Spring Core Module

How to inject a list based Collection into a spring bean?

step-1:-

In the spring bean class declare "List" type variable as dependecy.

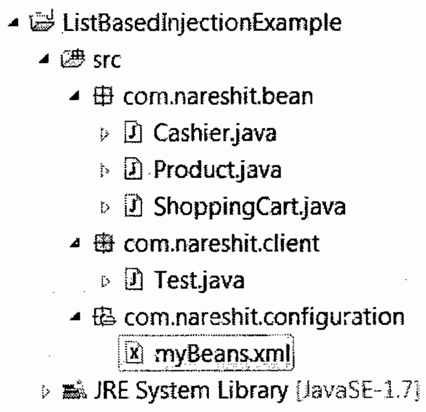
Example :

```
class College
{
List<Student> students;
//required setters and getters
}
```

step-2:-

Make use of Collection configuration elements in bean configuration file.

i,e we can use <list> tag.



```
ListBasedInjectionExample
  src
    com.nareshit.bean
      Cashier.java
      Product.java
      ShoppingCart.java
    com.nareshit.client
      Test.java
    com.nareshit.configuration
      myBeans.xml
  JRE System Library [JavaSE-1.7]
```

Product.java

```
package com.nareshit.bean;
public class Product {
	private int productId;
	private String productName;
	private double price;

	public int getProductId() {
		return productId;
	}

	pubiic void setProductId(int productId) {
		this.productId = productId;
	}

	public String getProductName() {
		return productName;
	}

	public void setProductName(String productName) {
		this.productName = productName;
	}

	public double getPrice() {
		return price;
	}
```

```
        public void setPrice(double price) {
                this.price = price;
        }
}
```
ShoppingCart.java
```
package com.nareshit.bean;
import java.util.List;
public class ShoppingCart {
        private List<Product> products;

        public List<Product> getProducts() {
                return products;
        }

        public void setProducts(List<Product> products) {
                this.products = products;
        }

}
```

Cashier.java

```
package com.nareshit.bean;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Date;
import java.util.List;
public class Cashier {
private double totalPrice;
private String fileName;
private String filePath;
private PrintWriter printWriter;
public Cashier(String filePath,String fileName){
this.filePath=filePath;
this.fileName=fileName;
openStream();
}
public void openStream(){
        try{
printWriter=new PrintWriter(filePath+fileName+".txt");
        }catch(FileNotFoundException fe){
System.out.println("Exception Occured while creating printWriter obj ::"+fe.getMessage());
        }
}
public void calculateTotalPrice(ShoppingCart cart){
List<Product> products=cart.getProducts();
System.out.println(products.getClass());
for(Product product:products){
        writeProductToFile(product);
totalPrice=totalPrice+product.getPrice();
}
printWriter.println("------------------");
printWriter.println("TotalPrice :"+totalPrice);
printWriter.flush();
}
public void writeProductToFile(Product product){
```

```
printWriter.println("productName :"+product.getProductName()+"\tprice:"+product.getPrice()+"\t
Date :"+new Date());
}
}
```

myBeans.xml

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                              "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
<bean id="product1" class="com.nareshit.bean.Product">
<property name="productId" value="4001"/>
<property name="productName" value="keyboard"/>
<property name="price" value="400"/>
</bean>
<bean id="product2" class="com.nareshit.bean.Product">
<property name="productId" value="4002"/>
<property name="productName" value="mouse"/>
<property name="price" value="200"/>
</bean>
<bean id="product3" class="com.nareshit.bean.Product">
<property name="productId" value="4003"/>
<property name="productName" value="LCD"/>
<property name="price" value="2000"/>
</bean>
<bean id="shoppingCart" class="com.nareshit.bean.ShoppingCart">
<property name="products">
<list>
<ref bean="product1"/>
<ref bean="product2"/>
<ref bean="product3"/>
</list>
</property>
</bean>
<bean id="cashier" class="com.nareshit.bean.Cashier">
<constructor-arg index="0" value="G://spring/"/>
<constructor-arg index="1"  value="productsInformation"/>
</bean>
</beans>
```

Test.java

```
package com.nareshit.client;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;

import com.nareshit.bean.Cashier;

import com.nareshit.bean.ShoppingCart;

public class Test {

public static void main(String[] args) {

ClassPathResource resource=

new ClassPathResource("com/nareshit/configuration/myBeans.xml");
XmlBeanFactory factory= new XmlBeanFactory(resource);

    ShoppingCart shoppingCart =(ShoppingCart)factory.getBean("shoppingCart");

Cashier cashier=(Cashier)factory.getBean("cashier");

cashier.calculateTotalPrice(shoppingCart);

}

}
```

How to configure setBased collection as dependency to a spring bean ?

step-1:-
the spring bean class declare "Set" type variable as dependecy.

Example :

```
class College{
Set<Student> students;
//required setters and getters

}
```

step-2:-
Make use of Collection configuration elements in bean configuration file.
i,e we can use <set> tag.

How to inject a Map based collection into a SpringBean?

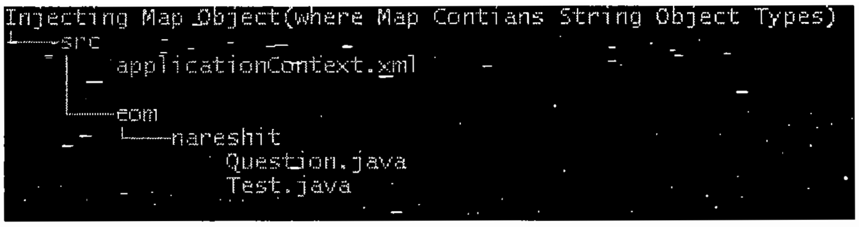
Step1 :-In the Spring bean class declare java.util.Map type variable as dependecy.

Step 2:- make the use of<map>tag in Spring configuration file

Example:-

```
<bean id="question" class="com.nareshit.Question">
<constructor-arg value="11"></constructor-arg>
<constructor-arg value="What is Java?"></constructor-arg>
<constructor-arg>
<map>
<entry key="Java is a Programming Language"  value="Ajay"></entry>
<entry key="Java is a Platform" value="Vijay"></entry>
<entry key="Java is an Island" value="Sanjay"></entry>
</map>
</constructor-arg>
</bean>
```

Injecting Map Object(Where Map Contains String Object Types)



```
Injecting Map Object(where Map Contians String Object Types)
└───src
    │  applicationContext.xml
    │
    └───com
        └───nareshit
                Question.java
                Test.java
```

Question.java

```
package com.nareshit;
import java.io.Serializable;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.Map.Entry;
public class Question implements  Serializable {
private int id;
private String name;
private Map<String,String> answers;
public Question() {}
public Question(int id, String name,Map<String, String> answers) {
	this.id = id;
	this.name = name;
	this.answers = answers;
}
public void displayInfo(){
	System.out.println("question id:"+id);
	System.out.println("question name:"+name);
	System.out.println("Answers....");
Set<Entry<String, String>> set=answers.entrySet();
Iterator<Entry<String, String>> itr=set.iterator();
	while(itr.hasNext()){
Entry<String,String> entry=itr.next();
System.out.println("Answer:"+entry.getKey()+" Posted By:"+entry.getValue());
	}
}
}
```

applicationConext.xml

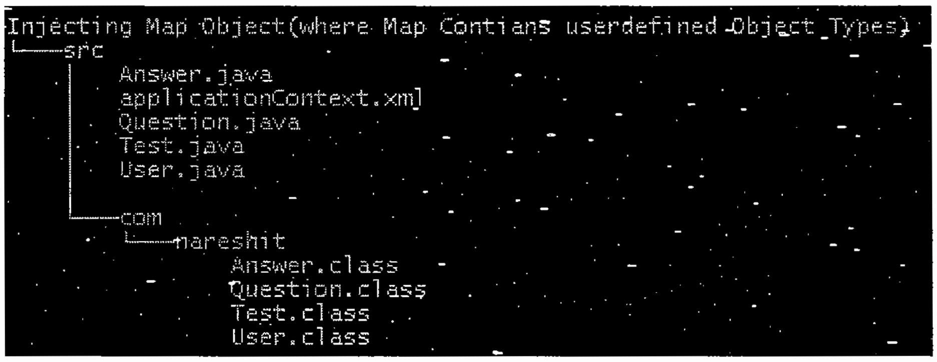
```
<?xml version="1.0" encoding="UTF-8"?>
<beans
	xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
<bean id="question" class="com.nareshit.Question">
<constructor-arg value="11"></constructor-arg>
<constructor-arg value="What is Java?"></constructor-arg>
<constructor-arg>
<map>
<entry key="Java is a Programming Language"  value="Ajay"></entry>
```

```
<entry key="Java is a Platform" value="Vijay"></entry>
<entry key="Java is an Island" value="Sanjay"></entry>
</map>
</constructor-arg>
</bean>
</beans>
```

Test.java

```
package com.nareshit;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;
public class Test {
public static void main(String[] args) {
Resource resource=new ClassPathResource("applicationContext.xml");
BeanFactory factory=new XmlBeanFactory(resource);
Question question=(Question)factory.getBean("question");
question.displayInfo();
}
}
```

Injecting Map Object(where Map conatins User-Defined Object Types)



```
Injecting Map Object(where Map Contians userdefined Object Types)
└───src
    │   Answer.java
    │   applicationContext.xml
    │   Question.java
    │   Test.java
    │   User.java
    │
    └───com
        └───nareshit
                Answer.class
                Question.class
                Test.class
                User.class
```

User.java

```
package com.nareshit;
public class User {
private int id;
private String name,email;
public String toString(){
        return "User Id:"+id+" Name:"+name+"Email Id:"+email;
}
//required setters and gettes
}
```

Answer.java

```
package com.nareshit;
import java.util.Date;
public class Answer {
private int id;
private String answer;
private Date postedDate;
public String toString(){
return "Answer Id:"+id+" Answer:"+answer+" Posted Date:"+postedDate;
}
//required setters and getters
}
```

Question.java

```
package com.nareshit;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.Map.Entry;
public class Question {
private int id;
private String name;
private Map<Answer,User> answers;
public int getId() {
        return id;
}
public void setId(int id) {
        this.id = id;
}
public String getName() {
        return name;
}
public void setName(String name) {
        this.name = name;
}
public Map<Answer, User> getAnswers() {
        return answers;
}
public void setAnswers(Map<Answer, User> answers) {
        this.answers = answers;
}
public void displayInfo(){
        System.out.println("question id:"+id);
        System.out.println("question name:"+name);
        System.out.println("Answers....");
Set<Entry<Answer, User>> set=answers.entrySet();
Iterator<Entry<Answer, User>> itr=set.iterator();
        while(itr.hasNext()){
                Entry<Answer, User> entry=itr.next();
                Answer ans=entry.getKey();
                User user=entry.getValue();
        System.out.println("Answer information:");
                System.out.println(ans);
                System.out.println("Posted By:");
                System.out.println(user);
        }
}
}
```

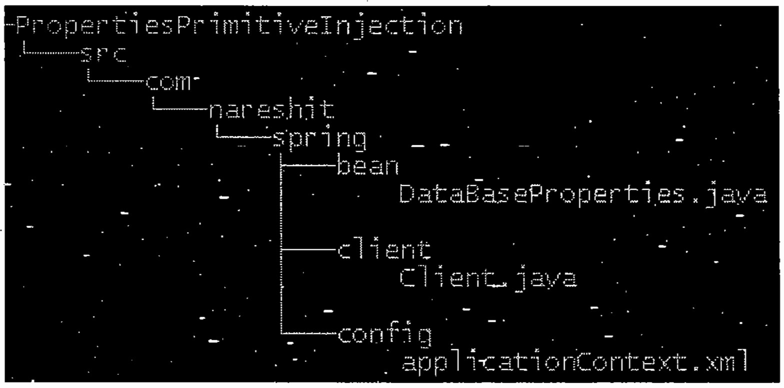
applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
        xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="answer1" class="com.nareshit.Answer">
<property name="id" value="1"/>
<property name="answer"  value="Java is a Programming Language"/>
<property name="postedDate" value="12/12/2001"/>
</bean>

<bean id="answer2" class="com.nareshit.Answer">
<property name="id" value="2"></property>
<property name="answer"
value="Java is a Platform"></property>
<property name="postedDate" value="12/12/2003"></property>
</bean>

<bean id="user1" class="com.nareshit.User">
<property name="id" value="1"></property>
<property name="name" value="Vijay Kumar">
</property>
<property name="email" value="Vijay@gmail.com"></property>
</bean>

<bean id="user2" class="com.nareshit.User">
<property name="id" value="2"></property>
<property name="name" value="Ajay Kumar"></property>
<property name="email" value="Ajay@gmail.com"></property>
</bean>

<bean id="question" class="com.nareshit.Question">
<property name="id" value="1"></property>
<property name="name" value="What is Java?"></property>
<property name="answers">
<map>
<entry key-ref="answer1" value-ref="user1"></entry>
<entry key-ref="answer2" value-ref="user2"></entry>
</map>
</property>
</bean>
</beans>
```

Test.java

```
package com.nareshit;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;
public class Test {
public static void main(String[] args) {
        Resource resource=new ClassPathResource("applicationContext.xml");
        BeanFactory factory=new XmlBeanFactory(resource);
        Question question=(Question)factory.getBean("question");
        question.displayInfo();
}
}
```

Injecting java.util.Properties Object



```
-PropertiesPrimitiveInjection
 └───src
     └───com
         └───nareshit
             └───spring
                 ├───bean
                 │       DataBaseProperties.java
                 │
                 ├───client
                 │       Client.java
                 │
                 └───config
                         applicationContext.xml
```

DataBaseProperties.java

```
package com.nareshit.spring.bean;
import java.util.Properties;
import java.util.Enumeration;
public class DataBaseProperties {
private Properties dbProperties;

public Properties getDbProperties() {
        return dbProperties;
}
public void setDbProperties(Properties dbProperties) {
        this.dbProperties = dbProperties;
}
public void displayDetails(){
Enumeration enumeration=dbProperties.keys();
while(enumeration.hasMoreElements())
        {
String id=(String)enumeration.nextElement();
String name=dbProperties.getProperty(id);
        System.out.println(id+"--->"+name);
        }
}
}
```

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
        xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
<bean id="databaseProperties"
class="com.nareshit.spring.bean.DataBaseProperties">
<property name="dbProperties">
<props>
<prop key="driverClass">
oracle.jdbc.driver.OracleDriver</prop>
<prop key="userName">system</prop>
<prop key="password">sathish</prop>
<prop key="url">
jdbc:oracle:thin:@localhost:1521:XE"</prop>
</props>
</property>
</bean>
</beans>
```

Client.java

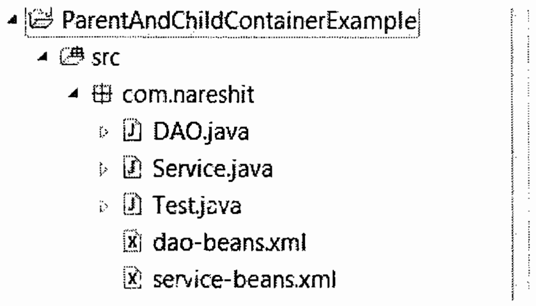
```
package com.nareshit.spring.client;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;
import com.nareshit.spring.bean.DataBaseProperties;
public class Client {
        public static void main(String[] args) {
Resource resource=new ClassPathResource("com/nareshit/spring/config/applicationContext.xml");
BeanFactory factory=new XmlBeanFactory(resource);
DataBaseProperties bean=(DataBaseProperties)factory.getBean("databaseProperties ");
bean.displayDetails();

        }
}
```

what is parent and child containers in Spring ?

- Spring supports setting parent-child relationship between two IOC containers
- If we have two beanFactory containers in the Application, we can set one BeanFactory into Another beanFactory to allow the beans in one bean factory to refer to the beans of other factory.in this we can declare one beanfactory as a parent and other as a child. This is similar to concept of base class and

Derived classes.Derived Class can access the properties of base class,but base classs can not access the properties of derived class.

In order to refer to parent beans, we can use tag <ref parent=" "/>
Apart from parent attribute it has local which indicates refer to the local bean.Along with it we have bean attribute as well,which indicates look in local first if not found then search in parent factory and peform Injection.



```
ParentAndChildContainerExample
  src
    com.nareshit
      DAO.java
      Service.java
      Test.java
      dao-beans.xml
      service-beans.xml
```

DAO.java

```
package com.nareshit;
public class DAO {
public void daoMethod(){
        System.out.println("daoMethod");
}
}
```

Service.java

```
package com.nareshit;
public class Service {
        private DAO dao;

public DAO getDao() {
                return dao;
        }

        public void setDao(DAO dao) {
                this.dao = dao;
        }

public void serviceMethod(){
        System.out.println("serviceMethod");
        dao.daoMethod();
}
}
```

dao-beans.xml

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                        "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
<bean id="dao" class="com.nareshit.DAO"/>
</beans>
```

Service-beans.xml

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

```
<beans>
<bean id="service" class="com.nareshit.Service">
<property name="dao">
<ref bean="dao"/>
</property>
        </bean>
</beans>
```

Test.java

```
package com.nareshit;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;
public class Test {
        public static void main(String[] args) {
                Resource parentResource = new ClassPathResource(
                                "com/nareshit/dao-beans.xml");
                BeanFactory parentFactory = new XmlBeanFactory(parentResource);
                Resource childResource = new ClassPathResource(
                                "com/nareshit/service-beans.xml");
                BeanFactory childFactory = new XmlBeanFactory(childResource,
                                parentFactory);
                Service service = childFactory.getBean("service", Service.class);
                service.serviceMethod();
        }
}
```

ApplicationContext container:-

It is an enhancement for BeanFactory Container. To activate Application Context Container we can create object for the Java class that implements Application Context interface. This is sub interface of BeanFactory(Interface).
There are three important implementation classes for ApplicationContext(Interface).
They are

1) FileSystemXmlApplicationContext

2) ClassPathXmlApplicatinContext

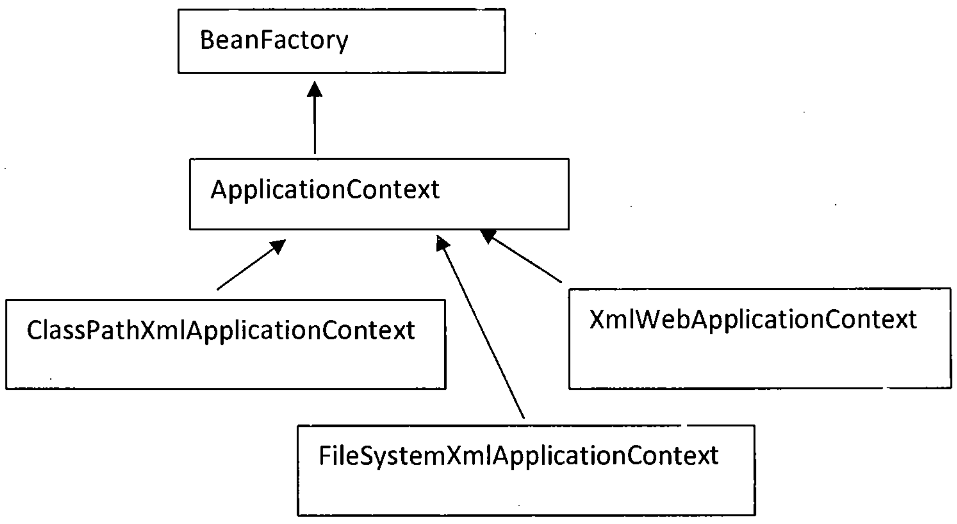
3) XmlWebApplicationcontext

Note: Application Context container is originally part of Spring context/spring JEE module.

Application Context Container can perform every activity of BeanFactory container but it also gives the following additional features.

1. Ability to work with multiple Spring cfg files.
2. Pre-Instantiation of Singleton class.
3. Ability to work with properties files.
4. Support to work with I18n(Internationalization)
5. Ability to work with Event Listeners and etc.,

By activating BeanFactory container for one time we can't make that container dealing multiple Spring configuration files but this activity is possible by using Application Context Container.


BeanFactory
ApplicationContext
ClassPathXmlApplicationContext
XmlWebApplicationContext
FileSystemXmlApplicationContext

Using FileSystemXmlApplicationContext:-
ApplicationContext context=new
FileSystemXmlApplicationContext("D://Sathish/spring/workspace/springcontainer/src/mybeans.xml
");
If we use FileSystemXmlApplicationContext container ,we need to pass absolute
Path of the spring configuration file to this constructor.so if the project location is
Changes we need to change our application code.so it is not recomned to use.

Using ClasspathXmlApplicationContext
ApplicationContext context= new ClassPathXmlApplicationContext("mybeans.xml");

In the above code container reading the configuration file
From class path,but we are not specifying the complete path(Absolute path)
Even if we change the location of the Project in the system still our application work
Without modification. Because it reads the configuration file from classpath.
So recomnded to use ClassPathXmlApplicationContext container.

Using XmlWebApplicationContext:-
This is used in Spring webmvc module.This container object created
By web container internally,we no need to create our own.

Key notes on ApplicationContext:-
It loads spring beans configured in spring configuration file. And manages

The Life cycle of the Spring bean as and when container starts. It won't wait until getBean() is called.

Application context container is generally used in Enterprise based applications.
Auto wiring:-

Auto wiring means automatically injecting the dependencies.(implict DependecyInjection)

Insead of manually configuring(Explict DIO) the injection we can done automatically by using autowiring.

To implement autowiring we use “autowire” attribute in <bean> tag configuration.

To perform autowiring for a particular bean we can use any one of the following values

1)byName

2)byType

3)contructor

4)autodetect

1. by using autowire attribute Auto wiring is possible only on reference type bean properties that means auto wiring is not possible on simple, Collection Framework type properties.

2. Auto wiring kills the readability of Spring configuration file.

byName:-

Mainly it checks for three conditions if all these are valid then it injects the values by setter approach

1.Dependency bean name

2.configured bean name

3.setter method name

-> if dependency name is abc,bean configuration should be abc and the setter Name method should i,e. setAbc(Abc abc)

when it finds autowire="byName" for any bean configuration ,then it first checks for dependency bean name in the dependent bean.

Then it will checks weather any bean is configured in the spring cfg file with the same name.

if it finds then it will call corresponding setterMethod of dependent bean.

Autowiring byType:-

Mainly it checks for theree conditions if all these are valid then it injects the values by setterappraoch

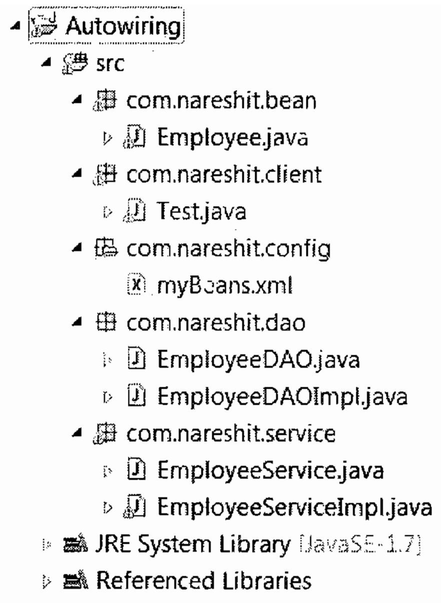
1.Dependency beanType

2.configured bean Type

3.setter method Argument Type

when it finds "auto wire="byType" for any bean configuration ,then it first checks for dependency

bean Type in the dependent bean then after it will checks weather any bean is configured in the spring cfg file with the same Type. if it finds then it will call corresponding setterMethod .


```
Autowiring
  src
    com.nareshit.bean
      Employee.java
    com.nareshit.client
      Test.java
    com.nareshit.config
      myBeans.xml
    com.nareshit.dao
      EmployeeDAO.java
      EmployeeDAOImpl.java
    com.nareshit.service
      EmployeeService.java
      EmployeeServiceImpl.java
  JRE System Library [JavaSE-1.7]
  Referenced Libraries
```

EmployeeService.java

```
package com.nareshit.service;
import com.nareshit.bean.Employee;
public interface EmployeeService {
        Employee searchEmployee(int empNo);
}
```

EmployeeServiceImpl.java

```
package com.nareshit.service;

import com.nareshit.bean.Employee;
import com.nareshit.dao.EmployeeDAO;

public class EmployeeServiceImpl implements EmployeeService {
        private EmployeeDAO employeeDao;

        public void setEmployeeDao(EmployeeDAO employeeDao) {
                this.employeeDao = employeeDao;
        }

        public Employee searchEmployee(int empNo) {
                Employee emp = employeeDao.searchEmployee(empNo);
                return emp;
        }

}
```

EmployeeDAO.java

```
package com.nareshit.dao;
import com.nareshit.bean.Employee;
public interface EmployeeDAO {
public Employee searchEmployee(int empNo);
}
```

EmployeeDAOImpl.java

```
package com.nareshit.dao;
```

```
import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import javax.sql.DataSource;

import com.nareshit.bean.Employee;

public class EmployeeDAOImpl implements EmployeeDAO {

        private DataSource dataSource;


        public void setDataSource(DataSource dataSource) {

                this.dataSource = dataSource;

        }


        public Employee searchEmployee(int empNo) {

                Employee emp = null;

                Connection con = null;

                try {

                        con = dataSource.getConnection();

                        String sql = "select *from employee where empNo=?";

                        PreparedStatement pst = con.prepareStatement(sql);

                        pst.setInt(1, empNo);

                        ResultSet rs = pst.executeQuery();

                        if (rs.next()) {

                                emp = new Employee();

                                emp.setEmpNo(rs.getInt("empNo"));

                                emp.setName(rs.getString("name"));

                                emp.setSalary(rs.getDouble("salary"));

                        }


                } catch (SQLException sqlException) {
                        System.out
                                    .println("Exception Occured while searching Employee() ::"
                                                + sqlException.getMessage());
                } finally {
                        if (con != null) {
                                try {
                                        con.close();
                                } catch (SQLException sqlException) {
                                        System.out
                                                    .println("Exception OCcured while closing
the connection ::"
                                                                    +
sqlException.getMessage());
                                }
                        }
                }

                return emp;
        }
}
```

myBeans.xml

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
            "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
<property name="url" value="jdbc:oracle:thin:@localhost:1521:XE"/>
<property name="username" value="system"/>
<property name="password" value="manager"/>
</bean>
<bean id="empDao"
class="com.nareshit.dao.EmployeeDAOImpl" autowire="byName">
</bean>
<bean id="empService"
class="com.nareshit.service.EmployeeServiceImpl" autowire="byType">
</bean>
</beans>
```

Employee.java

```
package com.nareshit.bean;
import java.io.Serializable;
public class Employee implements Serializable {
        private int empNo;
        private String name;
        private double salary;
//required setters and getters

}
```

Test.java

```
package com.nareshit.client;
import java.util.List;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import com.nareshit.bean.Employee;
import com.nareshit.service.EmployeeService;

public class Test {
        public static void main(String[] args) {
                String configFile = "com/nareshit/config/myBeans.xml";
                ClassPathResource resource = new ClassPathResource(configFile);
                XmlBeanFactory factory = new XmlBeanFactory(resource);
                EmployeeService empService = (EmployeeService) factory
                                .getBean("empService");
                System.out.println("Testing of searchEmployee(-) :");
                Employee emp = empService.searchEmployee(1004);
                if (emp != null) {
                        System.out.printin("Emp No :" + emp.getEmpNo());
                        System.out.println("Name :" + emp.getName());
                        System.out.println("Salary :" + emp.getSalary());

                } else {
                        System.out.println("Emp Not Found");
                }

        }
}
```

autoWiring Constructors:-

Mainly it checks for theree conditions if all these are valid then it injects the values by constructor appraoch

1.Dependency beanType

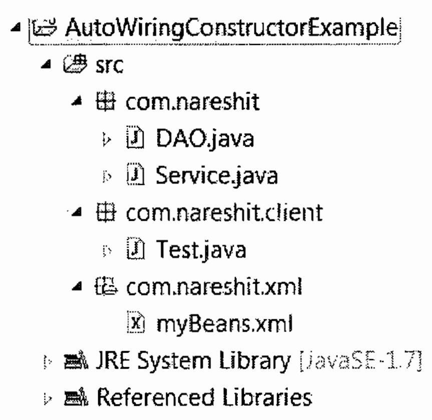
2.configured bean Type

3.constructor Argument Type

when it finds "autowire="constructor" for any bean configuration ,then it first checks for dependency bean Type in the dependent bean then it will checks spring cfg file

weather any bean is confiured with the dependency type,if it finds it will check for constructor

which will takes dependency type as an argument.then will call corresponding constructor.



```
AutoWiringConstructorExample
  src
    com.nareshit
      DAO.java
      Service.java
    com.nareshit.client
      Test.java
    com.nareshit.xml
      myBeans.xml
  JRE System Library [JavaSE-1.7]
  Referenced Libraries
```

Service.java

```java
package com.nareshit;
public class Service {
private DAO dao;
public Service(DAO dao){
        this.dao=dao;
}
public void serviceMethod(){
        System.out.println("Service Method");
        dao.daoMethod();
}
}
```

DAO.java

```java
package com.nareshit;
public class DAO {
public void daoMethod(){
System.out.println("daoMethod");
}
}
```

myBeans.xml

```xml
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                              "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
<bean id="serviceObj" class="com.nareshit.Service" autowire="constructor">
</bean>
<bean id="daoObj" class="com.nareshit.DAO"/>
</beans>
```

Test.java

```java
package com.nareshit.client;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.nareshit.Service;
public class Test {
public static void main(String[] args){
ApplicationContext context=new
ClassPathXmlApplicationContext("com/nareshit/xml/myBeans.xml");
Service service=(Service)context.getBean("serviceObj");
service.serviceMethod();
}
}
```

autoWiring (auto detect):-

auto detect chooses "constructor"(or) byType through injection of the bean class.

always auto-detect first gives preference to constructor

if a zero-arg constructor is found, "byType" gets applied. If not found it will apply constructor.

Autodetect already removed in spring-3.0 xsd so if we are using

spring-3.0. xsd not possible to work with autodetect.

Note :- autowire="no" →there is no autowiring

Global default-autowiring

Instead of defining autowire attribute for every bean configuration we can set a default-autowire attribute in the<beans> root element to force the entire beans declared within <beans> root element to apply this rule.However this root default mode will be overridden by a bean's own mode if it specified

<beans......... default-autowire="byType">

(or)byName

(or)constructor

(or)autodetect

mybeans.xml

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

<beans default-autowire="byName"

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

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

        xmlns:p="http://www.springframework.org/schema/p"

        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="service" class="com.nareshit.Service"/>

<bean id="dao " class="com.nareshit.DAO" >

</bean>

</beans>
```

Dependency-checking:-

It is used to verify all dependencies of bean that are configured via injection are injected or not

To implement dependency checking to a spring bean we make use of 'dependency-check' attribute of <bean> tag.

this attribute takes any one of the four following values

1) none :- it won't check wheter dependencies injected or not

2) simple :-it checks simple type dependencies injected or not

3) objects:- it checks all the object type dependencies injected or not

4) all:- it checks both simple and object type dependencies injected or not

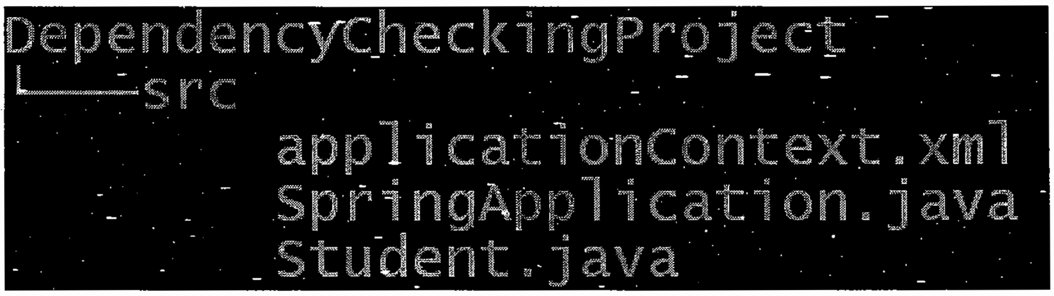
In case if we forgot to set any value either

Simple (or) object type it raise an Exception **like "UnSatisfiedDependencyException"**

Note :

By default a spring beans "dependency-check" value is"none"

To change this default nature of all the beans of the applicationContext we use "default-dependency-check" attribute of<beans> tag.



```
DependencyCheckingProject
└──src
        applicationContext.xml
        SpringApplication.java
        Student.java
```

Student.java

```
public class Student {

        private int regno;

        private String name,course;

        private float fees;

        public int getRegno() {

                return regno;

        }

        public void setRegno(int regno) {

                this.regno = regno;

        }

        public String getName() {

                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public String getCourse() {
                return course;
        }
        public void setCourse(String course) {
                this.course = course;
        }
        public float getFees() {
                return fees;
        }
        public void setFees(float fees) {
                this.fees = fees;
        }
        public void displayStudentDetails(){
                System.out.println("Reg No:"+regno);
                System.out.println("Name :"+name);
                System.out.println("Course :"+course);
                System.out.println("Fees :"+fees);
        }
}
```

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
        xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:p="http://www.springframework.org/schema/p"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

<bean id="std"  class="Student"  dependency-check="simple">

<property name="course" value="c"/>

<property name="fees" value="500"/>

<property name="name" value="xyz"/>

</bean>

</beans>
```

<u>SpringApplication.java</u>

```
import org.springframework.beans.factory.BeanFactory;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;

import org.springframework.core.io.Resource;

public class SpringApplication {

public static void main(String[] args) {

        Resource resource=new ClassPathResource("applicationContext.xml");

        BeanFactory factory=new XmlBeanFactory(resource);

        Student std=(Student)factory.getBean("std");

        std.displayStudentDetails();

}

}
```

Note:-

In the above Example We will get the Following Exception

```
org.springframework.beans.factory.UnsatisfiedDependencyException:

Error creating bean with name 'std'

defined in class path resource [applicationContext.xml]:

Unsatisfied dependency expressed through bean property 'regno':

Set this property value or disable dependency checking for this bean.
```

<u>Global default-dependency checking</u>

Explicitly define the dependency checking mode for every beans is tedious and error prone,you can set a default-dependency-check attribute in the<beans> root element to force the entire beans declared within <beans> root element to apply this rule.However this root default mode will be overridden by a bean's own mode if it specified

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

<beans default-dependency-check="all"

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

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

        xmlns:p="http://www.springframework.org/schema/p"

        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

<bean id="std"   class="Student" >

<property name="course" value="c"/>

<property name="fees" value="500"/>

<property name="name" value="xyz"/>

</bean>

</beans>
```

Spring Annotation Support :-

Spring 2.0 has a very little support to annotation, when it comes to spring 2.5 , it has extended it's framework to support various aspects of spring development using annotations. In spring 3.0 it started supporting java config project annotations as well.

In Spring 2.0 it has added @Repository and @Required annotations.

In Spring 2.5 it has added few more annotations @Autowired,@Qualifier and @Scope.

along with above annotation's in spring 2.5 sterotype annotations @Component,@Controller, @Service .

In Spring 3.0 few more annotations has been added like @Lazy,@Bean,@Configuration,@DependsOn,@Value etc..

In addition to spring based metadata annotation support,it has started adoption JSR-250 java config project annotations like @PostConstruct,@PreDestory,@Resource ,@Inject and @Named etc...

@Required annotation :- In spring 1.x onwords we have dependency

check.for a bean if you want to detect unresolved dependencies of a bean,you can use dependency check. In Spring 2.0 it has introduced an annotation @Required . by using this annotation you can

make , a particular property has been set with value (OR) not. in spring 2.5 @Required annotation and dependency -check completly removed.

@Autowired :- annotation to autowire bean on the setter method, constructor(OR) a filed.

Example :-

EmployeeService.java

```
public class EmployeeService{

@Autowired

private EmployeeDAO  empDao;


}
```

EmployeeDAO.java

```
public class EmployeeDAO{

@Autowired
private DataSource dataSource;


}
```

The @Autowired Annotation is highly flexible and powerful,and better than <autowire> attribute in bean configuration file.

About the @Autowired annotation to give an intimation to spring IOC container we can use

<context:annotation-config/> in spring file.

Note :

By default ,the @Autowired will perform the dependency checking to make sure the property has been wired properly. When Spring container can't find a matching bean to wire, it will throw an Exception. if you want to disable the dependency checking feature then we can use the "required" attribute of @Autowired annotation. if the required attribute value is false then the dependency-checking is disabled. else dependency -checking is enabled. the default value of required attrbute is true.

Example :-

```
public class EmployeeService{

@Autowired(required=false)

private EmployeeDAO empDao;


}
```

@Quälifier:-

for example,In bean configuration file for EmployeeDAO bean two configuration's are there.

```
<beans>

<bean id="employeeDao1" class="com.nareshit.dao.EmployeeDAO"/>

<bean id="employeeDao2" class="com.nareshit.dao.EmployeeDAO"/>

</beans>
```

In EmployeeService class

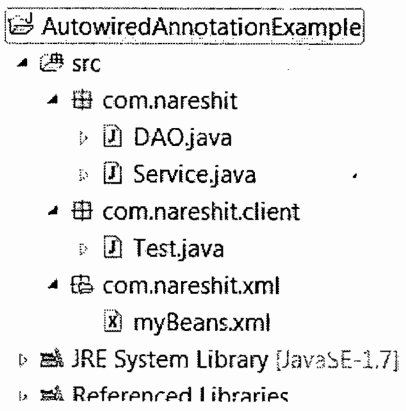
```
public EmployeeService{

@Autowired

private EmployeeDAO  employeeDao;

}
```

here employeeDao1(OR) employeeDao2 which bean is required to inject don't know by the container so Container Will rise An Exception. To Solve this problem and to inject a particular bean i,e employeeDao1 (OR) employeeDao2 we can use @Qualifier annotation

The @Qualifier annotation is used to control which bean should be autowire on a field.

Example:-

```
public class EmployeeService{

@Autowired

@Qualifier("employeeDao1")

private EmployeeDAO empDao;


}
```



```
AutowiredAnnotationExample
  src
    com.nareshit
      DAO.java
      Service.java
    com.nareshit.client
      Test.java
    com.nareshit.xml
      myBeans.xml
  JRE System Library [JavaSE-1.7]
  Referenced Libraries
```

DAO.java

```
package com.nareshit;
public class DAO{
        public void daoMethod(){
                System.out.println("DAOMethod");
        }
}
```

Service.java

```
package com.nareshit;
import org.springframework.beans.factory.annotation.Autowired;
public class Service {
        @Autowired
        private DAO dao;
        public void serviceMethod() {
                System.out.println("serviceMethod");
                dao.daoMethod();
        }
}
```

myBeans.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">
<context:annotation-config />
<bean id="dao" class="com.nareshit.DAO" />
<bean id="service" class="com.nareshit.Service" />
</beans>
```

Test.java

```
package com.nareshit.client;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.nareshit.Service;
public class Test{
        public static void main(String[] args){
        ApplicationContext context=
        new ClassPathXmlApplicationContext("com/nareshit/xml/myBeans.xml");
        Service service=(Service)context.getBean("service");
        service.serviceMethod();
        }

}
```

Stereotype Annotations :

In Spring there are 4 Stereotype Annotations

1)@Component : Indicates an auto scan component.

2) @Service – Indicates a Service component in the business layer/service Layer.

3) @Repository – Indicates DAO component in the persistence layer.

4) @Controller – Indicates a controller component in the Controller layer.

Example 1 :-

```
@Component

public  class EmployeeService{


}
```

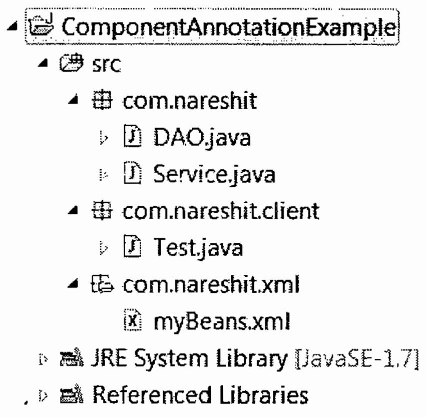
Example 2 :-

```
@Component("empService")

public  class EmployeeService{


}
```

@Component Annotation is indicating to the container EmployeeService is class is an auto scan component.

About the Stereotype annotation to give an intimation to the container we can use

<context:component-scan base-package="com.nareshit"/> in spring configuration file.



```
ComponentAnnotationExample
  src
    com.nareshit
      DAO.java
      Service.java
    com.nareshit.client
      Test.java
    com.nareshit.xml
      myBeans.xml
  JRE System Library [JavaSE-1.7]
  Referenced Libraries
```

DAO.java

```
package com.nareshit;
import org.springframework.stereotype.Component;
@Component
public class DAO{
        public void daoMethod(){
                System.out.println("DAOMethod");
        }
}
```

Service.java

```
package com.nareshit;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
@Component("service")
public class Service {
        @Autowired
        private DAO dao;
        public void serviceMethod() {
                System.out.println("serviceMethod");
                dao.daoMethod();
        }
}
```

myBeans.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">
<context:component-scan base-package="com.nareshit">
</context:component-scan>
</beans>
```

Test.java

```
package com.nareshit.client;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.nareshit.Service;
public class Test{
        public static void main(String[] args){
ApplicationContext context=
                new ClassPathXmlApplicationContext("com/nareshit/xml/myBeans.xml");
        Service service=(Service)context.getBean("service");
        service.serviceMethod();
        }

}
```

The @Repository, @Service and @Controller are annotated with @Component annotation internally. the Three annotation's also component Type annotations.
to increase the readability of the program we can use @Service annotation in Service Layer

@Repository annotation in DAO/Persistence Layer and @Contoller annotation in Controller Layer

Instead of @Component type annotation.

Example :

EmployeeService.java

```
@Service

public class EmployeeService{


}
```

EmployeeDAO.java

```
@Repository

public class EmployeeDAO {



}
```

@Value Annotation

@Value Annotation is given spring 3.0 version. It is used to inject the values on simple type dependencies

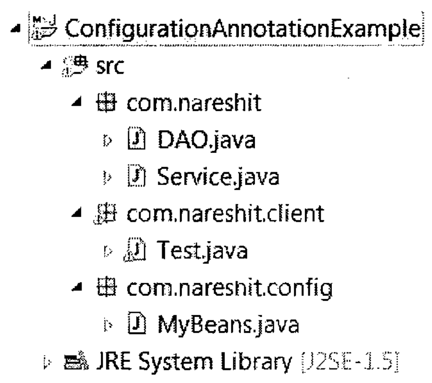
```
public class  CustomerBean{

@Value("1001")
private int customerId;

@Value("sathish")

private String customerName;


}
```

working with @Configuration and @Bean :-

Instead of declaring a class as spring bean in a configuration file, you can declare it in a class. The class in which you want to provide the configuration about other beans, that class is called configuration class and you need to annotate with @Configuration.

In this class you need to provide the methods which are responsible for creating object's of your bean classes, these methods has to be annotated with @Bean.

Example :-



```
ConfigurationAnnotationExample
  src
    com.nareshit
      DAO.java
      Service.java
    com.nareshit.client
      Test.java
    com.nareshit.config
      MyBeans.java
  JRE System Library [J2SE-1.5]
```

Service.java

```
package com.nareshit;
public class Service {
        private DAO dao;

        public void setDao(DAO dao) {
                this.dao = dao;
        }

        public void serviceMethod() {
                System.out.println("serviceMethod");
                dao.daoMethod();
        }
}
```

DAO.java

```
package com.nareshit;
public class DAO{
        public void daoMethod(){
                System.out.println("DAOMethod");
        }
}
```

MyBeans.java

```
package com.nareshit.config;

import org.springframework.beans.factory.annotation.Autowire;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import com.nareshit.DAO;

import com.nareshit.Service;

@Configuration

public class MyBeans {

        @Bean(name="service",autowire=Autowire.BY_TYPE)
public Service service(){

        return new Service();

}

        @Bean(name="dao")

        public DAO dao(){

        return   new DAO();

        }

}
```

Test.java

```
package com.nareshit.client;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.nareshit.Service;

public class Test{

        public static void main(String[] args){

        ApplicationContext context=

        new AnnotationConfigApplicationContext(com.nareshit.config.MyBeans.class);


        Service service=(Service)context.getBean("service");

        service.serviceMethod();

        }


}
```

@Inject :-

@Inject is J2ee specific annotation,used for Injectiong/autowiring one class into another.

This is similar to @Autowired spring annotation. But the difference between @Autowired supports required attribute where @Inject doesn't has it. @Inject also injects a bean into another bean as similar to @Autowired. it is present in javax.inject package.

Example :

EmployeeService.java

```
public class EmployeeService{

@Inject

private EmployeeDAO empDao;


}
```

EmployeeDAO.java

```
public class EmployeeDAO{

@Inject

private DataSource dataSource;


}
```

@Named it is a J2EE annotation and It is similar to @Qualifier spring Annotation

@Resource is a J2EE annotation similar to @Inject but it will perform the injection by using byName rules.

Factory-Methods :

we can make the spring container to create spring bean class Object either by using static factory method (OR) instance factory method (OR) regular constructors.

with static Factory-method:-

For some java classes object creation is possible only with static-factory-method's for that kind of classes to create object with static-factory-method we can use "factory-method" attribute along with "class" attribute of "bean" tag and give an intimation to spring container.

Example :-

```
<bean id="cls" class="java.lang.Class"  factory-method="forName">

<constructor-arg  value="oracle.jdbc.driver.OracleDriver"/>

</bean>
```

Note : To specifies the argument values of a factory method use <constructor-arg> tag

With Instance-Factory method :

All the classes object's are cannot be created by using new Operator,few can be constructed by calling factory method on the class,but few objects can be constructed by calling methods on other classes.

So, In order to create such type of Objects you need to instantiate by using Instance-factory methods.

Example :

```
<beans>

<bean name="student1" class="Student" ></bean>

<bean  name="student2"  factory-bean="student1" factory-method="getStudent"/>

</beans>
```

Student.class

```
public class Student{
public Student getStudent(){
return new Student();
}
}
```

Note :-

while configuring spring bean "factory-method" attribute is palces along with "class" attribute then spring container uses static factorymethod for instantiation.

while configuring spring bean if "factory-method" attribute is placed along with "factory-bean" attribute and without class attribute then spring container uses instance-factory method to create spring bean class Object.

Spring Bean Life Cycle

In addition to bean registration, the Spring IOC container is also responsible for managing the life cycle
of your beans, and it allows you to perform custom tasks at particular points of their life cycle. Your tasks should be encapsulated in callback methods for the Spring IOC container to call at a suitable time.

The following list shows the steps through which the Spring IOC container manages the life cycle of a bean. This list will be expanded as more features of the IOC container are introduced.

1. Create the bean instance either by a constructor or by a factory method.
2. Set the values and bean references to the bean properties.
3. Call the initialization callback methods.
4. The bean is ready to be used.
5. When the container is shut down, call the destruction callback methods

----

factory.destroySingletons();

Setting the init-method and destroy-method Attributes

A better approach of specifying the initialization and destruction callback methods is by setting the init-method and destroy-method attributes in your bean declaration.
With these two attributes set in the bean declaration, your Cashier class no longer needs to implement the InitializingBean and DisposableBean interfaces. You can also delete the afterPropertiesSet() and destroy() methods as well.

In the Cashier class, the openStream() method creates printWriter Stream based given filePath and fileName.
the closeStream() method closes the printWriter stream to release its system resources.

The openStream() and closeStream() methods are declared as callback Initialization and call back destruction methods by using init-method and destroy-method attributes in the bean configuration file as follows.

In spring cfg file

```
<beans>
<bean id="cashier"  class="com.nareshit.domain.Cashier"
 init-method="openStream" destroy-method="closeStream">
<property name="fileName" value="productsInformation"/>
<property name="filePath" value="G://spring/"/>
</bean>
</beans>
```

And Cashier class as follows

Cashier.java

```
public class Cashier  {
private double totalPrice=0.0;
private String fileName;
private String filePath;
private PrintWriter printWriter;
public void setFileName(String fileName) {
        this.fileName = fileName;
}
public void setFilePath(String filePath) {
        this.filePath = filePath;
}
public void openStream(){
        try{
        printWriter=new PrintWriter(filePath+"/"+fileName+".txt");
        }
        catch(FileNotFoundException fnfe){
System.out.println("Exception Occured while executing "
                + "the openStream() ::"+fnfe.getMessage());
        }
}
public void calculateTotalPrice(ShoppingCart cart){
        List<Product> list= cart.getProducts();
  System.out.println(list.getClass());
        for(Product product:list){
                totalPrice=totalPrice+product.getPrice();
                writeProductToFile(product);
        }
        printWriter.println("================");
        printWriter.println("Total Price :"+totalPrice);
        printWriter.flush();
}
public void writeProductToFile(Product product){
        printWriter.println(new Date()+"\t"+product.getProductName()+"\t"+product.getPrice());
}
public void closeStream(){
        printWriter.close();
}
}
```

By using @PostContruct and @PreDestroy

```
public class Cashier{

---

---

---

@PostContruct

public void openStream(){
}

@PreDestroy

public void closeStream(){
}
}
```

Q) What is container call back method (or) Container Life cycle method?

A) The method that is called by underlying container automatically based on the Event that is raised is called as container callback method (or) container Life cycle method.

Creating Bean Post Processors

You would like to register your own plug-ins in the Spring IoC container to process the bean instances
during construction.

A bean post processor allows additional bean processing before and after the initialization callback method.

BeanPostProcessor is used to provide some common logic to all the configured beans in Spring cfg file.

Typically, bean post processors are used for checking the validity of bean properties or altering bean properties according to particular criteria.

The basic requirement of a bean post processor is to implement the BeanPostProcessor interface.

You can process every bean before and after the initialization callback method by implementing the postProcessBeforeInitialization() and postProcessAfterInitialization() methods.

The Spring IOC Container will pass each bean instance to these two methods before and after calling the initialization callback method, as illustrated in the following list:

1. Create the bean instance either by a constructor or by a factory method.
2. Set the values and bean references to the bean properties.
3. Call the setter methods defined in the aware interfaces.
4. Pass the bean instance to the postProcessBeforeInitialization() method of each bean post processor.
5. Call the initialization callback methods.
6. Pass the bean instance to the postProcessAfterInitialization() method of each bean post processor.
7. The bean is ready to be used.
8. When the container is shut down, call the destruction callback methods.

When using a bean factory as your IoC container, bean post processors can only be registered programmatically, or more accurately, via the addBeanPostProcessor() method. However, if you are using an application context, the registration will be as simple as declaring an instance of the processor
in the bean configuration file, and then it will get registered automatically.

How It Works

Suppose you would like check the directory is existed (OR) not before executing Cashier class openStream method(callback Initialization method) in the above example to avoid the FileNotFoundException
We can implement the required logic by using BeanPostProcessor methods.

Cashier.java

Cashier Bean is Same as the Above Example

MyBeanPostProcessor.java

```
public class MyBeanPostProcessor implements BeanPostProcessor {
        public Object postProcessAfterInitialization(Object beanObject,
                        String beanName) throws BeansException {
                System.out.println("postProcessAfterInitialization() ::" + beanName);

                return beanObject;
        }
        public Object postProcessBeforeInitialization(Object beanObject,
                        String beanName) throws BeansException {
                System.out.println("postProcessBeforeInitialization() ::" + beanName);
                if (beanObject instanceof Cashier) {
                        Cashier cashier = (Cashier) beanObject;
                        String directory = cashier.getFilePath();
                        File file = new File(directory);
                        boolean b = file.mkdirs();
                        if (b == true) {
                                System.out.println("Directory is created ::" + directory);
                        } else {
                                System.out println("Directory is already existed ");
                        }
                }
                return beanObject;
        }
}
```

In Spring cfg file :

```
<beans>

<bean id="cashier"  class="com.nareshit.domain.Cashier"
 init-method="openStream" destroy-method="closeStream">
<property name="fileName" value="productsInformation"/>
<property name="filePath" value="G://spring/"/>
</bean>
<bean id="myBeanPostProcessor"
class="com.nareshit.bean.MyBeanPostProcessor">
</bean>
</beans>
```

Aware Interfaces :

Sometimes we need Spring Framework objects in our beans to perform some operations, for example reading ServletConfig and ServletContext parameters or to know the bean definitions loaded by the ApplicationContext. That's why spring framework provides a bunch of *Aware interfaces that we can implement in our bean classes.

org.springframework.beans.factory.Aware is the root marker interface for all these Aware interfaces. All of the *Aware interfaces are sub-interfaces of Aware and declare a single setter method to be implemented by the bean. Then spring context uses setter-based dependency injection to inject the corresponding objects in the bean and make it available for our use.

.Some of the important Aware interfaces are:

- **ApplicationContextAware** – to inject ApplicationContext object.
- **BeanFactoryAware** – to inject BeanFactory object.
- **BeanNameAware** – to know the bean name defined in the configuration file.
- **ServletContextAware** – to inject ServletContext object in MVC application, example usage is to read context parameters and attributes.
- **ServletConfigAware** – to inject ServletConfig object in MVC application, example usage is to get servlet config parameters.

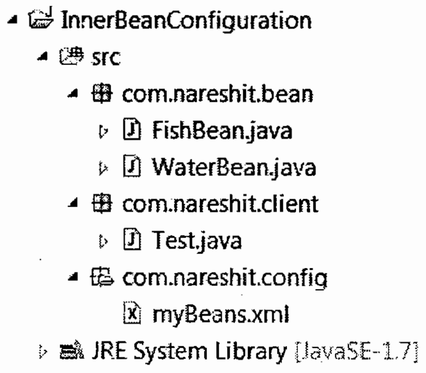
  Let's see these Aware interfaces usage in action by implementing few of them in a class that we will configure as spring bean.

MyAwareService.java

```
public class MyAwareService implements ApplicationContextAware,  BeanNameAware {

  @Override

  public void setApplicationContext(ApplicationContext ctx)  throws BeansException {

    System.out.println("setApplicationContext called");

    System.out.println("setApplicationContext:: Bean Definition Names="

      + Arrays.toString(ctx.getBeanDefinitionNames()));

  }

  @Override

  public void setBeanName(String beanName) {

    System.out.println("setBeanName called");

    System.out.println("setBeanName:: Bean Name defined in context= “  + beanName);

  }

}
```

What is InnerBean?

A spring bean configured with in <property> (OR) <constructor-arg> tag by using <bean> tag is known as Inner bean configuration while configuring Inner beans name/id attributes are not use ful. so we can say inner bean also called anonymous bean.



```
InnerBeanConfiguration
  src
    com.nareshit.bean
      FishBean.java
      WaterBean.java
    com.nareshit.client
      Test.java
    com.nareshit.config
      myBeans.xml
  JRE System Library [JavaSE-1.7]
```

FishBean.java

```
package com.nareshit.bean;
public class FishBean {
private String fishName;
public String getFishName() {
        return fishName;
}
public void setFishName(String fishName) {
        this.fishName = fishName;
}
}
```

WaterBean.java

```
package com.nareshit.bean;
public class WaterBean {
private FishBean fishBean;
public FishBean getFishBean() {
        return fishBean;
}
public void setFishBean(FishBean fishBean) {
        this.fishBean = fishBean;
}
}
```

myBeans.xml

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                        "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
<bean id="waterBean" class="com.nareshit.bean.WaterBean">
                <property name="fishBean">
                <bean class="com.nareshit.bean.FishBean">
                <property name="fishName" value="Goldfish"/>
                </bean>
                </property>
        </bean>
</beans>
```

Test.java

```
package com.nareshit.client;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.nareshit.bean.FishBean;
import com.nareshit.bean.WaterBean;
public class Test {
        public static void main(String[] args) throws Exception {
        String configFile ="com/nareshit/config/myBeans.xml";
        ApplicationContext context=new ClassPathXmlApplicationContext(configFile);
    WaterBean wb = context.getBean("waterBean", WaterBean.class);
        FishBean fb = wb.getFishBean();
        System.out.println("Fish Name :" + fb.getFishName());

        }
}
```

How to work with multiple Spring configuration Files?

when the spring cfg file size huge,it is difficult to maintain ,so in that scenarious we generally have multiple configuration files.
We can work with multiple spring cfg files in two ways
1. By importing other spring configuration file into the current configuration file.
Example:-
 <import resource ="other-spring-beans.xml"/>

2. By passing all spring bean cfg files as a string array to the constructor of the

ApplicationContext interface implementation classes.

ApplicationContext context=new classPathXmlApplicationContext(new String[]{"beans-1.xml","beans-2.xml","beans-3.xml"});

What is Bean Inheritance?

Inheritance is the concept of reusing the existing functionality .
In case of java you can inherit a class fom an interface (OR) another class.when you inherit a class from another class,your child (OR) Derived class get's all the functionalities of Your base class.

When it comes to spring Bean Inheritance,it talks about how to reuse the existing bean configuration instead of re-defining again. Let's consider a scenario where your class contains 5 properties , if we want to configure it as a spring bean you need to inject values for 5 properties via constructor (OR) setter injection.
If we want to create 20 beans of that class ,we need to configure for all the 20 beans for setter (OR)constructor injection. In case If most of the properties has same value,even then also we need to re-write the configuration. This leads to duplicate configuration declaration in high amount of maintenance.
In order to avoid this you can declare the configuration in one bean which acts as parent bean.And All the remaining 19 bean. Declarations can inherit their declaration values from the parent bean,so that we don't need to repeatedly wirte the same configuration in all the child beans.. In this way if we modify the attribute value in parent bean,it will automatically reflects in all Its 19 child beans. The child bean can override the inherited value of parent by re-declaring at the child level.

The following example is show s how to use the feature :

Person.java

```
package com.nareshit.bean;
public class Person {

   private String name;

   private String phone;

   private String email;

   private String city;

   private String country;

  //required setters and getters

public String toString() {
return "Person [name=" + name + ", phone=" + phone + ", email=" + email+ ", city=" + city + ",
country=" + country + "]";
}

 }
```

myBeans.xml

```
<beans>
        <bean id="parentPersonBean" class="com.nareshit.bean.Person">
```

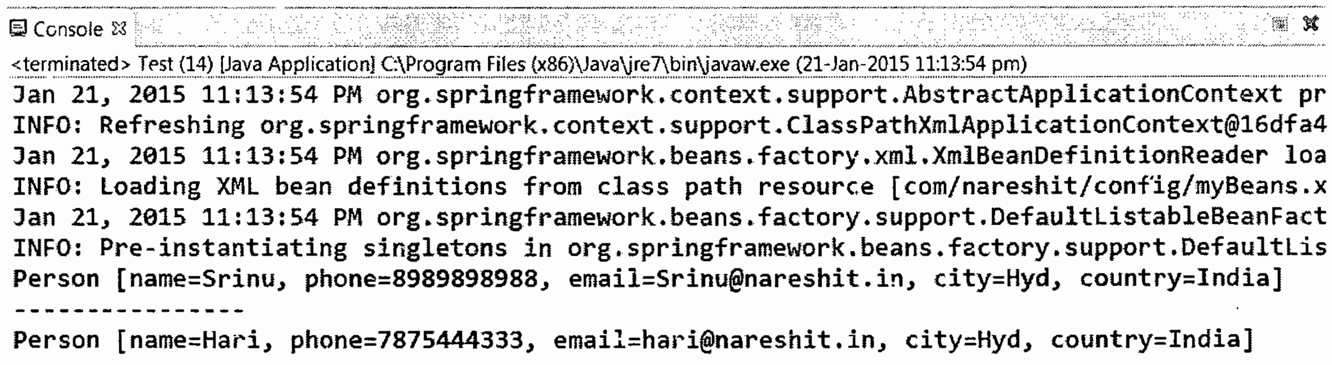
```
            <property name="city" value="Hyd" />
            <property name="country" value="India" />
        </bean>
        <bean id="childPersonBean1" parent="parentPersonBean">
            <property name="name" value="Srinu" />
            <property name="phone" value="8989898988" />
            <property name="email" value="Srinu@nareshit.in" />
        </bean>
        <bean id="childPersonBean2" parent="parentPersonBean">
            <property name="name" value="Hari" />
            <property name="phone" value="7875444333" />
            <property name="email" value="hari@nareshit.in" />
        </bean>

</beans>
```

In the above cfg file 'parentPersonBean' bean contains 'hyd' and 'India' values for city and country properties , and the 'childPersonBean1' bean and 'childPersonBean2' bean inherited city and country properties values from its parent ('parentPersonBean').

Test.java

```
package com.nareshit.client;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.nareshit.bean.Person;
public class Test {
public static void main(String[] args) {
ClassPathXmlApplicationContext context=new
ClassPathXmlApplicationContext("com/nareshit/config/myBeans.xml");
  Person childPersonBean1= context.getBean("childPersonBean1",Person.class);
  Person childPersonBean2= context.getBean("childPersonBean2",Person.class);
System.out.println(childPersonBean1);
System.out.println("----------------");
System.out.println(childPersonBean2);
        }
}
```

Output :



Console ⌧

```
<terminated> Test (14) [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (21-Jan-2015 11:13:54 pm)
Jan 21, 2015 11:13:54 PM org.springframework.context.support.AbstractApplicationContext pr
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@16dfa4
Jan 21, 2015 11:13:54 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loa
INFO: Loading XML bean definitions from class path resource [com/nareshit/config/myBeans.x
Jan 21, 2015 11:13:54 PM org.springframework.beans.factory.support.DefaultListableBeanFact
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultLis
Person [name=Srinu, phone=8989898988, email=Srinu@nareshit.in, city=Hyd, country=India]
----------------
Person [name=Hari, phone=7875444333, email=hari@nareshit.in, city=Hyd, country=India]
```

Inheritance with abstract

In above example, the 'parentPersonBean' is still able to instantiate, for example,

```
            Person parentPersonBean= context.getBean("parentPersonBean",Person.class);
```

If you want to make this base bean as a template and not allow others to instantiate it, you can add an '**abstract**' attribute in the <bean> element. For example

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                              "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
        <bean id="parentPersonBean" class="com.nareshit.bean.Person" abstract="true">
                <property name="city" value="Hyd" />
                <property name="country" value="India" />
        </bean>
        <bean id="childPersonBean1" parent="parentPersonBean">
                <property name="name" value="Srinu" />
                <property name="phone" value="8989898988" />
                <property name="email" value="Srinu@nareshit.in" />
        </bean>
        <bean id="childPersonBean2" parent="parentPersonBean">
                <property name="name" value="Hari" />
                <property name="phone" value="7875444333" />
                <property name="email" value="hari@nareshit.in" />
        </bean>

</beans>
```

Now, the 'parentPersonBean' bean is a pure template, for bean to inherit it only, if you try to instantiate it, you will encounter the following error message.

Person parentPersonBean= context.getBean("parentPersonBean",Person.class);

```
Exception : Exception in thread "main" org.springframework.beans.factory.BeanIsAbstractException:
Error creating bean with name 'parentPersonBean': Bean definition is abstract
```

Pure Inheritance Template

Actually, parent bean is not necessary to define class attribute, often times, you may just need a common property for sharing. Here's is an example

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                              "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
        <bean id="parentPersonBean"  abstract="true">
                <property name="city" value="Hyd" />
                <property name="country" value="India" />
        </bean>
        <bean id="childPersonBean1" parent="parentPersonBean"
class="com.nareshit.bean.Person">
                <property name="name" value="Srinu" />
                <property name="phone" value="8989898988" />
                <property name="email" value="Srinu@nareshit.in" />
        </bean>
        <bean id="childPersonBean2" parent="parentPersonBean"
class="com.nareshit.bean.Person">
                <property name="name" value="Hari" />
                <property name="phone" value="7875444333" />
                <property name="email" value="hari@nareshit.in" />
        </bean>

</beans>
```

In this case, the 'parentPersonBean' bean is a pure template, to share its 'city and country' properties only.

Overrride it

However, you are still allow to override the inherited value by specify the new value in the child bean. Let's see this example

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                    "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
<bean id="parentPersonBean"  class="com.nareshit.bean.Person"    abstract="true">
                <property name="city" value="Hyd" />
                <property name="country" value="India" />
        </bean>
        <bean id="childPersonBean1" parent="parentPersonBean" >
                <property name="name" value="Srinu" />
                <property name="phone" value="8989898988" />
                <property name="email" value="Srinu@nareshit.in" />
                <property name="city" value="Chennai"/>
        </bean>
        <bean id="childPersonBean2" parent="parentPersonBean"
class="com.nareshit.bean.Person">
                <property name="name" value="Hari" />
                <property name="phone" value="7875444333" />
                <property name="email" value="hari@nareshit.in" />
        </bean>

</beans>
```

Conclusion

The Spring bean configuration inheritance is very useful to avoid the repeated common value or configurations for multiple beans.

<u>What is collection merging?</u>

- In spring 2.0 the container supports the collection merging. In this your parent bean can declare a list as parent list.In your child beans you can declare a list with values ,you can inherit the values of your parent list values into your child bean list values .As the list merged with parent list values,this is called Collection merging.

  <u>Course.java</u>

  ```
  public class Course{
  private List<String> subjects;
  ```

```
        //setter and getter
        }
```

myBeans.xml

```
<beans>

<bean id="parentCourse" class="Course" abstract="true">

    <property name="subjects">

    <list>

   <value>c</value>

    <value>java</value>

   </list></property>
<bean id="childCourse" class="Course"  parent="parentCourse">

        <property name="subjects">

        <list merge="true">

        <value>DS</value>

        </list>

        </property>

</bean>

</beans>
```

Bean Aliasing

In Spring when you configure a class as bean you will declare an id with which you want to retrieve it back from the container. Along with id you can attach multiple names to beans, and these names act as alias names with which you can look up the bean from the container. To declare multiple names You need to declare an "name" attribute at the bean tag level . We can separate multiple names with comma(OR) with space.

Following Code showing Alias Names for Student bean

```
<bean id="student1" name="student2,student3,student4" class="com.nareshit.bean.Student">

</bean>
```

You can retrieve the above bean with either student1 (OR) student2(OR) student3 names. You can even get all the names of the bean using factory.getAliases("oneName");

Id attribute is holding actual name .

name attribute holding alias names.

In general bean aliasing is used for ease maintenance of the configuration.

In Spring 2.0 a new tag has been introduced <alias> using which you can declare multiple names for the bean. The Syntax is as follows

P-Namespace and C-Namespace:

- If we want to perform setter injection on a spring bean we need to use <property> tag.Instead of writing length<property> tag declaration under the <bean> tag we can replace with short form of representing the same with p-namespace.
- In order to use the p-namespace,you first need to import the "htttp://www.springframework.org/schema/p" Name space in the spring bean cfg file.

Consider following example:

Student.java

```
package com.nareshit.bean;

public class Student {
   private String name;
   private int age;
   private Course course;

  //required  setters and getters

   public void displayInfo(){

      System.out.println("Student Name : "+name);
      System.out.println("Student Age : "+ age);
      System.out.println("Course name : "+course.getCourseName());
   }
}
```

Course.java

```
package com.nareshit.bean;
public class Course {

private    String courseName;
   //required setters and getters
}
```

Test.java

```
package com.nareshit.client;

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

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

   ApplicationContext  context = new
ClassPathXmlApplicationContext("com/nareshit/config/myBeans.xml");
   Student stud = (Student)context.getBean("student");
   stud.displayInfo();
 }
}
```

myBeans.xml

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

<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:p="http://www.springframework.org/schema/p"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="student" class="com.nareshit.bean.Student">
<property name="name" value="sathish"/>
<property name="age" value="28"/>
<property name="course" ref="course"/>
</bean>

<bean id="course" class="com.nareshit.bean.Course">
<property name="courseName" value="java"/>
</bean>
</beans>
```

In the above program dependencies of Student and Course class have been injected using <property> tags.

Now the same program using p namespace is (you only need to change myBeans.xml, other classes remain same):

myBeans.xml

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

<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:p="http://www.springframework.org/schema/p"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="student" class="com.nareshit.bean.Student"  p:name="sathish" p:age="28" p:course-
ref="course"/>
<bean id="course" class="com.nareshit.bean.Course"  p:courseName="java"/>
</beans>
```

In the above program we are using p namespace to inject the dependencies of Student and Course classes.

For example,

p:age="28" is substitute of:
<property name="age" value="28">

Note how we are injecting reference to other bean as a value of property -**course** of Student class, using p namespace:
p:course-ref="course"
is a substitute of
<property name="course" ref="course">

Note the **-ref** at the end of p:course-ref="course". **-ref** means this property would accept a reference to other bean as its value.

Also note that unlike <property> tags, p namespace are not separate tags but they are the part of <bean> tag

For p namepsace to work you have to add its schema URI
http://www.springframework.org/schema/p in above xml file.

C-Name Space :

C-Namespace has been introduced in spring 3.1.1, in order to perform constructor injection we need to use<constructor-arg> tag.instead of writing the length <constructor-arg> tag,we can replace it with c:namespace.

Consider following example:

Student.java
```
package com.nareshit.bean;

public class Student {
   private String name;
   private int age;
   private Course course;

   //constructor
   Student(String name, int age, Course course){
      this.name = name;
      this.age = age;
      this.course = course;
   }

// setters and getters...

   public void displayInfo(){
      System.out.println("Student Name : "+name);
      System.out.println("Student Age : "+ age);
      System.out.println("Course name : "+course.getCourseName());
   }
}
```

Course.java
```
package com.nareshit.bean;

public class Course {
pivate    String courseName;
   public  Course(String course){
      this.courseName = course;
```

```
  }
   public String getCourseName(){
     return courseName;

  }

}
```

Test.java

```
package com.nareshit.client;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test{
public static void main(String args[]){
 ApplicationContext   context = new
ClassPathXmlApplicationContext("com/nareshit/config/myBeans.xml");
   Student stud = (Student)context.getBean("student");
   stud.displayInfo();
 }
}
```

myBeans.xml

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

<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
<bean id="student" class=" com.nareshit.bean.Student ">
<constructor-arg value="sathish"/>
<constructor-arg value="28"/>
<constructor-arg ref="course"/>
</bean>

<bean id="course" class="com.nareshi.bean.Course">
<constructor-arg value="java"/>
</bean>
</beans>
```

In the above program arguments of Student and Course class constructors have been injected using <constructor-arg> tags.

Now the same program using c namespace is (you only need to change myBeans.xml, other classes remain same):

myBeans.xml

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

```
<bean id="student" class="com.nareshit.bean.Student" c:name="sathish" c:age="28" c:course-ref="course"/>

<bean id="course" class="com.nareshit.bean.Course" c:course="java"/>
</beans>
```

In the above program we are using c namespace to inject the argument values of Student and Course class constructors. For example,

c:age="28" is substitute of:
<constructor-arg value="28"/>

Note how we are injecting reference to other bean as a value for the **'c'** argument of Student class constructor, using c namespace:

c:course-ref="course"
is a substitute of
<constructor-arg ref="course"/>

Note the **-ref** at the end of **c:course-ref="course"**. **-ref** means this property would accept a reference to other bean as its value.

Unlike <constructor-arg> tags, c namespace are not separate tags but they are the part of <bean> tag

For c namespace to work you have to add its schema URI http://www.springframework.org/schema/c in above xml file.

In case when we do not know the names of constructor arguments we can use following syntax in **myBeans.xml:**

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

<bean id="student" class="com.nareshit.bean.Student" c:_0="abc" c:_1="28" c:_2-ref="course"/>

<bean id="course" class="com.nareshit.bean.Course" c:_0="java"/>
</beans>
```

Here c:_0 means first argument of constructor, c:_1 means second argument and c:_2-ref means third argument which is accepting a reference to other bean as its value.

External Properties Files :

When configuring beans in the configuration file, you must remember that it's not a good practice to mix deployment details, such as the file path, server address, username, and password, with your bean configurations. Usually, the bean configurations are written by application developers while the deployment details are matters for the deployers or system administrators.

Spring's PropertyPlaceholderConfigurer is used for externalizing properties from the Spring bean definitions defined in XML or using Java Config.

PropertyPlaceholderConfigurer resolves ${key} placeholders against `local properties` and/or system properties and environment variables.

You can use location (OR) locations property of the PropertyPlaceholderConfigurer to specify the properties file path.

Example :

Create a properties file (database.properties), include your database details, put it into your project class path.

```
oracle.driverClassName=oracle.jdbc.driver.OracleDriver

oracle.url=jdbc:oracle:thin:@localhost:1521:XE

oracle.username=system

oracle.password=manager
```

Now Declare a **PropertyPlaceholderConfigurer** in bean configuration file and map to the 'database.properties' properties file you created just now.

Example :

```
<beans ...>
...
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
<property name="location">
<value>database.properties</value>
</property>
</bean>
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="${oracle.driverClassName}" />

<property name="url" value="${oracle.url}" />

<property name="username" value="${oracle.username}" />

<property name="password" value="${oracle.password}" />

</bean>

</beans>
```

Prior to Spring 3.1 the registration of PropertyPlaceholderConfigurer can be simply through the <context:property-placeholder> element.

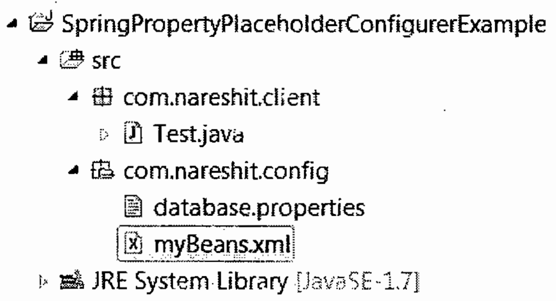
Example :

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:property-placeholder location="database.properties" />
...

</beans>
```

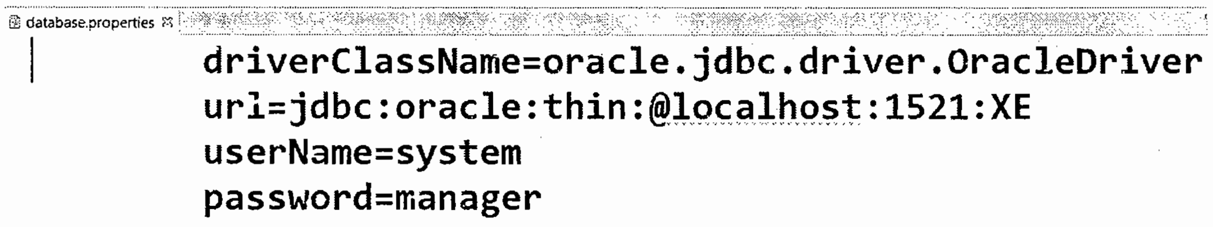
Note :

Bean Factory container can't work with properties files. Only Application Context container work with properties files.

Example :-



```
SpringPropertyPlaceholderConfigurerExample
  src
    com.nareshit.client
      Test.java
    com.nareshit.config
      database.properties
      myBeans.xml
  JRE System Library [JavaSE-1.7]
```



database.properties

```
driverClassName=oracle.jdbc.driver.OracleDriver
url=jdbc:oracle:thin:@localhost:1521:XE
userName=system
password=manager
```

myBeans.xml

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"

"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

<bean id="dataSource"     class="org.springframework.jdbc.datasource.DriverManagerDataSource">
```

```
            <property name="driverClassName" value="${driverClassName}" />

            <property name="url" value="${url}" />

            <property name="username" value="${userName}" />

            <property name="password" value="${password}" />

        </bean>

        <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

            <property name="location">

                <value>com/nareshit/config/database.properties

                </value>

            </property>

        </bean>

</beans>
```

Test.java

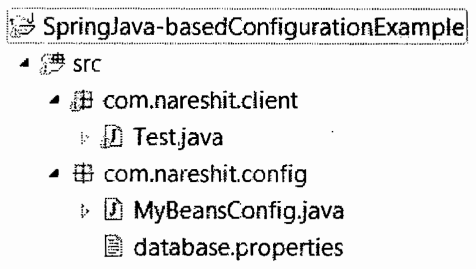
```
package com.nareshit.client;

import java.sql.Connection;

import java.sql.DatabaseMetaData;

import java.sql.SQLException;

import javax.sql.DataSource;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

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

        ApplicationContext context=new

                        ClassPathXmlApplicationContext("com/nareshit/config/myBeans.xml");

   DataSource ds=(DataSource)context.getBean("dataSource");

   Connection con=ds.getConnection();

   DatabaseMetaData dbmd=con.getMetaData();

   System.out.println(dbmd.getDatabaseProductVersion());

}

}
```

working with @Configuration and @Bean :-

Instead of declaring a class as spring bean in a configuration file, you can declare it in a class. The class in which you want to provide the configuration about other beans, that class is called configuration class and you need to annotate with @Configuration.

In this class you need to provide the methods which are responsible for creating object's of your bean classes, these methods has to be annotated with @Bean.

Example :-

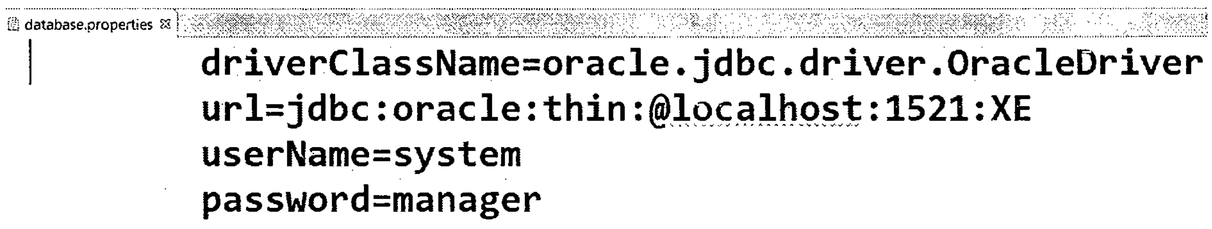


```
SpringJava-basedConfigurationExample
  src
    com.nareshit.client
      Test.java
    com.nareshit.config
      MyBeansConfig.java
      database.properties
```

MyBeansConfig.java

```
package com.nareshit.config;

import javax.sql.DataSource;

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

import org.springframework.beans.factory.config.PropertyPlaceholderConfigurer;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.core.io.ClassPathResource;

import org.springframework.core.io.Resource;

import org.springframework.jdbc.datasource.DriverManagerDataSource;

        @Configuration

        public class MyBeansConfig{

                @Value("${driverClassName}")

                private String driverClassName;

                @Value("${url}")

                private String url;

                @Value("${userName}")

                private String userName;

                @Value("${password}")

                private String password;

                @Bean(name="dataSource")
        public DataSource dataSource(){

                DriverManagerDataSource

                dataSource=new DriverManagerDataSource();

                dataSource.setDriverClassName(driverClassName);

                dataSource.setUrl(url);

                dataSource.setUsername(userName);

                dataSource.setPassword(password);

                return dataSource;

        }

        @Bean

        public static PropertyPlaceholderConfigurer placeholderConfigurer(){

        PropertyPlaceholderConfigurer      placeholderConfigurer=

       new      PropertyPlaceholderConfigurer();

        Resource resource=

        new ClassPathResource("com/nareshit/config/database.properties");

        placeholderConfigurer.setLocation(resource);

        return placeholderConfigurer;

        }

    }
```

database.properties



database.properties

```
driverClassName=oracle.jdbc.driver.OracleDriver
url=jdbc:oracle:thin:@localhost:1521:XE
userName=system
password=manager
```

Test.java

```
package com.nareshit.client;

import java.sql.Connection;

import java.sql.DatabaseMetaData;

import java.sql.SQLException;

import javax.sql.DataSource;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.nareshit.config.MyBeansConfig;

public class Test {

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

                ApplicationContext context = new AnnotationConfigApplicationContext(

                                MyBeansConfig.class);

                DataSource ds = (DataSource) context.getBean("dataSource");

                Connection con = ds.getConnection();

                DatabaseMetaData dbmd = con.getMetaData();

                System.out.println(dbmd.getDatabaseProductVersion());

        }

}
```

Spring JDBC

In Spring Core, we had the understanding on using Spring and the use of DI (Dependency Injection).
In this module, we will see how to deal with Databases using Spring framework.
Spring comes with a family of data access frameworks that integrate with a variety of data access technologies. You may use direct JDBC, iBATIS, or an object relational mapping (ORM) framework like Hibernate to persist your data. Spring supports all of these persistence mechanisms.

Problems of Plain Jdbc API :-

Problems of plain JDBC API as follows

1) Code Duplication : We need to write a lot of code before and after executing the query, such as creating connection, statement, etc.... is a major problem while using Jdbc API directly to access database, writing boilerplate code Over again is clear violation of the basic Object-Oriented (OO) principle of code resuse.This has some Side effects in terms of the Project cost,timelines and effort.

2) Resource Leakage problems

3)Exception-Handling problems

To Solve the above Problems The Springframework provides a solution by giving a robust and highly extensible Jdbc Abstraction F/w.

Spring Jdbc Approaches:-

JDBC and DAO module of Spring is used to build the data access layer of a Spring project

There are Two approaches in Spring JDBC f/w to develop the persistence logic.
1) Template Approach
2) Fine-Grained Object-oriented approach. (RdbmsOperation classes approach)

Note :- In computer programming, boilerplate code or boilerplate is the sections of code that have to be included in many places .
i.e. the programmer must write a lot of code to do minimal jobs
Template classes can help us in writing modular code with ease, without worrying about whether resources are closed properly or not.

DAO:-

It is the Design the Pattern which separates Persistence Logic from Business Logic and provides code reusability on Persistence logic.
DAO classes can be developed with JDBC API,Hibernate,Ibatis Spring/JDBC etc....
Note :
Spring f/w provides the Declarative Exception-Handling mechanism to deal with database,

DataAccessException:-

org.springframework.dao.DataAccessException is the top most Exception in springJDBC and Spring ORM exception -hierarchy. DataAccessException is the child class of
Runtime Exception so all the SpringJdbc and ORM Exceptions are Unchecked Exceptions

JdbcTemplate :

The JdbcTemplate class is the central class in org.springframework.jdbc.core package .

It handles the creation and release of resources, which helps you avoid common errors such as forgetting to close the connection.

It performs the basic tasks of the core JDBC workflow such as statement creation and execution, leaving application code to provide SQL and extract results.

The JdbcTemplate class executes all types of SQL statements and stored procedure calls, performs iteration over ResultSets and extraction of returned parameter values.

It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the org.springframework.dao package.

It is important to know that the JdbcTemplate class instances are threadsafe once configured. Meaning you can configure a single instance of a JdbcTeplate and then safely inject this shared reference into Multiple DAO's.

The JdbcTemplate class is non-abstract class and can be instantiated using any of the following three constructors.

JdbcTemplate() : constructs a new JdbcTemplate object.This constructor

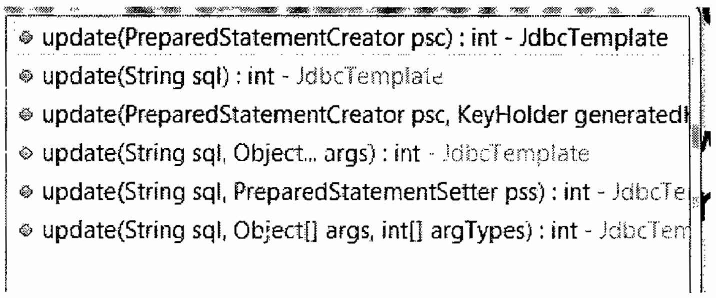
Is provided to allow Java Bean style of instantiation.

Note : when **constructing an Object using this constructor we need to use setDataSource() method to set the dataSource before using this object to execute the statements.**

JdbcTemplate(DataSource) : Constructs a new JdbcTemplate Object initializing it with the given dataSource to obtain connections for executing the requested statements.

JdbcTemplate(DataSource , boolean) : Constructs a new JdbcTemplate Object initializing it with the given dataSource to object connections for executing the requested statements,and the Boolean value describing the lazy initialization of the SQL Exception translator. If the Boolean value is true then the exception translator will not be initialized immediately.Instead it waits until this JdbcTemplate object is used to execute the statement. If the Boolean argument value is false then the exception translator is initialized when constructing this object. Using the JdbcTemplate(DataSource) constructor is the same as using the JdbcTemplate(dataSource,Boolean) constructor with the second argument as true. That means when a single argument constructor of JdbcTemplate is used the exception translator is not immediately initialized.

JdbcTemplate is providing below overloaded **update()**methods to implement DML statements.


update(PreparedStatementCreator psc) : int - JdbcTemplate
update(String sql) : int - JdbcTemplate
update(PreparedStatementCreator psc, KeyHolder generated
update(String sql, Object... args) : int - JdbcTemplate
update(String sql, PreparedStatementSetter pss) : int - JdbcTe
update(String sql, Object[] args, int[] argTypes) : int - JdbcTem

public int update(String sql):-

-->When the DML query is not having any placeholders then we will go for this method.
EX:- String sql="DELETE FROM Employee WHERE empno=1001";
Usage: jdbcTemplate.update(sql);

public int update(String sql,Object...args):-

-->When the DML query is having placeholders we will use this method.This method supports varargs feature of java.
Ex:- String sql="insert into Employee values(?,?,?)";
Usage:jdbcTemplate.update(sql,emp.getEmpNo(),emp.getName(),emp.getSalary());

public int update(String sql,Object[] args,int[] arqTypes):

-->When the query is having place holders & if we want to specify place holder types then we will use this method.
Ex:- String sql="insert into Employee values(?,?,?)";

Usage: jdbcTemplate.update(sql,new Object[]{emp.getEmpNo(),emp.getName(),emp.getSalary()},new int[]{Types.INTEGER ,Types.VARCHAR,Types.DOUBLE});

public int update(String sql,PreparedStatementSetter pss)

When the DML query is having placeholders we will use this method.

This method uses the given PreparedStatementSetter Object to set the bind

parameters.

The PreparedStatementSetter is a helper that sets the parameters of the PreparedStatement, PreparedStatementSetter interface contain only one method named as setValues, that takes PreparedStatement as an argument.

public void setValues(PreparedStatement pst)throws SQLException

Example :

EmployeeDAOImpl.java

@Repository("employeeDao")

public class EmployeeDAOImpl implements EmployeeDAO{

@Autowired

```
private JdbcTemplate jdbcTemplate;

public int createEmployee(Employee emp){

String sql="insert into Employee values(?,?,?)";

MyPreparedStatementSetter mpss=new MyPreparedStatementSetter(emp);

int count=jdbcTemplate.update(sql,mpss);

return count;

}

}
```

MyPreparedStatementSetter.java

```
public class MyPreparedStatementSetter implements PreparedStatementSetter{

Employee emp;

public MyPreparedStatement(Employee emp){

this.emp=emp;

}

public void setValues(PreparedStatement pst)throws SQLException{

pst.setInt(1,emp.getEid());

pst.setString(2,emp.getName());

pst.setDouble(3,emp.getSalary());

}

}
```

public int update(PreparedStatementCreator psc)

This method executes the DML/DDL Sql query

by using PreparedStatement object, The PreparedStatement is obtained by using the given PreparedStatementCreator.

The PreparedStatementCreator is an interface present in org.springfamework.jdbc.core package

PreparedStatementCreator interface contain only one method i,e.

public PreparedStatement createPreparedStatement(Connection connection)

Example :

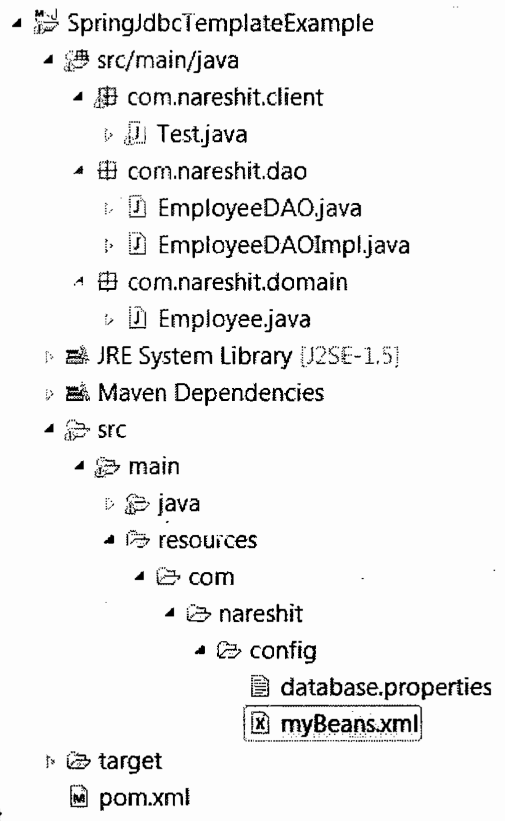
EmployeeDAOImp.java

```
@Repository("employeeDao")
```

```
public  class EmployeeDAOImpl implements EmployeeDAO{

@Autowired

private JdbcTemplate jdbcTemplate;

public void createEmployee(Employee emp){

MyPreparedStatementCreator mpsc=new MyPreparedStatementCreator(emp);

int count=jdbcTemplate.update(mpsc);

return count;

}

}
```

<u>MyPreparedStatementCreator.java</u>

```
public class MyPreparedStatementCreator implements PreparedStatementCreator{

private Employee emp;

public MyPreparedStatementCreator(Employee emp){

this.emp=emp;

}

public PreparedStatement createPreparedStatement (Connection con)throws SQLException{

PreparedStatement pst=con.prepareStatement("insert into Employee values(?,?,?)");

pst.setInt(1,emp.getEid());

pst.setString(2,emp.getName());

pst.setDouble(3,emp.getSalary());

return pst;

}
```



```
SpringJdbcTemplateExample
  src/main/java
    com.nareshit.client
      Test.java
    com.nareshit.dao
      EmployeeDAO.java
      EmployeeDAOImpl.java
    com.nareshit.domain
      Employee.java
  JRE System Library [J2SE-1.5]
  Maven Dependencies
  src
    main
      java
      resources
        com
          nareshit
            config
              database.properties
              myBeans.xml
  target
  pom.xml
```
}

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
 <groupId>com.nareshit</groupId>
  <artifactId>SpringJdbcTemplateExample</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>SpringJdbcTemplateExample</name>
  <url>http://maven.apache.org</url>
  <dependencies>
   <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>4.0.2.RELEASE</version>
   </dependency>
   <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>4.0.2.RELEASE</version>
   </dependency>
   <dependency>
    <groupId>oracle.jdbc.driver</groupId>
    <artifactId>ojdbc14</artifactId>
    <version>10.2.0</version>
   </dependency>
 </dependencies>
</project>
```

myBeans.xml

```
 <beans xmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <context:component-scan base-package="com.nareshit.dao"/>
    <context:property-placeholder
    location="classpath:/com/nareshit/config/database.properties"/>

    <!--
    (OR)
    <bean id="propertyPlaceholderConfigurer"

class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
    <list>
    <value>classpath:/com/nareshit/config/database.properties</value>
    </list>
    </property>
    </bean> -->
    <bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="${oracle.driver}"/>
    <property name="url"  value="${oracle.url}"/>
    <property name="username" value="${oracle.username}"/>
    <property name="password" value="${oracle.password}"/>
    </bean>
    <bean id="jdbcTemplate"
    class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"/>
    </bean>
     <bean id="employeeDao"
    class="com.nareshit.dao.EmployeeDAOImpl">
    <property name="jdbcTemplate" ref="jdbcTemplate"/>
    </bean>
    </beans>
```

database.properties

```
  oracle.driver=oracle.jdbc.driver.OracleDriver
   oracle.url=jdbc:oracle:thin:@localhost:1521:XE
   oracle.username=system
   oracle.password=manager
```

EmployeeDAO.java

```
package com.nareshit.dao;
import com.nareshit.domain.Employee;
public interface EmployeeDAO {
public int createEmployee(Employee employee);
public int deleteEmployee(int empNo);
public int updateEmployeeName(int empNo,String name);
}
```

EmployeeDAOImpl.java

```
package com.nareshit.dao;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementCreator;
import org.springframework.jdbc.core.PreparedStatementSetter;
import com.nareshit.domain.Employee;
public class EmployeeDAOImpl implements EmployeeDAO {
private JdbcTemplate jdbcTemplate;
public int createEmployee(Employee emp){
        String sql="insert into emp values(?,?,?)";
        int count=
    jdbcTemplate.update(sql,emp.getEmpNo(),emp.getName(),emp.getSalary());
        return count;
}
public int deleteEmployee(final int empNo) {
        String sql="delete from emp where empno=?";
        int count=jdbcTemplate.update(sql,
                        new PreparedStatementSetter(){
public void setValues(PreparedStatement ps)throws SQLException{
        ps.setInt(1,empNo);
    }
        });
        return count;
}
public int updateEmployeeName(final int empNo, final String name) {
        int count=jdbcTemplate.update(
                        new PreparedStatementCreator(){
        public PreparedStatement
        createPreparedStatement(Connection connection)throws SQLException{
        String sql="update emp set name=? where empno=?";
                PreparedStatement pst=connection.prepareStatement(sql);
                pst.setString(1,name);
                pst.setInt(2, empNo);
                return pst;
        }
        });
        return count;
}
}
```

Employee.java

```
package com.nareshit.domain;
public class Employee {
private int empNo;
private String name;
private double salary;
public int getEmpNo() {
        return empNo;
}
public void setEmpNo(int empNo) {
        this.empNo = empNo;
}
public String getName() {
        return name;
}
public void setName(String name) {
        this.name = name;
}
public double getSalary() {
        return salary;
}
public void setSalary(double salary) {
        this.salary = salary;
}
}
```

Test.java

```
package com.nareshit.client;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.nareshit.dao.EmployeeDAO;
import com.nareshit.domain.Employee;
public class Test{
public static void main( String[] args ){
 ApplicationContext context=new ClassPathXmlApplicationContext("com/nareshit/config/myBeans.xml");
 EmployeeDAO empDao=context.getBean("employeeDao",EmployeeDAO.class);
 System.out.println("Testing of updateEmployeeName(-,-)");
 int updateCount= empDao.updateEmployeeName(3001,"rama");
 System.out.println("updateCount :"+updateCount);
  System.out.println("Testing of deleteEmployee(-) ");
     int deleteCount= empDao.deleteEmployee(3003);
      System.out.println("deleteCount :"+deleteCount);

      System.out.println("Testing of createEmployee(-)");
       Employee emp=new Employee();
       emp.setEmpNo(3004);
       emp.setName("raju");
       emp.setSalary(29000.0);
       int insertCount=empDao.createEmployee(emp);
      System.out.println("InsertCount ::"+insertCount);
   }
}
```

Dao Supported classes:-

For every template class an associated Dao Support class given by Spring framework
In these classes corressponding template is declared as instance variables and associated setter and getter methods are defined.
JdbcTemplate class is associated with JdbcDaoSupport class this class is given by Spring framework

JdbcDaoSupport class Source Code :

```
public abstract class JdbcDaoSupport extends DaoSupport {

        private JdbcTemplate jdbcTemplate;

        public final void setDataSource(DataSource dataSource) {
        if (this.jdbcTemplate == null ||
    dataSource != this.jdbcTemplate.getDataSource()) {
        this.jdbcTemplate = createJdbcTemplate(dataSource);
        initTemplateConfig();
                }
        }


        protected JdbcTemplate createJdbcTemplate(DataSource dataSource) {
                return new JdbcTemplate(dataSource);
        }
          public final void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
                this.jdbcTemplate = jdbcTemplate;
                initTemplateConfig();
        }
    public final JdbcTemplate getJdbcTemplate() {
        return this.jdbcTemplate;
        }
}
```

To Work with DaoSupport classes Take an user-defined Dao class as a Child class to XxxDaoSupport class , set the DataSource object and to get the corresponding Xxxtemplate without explicitly declaraing the template as dependency we can call getXxxTemplate() method.

__Example :__

__EmployeeDAO.java__

```
public class EmployeeDAO extends JdbcDaoSupport {
                public int createEmployee(Employee emp){
                      String sql="insert into employee values(?,?,?)";
                int
count=getJdbcTemplate().update(sql,emp.getEmpNo(),emp.getName(),emp.getSalary());
                return count;
                }
        }
```

__myBeans.xml__

```
<beans
xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=
"http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

                <bean id="dataSource"
     class="org.springframework.jdbc.datasource.DriverManagerDataSource">
          <property name="driverClassName"
          value="oracle.jdbc.driver.OracleDriver"/>
          <property name="url"
```

```
        value="jdbc:oracle:thin:@localhost:1521:XE">
        </property>
        <property name="username"
        value="system"/>
        <property name="password" value="manager"/>
        </bean>
       <bean id="employeeDao"
       class="com.nareshit.dao.EmployeeDAO">
       <property name="dataSource" ref="dataSource"></property>
       </bean>
       </beans>
```

queryForXxx() methods :-
JdbcTemplate provides several convenience queryForXxx() methods to execute the select statements .

queryForInt(-) (in Spring 3.2.2 Deprecated)
public int queryForInt(String sql)
This method executes the select query and gives the result as an int value.
The Result of the Query is passed method is expected to be return a single row and Single coloum
If the Query does not returning exactly one row. It throws
"org.springframework.dao.IncoreectResultSizeDataAccessException"
And if the query does not returning exactly one column it throws
"org.springframework.jdbc.IncorrectResultSetColumnCountException"

public int queryForInt(String sql,Object[] params)
This method is similar to queryForInt(String sql)
but this method Have second argument that is used to set the bind parametes.
public int queryForInt(String sql,Object[] args,int[] Types)
this method is same as above method but this facilitates to describe the parameter types in addition.

queryForLong(-) in Spring 3.2.2 Deprecated
public long queryForLong(String sql)
This method executes the select query and gives the result as a long value.
The Result of the Query is passed method is expected to be return a single row and Single coloum
public long queryForLong(String sql,Object[] args)
This method is similar to queryForLong(String sql)
but this method Have second argument that is used to set the bind parametes.
public long queryForLong(String sql,Object[] args,int[] Types)
this method is same as above method but this facilitates to describe the parameter types in addition.
queryForObject(-) :
public <T> T queryForObject(String sql,Class<T> requiredType, Object... params)throws DataAccessException

public <T> T queryForObject(String sql,Object[] params,Class<T> requiredType)throws DataAccessException

This method executes the given select query and the returned result will be directly mapped to the corresponding object type . The query is expected to be a single row/single column query;
params is used to set the bind parametes.
public <T> T queryForObject(String sql,Object[] params,int[] types,Class<T> requiredType)throws DataAccessException
this method is same as above method but this facilitates to describe the parameter types in addition.

queryForMap(-):
public Map<String,Object> queryForMap(String sql):

This method executes the given select query and the sql Query should result in a single row with one (OR) more column of any type.
The method returns a Map Object with one entry for each column with column name as a key and it's values as an entry value.

public Map<String,Object> queryForMap(String sql,Object... params):

This method is similar to queryForMap(String sql)
but this method Have second argument that is used to set the bind parametes.

public Map<String,Object> queryForMap(String sql,Object[] params,int[] types):

this method is same as above method but this facilitates to describe the parameter types in addition.

queryForList(-):

public List<Map<String,Object>> queryForList(String sql)

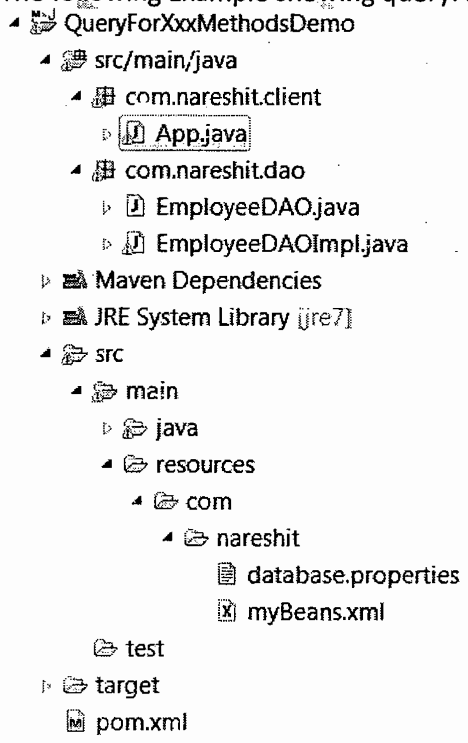
This method executes the given select query and the sql Query should result in one (OR) more rows with one (OR) more column of any type.
The method returns a List of Map Objects. One Map object is created for each row ,one entry for each column with column names as a key and it's value as an entry value.

public List<Map<String,Object>> queryForList(String sql,Object... params):

This method is similar to queryForList(String sql)
but this method Have second argument that is used to set the bind parametes.

public List<Map<String,Object>> queryForList(String sql,Object[] params,int[] types):

this method is same as above method but this facilitates to describe the parameter types in addition.
The following Example showing queryForXxx() methods usage.



```
QueryForXxxMethodsDemo
  src/main/java
    com.nareshit.client
      App.java
    com.nareshit.dao
      EmployeeDAO.java
      EmployeeDAOImpl.java
  Maven Dependencies
  JRE System Library [jre7]
  src
    main
      java
      resources
        com
          nareshit
            database.properties
            myBeans.xml
    test
  target
  pom.xml
```

myBeans.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">
 <context:component-scan base-package="com.nareshit.dao" />
 <context:property-placeholder location="classpath:/com/nareshit/database.properties" />
```

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName" value="${oracle.driver}" />
<property name="url" value="${oracle.url}"/>
<property name="username" value="${oracle.username}" />
<property name="password" value="${oracle.password}" />
</bean>
</beans>
```

database.properties

database.properties

```
oracle.driver=oracle.jdbc.driver.OracleDriver
oracle.url=jdbc:oracle:thin:@localhost:1521:XE
oracle.username=system
oracle.password=manager
```

EmployeeDAO.java

```
package com.nareshit.dao;
import java.util.List;
import java.util.Map;
public interface EmployeeDAO{
        int getDeptNo(int empno);
        String getName(int empno);
        double getSalary(int empno);
        Map<String,Object> getEmployee(int empno);
    List<Map<String,Object>> getAllEmployees();
 }
```

EmployeeDAOImpl.java

```
package com.nareshit.dao;
import java.util.List;
import java.util.Map;
import javax.sql.DataSource;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.support.JdbcDaoSupport;
import org.springframework.stereotype.Repository;
@Repository("employeeDao")
public class EmployeeDAOImpl extends JdbcDaoSupport implements EmployeeDAO {
        @Autowired
        public EmployeeDAOImpl(DataSource dataSource){
                setDataSource(dataSource);
        }
        public int getDeptNo(int empno){
                return getJdbcTemplate().queryForInt("select deptno from emp  where 
empno="+empno);
        }
        public String getName(int empno){
return getJdbcTemplate().queryForObject("select name from emp where empno="+empno,String.class);
        }
        public double getSalary(int empno){
return getJdbcTemplate().queryForObject("select salary from emp where empno=?",Double.class,empno);
        }
        public Map<String,Object> getEmployee(int empno){
```

```
return getJdbcTemplate().queryForMap("select name,deptno,salary from emp where empno=?",empno);
        }
        public List<Map<String,Object>> getAllEmployees(){
return getJdbcTemplate().queryForList("select empno,name,deptno,salary from emp");
        }
}
```

App.java

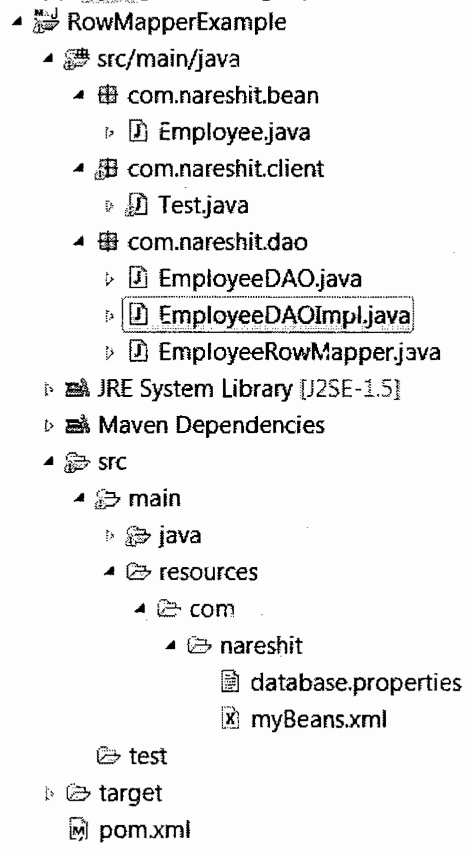
```
package com.nareshit.client;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.nareshit.dao.EmployeeDAO;
public class App {
public static void main(String[] args) {
ApplicationContext context = new ClassPathXmlApplicationContext("com/nareshit/myBeans.xml");
EmployeeDAO employeeDao = context.getBean("employeeDao",EmployeeDAO.class);
System.out.println("Testing of getAllEmployees() ");
List<Map<String, Object>> list = employeeDao.getAllEmployees();
for (Map<String, Object> map1 : list) {
Set<String> keys = map1.keySet();
for (String key : keys) {
Object value = map1.get(key);
System.out.println(key + "=" + value);
}
System.out.println("-----------");
  }
System.out.println("-------------------");
System.out.println("Testing of getEmployee(1001) ");
Map<String, Object> map2 = employeeDao.getEmployee(1001);
Set<Entry<String, Object>> set = map2.entrySet();
for (Entry<String, Object> entry : set) {
System.out.println(entry.getKey() + "=" + entry.getValue());
}
System.out.println("-------------------");
System.out.println("Testing of getName(1002) ");
System.out.println(employeeDao.getName(1002));
System.out.println("-------------------");
System.out.println("Testing of getDeptNo(1002) ");
System.out.println(employeeDao.getDeptNo(1002));
System.out.println("-------------------");
System.out.println("Testing of getSalary(1002) ");
System.out.println(employeeDao.getSalary(1002));
                }
}
```

query() methods:-

JdbcTemplate class includes query() methods to prepare and execute the SQL queries . The Spring Jdbc abstraction framework provides three different types of callbacks to read the results after executing the

SQL query using query() methods. The three callbacks of Spring Jdbc abstraction framework to retrieve the data are:
RowMapper
ResultSetExtractor
RowCallbackHandler

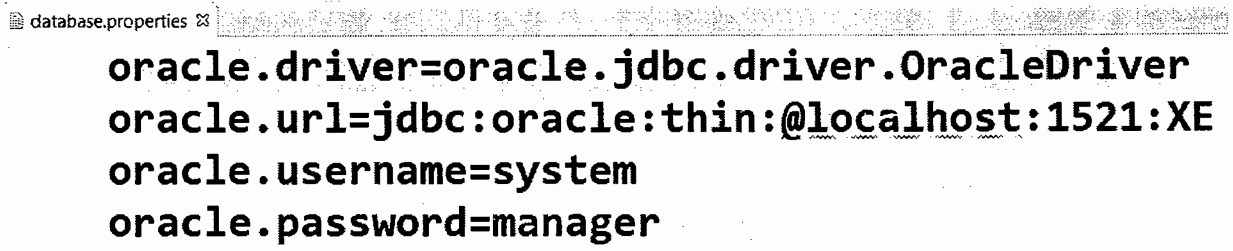
RowMapper :-
public List<T> query(String sql,RowMapper rm)
public List<T> query(String sql,Object[] params,RowMapper rm)
public List<T> query(String sql,RowMapper rm,Object... params)
RowMapper is a callback interface present in org.springframework.jdbc.core
and it has one method
public object mapRow(ResultSet rs,int rowNum)
The implementation classes of this interface needs to implement the mapRow() method,which takes the responsibility to process each row of data in the ResultSet. The mapRow() method should not call next() method on ResultSet.It requireds to extract values of the current row,process the data on per-row basis and map the current row values.
The mapRow() method implementations does not need to worry about handling the SQLException raised while reading the results.The SQLException will be caught and handled by the JdbcTemplate .
The mapRow() method can return an Object representing the result object for the current Row.
RowMapper objects are typically stateless and thus reusable; they are an ideal choice for implementing row-mapping logic in a single place.


```
RowMapperExample
  src/main/java
    com.nareshit.bean
      Employee.java
    com.nareshit.client
      Test.java
    com.nareshit.dao
      EmployeeDAO.java
      EmployeeDAOImpl.java
      EmployeeRowMapper.java
  JRE System Library [J2SE-1.5]
  Maven Dependencies
  src
    main
      java
      resources
        com
          nareshit
            database.properties
            myBeans.xml
    test
  target
  pom.xml
```

myBeans.xml
```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
```

```
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">
        <context:component-scan base-package="com.nareshit.dao" />
        <context:property-placeholder
                location="classpath:/com/nareshit/database.properties" />
        <bean id="dataSource"
                class="org.springframework.jdbc.datasource.DriverManagerDataSource">
                <property name="driverClassName" value="${oracle.driver}" />
                <property name="url" value="${oracle.url}" />
                <property name="username" value="${oracle.username}" />
                <property name="password" value="${oracle.password}" />
        </bean>
        <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
                <property name="dataSource" ref="dataSource" />
        </bean>
</beans>
```

database.properties



```
oracle.driver=oracle.jdbc.driver.OracleDriver
oracle.url=jdbc:oracle:thin:@localhost:1521:XE
oracle.username=system
oracle.password=manager
```

Employee.java

```
package com.nareshit.bean;
public class Employee {
private int empNo;
private String name;
private double salary;
//required setters and getters
}
```

EmployeeDAO.java

```
package com.nareshit.dao;
import java.util.List;
import com.nareshit.bean.Employee;
public interface EmployeeDAO {
        public List<Employee> getAllEmployees();
        public Employee getEmployee(int empNo);
}
```

EmployeeDAOImpl.java

```
package com.nareshit.dao;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;
import com.nareshit.bean.Employee;
   @Repository("employeeDao")
   public class EmployeeDAOImpl implements EmployeeDAO {
    @Autowired
   private JdbcTemplate jdbcTemplate;
        public List<Employee> getAllEmployees() {
                String sql="select empno,name,salary from EmployeeMaster";
                EmployeeRowMapper erm=new EmployeeRowMapper();
                List<Employee> list=jdbcTemplate.query(sql,erm);
                return list;
        }
        public Employee getEmployee(int empNo){
                Employee employee=null;
                String sql="select empno,name,salary from EmployeeMaster where empno=?";
                EmployeeRowMapper erm=new EmployeeRowMapper();
                List<Employee> list=jdbcTemplate.query(sql,erm,empNo);
            if(list.size()>0){
                 employee= list.get(0);
            }
            return employee;
        }
}
```

EmployeeRowMapper.java

```
package com.nareshit.dao;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;
import com.nareshit.bean.Employee;
public class EmployeeRowMapper implements RowMapper<Employee> {
        public Employee mapRow(ResultSet rs,int rowNum)throws SQLException{
                Employee employee=new Employee();
                employee.setEmpNo(rs.getInt("empNo"));
                employee.setName(rs.getString("name"));
                employee.setSalary(rs.getDouble("salary"));
                return employee;

        }

}
```

Test.java

Same as the above applications

ResultSetExtractor:

public T query(String sql,ResultSetExtractor rse)
public T query(String sql,Object[] params, ResultSetExtractor rse)
public T query(String sql, ResultSetExtractor rse,Object... params)

ResultSetExtractor is a callback interface present in org.springframework.jdbc.core

and it has one method
public T extractData(ResultSet rs)throws SQLException

The implementation classes of this interface needs to implement the extratData() method,which takes the responsibility to extracting results from a ResultSet,but does not need to wory about handling the SQLException raised while extracting the results.The SQLException will caught and handled by the JdbcTemplate.

The following **EmployeeDAO.java** shows how to use the ResultSetExtractor

EmployeeDAO.java

```
@Repository("employeeDao")
public class EmployeeDAO {
@Autowired
private JdbcTemplate jdbcTemplate;
public List<Employee> getAllEmployeeDetails(){
String sql="select * from employee ";
List<Employee> list=jdbcTemplate.query(sql,
new ResultSetExtractor<List<Employee>>(){
public List<Employee> extractData(ResultSet rs)
throws SQLException,DataAccessException {
        List<Employee> list=new ArrayList<Employee>();
        while(rs.next()){
        Employee emp=new Employee();
        emp.setEmpno(rs.getInt("empno"));
        emp.setName(rs.getString("name"));
        emp.setDeptno(rs.getInt("deptno"));
        emp.setDeptname(rs.getString("deptname"));
        emp.setSalary(rs.getDouble("salary"));
  list.add(emp);
        }
  return list;
        }//end of extractData()
});
return list;
}//end of getAllEmpolyeeDetails
public Employee getEmployee(int eno){
String sql="select * from employee where empno="+eno;
Employee emp=jdbcTemplate.query(sql,new ResultSetExtractor<Employee>(){
public Employee extractData(ResultSet rs)throws SQLException{
        Employee emp=null;
        if(rs.next()){
        emp=new Employee();
        emp.setEmpno(rs.getInt("empno"));
        emp.setName(rs.getString("name"));
        emp.setDeptno(rs.getInt("deptno"));
        emp.setDeptname(rs.getString("deptname"));
        emp.setSalary(rs.getDouble("salary"));
        }
        return emp;
        }//end of extractData()
});
return emp;
}//end of getEmployee(-)
}//end of EmployeeDAO-class
```

RowCallbackHandler

public void query(String sql, RowCallbackHandler rch)

public void query(String sql, RowCallbackHandler rch,Object... params)

public void query(String sql, Object[] params,RowCallbackHandler rch)

RowCallbackHandler is a callback interface present in **org.springframework.jdbc.core**

and it has one method
public void processRow(ResultSet rs)throws SQLException
The implementation classes of this interface needs to implement the processRow() method,which takes the responsibility to extracting results from a ResultSet,but does not need to wory about handling the SQLException raised while extracting the results.The SQLException will caught and handled by the JdbcTemplate.
The processRow() method should not call next() method on the ResultSet.It requires to extract values of the current Row and process the data on per row basis.
The processRow() method cannot return any result.

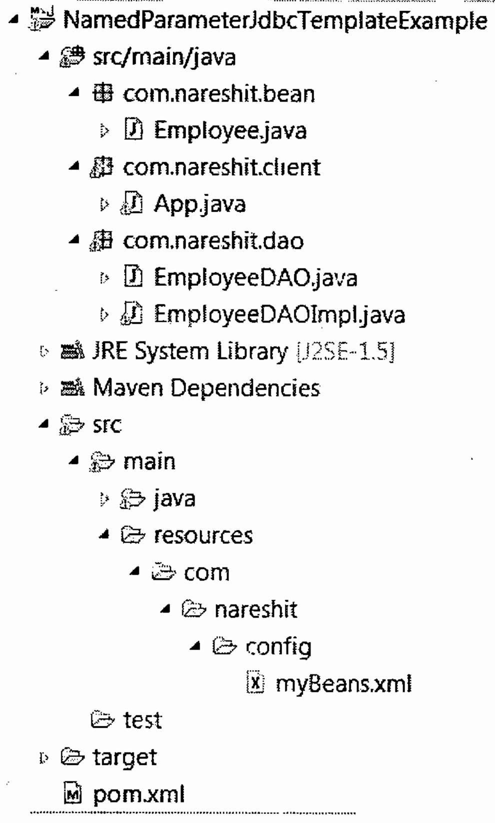
The following **EmployeeDAO.java** shows how to use the RowCallbackHandler
EmployeeDAO.java

```
public class EmployeeDAO extends JdbcDaoSupport {
public List<Employee> getAllEmployeeDetails(){
final    List<Employee> list=new ArrayList<Employee>();
String sql="select * from employee ";
getJdbcTemplate().query(sql,new RowCallbackHandler(){
public void processRow(ResultSet rs)
throws SQLException,DataAccessException {
        Employee emp=new Employee();
        emp.setEmpno(rs.getInt("empno"));
        emp.setName(rs.getString("name"));
        emp.setDeptno(rs.getInt("deptno"));
        emp.setDeptname(rs.getString("deptname"));
        emp.setSalary(rs.getDouble("salary"));
  list.add(emp);
        }//end of processRow()
});
return list;
}//end of getAllEmpolyeeDetails
public Employee getEmployee(int eno){
final Employee emp=new Employee();
String sql="select * from employee where empno="+eno;
getJdbcTemplate().query(sql,new RowCallbackHandler()
{//AIC
public void processRow(ResultSet rs)throws SQLException{
        emp.setEmpno(rs.getInt("empno"));
        emp.setName(rs.getString("name"));
        emp.setDeptno(rs.getInt("deptno"));
        emp.setDeptname(rs.getString("deptname"));
        emp.setSalary(rs.getDouble("salary"));
        }
}
);
return emp;
}//end of getEmployee(-)
}//end of EmployeeDAO-class
```

Q) when we can go for RowMapper?
if we don't want to control on extracting the ResultSet then we can go for RowMapper.
Q) when we can go for ResultSetExtractor?
if we want to control on processing the ResultSet then we can go for ResultSetExtractor.
Note : In JdbcTemplate ,Sql Parameters are represented by using a special place holder "?" symbol and bind it by position.
If the multiple positional parameter are there in a single sql query programmer may be confused while identifying the index's of parameters. To overcome this problem it is recommended to work with named parameters.
In the case of "Named Parameter" SQL parameters are defined by a starting column follow
By a name, rather than by position. In additional the named parameters are only supported in SimpleJdbcTemplate and NamedParameterJdbcTemplate.
NamedParameterJdbcTemplate :

NamedParameterJdbcTemplate class is present in org.springframework.jdbc.core.namedparam package

The template class allowing the use of named parameters rather than traditional '?' placeholders.
This class delegates to a wrapped JdbcTemplate once the substitution from named parameters to JDBC style '?' placeholders is done at execution time. It also allows for expanding a List of values to the appropriate number of placeholders.



```
NamedParameterJdbcTemplateExample
  src/main/java
    com.nareshit.bean
      Employee.java
    com.nareshit.client
      App.java
    com.nareshit.dao
      EmployeeDAO.java
      EmployeeDAOImpl.java
  JRE System Library [J2SE-1.5]
  Maven Dependencies
  src
    main
      java
      resources
        com
          nareshit
            config
              myBeans.xml
    test
  target
  pom.xml
```

Employee.java

```
package com.nareshit.bean;
public class Employee {
private int empNo;
private String name;
private double salary;
public int getEmpNo() {
        return empNo;
}
public void setEmpNo(int empNo) {
        this.empNo = empNo;
}
public String getName() {
        return name;
}
public void setName(String name) {
        this.name = name;
}
public double getSalary() {
        return salary;
}
public void setSalary(double salary) {
        this.salary = salary;
}
}
```

myBeans.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:component-scan base-package="com.nareshit.dao" />
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
<property name="url" value="jdbc:oracle:thin:@localhost:1521:XE"/>
<property name="username" value="system" />
<property name="password" value="manager" />
</bean>
<bean id="namedParameterJdbcTemplate"
class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
<constructor-arg ref="dataSource" />
</bean>

</beans>
```

EmployeeDAO.java

```
package com.nareshit.dao;
```

```
import com.nareshit.bean.Employee;
public interface EmployeeDAO {
public int createEmployee(Employee emp);
}
```

EmployeeDAOImpl.java

```
package com.nareshit.dao;
import java.util.HashMap;
import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.stereotype.Repository;
import com.nareshit.bean.Employee;
@Repository("employeeDao")
public class EmployeeDAOImpl  implements EmployeeDAO{
   @Autowired
        private NamedParameterJdbcTemplate namedParameterJdbcTemplate;
public int createEmployee(Employee emp) {
String sql="insert into emp values(:empno,:name,:salary)";
Map<String,Object> map=new HashMap<String, Object>();
map.put("empno",emp.getEmpNo());
map.put("name",emp.getName());
map.put("salary",emp.getSalary());
int count=namedParameterJdbcTemplate.update(sql,map);
            return count;
        }

}
```

App.java
Same as above applications

SimpleJdbcTemplate :

SimpleJdbcTemplate class is present in org.springframework.jdbc.core.simple packge And It is a Java-5-based convenience wrapper for the classic Spring JdbcTemplate, taking advantage of varargs and autoboxing, and exposing only the most commonly required operations in order to simplify JdbcTemplate usage.
since Spring 3.1 in favor of JdbcTemplate and NamedParameterJdbcTemplate SimpleJdbcTemplate class is deprecated . The JdbcTemplate and NamedParameterJdbcTemplate now provide all the functionality of the SimpleJdbcTemplate.

Calling Stored Procedures with JdbcTemplate

- The Stored procedures contains input,output (OR) both the input and output parameters.
- After the execution of the SQL statements, the stored procedures return a value through the OUT parameters
- The StoredProcedures can returns Multiple ResultSets.

The Stored Procedures encapsulates the values within three types of parameters Such as IN,OUT, and INOUT.

The following parameters can ve declared while creating a procedure for the database.

IN- The IN parameter can be referenced by the Procedure. The value of the Parameter cannot be overwritten Procedure.

OUT- This Parameter cannot be referenced by the procedure,but the value of the parameter can be overwritten by the procedure.

INOUT- The parameter can be referenced by the Procedure and the value of the parameter can be Overwritten by the Procedure.

The spring f/w provides a support for calling stored procedures. JdbcTemplate class contain a method named as call(-,-) to execute the procedures.

prototype of call() method:-
public Map call(CallableStatementCreator cst,List declaredParameterTypes)
The first argument of the call() method is a CallableStatementCreator, which is a callback interface used by the JdbcTemplate to get the CallableStatement and to execute a procedure.
The CallableStatementCreator interface contains only one method with the following signature.

CallableStatementCreator method

public CallableStatement createCallableStatement(Connection con)throws SQLException

By using the given Jdbc Connection create CallableStatement object and the set the parameters inside the createCallableStatement() method.

The second argument of the call() method is a java.util.List, which encapsulates the list of declared SQLParameter objects

Stored Procedure :

```
create or replace procedure GETEMPLOYEEDETAILS(eno IN NUMBER,
 ename OUT VARCHAR2, dname OUT VARCHAR2)  is
begin
select name,deptname into ename,dname from emp  where empno=eno;
end;
/
```

The above stored procedure takes one IN parameter and Two OUT parameters.
Now we will write a CallableStatementCreator callback to prepare a CallableStatement that can executes the above Stored Procedure .

MyCallableStatementCreator.java

```
public class MyCallableStatementCreator implements CallableStatementCreator{
 private int eno;
public  MyCallableStatementCreator(int eno){
  this.eno=eno;
 }
 public CallableStatement createCallableStatement(Connection con)throws SQLException{
CallableStatement cst=con.prepareCall("{call GETEMPLOYEEDETAILS1(?,?,?)}");
          cst.setInt(1,eno);
```

```
            cst.registerOutParameter(2,Types.VARCHAR);
            cst.registerOutParameter(3,Types.VARCHAR);
             return cst;
                        }
             }
```

The following EmployeeDAOImpl uses JdbcTemplate class to execute the Procedure

EmployeeDAOImpl.java

```
@Repository("employeeDao")
public class EmployeeDAOImpl implements EmployeeDAO{
                @Autowired
               private JdbcTemplate jdbcTemplate;
            public void getEmployeeDetails(int empno) {
            List<SqlParameter> list=new ArrayList<SqlParameter>();
            list.add(new SqlParameter("eno",Types.INTEGER));
            list.add(new SqlOutParameter("ename",Types.VARCHAR));
            list.add(new SqlOutParameter("dname",Types.VARCHAR));
MyCallableStatementCreator mcsc=new MyCallableStatementCreator(empno);
 Map<String,Object> results=jdbcTemplate.call(mcsc,list);
                        System.out.println("results :"+results);
                        }//end of getEmployeeDetails()
}
```

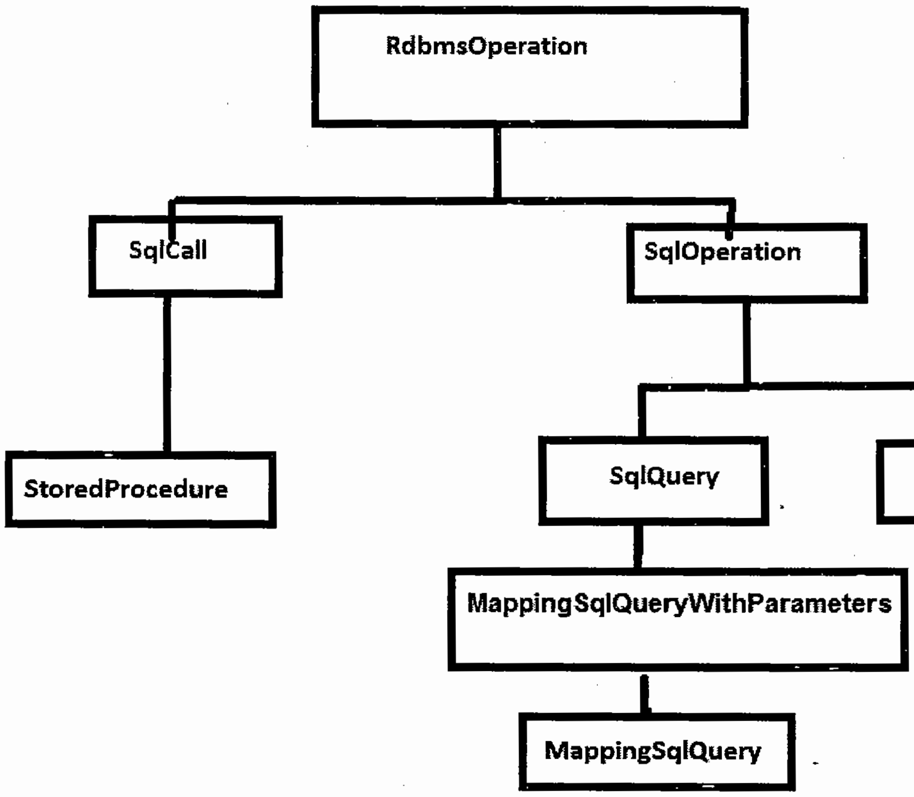
RDBMS Operation classes :

Spring Jdbc Abstraction Framework includes RDBMS operation classes,which provides a high level Object-Oriented abstraction to represent the RDBMS queries,updates,and stored procedures as threadsafe,reusable objects.

All the RDBMS classes are present in org.springframework.jdbc.object package.

org.springframework.jdbc.object.RdbmsOperation class is the root of the RDBMS operation classes.

RdbmsOperation class is a multithreaded, reusable type. The direct sub classes are SqlCall and SqlOperation.



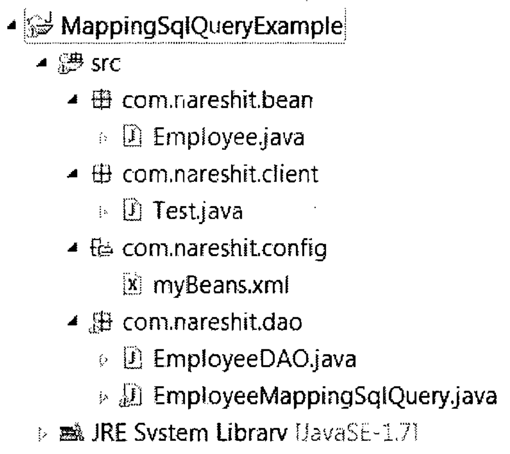
The SqlCall is used for representing and executing the stored procedures. The SqlOperation is used to represent the SQL query and updates using it's subtypes SqlQuery and SqlUpdate respectively.

The SqlQuery class:
The SqlQuery is an abstract class that provides an abstraction to represent a reusable SQL query to access the Database.
The SqlQuery class contain a number of convenient execute() methods for executing the query represented by this Object.
The most widely and easy to use subclass of this class is MappingSqlQuery .
The MappingSqlQuery class simplifies the MappingSqlQuery WithParameters by reducing the parameters and context details.
MappingSqlQuery Object is a threadSafe,reusable , multithreaded object.
The sub classes of MappingSqlQuery needs to implement the abstract mapRow() method to convert each row of the resultSet into and domain Object.

MappingSqlQuery class mapRow() method signature:

public abstract Object mapRow(ResultSet rs,int rowNum)throws SQLException

The following example shows how to use the MappingSqlQuery



```
MappingSqlQueryExample
  src
    com.nareshit.bean
      Employee.java
    com.nareshit.client
      Test.java
    com.nareshit.config
      myBeans.xml
    com.nareshit.dao
      EmployeeDAO.java
      EmployeeMappingSqlQuery.java
  JRE System Library [JavaSE-1.7]
```

Employee.java

```
package com.nareshit.bean;
public class Employee {
private int empno,deptno;
private String deptname,name;
private double salary;
//required setters and getters
}
```

EmployeeMappingSqlQuery.java

```
package com.nareshit.dao;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.sql.DataSource;
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.MappingSqlQuery;
import com.nareshit.bean.Employee;
public class EmployeeMappingSqlQuery extends MappingSqlQuery<Employee>{
        public EmployeeMappingSqlQuery(DataSource ds,String sql)      {
                super(ds,sql);
        }
public Employee mapRow(ResultSet rs,int rowNumber)throws SQLException{
Employee emp=new Employee();
emp.setEmpno(rs.getInt(1));
emp.setName(rs.getString(2));
emp.setDeptno(rs.getInt(3));
emp.setDeptname(rs.getString(4));
emp.setSalary(rs.getDouble(5));
return emp;
}
}
```

EmployeeDAO.java

```
package com.nareshit.dao;
import java.sql.Types;
import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.SqlParameter;
import com.nareshit.bean.Employee;
public class EmployeeDAO {
```

```
private DataSource dataSource;
public void setDataSource(DataSource dataSource) {
                this.dataSource = dataSource;
}
public List<Employee> getAllEmployees(){
String sql="select *from employee";
EmployeeMappingSqlQuery sqlQuery=new EmployeeMappingSqlQuery(dataSource,sql);
List<Employee> list=sqlQuery.execute();
        return list;
        }
public Employee getEmployee(int eno){
Employee emp=null;
String sql="select *from employee where empno=?";
EmployeeMappingSqlQuery sqlQuery=new EmployeeMappingSqlQuery(dataSource,sql);
sqlQuery.declareParameter(new SqlParameter("empno",Types.INTEGER));
List<Employee> list=sqlQuery.execute(eno);
if(list.size()>0){
emp=list.get(0);
}
return emp;
}
}
```

myBeans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
        <bean id="dataSource"
                class="org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
<property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
        <property name="username" value="system" />
        <property name="password" value="manager" />
        </bean>
        <bean id="empDao" class="com.nareshit.dao.EmployeeDAO">
                <property name="dataSource" ref="dataSource" />
        </bean>
</beans>
```

Test.java

```
package com.nareshit.client;
import java.util.List;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;
import com.nareshit.bean.Employee;
import com.nareshit.dao.EmployeeDAO;

public class Test {
public static void main(String[] args) {
        Resource r=new ClassPathResource("com/nareshit/config/myBeans.xml");
        BeanFactory factory=new XmlBeanFactory(r);
     EmployeeDAO empDao=(EmployeeDAO)factory.getBean("empDao");
        List<Employee> list=empDao.getAllEmployees();
        System.out.println("Response of getAllEmployees");
        for(Employee emp:list) {
                System.out.println(emp);
        }
        Employee emp=empDao.getEmployee(1001);
    System.out.println("Response of  getEmployee(1001) :");
        System.out.println(emp);
        }
}
```

SqlUpdate class :

The SqlUpdate is a concrete class and contain the number of convenient update() methods for executing the sqlupdates .

The following **EmployeeDAOImpl.java** shows how to use the SqlUpdate

EmployeeDAOImpl.java

```
@Repository("employeeDao")
public class EmployeeDAOImpl {
  @Autowired
private DataSource dataSource;
public int updateEmployeeName(int empNo,String name) {
String sql="update employee set name=? where empno=?";
SqlUpdate sqlUpdate=new SqlUpdate(dataSource,sql);
sqlUpdate.declareParameter(new SqlParameter(Types.VARCHAR));
sqlUpdate.declareParameter(new SqlParameter(Types.INTEGER));
    int count= sqlUpdate.update(name,empNo);
                return count;
        }
        public int deleteEmployee(int empNo) {
        String sql="delete from employee where empno=?";
        SqlUpdate sqlUpdate=new SqlUpdate(dataSource,sql);
        sqlUpdate.declareParameter(new SqlParameter(Types.INTEGER));
        int count=sqlUpdate.update(empNo);
                return count;
}
}
```

StoredProcedure :

The storedProcedure class is an abstract class ,which provides an abstraction for representing the RDBMS stored procedures.

To use the StoredProcedure functionality take an user-defined class as a sub class to StoredProcedure.

The following **MyStoredProcedure.java** shows the StoredProcedure implementation class that represents the procedure.

MyStoredProcedure.java

```
public class MyStoredProcedure extends StoredProcedure{
public MyStoredProcedure(DataSource dataSource,String procedure){
                super(dataSource,procedure);
declareParameter(new SqlParameter("eno",Types.INTEGER));
declareParameter(new SqlOutParameter("ename",Types.VARCHAR));
declareParameter(new SqlOutParameter("dname",Types.VARCHAR));
```

```
        compile();
        }
}
```

The following **EmployeeDAOImpl.java** shows how to use the storedProcedure implementation

EmployeeDAOImpl.java

```
@Repository("employeeDao")
public class EmployeeDAOImpl{
        @Autowired
private DataSource dataSource;
public void getEmployeeDetails(final int empno){
String procedure="GETEMPLOYEEDETAILS";
Map<String,Object> paramMap=new HashMap<String,Object>();
paramMap.put("eno",empno);
MyStoredProcedure storedProcedure=new MyStoredProcedure(dataSource, procedure);
Map<String,Object> resultMap=storedProcedure.execute(paramMap);
  System.out.println(resultMap);
   }
}
```

SPRING ORM

- Spring Jdbc approach has performance benefit when compared to ORM approach.But is has the following limitations.
- coding complexity
- SQL portability
- Manual operations on ResultSets
- Spring ORM provides integration with Hibernate, JDO, Oracle TopLink, and JPA: in terms of resource management, DAO implementation support, and transaction strategies.
- For example, for Hibernate there is first-class support with several convenient IoC features that address many typical Hibernate integration issues. You can configure all of the supported features for O/R (object relational) mapping tools through Dependency Injection. They can participate in Spring's resource and transaction management, and they comply with Spring's generic transaction and DAO exception hierarchies. The recommended integration style is to code DAOs against plain Hibernate, JPA, and JDO APIs.

Q) What is the traditional sytle of achieving persistence in java based enterprise application?

- Sending SQL statements to the Database using JDBC API

Q) What are the limitations of the traditional approach?

- Application portability to the DB is lost. (Vendor lock: diff SQL statement for the db's)
- Mismatches between Object oriented data representation and relational data representation are not properly addressed.
- Requires the extensive knowledge of DB
- Too many steps to maintain transactions.
- Manual operations on Result sets
- Need to tune your queries
- Need to implement caching manually.

Q) What is an alternative for traditional approach?

- ORM (Object Relational Mapping)
- Object Relational mapping is technique of mapping objected oriented data representation to that of relational data representation
- Through ORM technique persistence services (database) are provided to business layer in pure object oriented manner by overcoming all limitations of the traditional approach.

Q) What are the ORM frameworks are there?

- Hibernate
- Toplink
- Ibatis
- JDO
- JPA(Java Persistence API)

Q) Is Spring ORM an implementation of ORM (Object Relational Model) like Hibernate?

- No, it is not ORM implementation

- It is the one of the modules of spring framework, which is used to integrate any ORM into spring framework

Q) Why to integrate ORM modules into spring framework?

There are some of the following common practices are there in all most all ORM implementation frameworks

1. Exception Handling Logic
2. Transaction Management Logic
3. Resource allocation Logic
4. Resource Releasing Logic

Exception Handling Logic

- Spring Frame Work provides the **Fine Grind Exception Handling** mechanism to deal with DataBase i.e., it defines specific Exception for each and every problem that occurs while dealing with the DB.
- **DataAccessException:** org.springframework.dao.DataAccessException is the top most Exception in spring DAO exception hierarchy.
- In Spring DAO Exception Hierarchy we have a support (or **Spring Exception Translator)** to 2Transform lo-level Data Access API Exception to the Spring DAO Exception.
- **DataAccessException** is the child class of **RuntimeException** so all Spring DAO Exceptions are unchecked Exceptions.
- Spring provides **Declarative Exception Handling** Mechanism, means we can handle the Exceptions in the **"Spring-Configuration"** file.

Transaction Management Logic

- Spring allows you to wrap your O/R mapping code with either a declarative, AOP style method interceptor, or an explicit 'template' wrapper class at the java code leve.
- In either case, transaction semantics are handled for you, and proper transaction handling (rollback, etc) in case of exceptions is taken care of. As discussed below, you also get the benefit of being able to use and swap various transaction managers, without your Hibernate/JDO related code being affected:
- For example, between local transactions and JTA, with the same full services (such as declarative transactions) available in both scenarios.
- As an additional benefit, JDBC-related code can fully integrate transactionally with the code you use to do O/R mapping.
- Spring Framework supports Distributive Transaction Management.

Resource allocation Logic

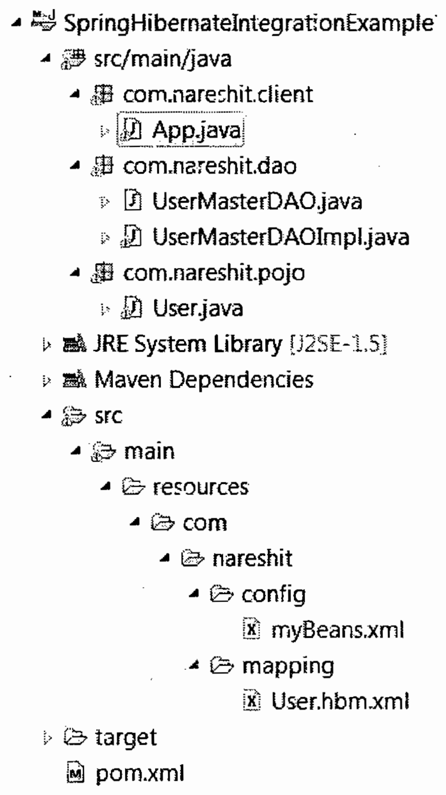
- **XXXTemplate** provided by the spring which, abstract the DB Resource Allocation Logic .

Resource Releasing Logic

- Spring provided **XXXTemplate,** which automatically releases the DB resources.

Spring-HibernateIntegration

- In spring-hibernate integration, we have to add both spring with hibernate capabilities.
- In hibernate, we have configuration file (hibernate.cfg.xml) and mapping file (EntityClass.hbm.xml).
- In hibernate.cfg.xml we provide the Database details ,mapping resources and hibernateProperties.
- So, there will be two configuration files i.e., spring configuration file (applicationContext.xml) and hibernate configuration file (hibernate.cfg.xml).
- But in general while integrating spring and hibernate we won't write hibernate.cfg.xml, we will give this file(hibernate.cfg.xml) information also in the spring configuration file. And It is recommend. So Database details(url, username, password, driver class), Dialect class, other hibernate properties (show_sql, hbm2.ddl.auto ect.), hibernate mapping file information in the spring configuration file.
- We will configure all these information to **LocalSessionFactoryBean** of spring framework.
- LocalSessionFactoryBean creates a Hibernate SessionFactory. This is the usual way to set up a shared Hibernate SessionFactory in a Spring application context; the SessionFactory can then be passed to Hibernate-based DAOs via dependency injection.



```
SpringHibernateIntegrationExample
  src/main/java
    com.nareshit.client
      App.java
    com.nareshit.dao
      UserMasterDAO.java
      UserMasterDAOImpl.java
    com.nareshit.pojo
      User.java
  JRE System Library [J2SE-1.5]
  Maven Dependencies
  src
    main
      resources
        com
          nareshit
            config
              myBeans.xml
            mapping
              User.hbm.xml
  target
  pom.xml
```

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
<groupId>com.nareshit</groupId>
```

```
 <artifactId>SpringHibernateIntegrationExample</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>jar</packaging>
 <name>SpringHibernateIntegrationExample</name>
 <url>http://maven.apache.org</url>
<properties>
 <spring-framework.version>3.0.1.RELEASE
 </spring-framework.version>
 <hibernate-framework.version>3.6.0.Final
 </hibernate-framework.version>
 </properties>
 <dependencies>
    <!-- Spring context -->
      <dependency>
         <groupId>org.springframework</groupId>
     <artifactId>spring-context</artifactId>
 <version>${spring-framework.version}</version>
      </dependency>

 <!-- Spring JDBC Support -->
      <dependency>
         <groupId>org.springframework</groupId>
         <artifactId>spring-jdbc</artifactId>
         <version>${spring-framework.version}</version>
      </dependency>
      <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-orm</artifactId>
          <version>${spring-framework.version}</version>
</dependency>

      <!-- Oracle Driver -->
      <dependency>
         <groupId>oracle.jdbc.driver</groupId>
         <artifactId>ojdbc14
         </artifactId>
         <version>10.2.0</version>
      </dependency>
      <dependency>
          <groupId>org.hibernate</groupId>
          <artifactId>hibernate-core</artifactId>
          <version>${hibernate-framework.version}</version>
</dependency>
      <dependency>
          <groupId>org.javassist</groupId>
          <artifactId>javassist</artifactId>
          <version>3.15.0-GA</version>
</dependency>
  </dependencies>
</project>
```

User.java

```
package com.nareshit.pojo;
```

```
public class User {
private Long userId;
private String name;
private String mobile;
private String email;
private Byte age;
//required setters and getters
}
```

User.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.User"
table="User">
<id name="userId" column="userId"/>
<property name="name" column="name"/>
<property name="age" column="age"/>
<property name="email" column="email"/>
<property name="mobile" column="mobile"/>
</class>
</hibernate-mapping>
```

UserMasterDAO.java

```
package com.nareshit.dao;
import com.nareshit.pojo.User;
public interface UserMasterDAO {
public long createUser(User user);
public User getUser(long userId);
public List<User> getUsers();
public int UpdateUserProfile(User user);
public int deleteUser(long userId);
}
```

UserMasterDAOImpl.java

```
package com.nareshit.dao;
import java.util.List;
import org.springframework.orm.hibernate3.HibernateTemplate;
import com.nareshit.pojo.User;
public class UserMasterDAOImpl implements UserMasterDAO {
private HibernateTemplate hibernateTemplate;
   public long createUser(User user){
    Long userId=(Long)hibernateTemplate.save(user);
   return userId;
      }
       public User getUser(long userId) {
              User user=null;
              if(userId>0){
 user=(User)hibernateTemplate.get(User.class,userId);
              }
       return user;
       }
       public List<User> getUsers() {
       //HQL QUERY
String hql="from com.nareshit.pojo.User";
 List<User> list=hibernateTemplate.find(hql);
 return list;
        }

        public int UpdateUserProfile(User user) {
String hql="update com.nareshit.pojo.User as u"
                + " set u.name=?,u.age=?,u.email=?,u.mobile=? "
                + "where u.userId=?";
 int
count=hibernateTemplate.bulkUpdate(hql,user.getName(),user.getAge(),user.getEmail(),user.getMobile()
,user.getUserId());

                return count;
        }

public int deleteUser(long userId) {
        String hql="delete from com.nareshit.pojo.User as u"
                        + " where u.userId=?";
        int count =hibernateTemplate.bulkUpdate(hql,userId);
                return count;
        }

        public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
                this.hibernateTemplate = hibernateTemplate;
        }

}
```

myBeans.xml

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

<beans>
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName"
value="oracle.jdbc.driver.OracleDriver"/>
<property name="url"
value="jdbc:oracle:thin:@localhost:1521:XE"/>
<property name="username" value="system"/>
<property name="password" value="manager"/>
</bean>
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
<property name="dataSource" ref="dataSource"/>
<property name="mappingResources">
<value>com/nareshit/mapping/User.hbm.xml</value>
</property>
<property name="hibernateProperties">
<props>
<prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect
</prop>
<prop key="hibernate.show_sql">true</prop>
</props>
```

```
</property>
</bean>
<bean id="hibernateTemplate" class="org.springframework.orm.hibernate3.HibernateTemplate">
<property name="sessionFactory" ref="sessionFactory"/>
</bean>
<bean id="userDao" class="com.nareshit.dao.UserMasterDAOImpl">
<property name="hibernateTemplate" ref="hibernateTemplate"/>
</bean>
</beans>
```

Note:-

while performing non-selection operations on table by using Plain Hibernate persistance logic we need to perform transaction Management Explicitly,this TransactionManagement is optional

while working with HibernateTemplate

HibernateTemplateclass methods:-

The HibernateTemplate class contains various methods that reflect the Hibernate Session methods.

HibernateTemplate takes the Responsibility to manage the hibernate sessions making the application free from implementing the code for opening and closing the Hibernate Session Accordingly.

The other important services is to convert the Hibernate Exceptions into DataAccessException.

Spring f/w includes three hibernateTemplate classes -

→ org.springframework.orm.hibernate.HibernateTemplate class supports to work with hibernate 2.1(2.x)

- ➔ org.springframework.orm.hibernate3.HibernateTemplate class supports to work with Hibernate 3.x
- ➔ org.springframework.orm.hibernate4.HibernateTemplate class supports to work with Hibernate 4.x

Note :- spring 2.5 version onwords supports Hibernate 3.1 (OR) higher version features

get(-,-),load(-,-),update(-),save(-),merge(-),saveOrUpdate(-),delete(-) and etc..

methods are given in HibernateTemplate class to perform single row operations(same as plain Hibernate Api)

find(),findXxx(),bulkUpdate() methods are given for Bulk operations

List find(String queryString) : this method Executes the given HQL query.

List find(String queryString, Object value): Execute an HQL query, binding one value to a "?"

List find(String queryString, Object... values) : Execute an HQL query, binding a number of values to "?"

List findByNamedParam(String queryString, String[] paramNames, Object[] values) Execute an HQL query, binding a number of values to ":" named parameters in the query string.

List findByNamedQuery(String queryName) : Execute a named query.

List findByCriteria(DetachedCriteria criteria) : Execute a query based on a given Hibernate criteria object.

List findByCriteria(DetachedCriteria criteria, int firstResult, int maxResults) : Execute a query based on the given Hibernate criteria object.

int bulkUpdate(String queryString) : Update/delete all objects according to the given hql query.

int bulkUpdate(String queryString, Object value) : Update/delete all objects according to the given hql query, binding one value to a "?"

int bulkUpdate(String queryString, Object[] values) : Update/delete all objects according to the given hql query, binding a number of values to "?"

public Object execute(HibernateCallback action) throws DataAccessException : Execute the action specified by the given action object within a Session.

Note:-

The following Operations are not possible directly with HibernateTemplate class persistence methods

1)execution of Native Sql Queries with Named,Positional parameter 's

2) Execution of Criterai Api based Persistance Logic

3) Execution of Non-select HQL Queries with NamedParameter 's

Q) Why we are going for Callback mechanism?

Ans: When we are unable to implement underlying framework (OR) technology functionality using xxxTemplate then we need to go for Callback mechanism.

Q) What we can achieve with xxxCallback?

Ans: Any Specific functionality of underlying framework or technology can be implemented. Because in the Callback mechanism they will pass underlying framework resource (Connection, Session, SqlMap... etc).

HibernateCallBack:

When we are not getting the specific functionality of hibernate by using HibernateTemplate those functionalities we can achieve by using HibernateCallback.

To solve all the above problems work with HibernateCallback(I) .

The HibernateCallback Interface declares only one method with the following signature

public Object doInHibernate(Session session)throws HibernateException,SQLException

the doInHibernate(-) method allows us to implement the Hibernate data access operations using the

given Session.Here we do not require to worry about Opening (OR) closing the Session,handling transactions (OR) handling HibernateException.

Executing NativeSQLQuery with Positional Parameter by using Hibernate Template class

Object with the support of Hibernate CallBack interface

```
public List getAllUsers(){

List list=(List) hibernateTemplate.execute(new HibernateCallback(){

public Object doInHibernate(Session session)throws HibernateException,SQLException{

SQLQuery query=session.createSQLQuery("select  *from User");

query.addEntity("com.nareshit.pojo.User");

//Mapping the Result with Hibernate pojo class

List list=query.list();

});

return list;

}
```

Executing hql NON-SELECT Query with named Parameter by using Hibernate Template class

Object with the support of Hibernate CallBack interface

```
public Integer deleteUser(){

Integer count=(Integer)hibernateTemplate.execute(new HibernateCallback(){

public Object doInHibernate(Session session)throws HibernateException,SQLException{

Query query=session.createQuery("delete from com.nareshit.pojo.User  as u where   u.userId=:userId")

query.setInteger("userId",1001L);

Integer count=query.executeUpdate();

return count;
});//doInHibernate(-)

return count;

}
```

Configuring Hibernate in a Spring context

Spring provides the *LocalSessionFactoryBean* class as a factory for a *SessionFactory* object. The *LocalSessionFactoryBean* object is configured as a bean inside the IOC container, with either a local JDBC DataSource or a shared DataSource from JNDI.

The local JDBC DataSource can be configured in turn as an object of *org.apache.commons.dbcp.BasicDataSource* in the Spring context:

```
<bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName">
    <value>oracle.jdbc.driver.OracleDriver</value>
  </property>
  <property name="url">
    <value>jdbc:oracle:thin:@localhost:1521:XE</value>
  </property>
  <property name="username">
    <value>system</value>
  </property>
  <property name="password">
    <value>manager</value>
  </property>
</bean>
```

In this case, the *org.apache.commons.dbcp.BasicDataSource* (the Jakarta Commons Database Connection Pool) must be in the application classpath.

Similarly, a shared DataSource can be configured as an object of *org.springframework.jndi.JndiObjectFactoryBean.* This is the recommended way, which is used when the connection pool is managed by the application server. Here is the way to configure it:

```
<bean id="dataSource"
class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>java:comp/env/jdbc/HiberDB</value>
  </property>
</bean>
```

When the DataSource is configured, you can configure the *LocalSessionFactoryBean* instance upon the configured DataSource as follows:

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
   <property name="dataSource">
      <ref bean="dataSource"/>
   </property>
```

```
    ...
</bean>
```

Alternatively, you may set up the *SessionFactory* object as a server-side resource object in the Spring context. This object is linked in as a JNDI resource in the JEE environment to be shared with multiple applications. In this case, you need to use *JndiObjectFactoryBean* instead of *LocalSessionFactoryBean*:

```
<bean id="sessionFactory"
class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>java:comp/env/jdbc/hiberDBSessionFactory</value>
  </property>
</bean>
```

JndiObjectFactoryBean is another factory bean for looking up any JNDI resource.

When you use JndiObjectFactoryBean to obtain a preconfigured SessionFactory object, the SessionFactory object should already be registered as a JNDI resource. For this purpose, you may run a server-specific class which creates a SessionFactory object and registers it as a JNDI resource.

LocalSessionFactoryBean uses three properties: *datasource*, *mappingResources*, and *hibernateProperties*. These properties are as follows:

- *datasource* refers to a JDBC *DataSource* object that is already defined as another bean inside the container.
- *mappingResources* specifies the Hibernate mapping files located in the application classpath.
- *hibernateProperties* determines the Hibernate configuration settings.

We have the *sessionFactory* object configured as follows:

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource">
    <ref bean="dataSource"/>
  </property>
  <property name="mappingResources">
   <list>
    <value>com/nareshit/mapping/User.hbm.xml</value>
   <value>com/nareshit/mapping/Student.hbm.xml</list>
  </property>
  <property name="hibernateProperties">
    <props>
     <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect
     </prop>
     <prop key="hibernate.show_sql">true</prop>
        </props>
  </property>
</bean>
```

The mappingResources property loads mapping definitions in the classpath. You may use mappingJarLocations, or mappingDirectoryLocations to load them from a JAR file, or from any directory of the file system, respectively.

It is still possible to configure Hibernate with *hibernate.cfg.xml*, instead of configuring Hibernate as just shown. To do so, configure *sessionFactory* with the configLocation property, as follows:

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="configLocation">
  <value>/com/nareshit/config/hibernate.cfg.xml</value>
 </property>
</bean>
```

Note that *hibernate.cfg.xml* specifies the Connection Details, Hibernate mapping definitions in addition to the other Hibernate properties.

When the *SessionFactory* object is configured, you can configure DAO implementations as beans in the Spring context. These DAO beans are the objects which are looked up from the Spring IoC container and consumed by the business layer. Here is an example of DAO configuration:

```
<bean id="userDao"
class="com.nareshit.dao.UserDAOImpl">
 <property name="sessionFactory">
  <ref local="sessionFactory"/>
 </property>
</bean>
```

This is the DAO configuration for a DAO class that extends *HibernateDaoSupport*, or directly uses a *SessionFactory* property. When the DAO class has a *HibernateTemplate* property, configure the DAO instance as follows:

```
<bean id="userDao"
class="com.nareshit.dao.UserDAOImpl">
 <property name="hibernateTemplate">
  <bean
class="org.springframework.orm.hibernate3.HibernateTemplate">
    <constructor-arg>
     <ref local="sessionFactory"/>
    </constructor-arg>
  </bean>
 </property>
</bean>
```

According to the preceding declaration, the *UserDAOImpl* class has a *hibernateTemplate* property that is configured via the IoC container, to be initialized through constructor injection and a *SessionFactory* instance as a constructor argument.

Now, any client of the DAO implementation can look up the Spring context to obtain the DAO instance.

Note :- if the hibernate Mapping is defined with Annotation's then in spring configration file

Instead of LocalSessionFactoryBean we can configure AnnotationSessionFactoryBean class.

AnnotationSessionFactory Bean is the Child class of LocalSessionFactoryBean.

User.java

```
package com.nareshit.pojo;

@Entity

@Table(name="user")

public class User {

@Id

@Column(name="userId")

private Long userId;

@Column(name="name",length=20)

private String name;

@Column(name="mobile",length=20)

private String mobile;

@Column(name="email",length=25)

private String email;

@Column(name="age")

private Byte age;

//required setters and getters

}
```

myBeans.xml

```
    <beans>

    <bean id="dataSource" class="....BasicDataSource">

                ....

        </bean>

        <bean id="sessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">

                <property name="dataSource" ref="dataSource"/>

                <property name="hibernateProperties">

                        ....same as the above applications...

                </property>
```

```
            <property name="annotatedClasses">

                    <value>com.nareshit.dao.User</value>

            </property>

        </bean>

        <bean id="hibernateTemplate" class="....HibernateTemplate">

            <property name="sessionFactory" ref="sessionFactory"/>

        </bean>

    </beans>
```

Q) What is Hibernate Dao Support?

- While working with hibernate, into all dao classes we need to inject **HibernateTemplate** object. So injecting **HibernateTemplate** object, logic is repating in all the dao classes. So spring people has given one abstract class called **HibernateDaoSupport**, which contains this common **HibernateTemplate** object injection logic.
- So while writing Dao class it is better to extend **HibernateDaoSupport**, so that we no need to write injection logic of **HibernateTemplate**.
- When we need **HibernateTemplate** object in dao class just we call **getHibernateTemplate()**. So, by extending this class we can get **HibernateTemplate**, on that we can perform our operations.
- One more advantage of **HibernateDaoSupport** is we can directly inject **LocalSessionFactoryBean** to dao class instead of **HibernateTemplate**

Example:-

```
public class UserMasterDAOImpl extends HibernateDaoSupport implements UserMasterDAO {
   public long createUser(User user){
     Long userId=(Long)getHibernateTemplate().save(user);
   return userId;
       }
         }
```

Spring WebMVC

Spring Web MVC Framework is an open Source web application framework which is a part of Spring Framework.

Spring Web MVC is one of the efficient and high performance open source implementation of Model2 based Model-View-Controller arch.

Spring WebMvc allows you to build web application's,Similar to Servlet and Struts etc....

Spring WebMvc is used to develop Model-2 Arch based web applications by using own classes and custom action tags.

The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that dispatches requests to handlers, with configurable handler mappings, view resolution, locale, time zone and theme resolution as well as support for uploading files.

Advantage of Spring MVC Framework

-->Spring MVC used to develop the web applications that uses MVC Arch

-->Spring MVC is used for making web application development faster, cost-effective, and flexible.

-->It provides Loose coupling among Model,View and Controller.

-->It Provides In built front controller(DispatcherServlet)

-->Validation implementations is simplified.

-->Exception Handling is simplified

-->User input is available in the object oriented format.

-->Forward logic is simplified.

-->Internationalization (i18n) logic is simplified.

-->Spring MVC provides a set of custom JSP tags,which are useful to implement presentation logic.

-->Also spring can intergrate with other popular Web Frameworks like Struts,WebWork,Java Server Faces and Tapestry.

-->Integration with other View technologies like Velocity,Freemarker,Excel or Pdf.

FrameWork :

A Framework is a reusable semi fininshed applicaiton that can be customised to develop a specific application.

Types of FrameWorks

1. Web FrameWorks

2. Application FrameWorks

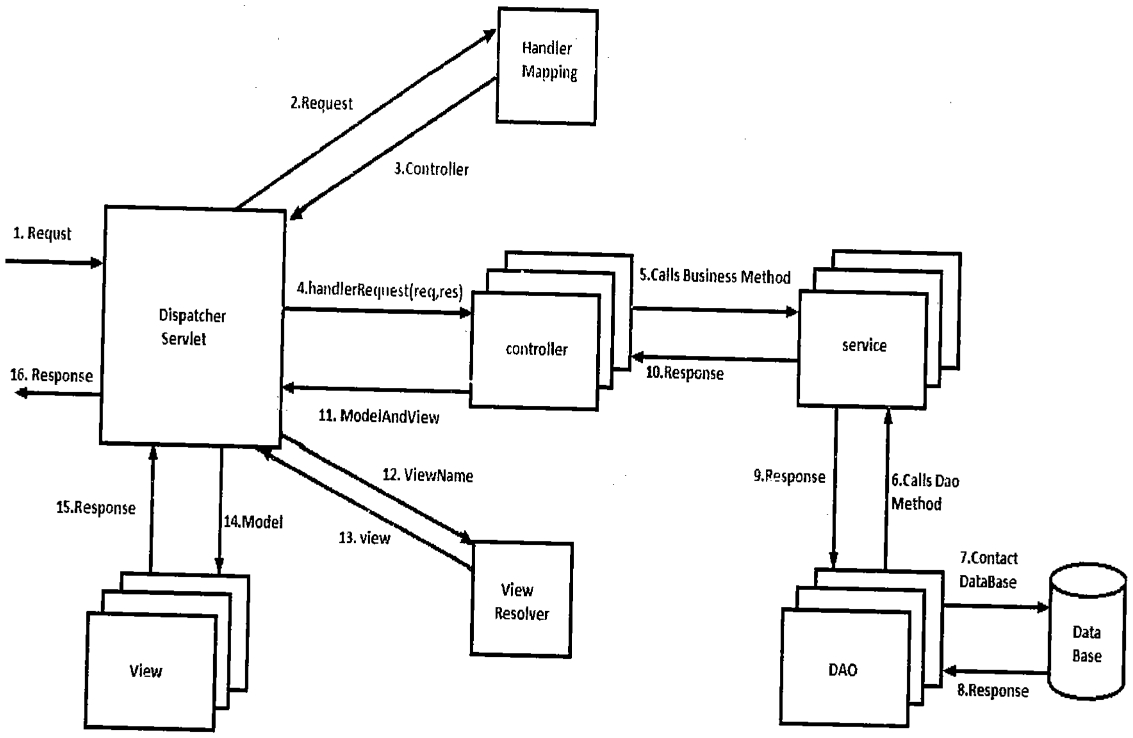
Web FrameWork will provide an environment to design and execute only web applications.

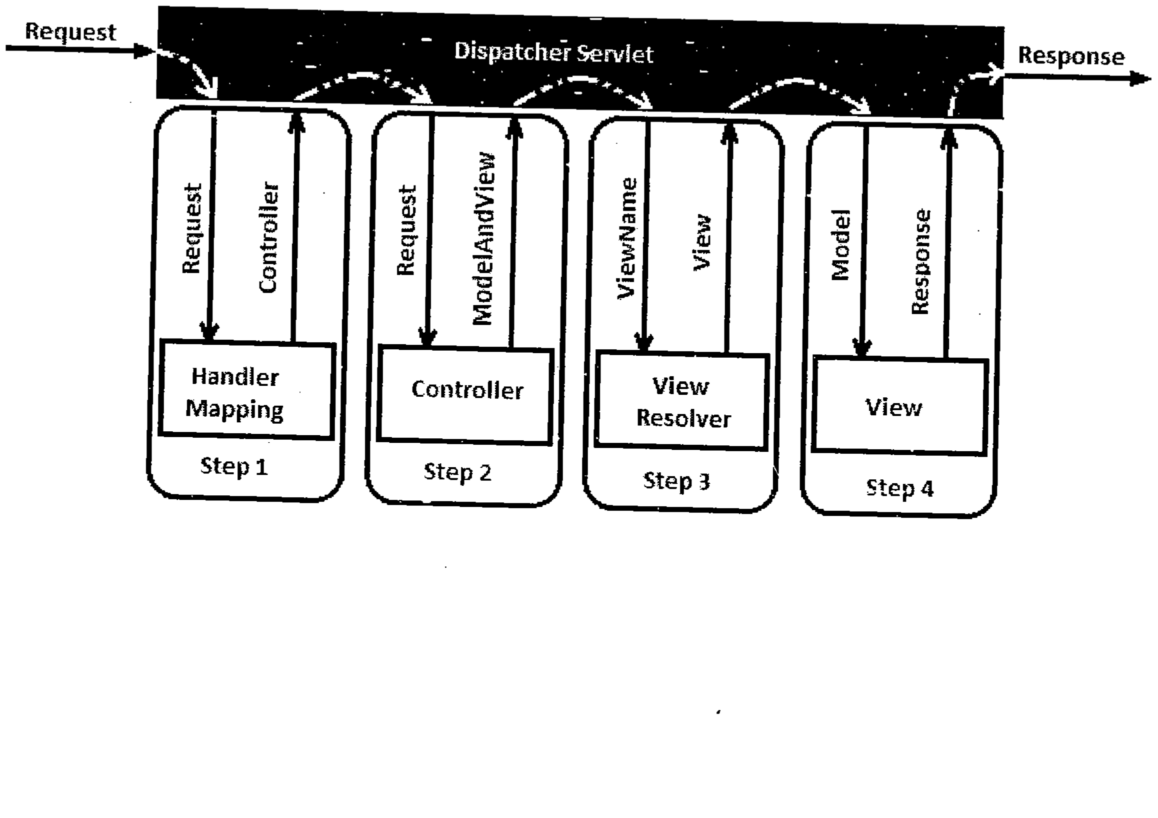
ex: Struts, JSF, Webwork, wicket,... etc.

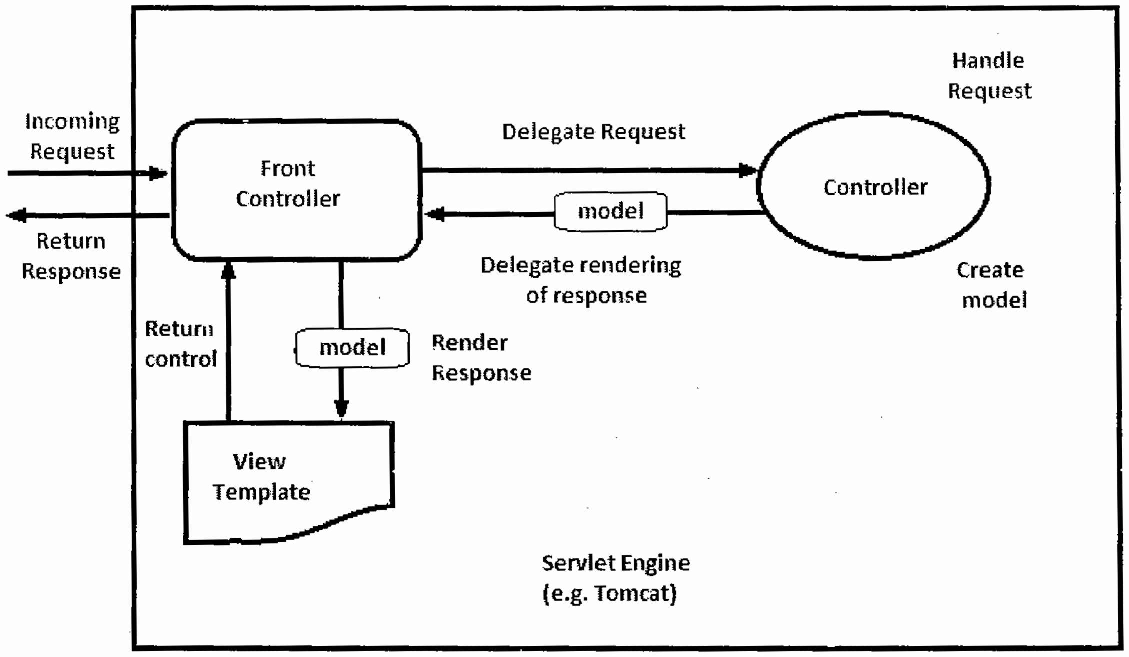
Application framework will provide an environment to design and execute distributed applications.

ex: Spring, Jboss seam.

Spring MVC Architecture


Handler Mapping
2.Request
3.Controller
1. Requst
Dispatcher Servlet
16. Response
4.handlerRequest(req,res)
controller
5.Calls Business Method
service
10.Response
11. ModelAndView
12. ViewName
13. view
View Resolver
15.Response
14.Model
View
9.Response
6.Calls Dao Method
7.Contact DataBase
DAO
Data Base
8.Response


Request
Dispatcher Servlet
Response
Request
Controller
Handler Mapping
Step 1
Request
ModelAndView
Controller
Step 2
ViewName
View
View Resolver
Step 3
Model
Response
View
Step 4



Major Components of Spring MVC

- DispatcherServlet
- HandlerMappings
- Controller
- ModelAndView
- View Resolver

The Spring MVC Workflow overview

1. The Client requests for a Resource in the web Application.
2. The Spring Front Controller i.e., DispatcherServlet makes a request to HandlerMapping to identify the particular controller for the given url.
3. HandlerMapping identifies the controller for the given request and sends to the DispatcherServlet.
4. DispatcherServlet will call the handleRequest(req,res) method on Controller. Controller is developed by writing a simple java class which implements Controller interface or extends any controller class.
5. Controller will call the business method according to bussiness requirement.
6. Service class calls the Dao class method for the business data
7. Dao interact with the DataBase to get the database data.
8. Database gives the result.
9. Dao returns the same data to service.
10. Dao given data will be processed according to the business requirements, and returns the results to Controller.
11. The Controller returns the Model and the View in the form of ModelAndView object back to the Front Controller.

12. The Front Controller i.e., DispatcherServlet then tries to resolve the actual View(which may be Jsp,Velocity or Free marker) by consulting the ViewResolver object.
13. ViewResolver selected View is rendered back to the DispatcherServlet.
14. DispatcherServlet consult the particular View with the Model.
15. View executes and returns HTML output to the DispatcherServlet.
16. DispatcherServlet sends the output to the Browser.

Note :

- We no need to develop DispatcherServlet just we have to configure in web.xml.
- We just have to configure HandlerMapping and ViewResolver in spring configuration file.
- We have to develop and configure controllers in spring configuration file.

DispatcherServlet :

The DispatcherServlet class is present org.springframework.web.servlet package and it is the child class of FrameworkServlet .DispatcherServlet follows the Front Controller Design Pattern for handling Client Requests. It means that whatever request comes from the Client, this Servlet will intercept the Client Request before passing the Request Object to the Controller.

```
javax.servlet.GenericServlet
        |
javax.servlet.http.HttpServlet
        |
org.springframework.web.servlet.HttpServletBean
        |
org.springframework.web.servlet.FrameworkServlet
        |
org.springframework.web.servlet.DispatcherServlet
```

Like Other Servlets ,The DispatcherServlet is also required to be configured in the web.xml(Web Deployment descriptor).The following code shows the declaration of DispatcherServlet in web.xml

web.xml

```
<web-app version="2.4">

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class> org.springframework.web.servlet.DispatcherServlet </servlet-class>

<load-on-startup>2</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>*.do</url-pattern>

</servlet-mapping>

</web-app>
```

Look into the definition of servlet-mapping tag. It tells that whatever be the Client Request represented by *.do meaning any Url with .do extension's are dispatched by the web container to the DispatcherServlet.

Note That as shown in the above example code it is not mandatory to configure only .do ,instead we are allowed to configure any extension.

Note :-

In Spring while initializing the Dispatcher Servlet, will looks the Spring Configuration File [servletname]-servlet.xml for web component bean declarations.Based on the above cfg ,the file name it looks for is dispatcher-servlet.xml,Now loads all the bean declarations and creates **a WebApplicationContext Container with those beans.**

We can customize the [servletname]-servlet.xml pattern by configuring a init param at the Dispatcher where the param-name is namespace (contextConfigLocation),value is the name of the configuration file shown below.

web.xml

```
<web-app version="2.4">

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class> org.springframework.web.servlet.DispatcherServlet </servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring-servlet.xml</param-value>

</init-param>

<load-on-startup>2</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>*.do</url-pattern>

</servlet-mapping>

</web-app>
```

→contextConfigLocation : speicifes the context config location. By using the this parameter name we can speicfy the multiple configuration file names also by seperating with commas(OR) spaces.

Few Points About DispactherServlet as follows

1) Dispacther Servlet act as front controller →entry point for all the Spring MVC requests
2) Loads XmlWebApplicationContext
3) Contrls the Workflow and mediates between various MVC components.

Configuring ApplicationContext :

Apart from the WebApplicationContext created by the DispatcherServlet,you can configure one

More container ApplicationContext which holds the Bussines Compoenents bean declarations.In Order to create this,you need to configure a Listener as shown Below.

web.xml

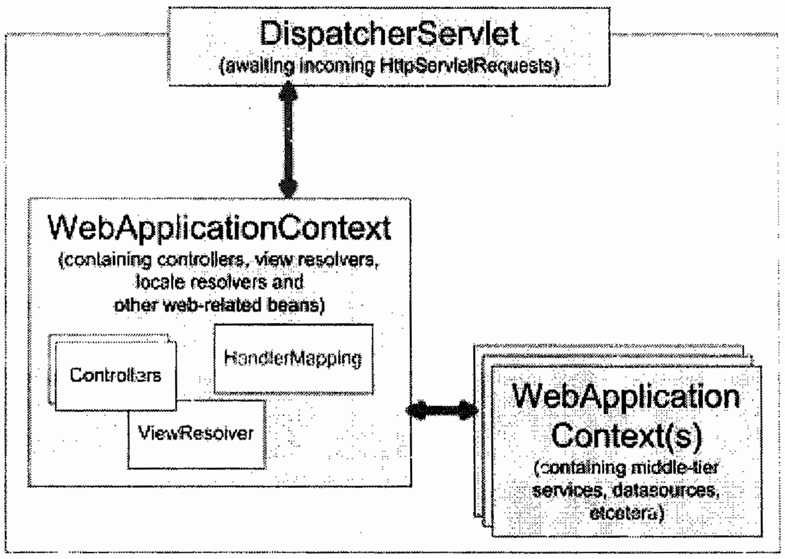
```
<web-app>
   <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/applicationContext.xml</param-value>
   </context-param>
   <servlet>
      <servlet-name>dispatcher</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
      <init-param>
         <param-name>contextConfigLocation</param-name>
         <param-value>dispatcher-servlet.xml</param-value>
      </init-param>
      <load-on-startup>2</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>dispatcher</servlet-name>
      <url-pattern>*.do</url-pattern>
   </servlet-mapping>
   <listener>
  <listener-class>
org.springframework.web.context.ContextLoaderListener
</listener-class>
   </listener>
</web-app>1
```

With the above cfg , the ContextLoaderListner will reads the context-param whose param-name is contextConfigLocation,and reads the value representing the cfg file path and creates IOC container which contains the Business
Component classes as beans

Note :-
The ApplicationContext created by ContextLoaderListener will acts a parent factory to the WebApplicationContext which means the WebComponents can
refer your business components where as business Components cannot refer our web components.

The main advantage of having two containers is the We want to quit from
Spring WebMVC ,you need to remove the DispatherServlet configuration and Your Business components still can be injected into other MVC f/w components
Through spring Integrations.

The WebApplicationContext is bound in the ServletContext, and by using static methods on the RequestContextUtils class you can always look up the WebApplicationContext if you need access to it.


DispatcherServlet
(awaiting incoming HttpServletRequests)
WebApplicationContext
(containing controllers, view resolvers, locale resolvers and other web-related beans)
Controllers
HandlerMapping
ViewResolver
WebApplication Context(s)
(containing middle-tier services, datasources, etcetera)

Diagrammatic Representation of WebApplicationContext referencing the ApplicationContext

Note the above architecture diagram. The WebApplicationContext specified in above diagram is an extension of the plain ApplicationContext with some extra feature necessary for web applications. The WebApplicationContext is capable of resolving themes and it is also associated with corresponding servlet.

HandlerMapping :-

The HandlerMapping is responsible for mapping the incoming request to controller that can handle

the request.when the DispatcherServlet receives the request it delegates the request to the Handler

Mapping.Which Identifies the appropriate HandlerExecutionChain that can handle the Request.

HandlerMapping Interface is show below:

```
public interface HandlerMapping {

 public  HandlerExecutionChain getHandler (HttpServletRequest rqst) throws Exception;

}
```

The single method getHandler(.) maps a request object to a execution

chain object of type HandlerExecutionChain. The execution chain object wraps the handler (controller).

The SpringBuilt in HandlerMapping implementations are

1)BeanNameUrlHandlerMapping

2)SimpleUrlHandlerMapping

3)ControllerClassNameHandlerMapping(spring 2.0)

1)BeanNameUrlHandlerMapping :

This is the default ***Handler Mapping*** and it is used to map the Url that comes from the Clients directly to the Bean Object. In the later section, we will see that the Bean is nothing but a Controller object. For example, consider that the following are the valid Url in a Web Application that a Client Application can request for.

http://myserver.com/eMail/showAllMails

http://myserver.com/eMail/composeMail

http://myserver.com/eMail/deleteMail

Note that the Url (excluding the Application Context) in the above cases are 'showAllMails', 'composeMail' and 'deleteMail'. This means that the Framework will look for Bean Definitions with Identifiers 'showAllMails', 'composeMail' and 'deleteMail'. Consider the following Xml code snippet in the Configuration file,

```
<beans>

 <bean name="/showAllMails.do"  class="com.nareshit.controller.ShowAllMailsController">  </bean>

   <bean name="/composeMail.do"  class="com.nareshit.controller.ComposeMailController">        </bean>

   <bean name="/ deleteMail.jsp"  class="com.nareshit.controller.DeleteMailController">

   </bean>

   </beans>
```

The BeanNameUrlHandlerMapping maps URLs to handlers based on the bean name of controllers.

The bean name of the controller should start with a leading '/'.

Actually, we don't need to define the BeanNameUrlHandlerMapping in the spring configuration, by default, if no handler mapping can be found, the DispatcherServelt will creates a BeanNameUrlHandlerMapping automatically.

SimpleUrlHandlerMapping:-

The SimpleUrlHandlerMapping provides a way to explicitly map URLs to controller beans

by direct configuration. This is done by setting property mappings.

Consider the following Configuration File,

```
<bean  class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

   <property name="mappings">

      <props>

         <prop key="/showAllMails.do">showController</prop>

         <prop key=""/composeMail.jsp">composeController</prop>

         <prop key="/deleteMail.jsp">deleteController</prop>

      </props> </property>

         </bean>
```

The set of mappings is encapsulated in the 'property' tag with each defined in a 'prop' element with the 'key' attribute being the Url, the value being the Identifier of the Controller Objects. Note that the Beans for the above Identifiers should be defined somewhere in the Configuration File.

ControllerClassNameHandlerMapping:-

The ControllerClassNameHandlerMapping maps URL to handlers based on the class name of

controllers. Controler classes should be suffixed with the word Controller, such as

in ShowAllMailsController,ComposeMailsController .

The mapping convention is that the mapped URL is derived from the uncapitalized class

name removed from the word Controller. More precisely, ComposeMailsController is

mapped to /composeMails*.

ControllerClassNameHandlerMapping needs to be configured explicitly since it is

not created by default by SpringMVC.

Consider the following Controller Definitions,

```
<bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"/>

<bean id="show" class="com.nareshit.controller.ShowAllMailsController"/>

<bean id="compose" class="com.nareshit.controller.ComposeMailsController"/>

<bean id="delete" class="com.nareshit.controller.DeleteMailsController"/>
```

/showAllMails will be handled by the ShowAllMailsController

/ composeMails will be handled by the ComposeMailsController

Note :

you can configure more than one handler mappings,based on the order(priority) will

maps the request to Handler(Controller).

Note :-

HandlerMapping can optionally contain HandlerInterceptors,the concept of Handler Interceptor will be discusses later, But the HandlerExecutionChain returned will contain the list of Handler Interceptors and

Handler for execution.

Handler Adapters

It is important to understand that the Spring Framework is so flexible enough to define what Components should be delegated the Request once the ***Dispatcher Servlet*** finds the appropriate ***Handler Mapping***. This is achieved in the form of ***Handler Adapters***. If you remember in the Spring Work flow section, that it is mentioned once the Dispatcher Servlet chooses the appropriate ***Handler Mapping***, the Request is then forwarded to the Controller object that is defined in the Configuration File. This is the default case. And this so happens because the ***Default Handler Adapter*** is ***Simple Controller Handler Adapter*** (represented by org.springframework.web.servlet.SimpleControllerHandlerAdapter), which will do the job of the Forwarding the Request from the Dispatcher to the Controller object.
Other types of ***Handler Adapters*** are ***Throwaway Controller HandlerAdapter*** (org.springframework.web.servlet.ThrowawayControllerHandlerAdapter) and ***SimpleServlet HandlerAdapter*** (org.springframework.web.servlet.SimpleServletHandlerAdapter). The ***Throwaway Controller HandlerAdapter***, for example, carries the Request from the Dispatcher Servlet to the ***Throwaway Controller*** (discussed later in the section on Controllers) and Simple Servlet Handler Adapter will carry forward the Request from the Dispatcher Servlet to a Servlet thereby making the Servlet.service() method to be invoked.

If, for example, you don't want the default ***Simple Controller Handler Adapter***, then you have to redefine the Configuration file with the similar kind of information as shown below,

```
<bean id="throwawayHandler" class = "org.springframework.web.servlet.mvc.throwaway.
ThrowawayControllerHandlerAdapter"/>
```

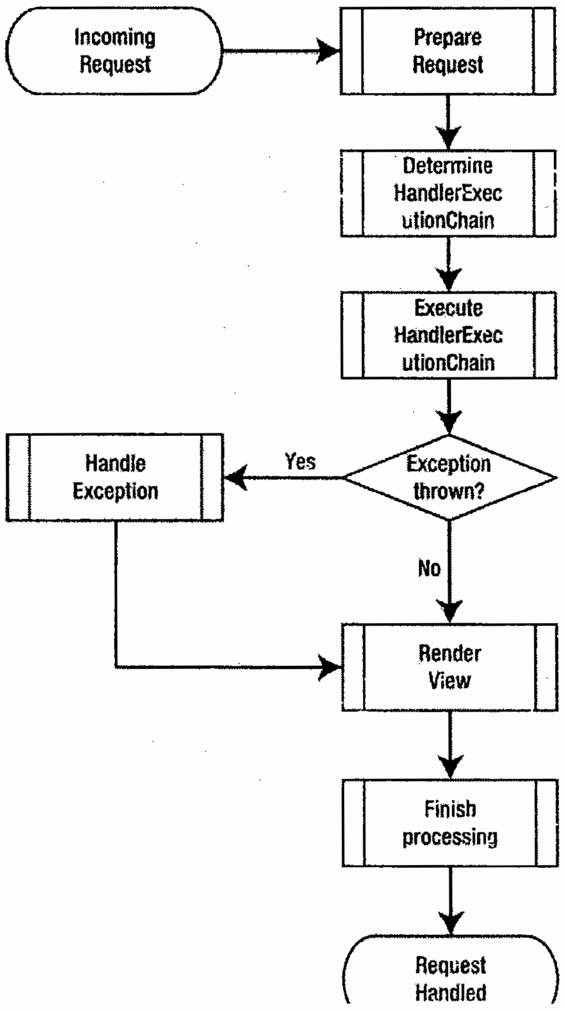
Or

```
<bean id="throwawayHandler"
class="org.springframework.web.servlet.mvc.throwaway.SimpleServletHandlerAdapter"/>
```

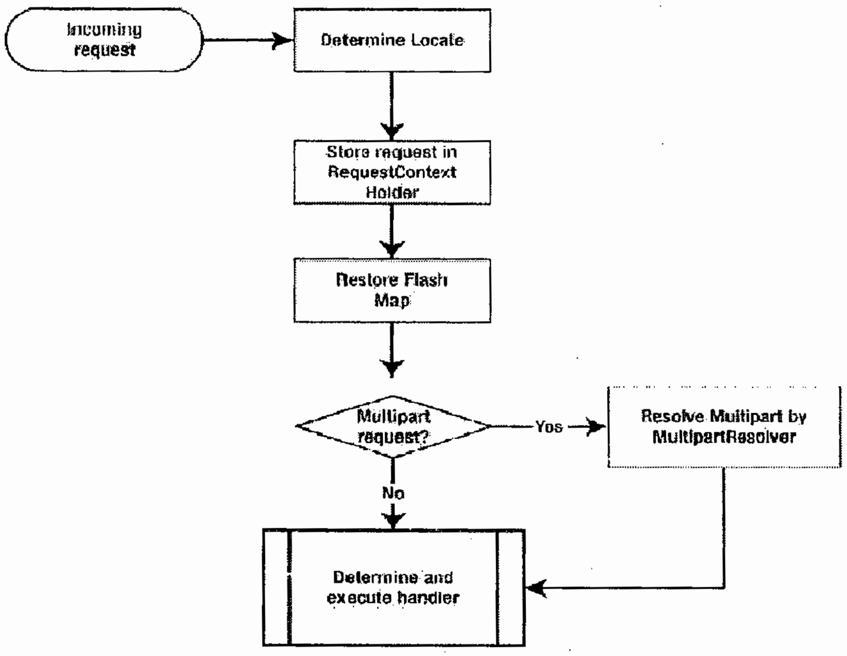
Even, it is possible to write a ***Custom Handler Adapter*** by implementing the HandlerAdapter interface available in the org.springframework.web.servlet package.

DispatcherServlet Internal WorkFlow:-

The figure below shows a more complete view of the request processing workflow, inside the `DispatcherServlet`.



Below is the flow diagram of the preprocessing of the request:


Incoming request
Determine Locale
Store request in RequestContext Holder
Restore Flash Map
Multipart request?
Yes
Resolve Multipart by MultipartResolver
No
Determine and execute handler

Before the `DispatcherServlet` will start dispatching and handling the request, it first does some preparation and preprocessing of the request, these are:

- exposing the current `java.util.Locale` of the request, using the `org.springframework.web.servlet.LocaleResolver`
- exposing the current request in `org.springframework.web.context.request.RequestContextHolder` (give code easy access to the request)
- construct a `org.springframework.web.servlet.FlashMap`, resolved by the `org.springframework.web.servlet.FlashMapManager`. This map contains attributes stored explicitly in the previous request.
- checking whether the request is a multipart HTTP request. If so, the request is wrapped in an `org.springframework.web.multipart.MultipartHttpServletRequest`, using `org.springframework.web.multipart.MultipartResolver`.

After all of these, the requset is ready to be dispatched to the correct handler.

Determine the HandleExecution Chain

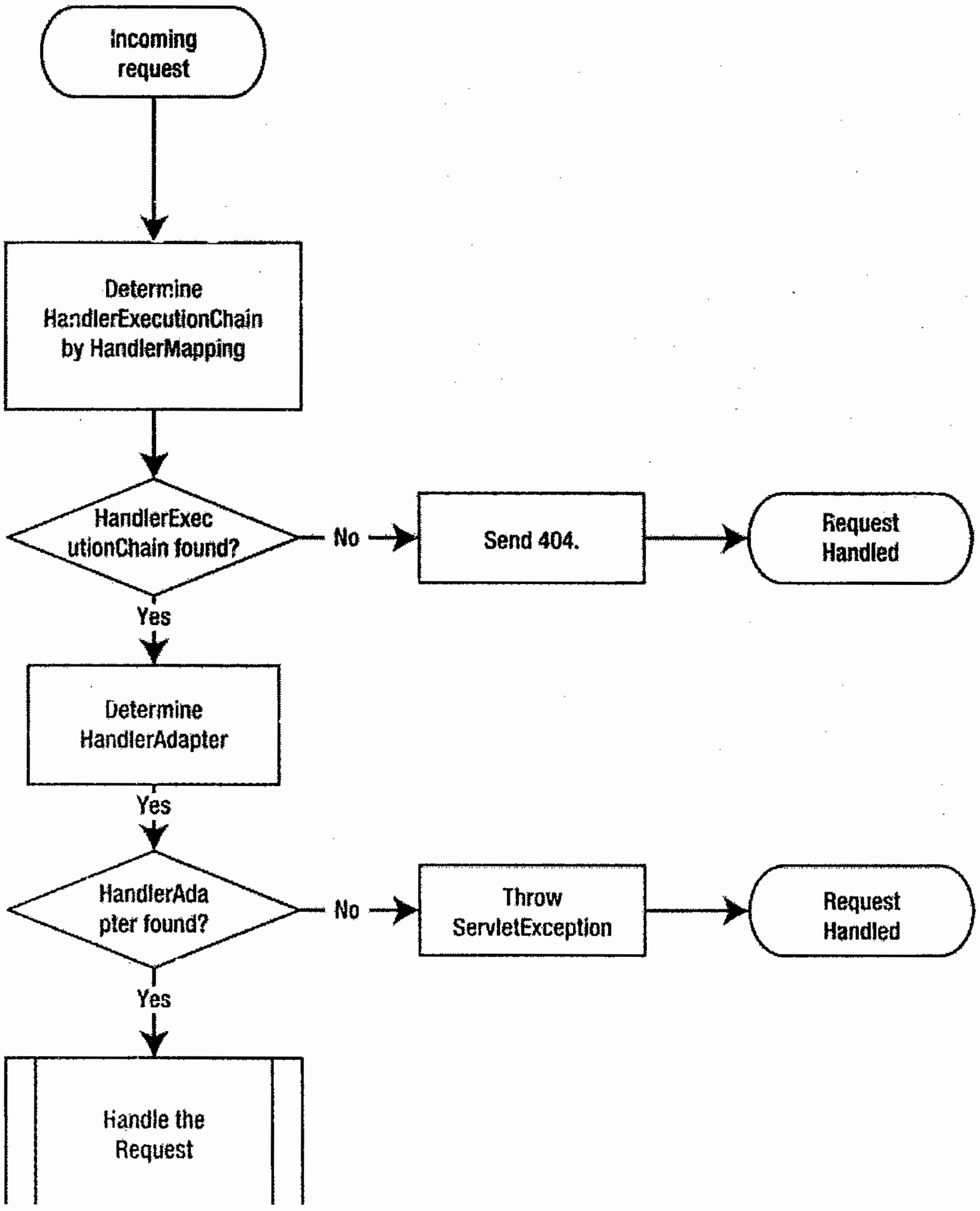
A couple of components are involved in dispatching the request. When a request is for dispatching, the `DispatcherServlet` will consult one or more `org.springframework.web.servlet.HandlerMapping` implementations to determine which handler can handle the request.

If no `HandlerMapping` is found, an HTTP 404 response is sent back to the client.

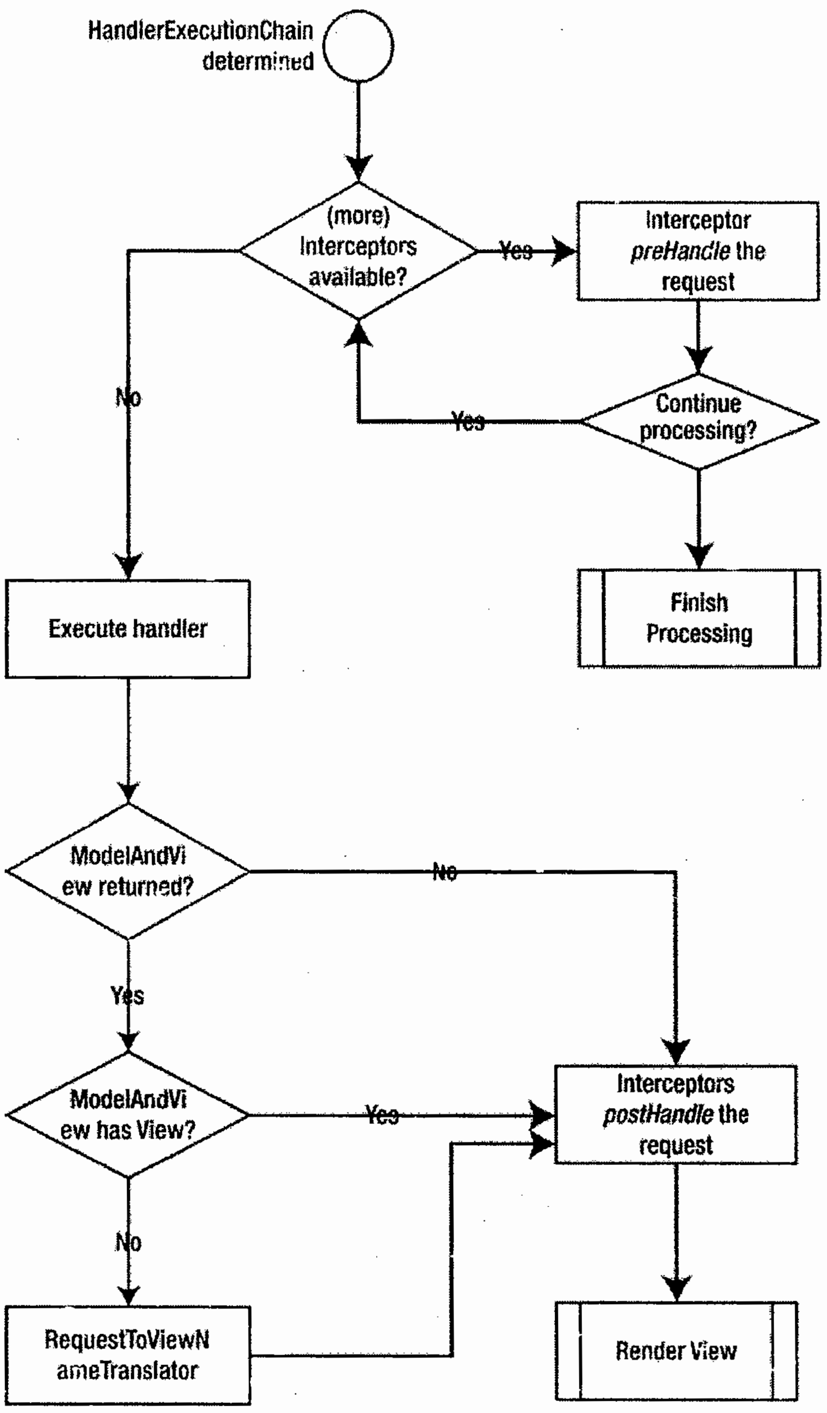
The `HandlerMapping` returns : HandlerExecutionChain

When the HandlerExecutionChain has been determined the servlet will attempt to find an

`org.springframework.web.servlet.HandlerAdapter` to actually execute the found handler. If no suitable `HandleAdapter` can be found, a `javax.servlet.ServletException` is thrown.



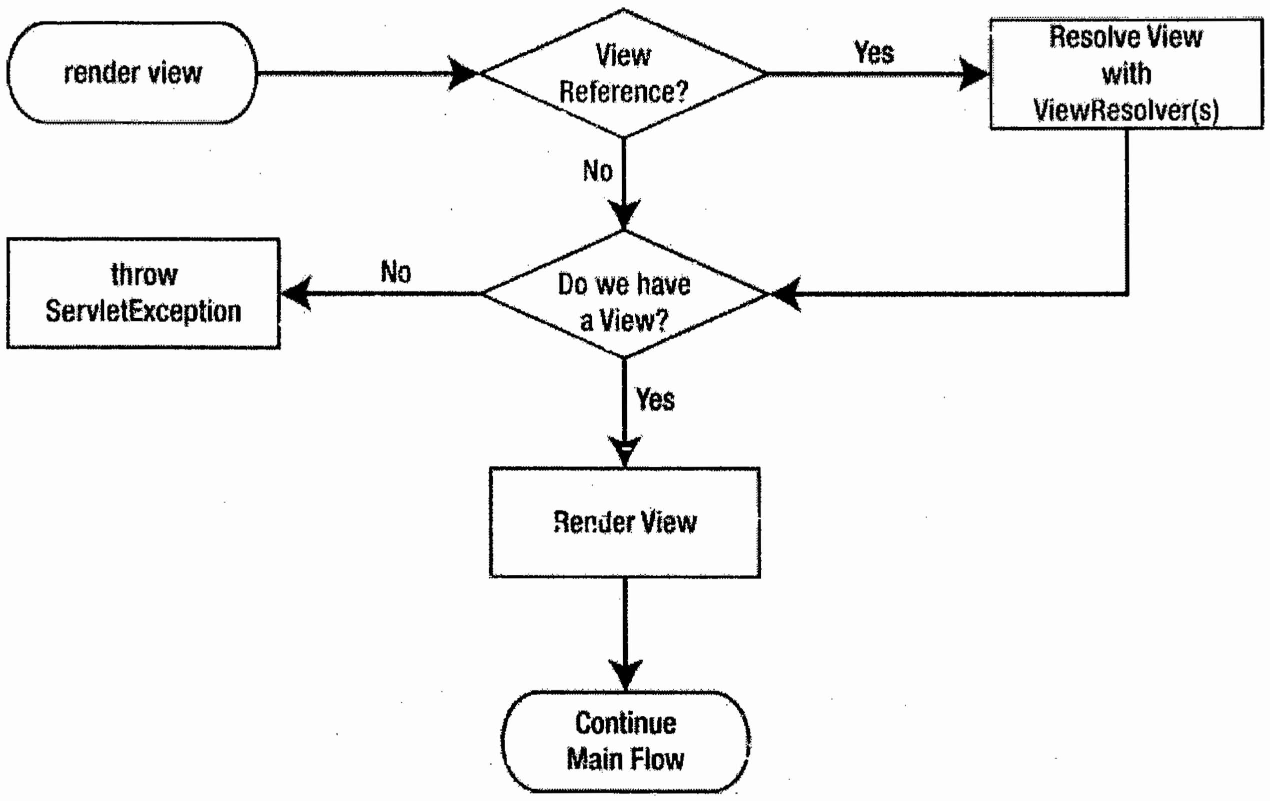
The Below Diagrams Shows the Execution of the HandlerExecutionChain :-


HandlerExecutionChain determined
(more) Interceptors available?
Yes
Interceptor preHandle the request
Continue processing?
Yes
No
Finish Processing
Execute handler
ModelAndView returned?
No
Yes
ModelAndView has View?
Yes
Interceptors postHandle the request
No
RequestToViewNameTranslator
Render View

The `DispatcherServlet` uses the `HandlerExecutionChain` to determine what to execute.
This class holds a reference to the actual handler that needs to be invoked.
It also (optionally) references `org.springframework.web.servlet.HandlerInterceptor` implementations that executed before (`preHandler` method) and after (`postHandler` method) the execution of the hander.

These interceptors can be used to apply crosscutting functionality.If the code executes successfully, the interceptors are called again; and finally, when needed, the view is rendered.

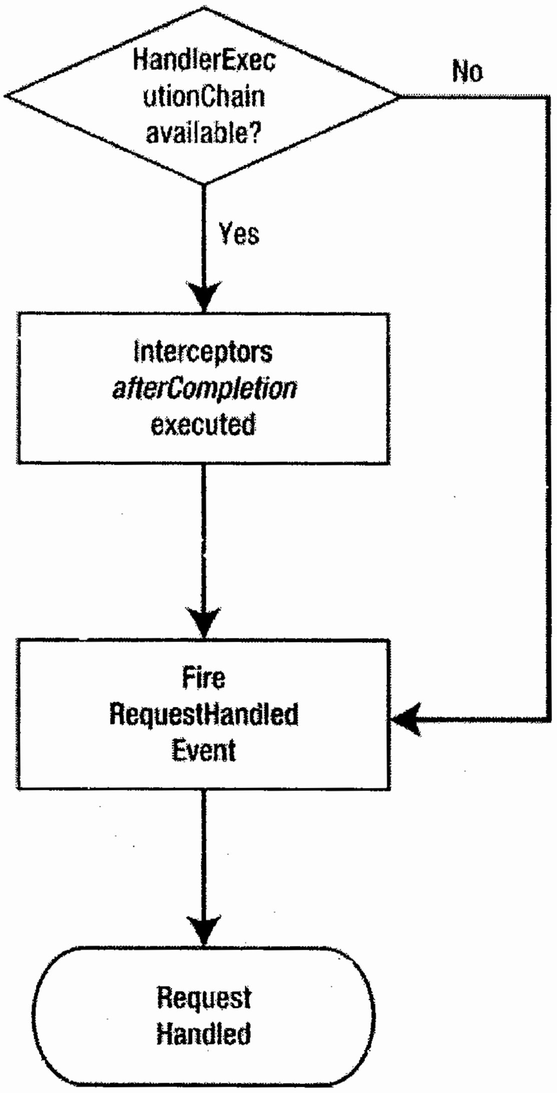
Render a view Work Flow :



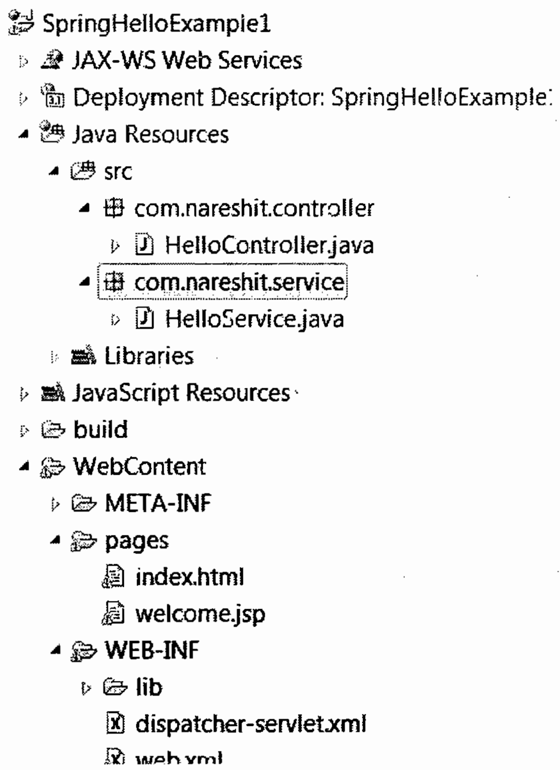
Finish the Processing

Each incoming request passes through this step, regardless of whether there are exceptions.

- execute `afterCompletion` method (only the interceptors where `preHandler` method was successfully invoked)
- execute the interceptors in the reverse order that the 'preHandler' method was called
- the `DispatcherServlet` uses the event mechanism in the Spring Framework to fire an `org.springframework.web.context.support.RequestHandledEvent`. You could create and configured an `org.springframework.web.context.ApplicationListener` to receive and log these events.


HandlerExecutionChain available?
No
Yes
Interceptors *afterCompletion* executed
Fire RequestHandled Event
Request Handled

Let us create a simple Hello application in Spring MVC



```
SpringHelloExample1
  JAX-WS Web Services
  Deployment Descriptor: SpringHelloExample:
  Java Resources
    src
      com.nareshit.controller
        HelloController.java
      com.nareshit.service
        HelloService.java
    Libraries
  JavaScript Resources
  build
  WebContent
    META-INF
    pages
      index.html
      welcome.jsp
    WEB-INF
      lib
      dispatcher-servlet.xml
      web.xml
```

index.html

```
<html>
<head><h1><center>SpringMVCExample</center></h1></head><br/><br/>
<hr/>
<body bgcolor="pink">
<form action="hello.do" method="post">
Name :<input type="text" name="name"/>
<input type="submit" value="Get"/>
</form>
</body></html>
```

welcome.jsp

```
<html>
<head><h1><center>SpringMVCExample</center></h1></head><br/><br/>
<hr/>
<body bgcolor="pink">
${message}
</body></html>
```

web.xml

```
<web-app>
<servlet>
<servlet-name>dispatcher</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>dispatcher</servlet-name>
<url-pattern>*.do</url-pattern>
</servlet-mapping>
<welcome-file-list>
<welcome-file>/pages/index.html</welcome-file>
</welcome-file-list>
</web-app>
```

dispatcher-servlet.xml

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                              "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
<bean id="handlerMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
<property name="mappings">
<props>
 <prop key="/hello.do">helloController</prop>
</props>
</property>
</bean>
<bean id="helloController" class="com.nareshit.controller.HelloController">
<property name="helloService" ref="helloService"/>
</bean>
<bean id="helloService" class="com.nareshit.service.HelloService">
</bean>
</beans>
```

HelloController.java

```
package com.nareshit.controller;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
import com.nareshit.service.HelloService;
public class HelloController implements Controller{
private HelloService helloService;
private String message;
private String targetView="/pages/welcome.jsp";
        public ModelAndView handleRequest(HttpServletRequest request,
                        HttpServletResponse response) throws Exception {
                String name=request.getParameter("name");
                if(name!=null && name.trim().length()>0){
                        message=helloService.sayHello(name);
                }
                else{
                        message=helloService.sayHello("Guest");
                }
ModelAndView modelAndView=
new ModelAndView(targetView,"message",message);
                return modelAndView;
        }
        public void setHelloService(HelloService helloService) {
                this.helloService = helloService;
        }

}
```

HelloService.java

```
package com.nareshit.service;
public class HelloService {
   public String sayHello(String name) {
                String message="Hello : "+name+" Welcome to Spring MVC";
                return message;
        }

}
```

ModelAndView:-

ModelAndView is a class present in org.springframework.web.servlet. it's job is returned by the controller back to the DispatcherServlet . This class is just a container class for holding the Model and the View information.

Example:-

ModelAndView mv=new ModelAndView("targetView","message",message);

Controller:-

Controllers are Components that are being called by the Dispatcher Servlet .

The notion of a controller is part of the MVC design pattern (more specifically, it is the *'C'* in MVC). Controller

Handles the incoming request maps the request data to object and sends that data to the Bussiness tier(typically a Service class) ciasses to process the request.

Spring contains a wide variety of controllers like form-specific controllers, command-based controllers, and wizard-style logic Controllers.

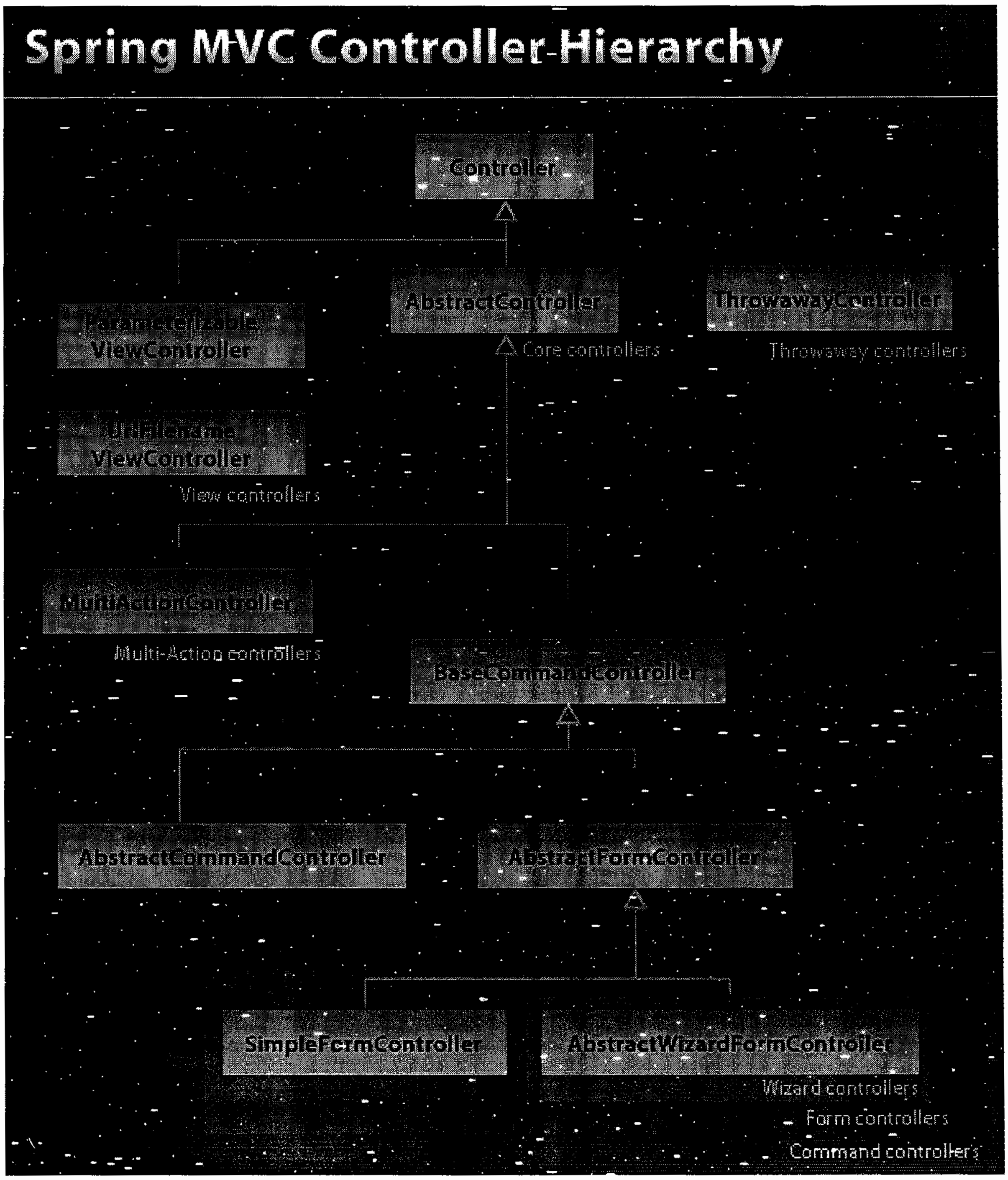
All controller Components in Spring implement the org.springframework.web.servlet.mvc.Controller interface.

The Controller interface source code is listed below.

```
public interface Controller {

   /**
    * Process the request and return a ModelAndView object which the DispatcherServlet
    * will render.
    */
   ModelAndView handleRequest(HttpServletRequest request,
     HttpServletResponse response) throws Exception;

}
```

As you can see, the Controller interface defines a single method that is responsible for handling a request and returning an appropriate model and view.

Following diagram shows the Controllers hierarchy in Spring MVC:



AbstractController:-

AbstractController is responsible for providing basic services applicable to any kind of HTTP request. When using the AbstractController as the baseclass for your controllers you have to override the handleRequestInternal(HttpServletRequest, HttpServletResponse) method, implement your logic, and return a ModelAndView object.

Here is short example consisting of a class and a declaration in the web application context.

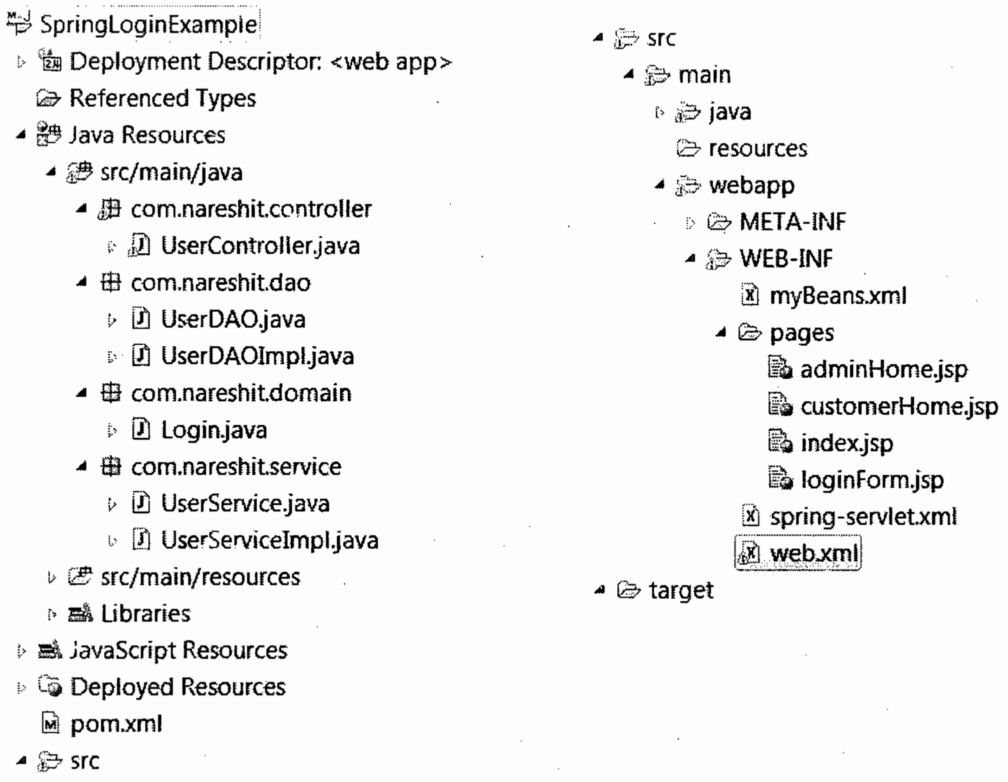
```
package com.nareshit.controller;
public class HelloController extends AbstractController {
public ModelAndView handleRequestInternal( HttpServletRequest request,  HttpServletResponse response) throws
Exception {
ModelAndView modelAndView = new ModelAndView("/pages/hello.jsp");
     mav.addObject("message", "welcome to Spring MVC");
     return modelAndView;
```

```
   }
}
```

In spring configuration file i,e in dispatcher-servlet.xml

```
<bean name="/hello.do" class="com.nareshit.controller.HelloController">
  </bean>
```

Let us create a simple Login application with Spring MVC Abstract Controller



```
SpringLoginExample
  Deployment Descriptor: <web app>
  Referenced Types
  Java Resources
    src/main/java
      com.nareshit.controller
        UserController.java
      com.nareshit.dao
        UserDAO.java
        UserDAOImpl.java
      com.nareshit.domain
        Login.java
      com.nareshit.service
        UserService.java
        UserServiceImpl.java
    src/main/resources
    Libraries
  JavaScript Resources
  Deployed Resources
  pom.xml
  src
    main
      java
      resources
      webapp
        META-INF
        WEB-INF
          myBeans.xml
          pages
            adminHome.jsp
            customerHome.jsp
            index.jsp
            loginForm.jsp
          spring-servlet.xml
          web.xml
  target
```

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.nareshit</groupId>
  <artifactId>SpringLoginExample</artifactId>
  <packaging>war</packaging>
  <version>1.0</version>
  <name>SpringLoginExample Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
   <dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-context</artifactId>
   <version>3.0.1.RELEASE</version>
   </dependency>
   <dependency>
   <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>3.0.1.RELEASE</version>
  </dependency>
  <dependency>
  <groupId>org.springframework</groupId>
 <artifactId>spring-orm</artifactId>
 <version>3.0.1.RELEASE</version>
  </dependency>
  <dependency>
  <groupId>org.springframework</groupId>
 <artifactId>spring-web</artifactId>
 <version>3.0.1.RELEASE</version>
  </dependency>
  <dependency>
  <groupId>org.springframework</groupId>
 <artifactId>spring-webMvc</artifactId>
 <version>3.0.1.RELEASE</version>
  </dependency>
  <dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>3.6.10.FINAL</version>
  </dependency>
  <dependency>
  <groupId>javassist</groupId>
 <artifactId>javassist</artifactId>
 <version>3.12.1.GA</version>
  </dependency>
  <dependency>
     <groupId>javax.servlet</groupId>
   <artifactId>javax.servlet-api</artifactId>
   <version>3.0.1</version>
  </dependency>
  <dependency>
  <groupId>javax.servlet</groupId>
 <artifactId>jstl</artifactId>
 <version>1.2</version>
  </dependency>
  <dependency>
  <groupId>oracle.jdbc.driver</groupId>
  <artifactId>ojdbc14</artifactId>
  <version>10.2.0</version>
  </dependency>

 </dependencies>
 <build>
   <finalName>SpringLoginExample</finalName>
 </build>
</project>
```

index.jsp

```
<h1>Welcome to MySite</h1>
<br/><hr/><h2><a href="showLoginForm.do">Login</a></h2>
```

loginForm.jsp

```
<%@page isELIgnored="false" %>
<html>
<head> <h1 align="center">LoginForm</head>
<br/><hr/>
${msg}
<body>
<div align="center">
<form action="login.do" method="post">
```

```
<div>
UserName :<input type="text" name="userName" placeholder="User Name">
</div>
<div>
Password :<input type="password" name="password" placeholder="Password">
</div>
&nbsp;
<div>
<input type="submit" value="Login"/>
</div>
</form>
</div>
</body>
</html>
```

web.xml

```
<web-app>
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>/WEB-INF/myBeans.xml</param-value>
</context-param>
 <servlet>
 <servlet-name>ds</servlet-name>
 <servlet-class>org.springframework.web.servlet.DispatcherServlet
 </servlet-class>
 <load-on-startup>1</load-on-startup>
 <init-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>/WEB-INF/spring-servlet.xml
 </param-value>
 </init-param>
 </servlet>
<servlet-mapping>
<servlet-name>ds</servlet-name>
<url-pattern>*.do</url-pattern>
</servlet-mapping>
<listener>
<listener-class>org.springframework.web.context.ContextLoaderListener
</listener-class>
</listener>
<welcome-file-list>
<welcome-file>/WEB-INF/pages/index.jsp
</welcome-file>
</welcome-file-list>
</web-app>
```

spring-servlet.xml

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                              "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
<bean id="hm"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
<property name="mappings">
<props>
<prop key="/login.do">userController</prop>
<prop key="/showLoginForm.do">showLoginFormController</prop>
</props>
</property>
</bean>
<bean id="showLoginFormController" class="org.springframework.web.servlet.mvc.ParameterizableViewController">
<property name="viewName" value="/WEB-INF/pages/loginForm.jsp"/>
</bean>
<bean id="userController" class="com.nareshit.controller.UserController">
<property name="userService" ref="userService"/></bean>
```

```
</beans>
```

myBeans.xml

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                    "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName"
value="oracle.jdbc.driver.OracleDriver"/>
<property name="url" value="jdbc:oracle:thin:@localhost:1521:XE"/>
<property name="username" value="system"/>
<property name="password" value="manager"/>
</bean>
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
<property name="dataSource" ref="dataSource"/>
<property name="hibernateProperties">
<props>
<prop key="hibernate.show_sql">true</prop>
<prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect
</prop>
</props>
</property>
<property name="annotatedClasses">
<value>com.nareshit.domain.Login</value>
</property>
</bean>
<bean id="userDao" class="com.nareshit.dao.UserDAOImpl">
<property name="sessionFactory" ref="sessionFactory"/>
</bean>
<bean id="userService"
class="com.nareshit.service.UserServiceImpl">
<property name="userDao" ref="userDao"/>
</bean>
</beans>
```

UserController.java

```
package com.nareshit.controller;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;
import com.nareshit.domain.Login;
import com.nareshit.service.UserService;
public class UserController extends AbstractController {
private UserService userService;
public void setUserService(UserService userService) {
        this.userService = userService;
}
protected ModelAndView handleRequestInternal(HttpServletRequest req,HttpServletResponse res)throws Exception {
        String userName=req.getParameter("userName");
        String password=req.getParameter("password");
        String targetView="/WEB-INF/pages/loginForm.jsp";
        String message="Invalid UserName (OR) Password";
        if(userName!=null && userName.trim().length()>0 && password!=null && password.trim().length()>0){
        Login login=new Login();
        login.setUserName(userName);
        login.setPassword(password);
        login=userService.login(login);
        if(login!=null && login.getUserRole()!=null){
                message="welcome,"+userName;
        if(login.getUserRole().equals("admin")){
        targetView="/WEB-INF/pages/adminHome.jsp";
        }
        else if(login.getUserRole().equals("customer")){
        targetView="/WEB-INF/pages/customerHome.jsp";
        }
        }
        }
        else{
        message="UserName And Password Can not be Empty";
        }
        return new ModelAndView(targetView,"msg",message);
}

}
```

Login.java

```
package com.nareshit.domain;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name="Login_Details")
public class Login {
@Id
private int userId;
private String userName;
private String password;
private String userRole;
public int getUserId() {
        return userId;
}
public void setUserId(int userId) {
        this.userId = userId;
}
public String getUserName() {
        return userName;
}
public void setUserName(String userName) {
        this.userName = userName;
}
public String getPassword() {
        return password;
}
public void setPassword(String password) {
        this.password = password;
}
public String getUserRole() {
        return userRole;
}
public void setUserRole(String userRole) {
        this.userRole = userRole;
}
}
```

UserService.java

```
package com.nareshit.service;
import com.nareshit.domain.Login;
public interface UserService {
public Login login(Login login);
}
```

UserServiceImpl.java

```
package com.nareshit.service;
import com.nareshit.dao.UserDAO;
import com.nareshit.domain.Login;
```

```
public class UserServiceImpl implements UserService {
private UserDAO userDao;
        public Login login(Login login) {
                login=userDao.login(login);
                return login;
        }
        public void setUserDao(UserDAO userDao) {
                this.userDao = userDao;
        }
}
```

UserDAO.java

```
package com.nareshit.dao;
import com.nareshit.domain.*;
public interface UserDAO {
public Login login(Login login);
}
```

UserDAOImpl.java

```
package com.nareshit.dao;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import com.nareshit.domain.Login;
public class UserDAOImpl implements UserDAO{
private SessionFactory sessionFactory;
        public Login login(Login login) {
    if(login!=null){
 Session session=sessionFactory.openSession();
 if(session!=null){
String hql="select login.userId,login.userRole from com.nareshit.domain.Login as login where login.userName=? and
login.password=?";
Query query=session.createQuery(hql);
 query.setParameter(0,login.getUserName());
         query.setParameter(1, login.getPassword());
Object[] obj=(Object[])query.uniqueResult();
         if(obj!=null){
                 login.setPassword(null);
                 if(obj.length>0){
  Integer userId=(Integer)obj[0];
         String userRole=(String)obj[1];
        login.setUserId(userId);
        login.setUserRole(userRole);
                 }
         }
 }
  }
                return login;
        }
        public void setSessionFactory(SessionFactory sessionFactory) {
                this.sessionFactory = sessionFactory;
        }

}
```

adminHome.jsp

```
<%@page isELIgnored="false" %>
<html><head><h1 align="center">
AdminHome
</h1></head>
<br/>${msg}
<br/><hr/>
</html>
```

customerHome.jsp

```
<%@page isELIgnored="false" %>
<html><head><h1 align="center">
CustomerHome</h1></head>
<br/>${msg}<br/><hr/></html>
```

ParameterizableViewController:

The ParameterizableViewController is a subclass of AbstractController, and return a ModelAndView based on the viewName property, it's purely a redirect class, nothing more, nothing less

The ParametrizableViewController class we can declare in spring cfg file as follows

```
  <bean name="/showLoginFormPage.do"
        class="org.springframework.web.servlet.mvc.ParameterizableViewController">
            <property name="viewName" value="/WEB-INF/pages/loginForm.jsp" />
  </bean>

</beans>
```

View Resolvers:-

A ViewResolver is a strategy and factory object used to map logical view names to actual view resources.
The interface for ViewResolver is shown below.

```
public interface ViewResolver {
  View resolveViewName(String name, Locale locale);
}
```

The single method resolveViewName(..) maps a logical view name together with the request locale to a View instance.

The most commonly used spring provided built-in VewResolver implementations are listed below:

InternalResourceViewResolver
URLBasedViewResolver
ResourceBundleViewResolver
XmlViewResolver
BeanNameViewResolver

InternalResourceViewResolver :
Convenient subclass of UrlBasedViewResolver that supports InternalResourceView (i.e. Servlets and JSPs) and subclasses such as JstlView and TilesView.

In Spring MVC, **InternalResourceViewResolver** is used to resolve "internal resource view" (in simple, it's final output, jsp or html page) based on a predefined URL pattern. In additional, it allow you to add some predefined prefix or suffix to the view name (prefix + view name + suffix), and generate the final view page URL.

Example :
```
<beans ... >

  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
   <property name="prefix" value="/WEB-INF/pages/" />
   <property name="suffix" value=".jsp" />
  </bean>

</beans>
```

URLBasedViewResolver:-

This viewResolver will maps the view name to a URL and handover's it to the Dispatcher Servlet to render the view.

Example :-

```
<bean id="urlBasedViewReslover" class="org.springframework.web.servlet.view.UrlBasedViewReslover">
<property name="prefix" value="/WEB-INF/JSP/"/>
<property name="suffix" value=".jsp"/>
<property name="viewClass">
<value type="java.lang.Class">org.springframework.web.servlet.view.InternalResourceView</value>
</property>
</bean>
(OR)
<bean id="urlBasedViewReslover" class="org.springframework.web.servlet.view.UrlBasedViewReslover">
<property name="prefix" value="/WEB-INF/JSP/"/>
<property name="suffix" value=".jsp"/>
<property name="viewClass">
<value type="java.lang.Class">org.springframework.web.servlet.view.JstlView</value>
</property>
</bean>
```

ResourceBundleViewResolver :

when working with different view technologies in a web application,you can use ResourceBundle ViewResolver. This view resolver class is used to reslove the
view based on viewname specified in the properties file.

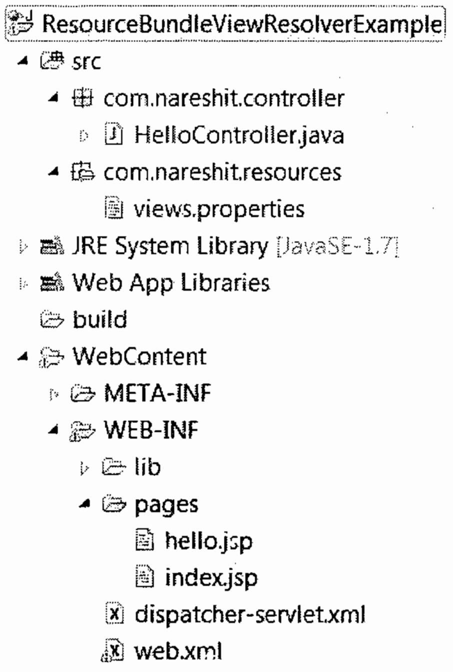
```
<beans....>
<bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
<property name="basename" value="views" >
</property>
</bean>
</beans>
```

With the above declaration all the view definitions have been declared in the file view.properties under your classpath(src directory).

views.properties

```
hello.(class)=org.springframework.web.servlet.view.JstlView
hello.url=/WEB-INF/pages/hello.jsp
```

the mapping information in the properties file would be like[viewName].class and
[viewname].url,where [viewname].class represents the Class that acts
as view class to render the view and the [viewname].url represents the path
to the view located in the application.



```
ResourceBundleViewResolverExample
  src
    com.nareshit.controller
      HelloController.java
    com.nareshit.resources
      views.properties
  JRE System Library [JavaSE-1.7]
  Web App Libraries
  build
  WebContent
    META-INF
    WEB-INF
      lib
      pages
        hello.jsp
        index.jsp
      dispatcher-servlet.xml
      web.xml
```

index.jsp

```
<html><head>ResourceBundleViewResolverExample</head>
<a href="helloController.do">click here</a>
</html>
```

hello.jsp

```
<h3>${msg}</h3>
```

web.xml

```
<web-app>
<servlet>
<servlet-name>dispatcher</servlet-name>
<servlet-class>
org.springframework.web.servlet.DispatcherServlet
</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>dispatcher</servlet-name>
<url-pattern>*.do</url-pattern>
</servlet-mapping>
<welcome-file-list>
<welcome-file>/WEB-INF/pages/index.jsp
</welcome-file>
</welcome-file-list>
</web-app>
```

dispatcher-servlet.xml

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                        "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
<bean
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
<property name="mappings">
<props>
<prop key="/helloController.do">helloController</prop>
</props>
</property>
</bean>
<bean id="helloController"
```

```
class="com.nareshit.controller.HelloController">
</bean>
<bean id="viewResolver"
class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
<property name="basename"
value="com/nareshit/resources/views" />
</bean>
</beans>
```

views.properties

```
hello.(class)=org.springframework.web.servlet.view.JstlView

hello.url=/WEB-INF/pages/hello.jsp
```

HelloController.java

```
package com.nareshit.controller;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;
public class HelloController  extends AbstractController{
        protected ModelAndView handleRequestInternal(HttpServletRequest request,
                        HttpServletResponse response) throws Exception {
        String message="Welcome to Spring ,this is ResourceBundleViewResolver Example";
                return new ModelAndView("hello","msg",message);
        }
}
```

XmlViewReslover:-

Instead of providing the mapping in a properties file we can declare in a xml file which contains bean definations of the views that has to be render.

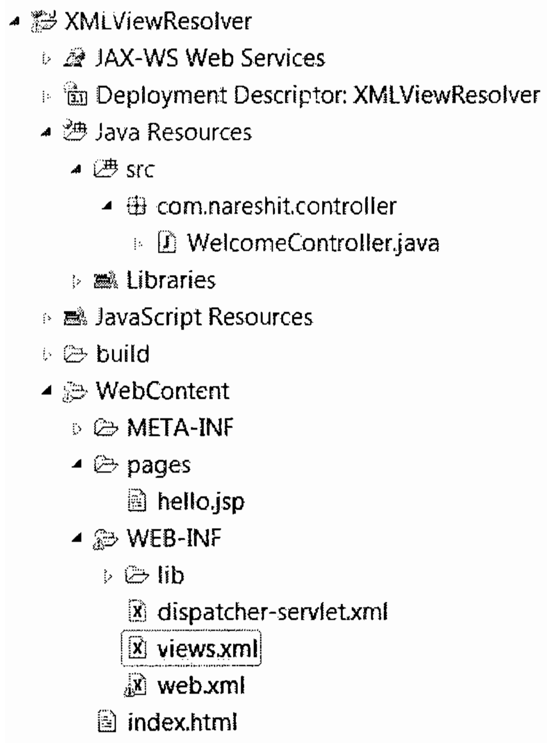
xmlViewResolver class is used to reslove the view based on view beans in the XML file,

By-default, XmlViewResolver will loads the views. xml file from /WEB-INF/views.xml

however, this location can be overridden through the location property of XmlViewResolver.

Example :

```
<beans....>
<bean class="org.springframework.web.servlet.view.XmlViewResolver">
<property name="location">
<value>/WEB-INF/spring-views.xml</value>
</bean>
</beans>
```


```
XMLViewResolver
  JAX-WS Web Services
  Deployment Descriptor: XMLViewResolver
  Java Resources
    src
      com.nareshit.controller
        WelcomeController.java
    Libraries
  JavaScript Resources
  build
  WebContent
    META-INF
    pages
      hello.jsp
    WEB-INF
      lib
      dispatcher-servlet.xml
      views.xml
      web.xml
    index.html
```

index.html

```
Welcome to Spring MVC
<a href="welcome.do">click</a>
```

web.xml

```
<web-app>
<servlet>
  <servlet-name>dispatcher</servlet-name>
   <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
<load-on-startup>2</load-on-startup>
</servlet>
   <servlet-mapping>
   <servlet-name>dispatcher</servlet-name>
   <url-pattern>*.do</url-pattern>
   </servlet-mapping>
      <welcome-file-list>
  <welcome-file>/index.html</welcome-file>
   </welcome-file-list>
</web-app>
```

dispatcher-servlet.xml

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                              "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
<bean name="/welcome.do"
class="com.nareshit.controller.WelcomeController" />
 <bean class=
 "org.springframework.web.servlet.view.XmlViewResolver">
<property name="location">
 <value>/WEB-INF/views.xml</value>
  </property>
  <property name="order" value="1" />
</bean>
</beans>
```

views.xml

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                              "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
<bean id="hello"
 class="org.springframework.web.servlet.view.JstlView">
<property name="url" value="/pages/hello.jsp" />
</bean>
</beans>
```

welcomeController.java

```
package com.nareshit.controller;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;
public class WelcomeController extends AbstractController {
protected ModelAndView handleRequestInternal(HttpServletRequest req,
                              HttpServletResponse res) throws Exception {
                String msg="welcome to Spring MVC";
                String viewName="hello";
                return new ModelAndView(viewName,"msg",msg);
        }
}
```

hello.jsp

```
<html>
        <body>
          <h1>Spring XMLView Resolver example</h1>
           <h3> ${msg}</h3>
        </body>
        </html>
```

Note:-
if multiple view resolver strategies are applied,then we can declare the priority order through "order" property. the lower order value has a higher priority.

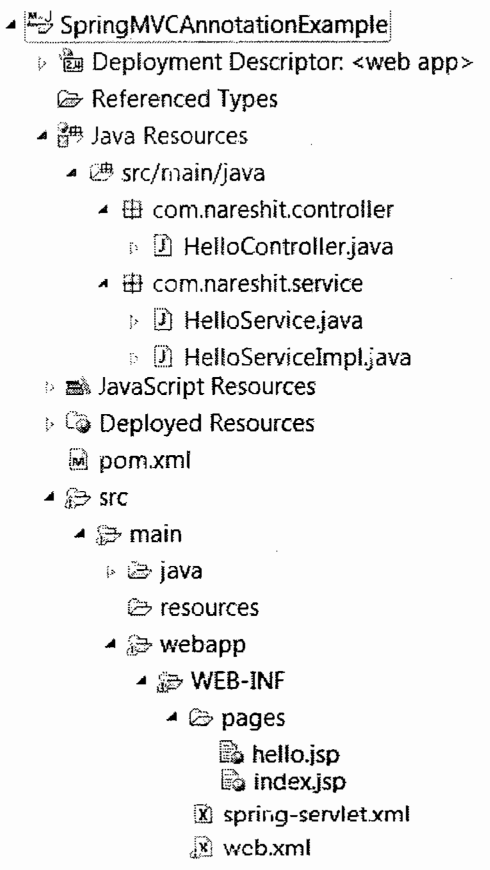
Spring Mvc Annotations:

@Controller annotation is used make a simple java class as a spring handler/controller class.
@Controller Annotation is a component type annotation.

@RequestMapping: this annotation is used to map an incoming request with a handler/handler method.
@RequestParam : Annotation which indicates that a method parameter should be bound to a web request parameter.

That means this annotation is used to gather the request parameter from request obj.

@ModelAttribute:-
This annotation can be used as the method arguments or before the method declaration. The primary objective of this annotation to bind the request parameters or form fields to an model object. The model object can be formed using the request parameters or already stored in the session object.



```
SpringMVCAnnotationExample
  Deployment Descriptor: <web app>
  Referenced Types
  Java Resources
    src/main/java
      com.nareshit.controller
        HelloController.java
      com.nareshit.service
        HelloService.java
        HelloServiceImpl.java
  JavaScript Resources
  Deployed Resources
  pom.xml
  src
    main
      java
      resources
      webapp
        WEB-INF
          pages
            hello.jsp
            index.jsp
          spring-servlet.xml
          web.xml
```

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.nareshit</groupId>
  <artifactId>SpringMVCAnnotationExample</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>SpringMVCAnnotationExample Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>3.0.1.RELEASE</version>
    </dependency>
   <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webMvc</artifactId>
    <version>3.0.1.RELEASE</version>
    </dependency>
    <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
   <version>3.0.1</version>
    </dependency>
  </dependencies>
  <build>
    <finalName>SpringMVCAnnotationExample</finalName>
  </build>
</project>
```

index.jsp

```
<html>
<head> <h1>
Spring MVC Annotation Example1</h1></head>
<body>
<form action="hello.do" method="post">
<h2>
Name :<input type="text" name="name"/>
<input type="submit" value="submit"/>
</h2>
</form>
</body>
</html>
```

hello.jsp

```
<%@page isELIgnored="false" %>
<h1>${msg}</h1>
```

web.xml

```
<web-app>
        <servlet>
                <servlet-name>ds</servlet-name>
                <servlet-class>
                        org.springframework.web.servlet.DispatcherServlet
                </servlet-class>
                <init-param>
                        <param-name>contextConfigLocation</param-name>
                        <param-value>/WEB-INF/spring-servlet.xml
                        </param-value>
                </init-param>
                <load-on-startup>1</load-on-startup>
        </servlet>
        <servlet-mapping>
                <servlet-name>ds</servlet-name>
                <url-pattern>/</url-pattern>
        </servlet-mapping>
        <welcome-file-list>
                <welcome-file>/WEB-INF/pages/index.jsp
                </welcome-file>
        </welcome-file-list>
</web-app>
```

spring-servlet.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">
        <context:component-scan base-package="com.nareshit.controller,com.nareshit.service">
        </context:component-scan>
</beans>
```

HelloController.java

```
package com.nareshit.controller;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.servlet.ModelAndView;
import com.nareshit.service.HelloService;
@Controller
public class HelloController {
        @Autowired
        private HelloService helloService;
        @RequestMapping(value = "hello", method = RequestMethod.POST)
        public ModelAndView sayHello(@RequestParam("name") String name) {
                String message = null;
                String targetView = "/WEB-INF/pages/hello.jsp";
                if (name != null && name.trim().length() > 0) {
                        message = helloService.sayHello(name);
                } else {
                        message = helloService.sayHello("Guest");
                }
                return new ModelAndView(targetView, "msg", message);
        }
}
```

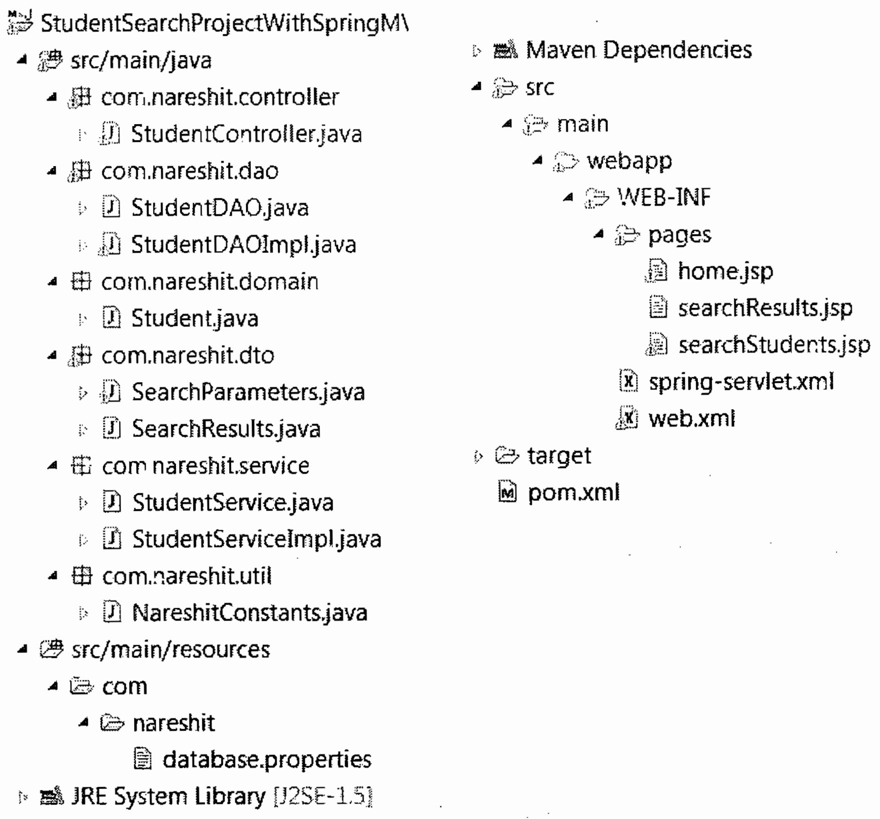
HelloService.java

```
package com.nareshit.service;
public interface HelloService {
String sayHello(String name);
}
```

HelloServiceImpl.java

```
package com.nareshit.service;
import org.springframework.stereotype.Service;
@Service
public class HelloServiceImpl
implements HelloService {
public String sayHello(String name) {
String message="Hello :"+name+" welcome to spring MVC";
return message;
        }
}
```

SearchStudentProjectWithSpringMVC Annotations



```
StudentSearchProjectWithSpringM\
  src/main/java
    com.nareshit.controller
      StudentController.java
    com.nareshit.dao
      StudentDAO.java
      StudentDAOImpl.java
    com.nareshit.domain
      Student.java
    com.nareshit.dto
      SearchParameters.java
      SearchResults.java
    com.nareshit.service
      StudentService.java
      StudentServiceImpl.java
    com.nareshit.util
      NareshitConstants.java
  src/main/resources
    com
      nareshit
        database.properties
  JRE System Library [J2SE-1.5]
  Maven Dependencies
  src
    main
      webapp
        WEB-INF
          pages
            home.jsp
            searchResults.jsp
            searchStudents.jsp
          spring-servlet.xml
          web.xml
  target
  pom.xml
```

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.nareshit</groupId>
    <artifactId>StudentSearchProjectWithSpringMVC</artifactId>
    <packaging>war</packaging>
    <version>0.0.1-SNAPSHOT</version>
    <name>StudentSearchProjectWithSpringMVC Maven Webapp</name>
    <url>http://maven.apache.org</url>
    <properties>
        <spring.version>3.0.5.RELEASE</spring.version>
        <hibernate.version>3.6.9.Final</hibernate.version>
        <commons.lang.version>3.3.2</commons.lang.version>
        <jstl.version>1.2</jstl.version>
        <servlet.version>3.0.1</servlet.version>
    </properties>
```

```
<dependencies>

        <dependency>

                <groupId>org.apache.commons</groupId>

                <artifactId>commons-lang3</artifactId>

                <version>${commons.lang.version}</version>

        </dependency>

        <dependency>

                <groupId>org.springframework</groupId>

                <artifactId>spring-web</artifactId>

                <version>${spring.version}</version>

        </dependency>

        <dependency>

                <groupId>org.springframework</groupId>

                <artifactId>spring-webmvc</artifactId>

                <version>${spring.version}</version>

        </dependency>


        <dependency>

                <groupId>org.springframework</groupId>

                <artifactId>spring-context</artifactId>

                <version>${spring.version}</version>

        </dependency>

        <dependency>

                <groupId>org.springframework</groupId>

                <artifactId>spring-jdbc</artifactId>

                <version>${spring.version}</version>

        </dependency>

        <dependency>

                <groupId>org.springframework</groupId>

                <artifactId>spring-tx</artifactId>

                <version>${spring.version}</version>

        </dependency>
```

```
<dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-orm</artifactId>
          <version>${spring.version}</version>
</dependency>
<dependency>
          <groupId>oracle.jdbc.driver</groupId>
          <artifactId>ojdbc14</artifactId>
          <version>10.2.0</version>
</dependency>
<dependency>
          <groupId>jstl</groupId>
          <artifactId>jstl</artifactId>
          <version>${jstl.version}</version>
</dependency>
<dependency>
          <groupId>javax.servlet</groupId>
          <artifactId>javax.servlet-api</artifactId>
          <version>${servlet.version}</version>
</dependency>
<dependency>
          <groupId>org.hibernate</groupId>
          <artifactId>hibernate-entitymanager</artifactId>
          <version>${hibernate.version}</version>
</dependency>
<dependency>
          <groupId>org.hibernate</groupId>
          <artifactId>hibernate-core</artifactId>
          <version>${hibernate.version}</version>
</dependency>
<dependency>
          <groupId>org.javassist</groupId>
```

```
                    <artifactId>javassist</artifactId>
                    <version>3.15.0-GA</version>
            </dependency>
    </dependencies>
    <build>
            <finalName>StudentSearchProjectWithSpringMVC</finalName>
    </build>
</project>
```

web.xml

```
<web-app>
    <servlet>
            <servlet-name>dispatcher</servlet-name>
            <servlet-class>
                    org.springframework.web.servlet.DispatcherServlet
            </servlet-class>
            <init-param>
                    <param-name>contextConfigLocation</param-name>
                    <param-value>/WEB-INF/spring-servlet.xml</param-value>
            </init-param>
            <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
            <servlet-name>dispatcher</servlet-name>
            <url-pattern>*.do</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
            <welcome-file>/WEB-INF/pages/home.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

home.jsp

```
<html>

<head><h1><center>Welcome to MySite</h1></center></head>

<a href="searchStudents.do">SearchStudents</a>

</html>
```

searchStudents.jsp

```
<html>

<head><h1><center>Welcome to MySite<br/><br/>

StudentSearchPage</h1></center></head><br/><hr/><body>

<form action="searchStudents.do" method="post">

<input type="text" placeholder="studentId" name="studentId"/> &nbsp;

<input type="text" placeholder="name" name="name"/> &nbsp;

<input type="text" placeholder="course"  name="course"/> &nbsp;

<input type="text" placeholder="mobile" name="mobile"/> &nbsp;

<input type="submit" value="search"/>

</form>

<br/><br/>

</body>

</html>
```

searchResults.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<jsp:include page="searchStudents.jsp"></jsp:include>

<html><body>

<c:choose>

<c:when test="${searchResults.size()>0} ">

<table border="1">

<c:forEach items="${searchResults} " var="searchResult">

<tr>

<td>${searchResult.studentId}</td>

<td>${searchResult.name}</td>

<td>${searchResult.mobile}</td>
```

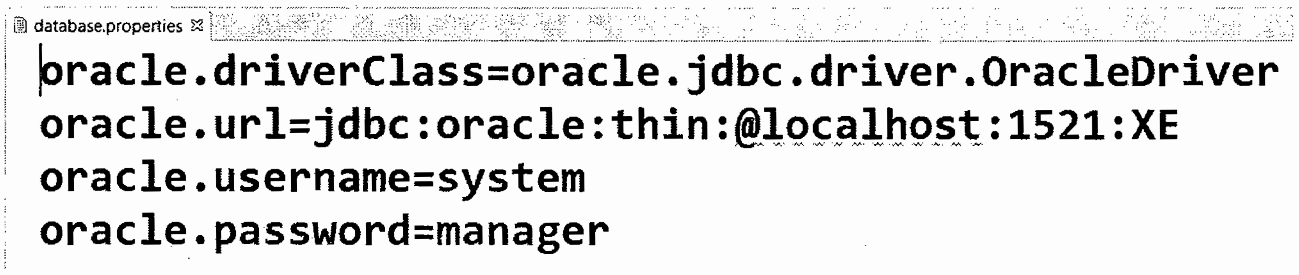
```
<td>${searchResult.course}</td>
<td>${searchResult.city} ${searchResult.state}</td>
</tr>
</c:forEach>
</table>
</c:when>
<c:otherwise>
<c:out value="No Records Found"/>
</c:otherwise>
</c:choose></body></html>
```

spring-servlet.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">
<context:component-scan base-package="com.nareshit"/>
<context:property-placeholder location="classpath:/com/nareshit/database.properties"/>
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName" value="${oracle.driverClass}"/>
<property name="url" value="${oracle.url}"/>
<property name="username" value="${oracle.username}"/>
<property name="password" value="${oracle.password}"/>
</bean>
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
<property name="dataSource" ref="dataSource"/>
<property name="hibernateProperties">
<props>
<prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>
```

```
<prop key="hibernate.show_sql">true</prop>
</props>
</property>
<property name="annotatcdClasses">
<list><value>com.nareshit.domain.Student</value></list>
</property>
</bean>
</beans>
```

database.properties



database.properties

```
oracle.driverClass=oracle.jdbc.driver.OracleDriver
oracle.url=jdbc:oracle:thin:@localhost:1521:XE
oracle.username=system
oracle.password=manager
```

NareshitConstants.java

```
package com.nareshit.util;
public class NareshitConstants {
public static final String CONST_STUDENT_ID="studentId";
public static final String CONST_STUDENT_NAME="name";
public static final String CONST_STUDENT_COURSE="course";
public static finai String CONST_MOBILE="mobile";
public static final String CONST_EMAIL="email";
public static final String CONST_ADDRESS="address";
public static final String CONST_SEARCH_RESULTS="searchResults";
}
```

StudentController.java

```
package com.nareshit.controller;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.servlet.ModelAndView;
import com.nareshit.dto.SearchParameters;
import com.nareshit.dto.SearchResults;
import com.nareshit.service.StudentService;
import com.nareshit.util.NareshitConstants;
@Controller
public class StudentController {
        @Autowired
        private StudentService studentService;

    private static final String WEB_SEARCH_STUDENTS = "/searchStudents.do";
    private static final String WEB_SEARCH_STUDENTS_PAGE =
     "/WEB-INF/pages/searchStudents.jsp";
        private static final String WEB_SEARCH_RESULTS_PAGE =
      "/WEB-INF/pages/searchResults.jsp";

        @RequestMapping(value = WEB_SEARCH_STUDENTS, method = RequestMethod.GET)
        public String showSearchStudentsPage() {
                return WEB_SEARCH_STUDENTS_PAGE;
        }

        @RequestMapping(value = WEB_SEARCH_STUDENTS, method = RequestMethod.POST)
        public ModelAndView searchStudents(
                @RequestParam(NareshitConstants.CONST_STUDENT_ID) Integer studentId,
                @RequestParam(NareshitConstants.CONST_STUDENT_NAME) String name,
                @RequestParam(NareshitConstants.CONST_STUDENT_COURSE) String course,
                @RequestParam(NareshitConstants.CONST_MOBILE) String mobile) {
                SearchParameters searchParams = new SearchParameters();
                searchParams.setStudentId(studentId);
                searchParams.setName(name);
                searchParams.setCourse(course);
                searchParams.setMobile(mobile);
                List<SearchResults> searchResults =
        studentService.searchStudents(searchParams);
                ModelAndView modelAndView = new ModelAndView();
                modelAndView.setViewName(WEB_SEARCH_RESULTS_PAGE);
                modelAndView.addObject(NareshitConstants.CONST_SEARCH_RESULTS,
                                searchResults);
                return modelAndView;
        }
}
```

StudentService.java

```
package com.nareshit.service;
import java.util.List;
import com.nareshit.dto.SearchParameters;
import com.nareshit.dto.SearchResults;
public interface StudentService {
public List<SearchResults> searchStudents(SearchParameters searchParams);
}
```

StudentServiceImpl.java

```
package com.nareshit.service;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.nareshit.dao.StudentDAO;
import com.nareshit.dto.SearchParameters;
import com.nareshit.dto.SearchResults;
@Service
public class StudentServiceImpl  implements StudentService{
```

```
@Autowired
        private StudentDAO studentDao;
        public List<SearchResults> searchStudents(SearchParameters searchParams) {
                List<SearchResults> searchResults=studentDao.searchStudents(searchParams);
                return searchResults;
        }
}
```

StudentDAO.java

```
package com.nareshit.dao;
import java.util.List;
import com.nareshit.dto.SearchParameters;
import com.nareshit.dto.SearchResults;
public interface StudentDAO {
        public List<SearchResults> searchStudents(SearchParameters searchParams);
}
```

StudentDAOImpl.java

```
package com.nareshit.dao;
import java.util.ArrayList;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import com.nareshit.dto.SearchParameters;
import com.nareshit.dto.SearchResults;

@Repository
public class StudentDAOImpl implements StudentDAO {
        @Autowired
        private SessionFactory sessionFactory;
        public List<SearchResults> searchStudents(SearchParameters searchParams) {

                List<SearchResults> searchResultsList = new ArrayList<SearchResults>();

StringBuffer sb =

new StringBuffer("select s.studentId,s.name,s.mobile,s.course,s.city,s.state from    com.nareshit.domain.Student as s ");

                boolean isFirst = true;

                        if (searchParams != null) {

                          if (searchParams.getStudentId() != null

                             && searchParams.getStudentId() > 0) {

                          sb.append(" where s.studentId=" + searchParams.getStudentId());

                          isFirst = false;

                          }

                          if (searchParams.getName() != null

            && searchParams.getName().trim().length() > 0) {

                                  if (isFirst) {

                          sb.append(" where s.name like '%" +searchParams.getName()

                                                            + "%'");

                                  isFirst = false;

                                  } else {

                          sb.append(" AND s.name like '%" +searchParams.getName()+"%'");

                                  }

                          }

                          if (searchParams.getCourse() != null

                                          && searchParams.getCourse().trim().length() > 0) {

                                  if (isFirst) {

                 sb.append(" where s.course like '"+searchParams.getCourse()+ "'");

                                  isFirst = false;

                                  } else {

                sb.append(" AND  s.course like '"+ searchParams.getCourse()+ "'");

                                  }

                          }

                          if (searchParams.getMobile() != null
                        && searchParams.getMobile().trim().length() > 0) {
                    if (isFirst) {
    sb.append(" where s.mobile like '"+searchParams.getMobile()+ "'");
                        isFirst = false;
                    } else {
    sb.append(" AND  s.mobile like '"+ searchParams.getMobile() + "'");
                    }
                }
            }
            Session session = sessionFactory.openSession();
            Query query = session.createQuery(sb.toString());
            List<Object[]> list = query.list();
            for (Object[] obj : list) {
                SearchResults searchResult = new SearchResults();
                searchResult.setStudentId((Integer)obj[0]);
                searchResult.setName((String)obj[1]);
                searchResult.setMobile((String)obj[2]);
                searchResult.setCourse((String)obj[3]);
                searchResult.setCity((String)obj[4]);
                searchResult.setState((String)obj[5]);
                // add searchResult into searchResultsList
                searchResultsList.add(searchResult);
            }

            return searchResultsList;
        }

}
```

Student.java

```
package com.nareshit.domain;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name="studentMaster")

public class Student {

        @Id

        private int studentId;

        @Column(length=20)

        private String name,course,email,mobile;

        private double fee;

        @Column(length=20)

        private String city,state;

        public int getStudentId() {

                return studentId;

        }

        public void setStudentId(int studentId) {

                this.studentId = studentId;

        }

        public String getName() {

                return name;

        }

        public void setName(String name) {

                this.name = name;

        }

        public String getCourse() {

                return course;

        }

        public void setCourse(String course) {

                this.course = course;

        }

        public String getEmail() {

                return email;
        }
        public void setEmail(String email) {
                this.email = email;
        }
        public String getMobile() {
                return mobile;
        }
        public void setMobile(String mobile) {
                this.mobile = mobile;
        }
        public double getFee() {
                return fee;
        }
        public void setFee(double fee) {
                this.fee = fee;
        }
        public String getCity() {
                return city;
        }
        public void setCity(String city) {
                this.city = city;
        }
        public String getState() {
                return state;
        }
        public void setState(String state) {
                this.state = state;
        }
}
```

<u>**SearchParameters.java**</u>

```
package com.nareshit.dto;
```

```
import java.io.Serializable;

public class SearchParameters implements Serializable{

private Integer studentId;

private String name;

private String course,mobile;

public Integer getStudentId() {

        return studentId;

}

public void setStudentId(Integer studentId) {

        this.studentId = studentId;

}

public String getName() {

        return name;

}

public void setName(String name) {

        this.name = name;

}

public String getCourse() {

        return course;

}

public void setCourse(String course) {

        this.course = course;

}

public String getMobile() {

        return mobile;

}

public void setMobile(String mobile) {

        this.mobile = mobile;

}

}
```

SearchResults.java

```
package com.nareshit.dto;

public class SearchResults {

private Integer studentId;

private String name,mobile,city,state,course;

public Integer getStudentId() {

        return studentId;

}

public void setStudentId(Integer studentId) {

        this.studentId = studentId;

}

public String getName() {

        return name;

}

public void setName(String name) {

        this.name = name;

}

public String getMobile() {

        return mobile;

}

public void setMobile(String mobile) {

        this.mobile = mobile;

}

public String getCity() {

        return city;

}

public void setCity(String city) {

        this.city = city;

}

public String getState() {

        return state;

}

public void setState(String state) {
        this.state = state;
}
public String getCourse() {
        return course;
}
public void setCourse(String course) {
        this.course = course;
}
}
```

HandlerInterceptors

In the Handler Mappings section we come across the HandlerInterceptors,which means Handler Mappings along with mapping URL to Handler will also configured with Handler Interceptors.
The main purpose of Handler Interceptors is to handle pre and post processing of incoming request this is similar to concept of filters in J2ee But the main difference between filters and interceptors, filters are applied to all the requests of the web application, where as interceptors are applied to certain group of handlers. you have three states of interceptor Execution like before processing request,before rendering the view and after the view has rendered to the user.
we can create our own interceptors in Spring by either implementing org.springframework.web.servlet.HandlerInterceptor interface or by extending abstract class org.springframework.web.servlet.handler.HandlerInterceptorAdapter that provides the base implementation of this interface.

HandlerInterceptor interface contain three methods

1. **boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)**: This method is used to intercept the request before it's handed over to the handler method. This method should return 'true' to let Spring know to process the request through another interceptor or to send it to handler method if there are no further interceptors.

If this method returns 'false' Spring framework assumes that request has been handled by the interceptor itself and no further processing is needed. We should use response object to send response to the client request in this case.

2. **void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView)**: This interceptor method is called when HandlerAdapter has invoked the handler but DispatcherServlet is yet to render the view. This method can be used to add additional attribute to the ModelAndView object to be used in the view pages. We can use this interceptor to determine the time taken by handler method to process the client request.
3. **void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)**: This is a callback method that is called once the handler is executed and view is rendered.

If there are multiple interceptors configured, *preHandle()* method is executed in the order of configuration whereas *postHandle()* and *afterCompletion()* methods are invoked in the reverse order.

Struts 1.x -Spring integration

Approach-1:-

step-1:- create a project and add the struts and spring jars(web libraries and core libraries)

step 2:-

write web.xml file

in web.xml file configure ActionServlet class given by struts 1.x version

ActionServlet is a FrontController and Singleton class.

Along With ActionServlet we can configure ContextLoaderListener.

The ContextLoaderListener will create ApplicationContext container.

The container searches the spring cfg file by-default in WEB-INF folder with the name of applicationContext.xml.

If we are customizing the spring cfg file name we can write <context-param> to specify the spring cfg filename.

step 3:-

Develop Struts Action class ,which extends ActionSupport given by Spring f/w.

ActionSupport class is deprecated in Spring 3.x version. ActionSupport class is the child class of struts Action class.

To get Spring container in Struts Action class and to communicate with any spring beans we can call getWebApplicationContext() method.

Step4:-

Write struts configuration file with ActionMapping configuration.

Step5:-

Write Spring configuration file and configure the Service Beans and DAO beans as per the requirement.

Approach-2:-

step-1:- create a project and add the struts and spring jars(web libraries and core libraries)

step 2:-

write web.xml file

in web.xml file configure ActionServlet class given by struts 1.x version

ActionServlet is a FrontController and Singleton class.

Along With ActionServlet we can configure ContextLoaderListener.

The ContextLoaderListener will create ApplicationContext container.

The container searches the spring cfg file by-default in WEB-INF folder with the name of applicationContext.xml.

If we are customizing the spring cfg file name we can write <context-param> to specify the spring cfg filename.

step 3:-

Develop Struts Action class ,which extends Struts Action classs.

In this case maintain the service class as a dependency in the action clas to invoke the Bussiness Methods.

Ex:-

```
public class UserAction extends Action{
private UserService userService;

public ActionForward execute(ActionMapping mapping,ActionForm form,HttpServletRequest req,HttpServletResponse
res){
--
--
}
--
}
```

step 4:-

Write struts-config.xml file. In stuts configuration file configure spring f/w given org.springframework.web.struts.DelegatingActionProxy controller class .

Example:

```
<action-mappings>

<action path="/saveUser"
type="org.springframework.web.struts.DelegatingActionProxy">
<forward name="success" path="/result.jsp"></forward>
<forward name="failure" path="/index.jsp"></forward>
</action>
</action-mappings>
```

Step5:-

Write spring cfg file .

In spring configuration file configure Action class Bean,Service classBeans,Dao class Beans .

```
<bean name="/saveUser"
class="com.nareshit.controller.UserAction">
<property name="userService" ref="userService"/>
</bean>
<bean id="userService"
class="com.nareshit.service.UserServiceImpl" >
<property name="userDao" ref="userDao">
</property>
</bean>
```

Spring AOP Module

Aspect Oriented programming in Spring:-

One of the major features available in the Spring Distribution is the provision for separating the cross-cutting concerns in an Application through the means of Aspect Oriented Programming. Aspect Oriented Programming is sensibly new and it is not a replacement for Object Oriented Programming. In fact, AOP is another way of organizing your Program Structure. This first section of this article looks into the various terminologies that are commonly used in the AOP Environment. Then it moves into the support that is available in the Spring API for embedding Aspects into an Application .

Introduction to AOP

The Real Problem

Since AOP is relatively new, this section devotes time in explaining the need for Aspect Oriented Programming and the various terminologies that are used within. Let us look into the traditional model of before explaining the various concepts.
Consider the following sample application,

```
public class Account{
 public long deposit(long depositAmount){
        newAmount = existingAccount + depositAccount;
        currentAmount = newAmount;
     }        return currentAmount;
            }
public long withdraw(long withdrawalAmount){
if (withdrawalAmount<=currentAmount){
  currentAmount=currentAmount-withdrawalAmount;
        }
        return currentAmount;
}
}
```

The above code models a simple Account Object that provides services for deposit and withdrawal operation in the form of Account.deposit() and Account.withdraw() methods. Suppose say we want to add some bit of the security to the Account class, telling that only users with BankAdmin privilege is allowed to do the operations. With this new requirement being added, let us see the modified class structure below.
Account.java

```
public class Account{
 public long deposit(long depositAmount){
 User user = getContext().getUser();
     if (user.getRole().equals("BankAdmin"){
        newAmount = existingAccount + depositAccount;
        currentAmount = newAmount;
     }        return currentAmount;
            }
public long withdraw(long withdrawalAmount){
User user = getContext().getUser();
    if (user.getRole().equals("BankAdmin"){
                if (withdrawalAmount<=currentAmount){
          currentAmount=currentAmount-withdrawalAmount;
                                }
        return currentAmount;
}
}
```

Assume that getContext().getUser() someway gives the current User object who is invoking the operation. See the modified code mandates the use of adding additional if condition before performing the requested operation. Assume that another requirement for the above Account class is to provide some kind of Logging and Transaction Management Facility. Now the code expands as follows,

Account.java

```
public class Account{

 public long deposit(long depositAmount){

 logger.info("Start of deposit method");

  Transaction trasaction  = getContext().getTransaction();

   transaction.begin();

    try{

     User user = getContext().getUser();

          if (user.getRole().equals("BankAdmin"){

           newAmount = existingAccount + depositAccount;

            currentAmount = newAmount;

               }

               transaction.commit();

            }catch(Exception exception){

               transaction.rollback();

            }

            logger.info("End of deposit method");

            return currentAmount;

        }

  public long withdraw(long withdrawalAmount){

                  logger.info("Start of withdraw method");

            Transaction trasaction = getContext().getTransaction();

            transaction.begin();

            try{
                User user = getContext().getUser();

                if (user.getRole().equals("BankAdmin"){

                  if (withdrawalAmount<=currentAmount){

          currentAmount=currentAmount-withdrawalAmount;

                                  }

        return currentAmount;

}
```

The above code has so many dis-advantages. The very first thing is that as soon as new requirements are coming it is forcing the methods and the logic to change a lot which is against the Software Design. Remember every piece of newly added code has to undergo the Software Development Lifecycle of Development, Testing, Bug Fixing, Development, Testing, ... This, certainly cannot be encouraged in particularly big projects where a single line of code may have multiple dependencies between other Components or other Modules in the Project.

<u>The Solution through AOP</u>

Let us re-visit the Class Structure and the Implementation to reveal the facts. The Account class provides services for depositing and withdrawing the amount. But when you look into the implementation of these services, you can find that apart from the normal business logic, it is doing so many other stuffs like Logging, User Checking and Transaction Management. See the pseudo-code below that explains this

```
public void deposit(){
 // Transaction Management
   // Logging
   // Checking for the Privileged User
   // Actual Deposit Logic comes here
 }
 public void withdraw(){
          // Transaction Management
            // Logging
            // Checking for the Privileged User
            // Actual Withdraw Logic comes here
         }
```

From the above pseudo-code, it is clear that Logging, Transaction Management and User Checking which are never part of the Deposit or the Service functionality are made to embed in the implementation for completeness. Specifically, AOP calls this kind of logic that cross-cuts or overlaps the existing business logic as Concerns or Cross-Cutting Concerns. The main idea of AOP is to isolate the cross-cutting concerns from the application code thereby modularizing them as a different entity. It doesn't mean that because the cross-cutting code has been externalized from the actual implementation, the implementation now doesn't get the required add-on functionalities. There are ways to specify some kind of relation between the original business code and the Concerns through some techniques which we will see in the subsequent sections.

Q) What is the difference between OOP and AOP?
Spring AOP is no way related with OOP because OOP is methodology of representing the Real word Object's in programming languages. AOP is the methodology of applying middleware services on the Spring applications.

In Spring AOP Middleware services are also called as Aspects or cross cutting concerns because they reside outside the application logics but will executed along with the application logics. Instead of writing Middleware service logics directly with application code it is recommended to separate them from application code and apply them on application code using
XML, Annotations by taking the support of Spring AOP.

AOP Terminologies:

1. Aspect
2. Advice
3. Advisor
4. Point cut
5. Wiring
6. Weaving
7. JoinPoint
8. Target Object
9. Proxy Object.

1. Aspect: Contains the plan of implementing Middleware Service
2. Advice: It is the practical implementation of Middleware Services.
3**. Join Point** :The position in business methods where advices are configured are called as Join Point

EX: Beginning of the business methods, end of the business methods. Exception is resided in the business method and etc., called as joinPoints.

```
public void someBussinessOperation(BusinessData data){
//MethodStart-->possible aspect code here like logging.
try{
        //Original Bussiness Logic Here.

}catch(Exception e)
        {
        //Exception-->Aspect code here when some exception is raised
        }
finally
        {
//Finally-->Even Possible to have aspect code at this point too.
        }
        //MethodEnd-->Aspect code here in the End of a method.
}
```

All possible Execution points in the Application code
for embedding Aspects are called **JoinPoints.**

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

4.Pointcut : The XML or Annotation configurations that are used to link advices with Join Points are called **Point Cuts.**

5. **Advisor:** It is a combination of Advice+ Joint Point + Point Cut.
6 **Wiring: The** process of configuring bean properties for dependency injection is called Wiring.

7. **Weaving :** The process of configuring advices/Middleware services/Aspects on bean class business methods is called as Weaving.
8.**Target :** The bean class object which we are planning to apply Middleware services (but not apply) is called Target object.

Target object means actualobject(OR)real object.

Note: All Spring Bean objects are Target Objects by default.

9)Proxy Objects:
Proxy is not a real object,proxy object we are deriving based on the real object.
Proxy object is working like a real object.

Example :- ATM is a proxy for Bank object.

ATM is not a real bank,but it is working like a Bank.

In Spring Bean The Middleware services we are applying on the proxy objects.

Note :-

If we call business methods on Target object only business logic will be executed. If we call business methods on Proxy Object the business logic will be executed along with Middleware services.

Generally we never apply Middleware services in small scale project development. While we apply Middleware services in medium and large scale projects.

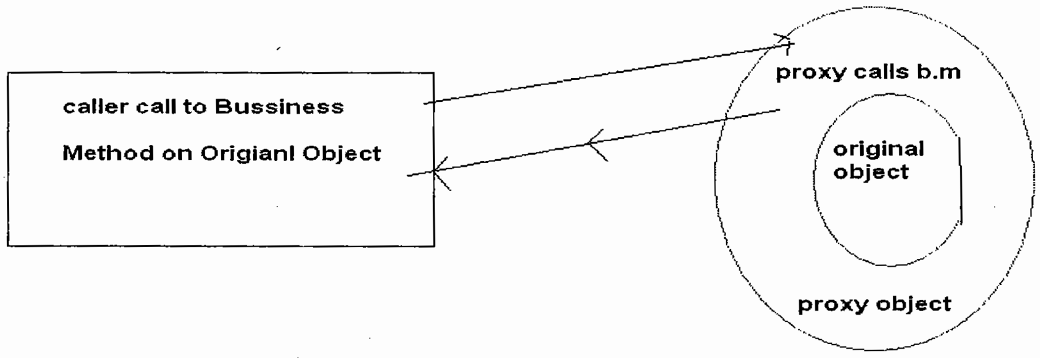
Spring AOP supports only method level Aspect oriented programming that means it does not support Field level Aspect oriented programming.

Spring beans borrowed AOP facilities from Alliance company but entire API is inbuilt of Spring software.

Advice: Advice is the code that implements the Aspect.. In general an Aspect defines the functionality in a more abstract manner. But, Advice that provides a Concrete Implementation for the Aspect.

How Does AOP Work?

→Aop Works on the Proxy Design Pattern.
According to proxy design pattern the Actual Object(Bussiness Object) is Wrapped into another Object is known as Proxy and Substitutes that object
In place of Actual object (B.O)



Note:

The org.springframework.aop.framework.ProxyFactoryBean class converts given Spring Bean class object to Proxy object and returns that object.

Note: When business methods are called on Proxy object the business logic will be executed along with Advices.

1. Programmatic Approach: keeping the logic of applying Middleware services as Java statements directly in the applications (not recommended).

2. Declarative Approach: Middleware services will be applied on the applications either through XML statements or through Annotations.

Creating Advices in Spring

There is possibility of developing four types of advices those are

i. BeforeAdvice (Executes at the beginning of the business method)
ii. AfterAdvice (executes at the end of the business method)
iii. ThrowsAdvice (executes when the business method throws an exception)
iv. Around Advice (executes at beginning , when the business method throws an exception And at the end of business method)

1. BeforeAdvice:

Before Advice is used to intercept before the method execution starts. In AOP, Before Advice is represented in the form of org.springframework.aop.MethodBeforeAdvice.

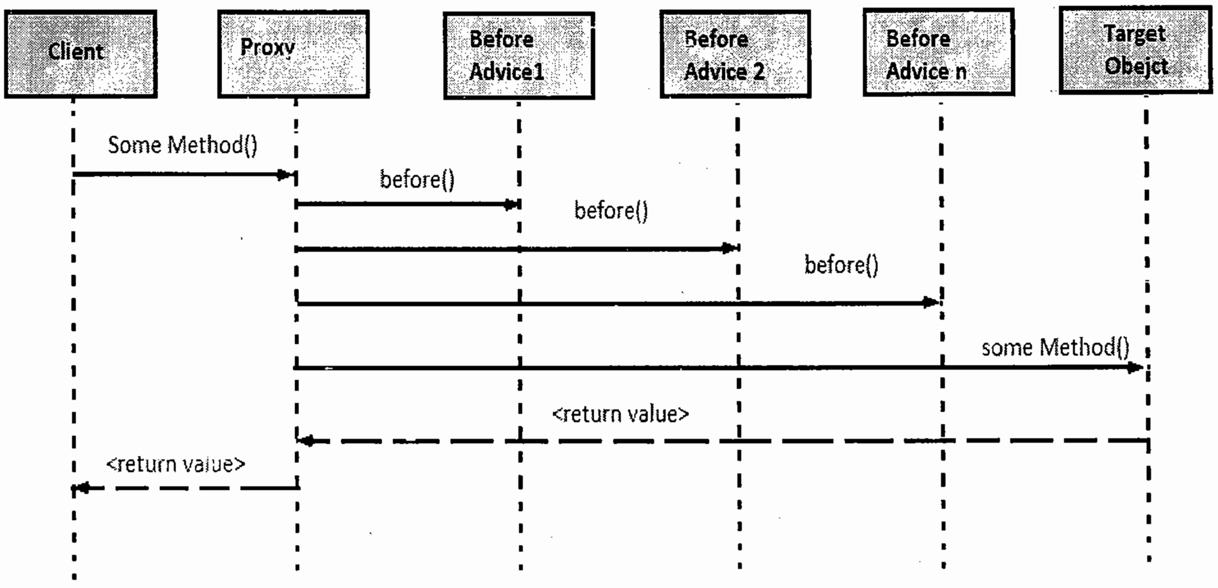
For example a sytem makes security checks before allowing then to accesing resources. In such a case, we can have a Before Advice that contains code whilch implements Authentication logic
consider the following piece of code.

EX:

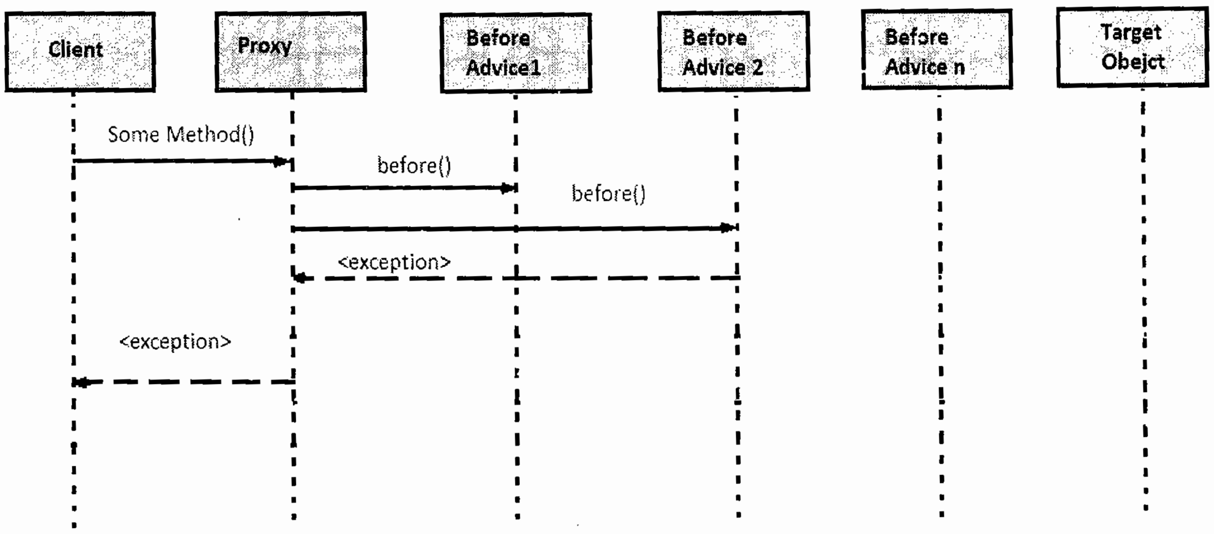
```
public class Authentication implements MethodBeforeAdvice
// Here MethodBefore Adviceis manadatory to make the Java class BeforeAdvice{
public void before(Method method, Object[] args, Object target) throws Throwable
{
//place the Authentication logic here.
}
}
```

Above class implements MethodBeforeAdvice, there by telling before method will be called before the execution of the bussiness method call. Note that the java.lang.reflect.Method method object represents target method to be invoked, Object[] args refers to the various arguments that are passed on the method and target refers to the object which is calling the method.

The following Sequence diagram describing the operations when before advice and joinpoint execution terminates normally, that is , without throwing exception



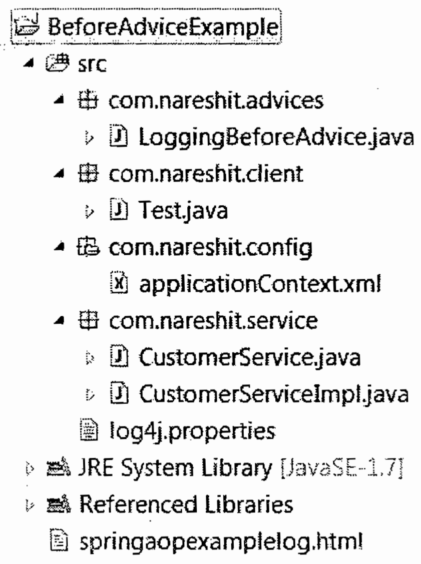
The following Sequence diagram describing the operations when before advice exection and terminates abnormally, that is , throwing exception



As shown in fig. in the case where before advice throws an exception then an exception is thrown to the client without continuing further the interceptor's execution and the target object method.

Working with before advice :-

In this section we will develop an example to demonstrate how to write an advice that has to be applied before the joinpoint execution, and how to configure it according to the spring 1.2 low-level approach. In this example we will implement the well known simple customer service which requires some seconday logics like logging.



```
BeforeAdviceExample
  src
    com.nareshit.advices
      LoggingBeforeAdvice.java
    com.nareshit.client
      Test.java
    com.nareshit.config
      applicationContext.xml
    com.nareshit.service
      CustomerService.java
      CustomerServiceImpl.java
    log4j.properties
  JRE System Library [JavaSE-1.7]
  Referenced Libraries
  springaopexamplelog.html
```

CustomerService.java

```
package com.nareshit.service;
public interface CustomerService {
public String printName();
public String printUrl();
public void printException()throws InterruptedException;
}
```

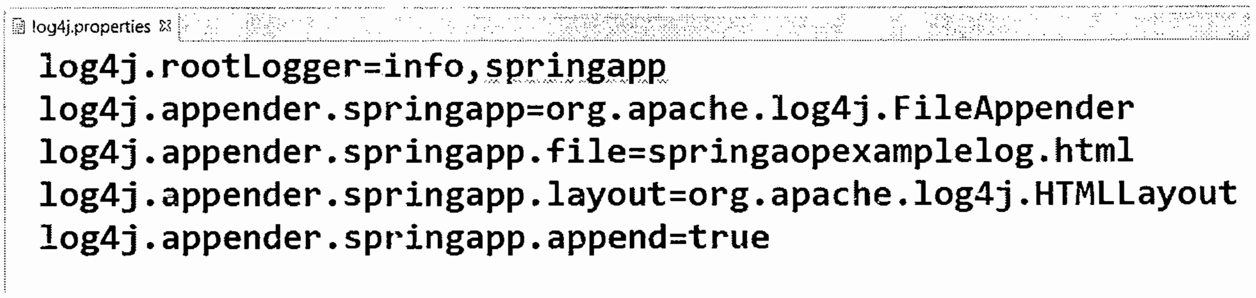
CustomerServiceImpl.java

```
package com.nareshit.service;
public class CustomerServiceImpl implements CustomerService {
        private String name, url;
   public void setName(String name) {
                this.name = name;
        }
   public void setUrl(String url) {
                this.url = url;
        }
   public String printName() {
                System.out.println("Bussiness Method : PrintName () :");
                return name;
        }
  public String printUrl() {
        System.out.println("Bussiness Method : PrintURL () :");
                return url;
        }
  public void printException() throws InterruptedException {
                System.out.println("Bussiness Method : PrintException () :");
                throw new InterruptedException("Don't Sleep in the class room");
        }
}
```

LoggingBeforeAdvice.java

```
package com.nareshit.advices;
import java.lang.reflect.Method;
import org.apache.log4j.Logger;
import org.springframework.aop.MethodBeforeAdvice;
public class LoggingBeforeAdvice implements  MethodBeforeAdvice{
public void before(Method method, Object[] arg, Object target)
                        throws Throwable {
Logger logger=Logger.getLogger(target.getClass().getName());
logger.info("Enterted into :"+target.getClass().getName()+" class  "+method.getName()+"
Method With the arguments "+arg.length);
        }
}
```

log4j.properties



```
log4j.rootLogger=info,springapp
log4j.appender.springapp=org.apache.log4j.FileAppender
log4j.appender.springapp.file=springaopexamplelog.html
log4j.appender.springapp.layout=org.apache.log4j.HTMLLayout
log4j.appender.springapp.append=true
```

applicationContext.xml

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

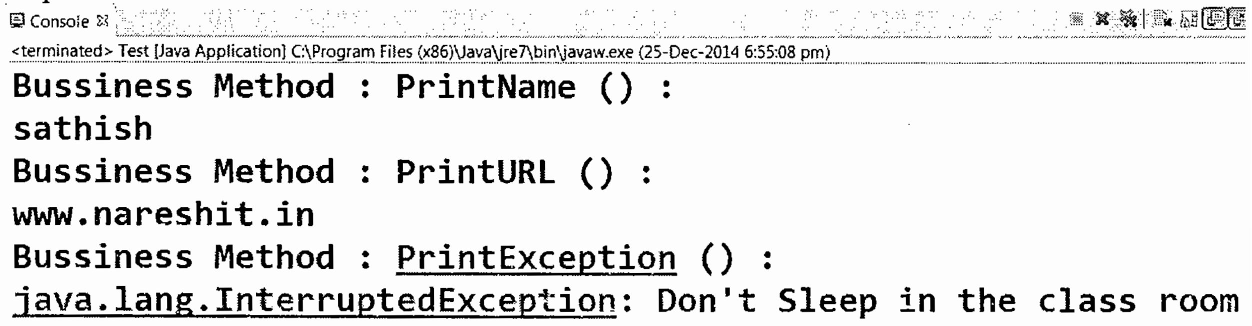
```
<bean id="loggingBeforeAdvice" class="com.nareshit.advices.LoggingBeforeAdvice">
</bean>
<bean id="customerService" class="com.nareshit.service.CustomerServiceImpl">
<property name="name" value="sathish"/>
<property name="url" value="www.nareshit.in"/>
</bean>
<bean id="customerServiceProxy"
class="org.springframework.aop.framework.ProxyFactoryBean">
<property name="proxyInterfaces" value="com.nareshit.service.CustomerService"/>
<property name="interceptorNames"  value="loggingBeforeAdvice"/>
<property name="target" ref="customerService"/>
</bean>
</beans>
```

Test.java

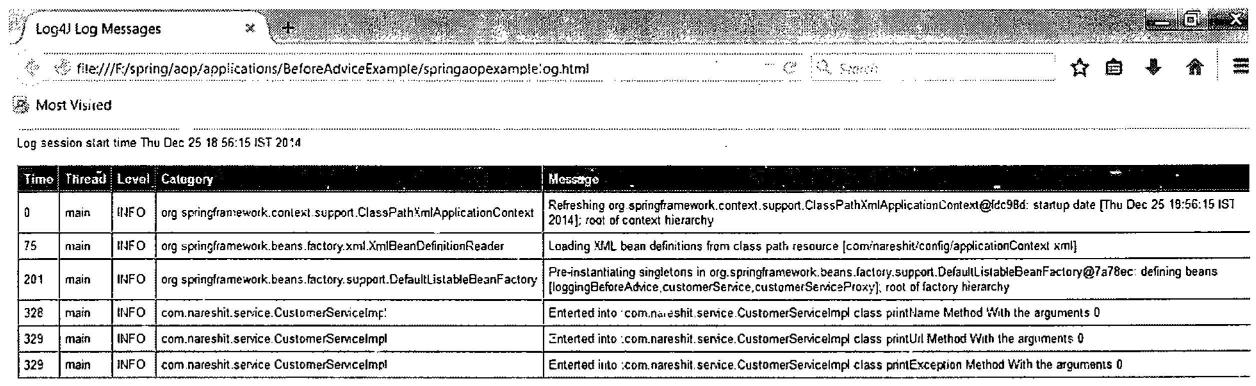
```
package com.nareshit.client;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.nareshit.service.CustomerService;
public class Test {
        public static void main(String[] args) {
                ApplicationContext context = new ClassPathXmlApplicationContext(
                                "com/nareshit/config/applicationContext.xml");
                CustomerService cs = (CustomerService) context
                                .getBean("customerServiceProxy");
                String name = cs.printName();
                System.out.println(name);
                String url = cs.printUrl();
                System.out.println(url);
                try {
                        cs.printException();
                } catch (InterruptedException ie) {
                        System.out.println(ie);
                }
        }
}
```

Upon execution of the above application, the log file will be generated and the output and logfile content as shown in the following fig's will be presented

output


```
Console
<terminated> Test [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (25-Dec-2014 6:55:08 pm)
Bussiness Method : PrintName () :
sathish
Bussiness Method : PrintURL () :
www.nareshit.in
Bussiness Method : PrintException () :
java.lang.InterruptedException: Don't Sleep in the class room
```

Logfile :-

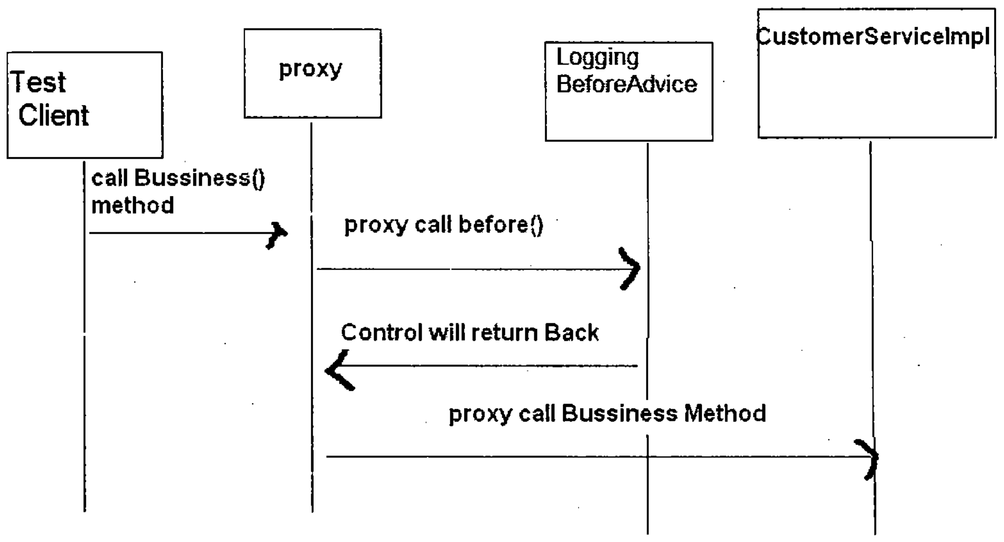

Log4J Log Messages
file:///F:/spring/aop/applications/BeforeAdviceExample/springaopexamplelog.html
Most Visited

Log session start time Thu Dec 25 18 56:15 IST 2014

Time	Thread	Level	Category	Message
0	main	INFO	org springframework.context.support.ClassPathXmlApplicationContext	Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@fdc98d: startup date [Thu Dec 25 18:56:15 IST 2014]; root of context hierarchy
75	main	INFO	org springframework.beans.factory.xml.XmlBeanDefinitionReader	Loading XML bean definitions from class path resource [com/nareshit/config/applicationContext xml]
201	main	INFO	org springframework.beans.factory.support.DefaultListableBeanFactory	Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@7a78ec: defining beans [loggingBeforeAdvice,customerService,customerServiceProxy]; root of factory hierarchy
328	main	INFO	com.nareshit.service.CustomerServiceImpl	Enterted into :com.nareshit.service.CustomerServiceImpl class printName Method With the arguments 0
329	main	INFO	com.nareshit.service.CustomerServiceImpl	Enterted into :com.nareshit.service.CustomerServiceImpl class printUrl Method With the arguments 0
329	main	INFO	com nareshit.service CustomerServiceImpl	Enterted into :com.nareshit.service.CustomerServiceImpl class printException Method With the arguments 0

The following diagram helps to understand the above example execution flow.
co

Sequence Diagram of Before Advice



As shown in the above fig the spring container dynamically creates a proxy object through the ProxyFactoryBean , where the dynamically built Proxy class implements all the interfaces given under the proxyInterfaces list (in the above example one interface com.nareshit.service.CustomerService) . All the methods are implemented to invoke the configured interceptors(advices) and the method on the given target Object.

Note that in the above example, we have designed an interface com.nareshit.service.CustomerService for target Object type but in all the cases we may not have an interfaces as it is not mandatory to design. It is not mandatory to design an Object implementing interface to proxy object, Spring allows to proxy Object without an interface.
To Configure ProxyFactoryBean with out proxyInterfaces just avoid proxyInterfaces property In the ProxyFactoryBean bean definition and set the CGLIB jar file into classpath.
If ProxyFactoryBean is configured without Proxy Interfaces then the Proxy that it builds dynamically will be a subtype of the target Object type. In the above example if you remove the <property name="proxyInterfaces"> element from applicationContext.xml then the proxy build will be extending com.nareshit.service.CustomerServiceImpl class instead of implementing the com.nareshit.service.CustomerService interface .

Note :-

In Spring 2.0 if we do not configure proxyInterfaces property , then ProxyFactoryBean builds the proxy class,implementing all the interfaces implemented by the target object class.
In case the target object does not implement an interfaces then it extends the target Object class.Make sure that in this case, the target object class has no-arg constructor where even the CGLIB.jar file is set into the classpath.

2. AfterAdvice:

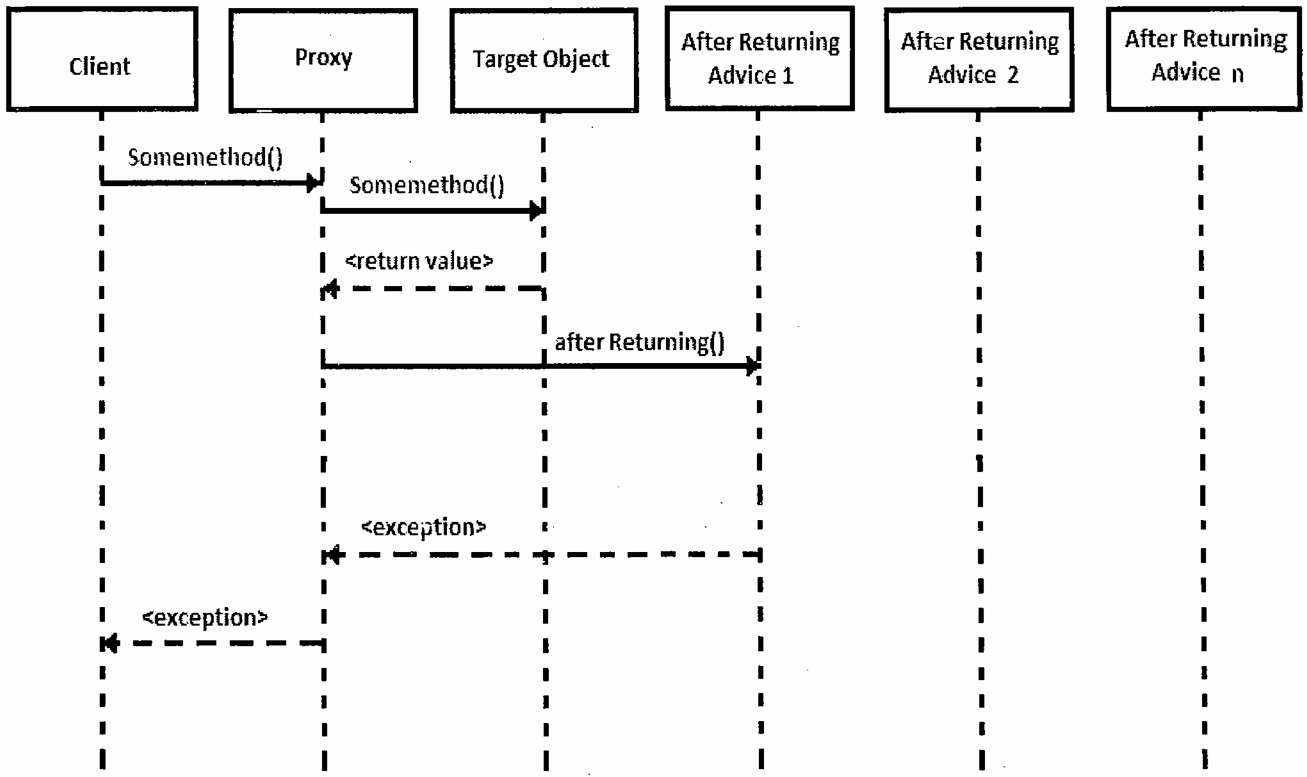
After Advice will be useful if some logic has to be executed before Returning the Control within a method execution. This advice is represented by the interface org.springframework.aop.AfterReturningAdvice. For example, it is common in Application to Delete the Session Data and the various information pertaining to a user, after he has logged out from the Application. These are ideal candidates for After Advice.

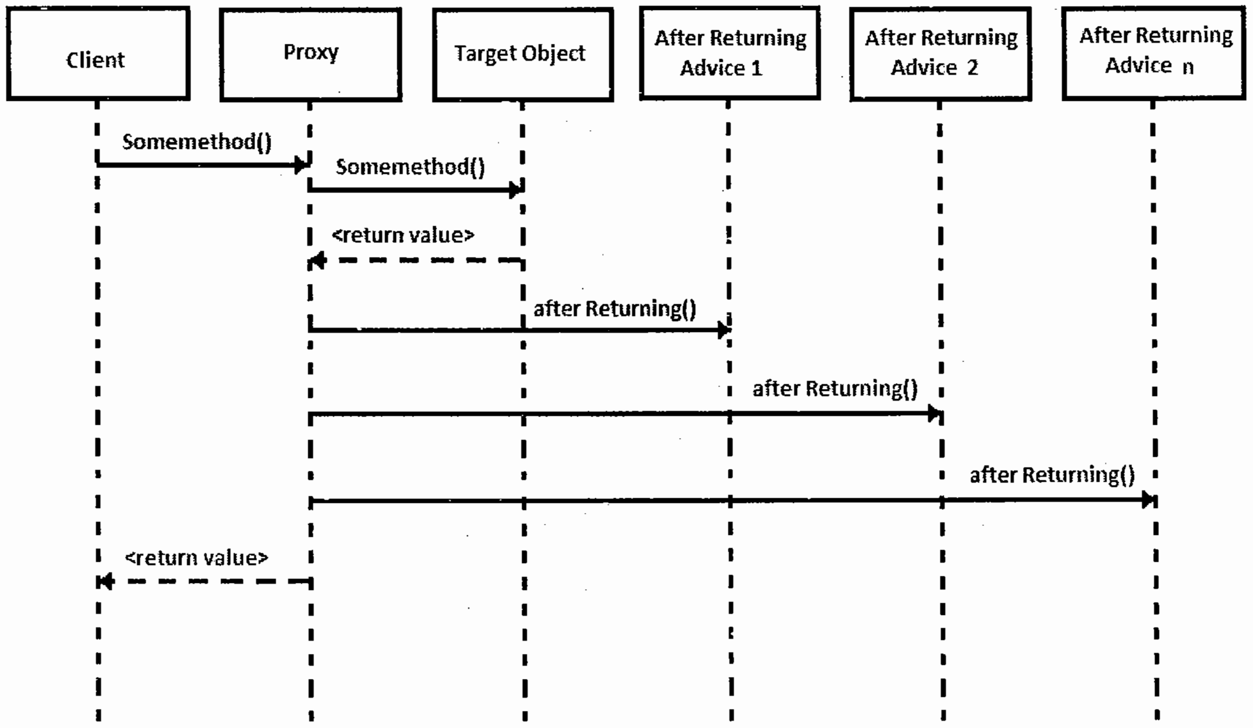
EX: CleanUpOperatoin.java

```
public class CleanUpOperation implements AfterReturningAdvice
{
Public void afterReturningObject returnValue, Method method, Object[] arg, Object target
throws Throwable
//Here afterReturning method represents the retun value given by the business method.
{
//clean up session and user information
}
}
```

Note that, afterReturning() method will be called once the method returns normal execution. If some exception happends in the method execution the afterReturning method will never be called.

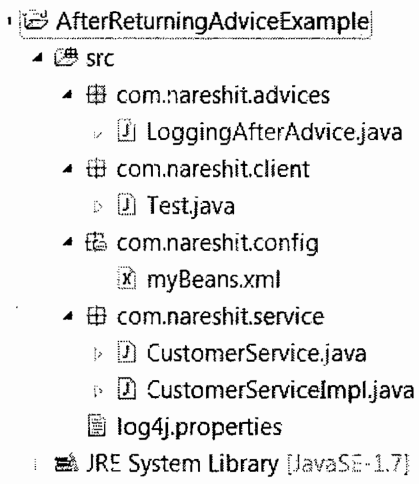
Sequence diagram for after returning advice :-





Working with After Returning Advice :-

In this section we will develop an example to demonstrate how to develop and configure afterReturning advice.



```
AfterReturningAdviceExample
  src
    com.nareshit.advices
      LoggingAfterAdvice.java
    com.nareshit.client
      Test.java
    com.nareshit.config
      myBeans.xml
    com.nareshit.service
      CustomerService.java
      CustomerServiceImpl.java
    log4j.properties
  JRE System Library [JavaSE-1.7]
```

CustomerService.java

```
package com.nareshit.service;
public interface CustomerService {
public  String printName();
public String printURL();
public void printException();
}
```

CustomerServiceImpl.java

```
package com.nareshit.service;

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

public class CustomerServiceImpl
implements CustomerService {
	@Value(value = "Naresh it")
private String name;
	@Value(value="www.nareshit.in")
private String url;


	public String printName() {
		System.out.println
		("Bussiness Method : printName() : "+name);
		return name;
	}

	public String printURL() {
		System.out.println
		("Bussiness Method : printURL() : "+url);
			return url;
	}
	public void printException()  {
		System.out.println("Bussiness Method : PrintException () :");
		throw new RuntimeException("some exception....");
	}

}
```

LoggingAfterAdvice.java

```
package com.nareshit.advices;
import java.lang.reflect.Method;
import org.apache.log4j.Logger;
import org.springframework.aop.AfterReturningAdvice;
public class LoggingAfterAdvice implements AfterReturningAdvice{
private static Logger logger=
Logger.getLogger(LoggingAfterAdvice.class);
        @Override
        public void afterReturning(Object returnValue,
                        Method method, Object[] arguments,
                        Object target) throws Throwable {
logger.info("After executing : "+target.getClass().getName()+" class Bussiness Method
:"+method.getName()+" return value : "+returnValue);
        }
}
```

myBeans.xml

```
<beans xmlns= "http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation=
        "http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd">
<context:annotation-config></context:annotation-config>
<bean id="loggingAfterAdvice" class="com.nareshit.advices.LoggingAfterAdvice"/>

<bean id="customerService" class="com.nareshit.service.CustomerServiceImpl"/>
<bean id="customerServiceProxy"
class="org.springframework.aop.framework.ProxyFactoryBean">
<property name="proxyInterfaces">
<value>com.nareshit.service.CustomerService</value>
</property>
<property name="interceptorNames">
<value>loggingAfterAdvice</value>
</property>
<property name="target" ref="customerService">
</property>
</bean>
</beans>
```

Test.java

```
package com.nareshit.client;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.nareshit.service.CustomerService;
public class Test {
```

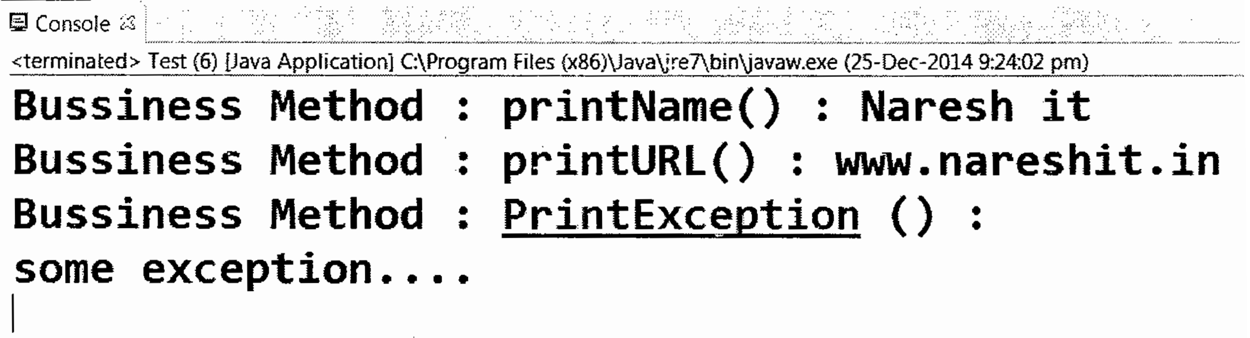
```
public static void main(String[] args){
ApplicationContext context=new
ClassPathXmlApplicationContext("com/nareshit/config/myBeans.xml");
CustomerService cs=(CustomerService)context.getBean("customerServiceProxy");
cs.printName();
cs.printURL();
try{
cs.printException();
}catch(Exception e){
        System.out.println(e.getMessage());
}
}
}
```

log4j.properties

```
log4j.rootLogger=info,springapp
log4j.appender.springapp=org.apache.log4j.FileAppender
log4j.appender.springapp.file=springaopexamplelog.html
log4j.appender.springapp.layout=org.apache.log4j.HTMLLayout
log4j.appender.springapp.append=true
```

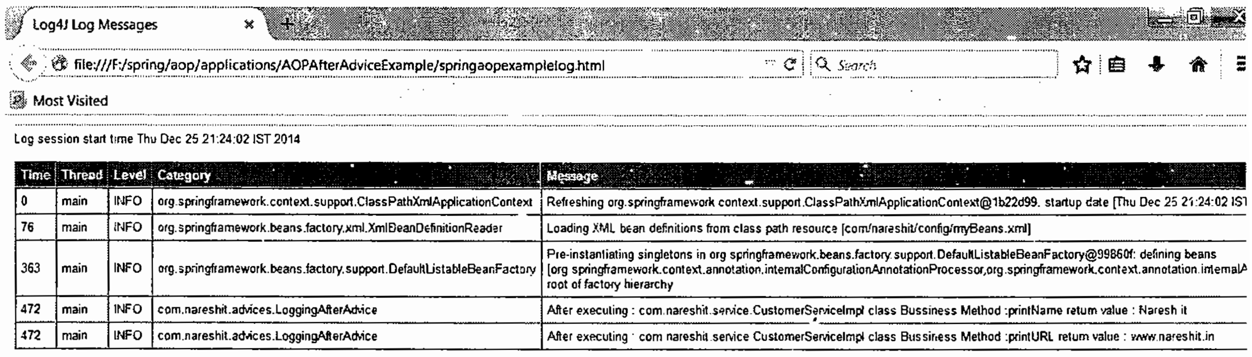
Upon execution of the above application, the log file will be generated and the output and logfile content as shown in the following fig's will be presented

output



```
Console
<terminated> Test (6) [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (25-Dec-2014 9:24:02 pm)
Bussiness Method : printName() : Naresh it
Bussiness Method : printURL() : www.nareshit.in
Bussiness Method : PrintException () :
some exception....
```

Log file :-



Log4J Log Messages

file:///F:/spring/aop/applications/AOPAfterAdviceExample/springaopexamplelog.html

Most Visited

Log session start time Thu Dec 25 21:24:02 IST 2014

Time	Thread	Level	Category	Message
0	main	INFO	org.springframework.context.support.ClassPathXmlApplicationContext	Refreshing org.springframework context.support.ClassPathXmlApplicationContext@1b22d99. startup date [Thu Dec 25 21:24:02 IST
76	main	INFO	org.springframework.beans.factory.xml.XmlBeanDefinitionReader	Loading XML bean definitions from class path resource [com/nareshit/config/myBeans.xml]
363	main	INFO	org.springframework.beans.factory.support.DefaultListableBeanFactory	Pre-instantiating singletons in org springframework.beans.factory support.DefaultListableBeanFactory@99860f: defining beans [org springframework.context.annotation.internalConfigurationAnnotationProcessor,org.springframework.context.annotation.internalA root of factory hierarchy
472	main	INFO	com.nareshit.advices.LoggingAfterAdvice	After executing : com.nareshit.service.CustomerServiceImpl class Bussiness Method :printName return value : Naresh it
472	main	INFO	com.nareshit.advices.LoggingAfterAdvice	After executing : com nareshit.service CustomerServiceImpl class Bussiness Method :printURL return value : www.nareshit.in

3. Throws Advice: -

When some kind of exception happens during the execution of a method, then to handle the exception properly, ThrowsAdvice can be used through the means of org.springframework.aop.ThrowsAdvice.

Note that this interface is a marker interface meaning that it doesn't have any method within it. The method signature inside the Throws Advice can take any of the following form.

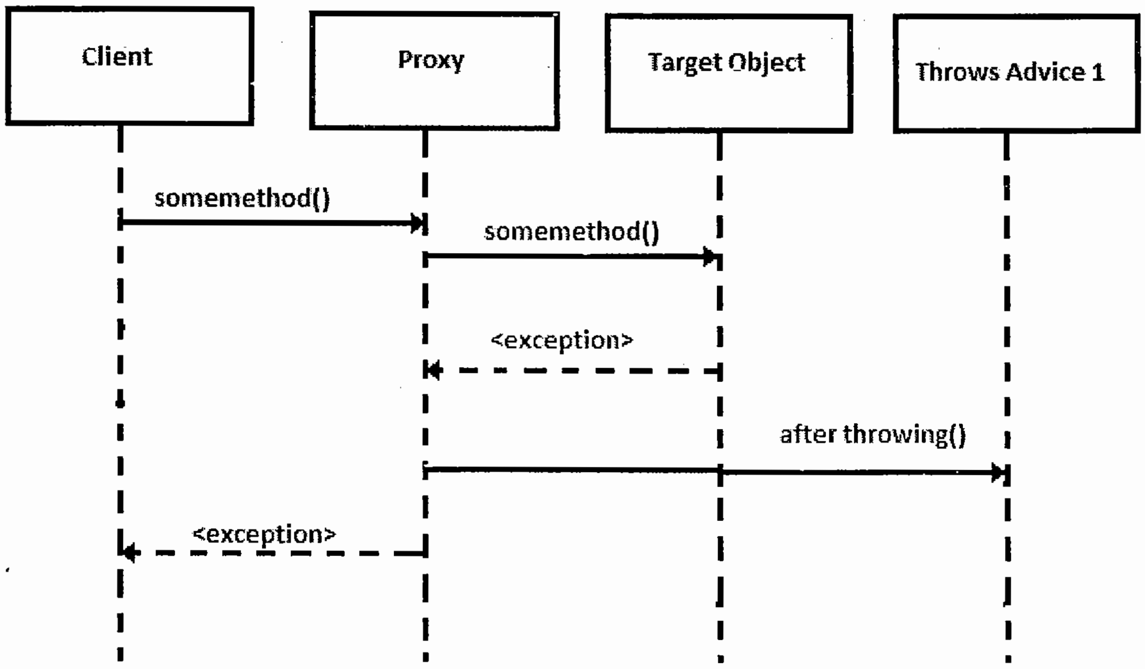
EX:
```
public class AfterThrowingException implements ThrowsAdvice{
public void afterThrowing(Method method, Object[] args, Object target, Exception exception){
//here afterThrowing method represents the exception raised in the business method.
}
}
```

For example, in a File Copy program, if some kind of exception happens in the mid-way then the newly created target file has to be deleted as the partial content in the file doesn't carry any sensible meaning. It can be easily achieved through the means of Throws Advice.

DeleteFile.java

```
public class DeleteFile implements ThrowsAdvice{
public void afterThrowing(Method method, Object[] args, Object target,
   IOException exception){
 String targetFileName = (String)args[2];
     // Code to delete the target file.
   }
        }
```

Sequence diagram for throws advice :-



AroundAdvice:

The Advice is very different from the other types of the advices because the advice allows us to

perform operations before and after the execution of Joinpoint.This is very useful if we want to share the state before and after the joinPoint execution in a thread-safe manner like starting the transactions before the joinpoint execution and ending it after the joinPoint execution .
The spring around advice is compliant with AOP alliance that provides interpretability with other AOP alliance compliant AOP implementations.
This Advice has to be a SubType of **org.aopalliance.intercept.MethodInterceptor** interface

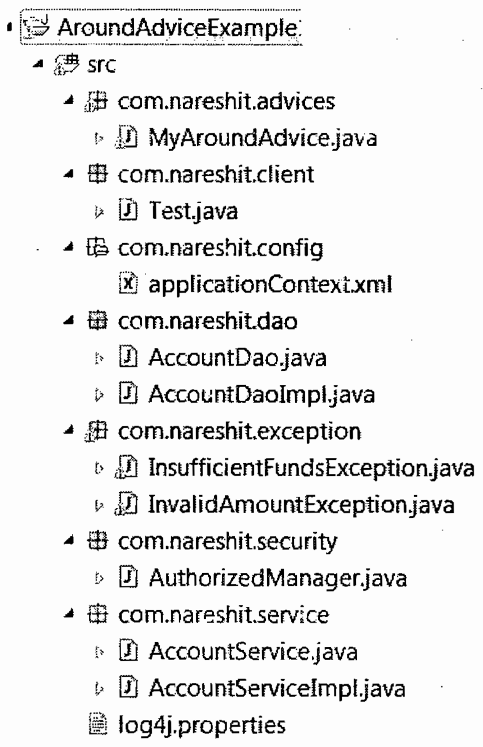
The **org.aopalliance.intercept.MethodInterceptor** interface has only one method with the Following signature

public Object invoke(MethodInvocation invocation) throws Throwable

The **org.aopalliance.intercept.MethodInterceptor** interface allows us to get the Details of the invoked method of the target Object,method arguments ,joinPoint. The MethodInvocation Interface extends Invocation,which is a subtype of JoinPoint.

Working with Around Advice :-

In this section we will develop an example to demonstrate how to develop and configure AroundAdvice.



```
AroundAdviceExample
  src
    com.nareshit.advices
      MyAroundAdvice.java
    com.nareshit.client
      Test.java
    com.nareshit.config
      applicationContext.xml
    com.nareshit.dao
      AccountDao.java
      AccountDaoImpl.java
    com.nareshit.exception
      InsufficientFundsException.java
      InvalidAmountException.java
    com.nareshit.security
      AuthorizedManager.java
    com.nareshit.service
      AccountService.java
      AccountServiceImpl.java
    log4j.properties
```

AccountService.java

```
package com.nareshit.service;
public interface AccountService {
int withdraw(int accno,double amt);
int deposit(int accno,double amt);
}
```

AccountServiceImpl.java

```
package com.nareshit.service;
```

```
import com.nareshit.dao.AccountDao;
import com.nareshit.exception.InsufficientFundsException;
import com.nareshit.exception.InvalidAmountException;
public class AccountServiceImpl implements AccountService {
private AccountDao accountDao;
public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
}

public int withdraw(int accno,double amt) {
        int count=0;
 double totalbalance=accountDao.getBalance(accno);
 if(totalbalance>=amt){
    totalbalance=totalbalance-amt;
    count=accountDao.setBalance(accno,totalbalance);
 }
 else{
         throw new InsufficientFundsException
         ("Insufficient Funds in your account :"+accno);
 }
 return count;  }
public int deposit(int accno,double amt) {
        int count=0;
 double totalbalance=accountDao.getBalance(accno);
            if(amt>0){
                    totalbalance=totalbalance+amt;
                    count=accountDao.setBalance(accno,totalbalance);
            }
            else{
      throw new InvalidAmountException(" amount "+amt+ "is Invalid ");
            }
 return count;
}
}
```

AccountDao.java

```
package com.nareshit.dao;
public interface AccountDao {
int setBalance(int accno,double amt);
double getBalance(int accno);
}
```

AccountDaoImpl.java

```
package com.nareshit.dao;
import org.springframework.jdbc.core.JdbcTemplate;
public class AccountDaoImpl   implements AccountDao{
private JdbcTemplate jdbcTemplate;
public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
}
public int setBalance(int accno,double newbalance) {
String sql="update account set bal=? where accountId=?";
int count =jdbcTemplate.update(sql,newbalance,accno);
return count;
        }
public double getBalance(int accno) {
String sql="select bal from account where accountId=?";
double balance=jdbcTemplate.queryForObject(sql,Double.class,accno);
                    return balance;
        }
}
```

InsufficientFundsException.java

```
package com.nareshit.exception;
public class InsufficientFundsException extends RuntimeException {
public InsufficientFundsException(String msg){
        super(msg);
}
}
```

InvalidAmountException.java

```
package com.nareshit.exception;
public class InvalidAmountException extends RuntimeException {
public InvalidAmountException(String msg){
        super(msg);
}
}
```

AuthorizedManager.java

```
package com.nareshit.security;
public class AuthorizedManager {
public boolean isAuthorized(String userRole){
        boolean flag=false;
  if(userRole!=null && userRole.equals("BankAdmin")){
   return true;
  }
return flag;
}
}
```

MyAroundAdvice.java

```
package com.nareshit.advices;
import java.lang.reflect.Method;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
import org.apache.log4j.Logger;
import com.nareshit.security.AuthorizedManager;
public class MyAroundAdvice implements
MethodInterceptor{
public Object invoke
(MethodInvocation invocation) throws Throwable {
Object target=invocation.getThis();
Method method=invocation.getMethod();
Logger logger=Logger.getLogger(target.getClass().getName());
logger.info("Entered into :"+target.getClass().getName()+" class  "+method.getName()+"
method ");
Integer count=0;
try{
        AuthorizedManager am=new AuthorizedManager();
        boolean flag=am.isAuthorized("BankAdmin");
        if(flag==faise){
                throw new IllegalAccessException(" User is Not Authorised ");
        }
        count=(Integer) invocation.proceed();
if(count>0){
logger.info(method.getName() +" operation Completed Successfully ");
}
else{
logger.info(method.getName() +" operation Failure...");
}
}catch(Exception exception){
logger.info
("while executing :"+method.getName()+" Exception Occured : "+exception.getMessage());
        logger.info(method.getName()+" Operation Failure ");
}
logger.info(method.getName()+ " method Execution completed ");

return count;
        }

}
```

applicationContext.xml

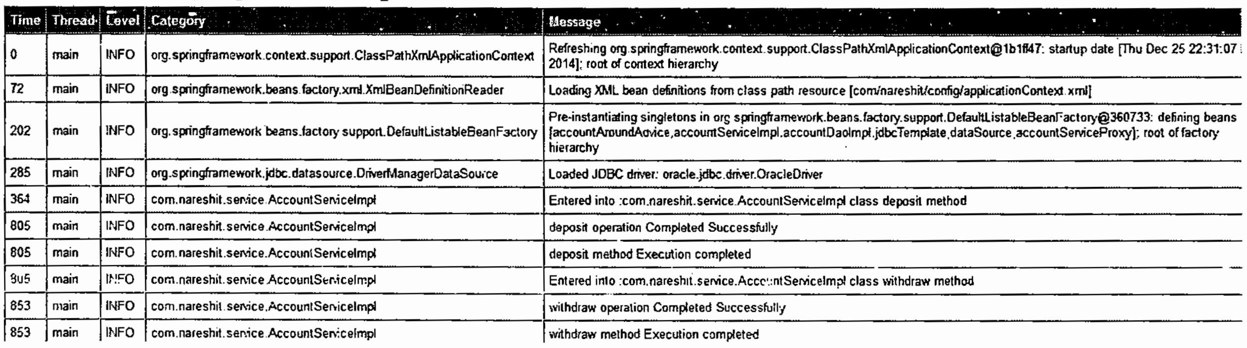
```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                        "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
<bean id="accountAroundAdvice"  class="com.nareshit.advices.MyAroundAdvice">
</bean>
<bean id="accountServiceImpl" class="com.nareshit.service.AccountServiceImpl">
<property name="accountDao" ref="accountDaoImpl"/>
</bean>
<bean id="accountDaoImpl" class="com.nareshit.dao.AccountDaoImpl">
<property name="jdbcTemplate" ref="jdbcTemplate"/>
</bean>
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
<property name="dataSource" ref="dataSource"/>
</bean>
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
```

```
<property name="url" value="jdbc:oracle:thin:@localhost:1521:XE"/>
<property name="username" value="system"/>
<property name="password" value="manager"/>
</bean>
<bean id="accountServiceProxy"
class="org.springframework.aop.framework.ProxyFactoryBean">
<property name="proxyInterfaces" value="com.nareshit.service.AccountService"/>
<property name="interceptorNames" value="accountAroundAdvice"/>
<property name="target" ref="accountServiceImpl"/>
</bean>
</beans>
```

Test.java

```
package com.nareshit.client;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.nareshit.service.AccountService;
public class Test {
public static void main(String[] args){
ApplicationContext context=new ClassPathXmlApplicationContext
("com/nareshit/config/applicationContext.xml");
AccountService accountService=
(AccountService)context.getBean("accountServiceProxy");
   accountService.deposit(3001,6000);
System.out.println("===================");
accountService.withdraw(3001,4000);
}
}
```

Upon execution of the above application, the log file will be generated logfile content as shown in the following fig's will be presented .

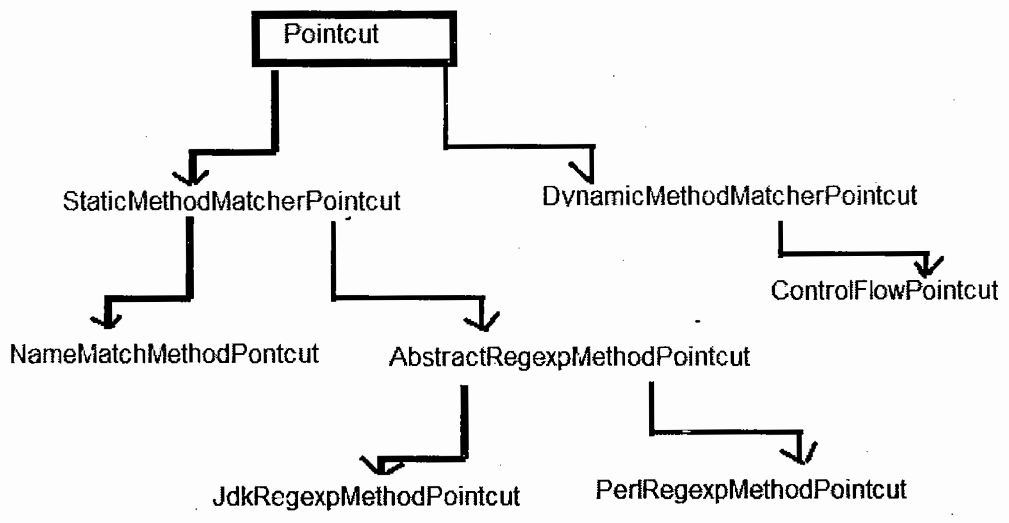


Time	Thread	Level	Category	Message
0	main	INFO	org.springframework.context.support.ClassPathXmlApplicationContext	Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@1b1f47: startup date [Thu Dec 25 22:31:07 2014]; root of context hierarchy
72	main	INFO	org.springframework.beans.factory.xml.XmlBeanDefinitionReader	Loading XML bean definitions from class path resource [com/nareshit/config/applicationContext.xml]
202	main	INFO	org.springframework.beans.factory.support.DefaultListableBeanFactory	Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@360733: defining beans [accountAroundAdvice,accountServiceImpl,accountDaoImpl,jdbcTemplate,dataSource,accountServiceProxy]; root of factory hierarchy
285	main	INFO	org.springframework.jdbc.datasource.DriverManagerDataSource	Loaded JDBC driver: oracle.jdbc.driver.OracleDriver
364	main	INFO	com.nareshit.service.AccountServiceImpl	Entered into :com.nareshit.service.AccountServiceImpl class deposit method
805	main	INFO	com.nareshit.service.AccountServiceImpl	deposit operation Completed Successfully
805	main	INFO	com.nareshit.service.AccountServiceImpl	deposit method Execution completed
805	main	INFO	com.nareshit.service.AccountServiceImpl	Entered into :com.nareshit.service.AccountServiceImpl class withdraw method
853	main	INFO	com.nareshit.service.AccountServiceImpl	withdraw operation Completed Successfully
853	main	INFO	com.nareshit.service.AccountServiceImpl	withdraw method Execution completed

Creating Point Cuts in Spring

Point Cuts define where exactly the Advices have to be applied in various Join Points. Generally they act as Filters for the application of various Advices into the real implementation.

Spring defines two types of PointCuts namely the static and the DynamicPointCuts.
Static point cut verifies whether that join point has to be advised (or) not, only once the result is catched & reused.
Dynamic pointcut verifies every time as it has to decide the join point based on the argument passed to method call.



Pointcut is an interface present in org.springframework.aop package

```
package org.springframework.aop;

public interface Pointcut {
        ClassFilter getClassFilter();
        MethodMatcher getMethodMatcher();
}
```

The class filter is a special type of object that describes which classes have potential joinpoints (read methods) on them. The method matcher simply describes which methods for a given class are considered valid joinpoints for this pointcut.

NameMatchMethodPointcut :-

NameMatchMethodPointcut class is going to verify whether the method names of a spring bean class are matching with the given criteria or not. While configuring this pointcut class into xml, we use mappedName or mappedNames property of the class into xml file. This class is a predefined class, so we need to configure the class directly into xml file like..

```
<bean id="pointcutBean" class="org.spfw.aop.support.NameMatchMethodPointcut">
   <property name="mappedName">
       <value>method1</value>
   </property>
</bean>
```

Means the pointcut class identifies that method1 of the bean class are eligible for getting advices. If there is no commonalities in method names we can provide individual method names by configuring mappedNames property.

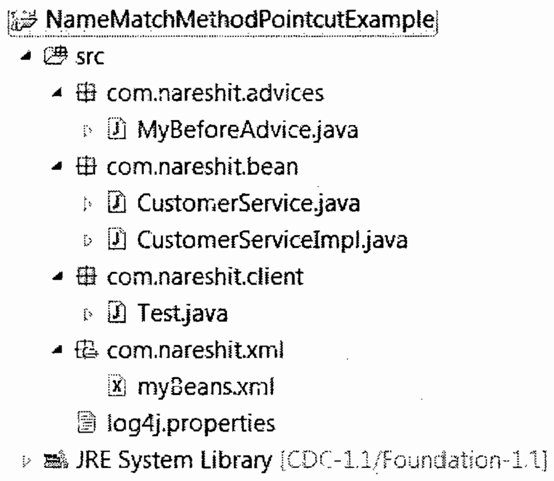
mappedNames is String[] type:-), so we need <list>—-</list> element while configuring the collection

```
<bean id=" pointcutBean " class="org.spfw.aop.support.NameMatchMethodPointcut">
   <property name="mappedNames">
```

```
        <list>
           <value>method1</value>
           <value>method2</value>
           <value>method3</value>
        </list>
   </property>
</bean>
```

Working with NameMatchMethodPointcut:-

In this section we will develop an example to demonstrate how to develop and configure NameMatchMethodPointcut



```
NameMatchMethodPointcutExample
  src
    com.nareshit.advices
      MyBeforeAdvice.java
    com.nareshit.bean
      CustomerService.java
      CustomerServiceImpl.java
    com.nareshit.client
      Test.java
    com.nareshit.xml
      myBeans.xml
    log4j.properties
  JRE System Library [CDC-1.1/Foundation-1.1]
```

CustomerService.java

```
package com.nareshit.bean;
public interface CustomerService {
public String printName();
public String printURL();
public void printException();
}
```

CustomerServiceImpl.java

```
package com.nareshit.bean;
public class CustomerServiceImpl implements CustomerService {
        private String name;
        private String url;
        public String printName() {
                System.out.println("printName :" + name);
                return name;
        }
        public String printURL() {
                System.out.println("printURL :" + url);
                return url;
        }
        public void printException() {
                System.out.println("PrintException ");
                throw new RuntimeException("Don't sleep in the class room");

        }
        public void setName(String name) {
                this.name = name;
        }
        public void setUrl(String url) {
                this.url = url;
        }
}
```

MyBeforeAdvice.java

```
package com.nareshit.advices;
import java.lang.reflect.Method;
import org.apache.log4j.Logger;
import org.springframework.aop.MethodBeforeAdvice;
public class MyBeforeAdvice
implements MethodBeforeAdvice {
private Logger logger=Logger.getLogger(MyBeforeAdvice.class);
public void before(Method method,  Object[] obj, Object target)throws Throwable {
logger.info("Entered into "+target.getClass().getName()+
"  class  Method "+method.getName()+" with arguments : "
                +obj.length);
}
}
```

myBeans.xml

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                        "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
<bean id="customerService"  class="com.nareshit.bean.CustomerServiceImpl">
<property name="name" value="sathish"/>
<property name="url" value="www.nareshit.in"/>
</bean>
<bean id="beforeAdvice"  class="com.nareshit.advices.MyBeforeAdvice">
</bean>
<!-- <bean id="nameMatchMethodPointcut"
class="org.springframework.aop.support.NameMatchMethodPointcut">
<property name="mappedName" value="printURL"/>
</bean>
<bean id="nameMatchMethodAdvisor"
class="org.springframework.aop.support.DefaultPointcutAdvisor">
<property name="pointcut" ref="nameMatchMethod"/>
<property name="advice" ref="beforeAdvice"/>
</bean> -->
  <bean id="nameMatchMethodAdvisor"
class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
 <!-- <property name="mappedName" value="printURL"></property> -->
 <property name="mappedNames">
 <list>
 <value>printName</value>
 <value>printURL</value>
 </list>
 </property>
 <property name="advice" ref="beforeAdvice"/>
 </bean>
<bean id="customerServiceProxy"
class="org.springframework.aop.framework.ProxyFactoryBean">
<property name="target"  ref="customerService"/>
<property name="proxyInterfaces" value="com.nareshit.bean.CustomerService"/>
<property name="interceptorNames" value=" nameMatchMethodAdvisor "/>
</bean>
</beans>
```

Test.java

```
package com.nareshit.client;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.nareshit.bean.CustomerService;
public class Test {
public static void main(String[] args){
ApplicationContext context=new
ClassPathXmlApplicationContext("com/nareshit/xml/myBeans.xml");
CustomerService cs=(CustomerService)context.getBean("customerServiceProxy");
        cs.printName();
        cs.printURL();
        try{
        cs.printException();
        }
        catch(Exception e){

        }
}
}
```

RegexpMethodPointcut:-

We can configure the RegexpMethodPointCut Advisor as follows

```
<bean id="regexpPointcutAdvisor"
class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
<property name="advice" ref="beforeAdvice"/>
<property name="patterns">
<list>
   <value>.*Exception.*</value>
   <value>.*URL.*</value>
   </list> </property>
</bean>
```

There are many frameworks in the market which allows you to work with AOP programming examples Spring AOP,AspectJ,JAC(javaAspect Components),Jboss AOP etc...The most populare ones is AspectJ and Spring AOP.

Difference between Spring AOP and Aspectj AOP.

Spring AOP	Aspectj AOP
Only supported Joinpoint is method execution	It supports Various types of JoinPoints like constructor execution, method execution,field set (OR) filed get etc...
Spring supports runtime weaving, this means the proxy object's will be built on fly at Runtime in the memory.	AspectJ uses compile-time weaving,this indicates your proxy classes will be availiable whenever you compile your code.
Spring supports static and dynamic pointcuts	Aspectj supports only static pointcuts

Note :-

Even Spring has integrated with Aspectj,it supports only few features of Aspectj described as below.
1) Still the supported Joinpoints is only a method execution
2) As Aspectj doesn't support dynamic pointcuts,integrating with Spring doesn't make any difference

At the time Spring released AOP, already AspectJ AOP has acceptance in the market, and seems to be more powerful than Spring AOP,Spring developers instead of comparing the strengths of each framework, they have provided integrations to AspectJ to take the advantage of it,so spring 2.x not only supports AOP it allows us to work with AspectJ AOP as well.

In Spring 2.x it provided two ways of working with Aspectj intergrations,

1) Declarative Programming Model

2)AspectJ annotation model.

Spring 2.0 also offers support for defining aspects using the new aop

namespace tags. The main problem with programmatic approach is your application code will

tightly couple with Spring,so that you can not detach from spring. If you use declarative approach,your advice(OR) aspect class are pojo's,you don't need to implement (OR) extend from any spring specific interface (OR) class.

Your advice classes uses AspectJ AOP classes,in order to declare the pojo as aspect you need to

declare it in the configuration file rather than weaving it using programmatic approach.

The declarative approach also supports all the four types of advices and static pointcut.

In the following section we will explore through the example how to work with various types of advices declaratively. To use the aop namespace tags we need to import the spring-aop schema

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation=
       "http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
```

Within your Spring configurations, all aspect and advisor elements must be placed within an <aop:config> element (you can have more than one <aop:config> element in an application context configuration).

An <aop:config> element can contain pointcut, advisor, and aspect elements (note these must be declared in that order).

Declaring an aspect:-

By using the schema support,an aspect is simply a regular java object defined as a bean in your spring application context. The state and behaviour is captured in the fields and methods of the object,and the an aspect is declared using the <aop:aspect> element.

Declaring a pointcut:-

a named pointcut can be declared inside an <aop:config>element, enabling the pointcut defination to be shared across several aspects and advisors.

beforeAdvice:- before advice runs before a matched method execution.It is declared inside an <aop:aspect> using the <aop:before>element

After returning advice:- After returning advice runs when a matched method execution completes normally. It is declared inside an <aop:aspect> using the <aop:after- returning> element

After advice : After advice runs however a matched method execution exits. It is declared inside an <aop:aspect> by using the <aop:after> element:

AfterThrowing Advice:- After throwing advice executes when a matched method

execution exits by throwing an exception. It is declared inside an <aop:aspect> using the

<aop:after-throwing> element.

aroundAdvice:-

Around advice runs "around" a matched method execution. It has the opportunity to do work both before and after the method executes, and to determine when, how, and even if, the method actually gets to execute at all. Around advice is often used if you need to share state before and after a method execution in a thread-safe manner (starting and stopping a timer for example). Always use the least powerful form of advice that meets your requirements; don't use around advice if simple before advice would do. Around advice is declared using the <aop:around> element. The first parameter of the advice method must be of type ProceedingJoinPoint. Within the body of the advice, calling proceed() on the ProceedingJoinPoint causes the underlying method to execute. The proceed method may also be

calling passing in an Object[] - the values in the array will be used as the arguments to the method execution when it proceeds.

Example:

```
<aop:aspect id="aroundExample"   ref="myAroundAdvice ">

   <aop:around  pointcut-ref="businessService"  method="doBasicProfiling"/>

      </aop:aspect>
```

The implementation of the doBasicProfiling advice is :-

```
public Object doBasicProfiling (ProceedingJoinPoint pjp) throws Throwable {

   // start stopwatch

   Object retVal = pjp.proceed();

   // stop stopwatch

   return retVal;

}
```

Note :-
Spring Aop supports the following aspect pointcut designators for use in pointcut expressions:
execution:- for matching method execution join points
within:- limits matching to join points with in certian types
this:- limits matching to join points(the execution of methods when using spring aop)
target :- limits matching to join points(the execution of methods when using spring aop) when the target object(application object being proxied) is an instance of given type
Examples:-
the execution of any public method
execution(public * *(..))

the execution of any method with a name begining with set

execution(* set*(..))

the execution of any method defined by the customer service interface execution(* com.nareshit.CustomerService.*(..))

the execution of any method defined in the service package :-

execution(* com.nareshit.service.*.*(..))

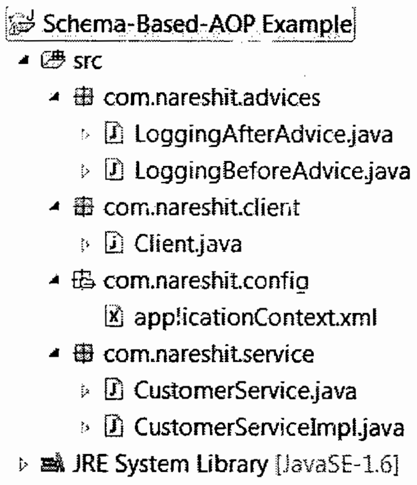
the execution of any method defined in the service package or a sub-package:
execution(* com.nareshit.service..*.*(..))

any join point (method execution only in Spring AOP) within the service package:

within(com.nareshit.service.*)

any join point (method execution only in Spring AOP) within the service package or a sub package:

within(com.nareshit.service..*)



```
Schema-Based-AOP Example
  src
    com.nareshit.advices
      LoggingAfterAdvice.java
      LoggingBeforeAdvice.java
    com.nareshit.client
      Client.java
    com.nareshit.config
      applicationContext.xml
    com.nareshit.service
      CustomerService.java
      CustomerServiceImpl.java
  JRE System Library [JavaSE-1.6]
```

CustomerService.java

```
package com.nareshit.service;
public interface CustomerService {
        String printName();
        void printUrl();
        void printException();

}
```

CustomerServiceImpl.java

```
package com.nareshit.service;
public class CustomerServiceImpl implements CustomerService{
private String name;
private String url;

        public String getName() {
```

```
        return name;
}

public void setName(String name) {
        this.name = name;
}

public String getUrl() {
        return url;
}

public void setUrl(String url) {
        this.url = url;
}


        public void printException() {
throw new RuntimeException("Custom Exception");

        }
public String printName() {

System.out.println("Bussiness Method:printName()=>"+name);
                return name;
        }
public void printUrl() {

System.out.println("Bussiness Method:printUrl()=>"+url);

        }

}
```

LoggingBeforeAdvice.java

```
package com.nareshit.advices;
import org.aspectj.lang.JoinPoint;
public class LoggingBeforeAdvice {
public void myBefore(JoinPoint joinPoint) {
System.out.println(" Entered into  :"+joinPoint.getTarget().getClass().getName()+" class ,
"+joinPoint.getSignature().getName()+" method");
}
}
```

LoggingAfterAdvice.java

```
package com.nareshit.advices;
import org.aspectj.lang.JoinPoint;
public class LoggingAfterAdvice {
public void myAfter(JoinPoint joinPoint,Object returnValue){
System.out.println(" Execution completed on  :"+joinPoint.getTarget().getClass().getName()+"
class , "+joinPoint.getSignature().getName()+" method  and return value is :"+returnValue);
```

```
        }
        }
```

applicationContext.xml

```
  <beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xsi:schemaLocation=
        "http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
<bean id="loggingBeforeAdvice"
class="com.nareshit.advices.LoggingBeforeAdvice"/>
<bean id="loggingAfterAdvice"
class="com.nareshit.advices.LoggingAfterAdvice"/>
<bean id="customerService" class="com.nareshit.service.CustomerServiceImpl">
<property name="name" value="sathish"/>
<property name="url" value="www.nareshit.in"/>
</bean>
<aop:config>
<aop:pointcut  expression= "within(com.nareshit.service.CustomerServiceImpl)"

id="myPointcutRef"/>

<aop:aspect ref="loggingBeforeAdvice">

<aop:before method="myBefore"

pointcut-ref="myPointcutRef"/>

</aop:aspect>

<aop:aspect ref="loggingAfterAdvice">

<aop:after-returning method="myAfter"

pointcut-ref="myPointcutRef"

 returning="returnValue"/>

</aop:aspect>

</aop:config>

</beans>
```

Client.java

```
package com.nareshit.client;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.nareshit.service.CustomerService;
public class Client {
	public static void main(String[] args) {
		ApplicationContext ctx = new ClassPathXmlApplicationContext(
				"com/nareshit/config/applicationContext.xml");
		CustomerService service = (CustomerService) ctx
				.getBean("customerService");
		service.printName();
		service.printUrl();
		try {
			service.printException();
		} catch (Exception e) {
			System.out.println(e);
		}
	}
}
```

We can develop the AroundAdvice as follows:-

```
import org.apache.log4j.Logger;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.Signature;

public class AroundAdvice {
	private static Logger logger = Logger.getLogger(AroundAdvice.class);

	public Object myAround(ProceedingJoinPoint joinPoint) {
		Object target = joinPoint.getTarget();
		Signature signatue = joinPoint.getSignature();
		Object[] args = joinPoint.getArgs();
		logger.info("Entered into :" + target.getClass().getName() + " class "
				+ signatue.getName() + " method with No.of args :"
				+ args.length);
		Object returnValue = null;
		try {

			returnValue = joinPoint.proceed();
		} catch (Throwable exception) {
			logger.info("Exception occured while executing  "
					+ signatue.getName() + " :: " + exception.getMessage());
		}
		if (returnValue != null) {
			logger.info("Respose of " + signatue.getName() + " method ::"
					+ returnValue);
		}
		return returnValue;
	}

}
```

In the cfg file :-

```
<bean id="aroundAdvice" class="AroundAdvice"/>

<aop:config>
<aop:pointcut expression= "within(com.nareshit.service.CustomerServiceImpl)"
 id="pointcutRef"/>
<aop:aspect ref="aroundAdvice">
<aop:around method="myAround" pointcut-ref=" pointcutRef "/>
</aop:aspect>
</aop:config>
```

AspectJ AnnotationBased Aop:-

In this approach instead of using declarations to expose the classes as aspect and advice, we will

annotate the classes with annotations.In order to make as class aspect,we need to annotate the class with @Aspect. To expose a method as advice method,we need to annotate the method in aspect class @Around,(OR) @Before (OR) @AfterReturnining (OR)@AfterThrowing representing the type of advice.

These annotations take the pointcut expression as value them.

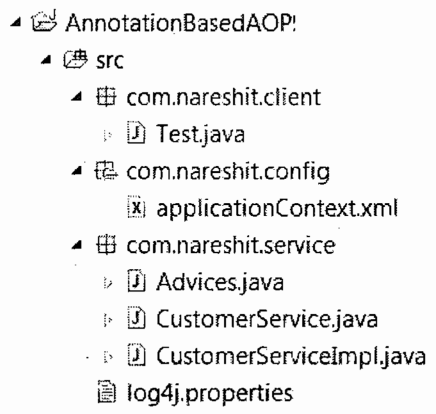
Note :- In order to detect the annotations marked on the classes ,we need to add

<aop:aspectj- autoproxy/> tag in the configuration.

(OR)

If we are working with configuration class instead of configuration file

```
@Configuration

@EnableAspectJAutoProxy

public class MyBeans {



}
```



```
AnnotationBasedAOP!
  src
    com.nareshit.client
      Test.java
    com.nareshit.config
      applicationContext.xml
    com.nareshit.service
      Advices.java
      CustomerService.java
      CustomerServiceImpl.java
    log4j.properties
```

CustomerService.java

```java
package com.nareshit.service;
public interface CustomerService {
        String displayName();
        String printUrl();
        void printException();
}
```

CustomerServiceImpl.java

```java
package com.nareshit.service;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
@Service("customerService")
public class CustomerServiceImpl  implements CustomerService{
@Value("sathish")
private String name;
@Value("www.nareshit.in")
private String url;
public void printException() {
System.out.println("Bussiness Method:printException()");
throw new RuntimeException("Custom Exception");
        }
public String displayName() {
System.out.println("Bussiness Method:displayName()=>"+name);
                return name;
        }
public String printUrl() {
System.out.println("Bussiness Method:printUrl()=>"+url);
                return url;
        }
}
```

Advices.java

```java
package com.nareshit.service;
import org.apache.log4j.Logger;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Service;
@Service("advices")
@Aspect
public class Advices {
        private  static Logger logger=Logger.getLogger(Advices.class);
        @Before("within(com.nareshit.service.CustomerServiceImpl)")
        public void myBefore(JoinPoint joinPoint){
        logger.info("Entered into "+joinPoint.getTarget().getClass().getName()+" class " );
        logger.info
        ("Method :"+joinPoint.getSignature().getName());

        }
        @AfterReturning(pointcut="execution(* print*(..))",
                    returning="result")
public void myAfterReturning(JoinPoint joinPoint,
                Object result){

        logger.info(" Execution Successfully completed
:"+joinPoint.getSignature().getName()+" And Return value "+result);
}
        @AfterThrowing(pointcut=
        "within(com.nareshit.service.CustomerServiceImpl)",
        throwing="exception")
        public void myAfterThrowing(JoinPoint joinPoint,
                    Throwable exception){
                logger.info(" execption occured while executing
:"+joinPoint.getSignature().getName()+" "+exception.getMessage());
        }

}
```

applicationContext.xml

```
<beans xmlns= "http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation=
        "http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd
        ">

<context:component-scan base-package="com.nareshit"></context:component-scan>
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

```
</beans>
```

Test.java

```
package com.nareshit.client;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.nareshit.service.CustomerService;
public class Test {
public static void main(String[] args){
        ApplicationContext context=new
ClassPathXmlApplicationContext("com/nareshit/config/applicationContext.xml");
CustomerService customerService=(CustomerService)context.getBean("customerService");
        customerService.displayName();
        customerService.printUrl();
        try{
        customerService.printException();
        }catch(Exception e){

        }
}
}
```