView ⟶ to specify the form page.
c) successView ⟶ to specify the result page.
d) commandName ⟶ to specify the Command class logical name.
e) commandClass ⟶ to specify the Command class name

and etc...

above diagram w.r.to diagram

⑤ Welcome page generates "GET" method based implicit request
⑥&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.
⑨a Formpage generates "POST" method based request. ⑩ Dispatcher Servlet Traps the request and links the request with CommandController class. Since the request method is "POST" the CommandController 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 refer apn (24) of the page no's (78) (79)

The Flow of execution of Appn (24) page no: (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 (GET method) to Webappn having "neopro.htm" in the request URL - (2658 reffer line no

(E) DispatcherServlet traps and takes the request (2695-2705).

(F) The SimpleUrlHandlerMapping class links the request with CommandController class (NewProductControl (2720, 2725) "GET"

(G) Since CommandController class type is SimpleFormController 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~~ 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 requestmethod is "POST"

(L) DispatcherServlet writes the received form data of form page to Command ~~Controller~~ class obj. (use setXxx(-) methods (ref: 2747, 49, 51))

(M) DispatcherServlet calls onSubmit(-,-,-,-) method of CommandControllerclass obj. (2761-2772)

(N) onSubmit(-,-,-,-) method returns ModelAndView obj pointing to "shoeproduct.jsp" (ref: 2770, 2730).

(O) DispatcherServlet passes the Controle to Shoeproduct.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 programatic FormValidations are possible using javascript

2. serverside

→ only programatic 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 seperate javaclass and this java class can be used in all kinds of Spring appns including SpringMVC appns. This class must implements org.springframework.Valition.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.sf.validation.Validator(I) and implement the FormValidation logic.

refer ProductValidator.java file of page no: (82) 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="p">        → logical name of Command Controller class
  <ul>
    <c:forEach var="e" items="${errors.allErrors}">
      <li>
```

↳ it represents only one Form validation error in each iteration.

↳ it contains all the Error messages (represents all FormValidation errors)

<spring:message text="${e.defaultMessage}" /> ⟶ It is foreach loop so, one by one validation errors displayed.

</c:forEach>

</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 Reslover called BeanNameView org.springframework.web.servlet.view.BeanNameViewResolver. To generate Results in the form of pdf document, X-cel sheets we dont 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 ⑤⑥⑦ 24/04/2013 handout.

→ 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.sf.web.servlet.mvc.multiaction.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 handout

Spring-Webservices

→ It provides the abstraction layer on JAX-RPC, JAX-WS webservices. In this process the Spring Container it self 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 refer appn (4) on 24/04/2013 handout

I would try to update our site 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