

Framework:

1. It is a special software that is capable of developing applications based on ceratian architecture with the ability to generate common logics of the application.
2. It is given based on core technologies.
3. It provides an abstraction layer on core technologies.

Struts: It is a framework to develop MVC-II architecture based web applications only.

Hibernate: Its a ORM Framework which is used only for database operations.

Spring: Its a framework software which can be used to develop any kind of Java/J2EE applications.

Note: Spring framework provides abstraction layer on:

1. java & J2EE core technologies
2. ORM Tools like:
 1. Hibernate
 2. JDO
 3. iBastis etc.,
3. AOP Framework etc.,

Spring Info:

Type: java-j2ee Framework software

Version: 2.5[Compatible with jdk1.5+]

Vendor: Interface21

Creator: Rod Johnson

Open Source

Download: www.springframework.org

Online Tutorials: www.roseindia.net

Articles:

1. www.javabeat.net
2. www.onjava.com
3. www.precisejava.com
4. www.devx.com

FAQ's: www.forum.springframework.org

Reference Books:

1. Spring Live
2. Spring in Action --> Manning Publishers

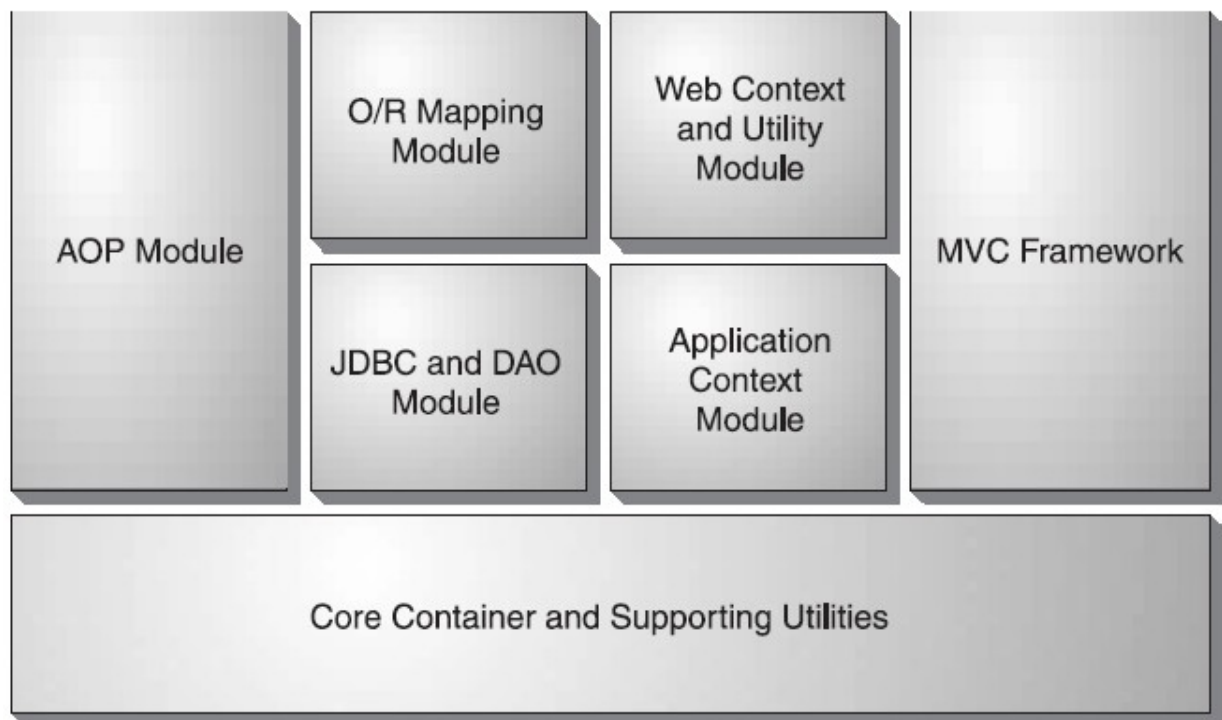
Introduction:

1. Spring framework is developed to simplify the development of enterprise applications in Java technologies.
2. It is an open source framework begin developed by Interface21.
3. The Spring provides light weight IoC Container and AOP framework.
4. It provides support for JPA, Hibernate, Web services, Schedulers, Ajax, Struts, JSF and many other frameworks.
5. The Spring MVC components can be used to develop MVC based web applications.
6. The Spring 3.0 framework has been released with major enhancements and support for the Java 5 [JDK1.5].
7. Spring can be used to configure declarative transaction management, remote access to your logic using RMI or web services, mailing facilities and various options in persisting your data to a database.
8. Spring framework can be used in modular fashion, it allows to use in parts and leave the other components which is not required by the application.

48 **Spring Architecture:** Spring is well-organized architecture consisting of seven modules. Modules
49 in the Spring framework are:

50 **Spring1.x:** It has 7 modules. They are:

- 51 1. Core
- 52 2. DAO
- 53 3. ORM
- 54 4. JEE/Context
- 55 5. Web
- 56 6. MVC
- 57 7. AOP



58 **Figure:** The Spring framework is composed of several well-defined modules.

59 **Spring2.x:** It has 6 modules only. Because Web & Web MVC modules were clubbed together &
60 given as Web MVC module.

61 **Core Module:**

- 62 1. It is base module for all modules.
- 63 2. Provides BeanFactory container which makes Spring as a container
- 64 3. Acts as a base module for other modules
- 65 4. provides the fundamental functionality of the Spring framework.
- 66 5. The Core package is the most import component of the Spring Framework.
- 67 6. This component provides the Dependency Injection [DI] features.

68 **DAO Module:**

- 69 1. It provides abstraction layer on JDBC & allows us to do database operations.
- 70 2. Using it we can focus only on database operations instead of doing repeated work by writing
71 common code again and again.

3. Here we can make use of JdbcTemplate class where we can call methods to do required database operation.
4. We can connect to database either by manual code same like jdbc or we can do easily by using JdbcTemplate class.
5. Using JdbcTemplate class we can retrieve data from database in the form of serializable objects.

ORM Module:

1. It provides an abstraction layer on ORM Framework software's like Hibernate, iBatis, JDO etc.,
2. Spring doesn't attempt to implement its own ORM solution
3. Spring's transaction management supports each of these ORM frameworks as well as JDBC.

JEE/Application Context Module:

1. It provides abstraction layer on Java/J2EE technologies like Java Mail, JMS, EJB etc.,
2. The context module is what makes spring as a framework.
3. This module extends the concept of BeanFactory
4. Adds support for internationalization (I18N) messages, application life cycle events, and validation.
5. Supplies many enterprise services such as e-mail, JNDI access, EJB integration, remoting, and scheduling.

AOP [Aspect Oriented Programming] Module:

1. One of the key components of Spring is the AOP framework.
2. It provides a methodology to configure middleware services or to perform any authentication, validations or to perform any pre processing logic or post processing logic on our business logic methods.
3. Without disturbing any kind of spring resources we can integrate our spring application with any kind of middleware service.
4. Spring provides a separate layer for middleware configuration.
5. This module serves as the basis for developing your own aspects for your Spring-enabled application.

Web MVC Module: It is given for two operations:

1. To integrate spring application with any other web based framework like Struts, JSF etc.,
2. It provides its own web framework named as Spring MVC to develop MVC-II architecture based web applications. It provides the MVC implementations for the web applications.

Note: We can develop a spring application either by using all modules or only by using few modules of spring. We can also integrate our spring module with any kind of java application. So spring is **loosely coupled**.

Spring is a open source, light weight, loosely coupled, aspect oriented & dependency injection based framework software which can be used to develop any kind of java/j2ee application.

Open Source: Spring software can be downloaded and used at free of cost. Spring developed source code can be referred by any one who is interested.

Container is a special software which can manage complete life cycle of a given resource.

113 **Loosely Coupled:** Need not to inherit any class or interface as we do for our servlet programming.
114 Instead of having "is-a" relationship here we can have "has-a" relationship.

115 **Lightweight:**

- 116 1. Spring is lightweight in terms of both size and overhead.
- 117 2. The entire Spring framework can be distributed in a single JAR file that weighs in at just
118 over 1 MB.
- 119 3. **Spring is nonintrusive:** objects in a Spring-enabled application typically have no
120 dependencies on Spring specific classes.
- 121 4. Mainly spring is used for business logic. The same business logic if we try to develop using
122 EJB, then its compulsory to have support of any application server like weblogic. But the
123 same logic can be executed in Spring without help of any web server or application server.
124 Because Spring provides its own containers in the form of pre defined classes.

125 **Note:**Servlet Container & EJB Container are heavy weight because unless we start web server or
126 application server we cannot activate either servlet container or EJB Container.

127 **Inversion of control:**

- 128 1. Spring promotes loose coupling through a technique known as inversion of control (IoC).
- 129 2. When IoC is applied, objects are passively given their dependencies instead of creating or
130 looking for dependent objects for themselves.
- 131 3. The container gives the dependencies to the object at instantiation without waiting to be
132 asked.

133 **Aspect-oriented:**

- 134 1. Spring comes with rich support for aspect-oriented programming.
- 135 2. It enables cohesive development by separating application business logic from system
136 services (such as auditing and transaction management).
- 137 3. Application objects do what they're supposed to do—perform business logic—and nothing
138 more. They are not responsible for (or even aware of) other system concerns, such as
139 logging or transactional support.

140 **Container:**

- 141 1. Spring is a container in the sense that it contains and manages the life cycle and
142 configuration of application objects.
- 143 2. We can configure how each of our beans should be created—either create one single
144 instance of bean or produce a new instance every time one is needed based on a
145 configurable prototype.

146 **Enterprise Application:** Its a large scale appliaiton with complex business logic & configured with
147 middleware services like transaction management, security etc., Eg: Banking Applications.

148 **Dependency Lookup:** This concept is used to find required resource among multiple resources. It
149 takes some time to look up waht we need. It performs "PULL" operation to get required values.

150 **Eg:** Getting JDBC DataSource object from registry server.

151 **Note:** Registry servers are used to provide global visibility [to access from any where through out
152 the world].

153 **Eg:** RMI Registry

154 **Dependency Injection [IOC – Inversion Of Control]:** Container software will gather all the
155 required values and injects into resources of application dynamically. Here Dependent values will
156 be pushed to resources dynamically by underlying container.
157 **Ex:** The way how JVM/JRE calls constructor of a class to assign some values at the time of object
158 creation.

159 **Note:** Using spring we can develop any kind of Java/J2EE application. But mostly in industry we
160 can find spring used for development of Business Logic.

161 **Installation of spring:** Download software & extract "**Spring-framework-2.5.1-with**
162 **dependencies.zip**" file.

163 **Spring Jars:** To work with any kind of spring applicaiton we need to set the classpath for two jar
164 files. They are:

- 165 1. **spring.jar** [main jar file] [Location: sping-home\dist\spring.jar]
- 166 2. **commons-logging.jar** [Dependency jar file] [Location:]

167 **Normal Spring Application:**

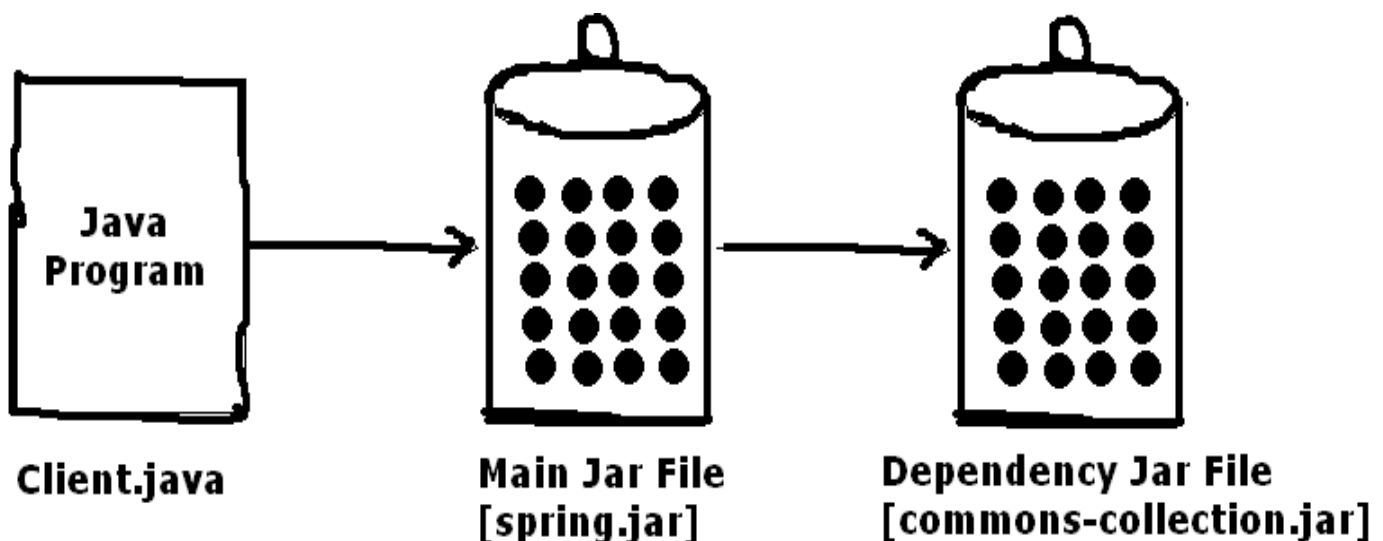
- 168 1. **Compilation:** Set the classpath for main jar file
- 169 2. **Execution:** Set the classpath for both main & dependency jar file. Becuase at run time of
170 program the control transfers from main jar file to dependency jar file.

171 **Spring Web Applicaiton:**

- 172 1. **Compilation:** Set the classpath for main jar file
- 173 2. **Execution:** Place both main & dependency jar files inside "lib" folder of WEB-INF.

174 **Note:** Spring programs will make use of classes and interfaces resideing inside "spring.jar"
175 file and the classes which are residing inside "spring.jar" file can make use of classes &
176 interfaces which resides in dependency jar file "commons-logging.jar".

177 **Classpath:** So when ever we need to work with a spring applicaiton its compulsory to set the
178 classpath for all related main jar files and also dependent jar files.



179 **Process of classpath setting:** My Computer icon -> advanced -> environment variables -> system
180 variables -> [if classpath variable is existing -> edit, if classpath variable doesn't exist then create a
181 new classpath variable] & specify c;asspath.

Note:

1. When we need set the classpath for multiple jar files, every jar file need to be specified with absolute path and every jar file path should be separated with semicolon [;]
2. The classpath what we specify in environmental variables will not be reflected to any ide like netbeans or eclipse or myeclipse etc.,
3. When we configure the same variable in user variable & system variable then user variables will dominate system variable [for path or classpath]
4. If any jar file missed in middle of classpath then the rest of classpath will be ignored. So always we need to specify our classpath in beginning of the path.

Features of Spring:

1. Supports POJO/POJI model programming.
2. Provides two light weight containers in the form of pre defined classes.
 1. **Bean Factory Container** or **Core Container** or **IOC Container**
-org.springframework.beans.factory.BeanFactory Interface
 2. **JEE Container** or **Application Context Container.**
-org.springframework.beans.factory.xml.ApplicationContext interface
3. Spring containers are activated from server side programs.
4. Can be used to develop any kind of java/j2ee applications
5. We can develop an application only by using spring or we can also integrate spring with any other framework.
6. Provides built in middleware services like connection pooling, transaction management, logging etc.,
7. Supports to work with third party supplied and web server or application server supplied middleware services.
8. Supports nested transaction & distributed transaction management.
9. Supports AOP framework to integrate with middleware services.
10. Provides annotation based mapping as a alternative to xml configuration which increases performance of the application.
11. **Transaction Management:**
 1. Spring framework provides a generic abstraction layer for transaction management.
 2. This allows the developer to add the pluggable transaction managers, and making it easy to do transactions without dealing with low-level issues.
 3. Spring's transaction support is not tied to J2EE environments and it can be also used in container less environments.
12. **JDBC Exception Handling:** The JDBC abstraction layer of the Spring offers a meaningful exception hierarchy, which simplifies the error handling strategy

Specific API: API which is used for development of only a particular kind of application. Eg: servlet-api is used only for the purpose of web application development.

POJO Classes: The classes that are not API dependent are called as POJO [Plain Old Java Object] classes.

POJO: [plain old java object] A pojo class is a class which should not extend any predefined class belongs to specific API and it should not implement any predefined interface belongs to specific api.

226 **Following are pojo classes:**

227 class C1

228 {

229 }

230 class C2 extends C1

231 {

232 }

233 class C3 extends java.lang.Thread

234 {

235 }

236 **Note:** Thread is a predefined class but its not confined for development of any particular kind of
237 application. In another words Thread class can be used for any kind of java application. So our class
238 C3 is a pojo class.

239 class C4 implements java.lang.Runnable

240 {

241 }

242 **Note:** Runnable is an interface which can be used to create a thread. So it can be used for
243 development of any kind of java application.

244 class C5 implements java.io.Serializable

245 {

246 }

247 **Note:** Serializable interface implementation class objects will become serializable objects. So that
248 we can transfer from one location to another location over the network.

249 **Following are not POJO classes:**

250 class C6 extends GenericServlet

251 {

252 }

253 **Note:** GenericServlet class is used only for the purpose of web application development. So our
254 class is not a pojo class.

255 class C7 implements Servlet

256 {

257 }

258 **Note:** Our class is implementing Servlet interface which is confined for development of web
259 applications.

260 class C8 implements java.rmi.Remote

261 {

262 }

263 **Note:** Remote interface is used only for development of Distributed Applications. So the above
264 class is not a pojo class.

265 **POJI Interfaces:** The interfaces that are not API dependent are called as POJI [Plain Old Java
266 Interface]

267 **POJI: [Plain Old Java Interface]** Poji is an interface which should not extend any predefined
268 interface which belongs to specific API.

269 **Examples of POJI are:**

270 interface I1

271 {

272 }

273 interface I2 extends I1

274 {

275 }

276 interface I3 extends Runnable

277 {

278 }

279 interface I4 extends Serializable

280 {

281 }

282 **Examples of non POJI are:**

283 interface I5 extends Remote

284 {

285 }

286 interface I6 extends Servlet

287 {

288 }

289 **Note:** The above interfaces are non poji because they are extending interfaces which are api
290 specific.

291 **Server:** A program which is providing service is called as server.

292 **Client:** A program which is giving request to get some service is called as client program.

293 **Local Client:** If server and client both are executing under same JVM [Java Virtual Machine] then
294 that client program can be called as a Local Client and server can be called as Local Server.

295 **Remote Client:** If server and client both are executing under different JVM's then we can consider
296 client as Remote client to server and server as Remote Server to client program.

297 **Spring Containers:**

298 **1. Core container or IOC container or BeanFactory container**

299 1. Its an implementation class of an interface BeanFactory

300 2. It supports **lazy loading**.

301 1. It means container will not create objects of cofigured pojo classes immediately.

302 2. It waits till it recieves the first request.

303 3. Once the object is created the same object will be maintained till end of the
304 application.

305 3. By default it considers all the configured classes as singleton classes.

306 4. So that the same object will be given to all requests which belongs to same class.

307 5. We cannot change singleton scope for core container.

308 6. provides the fundamental functionality of the Spring framework.

309 7. BeanFactory, the heart of any Spring-based application.

310 **2. Advanced container or ApplicationContext container**

311 1. It is advanced container.

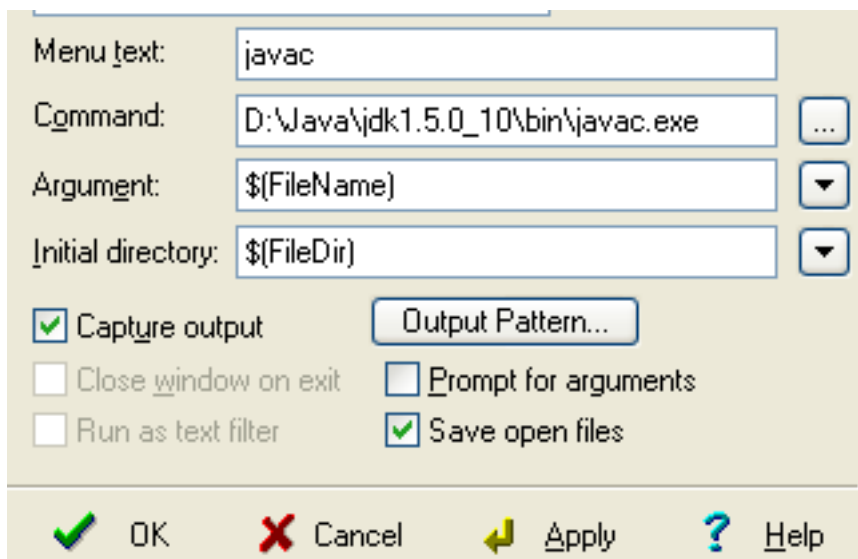
312 2. By default it supports early loading.

313 1. It means without recieving any request from client program immediately it will
314 create object for every singleton scoped class.

- 315 2. When ever the container gets acivated immediately it creates objects of all singleton
316 classes.
317 3. Has many advantages over core container.

318 **Execution of java program using Edit Plus editor:** Once we configure Edit Plus we can compile
319 and execute java programs just by using short cut keys. So it provides a great flexibility to develop
320 java programs.

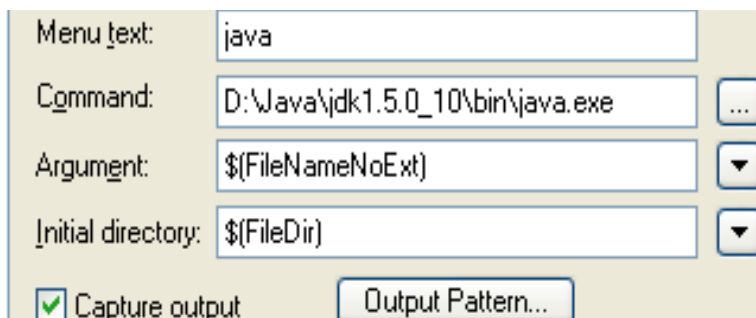
321 **Configure Compilation process:** Tools [menu bar] → configure user tools → add tool → program
322 → give menu text as javac/compile [which ever is comfortable to you] → for command click on
323 button with 3 dots [browse button] and locate **javac** from **bin** folder of JDK software.



324 Argument → File Name →
325 Initial Directory → File Directory
326 Select check box of capture output → Apply → ok.

327 **Configure Execution process:**

328 → give menu text as java/Execute [which ever is comfortable to you] → for command click on
329 button with 3 dots [browse button] and locate **java** from **bin** folder of JDK software.



330 Argument → File Name Without Extension →
331 Initial Directory → File Directory
332 Select check box of capture output → Apply → ok.
333 **Note:** Click on tools it shows you short cut keys to execute our java programs.

Resources of spring core module application:**1. Interface [POJI]**

1. It is optional in all applications except distributed applications.
2. It can contain prototype of our business logic methods.

2. Spring Bean class [POJO class]

1. It can be implementation class of our interface
2. It contains definition for business logic methods
3. It can have properties and values can be injected by spring container through DI [Dependency Injection].
4. It can have setters methods or constructors to support dependency injection.
5. It can have user defined methods which can act as life cycle methods.
6. Along with business logic methods we can also have helper methods.

3. Spring Configuration file [*.xml file]

1. We can give any name for this xml file
2. standard name is to give as "applicationContext.xml"
3. Initially we need to specify either **DTD (Document Type Definition)** or **XSD (XML Schema Definition)**
4. Xml file can be validated based up on either XSD or DTD.
5. Its compulsory to specify either DTD or XSD. [DTD is traditional & XSD is advanced]
6. We can configure all bean classes [business logic classes] here.
7. We can configure user dined classes & also predefined classes.
8. Every class can have a unique identification name.
9. We can also configure bean class properties here. Through this we can pass instructions to spring container to inject values with dependency injection.
10. Spring container works based up on the instructions what we have given in spring configuration file.

4. Client Program [*.java]

1. Locates spring configuration file
 1. To locate spring configuration file regularly we will be using either **ClassPathResource** or **FileSystemResource**
 2. **Resource** is interface
 3. **ClassPathResource** & **FileSystemResource** both are implementation classes or **Resource** interface.
 4. If client program & spring configuration file both resides in same folder then we can use any implementation class without any problem.
5. **FileSystemResource:**
 1. If it is required to specify either **abosolute path [Eg: c:\folder-name\file-name.xml]** or **relative path [Eg: sub-folder-name\file-name.xml]** in those cases we need to use **FileSystemResource** class.
 2. **Drawback:** If we modify location of spring configuration file then again we need to modify either aboslute path or relative path in our program.
6. **ClassPathResource:**
 1. We can create a jar file containing our spring configuration file and we can place it in our classpath.
 2. So that the spring configuration file can be any where. But still our client program can access it.
 3. **Drawback:** If we change our spring configuration file then again we need to create a jar file freshly.
2. Activates spring container

- 383 1. Here we can use **BeanFactory** interface and allocate memory of **XmlBeanFactory**
384 class which is an implementation class of BeanFactory interface.
385 **3. Reads spring configuration file:** Once the container is activated automatically it reads
386 spring configuration file.
387 4. Requests spring container to give object of any bean class by using identification name
388 of the bean class.
389 5. By using bean class object we can call business logic methods.

390 **Example: Simple Spring Application.**

391 **//interface program**

392 **//DemoInter.java**

393 public interface DemoInter

394 {

395 public String wish(String uname);

396 }

397 **//Implementation class**

398 **//DemoInterImpl.java**

399 public class DemoInterImpl implements DemoInter

400 {

401 private String message;

402 public DemoInterImpl()

403 {

404 System.out.println("Constructor executed");

405 }

406 public void setMessage(String message)

407 {

408 System.out.println("setter method executed");

409 this.message=message;

410 }

411 public String wish(String uname)

412 {

413 return message+" "+uname+" Have a great day";

414 }

415 }

416 **Note:** We can copy paste DTD or XSD into spring configuration file by searching xml files
417 provided by our spring software.

418 **Spring Configuration file**

419 **//spring.cfg.xml**

420 **//Here we are specifying DTD (Document Type Definition)**

421 <?xml version="1.0" encoding="UTF-8"?>

422 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"

423 "http://www.springframework.org/dtd/spring-beans.dtd">

```
424 <beans>
425     <bean id = "ic" class="DemoInterImpl">
426         <property name="message">
427             <value>Hello</value>
428         </property>
429     </bean>
430 </beans>
```

431 **(or)**

432 **//Here we are specifying XSD (XML Schema Definition)**

```
433 <?xml version="1.0" encoding="UTF-8"?>
434 <beans xmlns="http://www.springframework.org/schema/beans"
435     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
436     xsi:schemaLocation="http://www.springframework.org/schema/beans
437     http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
```

```
438     <bean id = "ic" class="DemoInterImpl">
439         <property name="message">
440             <value>Hello</value>
441         </property>
442     </bean>
443 </beans>
```

444 **//Client Program**

445 **//DemoClient.java**

```
446 import org.springframework.core.io.Resource;
447 import org.springframework.core.io.ClassPathResource;
448 import org.springframework.beans.factory.BeanFactory;
449 import org.springframework.beans.factory.xml.XmlBeanFactory;
450 public class DemoClient
451 {
452     public static void main(String args[])
453     {
454         //location of spring configuration file
455         Resource res=new ClassPathResource("spring.cfg.xml");
456
457         //activation of spring container & then it reads spring configuration file
458         BeanFactory factory=new XmlBeanFactory(res);
459
460         //requesting object of a bean class by using its identification name
461         DemoInter d=(DemoInter)factory.getBean("ic");
462
463         //calling business logic method
464         System.out.println(d.wish("Sai"));
465     }
466 }
```

467 **Note:** To execute any spring program basically we need to have the following jar files in classpath.

465 Main Jar file: **spring-framework-2.5.1\dist\spring.jar**

466 Dependency Jar file:

467 **spring-framework-2.5.1\lib\jakarta-commons\commons-logging.jar**

468 **Singleton Demo:**

469 **Note:** All resources are same as above program. Only change in client program.

```
470 import org.springframework.core.io.Resource;
471 import org.springframework.core.io.ClassPathResource;
472 import org.springframework.beans.factory.BeanFactory;
473 import org.springframework.beans.factory.xml.XmlBeanFactory;
474 public class DemoClient
475 {
476     public static void main(String args[])
477     {
478         Resource res=new ClassPathResource("Demo.xml");
479         BeanFactory factory=new XmlBeanFactory(res);
480         DemoInter d1=(DemoInter)factory.getBean("ic");
481         DemoInterImpl d2=(DemoInterImpl)factory.getBean("ic");
482         DemoInter d3=(DemoInter)factory.getBean("ic");
483         System.out.println(d1.wish("Sai"));
484     }
485 }
```

486 **Note:**

- 487 1. getBean() is a method which returns us super class **Object**. So we can classcast it to either
- 488 our interface or implementation class.
- 489 2. We have requested the same class object for 3 times. But container creates only one object &
- 490 the3 same object will be given for every request.
- 491 3. We can have any number of business logic methods inside the same class.

492 **Example: Configuring a predefined class object as property of pojo class.**

493 **//spring.cfg.xml**

```
494 <?xml version="1.0" encoding="UTF-8"?>
495 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
496 "http://www.springframework.org/dtd/spring-beans.dtd">
497 <beans>
498     <bean id="dat" class="java.util.Date">
499         <property name="year">
500             <value>85</value>
501         </property>
```

```
502         <property name="month">
503             <value>10</value>
504         </property>

505         <property name="date">
506             <value>19</value>
507         </property>
508     </bean>

509     <bean id="demo" class="DemoInterImpl" >
510         <property name="message">
511             <value>Njoy Spring Programming :)</value>
512         </property>

513         <property name="dt" >
514             <ref bean = "dat"/>
515         </property>
516     </bean>
517 </beans>

518 //DemoInter.java
519 public interface DemoInter
520 {
521     public void show();
522 }

523 //DemoInterImpl.java
524 import java.util.Date;

525 public class DemoInterImpl implements DemoInter
526 {
527     private String message;
528     private Date dt = null;

529     public DemoInterImpl()
530     {
531         System.out.println("DemoInterImpl zero argument constructor");
532     }

533     public void setMessage(String s)
534     {
535         System.out.println("setMessage() of DemoInterImpl class");
536         message=s;
537     }

538     public void setDt(Date dt)
539     {
540         System.out.println("setDt() of DemoInterImpl class");
541         this.dt = dt;
542     }
```

```
543     public void show()
544     {
545         System.out.println("Mesage is: "+message);
546         System.out.println("Date is: "+dt);
547     }
548 }
```

549 Client.java

```
550 import org.springframework.core.io.Resource;
551 import org.springframework.core.io.ClassPathResource;
552 import org.springframework.beans.factory.BeanFactory;
553 import org.springframework.beans.factory.xml.XmlBeanFactory;
554 public class DemoClient
555 {
556     public static void main(String args[])
557     {
558         Resource res=new ClassPathResource("spring.cfg.xml");
559         BeanFactory factory=new XmlBeanFactory(res);
560         DemoInter d1=(DemoInter)factory.getBean("demo");
561         d1.show();
562     }
563 }
```

564 **//Example: Configuring a user defined class object as property of pojo class.**

565 //DemoInter.java

```
566 public interface DemoInter
567 {
568     public void show();
569 }
```

570 DemoInterImpl.java

```
571 import java.util.Date;
572 public class DemoInterImpl implements DemoInter
573 {
574     private int age;
575     private Date dt = null; //pre defined class
576     private TestBean tb = null; //user defined class
577     public DemoInterImpl()
578     {
579         System.out.println("DemoInterImpl zero argument constructor");
580     }
581     public void setAge(int age)
582     {
583         this.age = age;
584     }
585     public void setDt(Date dt)
586     {
587         this.dt = dt;
588     }
```

```
589     public void setTb(TestBean tb)
590     {
591         this.tb = tb;
592     }
593     public void show()
594     {
595         System.out.println("Age is: "+age);
596         System.out.println("Date is: "+dt);
597         System.out.println("Message is: "+tb);
598     }
599 }
```

600 **//spring.cfg.xml**

```
601 <?xml version="1.0" encoding="UTF-8"?>
602 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
603 "http://www.springframework.org/dtd/spring-beans.dtd">
604 <beans>
605     <bean id="dat" class="java.util.Date">
606         <property name="year">
607             <value>85</value>
608         </property>
609         <property name="month">
610             <value>10</value>
611         </property>
612         <property name="date">
613             <value>19</value>
614         </property>
615     </bean>
616     <bean id="tbean" class="TestBean" >
617         <property name="message">
618             <value>Hello friends.... Njoy this beautiful n gifted life :)</value>
619         </property>
620     </bean>
621     <bean id="demo" class="DemoInterImpl">
622         <property name="age">
623             <value>25</value>
624         </property>
625         <property name="dt">
626             <ref bean = "dat"/>
627         </property>
628         <property name="tb" ref = "tbean"/>
629         <!-- <ref bean = "tbean"/>
630         </property> -->
631     </bean>
632 </beans>
633 //DemoClient.java
634 public class DemoClient
635 {
636     public static void main(String args[])
637     {
```



```

638         Resource res=new ClassPathResource("spring.cfg.xml");
639         BeanFactory factory=new XmlBeanFactory(res);
640         DemoInter d1=(DemoInter)factory.getBean("demo");
641         d1.show();
642     }
643 }

```

644 **//Example: Different Bean Property Types**

645 **Note:** In a bean class we can have any type of properties. For the following properties we have
646 corresponding tags to be onfigured in our spring configuration file.

Bean Property type	Tag in spring configuration file
Primitive (or) java.lang.String	<value>
Other bean type [class/interface type]	<ref>
Java.util.List type [stack, linkedlist]	<list>
Java.util.Set	<set>
Java.util.Map	<map>
Java.util.Property	<props>
Java.util.Array or arrays	<list>

```

647 //DemoInter.java
648 public interface DemoInter {
649     public String sayHello();
650 }
651 //DemoImple.java
652 import java.util.Date;
653 import java.util.List;
654 import java.util.Map;
655 import java.util.Properties;
656 import java.util.Set;
657 public class DemoImple implements DemoInter{
658     private float salary;
659     private TestBean tb = null;
660     private Date dt = null;
661     private List fruits = null;
662     private List veg = null;
663     private Set phone_nos = null;
664     private Set emails = null;
665     private Map perDetails = null;
666     private Map capitals = null;
667     private Properties faculties = null;
668     private String courses[];
669     private int runs[];
670     public String sayHello() {
671         return "Hello Friends... Gmg. :) "+
672             " salary = "+salary+
673             " tb = "+tb.toString()+
674             " dt = "+dt.toString()+
675             " fruits = "+fruits.toString()+
676             " veg = "+veg.toString()+

```

```
677         " phones = "+phone_nos.toString()+
678         " emails = "+emails.toString()+
679         " perDetails = "+perDetails.toString()+
680         " capitals = "+capitals.toString()+
681         " faculties = "+faculties.toString()+
682         " courses = {" +courses[0]+" "+courses[1]+"}" +
683         " runs = {" + runs[0]+" "+runs[1]+"}";
684     }
685     public void setSalary(float salary) {
686         this.salary = salary;
687     }
688     public void setTb(TestBean tb) {
689         this.tb = tb;
690     }
691     public void setDt(Date dt) {
692         this.dt = dt;
693     }
694     public void setFruits(List fruits) {
695         this.fruits = fruits;
696     }
697     public void setVeg(List veg) {
698         this.veg = veg;
699     }
700     public void setPhone_nos(Set phone_nos) {
701         this.phone_nos = phone_nos;
702     }
703     public void setEmails(Set emails) {
704         this.emails = emails;
705     }
706     public void setPerDetails(Map perDetails) {
707         this.perDetails = perDetails;
708     }
709     public void setCapitals(Map capitals) {
710         this.capitals = capitals;
711     }
712     public void setFaculties(Properties faculties) {
713         this.faculties = faculties;
714     }
715     public void setCourses(String[] courses) {
716         this.courses = courses;
717     }
718     public void setRuns(int[] runs) {
719         this.runs = runs;
720     }
721 }
```

722 //Note: Set doesn't allows duplicates but List allows

```
723 //spring.cfg.xml
724 <?xml version="1.0" encoding="UTF-8"?>
725 <beans
726     xmlns="http://www.springframework.org/schema/beans"
727     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
728     xsi:schemaLocation="http://www.springframework.org/schema/beans
729     http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
730     <bean id="dat" class="java.util.Date"/>
731     <bean id="tbean" class="TestBean">
732         <property name="msg">
733             <value>Hello</value>
734         </property>
735     </bean>
736     <bean id="db" class="DemoImple">
737         <property name="salary">
738             <value>5000</value>
739         </property>
740         <property name="tb">
741             <ref bean="tbean"/>
742         </property>
743         <property name="dt">
744             <ref bean="dat" />
745         </property>
746         <property name="fruits">
747             <list>
748                 <value>Apple</value>
749                 <value>Banana</value>
750                 <value>Orange</value>
751             </list>
752         </property>
753         <property name="veg">
754             <list>
755                 <value>Carrot</value>
756                 <value>Potato</value>
757                 <value>Mushrooms</value>
758                 <!-- any objects can be stored in a List-->
759                 <ref bean="dat"/>
760                 <ref bean="tbean"/>
761                 <value>Tomato</value>
762             </list>
763         </property>
764         <property name="phone_nos">
765             <set>
766                 <value>7569670669</value>
767                 <value>9502166767</value>
768             </set>
769         </property>
```

```
770         <property name="emails">
771             <set>
772                 <value>kanakavaraprasad@gmail.com</value>
773                 <value>s.varaprasad@yahoo.co.in</value>
774             </set>
775         </property>
776         <property name="perDetails">
777             <map>
778                 <entry>
779                     <key>
780                         <value>Sai</value>
781                     </key>
782
783                     <value>Universal King</value>
784                 </entry>
785
786                 <entry>
787                     <key>
788                         <value>Kanakadhar</value>
789                     </key>
790                     <value>Service Provider</value>
791                 </entry>
792             </map>
793         </property>
794         <property name="capitals">
795             <map>
796                 <entry>
797                     <key>
798                         <value>Sathya</value>
799                     </key>
800
801                     <value>Boon for learners</value>
802                 </entry>
803
804                 <entry>
805                     <key>
806                         <ref bean = "dat"/>
807                     </key>
808
809                     <value>Boon for learners</value>
810                 </entry>
811             </map>
812         </property>
813         <property name="faculties">
814             <props>
815                 <prop key="Kanakadhar">Java Faculty</prop>
816                 <prop key="Mahesh">.net Faculty</prop>
817             </props>
818         </property>
```

```
814 <!--Following both properties courses & runs are array variables. So we made use of list tag--  
815 >  
816     <property name="courses">  
817         <list>  
818             <value>Java</value>  
819             <value>.net</value>  
820             <value>Oracle</value>  
821         </list>  
822     </property>
```

```
823     <property name="runs">  
824         <list>  
825             <value>30</value>  
826             <value>40</value>  
827         </list>  
828     </property>  
829 </bean>
```

```
830 </beans>
```

831 **Note:** While configuring dependent values for collection framework type bean properties, we can
832 configure one data structure object as the element value of another data structure.

```
833 //TestBean .java
```

```
834 public class TestBean {  
835     String msg;
```

```
836     public void setMsg(String msg) {  
837         this.msg = msg;  
838     }  
839     public String toString()  
840     {  
841         return msg;  
842     }  
843 }
```

```
844 //DemoClient.java
```

```
845 import org.springframework.beans.factory.xml.XmlBeanFactory;  
846 import org.springframework.core.io.ClassPathResource;
```

```
847 public class DemoClient {  
848     public static void main(String[] args) {  
849         ClassPathResource res = new ClassPathResource("spring.cfg.xml");  
850         XmlBeanFactory factory = new XmlBeanFactory(res);  
851         DemoImple d1 = (DemoImple) factory.getBean("db");  
852         System.out.print(d1.sayHello());  
853     }  
854 }
```

```
855 Constructor injection:
```

- 856 1. If spring containers uses parameterized constructor to create spring bean class object and to
857 set values to bean properties as initial values then it is called as constructor injection.
- 858 2. Constructor injection supports injecting values to all types of bean properties.

- 859 3. If properties of bean class [few or all properties] are configured for constructor injection
860 then spring bean container creates spring bean class object by using parameterised
861 constructor.
- 862 4. If all properties of spring bean class are configured only for setter injection then spring
863 container creates spring bean class object by using zero parameterized constructor.
- 864 5. Using <constructor-arg> tag in spring configuration file the container performs constructor
865 injection on bean properties.
- 866 6. While configuring bean properties for constructor injection we must configure them in
867 spring configuration file in the same order or sequence as they are placed in the constructor
868 parameters.
- 869 7. On the same bean property of bean class if we configure both setter and constructor
870 injection then the value injected through setter injection gets activated because setter
871 methods gets executed after execution of constructor. So that the values of constructor
872 injection gets overridden with values of setter injection.
- 873 8. We can configure constructor argument values in 4 ways:
- 874 1. Sequentially as specified in order of constructor
- 875 2. By specifying datatype [here no two arguments should contain same data type]
- 876 3. Using index of constructor arguments. [index of arguments starts from 0 [zero]].
- 877 4. Combination of above 3.
- 878 9. Based on no. of <constructor-arg> tags specified in the bean class the same no. of
879 parameterised constructor will be called by spring container.
- 880 Eg:- If <constructor-arg> tag is placed for three times then three parameterised constructor
881 gets executed.
- 882 10. Constructor injection is quick than setter injection. Because constructor gets executed at the
883 time of object creation and setter methods gets executed after object creation.
- 884 **Constructor vs. Setter injection:** Use constructor injection when you have to inject one or few
885 properties and prefer setter injection when you have to inject more properties.
- 886 **Note:**
- 887 1. Its not mandatory that all the bean class properties to be configured for dependency
888 injection.
- 889 2. Maximum predefined classes supplied by spring are designed to support Setter injection.

890 **//Example:Constructor Injection**

891 **//Demo.java**

892 public class Demo

893 {

894 private String message;

895 public Demo() {} //dummy constructor

896 public Demo(String message) //one parameterised constructor

897 {

898 this.message = message;

899 }

```
900     public String toString()
901     {
902         return message;
903     }
904 }
```

905 **spring.cfg.xml**

```
906     <bean id="demo" class="Demo" >
907         <constructor-arg>
908             <value>Hello</value>
909         </constructor-arg>
910     </bean>
```

911 **Note:**

- 912 1. Here all the tags like <value>, <ref>, <props>, <list>, <set>, <map> will remain same. But
- 913 instead of using <property> tag we are making use of <constructor-arg> tag.
- 914 2. Client program is as usual.
- 915 3. As we are passing only one constructor argument in <bean> tag it invokes one
- 916 parameterized constructor.

917 **//Example: Invoking Multiple Parametersized constructor**

```
918 public class Bean
919 {
920     private int a;
921     private float b;
922     private String c;
923
924     public Bean() {}
925
926     public Bean(int a, float b, String c)
927     {
928         this.a = a;
929         this.b = b;
930         this.c = c;
931     }
932 }
```

```
930     public String toString()
931     {
932         return a + " " + b + " " + c;
933     }
934 }
```

935 **//spring.cfg.xml**

936 **//arguments in sequence**

```
937     <bean id = "be" class = "Bean">
938         <constructor-arg>
939             <value>1</value>
940         </constructor-arg>
941         <constructor-arg>
942             <value>1.1</value>
943         </constructor-arg>
```

```
944     <constructor-arg>
945         <value>Gmg</value>
946     </constructor-arg>
```

```
947 </bean>
```

```
948 //Note:
```

- ```
949 1. In above style we have passed all the values in the form of sequence.
950 2. It means first <constructor-arg> value "1" will be given to first paramter of constructor.
951 3. second <constructor-arg> value "1.1" will be given to second paramter of constructor.
952 4. third <constructor-arg> value "Gmg" will be given to third paramter of constructor.
```

```
953 //spring.cfg.xml
```

```
954 //constructor arguments by using data type
```

```
955 <bean id = "be" class = "Bean">
956 <constructor-arg type = "float" >
957 <value>1.1</value>
958 </constructor-arg>
959
960 <constructor-arg type = "java.lang.String" >
961 <value>Gmg</value>
962 </constructor-arg>
963
964 <constructor-arg type = "int" >
965 <value>1</value>
966 </constructor-arg>
967 </bean>
```

```
968 Note:
```

- ```
969     1. First argument of "float" data type will be given to float parameter of constructor
970     2. Scond argument of "java.lang.String" type will be given to String parameter of constructor
971     3. Third argument of "int" type will be given to int parameter of constructor.
972     4. But if the constructor has more than one parameter with same data type it will create error
973     message due to ambiguity [confusion].
```

```
974 //spring.cfg.xml
```

```
975 //constructor arguments by using index
```

```
976     <bean id = "be" class = "Bean">
977         <constructor-arg index = "1">
978             <value>1.1</value>
979         </constructor-arg>
980
981         <constructor-arg index = "2">
982             <value>Gmg</value>
983         </constructor-arg>
984
985         <constructor-arg index = "0">
986             <value>1</value>
987         </constructor-arg>
988     </bean>
```


985 **Note:**

- 986 1. Index always starts with 0 [zero]
- 987 2. 0th index argument value will be given to 1st parameter
- 988 3. 1st index argument value will be given to 2nd parameter
- 989 4. 2nd index argument value will be given to 3rd parameter

990 **//spring.cfg.xml**

991 **//constructor arguments by using sequence, data type & index combination**

```

992     <bean id = "be" class = "Bean">
993         <constructor-arg index = "0" type = "int" >
994             <value>1</value>
995         </constructor-arg>

996         <constructor-arg index = "1" type = "float">
997             <value>1.1</value>
998         </constructor-arg>

999         <constructor-arg index = "2" type = "java.lang.String">
1000             <value>Gmg</value>
1001         </constructor-arg>
1002     </bean>

```

1003 **Note:**

- 1004 1. We can make use of any of the above 4 styles.
- 1005 2. But combination of sequence, index & type is the best way of configuring parameters. It will
1006 avoid confusion & increases readability of program. Due to this maintenance of application
1007 will become easy.
- 1008 3. Client program is asusual.

1009 **Example: Constructor injection with different bean property types**

1010 **//DemoInter.java**

```

1011 public interface DemoInter {
1012     public String sayHello();
1013 }

```

1014 **//DemoImple.java**

```

1015 public class DemoImple implements DemoInter{
1016     private float salary;
1017     private TestBean tb = null;
1018     private Date dt = null;
1019     private List fruits = null;
1020     private List veg = null;
1021     private Set phone_nos = null;
1022     private Set emails = null;
1023     private Map perDetails = null;
1024     private Map capitals = null;
1025     private Properties faculties = null;
1026     private String courses[];
1027     private int runs[];

```

```

1028 public String sayHello() {
1029     return "Hello Friends... Gmg. :) "+
1030         " salary = "+salary+
1031         " tb = "+tb.toString()+
1032         " dt = "+dt.toString()+
1033         " fruits = "+fruits.toString()+
1034         " veg = "+veg.toString()+
1035         " phones = "+phone_nos.toString()+
1036         " emails = "+emails.toString()+
1037         " perDetails = "+perDetails.toString()+
1038         " capitals = "+capitals.toString()+
1039         " faculties = "+faculties.toString()+
1040         " courses = {" +courses[0]+" "+courses[1]+"}"+
1041         " runs = {" + runs[0]+" "+runs[1]+"}";
1042 }

1043 public DemoImple(float salary, TestBean tb, Date dt, List fruits, List veg,
1044                 Set phone_nos, Set emails, Map perDetails, Map capitals,
1045                 Properties faculties, String[] courses, int[] runs)
1046 {
1047     this.salary = salary;
1048     this.tb = tb;
1049     this.dt = dt;
1050     this.fruits = fruits;
1051     this.veg = veg;
1052     this.phone_nos = phone_nos;
1053     this.emails = emails;
1054     this.perDetails = perDetails;
1055     this.capitals = capitals;
1056     this.faculties = faculties;
1057     this.courses = courses;
1058     this.runs = runs;
1059 }
1060 public DemoImple() {} //dummy constructor
1061 }

```

1062 **//Note:**

- 1063 1. Same as the example of different property types with setter injection. There we made use of
- 1064 <property> tag. Now we are making use of <constructor-arg> tag. Internally body of the
- 1065 <property> tag or <constructor-arg> tag will remain same. The only change is with outer
- 1066 tag.
- 1067 2. Here we are following sequence while passing arguments to constructor.

1068 **//spring.cfg.xml**

```

1069 <?xml version="1.0" encoding="UTF-8"?>
1070 <beans
1071     xmlns="http://www.springframework.org/schema/beans"
1072     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1073     xsi:schemaLocation="http://www.springframework.org/schema/beans
1074     http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

```

```
1075 <bean id="dat" class="java.util.Date"/>
1076 <bean id="tbean" class="TestBean">
1077     <property name="msg">
1078         <value>Hello</value>
1079     </property>
1080 </bean>
1081 <bean id="db" class="DemoImple">
1082     <constructor-arg>
1083         <value>99.9</value>
1084     </constructor-arg>
1085     <constructor-arg>
1086         <ref bean="tbean"/>
1087     </constructor-arg>
1088     <constructor-arg>
1089         <ref bean="dat" />
1090     </constructor-arg>
1091     <constructor-arg>
1092         <list>
1093             <value>Apple</value>
1094             <value>Banana</value>
1095             <value>Orange</value>
1096         </list>
1097     </constructor-arg>
1098     <constructor-arg>
1099         <list>
1100             <value>Carrot</value>
1101             <value>Potato</value>
1102             <value>Mushrooms</value>
1103             <!-- any objects can be stored in a List-->
1104             <ref bean="dat"/>
1105             <ref bean="tbean"/>
1106             <value>Tomato</value>
1107         </list>
1108     </constructor-arg>
1109     <constructor-arg>
1110         <set>
1111             <value>7569670669</value>
1112             <value>9502166767</value>
1113         </set>
1114     </constructor-arg>
1115     <constructor-arg>
1116         <set>
1117             <value>kanakavaraprasad@gmail.com</value>
1118             <value>s.varaprasad@yahoo.co.in</value>
1119         </set>
1120     </constructor-arg>
```

```

1121         <constructor-arg>
1122             <map>
1123                 <entry>
1124                     <key>
1125                         <value>Sai</value>
1126                     </key>
1127
1128                     <value>Universal King</value>
1129                 </entry>
1130
1131                 <entry>
1132                     <key>
1133                         <value>Kanakadhar</value>
1134                     </key>
1135                     <value>Service Provider</value>
1136                 </entry>
1137             </map>
1138         </constructor-arg>
1139         <constructor-arg>
1140             <map>
1141                 <entry>
1142                     <key>
1143                         <value>Sathya</value>
1144                     </key>
1145
1146                     <value>Boon for learners</value>
1147                 </entry>
1148
1149                 <entry>
1150                     <key>
1151                         <ref bean = "dat"/>
1152                     </key>
1153
1154                     <ref bean = "tbean"/>
1155                 </entry>
1156             </map>
1157         </constructor-arg>
1158         <constructor-arg>
1159             <props>
1160                 <prop key="Kanakadhar">Java Faculty</prop>
1161                 <prop key="Mahesh">.net Faculty</prop>
1162             </props>
1163         </constructor-arg>
1164         <constructor-arg>
1165             <list>
1166                 <value>Java</value>
1167                 <value>.net</value>
1168                 <value>Oracle</value>
1169             </list>
1170         </constructor-arg>

```

```
1166         <constructor-arg>
1167             <list>
1168                 <value>30</value>
1169                 <value>40</value>
1170             </list>
1171         </constructor-arg>
1172     </bean>
```

```
1173 </beans>
```

1174 **Note:** Client program is asusual.

1175 **Factory method:** A method of a java class which is capable of constructing and returns its own
1176 class object.

1177 When spring bean class has private constructor we cannot create its object from outside. Then we
1178 can give instructions to spring container to create the object of bean class by using factory method.

1179 **Note:**

- 1180 1. Factory methods can be of static or not static [instance method]
- 1181 2. Factory method internally calls constructor at the time of object creation. No problem will
1182 be there even with private constructors. Because private constructors can be accessed from
1183 inside the class.
- 1184 3. Factory methods should be public methods. So that can be accessed from outside the class.

1185 **Eg of static factory methods:**

- 1186 1. Class c = Class.forName("ClassName");
- 1187 2. Thread t = Thread.currentThread();
- 1188 3. Calendar cl = Calendar.getInstance();

1189 **Eg. on not static factory method [insanmce method]**

```
1190 String s = new String("OK");
```

```
1191 String s1 = s.concat("Hello"); //instance factory method.
```

1192 **Instructions to spring container to call factory method:**

- 1193 1. We should use **factory-method** attribute of <bean> tag to configure the factory method to be
1194 called to create the object.
- 1195 2. If factory method expects any arguments then pass required arguments using <constructor-
1196 arg> tag.

1197 **Factory Class:** A class method which can return you another class object is called as Factory Class.

1198 **Eg:**

```
1199 Integer i1 = new Integer(100);
```

```
1200 String s1= i1.toString();
```

1201 **Note:** Here by using toString() of java.lang.Integer class we are getting java.lang.String. It is
1202 returning this object by creating immediately. So we can call Integer class as factory class.

1203 **Approaches to create spring bean class objects:**

- 1204 1. zero or parameterized constructor
- 1205 2. static factory method of bean class
- 1206 3. instance factory method of factory bean class

1207 **Example: Static Factory Method**1208 **//DemoInter.java**

1209 public interface DemoInter

1210 {

1211 public String sayHello();

1212 }

1213 **// DemoBean.java**

1214 import java.util.Calendar;

1215 **public class DemoBean implements DemoInter**

1216 {

1217 private Calendar cl = null;

1218 private TestBean tb = null;

1219 public DemoBean() { }

1220 public void setCl(Calendar cl) {

1221 this.cl = cl;

1222 }

1223 public void setTb(TestBean tb) {

1224 this.tb = tb;

1225 }

1226 public String sayHello() {

1227 return "tb = "+tb.toString()+" cl = "+cl.toString();

1228 }

1229 }

1230 **//TestBean.java**

1231 public class TestBean {

1232 String msg;

1233 public TestBean() {}

1234 private TestBean(String msg) {

1235 this.msg = msg;

1236 System.out.println("Private Constructor invoked");

1237 }

```
1238 //static factory method
1239 public static TestBean getTestBean(String msg)
1240 {
1241     System.out.println("static factory method invoked");
1242
1243     return new TestBean(msg);
1244 }
1245
1246 public String toString() //to display results
1247 {
1248     return msg;
1249 }
1250 }
1251 //spring.cfg.xml
1252 <?xml version="1.0" encoding="UTF-8"?>
1253 <beans
1254     xmlns="http://www.springframework.org/schema/beans"
1255     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1256     xsi:schemaLocation="http://www.springframework.org/schema/beans
1257     http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
1258
1259     <bean id = "cal" class = "java.util.Calendar" factory-method = "getInstance"/>
1260
1261     <bean id = "t1" class = "TestBean" factory-method = "getTestBean">
1262         <constructor-arg> <!-- passing argument to factory method -->
1263             <value>Gmg. my dear friends...</value>
1264         </constructor-arg>
1265     </bean>
1266
1267     <bean id = "db" class = "DemoBean">
1268         <property name = "cl">
1269             <ref bean = "cal"/>
1270         </property>
1271
1272         <property name = "tb">
1273             <ref bean = "t1"/>
1274         </property>
1275     </bean>
1276 </beans>
1277 //DemoClient.java
1278 public class DemoClient {
1279
1280     public static void main(String[] args) {
1281         ClassPathResource res = new ClassPathResource("spring.cfg.xml");
1282         XmlBeanFactory factory = new XmlBeanFactory(res);
1283         DemoBean d1 = (DemoBean) factory.getBean("db");
1284         System.out.print(d1.sayHello());
1285     }
1286 }
```

1280 **Note:**

- 1281 1. The spring container uses factory method getInstance() to create bean class object for
1282 java.util.Calendar.
- 1283 2. Spring container uses "getTestBean(String)" as factory method to create test bean class
1284 object.
- 1285 3. Spring container uses zero parameterised constructor to create DemoBean class object.

1286 **//Example: Instance Factory Method**

1287 **//DemoInter .java**

```
1288 public interface DemoInter {  
1289     public String sayHello();  
1290 }
```

1291 **// DemoBean.java**

1292 **public class DemoBean implements DemoInter{**

1293 **private TestBean tb = null;**

```
1294     public DemoBean() {  
1295         System.out.println("zero parametrized constructor of DemoBean class");  
1296     }
```

```
1297     public void setTb(TestBean tb) {  
1298         this.tb = tb;  
1299     }
```

```
1300     public String sayHello() {  
1301         return "tb = "+tb.toString();  
1302     }
```

1303 **//instance factory method**

```
1304     public TestBean getTestBean(String msg)  
1305     {  
1306         System.out.println("demo bean factory method of DemoBean class");  
1307         return new TestBean(msg);  
1308     }  
1309 }
```

1310 **//TestBean.java**

```
1311 public class TestBean {  
1312     String msg;
```

1313 public TestBean() {} //dummy constructor

```
1314     TestBean(String msg) {  
1315         this.msg = msg;  
1316         System.out.println("Constructor of TestBean class invoked");  
1317     }
```



```

1318     public String toString() //to display results
1319     {
1320         return msg;
1321     }
1322 }

```

1323 **//spring.cfg.xml**

```
1324 <?xml version="1.0" encoding="UTF-8"?>
```

```
1325 <beans
```

```
1326     xmlns="http://www.springframework.org/schema/beans"
```

```
1327     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
1328     xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
1329     http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
```

```
1330     <bean id = "t1" factory-bean = "db" factory-method = "getTestBean">
```

```
1331         <constructor-arg> <!-- passing argument to factory method -->
```

```
1332             <value>Gmg. my dear friends...</value>
```

```
1333         </constructor-arg>
```

```
1334     </bean>
```

```
1335     <bean id = "db" class = "DemoBean">
```

```
1336         <property name = "tb">
```

```
1337             <ref bean = "t1"/>
```

```
1338         </property>
```

```
1339     </bean>
```

```
1340 </beans>
```

1341 **//DemoClient .java**

```
1342 public class DemoClient {
```

```
1343     public static void main(String[] args) {
```

```
1344         ClassPathResource res = new ClassPathResource("spring.cfg.xml");
```

```
1345         XmlBeanFactory factory = new XmlBeanFactory(res);
```

```
1346         DemoBean d1 = (DemoBean) factory.getBean("db");
```

```
1347         System.out.print(d1.sayHello());
```

```
1348     }
```

```
1349 }
```

1350 **Note:**

- 1351 1. Making spring container to create spring bean class object through non static methods
1352 [instance methods].
- 1353 2. While configuring spring bean class if class attribute is not there and factory-bean, factory-
1354 method attributes are there then spring container uses instance factory method of specified
1355 factory bean class to create spring bean class object.

1356 **Application Context container** is enhancement of bean factory container with some advantages.

1357 Its features are as follows:

- 1358 1. Pre initialization of beans by default. [Early loading]. It means in our spring configuration
1359 file we may have any no. of bean tags. Objects will be created sequentially for each and
1360 every bean class as specified in the sequence of <bean> tags in xml file.
- 1361 2. Ability to read values of bean properties from properties file.

- 1362 3. supports Internationalization (I18n)
- 1363 4. Gives the ability to work with events & listeners.
- 1364 5. `org.springframework.context.ApplicationContext` interface is the sub interface of
- 1365 `org.springframework.beans.factory.BeanFactory` interface.
- 1366 6. Activating application context interface is nothing but creating object of a class that
- 1367 implements `org.springframework.context.ApplicationContext` interface.
- 1368 7. There are three regularly used implementation classes of application context interface. By
- 1369 creating object for any of these three classes we can activate `ApplicationContext` container.
- 1370 8. This container is available in JEE module.
- 1371 9. **`org.springframework.context.support.FileSystemXmlApplicationContext`**: It activates
- 1372 application context container by locating given spring configuration file in the specified
- 1373 path.
- 1374 **Ex:** `FileSystemXmlApplicationContext ctx = new`
- 1375 `FileSystemXmlApplicationContext("c:\fl\spring.cfg.xml")`
- 1376 10. **`org.springframework.context.support.ClassPathXmlApplicationContext`**: It activates
- 1377 application context container by locating spring configuration file in the same working
- 1378 directory or from jar files added in the classpath.
- 1379 **Ex:** `ClassPathXmlApplicationContext ctx = new`
- 1380 `ClassPathXmlApplicationContext("spring.cfg.xml");`
- 1381 11. **`org.springframework.context.support.XmlWebApplicationContext`**: this class activates
- 1382 application context container by locating spring configuration file in deployment directory
- 1383 structure of web application by default in `WEB-INF` folder.
- 1384 **Ex:** `XmlWebApplicationContext ctx = new XmlWebApplicationContext("spring.cfg.xml");`
- 1385 12. In real time applications we can find regularly working with Application Context container
- 1386 rather than working with Bean Factory container.
- 1387 13. Application context container can perform all modes of dependency injection like bean
- 1388 factory container.
- 1389 14. Application context container performs pre instantiation on all singleton scoped on spring
- 1390 bean classes of spring configuration file at the moment of application context container gets
- 1391 activated.
- 1392 15. Pre-instantiation means creating spring bean class objects immediately after the application
- 1393 context container activation.
- 1394 16. BeanFactory cannot perform this pre instantiation on spring bean classes.
- 1395 17. when we call `factory.getBean()` then bean factory container immediately creates the object
- 1396 and uses the same for further requests on same id-value.
- 1397 18. When we call `ctx.getBean()` then application context container gives access to bean id
- 1398 related spring bean class object which is created at pre-instantiation process.
- 1399 19. Application context container can perform pre-instantiation only on singleton scoped spring
- 1400 bean classes that are configured in spring configuration file.
- 1401 20. If we give scope value as prototype then object will not be created for bean at the time of
- 1402 pre-instantiation.

- 1403 21.If the object is not created at pre-instantiation then it will be created after receiving the call
1404 along with its dependent objects.
- 1405 22.If singleton scope bean class property has prototype scoped bean class object as dependent
1406 value then the application context container also creates prototype scoped spring bean class
1407 object during pre instantiation process along with singleton scope bean class object.
- 1408 23.It happens to satisfy dependency injection needs done on singleton scope bean class
1409 properties.

1410 **Example: Creation of our own singleton java class.**

1411 //Designing our own singleton java class

1412 **//Test.java**

1413 public class Test

1414 {

1415 private static Test t = null;

1416 //private constructor-so no one can create object from outside the class

1417 private Test()

1418 {

1419 System.out.println("Private constructor of Test class invoked");

1420 }

1421 //static factory method

1422 public static Test getTest()

1423 {

1424 //singleton logic

1425 if(t == null)

1426 t = new Test();

1427 return t;

1428 }

1429 }

1430 **Note:** Also override clone() method of super class Object and throw CloneNotSupportedException.

1431 **//Client.java**

1432 class Client

1433 {

1434 public static void main(String[] args)

1435 {

1436 Test t1 = Test.getTest();

1437 Test t2 = Test.getTest();

1438 Test t3 = Test.getTest();

1439 System.out.println(t1);

1440 System.out.println(t2);

1441 System.out.println(t3);

1442 }

1443 }

1444 **Note:** We have requested the same class object 3 times. But only one object gets created & the same
1445 object will be given for all 3 times.

1446 **Scopes of Bean class objects under ApplicationContext container:**

1447 We can make spring container to keep the created spring bean objects in different scopes. They are:

1448 1. singleton [**default scope**]

1449 2. prototype

1450 3. request

1451 4. session

1452 5. global session

1453 **singleton:** container makes spring bean class object as **shared object for multiple calls** given to
1454 factory.getBean() with bean id.

1455 **Eg:**

1456 <bean id = “bd” class = “BeanDemo” **scope = “singleton”**>

1457 The client application may call factory.getBean(“bd”) for multiple times. Still the spring container
1458 creates and returns only one DemoBean class object.

1459 Singleton java class means the java class that allows to create only one object per jvm. It means the
1460 class has restriction related logic.

1461 **Note:** In the above example **BeanDemo** class need not to be designed as singleton java class. But
1462 container creates only one object for DemoBean class. Here it’s the responsibility of spring
1463 container.

1464 When spring bean scope is singleton the spring container never makes spring bean class as
1465 singleton java class but spring container restricts itself and creates only one object for spring bean
1466 class though factory.getBean(id) is called for multiple times.

1467 **Prototype:** It gives an independent spring bean class object for each call to factory.getBean(id); **Eg:**

1468 <bean id = “db” class = “DemoBean” scope = “prototype”>

1469 “request”, “session”, “global session” are used for web applications.

1470 **request:** Only one object will be used from beginning to end of the same request.

1471 **session:** from beginning to expiry of same session single object will be used.

1472 **global session:** Its commonly shared by the entire web application.

1473 **Note:** When we give setter injection we should not give only parameterized constructor in the class.

1474 Because when no <constructor-arg> tags are available in spring configuration file then spring
1475 container tries to create object by using default [zero parameterized] constructor. If its not available
1476 then immediately error arises. When no constructors (zero or parameterized) are available in the
1477 class then automatically compiler creates a default and zero parameterized constructor.

1478 **//Example: First application with ApplicationContext or JEE or Advanced Container**

1479 **//Client steps:**

- 1480 1. Locate Spring-Configuration file + Activate ApplicationContext Container + Read Spring-
1481 Configuration file
- 1482 2. Request object of business-logic class.
- 1483 3. Call business-logic methods.
- 1484 4. Close container

1485 **//DemoInter.java**

1486 public interface DemoInter

1487 {

1488 public String wish(String uname);

1489 }

1490 **//DemoInterImpl.java**

1491 **public class DemoInterImpl implements DemoInter**

1492 {

1493 private String message;

1494 public DemoInterImpl()

1495 {

1496 System.out.println("Constructor executed");

1497 }

1498 public void setMessag(String s)

1499 {

1500 message=s;

1501 }

1502 public String wish(String uname)

1503 {

1504 return message+" "+uname+" Have a great day";

1505 }

1506 }

1507 //spring.cfg.xml

1508 <?xml version="1.0" encoding="UTF-8"?>

1509 <beans

1510 xmlns="http://www.springframework.org/schema/beans"

1511 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

1512 xsi:schemaLocation="http://www.springframework.org/schema/beans

1513 http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

1514 <bean id="demo" class="DemoInterImpl" >

1515 <property name="messag">

1516 <value>Hello</value>

1517 </property>

1518 </bean>

1519 </beans>

1520 //DemoClient.java

1521 import org.springframework.context.support.ClassPathXmlApplicationContext;

1522 public class DemoClient

1523 {

1524 public static void main(String args[])

1525 {

1526 ClassPathXmlApplicationContext ctx = new

1527 ClassPathXmlApplicationContext("spring.cfg.xml");

1528 DemoInter d1=(DemoInter)ctx.getBean("demo");

1529 DemoInterImpl d2=(DemoInterImpl)ctx.getBean("demo");

1530 System.out.println(d1.wish("Sai"));

1531 ctx.close();

1532 }

1533 }

1534 **Note:** We are requesting the same class object twice. But container creates only one object & gives
1535 the same object for every request.

1536 **Example: Prototype scope**

1537 **Note:**

1538 1. Same as above example. We are giving a new attribute for our <bean> tag as shown below
1539 in spring configuration file.

1540 2. If we donot specify any scope by default the scope will be **singleton**.

1541 3. If the scope is prototype:

- 1542 1. for every request a separate object will be created.
- 1543 2. Without receiving the request object will not be created by container.

```

1544 <bean id="demo" class="DemoInterImpl" scope="prototype">
1545     <property name="messag">
1546         <value>Hello friends... Gmg. :)</value>
1547     </property>
1548 </bean>

```

1549 **Interface Injection:**

- 1550 1. If spring container performs dependency injection on bean properties by implementing the
1551 XxxAware interfaces then it is called as interface injection.
- 1552 2. All these XxxAware interfaces are predefined interfaces supplied by spring api on one bean
1553 spring bean class property.
- 1554 3. We can perform all 3 types of dependency injection.
- 1555 4. We can't use interface injection to inject our own values.
- 1556 5. Interface injection can perform special values injection given by spring container.

1557 **Predefined interfaces used for dependency injection:**

- 1558 1. org.springframework.beans.factory.BeanNameAware interface makes spring bean class
1559 getting its bean id given in spring configuration file through interface injection.
- 1560 2. org.springframework.beans.factory.BeanFactoryAware interface injects the current
1561 underlying bean factory container to spring bean class through interface injection.
- 1562 3. org.springframework.context.ApplicationContextAware injects the current underlying
1563 application context container to spring bean class.

1564 **Note:**

- 1565 1. While performing interface injection on spring bean class the spring bean class can't act as
1566 "pojo class".
- 1567 2. Interface injection is useful to make spring bean to know about itself and other spring beans
1568 managed by the container.

1569 **Example: Interface Injection**

1570 **//Demo.java**

```

1571 public interface Demo {
1572     String hi();
1573 }

```

1574 **//DemoBean.java**

```

1575 import org.springframework.beans.BeansException;
1576 import org.springframework.beans.factory.BeanNameAware;
1577 import org.springframework.context.ApplicationContext;
1578 import org.springframework.context.ApplicationContextAware;

```

```
1579 public class DemoBean implements Demo, BeanNameAware, ApplicationContextAware{
1580     String msg, bname;
1581     ApplicationContext ctx = null;
1582     public void setMsg(String msg) {
1583         this.msg = msg;
1584     }
1585     public void setApplicationContext(ApplicationContext ctx)
1586         throws BeansException {
1587         this.ctx = ctx;
1588     }
1589     public void setBeanName(String bname) {
1590         this.bname = bname;
1591     }
1592     public String hi() {
1593         System.out.println("Logical current bean name is: "+bname);
1594         System.out.println("No. of beans managed by underlying container is:
1595 "+ctx.getBeanDefinitionCount());
1596         System.out.println("Current bean is singleton: "+ctx.isSingleton(bname));
1597         System.out.println("Current bean is proptotype bean: "+ctx.isPrototype(bname));
1598         return msg;
1599     }
1600 }
1601 // applicationContext.xml
1602 <?xml version="1.0" encoding="UTF-8"?>
1603 <beans xmlns="http://www.springframework.org/schema/beans"
1604     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1605     xsi:schemaLocation="http://www.springframework.org/schema/beans
1606 http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
1607     <bean id = "db" class = "DemoBean">
1608         <property name="msg">
1609             <value>Hi Friends... Gmg. Njoy Spring concepts n coding. :)</value>
1610         </property>
1611     </bean>
1612     <bean id = "dt" class = "java.util.Date" scope = "prototype"/>
1613     <bean id = "dt1" class = "java.util.Date"/>
1614 </beans>
```


1615 //DemoClient.java

1616 import org.springframework.context.support.ClassPathXmlApplicationContext;

1617 import java.util.Date;

1618 public class DemoClient

1619 {

1620 public static void main(String args[])

1621 {

1622 ClassPathXmlApplicationContext ctx = new

1623 ClassPathXmlApplicationContext("applicationContext.xml");

1624 Demo d1=(Demo)ctx.getBean("db");

1625 Date d2=(Date)ctx.getBean("dt");

1626 //DemoImpl d2=(DemoImpl)ctx.getBean("db");

1627 System.out.println(d1.hi());

1628 System.out.println(d2);

1629 }

1630 }

1631 **Reading Properties File:** Place holder represents bean property value that should be collected from
1632 properties file based on the given key name dynamically at runtime of the program.

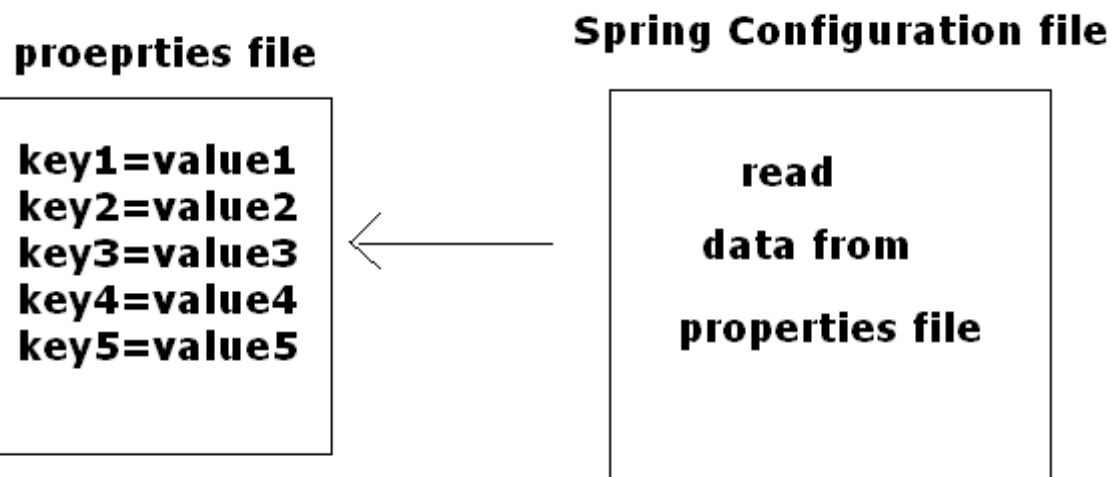
1633 **Purpose of properties file in spring application:** properties files and xml files are given to make
1634 java application code as flexible by passing input values of application from outside the application.

1635 **Note:**

- 1636 1. Work with properties files only when our spring application is forced to work with existing
1637 properties file. Don't create new properties files because the same flexibility can be
1638 achieved with xml file (spring configuration file) also.
- 1639 2. The standard principle of s/w industry is "don't hard code any values that can be modified in
1640 future or changed for regular intervals".

1641 **Eg:** jdbc driver class name, url, username, password etc.,

1642 **Example: Reading a values from properties file & writing to spring configuration file.**



1643 **Note:** In properties file we can specify comments by using # or !

1644 **//data.properties**

1645 #demo.properties

1646 our.name = Dharani

1647 our.age = 25

1648 our.address = Hyd

1649 our.email = dharani@gmail.com

1650 **//DemoBean.java**

1651 public class DemoBean {

1652 String name, email, address;

1653 int age;

1654 public void setName(String name) {

1655 this.name = name;

1656 }

1657 public void setEmail(String email) {

1658 this.email = email;

1659 }

1660 public void setAddress(String address) {

1661 this.address = address;

1662 }

1663 public void setAge(int age) {

1664 this.age = age;

1665 }

```
1666     public String hi()
1667     {
1668         return name+" "+age+" "+email+" "+address;
1669     }
1670 }
1671 //applicationContext.xml
1672 <?xml version="1.0" encoding="UTF-8"?>
1673 <beans
1674     xmlns="http://www.springframework.org/schema/beans"
1675     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1676     xsi:schemaLocation="http://www.springframework.org/schema/beans
1677 http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
1678
1679     <bean id = "propConfig" class =
1680         "org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
1681         <property name="location" value = "pf\demo.properties">
1682             <!-- <value>demo.properties</value> -->
1683         </property>
1684     </bean>
1685
1686     <bean id = "db" class = "DemoBean">
1687         <property name="name">
1688             <value>${our.name}</value>
1689         </property>
1690
1691         <property name="email">
1692             <value>${our.email}</value>
1693         </property>
1694
1695         <property name="address">
1696             <value>${our.address}</value>
1697         </property>
1698
1699         <property name="age">
1700             <value>${our.age}</value>
1701         </property>
1702     </bean>
1703 </beans>
```

1699 //DemoClient.java

1700 import org.springframework.context.support.ClassPathXmlApplicationContext;

1701 public class DemoClient

1702 {

1703 public static void main(String args[])

1704 {

1705 ClassPathXmlApplicationContext ctx = new

1706 ClassPathXmlApplicationContext("applicationContext.xml");

1707 DemoBean d = (DemoBean) ctx.getBean("db");

1708 System.out.println(d.hi());

1709 }

1710 }

1711 **Example: Reading data from multiple properties files**

1712 **#pf1.properties**

1713 name = Dharani

1714 age = 25

1715 **#pf2.properties**

1716 address = Hyd

1717 email = dharani@gmail.com

1718 name = Sathya

1719 **Note:**

1720 1. "name" property is there in both properties files. So first property file value will be replaces
1721 with second property file value.

1722 2. DemoBean.java & DemoClient.java files are same as above example.

1723 **//applicationContext.xml**

1724 <bean id = "propConfig" class =

1725 "org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

1726 <property name="locations">

1727 <list>

1728 <value>pf1.properties</value>

1729 <value>pf2.properties</value>

1730 </list>

1731 </property>

1732 </bean>

```
1733 <bean id = "db" class = "DemoBean">
1734     <property name="name">
1735         <value>${name}</value>
1736     </property>
1737     <property name="email">
1738         <value>${email}</value>
1739     </property>
1740     <property name="address">
1741         <value>${address}</value>
1742     </property>
1743     <property name="age">
1744         <value>${age}</value>
1745     </property>
1746 </bean>
```

1747 **Example: Reading multiple configuration files**

1748 **//applicationContext1.xml**

```
1749 <beans>
1750     <bean id = "t1" class = "TestBean">
1751         <property name="msg">
1752             <value>Hello friends Gmg.</value>
1753         </property>
1754     </bean>
1755 </beans>
```

1756 **//applicationContext2.xml**

```
1757 <beans>
1758     <bean id = "d1" class = "DemoBean">
1759         <property name="tb">
1760             <ref bean = "t1"/>
1761         </property>
1762     </bean>
1763 </beans>
```

1764 **Note:** "applicationContext2.xml" is reading identification name of "applicationContext1.xml" file.
1765 It means we may have any number of configuration files for the purpose of easy organization, still
1766 programatically we can consider them as a single file.

```
1767 //DemoBean.java
1768 public class DemoBean {
1769     TestBean tb = null;
1770     public void setTb(TestBean tb) {
1771         this.tb = tb;
1772     }
1773     public String toString()
1774     {
1775         return tb.msg;
1776     }
1777 }
1778 // TestBean.java
1779 public class TestBean {
1780     String msg;
1781     public void setMsg(String msg) {
1782         this.msg = msg;
1783     }
1784     public String toString()
1785     {
1786         return msg;
1787     }
1788 }
1789 //DemoClient.java
1790 import org.springframework.context.support.ClassPathXmlApplicationContext;

1791 public class DemoClient
1792 {
1793     public static void main(String args[])
1794     {
1795         String cfg[] = {"applicationContext2.xml", "applicationContext1.xml"};
1796         ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext(cfg);
1797         DemoBean d = (DemoBean) ctx.getBean("d1");
1798         System.out.println(d);
1799     }
1800 }
```

1801 **Internationalization (I18n):**

1802 1. Making our application projects working for multiple locales with flexibility in presentation
1803 logic to pass display labels of input fields or any information from outside the application is
1804 called applying **I18n** on the project.

1805 2. The project i.e., enabled with I18n can give service to multiple client organizations which
1806 may belong to multiple communities without changing source code.

1807 Locale = language + country

1808 **Examples for Language code & country code:**

1809 ○ En-us [English – United States]

1810 ○ En-au [English – Australia]

1811 ○ En-ca [English – Canada]

1812 ○ En-ie [English Ireland]

1813 ○ En-nz [English new zealand]

1814 ○ En-zw [English – Zimbabwe]

1815 ○ En-gb [English – United Kingdom]

1816 ○ Fr-be [French – Belgium]

1817 ○ Fr-ca [French – Canada]

1818 ○ Fr [French france]

1819 ○ Fr-ch [French Switzerland]

1820 3. All the above locales codes are given by “ISO” we can collect them from “IE” [Internet
1821 explorer] browser window settings as follows: Tools à Internet options à languages à Add [it
1822 shows you multiple countries with multiple languages. So we can choose what we need for
1823 our applications]

1824 4. Here we write a separate properties file for each language and we furnish all the labels in it
1825 for display of labels based on internet options of browser.

1826 5. In java application we use Locale class & ResourceBundle class of java.util package to
1827 apply the effect of I18n.

1828 6. While preparing multiple locale specific properties files the base file must be there because
1829 when no matching file is available then values will be taken from this **base file**.

1830 7. Org.springframework.conext.support.**ResouceBundleMessageSource** class makes the
1831 underlying application context container recognizing multiple locale properties files on I18n
1832 that are given to the application.

1833 8. In order to make the container to get message from properties file we need to call
1834 getMessage() method on the container object.

1835 9. For **getMessage()** method we have to pass 4 argumetns. They are:

1836 1. key in properties file

1837 2. java.lang.Object class array to pass argument values to place holders available in
1838 property values. If place holders are not there then we can pass “null”.as a value.

1839 3. **Default message:** when message from given key is not collectable default message will
1840 be displayed.

1841 4. **java.util.Locale** class object with language and country code.

1842 **Example: Spring Internationalization**

1843 **#app.preoperties [base file]**

1844 str1 = save - {0}english {1}

1845 str2 = clear - english

1846 str3 = delete - english

1847 str4 = cancel - english

1848 **Note:**

1849 1. "Str1" property has two place holders.

1850 2. Every property value can have place holders

1851 3. Always place holders starts with zero.

1852 4. For every property value place holder values can be supplied through string array variable.

1853 **#app_en_ca.preoperties [english canada]**

1854 str1 = save - {0} english {1} canada

1855 str2 = clear - english canada

1856 str3 = delete - english canada

1857 str4 = cancel - english canada

1858 **#app_en_gh.preoperties [uk]**

1859 str1 = save -{0} english {1} uk

1860 str2 = clear - english uk

1861 str3 = delete - english uk

1862 str4 = cancel - english uk

1863 **#app.preoperties [french belgium]**

1864 str1 = save - {0} french {1} belgium

1865 str2 = clear - french belgium

1866 str3 = delete - french belgium

1867 str4 = cancel - french belgium

1868 **//applicationContext.xml**

1869 <?xml version="1.0" encoding="UTF-8"?>

1870 <beans

1871 xmlns="http://www.springframework.org/schema/beans"

1872 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

1873 xsi:schemaLocation="http://www.springframework.org/schema/beans

1874 http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">


```
1875 <bean id = "messageSource" class =  
1876 "org.springframework.context.support.ResourceBundleMessageSource">  
1877     <property name="basename">  
1878         <value>app</value>  
1879     </property>  
1880 </bean>  
1881 </beans>
```

```
1882 //I18nApp.java
```

```
1883 import java.util.Locale;  
1884 import org.springframework.context.support.ClassPathXmlApplicationContext;  
1885 public class I18nApp {  
1886     public static void main(String args[])throws Exception  
1887     {  
1888         Locale l = new Locale(args[0], args[1]);  
1889         ClassPathXmlApplicationContext ctx = new  
1890         ClassPathXmlApplicationContext("applicationContext.xml");  
1891         String msg = ctx.getMessage("str1", new String[] {"hi", "Sai"}, "default msg", l);  
1892         msg += " " + ctx.getMessage("str2", null, "default msg", l);  
1893         msg += " " + ctx.getMessage("str3", null, "default msg", l);  
1894         msg += " " + ctx.getMessage("str4", null, "default msg", l);  
1895         System.out.println("Message is: "+msg);  
1896     }  
1897 }
```

1898 **Event Handling:** Event is specific action performed on the component or object. To handle events
1899 and execute same logic every time we can depend upon listeners. These listeners provide event
1900 handling methods to handle the events.

1901 All events are objects and all listeners are interfaces in java environment

1902 **In order to perform event handling four details are required:**

- 1903 1. Source object [Eg: Button component]
- 1904 2. Event [Eg: ActionEvent]
- 1905 3. Listener [Eg: ActionListener]
- 1906 4. Event handling methods [Eg: public void actionPerformed()]

1907 In spring environment event handling is given to notify when the spring container is activated or
1908 deactivated in spring based applications or other java applications where spring is also used.

1909 This event handling helps the programmer to notice how much time the spring related business
1910 logic is executed in the project.

1911 To perform event handling on spring application we need the following details:

1912 1.source object [application context container]

1913 2.event [application event]

1914 3. listener [application listener]

1915 4.event handling methods on application event

1916 **ApplicationContext container supports 3 events:**

1917 1. ContextRefreshedEvent

1918 2. RequestHandledEvent[Supported only in web applications. It gets activated for every
1919 request]

1920 3. ContextClosedEvent

1921 **Note:**

1922 1. Event handling on spring container can be performed without touching existing java source
1923 code of application.

1924 2. For more information on this refer 3.8.3 Events topic of spring PDF file.

1925 **Example: Event Handling**

1926 public interface DemoInter

1927 {

1928 public String wish(String uname);

1929 }

1930 **public class DemoInterImpl implements DemoInter**

1931 {

1932 private String message;

1933 public DemoInterImpl()

1934 {

1935 System.out.println("Constructor executed");

1936 }

1937 public void setMessag(String s)

1938 {

1939 message=s;

1940 }

1941 public String wish(String uname)

1942 {

1943 return message+" "+uname+" Have a great day";

1944 }

1945 }

```
1946 // Helper.java
1947 import java.util.Date;
1948 import org.springframework.context.ApplicationEvent;
1949 import org.springframework.context.ApplicationListener;
1950 public class Helper implements ApplicationListener {
1951     long startTime, endTime;
1952     public void onApplicationEvent(ApplicationEvent ae)
1953     {
1954         System.out.println("control in onApplicationEvent method");
1955         System.out.println(ae.toString());
1956         int index = ae.toString().indexOf("ContextRefreshedEvent");
1957         System.out.println("Index is: "+index);
1958         if(index != -1)
1959         {
1960             startTime = System.currentTimeMillis();
1961             System.out.println("Container started at "+new Date().toString());
1962         }
1963         else if(ae.toString().indexOf("ContextClosedEvent") != -1)
1964         {
1965             endTime = System.currentTimeMillis();
1966             System.out.println("Container is Closed at "+new Date().toString());
1967             System.out.println("Container is in activated mode for "+(endTime -
1968 startTime)+" milli seconds");
1969         }
1970     }
1971 }
1972 //spring.cfg.xml
1973 <?xml version="1.0" encoding="UTF-8"?>
1974 <beans
1975     xmlns="http://www.springframework.org/schema/beans"
1976     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1977     xsi:schemaLocation="http://www.springframework.org/schema/beans
1978 http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
1979     <bean id = "help" class = "Helper"/>
1980     <bean id="demo" class="DemoInterImpl" >
1981         <property name="messag">
```

```
1982         <value>Hello</value>
```

```
1983     </property>
```

```
1984 </bean>
```

```
1985 </beans>
```

```
1986 //DemoClient.java
```

```
1987 import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
1988 public class DemoClient
```

```
1989 {
```

```
1990     public static void main(String args[])throws Exception
```

```
1991     {
```

```
1992         ClassPathXmlApplicationContext ctx = new
```

```
1993         ClassPathXmlApplicationContext("spring.cfg.xml");
```

```
1994         DemoInter d1=(DemoInter)ctx.getBean("demo");
```

```
1995         //DemoInterImpl d2=(DemoInterImpl)ctx.getBean("demo");
```

```
1996         System.out.println(d1.wish("Sai"));
```

```
1997         Thread.sleep(2000);
```

```
1998         ctx.close(); //container gets closed now
```

```
1999     }
```

```
2000 }
```

```
2001 User defined life cycle methods: Spring bean class allows configuring our user defined methods to  
2002 act like life cycle methods.
```

```
2003     1. init () life cycle method: container calls this method automatically after dependency  
2004        injection process is completed on bean properties.
```

```
2005     2. destroy() life cycle method: Container calls this method just before destruction of spring  
2006        bean class object created by spring container.
```

```
2007 We can configure these life cycle methods in two ways:
```

```
2008     1. Programmatic statements [by implementing the following interfaces]
```

```
2009         1. InitializingBean
```

```
2010             1. afterPropertiesSet()
```

```
2011         2. DisposableBean
```

```
2012             1. destroy()
```

```
2013     2. Declarative statements
```

```
2014         1. <bean> attributes
```

```
2015             1. init-method
```

```
2016             2. destroy-method
```

2017 As these life cycle methods are user defined methods their names should be configured during
2018 spring bean class configuration in spring configuration file by using **init-method** and **destroy-**
2019 **method** attributes of **<bean>** tag.

2020 **Note:** We can give any names for these life cycle methods. But init & destroy are standard names.
2021 Any way what ever the name it is, it's mandatory to configure them in configuration file.

2022 Init() method is useful to check whether appropriate values were injected to spring bean properties
2023 are not through dependency injection.

2024 destroy() life cycle method is useful to nullify bean properties and to release non java resources
2025 associated with bean properties when spring bean class object is about to destroy.

2026 **Example: Programmatic initialization method**

2027 **import org.springframework.beans.factory.InitializingBean;**

2028 **public class Init implements InitializingBean{**

2029 int p1;

2030 public Init()

2031 {

2032 System.out.println("In constructor");

2033 }

2034 public void setP1(int p1)

2035 {

2036 System.out.println("Insetter method");

2037 this.p1 = p1;

2038 }

2039 public void afterPropertiesSet() throws Exception

2040 {

2041 System.out.println("in after proeprties set "+p1);

2042 }

2043 }

2044 **//applicationContext.xml**

2045 **<beans>**

2046 **<bean id="in" class="Init"> <!-- init-method = "init" -->**

2047 **<property name = "p1" value = "10"/>**

2048 **</bean>**

2049 **</beans>**

2050 **//DemoClient.java**

2051 **import org.springframework.context.support.ClassPathXmlApplicationContext;**

2052 **public class DemoClient {**

2053 **public static void main(String[] args) {**

```
2054         ClassPathXmlApplicationContext ctx = new
2055         ClassPathXmlApplicationContext("applicationContext.xml");
2056         Init i1 = (Init) ctx.getBean("in");
2057         System.out.print(i1);
2058     }
2059 }

2060 Example: Programmatic destroy method
2061 import org.springframework.beans.factory.DisposableBean;
2062 public class Destroy implements DisposableBean{
2063     int p1;
2064     public Destroy() {
2065         System.out.println("In constructor of Destroy class");
2066     }
2067     public void setP1(int p1) {
2068         System.out.println("In setter method");
2069         this.p1 = p1;
2070     }
2071     public void destroy() throws Exception {
2072         System.out.println("In destructor of Destroy class");
2073     }
2074 }

2075 //applicationContext.xml
2076 <beans>
2077     <bean id="de" class="Destroy">
2078         <property name = "p1" value = "20"/>
2079     </bean>
2080 </beans>

2081 //DemoClient.java
2082 import org.springframework.context.support.ClassPathXmlApplicationContext;
2083 public class DemoClient {
2084     public static void main(String[] args) {
2085         ClassPathXmlApplicationContext ctx = new
2086         ClassPathXmlApplicationContext("applicationContext.xml");
2087         Destroy d = (Destroy) ctx.getBean("de");
2088         System.out.print(d);
2089         ctx.close();
```

```
2090     }
2091 }
2092 //Example: Declarative init() method
2093 Note:
2094     1. By using declarative statements we can configure any method as a init() method.
2095     2. Need not to implement any interface in case of declarative approach.

2096 //Init.java
2097 public class Init{
2098     int p1;
2099     public Init() {
2100         super();
2101         System.out.println("In constructor");
2102     }
2103     public void setP1(int p1) {
2104         System.out.println("In setter method");
2105         this.p1 = p1;
2106     }
2107     public void init()
2108     {
2109         System.out.println("in user defined call back method");
2110     }
2111 }
2112 //applicationContext.xml
2113 <beans>
2114     <bean id="in" class="Init" init-method = "init">
2115         <property name = "p1" value = "100"/>
2116     </bean>
2117 </beans>
2118 // DemoClient.java
2119 import org.springframework.beans.factory.xml.XmlBeanFactory;
2120 import org.springframework.core.io.ClassPathResource;
2121 public class DemoClient {
2122     public static void main(String[] args) {
2123         ClassPathResource res = new ClassPathResource("applicationContext.xml");
```

```
2124         XmlBeanFactory factory = new XmlBeanFactory(res);
2125         Init i1 = (Init) factory.getBean("in");
2126         System.out.print(i1);
2127     }
2128 }
```

2129 **//Example: Declarative destroy() method**

2130 **Note:**

- 2131 1. By using declarative statements we can configure any method as a destroy() method.
- 2132 2. Need not to implement any interface in case of declarative approach.

2133 **//Destroy.java**

```
2134 public class Destroy
2135 {     int p1;
2136     public Destroy() {
2137         System.out.println("In constructor of Destroy class");
2138     }
2139     public void setP1(int p1) {
2140         System.out.println("In setter method");
2141         this.p1 = p1;
2142     }
2143     public void destroy() {
2144         System.out.println("In destructor of Destroy class");
2145     }
2146 }
```

2147 **//applicationContext.xml**

```
2148 <beans>
2149 <bean id="de" class="Destroy" destroy-method = "destroy">
2150     <property name="p1">
2151         <value>11</value>
2152     </property>
2153 </bean>
2154 </beans>
```

2155 **Note:** Client program is same as above

2156 **Inner Beans:** If we define a <bean> tag inside another bean tag then we call it as inner bean. Inner
2157 beans are used to provide security. If a class is declared as independent bean then the same class can
2158 be used by any other class. The same class if we declare as a inner bean then it can be used only by
2159 outer bean class.

2160 **Example:**

```
2161 public class First {
2162     int a;
2163     int b;
2164     public void setA(int a) {
2165         this.a = a;
2166     }
2167     public void setB(int b) {
2168         this.b = b;
2169     }
2170     @Override
2171     public String toString() {
2172         return a+" "+b;
2173     }
2174 }
2175 public class Second {
2176     First f;
2177     public void setF(First f) {
2178         this.f = f;
2179     }
2180     @Override
2181     public String toString() {
2182         return f.toString();
2183     }
2184 }
```

2185 **//applicationContext.xml**

```
2186 <beans>
2187     <bean id = "two" class = "Second">
2188         <property name="f">
2189             <bean class = "First">
2190                 <property name="a">
2191                     <value>10</value>
2192                 </property>
2193                 <property name="b">
2194                     <value>20</value>
```

```
2195         </property>
```

```
2196     </bean>
```

```
2197 </property>
```

```
2198 </bean>
```

```
2199 </beans>
```

```
2200 //Client.java
```

```
2201 import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
2202 public class Client {
```

```
2203     public static void main(String[] args) {
```

```
2204         ClassPathXmlApplicationContext ctx = new
```

```
2205         ClassPathXmlApplicationContext("applicationContext.xml");
```

```
2206         Second s = (Second) ctx.getBean("two");
```

```
2207         System.out.println(s);
```

```
2208     }
```

```
2209 }
```

2210 **Multiple Constructors:** In a same class we can have multiple constructors. When ever we need to
2211 call multiple constructors through constructor injection then we need to configure the same class
2212 more than one time as shown in below example.

2213 **Example: Calling different constructors of the same class**

```
2214 //Bean.java
```

```
2215 public class Bean {
```

```
2216     int sno;
```

```
2217     String sname;
```

```
2218     public Bean() {}
```

```
2219     public Bean(String sname) {
```

```
2220         this.sname = sname;
```

```
2221     }
```

```
2222     public Bean(int sno, String sname) {
```

```
2223         super();
```

```
2224         this.sno = sno;
```

```
2225         this.sname = sname;
```

```
2226     }
```

```
2227     public String toString() {
```

```
2228         return sno+" "+sname;
```

```
2229     }
```

```
2230 }
```

2231 // applicationContext.xml

2232 <beans>

2233 <bean id = "one" class = "Bean">

2234 <constructor-arg>

2235 <value>Sai</value>

2236 </constructor-arg>

2237 </bean>

2238 <bean id = "two" class = "Bean">

2239 <constructor-arg>

2240 <value>1</value>

2241 </constructor-arg>

2242 <constructor-arg>

2243 <value>Kanakadhar</value>

2244 </constructor-arg>

2245 </bean>

2246 </beans>

2247 //Client.java

2248 import org.springframework.context.support.ClassPathXmlApplicationContext;

2249 public final class Client {

2250 public Client() {}

2251 public static void main(String[] args) {

2252 ClassPathXmlApplicationContext ctx = new

2253 ClassPathXmlApplicationContext("applicationContext.xml");

2254 Bean b1 = (Bean) ctx.getBean("one"); //calls one parameter constructor

2255 Bean b2 = (Bean) ctx.getBean("two");//calls two parameterized constructor

2256 System.out.println(b1);

2257 System.out.println(b2);

2258 }

2259 }

2260 **FactoryBean: BeanFactory:**

2261 1. A special bean class that can return other class object instead of its own object is called as
2262 FactoryBean.

2263 2. In our regular programming we will come across many predefined FactoryBean classes.

2264 3. When factory bean configuration bean id is configured as dependent value to our bean class
2265 property will not be injected with factory bean class object. It will be injected with some
2266 other class object returned by factory bean class.

2267 4. To make spring bean class as factory bean class make that bean class to implement
2268 "org.springframework.beans.factory.FactoryBean" interface.

2269 **Example: To create our own FactoryBean class.**

2270 **//Demo.java**

2271 public interface Demo {

2272 String hi();

2273 }

2274 **//DemoBean.java**

2275 import java.util.Calendar;

2276 import java.util.Date;

2277 public class DemoBean implements Demo{

2278 private Date dt;

2279 private Calendar cl;

2280 public void setDt(Date dt)

2281 {

2282 this.dt = dt;

2283 }

2284 public void setCl(Calendar cl)

2285 {

2286 this.cl = cl;

2287 }

2288 public String hi()

2289 {

2290 return "date: "+dt+" cl = "+cl;

2291 }

2292 }

2293 **//TestBean.java [FactoryBean class]**

2294 import java.util.Calendar;

2295 **import org.springframework.beans.factory.FactoryBean;**

2296 **public class TestBean implements FactoryBean{**

2297 public Object getObject() throws Exception {

2298 System.out.println("getObject()");

2299 Calendar cl = Calendar.getInstance();

2300 return cl;

2301 }

```
2302     public Class getObjectType() {
2303         System.out.println("getObjectType()");
2304         return Calendar.class;
2305     }
2306     public boolean isSingleton() {
2307         System.out.println("isSingleton()");
2308         return true;
2309     }
2310 }
2311 Note:TestBean class returns object of Calendar class.
2312 // applicationContext.xml
2313 <beans>
2314     <bean id = "dat" class = "java.util.Date"/>
2315     <bean id = "tbean" class = "TestBean"/>
2316     <bean id = "db" class = "DemoBean">
2317         <property name="dt">
2318             <ref bean = "dat" />
2319         </property>
2320         <property name="cl">
2321             <ref bean = "tbean"/>
2322         </property>
2323     </bean>
2324 </beans>
2325 //DemoClient.java
2326 import org.springframework.context.support.ClassPathXmlApplicationContext;
2327 public class DemoClient
2328 {
2329     public static void main(String args[])
2330     {
2331         ClassPathXmlApplicationContext ctx = new
2332         ClassPathXmlApplicationContext("applicationContext.xml");

2333         DemoBean d = (DemoBean) ctx.getBean("db");
2334         System.out.println(d.hi());
2335     }
```

2336 }

2337 **Wiring:** The process of configuring bean properties and performing dependency injection on bean
2338 is called as wiring operation.

2339 There are two types of wiring operations:

2340 1. Explicit wiring

2341 2. Auto wiring

2342 If spring bean properties are configured explicitly in spring configuration file for dependency
2343 injection is called as **explicit wiring**.

2344 The dependency injection configurations that we have done so far in our programming come under
2345 explicit wiring.

2346 If container is detecting dependant values to bean properties automatically without any
2347 configuration of properties, then it is called as **auto wiring**.

2348 **Limitations of auto wiring:**

2349 1. Only bean references can be injected. It means simple values of int, float etc..., can't be
2350 injected.

2351 2. There is a possibility of ambiguity (confusion) problem. If multiple suitable dependent
2352 values are found for a bean property in auto wiring.

2353 Spring allows auto wiring in 4 modes:

2354 1. by name

2355 2. by type

2356 3. by constructor

2357 4. by auto detect

2358 To enable auto wiring on spring bean we need to use auto wire attribute of <bean> tag.

2359 1. **byName:** This performs setter injection on bean property. The name of bean property and
2360 bean id of dependent bean class object must match by name.

2361 **Note:** We should use auto wiring on bean property only when the bean property is ready to take
2362 another spring bean class object as a dependent value. No two bean tags should have same bean id.
2363 But same bean class can be configured with separate bean id's using separate bean tags.

2364 2. **byType:** Performs setter injection on bean properties. In this mode of auto wiring the data
2365 type of bean properties must match with the dependent bean object data type. Here it doesn't
2366 checks whether the names were matched or not.

2367 **Note:** No two properties should be configured with same data type. If it happens it will raise
2368 ambiguity error.

2369 3. **constructor:** it performs dependency injection on bean properties by using the
2370 parameterized constructors [constructor injection]. Here also names need not to match but an
2371 appropriate parameterized constructor should be available.

2372 4. **autodetect:** Chooses "constructor" or "byType" mode of auto wiring by scanning the details
2373 of spring bean class. If zero argument constructor is found inspring bean class then
2374 "byType" mode of auto wiring takes place. If appropriate and only parameterized
2375 constructor is available then "constructor" mode of auto wiring takes place.

2376 **Note:** While working with pre defined spring bean classes and if we do not know its properties
2377 names or constructor details particularly then we can use this auto wiring concept.

2378 Auto wiring is useful to work with predefined classes and third party API's.

2379 In auto wiring and explicit wiring both were configured with same setter injection on single bean
2380 property with two different values then the explicit values will be affected.

2381 Eg: byName & explicit.

2382 If auto wiring is performing setter injection and explicit wiring is performing constructor injection
2383 on same bean properties with different values then auto wiring values will be affected.

2384 If auto wiring performs constructor injection and explicit wiring is performing setter injection on
2385 the same property with different values then explicit values will be affected.

2386 **Note:** We can also go for few properties with explicit wiring and few properties with autowiring.

2387 **//Demo.java**

2388 public interface Demo {

2389 String hi();

2390 }

2391 **// DemoBean.java**

2392 public class DemoBean implements Demo{

2393 TestBean tb = null;

2394 public DemoBean()

2395 {

2396 System.out.println("Zero arg constructor of DemoBean class");

2397 }

2398 public DemoBean(TestBean tb)

2399 {

2400 System.out.println("One arg constructor of DemoBean class");

2401 this.tb = tb;

2402 }

2403 public void setTb(TestBean tb)

2404 {

2405 System.out.println("setTb() of DemoBean class");

2406 this.tb = tb;

2407 }

2408 public String hi()

2409 {

2410 return tb.toString();

2411 }

```
2412 }
2413 //TestBean.java
2414 public class TestBean
2415 {
2416     String msg;
2417     public TestBean() {
2418         System.out.println("zero arg. constructor of TestBean class.");
2419     }
2420     public void setMsg(String msg) {
2421         System.out.println("setMsg() of TestBean class");
2422         this.msg = msg;
2423     }
2424     @Override
2425     public String toString() {
2426         return msg;
2427     }
2428 }
2429 //applicationContext.xml [Autowiring by name: Here TestBean class "id" & DemoBean class
2430 property both are having same names. ]
2431 <beans>
2432     <bean id = "tb" class = "TestBean"> <!-- Explicit wiring -->
2433         <property name="msg">
2434             <value>Hi friends... Keep Rocking. :)</value>
2435         </property>
2436     </bean>
2437     <bean id = "db" class = "DemoBean" autowire = "byName"/>
2438 </beans>
2439 Note:
2440     1. When we go for explicit wiring through setter injection then autowiring values will be
2441        overridden and explicit wiring values will be displayed.
2442     2. When we go for explicit wiring through constructor injection then constructor injection
2443        values will be overridden by autowiring values
```


2444 //DemoClient.java

2445 import org.springframework.context.support.ClassPathXmlApplicationContext;

2446 public class DemoClient

2447 {

2448 public static void main(String args[])

2449 {

2450 ClassPathXmlApplicationContext ctx = new

2451 ClassPathXmlApplicationContext("applicationContext.xml");

2452 DemoBean d = (DemoBean) ctx.getBean("db");

2453 System.out.println(d.hi());

2454 }

2455 }

2456 **Exmple: Setter injection will always dominate Constructor injection**

2457 <!-- applicationContext.xml -->

2458 <beans>

2459 <bean id = "tb" class = "TestBean"> <!-- Explicit wiring -->

2460 <property name="msg">

2461 <value>Hi friends... Happy Republic Day.... :)</value>

2462 </property>

2463 </bean>

2464 <bean id = "tb1" class = "TestBean">

2465 <property name="msg">

2466 <value>Gmg. friends.... :)</value>

2467 </property>

2468 </bean>

2469 <bean id = "db" class = "DemoBean" autowire = "byName"/>

2470 <constructor-arg>

2471 <ref bean = "tb1"/>

2472 </constructor-arg>

2473 </bean>

2474 </beans>

2475 **Note:**

2476 1. Here explicitly we are doing constructor injection.

2477 2. First of all constructor values will be injected & again setter values will be injected.

2478 3. So that setter values will be displayed & constructor values will be ignored.

2479 **//Example: Explicit injection dominates implicit injection when both are using either setter or**
2480 **constructor injection**

```
2481 <beans>
2482     <bean id = "tb" class = "TestBean"> <!-- Explicit wiring -->
2483         <property name="msg">
2484             <value>Hi friends... Happy Republic Day.... :)</value>
2485         </property>
2486     </bean>
2487     <bean id = "tb1" class = "TestBean">
2488         <property name="msg">
2489             <value>Gmg. friends.... :)</value>
2490         </property>
2491     </bean>
2492     <bean id = "db" class = "DemoBean" autowire = "byName"/>
2493         <property name="tb">
2494             <ref bean = "tb1"/>
2495         </property>
2496     </bean>
2497 </beans>
```

2498 **Note:**

- 2499 **1.** In the above example auto wiring & explicit wiring both are trying to do setter injection. In
2500 this case explicit injection will dominate implicit injection.
- 2501 **2.** Similarly if auto wiring & explicit wiring both are trying to do constructor injection, then
2502 also explicit injection will dominate implicit injection.

2503 **//Example: byType**

2504 **Note:** The entire example is same. The only modification is with configuration file.

```
2505 <!-- applicationContext.xml -->
2506     <bean id = "t1" class = "TestBean"> <!-- Explicit wiring -->
2507         <property name="msg">
2508             <value>Gmg. friends... Have a joyful day. :)</value>
2509         </property>
2510     </bean>
2511     <bean id = "db" class = "DemoBean" autowire = "byType"/>
2512 </beans>
```

2513 **Note:**

2514 **1.** Here TestBean class "id" & DemoBean class property both are different. But still here
2515 autowiring is working based up on data type of bean class property & Bean class.

2516 **2.** In autowiring with byType the same class should not be configured more than once.

2517 **//Example: We should not do as follows:** Because TestBean class is configured more than once
2518 where byType is used. If we configure more than once then it will throw amiguity error.

2519 **<!-- applicationContext.xml -->**

2520 <beans>

2521 <bean id = "t1" class = "TestBean"> <!-- Explicit wiring -->

2522 <property name="msg">

2523 <value>Gmg. friends... Have a joyful day. :)</value>

2524 </property>

2525 </bean>

2526 <bean id = "t2" class = "TestBean">

2527 <property name="msg">

2528 <value>Gmg.... friends Have a joyful day. :)</value>

2529 </property>

2530 </bean>

2531 <bean id = "db" class = "DemoBean" autowire = "byType"/>

2532 </beans>

2533 **//Example: constructor**

2534 **Note:** The entire example is same. The only modification is with configuration file.

2535 **<!-- applicationContext.xml -->**

2536 <beans>

2537 <bean id = "t2" class = "TestBean"> <!-- Explicit wiring -->

2538 <property name="msg">

2539 <value>Gmg. friends... Have a joyful day. :)</value>

2540 </property>

2541 </bean>

2542 <bean id = "db" class = "DemoBean" autowire = "**constructor**"/>

2543 </beans>

2544 //Example: autodetect

2545 <!-- applicationContext.xml -->

2546 **Note:** The entire example is same. The only modification is with configuration file.

2547 <beans>

2548 <bean id = "t1" class = "TestBean"> <!-- Explicit wiring -->

2549 <property name="msg">

2550 <value>Gmg. friends... Have a joyful day. :)</value>

2551 </property>

2552 </bean>

2553 <bean id = "db" class = "DemoBean" autowire = "**autodetect**"/>

2554 </beans>

2555 **DAO Module:**

2556 **Spring DAO:**

2557 **Connection pooling:**

2558 Every jdbc data source object represents one jdbc connection pool. To access each connection
2559 object of connection pool we need to work with jdbc data source object.

2560 Jdbc connection pool is a factory that contains set of readily available jdbc connection objects
2561 before actually being used.

2562 In spring application we can work with 3 types of connection pools:

2563 1. spring built in jdbc connection pool [org.sf.jdbc.datasource.DriverManagerDataSource
2564 class]

2565 **Note:** This connection pool is not real jdbc connection pool. Because it does not pool jdbc
2566 connection objects. Moreover it creates one jdbc connection object dynamically based on demand.
2567 Using this style is not industrial standard.

2568 2. Using third party connection pool software's. [c3p0, proxool, dc30, apache.. etc.,]

2569 3. Using web/application server managed jdbc connection pools.

2570 If our application is stand alone application or the application that runs outside the server
2571 [web/application server] then use third party connection pooling.

2572 If our spring application is deployed in web/application server then use server managed connection
2573 pooling.

2574 **DAO class:**

2575 1. The java class or application that separates persistence logic from other applications is
2576 called as DAO.

2577 2. DAO is very useful to make persistence logic as flexible logic for modification with out
2578 affecting the other logics of the application.

2579 3. Spring DAO module is given to develop jdbc style persistence logic in any layer of the
2580 project by getting abstraction layer on jdbc programming.

2581 4. Using spring DAO module we can develop jdbc style persistence logic without directly
2582 using jdbc logic and without performing common jdbc operations like establishing

2583 connections, transaction management, jdbc operations, createing & closing of statement
2584 objects etc...

2585 **Plain jdbc code:**

- 2586 1. load or register jdbc driver
- 2587 2. establish connection to database
- 2588 3. create statement object
- 2589 4. begin transaction
- 2590 5. Perform persistent operations like insert, update, delete or read etc...,
- 2591 6. gather results and process results
- 2592 7. Commit or rollback based on transaction execution.
- 2593 8. Close the connection

2594 **Spring DAO based persistence logic:**

- 2595 1. Get access to jdbc template class object through dependency injection process
- 2596 2. perform persistence operations like insert, update, delete, read/select.
- 2597 3. Gather results and process.

2598 If we use jdbc template class given by spring dao module, it will take care of the common activities
2599 of jdbc programming like establishing connection, closing connection, transaction management &
2600 other common operations. So programmer can concentrate only on application specific operations.

2601 Jdbc template class provides lots of methods to perform persistence operations on database s/w
2602 without using any kind of jdbc api.

2603 query(_), queryForXxx(_) of jdbc template class are given to execute sql select queries.

2604 update() of jdbc template class is given to execute non-select queries.

2605 batchUpdate() is used for batch processing.

2606 **Limitations with jdbc:**

- 2607 1. All the steps are mandatory to follow.
- 2608 2. ResultSet object is not serializable object so we cannot send it over network.
- 2609 3. All jdbc exception are checked exceptions so programmer must catch and handle them.
- 2610 4. Allows only positional parameters for queries (?).

2611 **Features of spring DAO:**

- 2612 1. Allows to develop jdbc style persistence logic without using jdbc api.
- 2613 2. The jdbc template class takes care of the common jdbc workflow or operations. So
2614 programmer can concentrate only on application specific persistence logic development.
- 2615 3. Select query results can be stored directly in collection framework data structures which are
2616 serializable objects by default.
- 2617 4. Spring DAO internally uses jdbc code. So it catches all jdbc checked exceptions thrown by
2618 jdbc code and rethrows as unchecked exceptions.

- 2619 5. Simplifies the process of developing jdbc style persistence logic by providing abstraction
2620 layer on jdbc programming.
- 2621 6. Spring DAO based persistence logic is not database independent because it internally uses
2622 jdbc code and database specific queries given by programmer.
- 2623 7. To execute select queries that returns multiple records use “queryForList()”.
- 2624 8. To execute select query that returns a single record use “queryForMap()”.
- 2625 9. To execute select query that contains aggregate function results or any individual values use
2626 “queryForLong()”, “queryForInt()” etc.,
- 2627 10. Allows both positional (?) and named (<:pname> parameters for queries.
- 2628 11. To work with named parameters we must use NamedParameterJdbcTemplate class.
- 2629 12. Single connection data source represents a connection pool that contains only one jdbc
2630 connection object and will not be closed automatically when spring container is stopped.
- 2631 13. In order to close this single connection we must call close() method implicitly. This class is
2632 subclass of DriverManagerDataSource and both these classes related connection pools are
2633 not meant for real time programming.

2634 **Connection Pool:**

- 2635 1. Connection Pool is a pool of ready made specified number of database connection objects.
- 2636 2. When ever client program needs a database connection they can request our pool and can
2637 get one connection object.
- 2638 3. Client program can do all database operations and can release the connection object back to
2639 connection pool.
- 2640 4. The same connection object can be reused any number of times till the connection pool is
2641 destroyed.
- 2642 5. Connection objects will be in waiting state in connection pool for client request.
- 2643 6. At a time only one client can use one connection object.
- 2644 7. If all connection objects of connection pool are busy then client program should be waiting
2645 state till any client releases a connection object.
- 2646 8. With connection pool we can increase performance of entire application.
- 2647 9. In real time applications regularly we work with connection pool.
- 2648 10. When huge amount of customers are trying to connect with database server in those cases
2649 better we use connection pooling concept.

2650 **Connection pool can be created in 3 ways:**

- 2651 1. Dummy Connection pool
- 2652 2. Third Party Connection pool
- 2653 3. Web server or application server supplied connection pool

2654 **Example: Integrating spring with Jdbc [Dummy Connection pool]**

2655 **//SelectInter.java**

2656 public interface SelectInter

2657 {

2658 public String fetchEmpName(int eno);

2659 public long fetchEmpSalary(int eno);

2660 }

2661 **//SelectImple.java**

2662 import java.sql.*;

2663 **import javax.sql.DataSource;**

2664 public class SelectImple implements SelectInter

2665 {

2666 **DataSource ds = null;**

2667 public void setDs(DataSource ds)

2668 {

2669 this.ds = ds;

2670 }

2671 public String fetchEmpName(int eno)

2672 {

2673 try

2674 {

2675 //get access to jdbc connection object from jdbc connection pool through data

2676 source object "ds"

2677 **Connection con = ds.getConnection();**

2678 //persistence logic

2679 Statement st = con.createStatement();

2680 ResultSet rs = st.executeQuery("select ename from emp where empno =

2681 "+eno);

2682 if(rs.next())

2683 {

2684 return rs.getString(1);

2685 }

2686 else

2687 {

2688 System.out.println("No record found with employee number: "+eno);

2689 }

2690 rs.close();

2691 st.close();

2692 con.close();

2693 }

2694 catch(SQLException e)

2695 {

2696 e.printStackTrace();

2697 return "Record not found";

2698 }

2699 catch(Exception e)

```

2700         {
2701             e.printStackTrace();
2702             return "Record not found";
2703         } //catch
2704         return "Record not found";
2705     } //fetch empName
2706     public long fetchEmpSalary(int eno)
2707     {
2708         try
2709         {
2710             //get access to jdbc connection object from jdbc connection pool through data
2711             source object "ds"
2712             Connection con = ds.getConnection();
2713             //persistence logic
2714             Statement st = con.createStatement();
2715             ResultSet rs = st.executeQuery("select sal from emp where empno = "+eno);
2716             if(rs.next())
2717             {
2718                 return rs.getLong(1);
2719             }
2720             else
2721             {
2722                 System.out.println("No record found with employee number: "+eno);
2723             }
2724             rs.close();
2725             st.close();
2726             con.close();
2727         }
2728         catch(SQLException e)
2729         {
2730             e.printStackTrace();
2731             return 0;
2732         }
2733         catch(Exception e)
2734         {
2735             e.printStackTrace();
2736             return 0;
2737         } //catch
2738         return 0;
2739     } //fetch empsalary
2740 }

```

Note:

- 2742 1. Its same as regular Jdbc program. But here we are getting Connection object by using
- 2743 DataSource object.
- 2744 2. **DataSource** is an interface belongs to javax.sql package. The same interface has many
- 2745 implementation classes. One of the class is **DriverManagerDataSource**.

2746 **//spring.cfg.xml**

```

2747 <?xml version="1.0" encoding="UTF-8"?>
2748 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"

```



```

2749 "http://www.springframework.org/dtd/spring-beans.dtd">
2750 <beans>
2751     <bean id="dmds"
2752         class="org.springframework.jdbc.datasource.DriverManagerDataSource">
2753         <property name="driverClassName">
2754             <value>oracle.jdbc.driver.OracleDriver</value>
2755         </property>
2756         <property name="url">
2757             <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
2758         </property>
2759         <property name="username">
2760             <value>scott</value>
2761         </property>
2762         <property name="password">
2763             <value>tiger</value>
2764         </property>
2765     </bean>
2766     <bean id="sel" class="SelectImple">
2767         <property name="ds">
2768             <ref bean = "dmds"/>
2769         </property>
2770     </bean>
2771 </beans>

2772 Note: In the above xml file we have given service id in url.
2773 Q) How to find service id of oracle
2774 Ans:
2775 SQL> select * from global_name;
2776 GLOBAL_NAME
2777 -----
2778 SATYA.US.ORACLE.COM

2779 //SelectClient.java
2780 import org.springframework.core.io.FileSystemResource;
2781 import org.springframework.beans.factory.xml.XmlBeanFactory;
2782 public class SelectClient
2783 {
2784     public static void main(String args[])
2785     {
2786         FileSystemResource res=new FileSystemResource("spring.cfg.xml");
2787         XmlBeanFactory factory=new XmlBeanFactory(res);
2788         SelectImple s=(SelectImple)factory.getBean("sel");
2789         System.out.println("Employee name is: "+s.fetchEmpName(7839));
2790         System.out.println("Employee salary is: "+s.fetchEmpSalary(7839));
2791     }
2792 }

```

2793 **Example: Third Party Connection pool – Apache Tomcat supplied jar file**

2794 **Note:** Many third party connection pools are available. We'll see two among them.

2795 **1.** Tomcat supplied connection pool [Need not to start tomcat server. Simply we need to **set the**
2796 **class path for "naming-factory-dbcp.jar" file]**

2797 **2.** It is available in **Tomcat 5.5\common\lib\naming-factory-dbcp.jar**

2798 **3.** Also set classpath to **classes12.jar**

2799 **Note:** Only change is in configurationm file. The rest of resources will be same.

2800 **//spring.cfg.xml**

2801 `<?xml version="1.0" encoding="UTF-8"?>`

2802 `<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"`

2803 `"http://www.springframework.org/dtd/spring-beans.dtd">`

2804 `<beans>`

2805 `<bean id="dbcpds" class="org.apache.tomcat.dbcp.dbcp.BasicDataSource"`
2806 `destroy-method = "close">`

2807 `<property name="driverClassName">`

2808 `<value>oracle.jdbc.driver.OracleDriver</value>`

2809 `</property>`

2810 `<property name="url">`

2811 `<value>jdbc:oracle:thin:@localhost:1521:sathya</value>`

2812 `</property>`

2813 `<property name="username">`

2814 `<value>scott</value>`

2815 `</property>`

2816 `<property name="password">`

2817 `<value>tiger</value>`

2818 `</property>`

2819 `<property name="initialSize">`

2820 `<value>3</value>`

2821 `</property>`

2822 `</bean>`

2823 `<bean id="sel" class="SelectImple">`

2824 `<property name="ds">`

2825 `<ref bean = "dbcpds"/>`

2826 `</property>`

2827 `</bean>`

2828 `</beans>`

2829 **Example: Third Party Connection pool – Hibernate supplied jar file**

- 2830 **1.** Tomcat supplied connection pool [Need not to start tomcat server. Simply we need to **set the**
2831 **class path for "c3p0-0.9.1.jar" file]**
- 2832 **2.** It is available in **hibernate-home\lib\c3p0-0.9.1.jar**
- 2833 **3.** Also set classpath to **classes12.jar**

2834 **Note:** Only change is in configurationm file. The rest of resources will be same.

2835 **//spring.cfg.xml**

2836 `<?xml version="1.0" encoding="UTF-8"?>`

2837 `<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"`

2838 `"http://www.springframework.org/dtd/spring-beans.dtd">`

2839 `<beans>`

2840 `<bean id="c3p0ds" class="com.mchange.v2.c3p0.ComboPooledDataSource"`
2841 `destroy-method = "close">`

2842 `<property name="driverClass">`

2843 `<value>oracle.jdbc.driver.OracleDriver</value>`

2844 `</property>`

2845 `<property name="jdbcUrl">`

2846 `<value>jdbc:oracle:thin:@localhost:1521:sathya</value>`

2847 `</property>`

2848 `<property name="user">`

2849 `<value>scott</value>`

2850 `</property>`

2851 `<property name="password">`

2852 `<value>tiger</value>`

2853 `</property>`

2854 `<property name="initialPoolSize">`

2855 `<value>3</value>`

2856 `</property>`

2857 `<property name="maxPoolSize">`

2858 `<value>30</value>`

2859 `</property>`

2860 `<property name="acquireIncrement">`

2861 `<value>5</value>`

2862 `</property>`

2863 `</bean>`

2864 `<bean id="sel" class="SelectImple">`

```

2865         <property name="ds">
2866             <ref bean = "c3p0ds"/>
2867         </property>
2868     </bean>
2869 </beans>

```

2870 **Web or application server based connection pool:**

- 2871 1. The nick name or alias name given to the object that is registered with registry s/w is
- 2872 technically called as **JNDI name**.
- 2873 2. Data source represents server managed connection pool.
- 2874 3. To provide global visibility to this data source object from multiple client applications we
- 2875 need to register it with registry s/w having jndi name.
- 2876 4. Client application always uses jdbc data source object to get access to connection object of
- 2877 connection pool.
- 2878 5. Jdbc data source object is an object of a class which implements java.sql.DataSource
- 2879 interface.

2880 **Steps:**

- 2881 1. Create Jdbc connection pool for certain database s/w.
- 2882 2. Create jdbc data source and link it with jdbc connection pool.
- 2883 3. register jdbc data source with registry s/w for global visibility
- 2884 4. Client application performs jndi lookup operation on registry & gets jdbc data source object.
- 2885 5. getConnection() method on data source object gets one jdbc connection object from jdbc
- 2886 connection pool.
- 2887 6. Client application uses this jdbc connection object for persistence operations on database.
- 2888 7. con.close() releases jdbc connection object back to the connection pool so that the same
- 2889 object can be used by other clients.

2890 **Example: Application server supplied connection pool with Weblogic**

2891 **set the class path for:**

- 2892 1. "weblogic1.jar" file [C:\bea\weblogic81\server\lib\weblogic.jar]
- 2893 2. classes12.jar

2894 **Weblogic8.1 connection pooling:**

2895 **Step1:** Create user in weblogic & start admin server

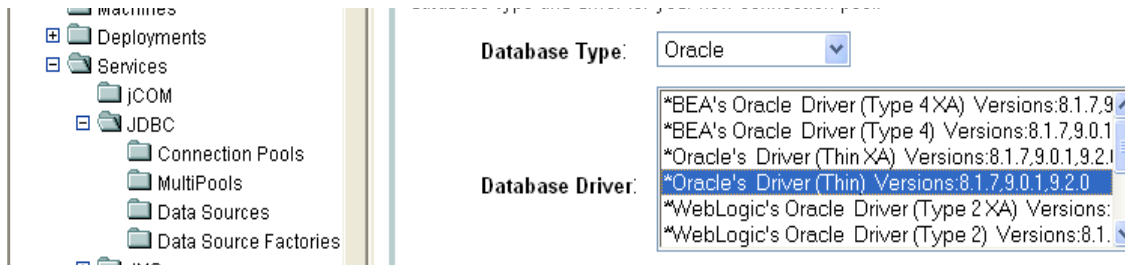
2896 Start menu à programs à bea weblogic platform8.1 à configuration wizard à next à next à give
 2897 password and confirm password as same [remind this password.] à next à next à specify
 2898 configuration name [domain name] at right bottom of window à create à select start admin server à
 2899 done.

2900 **Step2:** open administrator console

2901 Start menu à programs à bea weblogic platform8.1 à examples à weblogic workshop à server admin
 2902 console à give password here à login.

2903 **Step3:** Create Connection pool

2904 Choose your domain name [in left pane of admin console window]à services à jdbc à click on
 2905 **connection pools** à [Configure a new JDBC Connection Pool](#) à choose database and its
 2906 corresponding driver as shown belowà continue



2907 **Define connection properties:**

2908 **Name:** give some name for connection pool

2909 **Database name:** give service id if it is oracle

2910 **Host Name:** ip address or server name where database server resides

2911 **Port:** port no. of database server

2912 Also provide database username & password and click on **continue** button à click on button “**test driver connection**”. à if it shows connection successful then click on à “**Create & deploy**”.

2914 **Step4:** Create data source

2915 Choose your domain name [in left pane of admin console window]à services à jdbc à click on **Data**
 2916 **sources** à click on “[Configure a new JDBC Data Source](#)” à

2917 **Name:** Give some name for Data source

2918 **JNDI name:** Give some JNDI name. [remind this. We have to configure this in our program]

2919 Select check box of : Emulate Two-Phase Commit for non-XA Driverà continue. à choose our
 2920 connection pool from combo box à continue à create.

2921 **Note:** Only change is in configurationm file. The rest of resources will be same.

2922 **//spring.cfg.xml**

2923 `<?xml version="1.0" encoding="UTF-8"?>`

2924 `<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"`

2925 `"http://www.springframework.org/dtd/spring-beans.dtd">`

2926 `<beans>`

2927 `<bean id="jofb" class="org.springframework.jndi.JndiObjectFactoryBean">`

2928 `<property name="jndiName">`

2929 `<value>oraJNDI</value>`

2930 `</property>`

2931 `<property name="jndiEnvironment">`

2932 `<props>`

2933 `<prop key = "java.naming.factory.initial">weblogic.jndi.WLInitialContextFactory</prop>`

```

2934         <prop key = "provider_url">t3://localhost:7001</prop>
2935     </props>
2936 </property>
2937 </bean>
2938 <bean id="sel" class="SelectImple">
2939     <property name="ds">
2940         <ref bean = "jofb"/>
2941     </property>
2942 </bean>
2943 </beans>

```

2944 **Note:**

- 2945 1. As we have specified driver class, database url, user name & password in weblogic need not
- 2946 to specify once again in spring configuration file.
- 2947 2. A connection pool is a factory that contains set of readily available jdbc connection objects
- 2948 before actually being used.
- 2949 3. Jdbc DataSource object always represents one jdbc connection pool.
- 2950 4. To access each connection object of connection pool we need to use jdbc data source object.
- 2951 5. org.springframework.jdbc.datasource.DriverManagerDataSource represents spring built in
- 2952 jdbc dummy connection pool by giving data source object.
- 2953 6. This is called as dummy connection pool because it does not pools the connection objects. It
- 2954 creates connection object on demand.
- 2955 7. DriverManagerDataSource implements javax.sql.DataSource interface.
- 2956 8. This class uses given jdbc driver details and uses jdbc data source object which represents
- 2957 dummy connection pool.

2958 **Note:** Do not use **DriverManagerDataSource** in your real time projects.

2959 **Example: Working with JdbcTemplate class [queryForXxx() methods]**

2960 **//SelectInter.java**

```

2961 import java.util.List;
2962 import java.util.Map;
2963 public interface SelectInter {
2964     int getEmpCount(String desg);
2965     List getEmpDetails(String desg);
2966     Map getEmpDetails(int eno);
2967     boolean registerEmp(int no, String name, String desg, int sal);
2968     boolean modifyDesignation(int no, String newDesig);
2969     boolean fireEmp(int eno);
2970 }

```

```
2971 //SelectImple.java
2972 import java.util.List;
2973 import java.util.Map;
2974 import org.springframework.jdbc.core.JdbcTemplate;
2975 public class SelectImple implements SelectInter{
2976     JdbcTemplate jt;
2977     String qry;
2978     public void setJt(JdbcTemplate jt) {
2979         this.jt = jt;
2980     }
2981     public boolean fireEmp(int eno) {
2982         qry = "delete from emp where empno = ?";
2983         int res = jt.update(qry, new Object[]{new Integer(eno)});
2984
2985         if(res == 0)
2986             return false;
2987         else
2988             return true;
2989     }
2990     public int getEmpCount(String desg) {
2991         qry = "select count(*) from emp where job = ?";
2992         int count = jt.queryForInt(qry, new String[]{desg});
2993         return count;
2994     }
2995     public Map getEmpDetails(int eno) {
2996         qry = "select * from emp where empno = ? ";
2997         Map m = jt.queryForMap(qry, new Object[]{new Integer(eno)});
2998         return m;
2999     }
3000     public List getEmpDetails(String desg) {
3001         qry = "select * from emp where job = ?";
3002         List l = jt.queryForList(qry, new Object[]{desg});
3003         return l;
3004     }
3005     public boolean modifyDesignation(int no, String newDesig) {
```

```

3005         qry = "update emp set job = ? where empno = ?";
3006         int res = jt.update(qry, new Object[]{new Desig, new Integer(no)});
3007         if(res == 0)
3008             return false;
3009         else
3010             return true;
3011     }
3012     public boolean registerEmp(int no, String name, String desg, int sal) {
3013         qry = "insert into emp (empno, ename, job, sal) values(?, ?, ?, ?)";
3014         int res = jt.update(qry, new Object[]{new Integer(no), name, desg, new
3015 Integer(sal)});
3016         if(res == 0)
3017             return false;
3018         else
3019             return true;
3020     }
3021 }

```

3022 **//applicationContext.xml**

```

3023 <?xml version="1.0" encoding="UTF-8"?>
3024 <beans
3025     xmlns="http://www.springframework.org/schema/beans"
3026     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3027     xsi:schemaLocation="http://www.springframework.org/schema/beans
3028 http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
3029 <!-- note: add naming-factory-dbcp.jar file from tomcat-home\common\lib -->
3030     <bean id="dbcp" class="org.apache.tomcat.dbcp.dbcp.BasicDataSource">
3031         <property name="driverClassName">
3032             <value>oracle.jdbc.driver.OracleDriver</value>
3033         </property>
3034         <property name="url">
3035             <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
3036         </property>
3037         <property name="username">
3038             <value>scott</value>
3039         </property>

```

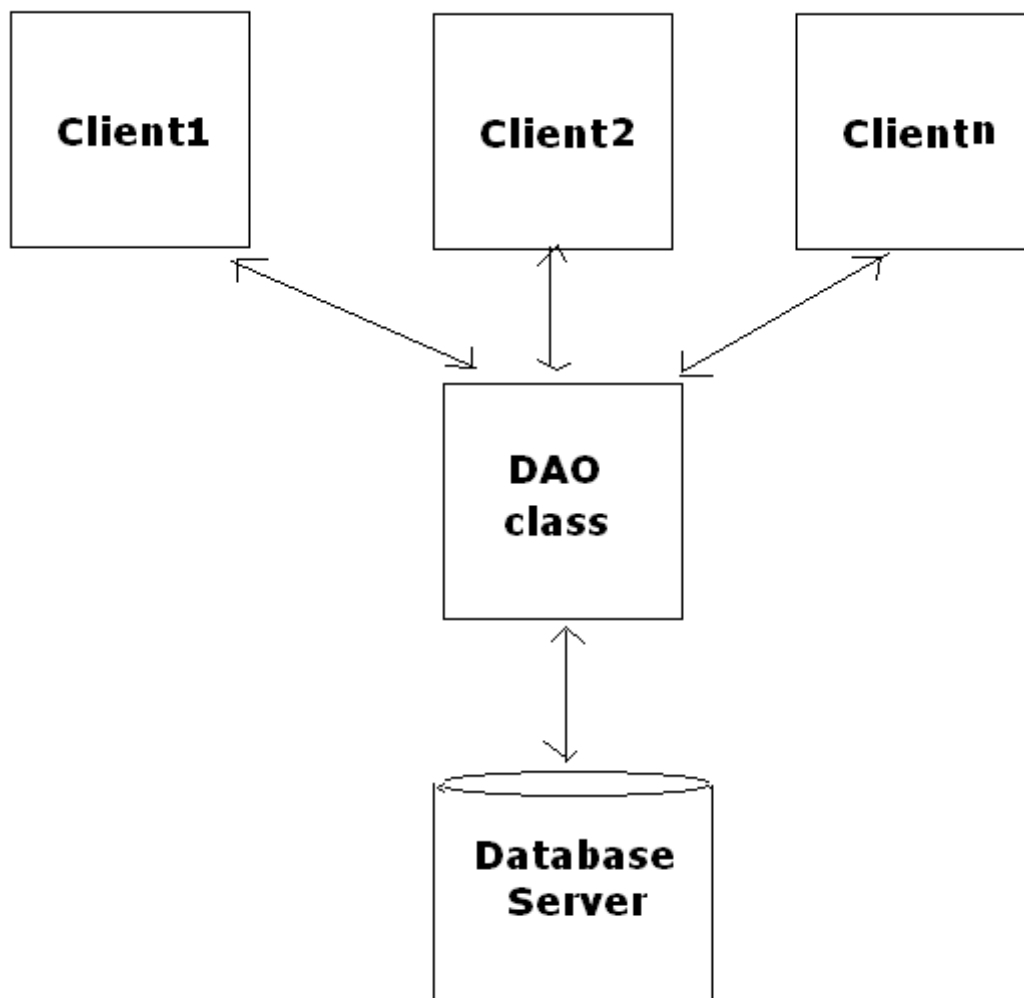


```

3040         <property name="password">
3041             <value>tiger</value>
3042         </property>
3043         <property name="initialSize">
3044             <value>3</value>
3045         </property>
3046     </bean>
3047     <bean id="template" class="org.springframework.jdbc.core.JdbcTemplate">
3048         <property name="dataSource">
3049             <ref bean = "dbcp"/>
3050         </property>
3051     </bean>
3052     <bean id="sel" class="SelectImple">
3053         <property name="jt">
3054             <ref bean = "template"/>
3055         </property>
3056     </bean>
3057 </beans>
3058 //SelectClient.java
3059 import java.util.List;
3060 import java.util.Map;
3061 import org.springframework.context.support.ClassPathXmlApplicationContext;
3062 public class SelectClient
3063 {
3064     public static void main(String args[])throws Exception
3065     {
3066         ClassPathXmlApplicationContext ctx = new
3067         ClassPathXmlApplicationContext("applicationContext.xml");
3068         SelectImple s=(SelectImple)ctx.getBean("sel");
3069         System.out.println("Employee count with Clerk designation is:
3070 "+s.getEmpCount("CLERK"));
3071         Map m = s.getEmpDetails(7934);
3072         System.out.println("Details of empno: 7934 are: "+m.toString());
3073         System.out.println("Clerk designation employees details are: ");
3074         List l = s.getEmpDetails("CLERK");
3075         for(int i = 0; i < l.size(); i++)

```

```
3076         {
3077             Map m1 = (Map) l.get(i);
3078             System.out.println(m1.toString());
3079         }
3080         boolean bool = s.registerEmp(1, "Kanakadhar", "MANAGER", 50000);
3081         System.out.println("Employee registered "+bool);
3082         bool = s.modifyDesignation(1, "PRESIDENT");
3083         System.out.println("Employee designation updated "+bool);
3084         Thread.sleep(10000);
3085         bool = s.fireEmp(1);
3086         System.out.println("Employee fired "+bool);
3087     }
3088 }
```



3089 **Note:** The same DAO class can be shared by multiple client programs. But in our example we have
3090 seen only one client program which is calling all methods of DAO.

3091 **Example: Named parameter without container**

```
3092 import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
3093 import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
3094 import org.springframework.jdbc.datasource.SingleConnectionDataSource;
3095 public class Client {
3096     public static void main(String[] args) {
3097         SingleConnectionDataSource ds = new SingleConnectionDataSource();
3098         ds.setDriverClassName("oracle.jdbc.driver.OracleDriver");
3099         ds.setUrl("jdbc:oracle:thin:@localhost:1521:sathya");
3100         ds.setUsername("scott");
3101         ds.setPassword("tiger");
3102         NamedParameterJdbcTemplate template = new NamedParameterJdbcTemplate(
3103             String qry = "select count (*) from emp where job = :desig";
3104             MapSqlParameterSource params = new MapSqlParameterSource();
3105             params.addValue("desig", "CLERK");
3106             int count = template.queryForInt(qry, params);
3107             System.out.println("No. of Clerks are: "+count);
3108             ds.destroy();
3109         }
3110     }
```

3111 **//Example: Named parameter with container**

3112 **// SelectInter.java**

```
3113 public interface SelectInter {
3114     int getEmpCount(int eno1, int eno2);
3115 }
```

3116 **//SelectImple.java**

```
3117 import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
3118 import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
3119 //Add apache tomcat third party jar "naming-factory-dbcp.jar" file to build path
3120 //Add classes12.jar for build path
3121 public class SelectImple implements SelectInter{
3122     NamedParameterJdbcTemplate nt;
3123     String qry = "select count(*) from emp where empno >= :e1 and empno <= :e2";
3124     public void setNt(NamedParameterJdbcTemplate nt) {
3125         this.nt = nt;
```

```

3126         }
3127         public int getEmpCount(int eno1, int eno2) {
3128             MapSqlParameterSource params = new MapSqlParameterSource();
3129             params.addValue("e1", new Integer(eno1));
3130             params.addValue("e2", new Integer(eno2));
3131             int count = nt.queryForInt(qry, params);
3132             return count;
3133         }
3134     }
3135 // applicationContext.xml
3136 <!-- note: add naming-factory-dbcp.jar file from tomcat-home\common\lib -->
3137 <beans>
3138     <bean id="dbcp" class="org.apache.tomcat.dbcp.dbcp.BasicDataSource">
3139         <property name="driverClassName">
3140             <value>oracle.jdbc.driver.OracleDriver</value>
3141         </property>
3142         <property name="url">
3143             <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
3144         </property>
3145         <property name="username">
3146             <value>scott</value>
3147         </property>
3148         <property name="password">
3149             <value>tiger</value>
3150         </property>
3151         <property name="initialSize">
3152             <value>3</value>
3153         </property>
3154     </bean>
3155     <bean id="namedTemplate"
3156         class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate"
3157         <constructor-arg>
3158             <ref bean = "dbcp"/>
3159         </constructor-arg>
3160     </bean>

```

```
3161     <bean id="sel" class="SelectImple">
3162         <property name="nt">
3163             <ref bean = "namedTemplate"/>
3164         </property>
3165     </bean>
3166 </beans>
3167 //SelectClient.java
3168 import org.springframework.context.support.ClassPathXmlApplicationContext;
3169 public class SelectClient
3170 {
3171     public static void main(String args[])
3172     {
3173         ClassPathXmlApplicationContext ctx = new
3174         ClassPathXmlApplicationContext("applicationContext.xml");
3175         SelectImple s=(SelectImple)ctx.getBean("sel");
3176         System.out.println("Employees in between empid 7500 to 7800 count is:
3177 "+s.getEmpCount(7500, 7800));
3178     }
3179 }
3180 Example: To call oracle stored procedure
3181 Note:
3182     1. Create this procedure through sql> prompt of oracle client.
3183     2. Make sure that "emp" table is existing in oracle. Normally it is a default table in scott user.
3184 CREATE OR REPLACE PROCEDURE GET_EMP_DATA
3185     (NAME IN VARCHAR,
3186     NO OUT NUMBER,
3187     DESG OUT VARCHAR,
3188     SALARY OUT NUMBER)
3189 AS
3190 BEGIN
3191     SELECT EMPNO, JOB, SAL
3192     INTO NO, DESG, SALARY
3193     FROM EMP
3194     WHERE ENAME = NAME;
3195 END;
```

3196 **Employee.java [POJO class]**

```
3197 class Employee {
3198     private Integer no;
3199     private String name;
3200     private String designation;
3201     private Double salary;
3202     public void setNo(Integer no) {
3203         this.no = no;
3204     }
3205     public void setName(String name) {
3206         this.name = name;
3207     }
3208     public void setDesignation(String designation) {
3209         this.designation = designation;
3210     }
3211     public void setSalary(Double salary) {
3212         this.salary = salary;
3213     }
3214     public String toString() {
3215         return this.no + "\t"
3216             + this.name + "\t"
3217             + this.designation + "\t"
3218             + this.salary;
3219     }
3220 }
```

3221 **// SimpleJdbcCallTest.java**

```
3222 /*
3223 steps:
3224 1. create datasource
3225 2. pass datasource to subclass of StoredProcedure which is a static innerclass
3226 3. Register all in & out parameters
3227 4. read data from Map object, set to bean class object & display
3228 */
3229 import java.sql.Types;
3230 import java.util.HashMap;
```

```
3231 import java.util.Map;
3232 import javax.sql.DataSource;
3233 import org.springframework.jdbc.core.SqlOutParameter;
3234 import org.springframework.jdbc.core.SqlParameter;
3235 import org.springframework.jdbc.object.StoredProcedure;
3236 import org.springframework.jdbc.datasource.*;
3237 public class SimpleJdbcCallTest {
3238     public static void main(String args[]) {
3239         DriverManagerDataSource dataSource = new DriverManagerDataSource();
3240         dataSource.setDriverClassName("oracle.jdbc.driver.OracleDriver");
3241         dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:SATHYA");
3242         dataSource.setUsername("scott");
3243         dataSource.setPassword("tiger");
3244         Employee emp = (Employee)
3245             getEmployeeDetailsWithStoredProcedure(dataSource,"ALLEN");
3246         System.out.println("Allen Details : " + emp);
3247     }

3248     private static Object getEmployeeDetailsWithStoredProcedure
3249         (DataSource ds, String ename) {
3250         MyStoredProcedure sp = new MyStoredProcedure(ds);
3251         //call procedure
3252         Map results = sp.myexecute(ename);
3253         //set outparmeter values to emp object
3254         Employee emp = new Employee();
3255         emp.setName(ename);
3256         emp.setNo((Integer)results.get("NO"));
3257         emp.setDesignation((String)results.get("DESG"));
3258         emp.setSalary((Double)results.get("SALARY"));
3259         return emp;
3260     } //method2
3261 //MyStoredProcedure.java
3262 private static class MyStoredProcedure extends StoredProcedure {
3263     public MyStoredProcedure(DataSource ds) {
3264         super(ds, "GET_EMP_DATA");
```

```
3265         this.setFunction(false); //false -> indicates its a stored procedure
3266         SqlParameter[] params = {
3267             new SqlParameter("NAME", Types.VARCHAR),
3268             new SqlParameter("NO", Types.INTEGER),
3269             new SqlParameter("DESG", Types.VARCHAR),
3270             new SqlParameter("SALARY", Types.DOUBLE)
3271         };
3272         this.setParameters(params);
3273         compile();
3274     } //constructor
3275     public Map myexecute(String name) {
3276         HashMap map = new HashMap();
3277         map.put("NAME", name);
3278         return super.execute(map);
3279     }
3280 } //inner class
3281 } //outerclass
```

3282 **ORM Module:**

- 3283 1. Spring doesn't provide any of its own ORM tool. Simply it provides us a chance to integrate
3284 with any orm tool.
- 3285 2. We can integrate spring with any ORM module in two ways:
 - 3286 1. **Plain:** In plain style almost complete code should be written by programmer.
 - 3287 2. **Template class:** Using template class we can focus only on database operations instead
3288 of focusing on common code.

3289 **Note:** We will see how to integrate spring & hibernate which is a ORM tool. So before we see this
3290 lets see a pure example regarding hibernate. So we can feel comfortable regarding integration of
3291 spring n hibernate.

3292 **Example:** Insertion of a record into employee table of oracle database using java hibernate
3293 application.

3294 **Files to be created:**

- 3295 1. hibernate.cfg.xml [Hibernate configuration file]
- 3296 2. Employee.java [persistent class/POJO]
- 3297 3. Employee.hbm.xml [Hibernate Mapping file]
- 3298 4. Client.java [Java application using hibernate API & setup to interact with Database
3299 software.]

3300 **Step1: Create table in oracle database software as follows:**

3301 create table employee

3302 (

3303 eid number primary key,

3304 firstname varchar2(20),

3305 lastname varchar2(20),

3306 email varchar2(20)

3307);

3308 **Note:** Whether we configure primary key while creating table or not, but its compulsory to have
3309 primary key configuration in mapping file.

3310 **Step2:** Create hibernate configuration file as follows:[Note: For reference open file from Hibernate-
3311 home\etc\hibernate.cfg.xml

3312 Note: Here **Hibernate-home** indicates hibernate setup.[that is extracted folder of hibernate zip file
3313 which is a installation process of hibernate software.]

3314 **hibernate.cfg.xml**

3315 <!DOCTYPE hibernate-configuration PUBLIC

3316 "-//Hibernate/Hibernate Configuration DTD 3.0//EN"

3317 "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

3318 <hibernate-configuration>

3319 <session-factory>

3320 <property name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDriver</property>

3321 <property name="hibernate.connection.url">jdbc:oracle:thin:@localhost:1521:sathya</property>

3322 <property name="hibernate.connection.username">scott</property>

3323 <property name="hibernate.connection.password">tiger</property>

3324 <property name="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</property>

3325 <property name="show_sql">true</property>

3326 <mapping resource="Employee.hbm.xml"/>

3327 </session-factory>

3328 </hibernate-configuration>

3329 **Note:** Hibernate property tag names are fixed but their values will be changed based on version &
3330 name of the Database software. Refer **Hibernate-home\etc\hibernate.properties** file to get each
3331 database specific Hibernate properties values.

3332 **Step3:** Develop POJO class/persistent class

```
3333 //Employee.java
3334 public class Employee implements java.io.Serializable
3335 {
3336     int no;
3337     String fname, lname, email;
3338     public void setNo(int no)
3339     {
3340         this.no = no;
3341     }
3342     public int getNo()
3343     {
3344         return no;
3345     }
3346     public void setFname(String fname)
3347     {
3348         this.fname = fname;
3349     }
3350     public String getFname()
3351     {
3352         return fname;
3353     }
3354     public void setLname(String lname)
3355     {
3356         this.lname = lname;
3357     }
3358     public String getLname(){
3359         return lname;
3360     }
3361     public void setEmail(String email){
3362         this.email = email;
3363     }
3364     public String getEmail(){
3365         return email;
3366     }
3367 }//class
```

3368 **Step 4:** Develop Hibernate Mapping file (Employee.hbm.xml)

3369 **Note:** Take reference file from **Hibernate-home\eg*.hbm.xml**

3370 **Employee.hbm.xml**

3371 <?xml version="1.0"?>

3372 <!DOCTYPE hibernate-mapping PUBLIC

3373 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"

3374 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

3375 <hibernate-mapping>

3376 <class name="Employee" table="employee">

3377 <id name="no" column = "eid"/> <!-- Singular primary key field -->

3378 <property name="fname" column="firstname"/>

3379 <property name="lname" column = "lastname"/>

3380 <property name = "email" column = "email"/>

3381 </class>

3382 </hibernate-mapping>

3383 **Note:** To see these SQL queries as log messages use **hibernate.show_sql** property in hibernate
3384 configuration file as shown below.

3385 <property name = "hibernate.show_sql">true</property>

3386 **Step 5:** Develop hibernate based java application with persistence logic operations. [Insertion of
3387 record]

3388 **Client.java**

3389 import org.hibernate.Session;

3390 import org.hibernate.SessionFactory;

3391 import org.hibernate.cfg.Configuration;

3392 import org.hibernate.Transaction;

3393 public class TestClient

3394 {

3395 public static void main(String[] args) throws Exception

3396 {

3397 //locate and read configuration file

3398 Configuration conf = new Configuration();

3399 conf = conf.configure(); //if we name configuration file differently then we need to specify it
3400 as a argument

3401 **//conf = conf.configure("oracle.cfg.xml");**

3402 //creation of session factory object based on configuration file details

3403 SessionFactory factory = conf.buildSessionFactory();

```

3404 //get session object
3405 Session ses = factory.openSession();
3406 Transaction tx = ses.beginTransaction();
3407 //object creation with data
3408 Employee e1 = new Employee(); //Transient state
3409 e1.setNo(1);
3410 e1.setFname("Vara");
3411 e1.setLname("Prasad");
3412 e1.setEmail("kanakavaraprasad@gmail.com");
3413 //Note: Even after storing data, the object "e1" is in transient state.
3414 //record insertion
3415 ses.save(e1); //now the object "e1" is in persistent state.
3416 tx.commit();
3417 ses.close(); //closes statement and connection objects
3418 //Note: Now the object "e1" is in transient state.
3419 factory.close(); //destroys entire connection pool
3420 }

```

```

3421 }

```

3422 Points to ponder:

- 3423 • conf.configure() method takes "**hibernate.cfg.xml**" file as a default configuration file. If
- 3424 you change the configuration file name as any other name then we need to pass its name as a
- 3425 string argument to the method configure().
- 3426 • Non select operation must be executed as Transactional operations in hibernate based
- 3427 application. If not no error arises but the updation you have to database wont be affected. So
- 3428 mind it.
- 3429 • After completion of task with database software close database connection without fail using
- 3430 ses.close(); If not no error will arise but object will be engaged as a busy connection till end
- 3431 of the particular client application.
- 3432 • ses.save() method inserts record into the database table returns primary key value in the
- 3433 form of object and makes POJO class object to represent inserted record with
- 3434 synchronization between them.
- 3435 • Every hibernate session object contains buffer. In order to flush this buffer use ses.flush()
- 3436 method.
- 3437 • Hibernate software generates database specific sql queries internally for the methods we call
- 3438 from hibernate software to do persistent operations.

3439 **Step 6:** Add the following jar files to class path to recognize hibernate API which is a third party

3440 API. They are as follows:

3441 1.Main Jar file

3442 1. hibernate3.jar [available in hibernate-home directory]

3443 **2. Dependent jar files of “hibernate3.jar”** [available in hibernate-home\lib folder]

3444 1. dom4j-version.jar

3445 2. cglib-version.jar

3446 3. commons-collections-version.jar

3447 4. commons-logging-version.jar

3448 5. jta.jar

3449 6. asm.jar

3450 7. antlr-version.jar

3451 classes12.jar [to support oracle type 4 driver for oracle9i] (available in **oracle-**
3452 **home\ora92\jdbc\lib**)

3453 **Note:**

- 3454 • To get information about required jar files refer _readme.txt file in **hibernate-home\lib**
3455 folder
- 3456 • When java application uses third party API then the API related jar files must be added in
3457 the class path.
- 3458 • It is always recommended to add jar files in my computer environmental variables class
3459 path.

3460 **Step 7:** Compile *.java files using javac tool.

3461 **Step 8:** Execute client application [Client]

3462 **Example: Plain integration of spring & Hibernate**

3463 **// DemoInter.java**

3464 import java.util.Iterator;

3465 public interface DemoInter {

3466 public Iterator getData() throws Exception;

3467 }

3468 **// DemoImpl.java**

3469 import java.util.Iterator;

3470 import org.hibernate.Query;

3471 import org.hibernate.Session;

3472 import org.hibernate.SessionFactory;

3473 public class DemoImpl implements DemoInter{

3474 private SessionFactory factory = null;

3475 public void setFactory(SessionFactory factory) {

3476 System.out.println("Setter of spring class");

3477 this.factory = factory;

```
3478     }
3479     public Iterator getData() throws Exception {
3480         System.out.println("In spring class getData()");
3481         Session ses = factory.openSession();
3482         Query query = ses.createQuery("from User");
3483         Iterator i1 = query.iterate();
3484         //ses.close();
3485         return i1;
3486     }
3487 }
```

3488 **Table creation in oracle:**

```
3489 create table users
3490 (
3491     userid number(5) primary key,
3492     uname varchar2(20),
3493     role varchar2(20)
3494 )
```

3495 **//User.java**

```
3496 public class User {
3497     private int uid;
3498     private String uname, role;
3499     public String getUname() {
3500         return uname;
3501     }
3502     public void setUname(String uname) {
3503         this.uname = uname;
3504     }
3505     public String getRole() {
3506         return role;
3507     }
3508     public void setRole(String role) {
3509         this.role = role;
3510     }
3511     public int getUid() {
3512         return uid;
```

```
3513     }
3514     public void setUid(int uid)
3515     {
3516         this.uid = uid;
3517     }
3518 }
3519 // User.hbm.xml
3520 <?xml version="1.0" encoding="utf-8"?>
3521 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3522 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
3523 <hibernate-mapping>
3524     <class name="User" table="users" schema="SCOTT">
3525         <id name="uid" column = "userid"/>
3526         <property name="uname"/>
3527         <property name="role"/>
3528     </class>
3529 </hibernate-mapping>
3530 // SpringHibernate.xml
3531 <?xml version="1.0" encoding="UTF-8"?>
3532 <beans
3533     xmlns="http://www.springframework.org/schema/beans"
3534     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3535     xsi:schemaLocation="http://www.springframework.org/schema/beans
3536 http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
3537     <bean id = "ds" class =
3538 "org.springframework.jdbc.datasource.DriverManagerDataSource">
3539         <property name="driverClassName">
3540             <value>oracle.jdbc.driver.OracleDriver</value>
3541         </property>
3542         <property name="url">
3543             <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
3544         </property>
3545         <property name="username">
3546             <value>scott</value>
3547         </property>
```

```

3548         <property name="password">
3549             <value>tiger</value>
3550         </property>
3551     </bean>
3552     <bean id = "mySessionFactory" class =
3553     "org.springframework.orm.hibernate3.LocalSessionFactoryBean">
3554         <property name="dataSource" ref = "ds"/>
3555         <property name="mappingResources">
3556             <list>
3557                 <value>User.hbm.xml</value>
3558             </list>
3559         </property>
3560         <property name="hibernateProperties">
3561             <props>
3562                 <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>
3563                 <prop key="show_sql">true</prop>
3564             </props>
3565         </property>
3566     </bean>
3567     <bean id = "d1" class = "DemoImpl">
3568         <property name="factory" ref = "mySessionFactory" />
3569     </bean>
3570 </beans>

```

3571 **Note:**

- 3572 1. We can configure both spring & hibernate properties inside spring configuration file as
3573 shown above.
- 3574 2. In above xml file we have specified **show_sql** as true. But still it work incase of spring. If
3575 we use a separate xml file for hibernate configuration then it supports. We'll see its example
3576 later.

3577 **//Client.java**

```

3578 import java.util.Iterator;
3579 import org.springframework.beans.factory.BeanFactory;
3580 import org.springframework.context.ApplicationContext;
3581 import org.springframework.context.support.ClassPathXmlApplicationContext;
3582 public class Client {
3583     public static void main(String[] args) throws Exception {
3584         ApplicationContext ctx = new

```



```
3585 ClassPathXmlApplicationContext("SpringHibernate.xml");
3586         BeanFactory factory = (BeanFactory)ctx;
3587         DemoInter d = (DemoInter)factory.getBean("d1");
3588         Iterator i1 = d.getData();
3589         while(i1.hasNext())
3590         {
3591             User u1 = (User)i1.next();
3592             System.out.println(u1.getId()+" "+u1.getUsername()+" "+u1.getRole());
3593         }
3594     }
3595 }
3596 //Example: Integration of spring & Hibernate using HiberanteTemplate class
3597 Note: Interface, pojo class, table, mapping file & client program are same as above example.
3598 //DemoImpl.java
3599 import java.util.Iterator;
3600 import java.util.List;
3601 import org.hibernate.Query;
3602 import org.hibernate.Session;
3603 import org.hibernate.SessionFactory;
3604 import org.springframework.orm.hibernate3.HibernateTemplate;
3605 public class DemoImpl implements DemoInter{
3606     private HibernateTemplate ht = null;
3607     public void setHt(HibernateTemplate ht) {
3608         System.out.println("setter of spring");
3609         this.ht = ht;
3610     }
3611     public Iterator getData() throws Exception {
3612         System.out.println("In spring class getData()");
3613         List l = ht.find("from User");
3614         Iterator i1 = l.iterator();
3615         return i1;
3616     }
3617 }
```

```
3618 <!--SpringHibernate.xml-->
3619 <?xml version="1.0" encoding="UTF-8"?>
3620 <beans
3621     xmlns="http://www.springframework.org/schema/beans"
3622     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3623     xsi:schemaLocation="http://www.springframework.org/schema/beans
3624 http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
3625     <bean id = "ds" class =
3626     "org.springframework.jdbc.datasource.DriverManagerDataSource">
3627         <property name="driverClassName">
3628             <value>oracle.jdbc.driver.OracleDriver</value>
3629         </property>
3630         <property name="url">
3631             <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
3632         </property>
3633         <property name="username">
3634             <value>scott</value>
3635         </property>
3636         <property name="password">
3637             <value>tiger</value>
3638         </property>
3639     </bean>
3640     <bean id = "mySessionFactory" class =
3641     "org.springframework.orm.hibernate3.LocalSessionFactoryBean">
3642         <property name="dataSource" ref = "ds"/>
3643         <property name="mappingResources">
3644             <list>
3645                 <value>User.hbm.xml</value>
3646             </list>
3647         </property>
3648         <property name="hibernateProperties">
3649             <props>
3650                 <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>
3651                 <prop key="show_sql">true</prop>
3652             </props>
3653         </property>
```

```

3654     </bean>
3655     <bean id = "template" class = "org.springframework.orm.hibernate3.HibernateTemplate">
3656         <constructor-arg>
3657             <ref bean = "mySessionFactory"/>
3658         </constructor-arg>
3659     </bean>
3660     <bean id = "d1" class = "DemoImpl">
3661         <property name="ht" ref = "template"/>
3662     </bean>
3663 </beans>

```

3664 **Example: Named Query**

3665 **Note:** Instead of writing query in client program we can write it inside mapping file. So the same
 3666 query can be used by multiple files & maintenance will becomes easy.

3667 **Note:** Table, Pojo class, Sping n hibernate configuration file are same as above example

```

3668 <!--User.hbm.xml-->
3669 <?xml version="1.0" encoding="utf-8"?>
3670 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3671 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
3672 <hibernate-mapping>
3673     <class name="User" table="users" schema="SCOTT">
3674         <id name="uid" column = "userid"/>
3675         <property name="uname"/>
3676         <property name="role"/>
3677     </class>
3678     <query name = "users">
3679         from User where uname like ?    <!--hql query-->
3680     </query>
3681 </hibernate-mapping>

```

3682 **//DemoClient.java**

```

3683 import java.util.Iterator;
3684 import java.util.List;
3685 import org.hibernate.Query;
3686 import org.hibernate.Session;
3687 import org.hibernate.SessionFactory;

```

```
3688 import org.springframework.orm.hibernate3.HibernateTemplate;
3689 import org.springframework.beans.factory.BeanFactory;
3690 import org.springframework.context.ApplicationContext;
3691 import org.springframework.context.support.ClassPathXmlApplicationContext;
3692 public class DemoClient
3693 {
3694     private HibernateTemplate ht = null;
3695     public void setHt(HibernateTemplate ht) {
3696         System.out.println("setter of spring");
3697         this.ht = ht;
3698     }
3699     public static void main(String[] args) throws Exception {
3700         ApplicationContext ctx = new
3701 ClassPathXmlApplicationContext("SpringHibernate.xml");
3702         BeanFactory factory = (BeanFactory)ctx;
3703         DemoClient d = (DemoClient)factory.getBean("d1");
3704         List l=d.ht.findByNameQuery("users", new Object[]{"%S%"});
3705         Iterator it=l.iterator();
3706         while(it.hasNext())
3707         {
3708             User ob=(User)it.next();
3709             System.out.print(ob.getUid());
3710             System.out.println(" "+ob.getUname()+" "+ob.getRole());
3711             System.out.println("-----");
3712         }
3713     }
3714 }
```

3715 **Example: Separate configuration files for Spring & Hibernate**

```
3716 <!--spring.cfg.xml-->
3717 <?xml version="1.0" encoding="UTF-8"?>
3718 <beans
3719     xmlns="http://www.springframework.org/schema/beans"
3720     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3721     xsi:schemaLocation="http://www.springframework.org/schema/beans
3722 http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
```

```
3723 <bean id = "ds" class = "org.springframework.jdbc.datasource.DriverManagerDataSource">
3724     <property name="driverClassName">
3725         <value>oracle.jdbc.driver.OracleDriver</value>
3726     </property>
3727     <property name="url">
3728         <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
3729     </property>
3730     <property name="username">
3731         <value>scott</value>
3732     </property>
3733     <property name="password">
3734         <value>tiger</value>
3735     </property>
3736 </bean>
3737 <bean id = "mySessionFactory" class =
3738 "org.springframework.orm.hibernate3.LocalSessionFactoryBean">
3739     <property name="dataSource" ref = "ds"/>
3740     <property name = "configLocation">
3741         <value>hibernate.cfg.xml</value>
3742     </property>
3743 </bean>
3744 <bean id = "template" class = "org.springframework.orm.hibernate3.HibernateTemplate">
3745     <constructor-arg>
3746         <ref bean = "mySessionFactory"/>
3747     </constructor-arg>
3748 </bean>
3749 <bean id = "d1" class = "DemoClient">
3750     <property name="ht" ref = "template"/>
3751 </bean>
3752 </beans>
3753 <!--hibernate.cfg.xml-->
3754 <!DOCTYPE hibernate-configuration PUBLIC
3755     "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
3756     "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
```

```
3757 <hibernate-configuration>
3758     <session-factory>
3759         <property name = "hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</property>
3760         <property name="show_sql">true</property>
3761         <mapping resource="User.hbm.xml"/>
3762     </session-factory>
3763 </hibernate-configuration>
```

3764 **Note:** The rest of resources are same as earlier examples.

3765 **Example: Updating multiple records using HibernateTemplate class**

3766 **//DemoClient.java**

```
3767 import java.util.Iterator;
3768 import java.util.List;
3769 import org.hibernate.Query;
3770 import org.hibernate.Session;
3771 import org.hibernate.SessionFactory;
3772 import org.springframework.orm.hibernate3.HibernateTemplate;
3773 import org.springframework.beans.factory.BeanFactory;
3774 import org.springframework.context.ApplicationContext;
3775 import org.springframework.context.support.ClassPathXmlApplicationContext;
3776 public class DemoClient{
3777     private HibernateTemplate ht = null;
3778     public void setHt(HibernateTemplate ht) {
3779         System.out.println("setter of spring");
3780         this.ht = ht;
3781     }
3782     public static void main(String[] args) throws Exception {
3783         ApplicationContext ctx = new
3784         ClassPathXmlApplicationContext("SpringHibernate.xml");
3785         BeanFactory factory = (BeanFactory)ctx;
3786         DemoClient d = (DemoClient)factory.getBean("d1");
3787         String qry = "update User set role = ? where role = ?";
3788         int res = d.ht.bulkUpdate(qry, new Object[]{"PM", "SSE"});
3789         System.out.println("No. of records updated are: "+res);
3790     }
3791 }
```

3792 **Note:**The rest of resources are same as earlier example.

3793 **Example: Retrieving all records of a table using HibernateTemplate class**

3794 **Table in oracle:**

3795 create table event

3796 (

3797 eid number(5) primary key,

3798 evname varchar2(25)

3799)

3800 **//eventpojo.java**

3801 public class eventpojo

3802 {

3803 Integer eventid;

3804 String eventname;

3805 public eventpojo(){ }

3806 public Integer getEventid() {

3807 return eventid;

3808 }

3809 public void setEventid(Integer eventid) {

3810 this.eventid = eventid;

3811 }

3812 public String getEventname() {

3813 return eventname;

3814 }

3815 public void setEventname(String eventname)

3816 {

3817 this.eventname = eventname;

3818 }

3819 }

3820 **<!-- eventpojo.hbm.xml -->**

3821 <?xml version="1.0"?>

3822 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"

3823 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

```
3824 <hibernate-mapping>
3825     <class name="eventpojo" table="event">
3826         <id name="eventid" column="eid"/>
3827         <property name="eventname" column="evname"/>
3828     </class>
3829 </hibernate-mapping>
3830 //eventdao.java
3831 import org.springframework.orm.hibernate3.*;
3832 import java.util.*;
3833 public class eventdao
3834 {
3835     HibernateTemplate ht;
3836     public void setHt(HibernateTemplate ht)
3837     {
3838         this.ht=ht;
3839     }
3840     public void saveObject(Object o)
3841     {
3842         ht.save(o);
3843     }
3844     public void selectAll()
3845     {
3846         List l=ht.loadAll(eventpojo.class);
3847         Iterator it=l.iterator();
3848         while(it.hasNext())
3849         {
3850             eventpojo ob=(eventpojo)it.next();
3851             System.out.print(ob.getEventid());
3852             System.out.println(" "+ob.getEventname());
3853             System.out.println("-----");
3854         }
3855     }
3856 }
```



```
3857 <!-- spring.cfg.xml -->
3858 <?xml version="1.0" encoding="UTF-8"?>
3859 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
3860 "http://www.springframework.org/dtd/spring-beans.dtd">
3861 <beans>
3862   <bean id="id1" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
3863     <property name="driverClassName">
3864       <value>oracle.jdbc.driver.OracleDriver</value>
3865     </property>
3866     <property name="url">
3867       <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
3868     </property>
3869     <property name="username">
3870       <value>scott</value>
3871     </property>
3872     <property name="password">
3873       <value>tiger</value>
3874     </property>
3875   </bean>
3876   <bean id="id2" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
3877     <property name="dataSource">
3878       <ref bean="id1"/>
3879     </property>
3880     <property name="mappingResources">
3881       <list>
3882         <value>eventpojo.hbm.xml </value>
3883       </list>
3884     </property>
3885     <property name="hibernateProperties">
3886       <props>
3887         <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect </prop>
3888         <prop key="show_sql"> true </prop>
3889       </props>
3890     </property>
3891   </bean>
```

```
3892 <bean id="id3" class="org.springframework.orm.hibernate3.HibernateTemplate">
3893 <property name="sessionFactory">
3894 <ref bean="id2"/>
3895 </property>
3896 </bean>
3897 <bean id="id4" class="eventdao">
3898 <property name="ht">
3899 <ref bean="id3"/>
3900 </property>
3901 </bean>
3902 </beans>
```

```
3903 //client.java
```

```
3904 import org.springframework.core.io.*;
3905 import org.springframework.beans.factory.*;
3906 import org.springframework.beans.factory.xml.*;
3907 public class client
3908 {
3909     public static void main(String[] args)
3910     {
3911         BeanFactory factory=new XmlBeanFactory(new
3912         ClassPathResource("spring.cfg.xml"));
3913         eventdao ed =(eventdao)factory.getBean("id4");
3914         eventpojo ep=new eventpojo();
3915         ep.setEventid(new Integer(2));
3916         ep.setEventname("WindowEvent");
3917         ed.saveObject(ep);
3918         ed.selectAll();
3919         System.out.println("completed...");
3920     }
3921 }
```

```
3922 JEE Module:
```

```
3923 Example: TimerTask [Job Scheduling] When ever we need to execute any task at particular time
3924 either for one time or for every particular period then we can use scheduling as follows:
```

```
3925 //task1.java [Core java program]
```

```
3926 import java.util.*;
```

```
3927 public class task1 extends TimerTask
3928 {
3929     public void run()
3930     {
3931         System.out.println("welcome to sathya ");
3932     }
3933 }
3934 //scheduledemo.java
3935 import java.util.*;
3936 public class scheduledemo
3937 {
3938     public static void main(String args[])
3939     {
3940         Timer t1=new Timer();
3941         t1.schedule(new task1(),5000); //initial delay
3942         t1.schedule(new task1(),5000,3000); //initial delay, periodical delay
3943     }
3944 }
3945 Example: Multiple Tasks
3946 Note: "task1" class is same as above.
3947 //task2 .java
3948 import java.util.*;
3949 import java.io.*;
3950 public class task2 extends TimerTask
3951 {
3952     public void run()    {
3953         try
3954         {
3955             FileWriter fp=new FileWriter("demofile.txt", true);
3956             fp.write(new Date().toString()+"\n");
3957             fp.close();
3958             System.out.println("data stored");
3959         }catch(Exception e){}
3960     }
3961 }
```

```
3962 <!-- Timer.xml -->
3963 <?xml version="1.0" encoding="UTF-8"?>
3964 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
3965 "http://www.springframework.org/dtd/spring-beans.dtd">
3966 <beans>
3967 <bean id="job1" class="task1" />
3968 <bean id="job2" class="task2" />
3969 <bean id="schedule1" class="org.springframework.scheduling.timer.ScheduledTimerTask" >
3970     <property name="delay"> <value>5000</value></property>
3971     <property name="period"><value>2000</value></property>
3972     <property name="timerTask" >
3973         <ref bean="job1" />
3974     </property>
3975 </bean>
3976 <bean id="schedule2" class="org.springframework.scheduling.timer.ScheduledTimerTask" >
3977     <property name="delay"> <value>5000</value></property>
3978     <property name="period"><value>3000</value></property>
3979     <property name="timerTask" >
3980         <ref bean="job2" />
3981     </property>
3982 </bean>
3983 <bean id="stt" class="org.springframework.scheduling.timer.TimerFactoryBean">
3984     <property name="scheduledTimerTasks">
3985         <list>
3986             <ref bean="schedule1" />
3987             <ref bean="schedule2" />
3988         </list>
3989     </property>
3990 </bean>
3991 </beans>
3992 //scheduledemo.java
3993 import org.springframework.context.*;
3994 import org.springframework.context.support.*;
3995 import java.io.*;
```

```
3996 public class scheduledemo
3997 {
3998     public static void main(String args[]) throws IOException
3999     {
4000         ApplicationContext ctx=new FileSystemXmlApplicationContext("Timer.xml");
4001     }
4002 }
```

4003 **Example: Scheduling using spring classes.**

4004 **//democlass1.java**

```
4005 public class democlass1
4006 {
4007     public void method1()
4008     {
4009         System.out.println("welcome to sathya");
4010     }
4011 }
```

4012 **//democlass2.java**

```
4013 import java.io.*;
4014 public class democlass2
4015 {
4016     public void method2()
4017     {
4018         try{
4019             FileWriter fp=new FileWriter("demofile.txt",true);
4020             fp.write(new Date().toString()+"\n");
4021             fp.close();
4022             System.out.println("data stored");
4023         }catch(Exception e){}
4024     }
4025 }
```

4026 **Note:** Our classes are not extending any predefined class

4027 **Injection Process:**

```
4028 democlass1 -> MethodInvokingTimerTaskFactoryBean -> ScheduledTimerTask ->
4029 TimerFactoryBean
```

```
4030 <!-- Timer.xml -->
4031 <?xml version="1.0" encoding="UTF-8"?>
4032 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
4033 "http://www.springframework.org/dtd/spring-beans.dtd">
4034 <beans>
4035 <bean id="c1" class="democlass1" />
4036 <bean id="c2" class="democlass2" />
4037 <bean id="job1"
4038 class="org.springframework.scheduling.timer.MethodInvokingTimerTaskFactoryBean>
4039 <property name="targetObject">
4040 <ref bean="c1" />
4041 </property>
4042 <property name="targetMethod">
4043 <value>method1</value>
4044 </property>
4045 </bean>
4046 <bean id="job2"
4047 class="org.springframework.scheduling.timer.MethodInvokingTimerTaskFactoryBean>
4048 <property name="targetObject">
4049 <ref bean="c2" />
4050 </property>
4051 <property name="targetMethod">
4052 <value>method2</value>
4053 </property>
4054 </bean>
4055 <bean id="schedule1" class="org.springframework.scheduling.timer.ScheduledTimerTask" >
4056 <property name="delay"> <value>5000</value></property>
4057 <property name="period"><value>2000</value></property>
4058 <property name="timerTask" >
4059 <ref bean="job1" />
4060 </property>
4061 </bean>
4062 <bean id="schedule2" class="org.springframework.scheduling.timer.ScheduledTimerTask" >
4063 <property name="delay"> <value>5000</value></property>
4064 <property name="period"><value>3000</value></property>
```

```

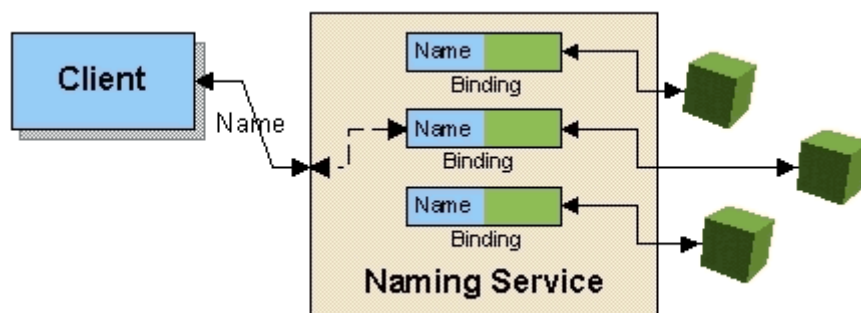
4065     <property name="timerTask" >
4066         <ref bean="job2" />
4067     </property>
4068 </bean>
4069 <bean id="stt" class="org.springframework.scheduling.timer.TimerFactoryBean">
4070     <property name="scheduledTimerTasks">
4071         <list>
4072             <ref bean="schedule1" />
4073             <ref bean="schedule2" />
4074         </list>
4075     </property>
4076 </bean>
4077 </beans>

```

4078 **Note:** Client program is same like earlier example.

4079 **JNDI: [Java Naming Directory Interface]**

- 4080 1. When ever we need to give global visibility for any java object then we can use JNDI. So
- 4081 that any program can bind the object with server from any where through out the world. The
- 4082 same object can be used by some other program through JNDI.
- 4083 2. JNDI can act as a mediator between server program & client program to exchange objects.
- 4084 3. To work with JNDI we need to start any application server like weblogic or Jboss.
- 4085 4. We can do following JNDI operations
- 4086 1. bind(id, obj):
- 4087 1. We can bind any object with some identification name.
- 4088 2. If the object is already binded immediately it throws BindException
- 4089 2. rebind(id, obj)
- 4090 1. Same as bind() method
- 4091 2. But if the object is already exists it replace the existing object with new object.
- 4092 3. unbind(id): Specific id related object will be immediately removed.
- 4093 4. lookup(id): When ever we need to fetch the object through JNDI then we can use it.
- 4094 5. list(): returns all the registered objects



4095 **Example: Plain JNDI without using Spring**4096 **Note:**4097 **1.** start -> programs -> bea products -> example -> **weblogic server** -> start examples server4098 **2.** set the classpath for **weblogic.jar**4099 **//TestJndi.java**

4100 import javax.naming.*;

4101 import java.util.*;

4102 public class TestJndi

4103 {

4104 public static void main(String[] args) throws Exception

4105 {

4106 //prepare Jndi properties

4107 Hashtable ht=new Hashtable();

4108 ht.put(Context.INITIAL_CONTEXT_FACTORY,

4109 "weblogic.jndi.WLInitialContextFactory");

4110 ht.put(Context.PROVIDER_URL,"t3://localhost:7001");

4111 InitialContext ic=new InitialContext(ht);

4112 // ic represents connectivity with Naming/Directory registry S/w

4113 //bind operation

4114 ic.rebind("today",new Date());

4115 ic.bind("banana",new String("yellow"));

4116 //listing

4117 System.out.println("---->Listing (after binding)");

4118 NamingEnumeration e=ic.list("");

4119 while(e.hasMore())

4120 {

4121 NameClassPair np=(NameClassPair)e.next(); // each binding is accessed

4122 System.out.println(np.getName()+"-----> "+np.getClassName());

4123 }

4124 }//main

4125 }//class


```
4126 //TestJndi1.java
4127 import javax.naming.*;
4128 import java.util.*;
4129 public class TestJndi1
4130 {
4131     public static void main(String[] args) throws Exception
4132     {
4133         //prepare Jndi properties
4134         Hashtable ht=new Hashtable();
4135         ht.put(Context.INITIAL_CONTEXT_FACTORY,
4136             "weblogic.jndi.WLInitialContextFactory");
4137         ht.put(Context.PROVIDER_URL,"t3://localhost:7001");
4138         InitialContext ic=new InitialContext(ht);
4139         // ic represents connectivity with Naming/Directory registry S/w
4140         //listing
4141         System.out.println("Listing");
4142         NamingEnumeration e=ic.list("");
4143         while(e.hasMore())
4144         {
4145             NameClassPair np=(NameClassPair)e.next();
4146             System.out.println(np.getName()+" ---> "+np.getClassName());
4147         }
4148         //lookup
4149         System.out.println("lookup");
4150         Object obj = ic.lookup("today");
4151         Date d1=(Date)obj;
4152         System.out.println("date is "+d1.toString());
4153         obj = ic.lookup("banana");
4154         String color=(String)obj;
4155         System.out.println("Banana color is "+color);
4156         //unbind operation
4157         ic.unbind("banana");
4158         //listing
4159         System.out.println("\n\n\nListing-----");
4160         e=ic.list("");
```

```
4161         while(e.hasMore())
4162             {
4163                 NameClassPair np=(NameClassPair)e.next();
4164                 System.out.println(np.getName()+" ---> "+np.getClassName());
4165             }
4166     } //main
4167 } //class
```

4168 **//To execute this application with jboss add jbossall-client.jar in the classpath**

```
4169 ht.put(Context.INITIAL_CONTEXT_FACTORY,
4170         "org.jnp.interfaces.NamingContextFactory");
4171 ht.put(Context.PROVIDER_URL,"jnp://localhost:1099");
```

4172 **Note:** The rest of steps are same as above program.

4173 **Example: Spring JNDI**

```
4174 <?xml version="1.0" encoding="UTF-8"?>
4175 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
4176 "http://www.springframework.org/dtd/spring-beans.dtd">
4177 <beans>
4178     <bean id="jt" class="org.springframework.jndi.JndiTemplate">
4179         <constructor-arg>
4180             <props>
4181                 <prop key="java.naming.factory.initial">
4182                     weblogic.jndi.WLInitialContextFactory</prop>
4183                 <prop key="PROVIDER_URL">t3://localhost:7001</prop>
4184             </props>
4185         </constructor-arg>
4186     </bean>
4187     <bean id="test" class="JndiTemplateTest">
4188         <property name="template"><ref bean="jt"/></property>
4189     </bean>
4190 </beans>
```

4191 **//JndiTemplateTest.java**

```
4192 import org.springframework.context.support.*;
4193 import org.springframework.jndi.*;
4194 import java.util.*;
```

```
4195 public class IndiTemplateTest
4196 {
4197     static IndiTemplate template;
4198     public void setTemplate(IndiTemplate template)
4199     {
4200         this.template=template;
4201     }
4202     public static void main(String[] args) throws Exception
4203     {
4204         FileSystemXmlApplicationContext ctx=
4205             new FileSystemXmlApplicationContext("DemoCfg.xml");
4206         System.out.println("bind()......");
4207         template.bind("today1",new Date());
4208         template.bind("apple",new String("it is red"));
4209         System.out.println("lookup operation");
4210         Date d1=(Date)template.lookup("today1");
4211         System.out.println("(today1)lookup value is"+d1.toString());
4212         template.unbind("today1");
4213         template.rebind("apple",new StringBuffer("it is green"));
4214         System.out.println("lookup operation");
4215         StringBuffer s1=(StringBuffer)template.lookup("apple");
4216         System.out.println("(apple)lookup value is"+s1.toString());
4217     }
4218 }
```

Note:

- 4220 1. start -> programs -> bea products -> example -> **weblogic server** -> start examples server
- 4221 2. set the classpath for weblogic.jar

Example: Plain RMI [Remote Method Invocation]

- 4223 1. Its a distributed application
- 4224 2. Server side implementation class will contain logics in the form of methods.
- 4225 3. Server side business logic methods will be called by client program

```
4226 //ServerInter.java
4227 import java.rmi.*;
4228 public interface ServerInter extends Remote
4229 {
4230     public String wish(String uname) throws RemoteException;
4231 }
4232 // ServerImpl.java
4233 import java.rmi.*;
4234 import java.rmi.server.*;
4235 public class ServerImpl extends UnicastRemoteObject implements ServerInter
4236 {
4237     public ServerImpl() throws RemoteException {}
4238     public String wish(String uname) throws RemoteException
4239     {
4240         return "Good Afternoon "+uname;
4241     }
4242 }
4243 //Server.java
4244 import java.rmi.*;
4245 import java.net.*;
4246 public class Server
4247 {
4248     public static void main(String args[]) throws RemoteException, MalformedURLException
4249     {
4250         ServerInter i1 = new ServerImpl();
4251         Naming.rebind("rmi://localhost:1099/first", i1);
4252         System.out.println("server started..");
4253     }
4254 }
```

```
4255 //Client.java
4256 import java.rmi.*;
4257 import java.net.*;
4258 public class Client
4259 {
4260     public static void main(String args[]) throws
4261     RemoteException,NotBoundException,MalformedURLException
4262     {
4263         ServerInter i1=(ServerInter)Naming.lookup("rmi://localhost:1099/first");
4264         System.out.println(i1.wish("Sai"));
4265     }
4266 }
4267 RmiServer
4268 -----
4269     1) In server machine
4270         1. ServerInter.class
4271         2. ServerImpl.class
4272         3. Server.class
4273     2) In client Machine
4274         1. ServerInter.class
4275         2. Client.class
4276         3. ServerImpl_stub.class [We'll get this through rmic (rmi compilation)]
4277 Compilation:
4278     javac *.java
4279     rmic ServerImpl
4280     start rmiregistry [automatically it starts a rmi registry window. It will be a balck console
4281 window. Dont close this. Simply minimise & work. Same jndi operation can be done here. So all
4282 the registered objects will be stored in rmi registry]
4283 Execute Server program:     java Server
4284 RmiClient
4285 -----
4286     ( copy ServerImpl_stub.class & ServerInter.class)
4287     javac Client.java
4288     java Client
```

4289 **Example: Spring RMI**

4290 **Note:** Server side programs & set up every thing is same. But client side programs will be changed
4291 as shown below.

4292 **//ClientInter.java**

4293 public interface ClientInter

4294 {

4295 public ServerInter getServerInter();

4296 }

4297 **// ClientImpl.java**

4298 public class ClientImpl implements ClientInter

4299 {

4300 private ServerInter si;

4301 public void setSi(ServerInter si)

4302 {

4303 this.si = si;

4304 }

4305 public firstinter getServerInter()

4306 {

4307 return si;

4308 }

4309 }

4310 **<!-- first.xml -->**

4311 <?xml version="1.0" encoding="UTF-8"?>

4312 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"

4313 "http://www.springframework.org/dtd/spring-beans.dtd">

4314 <beans>

4315 <bean id="pfb" class="org.springframework.remoting.rmi.RmiProxyFactoryBean" >

4316 <property name="serviceUrl" >

4317 <value>rmi://localhost:1099/first</value>

4318 </property>

4319 <property name="serviceInterface">

4320 <value>ServerInter</value>

4321 </property>

4322 </bean>

```
4323 <bean id="c1" class="ClientImpl" >
```

```
4324 <property name="si">
```

```
4325 <ref bean="pfb" />
```

```
4326 </property>
```

```
4327 </bean>
```

```
4328 </beans>
```

```
4329 //Client.java
```

```
4330 import org.springframework.beans.factory.*;
```

```
4331 import org.springframework.context.*;
```

```
4332 import org.springframework.context.support.*;
```

```
4333 import java.rmi.*;
```

```
4334 public class Client
```

```
4335 {
```

```
4336     public static void main(String args[]) throws RemoteException
```

```
4337     {
```

```
4338         ApplicationContext ctx=new FileSystemXmlApplicationContext("first.xml");
```

```
4339         BeanFactory factory=(BeanFactory)ctx;
```

```
4340         ClientInter i1=(ClientInter)factory.getBean("c1");
```

```
4341         ServerInter si=i1.getServerInter();
```

```
4342         System.out.println(si.wish("Sai"));
```

```
4343     }
```

```
4344 }
```

```
4345 Example: HttpInvoker
```

```
4346 Note:
```

```
4347     1. Our business logic classes can be placed in classes folder of a web application.
```

```
4348     2. Our business logic classes need not to extend any predefined class or need not to implement  
4349     any predefined interface. They can be perfect POJO's.
```

```
4350     3. Same like a common web application. But through concept of HttpInvoker any kind of java  
4351     applicaiton like console program, desktop application etc., can give request from any where  
4352     through out the world and can get response.
```

```
4353 //httpinter.java
```

```
4354 public interface httpinter
```

```
4355 {
```

```
4356     public String getWeek();
```

```
4357     public String getMonth();
```

```
4358 }
```

```
4359 // httpimpl.java
4360 import java.util.*;
4361 public class httpimpl implements httpinter
4362 {
4363     public String getWeek()
4364     {
4365         String weeks[]=new String[]
4366 {"","Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"};
4367         Calendar c1=Calendar.getInstance();
4368         int w=c1.get(Calendar.DAY_OF_WEEK);
4369         return "current week :"+weeks[w];
4370     }
4371     public String getMonth()
4372     {
4373         String months[]=new String[]
4374 {"January","February","March","April","May","June","July","August","September","October","No
4375 vember","December"};
4376         Calendar c1=Calendar.getInstance();
4377         int m=c1.get(Calendar.MONTH);
4378         return "current month :"+months[m];
4379     }
4380 }
4381 <!-- web.xml -->
4382 <web-app>
4383     <servlet>
4384         <servlet-name>remoting</servlet-name>
4385         <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
4386         <load-on-startup>1</load-on-startup>
4387     </servlet>
4388     <servlet-mapping>
4389         <servlet-name>remoting</servlet-name>
4390         <url-pattern>/http/*</url-pattern>
4391     </servlet-mapping>
4392 </web-app>
```



```
4393 <!-- remoting-servlet.xml -->
4394 <?xml version="1.0" encoding="UTF-8"?>
4395 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
4396 "http://www.springframework.org/dtd/spring-beans.dtd">
4397 <beans>
4398     <bean id="h1" class="httpimpl" />
4399     <bean name="/DemoService"
4400 class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter" >
4401         <property name="service">
4402             <ref bean="h1" />
4403         </property>
4404         <property name="serviceInterface" >
4405             <value>httpinter</value>
4406         </property>
4407     </bean>
4408 </beans>
4409 //Client program Setup
4410 //clientinter.java
4411 public interface clientinter
4412 {
4413     public httpinter getInter();
4414 }
4415 //clientimpl.java
4416 public class clientimpl implements clientinter
4417 {
4418     private httpinter httpin;
4419     public void setHttpin(httpinter f1)
4420     {
4421         httpin=f1;
4422     }
4423     public httpinter getInter()
4424     {
4425         return httpin;
4426     }
4427 }
```

```
4428 <!-- Demo.xml -->
4429 <?xml version="1.0" encoding="UTF-8"?>
4430 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
4431 "http://www.springframework.org/dtd/spring-beans.dtd">
4432 <beans>
4433 <bean id="pfb"
4434 class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBeanhttp://localhost:2010/httpserver/http/DemoService</value>
4437     </property>
4438     <property name="serviceInterface" >
4439         <value>httpinter</value>
4440     </property>
4441 </bean>
4442 <bean id="c1" class="clientimpl" >
4443     <property name="httpin">
4444         <ref bean="pfb"/>
4445     </property>
4446 </bean>
4447 </beans>
4448 //DemoClient.java
4449 import org.springframework.beans.factory.*;
4450 import org.springframework.context.*;
4451 import org.springframework.context.support.*;
4452 public class DemoClient{
4453     public static void main(String args[])
4454     {
4455         ApplicationContext ctx=new FileSystemXmlApplicationContext("Demo.xml");
4456         BeanFactory factory=(BeanFactory)ctx;
4457         clientinter c1=(clientinter)factory.getBean("c1");
4458         httpinter h1=c1.getInter();
4459         System.out.println(h1.getWeek());
4460         System.out.println(h1.getMonth());
4461     }
4462 }
```

4463 **Note:**

- 4464 1. Deploy server side ssetup in the form of a web application inside any web server like tomcat
4465 or application server like weblogic & start the server & execute the client program.
- 4466 2. Do not forget to place main jar file & dependency jar file inside "**lib**" folder of a web
4467 application.

4468 **Example: Working with Spring James Mail Server**

4469 **Note:**

- 4470 1. The mails what we send & recieve will not be stored in database server like oracle. They
4471 will be stored inside mail server.
- 4472 2. We can have commerical servers & also free servers
- 4473 3. James mail server is free server. But we can send mails only in same network. We cannot
4474 send mails to gmail or yahoomail etc.,
- 4475 4. Microsoft exchange server is commercial server

4476 **Working with mail server:**

- 4477 1. copy james server in C:\james-2.2.0. [in any root directory]
- 4478 2.start james server by clicking on C:\james-2.2.0\bin\run.bat
- 4479 3. start -> run -> telnet -> open localhost 4555
- 4480 login id: root
- 4481 password: root
- 4482 type: help -> it gives u a list of commands with explanation.

4483 **Note:** While you are typing username or password if any mistake ignore it & retype again. It will
4484 also recognise back space or space as a part of password. So be careful while typing.

4485 **//demointer.java**

4486 public interface demointer

4487 {

4488 public void demo();

4489 }

4490 **// demoimpl.java**

4491 import org.springframework.mail.*;

4492 import org.springframework.mail.javamail.*;

4493 public class demoimpl implements demointer

4494 {

4495 private SimpleMailMessage message;

4496 private JavaMailSenderImpl sender;

```
4497 public void setMessage(SimpleMailMessage msg)
4498 {
4499     message=msg;
4500 }
4501 public void setSender(JavaMailSenderImpl sen)
4502 {
4503     sender=sen;
4504 }
4505 public void demo()
4506 {
4507     message.setText("this is first spring mail demo..");
4508     sender.send(message);
4509 }
4510 }
4511 <!-- mail.xml -->
4512 <?xml version="1.0" encoding="UTF-8"?>
4513 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
4514 "http://www.springframework.org/dtd/spring-beans.dtd">
4515 <beans>
4516     <bean id="msg" class="org.springframework.mail.SimpleMailMessage" >
4517         <property name="to"><value>sravanthi</value></property>
4518         <property name="from"><value>anil</value></property>
4519         <property name="subject"><value>spring mail1</value></property>
4520     </bean>
4521     <bean id="s1" class="org.springframework.mail.javamail.JavaMailSenderImpl">
4522 <!--     <property name="host"><value>m8.hamarashehar.com</value></property> -->
4523         <property name="host"><value>localhost</value></property>
4524     </bean>
4525     <bean id="d1" class="demoimpl" >
4526         <property name="message">
4527             <ref bean="msg" />
4528         </property>
```

```
4529     <property name="sender">
4530         <ref bean="s1" />
4531     </property>
4532 </bean>
4533 </beans>
4534 //demoapp.java
4535 import org.springframework.context.*;
4536 import org.springframework.context.support.*;
4537 import org.springframework.beans.factory.*;
4538 public class demoapp
4539 {
4540     public static void main(String args[])
4541     {
4542         ApplicationContext ctx=new FileSystemXmlApplicationContext("mail.xml");
4543         BeanFactory factory=(BeanFactory)ctx;
4544         demointer d1=(demointer)factory.getBean("d1");
4545         d1.demo();
4546         System.out.println("mail sent..");
4547     }
4548 }
4549 //Read mail program
4550 //ReceiveMail.java
4551 // Java Application to receive an E-mail using JAVA Mail API.
4552 import javax.mail.*;
4553 import javax.mail.internet.*;
4554 import java.util.*;
4555 public class ReceiveMail
4556 {
4557     public static void main(String []args) throws Exception
4558     {
4559         Properties p=new Properties();
4560         p.put("mail.transport.protocol","pop");
4561         p.put("mail.pop.host","localhost");
4562         p.put("mail.pop.port","110");
4563         Session session=Session.getInstance(p);
```

```

4564     Store store = session.getStore("pop3");
4565     store.connect("localhost","user_name","pass_word");
4566     Folder myinbox=store.getFolder("INBOX");
4567     myinbox.open(Folder.READ_ONLY);
4568     System.out.println("No.of messages in the inbox:"+myinbox.getMessageCount());
4569     Message message=myinbox.getMessage(1);
4570     message.writeTo(System.out);
4571     myinbox.close(false);
4572     store.close();
4573 }
4574 }
4575 //Example: Spring Mail through gmail composing mails to any gmail id or yahoo mail id
4576 //OrderManager.java
4577 public interface OrderManager {
4578     void placeOrder();
4579 }
4580 //JavaMailOrderManager.java
4581 import javax.mail.Message;
4582 import javax.mail.internet.InternetAddress;
4583 import javax.mail.internet.MimeMessage;
4584 import org.springframework.mail.javamail.JavaMailSender;
4585 import org.springframework.mail.javamail.JavaMailSenderImpl;
4586 public class JavaMailOrderManager implements OrderManager {
4587     private JavaMailSender mailSender;
4588     public void setMailSender(JavaMailSender mailSender) {
4589         this.mailSender = mailSender;
4590     }
4591     public void placeOrder() {
4592         try{
4593             MimeMessage msg = mailSender.createMimeMessage();
4594             msg.setSubject("Thank you..... :) ");
4595             msg.setContent("Dear Customer thank you for placing order. ","text/plain");
4596             msg.setRecipient(Message.RecipientType.TO, new
4597 InternetAddress("s.varaprasad@yahoo.co.in"));
4598             this.mailSender.send(msg);

```

```
4599         }
4600         catch(Exception ex) {
4601             ex.printStackTrace();
4602         }
4603     }
4604 }
4605 <!-- mailsender.xml -->
4606 <?xml version="1.0" encoding="UTF-8"?>
4607 <beans xmlns="http://www.springframework.org/schema/beans"
4608     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4609     xsi:schemaLocation="http://www.springframework.org/schema/beans
4610     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
4611     <bean id="ms" class="org.springframework.mail.javamail.JavaMailSenderImpl">
4612         <property name="host" value="smtp.gmail.com"/>
4613         <property name="username" value="kanakavaraprasad@gmail.com"/> <!--Mails will be sent
4614 with same id-->
4615         <property name="password" value="Secret_Pls"/> <!-- Here we need to give original password
4616 of gmail-->
4617         <property name="javaMailProperties">
4618             <props>
4619                 <prop key="mail.transport.protocol">smtp</prop>
4620                 <prop key="mail.smtp.auth">true</prop>
4621                 <prop key="mail.smtp.port">465</prop>
4622                 <prop key="mail.smtp.socketFactory.port">465</prop>
4623                 <prop key="mail.smtp.socketFactory.class">javax.net.ssl.SSLSocketFactory</prop>
4624                 <prop key="mail.smtp.socketFactory.fallback">>false</prop>
4625                 <prop key="mail.smtp.quitwait">>false</prop>
4626                 <prop key="mail.smtp.starttls.enable">>true</prop>
4627             </props>
4628         </property>
4629     </bean>
4630     <bean id="orderManager" class="JavaMailOrderManager">
4631         <property name="mailSender" ref="ms"/>
4632     </bean>
4633 </beans>
```

```
4634 //client.java
4635 import org.springframework.core.io.*;
4636 import org.springframework.beans.factory.*;
4637 import org.springframework.beans.factory.xml.*;
4638 class client
4639 {
4640     public static void main(String[] args)
4641     {
4642         BeanFactory factory=new XmlBeanFactory(new
4643         FileSystemResource("mailsender.xml"));
4644         OrderManager om =(OrderManager)factory.getBean("orderManager");
4645         om.placeOrder();
4646     }
4647 }
```

4648 **Note:** Set the classpath for following jar files

- 4649 1. mail.jar
- 4650 2. activation.jar

4651 **Note:** When its needed to send any attachments add the following code:

```
4652 MimeMessage msg = sender.createMimeMessage();
4653 MimeMessageHelper helper = new MimeMessageHelper(msg, true);
4654 FileSystemResource img = new FileSystemResource(new File("alice.gif"));
4655 helper.addAttachment("alice", img);
4656 sender.send(msg);
```

4657 **Note:** In the above code "sender" is reference of "JavaMailSender".

4658 **If wee need to foramt data through html, then we can do it as follows:**

```
4659 helper.setText("<html><head></head><body text=green><h1>Hello World!"
4660               + "</h1></body></html>", true);
```

4661 **AOP Module[Aspect Oriented Programming]:**

4662 **Aspect:** An *Aspect* is a functionality or a feature that *cross-cuts [commonly shared] over* objects.

4663 Eg: example, **Logging** and **Transaction Management** are the aspects.

```
4664 public void businessOperation(BusinessData data){
4665     // Logging
4666     logger.info("Business Method Called");
4667     // Transaction Management Begin
```



```
4668 transaction.begin();
4669 // Do the original business operation here
4670 transaction.end();
4671 }
```

4672 **Join Points** defines the various Execution Points where an Aspect can be applied. For example,
4673 consider the following piece of code,

```
4674 public void someBusinessOperation(BusinessData data){
```

```
4675     //Method Start -> Possible aspect code here like logging.
```

```
4676     try{
4677         // Original Business Logic here.
4678     }catch(Exception exception){
4679         // Exception -> Aspect code here when some exception is raised.
4680     }finally{
4681         // Finally -> Even possible to have aspect code at this point too.
4682     }
```

```
4683     // Method End -> Aspect code here in the end of a method.
4684 }
```

4685 It is not necessary that an Aspect should be applied to all the possible Join Points.

4686 **Pointcut**: on which Join Points the Aspects will be applied is called as pointcut.

4687 **Advice** is the code that implements the **Aspect**

4688 Other Aspect Oriented Programming Languages also provide support for Field Aspect, i.e.
4689 intercepting a field before its value gets affected. But Spring provides support only Method Aspect.
4690 The following are the different types of aspects available in Spring.

- 4691 • Before Advice : Executes before business logic method
- 4692 • After Advice : Executes after business logic method
- 4693 • Throws Advice : Executes when exception is thrown in a business logic class
- 4694 • Around Advice: Executes before & after business logic method.

4695 **Around Advice**: When ever we need to perform any pre processing logic or post processing logic
4696 for our business logic methods then we can use it by implementing an interface named as
4697 `MethodInterceptor`.

```
4698 //Example: AroundAdvice
4699 //IBusinessLogic.java
4700 public interface IBusinessLogic
4701 {
4702     public void method1();
4703     public void method2();
4704 }
4705 // BusinessLogic.java
4706 public class BusinessLogic implements IBusinessLogic
4707 {
4708     public void method1()
4709     {
4710         System.out.println("execution of method1");
4711     }
4712     public void method2()
4713     {
4714         System.out.println("execution of method2");
4715     }
4716 }
4717 // AroundAdvice.java
4718 import org.aopalliance.intercept.MethodInvocation;
4719 import org.aopalliance.intercept.MethodInterceptor;
4720 public class AroundAdvice implements MethodInterceptor
4721 {
4722     public Object invoke(MethodInvocation i1) throws Throwable
4723     {
4724         System.out.println("Hi.....");
4725         i1.proceed();
4726         System.out.println("Goodbye! ");
4727         return null;
4728     }
4729 }
4730 <!-- springconfig.xml -->
4731 <beans>
```

```
4732 <bean id="targetobj" class="BusinessLogic"/>
4733 <bean id="adv" class="AroundAdvice"/>

4734 <bean id="advr" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
4735     <property name="advice">
4736         <ref local="adv"/>
4737     </property>
4738     <property name="pattern">
4739         <value>.*</value>
4740     </property>
4741 </bean>

4742 <bean id="proxyobj" class=
4743     "org.springframework.aop.framework.ProxyFactoryBean">
4744
4745     <property name="proxyInterfaces">
4746         <value>IBusinessLogic</value>
4747     </property>

4748     <property name="target">
4749         <ref local="targetobj"/>
4750     </property>

4751     <property name="interceptorNames">
4752         <list>
4753             <value>advr</value>
4754         </list>
4755     </property>
4756 </bean>
4757 </beans>

4758 //MainApplication.java
4759 import org.springframework.context.ApplicationContext;
4760 import org.springframework.context.support.FileSystemXmlApplicationContext;
4761 public class MainApplication
4762 {
```

```
4763 public static void main(String [] args)
4764 {
4765     ApplicationContext ctx=
4766         new FileSystemXmlApplicationContext("springconfig.xml");
4767     IBusinessLogic testObject=(IBusinessLogic) ctx.getBean("proxyobj");
4768     testObject.method1();
4769     testObject.method2();
4770 }
4771 }
4772 //Exmple: Buisness logic method with return value
4773 //IBusinessLogic.java
4774 public interface IBusinessLogic
4775 {
4776     public int method1();
4777 }
4778 // BusinessLogic.java
4779 public class BusinessLogic implements IBusinessLogic
4780 {
4781     public int method1()
4782     {
4783         System.out.println("execution of method1");
4784         return 1000;
4785     }
4786 }
4787 // AroundAdvice.java
4788 import org.aopalliance.intercept.MethodInvocation;
4789 import org.aopalliance.intercept.MethodInterceptor;
4790 public class AroundAdvice implements MethodInterceptor
4791 {
4792     public Object invoke(MethodInvocation i1) throws Throwable
4793     {
4794         System.out.println("good mng everybody..");
4795         int n=((Integer)i1.proceed()).intValue();
4796         System.out.println("Goodbye! ");
4797         if(n <=100)
```

```
4798         return new Integer(0);
4799     else
4800         return new Integer(n);
4801     //return new Integer(2000);
4802 }
4803 }
```

4804 **//Note:** What ever the value returned by invoke() method will be returned to client proram. So we
4805 can use it when we need to add some value added tax or to give some discount etc..,

4806 **<!-- springconfig.xml -->**

4807 **<?xml version="1.0" encoding="UTF-8"?>**

4808 **<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"**

4809 **"http://www.springframework.org/dtd/spring-beans.dtd">**

4810 **<beans>**

4811 **<bean id="beanTarget" class="BusinessLogic"/>**

4812 **<bean id="ad" class="AroundAdvice"/>**

4813 **<bean id="advisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">**

4814 **<property name="advice">**

4815 **<ref local="ad"/>**

4816 **</property>**

4817 **<property name="pattern">**

4818 **<value>.*</value>**

4819 **</property>**

4820 **</bean>**

4821 **<bean id="proxyobj" class=**

4822 **"org.springframework.aop.framework.ProxyFactoryBean">**

4823 **<property name="proxyInterfaces">**

4824 **<value>IBusinessLogic</value>**

4825 **</property>**

4826 **<property name="target">**

4827 **<ref local="beanTarget"/>**

4828 **</property>**

4829 **<property name="interceptorNames">**

4830 **<list>**

4831 **<value>advisor</value>**

```
4832     </list>
4833     </property>
4834 </bean>
4835 </beans>
4836 //MainApplication.java
4837 import org.springframework.context.ApplicationContext;
4838 import org.springframework.context.support.FileSystemXmlApplicationContext;
4839 public class MainApplication
4840 {
4841     public static void main(String [] args)
4842     {
4843         ApplicationContext ctx =
4844             new FileSystemXmlApplicationContext("springconfig.xml");
4845         IBusinessLogic testObject = (IBusinessLogic) ctx.getBean("proxyobj");
4846         int res=testObject.method1();
4847         System.out.println("test result :"+res);
4848     }
4849 }
4850 //Example: To do arguments validation before calling business logic method
4851 //IBusinessLogic.java
4852 public interface IBusinessLogic
4853 {
4854     public void method1(int id,String name);
4855 }
4856 //BusinessLogic.java
4857 public class BusinessLogic implements IBusinessLogic{
4858     public void method1(int id,String name)
4859     {
4860         System.out.println("");
4861         System.out.println("execution of method1");
4862         System.out.println(" id :"+id);
4863         System.out.println("name :"+name);
4864         System.out.println("");
4865     }
4866 }
```

```
4867 // AroundAdvice.java
4868 import org.aopalliance.intercept.MethodInvocation;
4869 import org.aopalliance.intercept.MethodInterceptor;

4870 public class AroundAdvice implements MethodInterceptor
4871 {
4872     public Object invoke(MethodInvocation i1) throws Throwable
4873     {
4874         System.out.println("good mng. everybody..");
4875         int x = ((Integer)i1.getArguments()[0]).intValue();
4876         String y=(String)i1.getArguments()[1];
4877         if( x <= 0 )
4878             i1.getArguments()[0] = new Integer(101);
4879         if( y.length() <= 3)
4880             i1.getArguments()[1]="sathya tech";
4881         i1.proceed();
4882         System.out.println("Goodbye! ");
4883         return null;
4884     }
4885 }

4886 <!-- springconfig.xml -->
4887 <?xml version="1.0" encoding="UTF-8"?>
4888 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
4889     "http://www.springframework.org/dtd/spring-beans.dtd">
4890 <beans>
4891     <bean id="beanTarget" class="BusinessLogic"/>
4892     <bean id="ad" class="AroundAdvice"/>
4893     <bean id="advisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
4894         <property name="advice">
4895             <ref local="ad"/>
4896         </property>
4897         <property name="pattern">
4898             <value>.*</value>
4899         </property>
4900     </bean>
```

```
4901 <bean id="proxyobj" class="org.springframework.aop.framework.ProxyFactoryBean">
4902   <property name="proxyInterfaces">
4903     <value>IBusinessLogic</value>
4904   </property>
4905   <property name="target">
4906     <ref local="beanTarget"/>
4907   </property>
4908   <property name="interceptorNames">
4909     <list>
4910       <value>advisor</value>
4911     </list>
4912   </property>
4913 </bean>
4914 </beans>
```

4915 //MainApplication.java

```
4916 import org.springframework.context.ApplicationContext;
4917 import org.springframework.context.support.FileSystemXmlApplicationContext;
4918 public class MainApplication
4919 {
4920   public static void main(String [] args)
4921   {
4922     ApplicationContext ctx =
4923       new FileSystemXmlApplicationContext("springconfig.xml");
4924     IBusinessLogic testObject = (IBusinessLogic) ctx.getBean("proxyobj");
4925     testObject.method1(22,"Sai Charan");
4926     testObject.method1(-22,"Kanakadhar");
4927     testObject.method1(-22,"SK");
4928   }
4929 }
```

4930 //Example: To work with before & anround advice

4931 //IBusinessLogic.java

```
4932 public interface IBusinessLogic
4933 {
4934   public void method1();
4935 }
```



```
4936 //BusinessLogic.java
4937 public class BusinessLogic implements IBusinessLogic
4938 {
4939     public void method1()
4940     {
4941         System.out.println("");
4942         System.out.println("execution of method1");
4943         System.out.println("");
4944     }
4945 }
4946 // BeforeAdvice1.java
4947 import org.springframework.aop.*;
4948 import java.lang.reflect.*;
4949 public class BeforeAdvice1 implements MethodBeforeAdvice
4950 {
4951     public void before(Method method, Object[] args, Object target)
4952     throws Throwable
4953     {
4954         System.out.println("before advice");
4955     }
4956 }
4957 // AroundAdvice2.java
4958 import org.aopalliance.intercept.MethodInvocation;
4959 import org.aopalliance.intercept.MethodInterceptor;
4960 public class AroundAdvice2 implements MethodInterceptor
4961 {
4962     public Object invoke(MethodInvocation i1) throws Throwable
4963     {
4964         System.out.println(" AroundAdvice : good mng everybody..");
4965         i1.proceed();
4966         System.out.println("AroundAdvice : Goodbye! ");
4967         return null;
4968     }
4969 }
```

```
4970 <!-- springconfig.xml -->
4971 <?xml version="1.0" encoding="UTF-8"?>
4972 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
4973     "http://www.springframework.org/dtd/spring-beans.dtd">
4974 <beans>
4975     <bean id="adv1" class="BeforeAdvice1"/>
4976     <bean id="adv2" class="AroundAdvice2"/>
4977     <bean id="targetobj" class="BusinessLogic"/>
4978     <bean id="advr1"
4979 class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
4980         <property name="advice">
4981             <ref local="adv1"/>
4982         </property>
4983         <property name="mappedName">
4984             <value>method1</value>
4985         </property>
4986     </bean>
4987 <bean id="advr2" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
4988     <property name="advice">
4989         <ref local="adv2"/>
4990     </property>
4991     <property name="pattern">
4992         <value>.*</value>
4993     </property>
4994 </bean>
4995 <bean id="proxyobj" class="org.springframework.aop.framework.ProxyFactoryBean">
4996     <property name="proxyInterfaces">
4997         <value>IBusinessLogic</value>
4998     </property>
4999     <property name="target">
5000         <ref local="targetobj"/>
5001     </property>
5002     <property name="interceptorNames">
5003         <list>
5004             <value>advr2</value>
```

```
5005         <value>advr1</value>
```

```
5006     </list>
```

```
5007 </property>
```

```
5008 </bean>
```

```
5009 </beans>
```

```
5010 //MainApplication.java
```

```
5011 import org.springframework.context.ApplicationContext;
```

```
5012 import org.springframework.context.support.FileSystemXmlApplicationContext;
```

```
5013 public class MainApplication
```

```
5014 {
```

```
5015     public static void main(String [] args)
```

```
5016     {
```

```
5017         ApplicationContext ctx =
```

```
5018             new FileSystemXmlApplicationContext("springconfig.xml");
```

```
5019         IBusinessLogic testObject = (IBusinessLogic) ctx.getBean("proxyobj");
```

```
5020         testObject.method1();
```

```
5021     }
```

```
5022 }
```

```
5023 Example: Working with all 4 advices [Before, After Around & Throws]
```

```
5024 Note: Here we will work with log4j [logging 4 java]
```

```
5025 Log4j: is a logging framework which is used to identify the control flow through log files. We can
5026 know where the problem is arised. Normally we can display messages on server console using
5027 s.o.p() statements. But those statements will be skipped soon & we cannot capture them. We can
5028 capture complete logs inforamtion through this log4j framework & we can store in the form of files.
5029 So we ever we need to we can open log files & can refer.
```

```
5030 Note: To work with this we need to set the classpath for log4j.jar
```

```
5031 // Adder.java
```

```
5032 package aop;
```

```
5033 public interface Adder {
```

```
5034     public int add(int a,int b);
```

```
5035 }
```

```
5036 //AdderImpl .java
```

```
5037 package aop;
```

```
5038 public class AdderImpl implements Adder {
```

```
5039     public int add(int a, int b){
```

```
5040         return a+b;
```

```
5041     }
5042 }
5043 //LogBeforeCallAdvice.java
5044 package aop;
5045 import java.lang.reflect.Method;
5046 import org.springframework.aop.MethodBeforeAdvice;
5047 import org.apache.log4j.*;
5048 public class LogBeforeCallAdvice implements MethodBeforeAdvice{
5049     Logger logger = Logger.getLogger(LogBeforeCallAdvice.class);
5050     public LogBeforeCallAdvice()
5051     {
5052         SimpleLayout layout = new SimpleLayout();
5053         FileAppender appender = null;
5054         try
5055         {
5056             appender = new FileAppender(layout,"output1.txt",true);
5057         }
5058         catch(Exception e) {}
5059         logger.addAppender(appender);
5060         logger.setLevel((Level) Level.DEBUG);
5061     }
5062     public void before(Method method, Object[] args, Object target)throws Throwable {
5063         logger.info("Before Calling the Method at: "+new java.util.Date());
5064     }
5065 }
5066 // LogAfterReturningAdvice.java
5067 package aop;
5068 import java.lang.reflect.Method;
5069 import org.springframework.aop.AfterReturningAdvice;
5070 import org.apache.log4j.*;
5071 public class LogAfterReturningAdvice implements AfterReturningAdvice{
5072     Logger logger = Logger.getLogger(LogAfterReturningAdvice.class);
5073     public LogAfterReturningAdvice()
5074     {
5075         SimpleLayout layout = new SimpleLayout();
```

```
5076         FileAppender appender = null;
5077         try
5078         {
5079             appender = new FileAppender(layout,"output1.txt",true);
5080         }
5081         catch(Exception e) {}
5082         logger.addAppender(appender);
5083         logger.setLevel((Level) Level.DEBUG);
5084     }
5085     public void afterReturning(Object returnValue, Method method, Object[] args, Object target)
5086     throws Throwable {
5087         logger.info("After Normal Return from Method at "
5088                     +new java.util.Date());
5089     }
5090 }
5091 // LogAroundAdvice.java
5092 package aop;
5093 import org.aopalliance.intercept.*;
5094 public class LogAroundAdvice implements MethodInterceptor{
5095     public Object invoke(MethodInvocation i1) throws Throwable {
5096         Object arguments[] = i1.getArguments();
5097         int number1 = ((Integer)arguments[0]).intValue();
5098         int number2 = ((Integer)arguments[1]).intValue();
5099         if (number1 == 0 && number2 == 0){
5100             throw new Exception("Dont know how to add 0 and 0!!!");
5101         }
5102         return i1.proceed();
5103     }
5104 }
5105 // LogAfterThrowsAdvice.java
5106 package aop;
5107 import java.lang.reflect.Method;
5108 import org.springframework.aop.ThrowsAdvice;
5109 public class LogAfterThrowsAdvice implements ThrowsAdvice{
```

```
5110     public void afterThrowing(Method method, Object[] args, Object target,
5111     Exception exception){
5112         System.out.println("Exception is thrown on method "+method.getName());
5113         System.out.println("Exception that is raised is " +exception.toString());
5114     }
5115 }
5116 <!-- aop-test.xml -->
5117 <?xml version="1.0" encoding="UTF-8"?>
5118 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
5119     "http://www.springframework.org/dtd/spring-beans.dtd">
5120 <beans>
5121     <!-- Advices -->
5122     <bean id = "beforeCall" class = "aop.LogBeforeCallAdvice" />
5123     <bean id = "afterCall" class = "aop.LogAfterReturningAdvice" />
5124     <bean id = "throwCall" class = "aop.LogAfterThrowsAdvice" />
5125     <bean id = "aroundCall" class = "aop.LogAroundAdvice" />
5126     <!-- Implementation Class -->
5127     <bean id = "adderImpl" class = "aop.AdderImpl" />
5128     <!-- Proxy Implementation Class -->
5129     <bean id="proxy" class=
5130         "org.springframework.aop.framework.ProxyFactoryBean">
5131         <property name = "proxyInterfaces">
5132             <value>aop.Adder</value>
5133         </property>
5134         <property name = "interceptorNames">
5135             <list>
5136                 <value>beforeCall</value>
5137                 <value>afterCall</value>
5138                 <value>throwCall</value>
5139                 <value>aroundCall</value>
5140             </list>
5141         </property>
5142         <property name = "target">
5143             <ref bean = "adderImpl"/>
5144         </property>
```

```
5145     </bean>
5146 </beans>
5147 //AopTestClient .java
5148 package aop;
5149 import org.springframework.beans.factory.BeanFactory;
5150 import org.springframework.beans.factory.xml.XmlBeanFactory;
5151 import org.springframework.core.io.*;
5152 public class AopTestClient {
5153     public static void main(String args[]){
5154         Resource resource = new FileSystemResource("aop-test.xml");
5155         BeanFactory factory = new XmlBeanFactory(resource);
5156         Adder adder = (Adder)factory.getBean("proxy");
5157         int result = adder.add(0, 0);
5158         // int result = adder.add(10, 20);
5159         System.out.println("Result = " + result);
5160     }
5161 }
5162 //Example: Integration spring – servlet & jdbc
5163 Note: jar files required in WEB-INF\lib folder & classpath are spring.jar, commons-logging.jar,
5164 classes12.jar
5165 <!-- index.html -->
5166 <frameset rows="30%,*" border = "0">
5167     <frame src="Search.jsp">
5168     <frame name="resultframe">
5169 </frameset>
5170 <!-- Search.jsp -->
5171 <body bgcolor="pink" >
5172     <form action="controller" target="resultframe">
5173         <b>Select job</b>
5174         <select name="job">
5175             <option>CLERK</option>
5176             <option>ANALYST</option>
5177             <option>SALESMAN</option>
5178             <option>MANAGER</option>
5179         </select>
```

```
5180     <input type="submit" value="search">
5181     </form>
5182 </body>
5183 <!-- Result.jsp -->
5184 <%@page import="java.util.*, p1.EmpBean"%>
5185 <table>
5186 <% ArrayList list=(ArrayList)request.getAttribute("result");
5187     if(list!=null)
5188     { %>
5189     <%System.out.println("No. of records are : "+list.size()); %>
5190     <tr>
5191         <th>ID</th>
5192         <th>Name</th>
5193         <th>Desg</th>
5194         <th>Salary</th>
5195     </tr>
5196     <%for(int i=0;i<list.size();++i)
5197     {
5198     EmpBean eb=(EmpBean)list.get(i); %>
5199     <tr>
5200         <td><%=eb.getId()%></td>
5201         <td><%=eb.getName()%></td>
5202         <td><%=eb.getDesg()%></td>
5203         <td><%=eb.getBsal()%></td>
5204     </tr>
5205     <% } %>
5206     </table>
5207     <% } %>
5208 <!-- web.xml -->
5209 <?xml version="1.0" encoding="UTF-8"?>
5210 <web-app>
5211     <servlet>
5212         <description>This is the description of my J2EE component</description>
5213         <display-name>This is the display name of my J2EE component</display-name>
5214         <servlet-name>MainServlet</servlet-name>
```



```
5215     <servlet-class>MainServlet</servlet-class>
5216 </servlet>
5217 <servlet-mapping>
5218     <servlet-name>MainServlet</servlet-name>
5219     <url-pattern>/controller</url-pattern>
5220 </servlet-mapping>
5221 </web-app>
5222 //EmpBean.java
5223 package p1;
5224 public class EmpBean {
5225     int id;
5226     String name,desg;
5227     float bsal;
5228     public EmpBean()
5229     {
5230         System.out.println("EmpBean Constructor");
5231     }
5232     public float getBsal() {
5233         return bsal;
5234     }
5235     public void setBsal(float bsal) {
5236         this.bsal = bsal;
5237     }
5238     public String getDesg() {
5239         return desg;
5240     }
5241     public void setDesg(String desg) {
5242         this.desg = desg;
5243     }
5244     public int getId() {
5245         return id;
5246     }
5247     public void setId(int id) {
5248         this.id = id;
5249     }
```

```
5250     public String getName() {
5251         return name;
5252     }
5253     public void setName(String name) {
5254         this.name = name;
5255     }
5256 }
5257 //Model.java
5258 import java.util.*;
5259 public interface Model {
5260     public ArrayList search(String desg);
5261 }
5262 // ModelBean.java
5263 import java.util.*;
5264 import javax.sql.*;
5265 import java.sql.*;
5266 import p1.EmpBean;
5267 public class ModelBean implements Model{
5268     public DataSource ds = null;
5269     public void setDs(DataSource ds) {
5270         this.ds = ds;
5271     }
5272     public ArrayList search(String desg)
5273     {
5274         System.out.println("search() of ModelBean class");
5275         ArrayList al=new ArrayList();
5276         try {
5277             Connection con=ds.getConnection();
5278             String qry = "select empno, ename, job, sal from emp where job=?";
5279             PreparedStatement ps=con.prepareStatement(qry);
5280             ps.setString(1, desg);
5281             ResultSet rs=ps.executeQuery();
5282             while(rs.next())
5283             {
5284                 EmpBean eb=new EmpBean();
```

```

5285         eb.setId(rs.getInt(1));
5286         eb.setName(rs.getString(2));
5287         eb.setDesg(rs.getString(3));
5288         eb.setBsal(rs.getFloat(4));
5289         al.add(eb);
5290     }
5291     } catch (SQLException e) {
5292         e.printStackTrace();
5293     }
5294     return al;
5295 }
5296 }
5297 <!-- modelconfig.xml -->
5298 <?xml version="1.0" encoding="UTF-8"?>
5299 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
5300     "http://www.springframework.org/dtd/spring-beans.dtd">
5301 <beans>
5302     <bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
5303         <property name="driverClassName">
5304             <value>oracle.jdbc.driver.OracleDriver</value>
5305         </property>
5306         <property name="url">
5307             <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
5308         </property>
5309         <property name="username"><value>scott</value></property>
5310         <property name="password"><value>tiger</value></property>
5311     </bean>
5312     <bean id="mdb" class="ModelBean" autowire="byName"/>
5313 </beans>
5314 // MainServlet.java
5315 import java.io.IOException;
5316 import java.util.ArrayList;
5317 import javax.servlet.*;
5318 import javax.servlet.http.*;
5319 import org.springframework.beans.factory.xml.*;

```

```
5320 import org.springframework.core.io.*;
5321 public class MainServlet extends HttpServlet {
5322     Model mod1=null;
5323     public void init()
5324     {
5325         System.out.println("Init Method");
5326         ClassPathResource res=new ClassPathResource("modelconfig.xml");
5327         XmlBeanFactory factory=new XmlBeanFactory(res);
5328         mod1=(Model)factory.getBean("mdb");
5329     }//init
5330     public void destroy() {
5331         System.out.println("destroy()");
5332         mod1=null;
5333     }
5334     public void doGet(HttpServletRequest request, HttpServletResponse response)
5335         throws ServletException, IOException {
5336         System.out.println("doGet() of MainServlet");
5337         try {
5338             String job=request.getParameter("job").trim();
5339             ArrayList al=mod1.search(job);
5340             request.setAttribute("result",al);
5341             RequestDispatcher rd=request.getRequestDispatcher("Result.jsp");
5342             if(rd!=null)
5343                 rd.forward(request,response);
5344         } catch (ServletException e) {
5345             e.printStackTrace();
5346         } catch (IOException e) {
5347             e.printStackTrace();
5348         }
5349     }
5350     public void doPost(HttpServletRequest request, HttpServletResponse response)
5351         throws ServletException, IOException {
5352         System.out.println("doPost() of MainServlet");
5353         try {
5354             doGet(request,response);
```

```
5355         } catch (ServletException e) {
5356             e.printStackTrace();
5357         } catch (IOException e) {
5358             e.printStackTrace();
5359         }
5360     } //doPost
5361 } //MainServlet
5362 //Example: Spring – Hibernate -Servlet
5363 <!-- web.xml -->
5364 <web-app>
5365     <servlet>
5366         <servlet-name>select</servlet-name>
5367         <servlet-class>selectservlet</servlet-class>
5368     </servlet>
5369     <servlet-mapping>
5370         <servlet-name>select</servlet-name>
5371         <url-pattern>/selectaction</url-pattern>
5372     </servlet-mapping>
5373 </web-app>
5374 //demointer.java
5375 import org.hibernate.*;
5376 public interface demointer
5377 {
5378     public SessionFactory getFactory();
5379 }
5380 // demoimpl.java
5381 import org.hibernate.*;
5382 public class demoimpl implements demointer
5383 {
5384     private SessionFactory sesfact;
5385     public void setSesfact(SessionFactory f1)
5386     {
5387         sesfact=f1;
5388     }
5389 }
```

```
5388     public SessionFactory getFactory()
5389     {
5390         return sesfact;
5391     }
5392 }
```

5393 **Note:** Create following table in oracle & insert some records So you can see stored records as a
5394 report

5395 create table users

```
5396 (
5397     userid number(5) primary key,
5398     uname varchar2(20),
5399     role varchar2(20)
5400 );
```

5401 **//User.java**

```
5402 public class User
5403 {
5404     private int uid;
5405     private String uname,role;
5406     public void setUid(int n)
5407     {
5408         uid=n;
5409     }
5410     public int getUid(){ return uid; }
5411     public void setUname(String s){ uname=s; }
5412     public String getUname(){ return uname; }
5413     public void setRole(String r){ role=r; }
5414     public String getRole(){ return role; }
5415 }
```

5416 **<!-- User.hbm.xml -->**

5417 **<?xml version="1.0"?>**

5418 **<!DOCTYPE hibernate-mapping PUBLIC**

5419 **"-//Hibernate/Hibernate Mapping DTD 3.0//EN"**

5420 **"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">**

5421 **<hibernate-mapping>**

```
5422     <class  name="User"  table="users" >
5423         <id  name="uid"  column="userid" />
5424         <property  name="uname" />
5425         <property  name="role" />
5426     </class>
5427 </hibernate-mapping>
5428 <!-- ApplicationContext.xml -->
5429 <?xml version="1.0" encoding="UTF-8"?>
5430 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
5431 "http://www.springframework.org/dtd/spring-beans.dtd">
5432 <beans>
5433     <bean id="myDataSource"
5434         class="org.springframework.jdbc.datasource.DriverManagerDataSource">
5435         <property name="driverClassName">
5436             <value>oracle.jdbc.driver.OracleDriver</value>
5437         </property>
5438         <property name="url">
5439             <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
5440         </property>
5441         <property name="username">
5442             <value>scott</value>
5443         </property>
5444         <property name="password">
5445             <value>tiger</value>
5446         </property>
5447     </bean>
5448     <bean id="mySessionFactory"
5449         class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
5450         <property name="dataSource" ref="myDataSource"/>
5451         <property name="mappingResources">
5452             <list>
5453                 <value>User.hbm.xml</value>
5454             </list>
5455         </property>
```

```
5456 <property name="hibernateProperties">
5457     <props>
5458         <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>
5459     </props>
5460 </property>
5461 </bean>
5462 <bean id="d1" class="demoimpl">
5463     <property name="sesfact">
5464         <ref local="mySessionFactory"/>
5465     </property>
5466 </bean>
5467 </beans>
5468 //selectservlet.java
5469 import javax.servlet.*;
5470 import javax.servlet.http.*;
5471 import java.io.*;
5472 import org.hibernate.*;
5473 import java.util.*;
5474 import org.springframework.beans.factory.*;
5475 import org.springframework.context.*;
5476 import org.springframework.context.support.*;
5477 public class selectservlet extends HttpServlet
5478 {
5479     public void service(HttpServletRequest request,HttpServletResponse response)
5480         throws ServletException,IOException
5481     {
5482         PrintWriter out=response.getWriter();
5483
5484         try{
5485             ApplicationContext ctx=new
5486             ClassPathXmlApplicationContext("ApplicationContext.xml");
5487             BeanFactory factory=(BeanFactory)ctx;
5488             demointer d1=(demointer)factory.getBean("d1");
5489             SessionFactory sf=d1.getFactory();
5490             Session ses=sf.openSession();
5491             Query query=ses.createQuery("from User");
```



```

5490     Iterator i1=query.iterate();
5491     out.println("<body bgcolor=#ffffcc text=red>");
5492     out.println("<h1><center>all users</h1><hr><br><h3>");
5493     out.println("<table width=80% border=2>");
5494     while(i1.hasNext())
5495     {
5496         User u1=(User)i1.next();
5497         out.println("<tr><td>"+u1.getId()+" <td>"+u1.getUsername()+" <td>"+u1.getRole()
5498 + "</tr>");
5499     }
5500     out.println("</table>");
5501     ses.close();
5502 } catch(HibernateException e)
5503 {
5504     out.println(e);
5505     e.printStackTrace();
5506 }
5507 }
5508 }

```

5509 **Note:** Set the classpath for:

- 5510 1. 8 hibernate jars [1 main har + 7 dependncy jars]
- 5511 2. 2 spring jars [1 main + 1 dependency jar]
- 5512 3. oracle database specific jar

5513 **Note:** After deplying & starting web or application server give url as follows in browser:

5514 <http://localhost:2010/SpringHibernateServlet/selectaction>

5515 **Spring MVC:**

5516 **Example: Spring MVC example with SimpleFormController class**

```

5517 <!-- index.jsp -->
5518 <jsp:forward page="/newpro.htm" />
5519 <!-- web.xml -->
5520 <web-app>
5521     <servlet>
5522         <servlet-name>productaction</servlet-name>
5523         <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

```

```
5524     <load-on-startup>1</load-on-startup>
5525 </servlet>
5526 <servlet-mapping>
5527     <servlet-name>productaction</servlet-name>
5528     <url-pattern>*.htm</url-pattern>
5529 </servlet-mapping>
5530 <taglib>
5531     <taglib-uri>spring-tld</taglib-uri>
5532     <taglib-location>/WEB-INF/spring.tld</taglib-location>
5533 </taglib>
5534 </web-app>
5535 Note: we need to name spring configuration file name as follows:
5536 <front controller class servlet name>-servlet.xml
5537 Eg: In above example our front controller class "DispatcherServlet" is named as "
5538 productaction" so we have named the spring configuration file as "productaction-servlet.xml".
5539 <!-- productaction-servlet.xml -->
5540 <?xml version="1.0" encoding="UTF-8"?>
5541 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
5542 "http://www.springframework.org/dtd/spring-beans.dtd">
5543 <beans>
5544     <bean id="simpleUrlMappings"
5545         class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping" >
5546         <property name="mappings">
5547             <props>
5548                 <prop key="/newpro.htm">newproController</prop>
5549             </props>
5550         </property>
5551     </bean>
5552     <bean id="newproController" class="NewProductController" >
5553         <property name="sessionForm" > <value>true</value></property>
5554         <property name="commandName" > <value>product</value></property>
5555         <property name="commandClass" > <value>Product</value></property>
5556         <property name="formView" > <value>newproduct</value></property>
5557         <property name="successView" > <value>showproduct</value></property>
5558     </bean>
```

```
5559     <bean id="viewResolver" class =
5560         "org.springframework.web.servlet.view.InternalResourceViewResolver">
5561         <property name="viewClass">
5562             <value>org.springframework.web.servlet.view.JstlView</value>
5563         </property>
5564         <property name="prefix"><value>/</value></property>
5565         <property name="suffix"><value>.jsp</value></property>
5566     </bean>
5567 </beans>
5568 <!-- newproduct.jsp -->
5569 <%@ page language="java" %>
5570 <%@ taglib prefix="spring" uri="spring-tld" %>
5571 <html>
5572 <body bgcolor=#ffffcc text=green>
5573 <h1><center>new product information</h1><hr><br><h3>
5574 <form method="post" >
5575     enter product id :
5576     <spring.bind path="product.pid" >
5577     <input type="text" name="pid" />
5578 </spring.bind>
5579     <br><br>
5580     enter product name :
5581     <spring.bind path="product.pname" >
5582     <input type="text" name="pname" />
5583 </spring.bind>
5584     <br><br>
5585     enter product price :
5586     <spring.bind path="product.price" >
5587     <input type="text" name="price" />
5588 </spring.bind>
5589     <br><br>
5590     <input type="submit" value="send" />
5591 </form>
5592 </body>
5593 </html>
```

5594 //Product.java

5595 public class Product

5596 {

5597 private int pid;

5598 private String pname;

5599 private double price;

5600 public void setPid(int n){ pid=n; }

5601 public int getPid(){ return pid; }

5602 public void setPname(String s){ pname=s; }

5603 public String getPname(){ return pname; }

5604 public void setPrice(double p){ price=p; }

5605 public double getPrice(){ return price; }

5606 }

5607 //NewProductController.java

5608 import org.springframework.web.servlet.mvc.*;

5609 import org.springframework.web.servlet.*;

5610 import org.springframework.validation.*;

5611 import javax.servlet.http.*;

5612 public class NewProductController extends SimpleFormController

5613 {

5614 public ModelAndView onSubmit(HttpServletRequest request, HttpServletResponse
5615 response, Object command, BindException e) throws Exception

5616 {

5617 Product p=(Product)command;

5618 String str="Product id :"+p.getPid()+"
 "

5619 +"product name :"+p.getPname()+"
 "

5620 +"Price :"+p.getPrice();

5621 System.out.println(str);

5622 request.setAttribute("productinfo",str);

5623 return new ModelAndView(getSuccessView());

5624 // return new ModelAndView("showproduct");

5625 }

5626 }

```
5627 <!-- showproduct.jsp -->
5628 <%@ page language="java" %>
5629 <html>
5630 <body bgcolor=#ffffcc text=green>
5631 <h1><center>new product information</h1><hr><br><h3>
5632 <%= request.getAttribute("productinfo") %>
5633 </body>
5634 </html>
5635 Example: Spring MVC example with "Controller" interface
5636 <!-- index.jsp -->
5637 <jsp:forward page="/display.form" />
5638 <!-- web.xml -->
5639 <web-app>
5640   <servlet>
5641     <servlet-name>accaction</servlet-name>
5642     <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
5643     <load-on-startup>1</load-on-startup>
5644   </servlet>
5645   <servlet-mapping>
5646     <servlet-name>accaction</servlet-name>
5647     <url-pattern>*.form</url-pattern>
5648   </servlet-mapping>
5649   <taglib>
5650     <taglib-uri>spring-tld</taglib-uri>
5651     <taglib-location>/WEB-INF/spring.tld</taglib-location>
5652   </taglib>
5653 </web-app>
5654 <!-- accaction-servlet.xml -->
5655 <?xml version="1.0" encoding="UTF-8"?>
5656 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
5657 "http://www.springframework.org/dtd/spring-beans.dtd">
5658 <beans>
5659   <bean id="simpleUrlMappings"
5660     class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping" >
5661     <property name="mappings">
```

```
5662         <props>
5663             <prop key="/display.form">displayaccController</prop>
5664             <prop key="/newacc.form">newaccController</prop>
5665         </props>
5666     </property>
5667 </bean>
5668 <bean id="displayaccController" class="DisplayAccountsController" >
5669     <property name="dsource">
5670         <ref bean="ds" />
5671     </property>
5672 </bean>
5673 <bean id="newaccController" class="NewAccountController" >
5674     <property name="sessionForm" > <value>true</value></property>
5675     <property name="commandName" > <value>account</value></property>
5676     <property name="commandClass" > <value>Account</value></property>
5677     <property name="formView" > <value>newaccount</value></property>
5678     <property name="successView" > <value>index</value></property>
5679     <property name="dsource">
5680         <ref bean="ds" />
5681     </property>
5682 </bean>
5683 <bean id="viewResolver" class =
5684     "org.springframework.web.servlet.view.InternalResourceViewResolver">
5685     <property name="viewClass">
5686         <value>org.springframework.web.servlet.view.JstlView</value>
5687     </property>
5688     <property name="prefix"><value>/</value></property>
5689     <property name="suffix"><value>.jsp</value></property>
5690 </bean>
5691 <bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource" >
5692     <property name="driverClassName">
5693         <value>oracle.jdbc.driver.OracleDriver</value></property>
5694     <property name="url">
5695         <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
5696     </property>
```

```
5697     <property name="username" ><value>scott</value></property>
5698     <property name="password" ><value>tiger</value></property>
5699 </bean>
5700 </beans>
5701 //Account.java
5702 public class Account
5703 {
5704     private int acno;
5705     private String acname;
5706     private double balance;
5707     public Account(){}
5708     public Account(int a,String b,double c)
5709     {
5710         acno=a;
5711         acname=b;
5712         balance=c;
5713     }
5714     public void setAcno(int n){ acno=n; }
5715     public int getAcno(){ return acno; }
5716     public void setAcname(String s){ acname=s; }
5717     public String getAcname(){ return acname; }
5718     public void setBalance(double p){ balance=p; }
5719     public double getBalance(){ return balance; }
5720 }
5721 // DisplayAccountsController.java
5722 import org.springframework.web.servlet.mvc.*;
5723 import org.springframework.web.servlet.*;
5724 import org.springframework.validation.*;
5725 import javax.servlet.http.*;
5726 import javax.sql.*;
5727 import java.util.*;
5728 import org.springframework.jdbc.core.*;
5729 import org.springframework.jdbc.support.rowset.*;
```

```
5730 public class DisplayAccountsController implements Controller
5731 {
5732     private DataSource dsource;
5733     public void setDsource(DataSource ds)
5734     {
5735         dsource=ds;
5736     }
5737     public ModelAndView handleRequest(HttpServletRequest request,HttpServletResponse
5738 response)
5739     {
5740         List accs=new ArrayList();
5741         try{
5742             JdbcTemplate jt=new JdbcTemplate(dsource);
5743             SqlRowSet rs=jt.queryForRowSet("select * from bank");
5744             while(rs.next())
5745             {
5746                 accs.add(new Account(rs.getInt(1),rs.getString(2),rs.getDouble(3)));
5747             }
5748         }catch(Exception e){ System.out.println(e); }
5749         return new ModelAndView("displayaccounts","listaccounts",accs);
5750     }
5751 }
5752 <!-- displayaccounts.jsp -->
5753 <%@ page language="java" %>
5754 <%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
5755 <html>
5756 <body bgcolor=#ffffcc text=green>
5757 <h1><center>all accounts information</h1><hr><br><h3>
5758 <table width=80% border=2>
5759 <tr><th>a/c number<th>a/c holder name<th> balance</tr>
5760 <c:forEach items="${listaccounts}" var="acc" >
5761     <tr>
5762         <td><c:out value="${acc.acno}" />
5763         <td><c:out value="${acc.acname}" />
5764         <td><c:out value="${acc.balance}" />
```



```
5765     </tr>
5766 </c:forEach>
5767 </table>
5768 <center><br><br>
5769 <a href="/AccountsMvc/newacc.form">new account</a>
5770 </body>
5771 </html>
5772 <!-- newaccount.jsp -->
5773 <%@ page language="java" %>
5774 <%@ taglib prefix="spring" uri="spring-tld" %>
5775 <html>
5776 <body bgcolor=#ffffcc text=green>
5777 <h1><center>new account information</h1><hr><br><h3>
5778 <form method="post" >
5779   enter a/c number :
5780   <spring.bind path="account.acno" >
5781   <input type="text" name="acno" />
5782 </spring.bind>
5783 <br><br>
5784   enter a/c holder name :
5785   <spring.bind path="account.acname" >
5786   <input type="text" name="acname" />
5787 </spring.bind>
5788 <br><br>
5789   enter open balance :
5790   <spring.bind path="account.balance" >
5791   <input type="text" name="balance" />
5792 </spring.bind>
5793 <br><br>
5794   <input type="submit" value="save" />
5795 </form>
5796 </body>
5797 </html>
```

5798 // **NewAccountController.java**

5799 import org.springframework.web.servlet.mvc.*;

5800 import org.springframework.web.servlet.*;

5801 import org.springframework.validation.*;

5802 import javax.servlet.http.*;

5803 import javax.sql.*;

5804 import org.springframework.jdbc.core.*;

5805 public class NewAccountController extends SimpleFormController

5806 {

5807 private DataSource dsource;

5808 public void setDsource(DataSource ds)

5809 {

5810 dsource=ds;

5811 }

5812 public ModelAndView onSubmit(HttpServletRequest request, HttpServletResponse
5813 response, Object command, BindException e1) throws Exception

5814 {

5815 Account a1=(Account)command;

5816 try{

5817 JdbcTemplate jt=new JdbcTemplate(dsource);

5818 jt.update("insert into bank values(?,?,?)",new Object[]{new

5819 Integer(a1.getAcno()),a1.getAcname(),new Double(a1.getBalance())});

5820 }catch(Exception e){ System.out.println(e); }

5821 return new ModelAndView(getSuccessView());

5822 }

5823 }

5824 **Example: Pure Struts Application**

5825 <!-- web.xml -->

5826 <web-app>

5827 <!-- Action Servlet Configuration -->

5828 <servlet>

5829 <servlet-name>action</servlet-name>

5830 <servlet-class>org.apache.struts.action.**ActionServlet**</servlet-class>

5831 <init-param>

5832 <param-name>config</param-name>

```
5833     <param-value>/WEB-INF/struts-config.xml</param-value>
5834     </init-param>
5835     <load-on-startup>1</load-on-startup>
5836 </servlet>
5837 <!-- Action Servlet Mapping -->
5838 <servlet-mapping>
5839     <servlet-name>action</servlet-name>
5840     <url-pattern>*.do</url-pattern>
5841 </servlet-mapping>
5842 <!-- The Usual Welcome File List -->
5843 <welcome-file-list>
5844     <welcome-file>register.jsp</welcome-file>
5845 </welcome-file-list>
5846 <!-- Struts Tag Library Descriptors -->
5847 <taglib>
5848     <taglib-uri>/hi-friends-geve</taglib-uri>
5849     <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
5850 </taglib>
5851 </web-app>
5852 <!-- register.jsp -->
5853 <%@ taglib uri="/hi-friends-gmg" prefix="html" %>
5854 <html:html>
5855     <html:form action = "/register">
5856         <table border = 0 align = center>
5857             <tr>
5858                 <th>UserName
5859                 <td><html:text property = "username"/>
5860             </tr>
5861             <tr>
5862                 <th>PassWord
5863                 <td><html:password property = "password"/>
5864             </tr>
5865             <tr>
5866                 <td colspan = 2 align = center>
5867                 <html:submit value = "Register"/>
```

```
5868         </tr>
5869     </table>
5870 </html:form>
5871 </html:html>
5872 <!-- struts-config.xml -->
5873 <?xml version="1.0" encoding="UTF-8"?>
5874 <!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
5875 1.1//EN" "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
5876 <struts-config>
5877     <form-beans>
5878         <form-bean name="registerForm" type="app.RegisterForm"/>
5879     </form-beans>
5880     <action-mappings>
5881         <action name="registerForm" path="/register" type="app.RegisterAction">
5882             <forward name="ok" path="/success.jsp"/>
5883             <forward name="fail" path="/failure.jsp"/>
5884         </action>
5885     </action-mappings>
5886 </struts-config>
5887 //RegisterForm.java
5888 package app;
5889 import org.apache.struts.action.*;
5890 public class RegisterForm extends ActionForm
5891 {
5892     private String username = null;
5893     private String password = null;
5894     public void setUsername(String username)
5895     {
5896         this.username = username;
5897     }
5898     public String getUsername()
5899     {
5900         return username;
5901     }
}
```

```
5902     public void setPassword(String password)
5903     {
5904         this.password = password;
5905     }
5906     public String getPassword()
5907     {
5908         return password;
5909     }
5910 }
5911 // RegisterAction.java
5912 package app;
5913 import org.apache.struts.action.*;
5914 import javax.servlet.http.*;
5915 public class RegisterAction extends Action
5916 {
5917     public ActionForward execute(ActionMapping mapping,ActionForm
5918 form,HttpServletRequest request,HttpServletResponse response)throws Exception
5919     {
5920         RegisterForm rf = (RegisterForm)form;
5921         String user = rf.getUsername();
5922         String pass = rf.getPassword();
5923         if(user.equals("sathya") && pass.equals("java"))
5924             return mapping.findForward("ok");
5925         else
5926             return mapping.findForward("fail");
5927     }
5928 }
5929 <!-- success.jsp -->
5930 <html>
5931     <body>
5932         <center>
5933             <font size = 5 color = green>Login Successfull</font>
5934         </center>
5935     </body>
5936 </html>
```

```

5937 <!-- failure.jsp -->
5938 <html>
5939     <body>
5940         <center>
5941             <font size = 5 color = red>Login Failure</font><br>
5942             <a href = "register.jsp">Try Again</a>
5943         </center>
5944     </body>
5945 </html>
5946 Note: Place struts related jar files inside "lib" folder of WEB-INF.

5947 //Example: Integeration of struts & Spring
5948 <!-- web.xml -->
5949 <web-app>
5950     <servlet>
5951         <servlet-name>action</servlet-name>
5952         <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
5953         <init-param>
5954             <param-name>config</param-name>
5955             <param-value>/WEB-INF/struts-config.xml</param-value>
5956         </init-param>
5957         <load-on-startup>1</load-on-startup>
5958     </servlet>
5959     <servlet-mapping>
5960         <servlet-name>action</servlet-name>
5961         <url-pattern> *.do </url-pattern>
5962     </servlet-mapping>
5963     <welcome-file-list>
5964         <welcome-file>index.jsp</welcome-file>
5965     </welcome-file-list>
5966     <taglib>
5967         <taglib-uri>struts-html</taglib-uri>
5968         <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
5969     </taglib>
5970 </web-app>

```

```
5971 <!-- index.jsp -->
5972 <%@ page language="java" %>
5973 <%@ taglib prefix="html" uri="struts-html" %>
5974 <html>
5975 <body bgcolor=#ffffcc text=green>
5976 <h1><center>struts framewrok</h1><hr><br><h3>
5977 <html:form action="/demo">
5978     enter user name :<html:text property="uname" /> <br><br>
5979     <html:submit value="wish" />
5980 </html:form>
5981 </body>
5982 </html>
5983 <!-- struts-config.xml -->
5984 <?xml version="1.0" encoding="UTF-8"?>
5985 <!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
5986 1.1//EN"
5987     "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
5988 <struts-config>
5989     <form-beans>
5990         <form-bean name="demofrm" type="demopack.DemoForm" />
5991     </form-beans>
5992     <action-mappings>
5993         <action path="/demo" name="demofrm"
5994             type="org.springframework.web.struts.DelegatingActionProxy" >
5995             <forward name="success" path="/result.jsp" />
5996         </action>
5997     </action-mappings>
5998     <plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
5999     </plug-in>
6000 </struts-config>
6001 <!--action-servlet.xml--> [Spring Configuration File]
6002 <?xml version="1.0" encoding="UTF-8"?>
6003 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
6004     "http://www.springframework.org/dtd/spring-beans.dtd">
6005 <beans>
```

```
6006 <bean name="/demo" class="demopack.DemoAction" >
6007     <property name="model">
6008         <ref bean="d1" />
6009     </property>
6010 </bean>
6011 <bean id="d1" class="demopack.ModelImpl" >
6012 </bean>
6013 </beans>
6014 // DemoForm.java
6015 package demopack;
6016 import org.apache.struts.action.*;
6017 public class DemoForm extends ActionForm
6018 {
6019     private String uname;
6020     public void setUname(String s)
6021     {
6022         uname=s;
6023     }
6024     public String getUname()
6025     {
6026         return uname;
6027     }
6028 }
6029 // ModelInter.java
6030 package demopack;
6031 public interface ModelInter
6032 {
6033     public String getWish();
6034 }
6035 // ModelImpl.java
6036 package demopack;
6037 import java.util.*;
6038 public class ModelImpl implements ModelInter
6039 {
6040     public String getWish()
```



```
6041     {
6042         Calendar c1=Calendar.getInstance();
6043         int h=c1.get(Calendar.HOUR_OF_DAY);
6044         if(h < 12)
6045             return "Good Morning";
6046         else if( h < 17)
6047             return "Good Afternoon";
6048         else
6049             return "Good Evening";
6050     }
6051 }
6052 // DemoAction.java
6053 package demopack;
6054 import javax.servlet.*;
6055 import javax.servlet.http.*;
6056 import java.io.*;
6057 import org.apache.struts.action.*;
6058 public class DemoAction extends Action
6059 {
6060     private ModelInter model;
6061     public void setModel(ModelInter i1)
6062     {
6063         model=i1;
6064     }
6065     public ActionForward execute(ActionMapping mapping,ActionForm form,HttpServletRequest
6066 request,HttpServletResponse response)
6067     throws ServletException,IOException
6068     {
6069         DemoForm f1=(DemoForm)form;
6070         String uname=f1.getUname();
6071         String msg=model.getWish()+" "+ uname;
6072         request.setAttribute("message",msg);
6073
6074         return mapping.findForward("success");
6075     }
}
```

```
6075 }
6076 <!-- result.jsp -->
6077 <%@ page language="java" %>
6078 <%@ taglib prefix="html" uri="struts-html" %>
6079 <html>
6080 <body bgcolor=#ffffcc text=green>
6081 <h1><center>struts framewrok</h1><hr><br><h3>
6082 <%= request.getAttribute("message") %>
6083 <br><br>
6084 <html:link href="index.jsp" >home</html:link>
6085 </body>
6086 </html>
6087 Note: In "lib" folder of WEB_INF we need to place related jar files of spring & struts.
6088 //Example: Integration of struts, spring & hibernate
6089 <!-- web.xml -->
6090 <!DOCTYPE web-app
6091 PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
6092 "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
6093 <web-app>
6094 <servlet>
6095     <servlet-name>action</servlet-name>
6096     <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
6097     <init-param>
6098         <param-name>config</param-name>
6099         <param-value>/WEB-INF/struts-config.xml</param-value>
6100     </init-param>
6101     <load-on-startup>1</load-on-startup>
6102 </servlet>
6103 <servlet-mapping>
6104     <servlet-name>action</servlet-name>
6105     <url-pattern> *.do </url-pattern>
6106 </servlet-mapping>
6107 <welcome-file-list>
6108     <welcome-file>index.jsp</welcome-file>
6109 </welcome-file-list>
```

```

6110     <taglib>
6111         <taglib-uri>struts-html</taglib-uri>
6112         <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
6113     </taglib>
6114 </web-app>
6115 <!-- index.jsp -->
6116 <%@ page language="java" %>
6117 <%@ taglib prefix="html" uri="struts-html" %>
6118 <html>
6119 <body bgcolor=#ffffcc text=green>
6120 <h1><center>new user information</h1><hr><br><h3>
6121 <html:form action="/user">
6122     enter user name :<html:text property="uname" /> <br><br>
6123     select role:
6124 <table>
6125 <tr><td>
6126     <html:radio property="role" value="Administrator">Administrator</html:radio>
6127     <td><html:radio property="role" value="User">User</html:radio>
6128 </tr>
6129 <tr>
6130     <td><html:radio property="role" value="Salesman">Salesman</html:radio>
6131     <td> <html:radio property="role" value="Manager">Manager</html:radio>
6132 </tr>
6133 </table>
6134     <html:submit value="save" />
6135 </html:form>
6136 </body>
6137 </html>
6138 <!-- struts-config.xml -->
6139 <?xml version="1.0" encoding="ISO-8859-1" ?>
6140 <!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
6141 1.1//EN"
6142     "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
6143 <struts-config>

```

```
6144 <form-beans>
6145     <form-bean name="userform" type="demo.UserForm" />
6146 </form-beans>
6147 <action-mappings>
6148     <action path="/user" name="userform"
6149     type="org.springframework.web.struts.DelegatingActionProxy" >
6150         <forward name="success" path="/result.jsp" />
6151         <forward name="failure" path="/index.jsp" />
6152     </action>
6153 </action-mappings>
6154 <plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
6155 </plug-in>
6156 </struts-config>
6157 <!-- action-servlet.xml--> [Spring Configuration file]
6158 <?xml version="1.0" encoding="UTF-8"?>
6159 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
6160 "http://www.springframework.org/dtd/spring-beans.dtd">
6161 <beans>
6162     <bean name="/user" class="demo.UserAction" >
6163         <property name="model">
6164             <ref bean="d1" />
6165         </property>
6166     </bean>
6167     <bean id="d1" class="demo.ModelImpl" >
6168         <property name="sesfact">
6169             <ref bean="mySessionFactory" />
6170         </property>
6171     </bean>
6172     <bean id="myDataSource"
6173         class="org.springframework.jdbc.datasource.DriverManagerDataSource">
6174         <property name="driverClassName">
6175             <value>oracle.jdbc.driver.OracleDriver</value>
6176         </property>
6177         <property name="url">
6178             <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
```

```
6179     </property>
6180     <property name="username">
6181         <value>scott</value>
6182     </property>
6183     <property name="password">
6184         <value>tiger</value>
6185     </property>
6186 </bean>
6187 <bean id="mySessionFactory"
6188 class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
6189     <property name="dataSource" ref="myDataSource"/>
6190     <property name="mappingResources">
6191         <list>
6192             <value>User.hbm.xml</value>
6193         </list>
6194     </property>
6195     <property name="hibernateProperties">
6196         <props>
6197             <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>
6198         </props>
6199     </property>
6200 </bean>
6201 </beans>
6202 //ModelInter.java
6203 package demo;
6204 public interface ModelInter
6205 {
6206     public boolean insertInfo(String uname,String role);
6207 }
6208 // ModelImpl.java
6209 package demo;
6210 import java.util.*;
6211 import org.hibernate.*;
6212 public class ModelImpl implements ModelInter
6213 {
```

```
6214     private SessionFactory sesfact;
6215     public void setSesfact(SessionFactory sf)
6216     {
6217         sesfact=sf;
6218     }
6219     public boolean insertInfo(String uname,String role)
6220     {         Session ses=sesfact.openSession();
6221         Transaction tx=ses.beginTransaction();
6222         boolean flag=false;

6223         try{

6224             User u1=new User();
6225             u1.setUname(uname);
6226             u1.setRole(role);
6227             ses.save(u1);
6228             tx.commit();
6229             flag=true;
6230         }catch(Exception e)
6231         {         tx.rollback();
6232             System.out.println(e);
6233         }
6234         ses.close();
6235         return flag;
6236     }
6237 }
6238 //User.java
6239 package demo;
6240 public class User{
6241     private int uid;
6242     private String uname,role;
6243     public void setUid(int n)
6244     {
6245         uid=n;
6246     }
```

```
6247     public int getUid(){ return uid; }
6248     public void setUsername(String s){ username=s; }
6249     public String getUsername(){ return username; }
6250     public void setRole(String r){ role=r; }
6251     public String getRole(){ return role; }
6252 }
6253 <!-- User.hbm.xml -->
6254 <?xml version="1.0"?>
6255 <!DOCTYPE hibernate-mapping PUBLIC
6256 "Hibernate/Hibernate Mapping DTD 3.0//EN"
6257 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
6258 <hibernate-mapping>
6259     <class name="demo.User" table="users" >
6260         <id name="uid" column="userid" >
6261             <generator class="increment" />
6262         </id>
6263         <property name="username" />
6264         <property name="role" />
6265     </class>
6266 </hibernate-mapping>
6267 // UserForm.java
6268 package demo;
6269 import org.apache.struts.action.*;
6270 public class UserForm extends ActionForm
6271 {
6272     private String username,role;
6273     public void setUsername(String s)
6274     {
6275         username=s;
6276     }
6277     public String getUsername()
6278     {
6279         return username;
6280     }
6281     public void setRole(String s){ role=s; }
```

```
6282     public String getRole(){ return role; }
6283
6284 }
6285 // UserAction.java
6286 package demo;
6287 import javax.servlet.*;
6288 import javax.servlet.http.*;
6289 import java.io.*;
6290 import org.apache.struts.action.*;
6291 public class UserAction extends Action
6292 { private ModelInter model;
6293     public void setModel(ModelInter i1)
6294     {
6295         model=i1;
6296     }
6297     public ActionForward execute(ActionMapping mapping,ActionForm form,HttpServletRequest
6298 request,HttpServletResponse response)
6299     throws ServletException,IOException
6300     {
6301         UserForm f1=(UserForm)form;
6302         String uname=f1.getUsername();
6303         String role=f1.getRole();
6304         if(model.insertInfo(uname,role))
6305             return mapping.findForward("success");
6306         else
6307             return mapping.findForward("failure");
6308     }
6309 }
6310 <!-- result.jsp -->
6311 <%@ page language="java" %>
6312 <%@ taglib prefix="html" uri="struts-html" %>
6313 <html> <body bgcolor=#ffffcc text=green>
6314 <h1><center>welcome user</h1><hr>
6315 </body>
6316 </html>
```