HIBERNATE

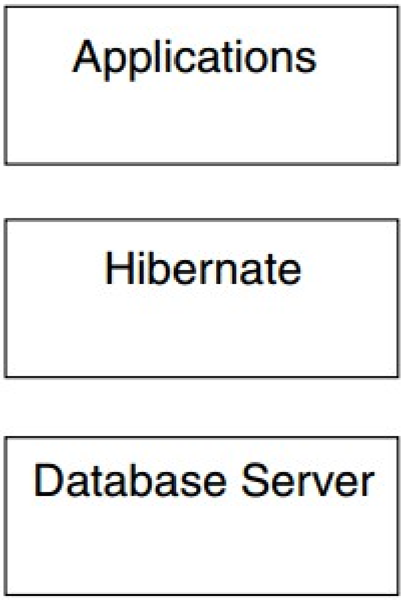
Introduction to Hibernate

- Hibernate is Java-based middleware designed to complete the Object Relational (O/R) mapping model.

- For the most part, you'll see little disruption to your classes and code as a result of using Hibernate.

- In fact, one of the most complex parts of many O/R mapping mechanisms—writing SQL code—can be simplified significantly with Hibernate.

- Figure shows a high-level illustration of where Hibernate fits into the development picture.



- In the figure, you can see that Hibernate sits between traditional Java objects and handles all the work in persisting those objects based on the appropriate O/R mechanisms and patterns.
- Your goal is to accurately map how Hibernate will persist objects to the database tables of the underlying database server; in most cases, the mapping exercise is straightforward.
- Hibernate can persist even the most complex classes, including those using composition, inheritance, and collections.
- Vital to the functionality of Hibernate is a database server and the connection between itself and the server.
- Fortunately, Hibernate is rich in this area: It supports a variety of the most popular open-source and commercial servers.

- However, Hibernate isn't limited to these servers; you can use just about any server, assuming an appropriate JDBC driver is available.

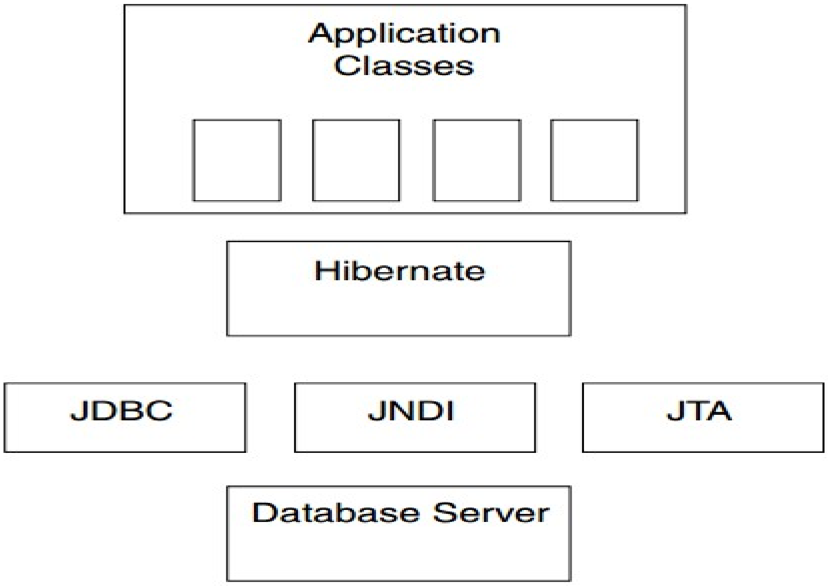
- Hibernate supports basic SQL and allows individual dialect changes to be made as appropriate for the database being used in the application. Thus even if you're using a database that includes some quirky syntax, Hibernate can support it using a specific dialect class.

- Figureshows that JDBC isn't the only additional technology used between Hibernate and the database server.

- Hibernate can be used with applications that rely on an application server to handle servlets and Enterprise JavaBeans. In these situations, it's normal to use Java Naming and Directory Interface (JNDI) to handle the database resource associated with the environment.

- Once a JNDI resource has been created, Hibernate can access the resource with just a few changes to its configuration file.

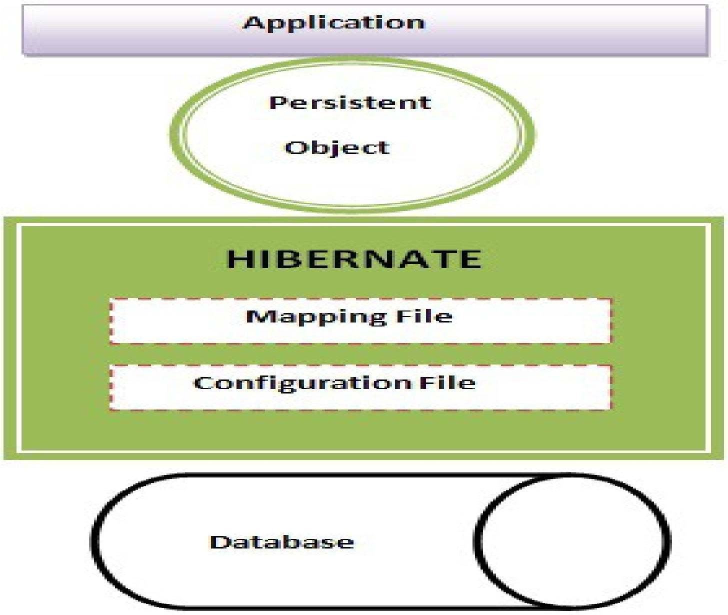
- Further, transaction support is available through both the JDBC connection and the Java Transaction API (JTA) specification and its implementation classes.



Hibernate Architecture

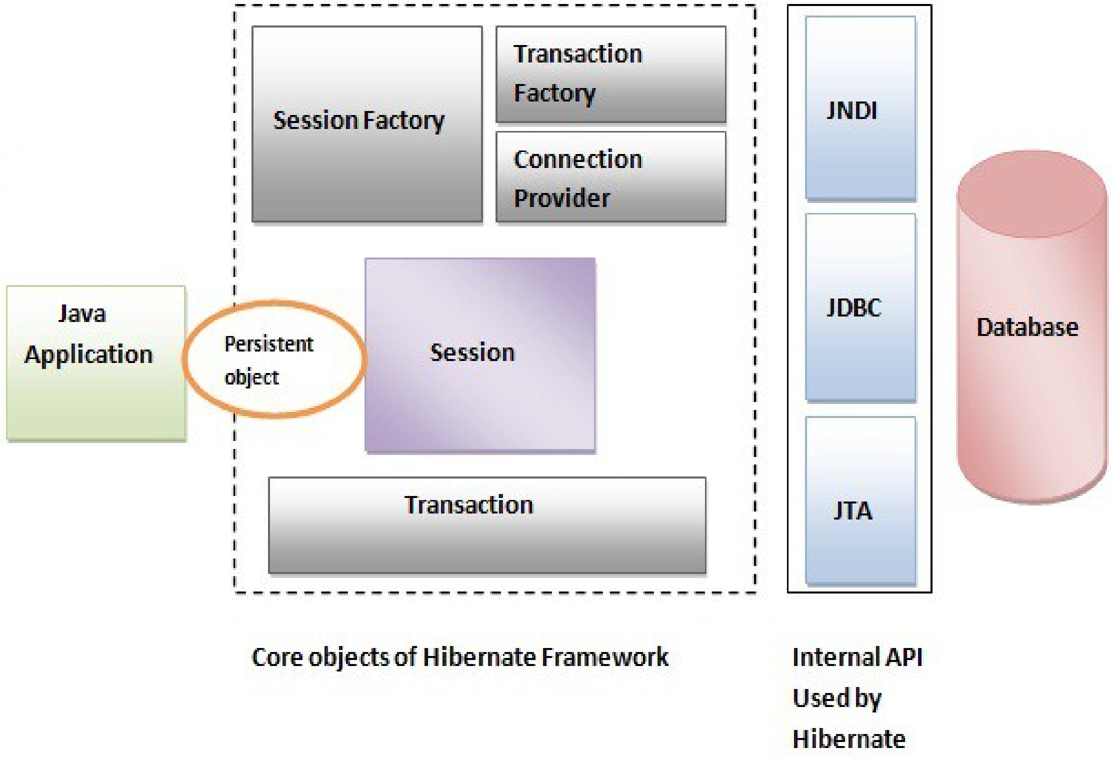
- The Hibernate architecture includes many objects persistent object, session factory, transaction factory, connection factory, session, transaction etc.
- There are 4 layers in hibernate architecture java application layer, hibernate framework layer, backhand api layer and database layer.

Let's see the diagram of hibernate architecture:



- Hibernate framework uses many objects session factory, session, transaction etc. alongwith existing Java API such as JDBC (Java Database Connectivity), JTA (Java Transaction API) and JNDI (Java Naming Directory Interface).

- This is the high level architecture of Hibernate with mapping file and configuration file.



Elements of Hibernate Architecture

- For creating the first hibernate application, we must know the elements of Hibernate architecture. They are as follows:

SessionFactory

- The SessionFactory is a factory of session and client of ConnectionProvider.
- It holds second level cache (optional) of data.
- The org.hibernate.SessionFactory interface provides factory method to get the object of Session.

Session

- The session object provides an interface between the application and data stored in the database.
- It is a short-lived object and wraps the JDBC connection.
- It is factory of Transaction, Query and Criteria.
- It holds a first-level cache (mandatory) of data.
- The org.hibernate.Session interface provides methods to insert, update and delete the object.
- It also provides factory methods for Transaction, Query and Criteria.

Transaction

- The transaction object specifies the atomic unit of work.
- It is optional.
- The org.hibernate.Transaction interface provides methods for transaction management.

ConnectionProvider

- It is a factory of JDBC connections.
- It abstracts the application from DriverManager or DataSource.

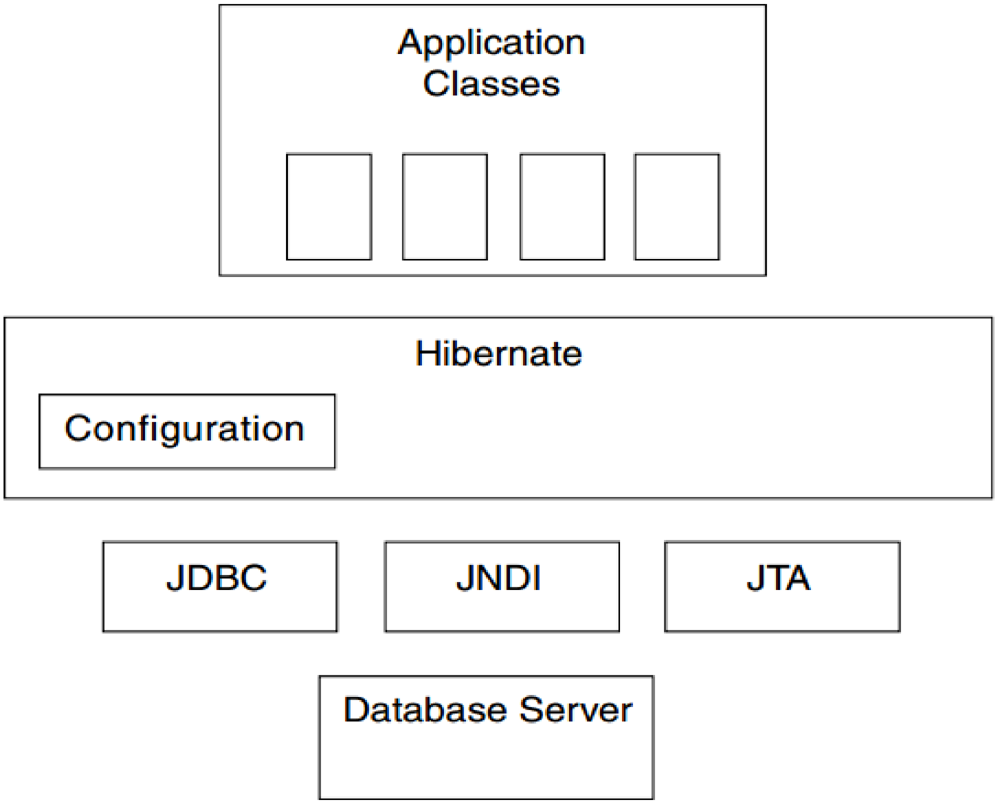
- It is optional.

TransactionFactory

- It is a factory of Transaction.
- It is optional.

Hibernate Configuration

- Figure shows the Configuration class, which is critical to launching Hibernate.
- The Configurationclass's operation has two keys components: the database connection and the class mapping setup



- The first component is handled through one or more configuration files supported by Hibernate. These files are hibernate.properties and hibernate.cfg.xml, as shown in Listing

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

    <!DOCTYPE hibernate-configuration PUBLIC

    "-//Hibernate/Hibernate Configuration DTD//EN"

    "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">

    <hibernate-configuration>

    <session-factory>

    <property name="connection.driver">com.mysql.jdbc.Driver</property>

    <property name="connection.url"

    jdbc:mysql://localhost/products</property>

    <property name="dialect">

    net.sf.hibernate.dialect.MySQLDialect</property>

    </session-factory>

</hibernate-configuration>

    hibernate.dialect net.sf.hibernate.dialect.MySQLDialect

    hibernate.connection.driver_class com.mysql.jdbc.Driver
    hibernate.connection.url jdbc:mysql://localhost/products

    hibernate.connection.username

    hibernate.connection.password

    hibernate.show_sql true

    hibernate.cglib.use_reflection_optimizer false
```

- One of the files handles setting up Hibernate using XML, and another doesn't. You can set up both JDBC and JNDI connections through these files as well as a host of other options.

- The second component makes the connection between the Java classes and database tables. Listing shows an example of a mapping for a simple Java class.

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

<!DOCTYPE hibernate-mapping

    PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"

    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

    <hibernate-mapping>

    <class name="Notes" table="notes">

    <id name="id" unsaved-value="0">

    <generator class="native"/>
```

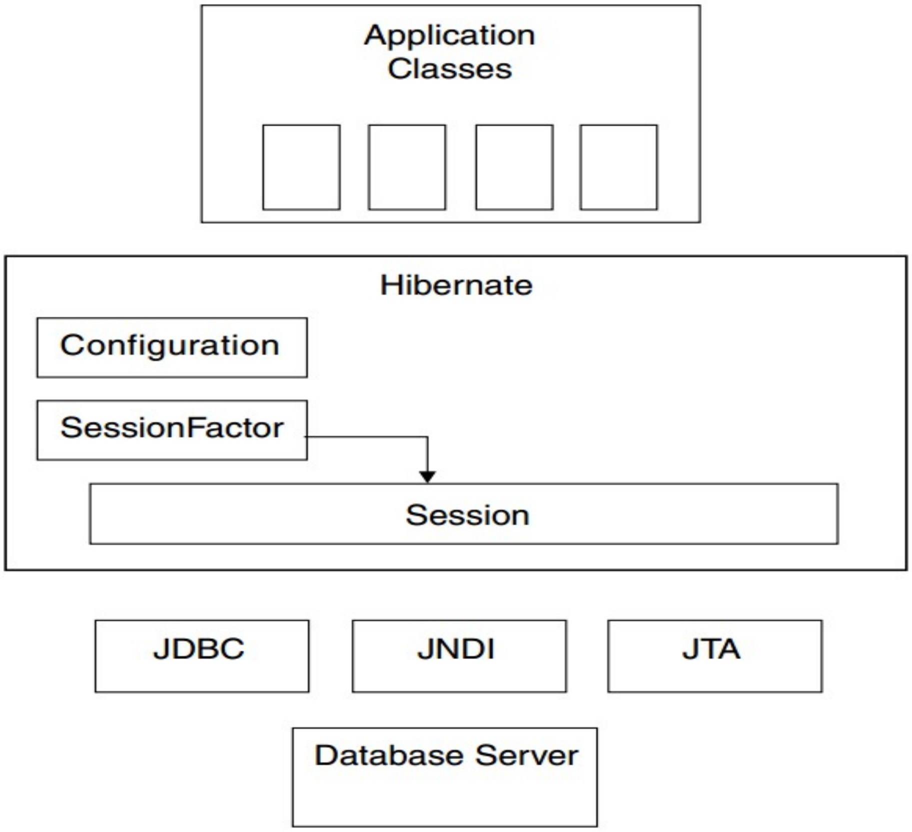
```
        </id>

        <property name="info" type="string"/>

        <property name="count" type="integer" not-null="true"/>

        <property name="zipcode" type="string"/>

        <property name="fullname" type="string"/>

        </class>

</hibernate-mapping>
```

- The mapping document is XML based and includes elements for specifying an identifier as well as the attributes of the mapped object.
- Once a mapping document has been created,the appropriate database table can be added to the database server, thus completing an O/R mapping from a Java class to the database.

Hibernate Sessions

- When you begin using Hibernate, a singleton class called SessionFactory is instantiated.
- As shown in Figure, the SessionFactory configures Hibernate for the application using the supplied configuration file and allows for a Session object to be instantiated.

- As you might expect, the SessionFactoryobject is heavyweight; you should take care to create only one SessionFactory per application.

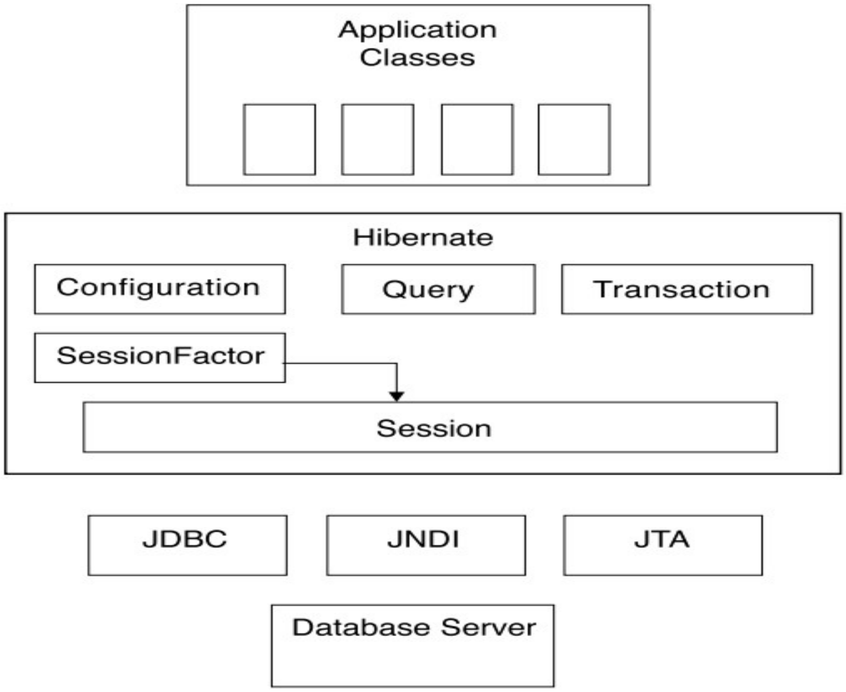


- When you begin persisting or loading objects to/from the database, a Sessionobject is instantiated.
- The Sessionobject is lightweight and designed to be instantiated each time an interaction is needed with the database. Thus, the Sessionobject is created, an object or two is persisted or loaded from the database, and the Sessionobject is closed.
- When a Sessionobject is instantiated, a connection is made to the database; thus the interactions should be fairly quick to ensure the application doesn't keep database connections open needlessly.

- In the process of working with persisted objects, two additional interfaces are commonly used, as shown in Figure. The Transaction interface lets you make sure that all the operations on persisted objects occur in a transaction supported by either JDBC or JTA.

- When you're using a transaction, any error that occurs between the start and end of the transaction will cause all the operations to fail; this is important when the persisted objects rely on each other to maintain integrity in the database.

- The Queryinterface lets you query the database for objects instead of having to rely solely n an object's identifier.



Hibernate Callbacks

- When Hibernate is in the process of persisting or loading an object, it provides support for you to be warned of certain events so you can respond appropriately.
- The process is implemented through callbacks. The available events are as follows:
  - public Boolean onSave(Session s); - called before the object is saved to the database
  - public Boolean onUpdate(Session s); - called before the object is updated
  - public boolean onDelete(Session s); - called before the object is deleted
  - public Boolean onLoad(Session s); - called after the object is loaded from the database

Hibernate Development Environment

Obtaining Hibernate

- To use Hibernate, you first need to obtain it from the Hibernate home page. Follow these steps:
- Browse to www.hibernate.org.

- Locate the Download link in the left navigational bar. Click it to bring up the download page.

- Part way down the new page is a Download from Sourceforge link. Click it to display a list of available downloads.

- The first two files in the list are the most recent ZIP and GZ distributions. Download the appropriate version for your platform.

Installing Hibernate

- One of the JARs in the Hibernate distribution is called hibernate2.jar; it's in the root directory of the installation.

- This is the primary JAR that Hibernate needs to do its work. Place this file in your classpath.

A Standalone Hibernate Application

Creating the Java Class

- The first step in creating an application is to build the Java class or classes, depending on the application that will be persisted to the database.

- As you might expect, a typical application has many objects; but only a few key objects hold information that should be kept permanently.

- The primary data in our application describes a CD, so we need an appropriate class. Listing shows an example CDclass.

```
import java.io.*;
import java.util.*;
public class CD {
int id;
String title;
String artist;
Date purchaseDate;
double cost;
public CD() {
}
public CD(String title, String artist, Date purchaseDate, double cost) {
this.title = title;
this.artist = artist;
this.purchaseDate = purchaseDate;
this.cost = cost;
}
```

```
public void setId(int id) {

this.id = id;

}

public int getId(){

return id;

}

public void setTitle(String title) {

this.title = title;

}

public String getTitle() {

return title;

}

public void setArtist(String artist) {

this.artist = artist;

}

public String getArtist() {

return artist;
}

public void setPurchasedate(Date purchaseDate) {

this.purchaseDate = purchaseDate;

}

public Date getPurchasedate() {

return purchaseDate;

}

public void setCost(double cost) {

this.cost = cost;

}

public double getCost() {

return cost;

}

}
```

- The CDclass is a JavaBeans-compliant class that holds the necessary information for CDs.

- When you’re writing classes to be persisted by Hibernate, it’s important to provide JavaBeans-compliant code as well as an index attribute like the id attribute in the CDclass.

- Hibernate uses the idattribute to determine uniqueness among the objects persisted to the database.

- Notice thatwe haven’t inherited or implemented any type of class or interface. Thus, Hibernate will use reflection based on the JavaBean setter/getter methods to persist the object.

Creating a Database Table

Once the object or objects to be persisted have been created, you create the database table. The hardest part of this process is matching the database column types to the defined class. The following command creates the necessary table for a Oracle database:

```
create table CD(

ID number not null primary key,

Title varchar2(15),

artist varchar2(15),

purchasedate date,

cost number(2,3)
```

);

- Use the appropriate interface for the database you’ve chosen to create the database tables in a database called products.

Building Configuration File

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

<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate
Configuration DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-
configuration-3.0.dtd">

<hibernate-configuration>

  <session-factory>

    <property
name="hibernate.dialect">org.hibernate.dialect.OracleDialect</property>

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

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

    <property name="hibernate.connection.username">smitha</property>

    <property name="hibernate.connection.password">java</property>

    <property name="hibernate.hbm2ddl.auto">update</property>
    <mapping package="hibernate1.hbm.xml" resource="hibernate1.hbm.xml"/>

  </session-factory>

</hibernate-configuration>
```

Building a Mapping Document for Database Tables

- At this point, we’ve defined a class to hold information about a CD as well as a database table where the CD information will be permanently kept.
- The next step is to build a mapping document that tells our CDclass.

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

<!DOCTYPE hibernate-mapping PUBLIC

"-//Hibernate/Hibernate Mapping DTD//EN"

http://www.sourceforae.net/hibernate-mapping-2.0.dtd>

<hibernate-mapping>

<class name="CD"  table="cd">

<id name="id" >

<column name="ID"  />
```

```
<generator class="increment"/>

</id>

<property name="title"/>

<property name="artist"/>

<property name="purchasedate" />

<property name="cost"/>

</class>

</hibernate-mapping>
```

- The mapping document is an XML document, and it includes the necessary tags to indicate its status.
- The root element is <hibernate-mapping>; it contains all the <class>elements used to define specific mappings from a Java class to the database.
- The <hibernate-mapping>element in our example doesn’t include any attributes; but if the class to be persisted was part of a package, we’d have an attribute called package.
- The full package declaration can be used to provide a package prefix for all the classes in this mapping file. If you don’t include a package attribute in the <hibernate-mapping> element, you must provide a fully qualified class name in the <class>element.

- The <class>element in our example includes two attributes:

  ❑ name: Identifies the fully qualified class name for the class or interface being mapped; if the package prefix is used in the <hibernate-mapping>element, the class name can be the unqualified name.

  ❑ table: Specifies the name of the table in the database where the mapped class will be persisted.

- The <class>element includes numerous elements to fully map the Java class. In our example, we use two subelements: <id>and <property>.

- The <id>element maps the unique IDattribute added to our CDclass to the primary key of the database. In our example, the <id>element includes both attributes and subelements. The attributes include the following:

  ❑ name:The attribute name found in the CD.

  ❑ type: The class type given to the IDattribute.

- The <id>element also includes a couple of subelements: <column>and <generator>. The <column> element tells Hibernate what specific column in the mapped database table relates to the identifier for the class. In our case, we use a <column>element with the name, sql-type, and not-null attributes set to the values used in the definition of the cddatabase table

- When Hibernate places an object in the database, it must be sure that each row is unique, because the rows represent specific objects. It's possible for two objects in the system to have the same attribute values. If this occurs, the system needs to be able to distinguish the objects in the database. To do this, we specified the identifier with our Java class and added the IDcolumn to the database table.

- Hibernate uses the identifier to assign a unique value, which is created using a generator. The <generator>element specifies the generator to be used when creating the unique identifier. The generator class is specified in the name attribute of the <generator>element. Hibernate includes quite a few generators. For our example, we've chosen the increment generator; it generates value for a given database table.
- After the <id>element has been specified, we need to let Hibernate know about the attributes to be persisted from our CDclass. The specification comes from the <property>element.
- You can build objects in an application and not persist all the attributes. When Hibernate pulls an object from the database, those attributes that aren't part of the persisted attributes are given their Java default value. In our example, we've persisted four attributes using the following elements:

```
<property name="title"/>

<property name="artist"/>

<property name="purchasedate" />
```

Application Skeleton Code

```
import java.util.Scanner;
```

```
    import org.hibernate.*;
    import org.hibernate.cfg.*;
    class  CDTest
    {
    public static void main(String args[]){
SessionFactory sessionFactory=null;
try {
Configuration cfg = new Configuration();
cfg.configure("hibernate.cfg.xml");
sessionFactory = cfg.buildSessionFactory();
} catch (Exception e) {
e.printStackTrace();
}
try {

Session session = sessionFactory.openSession();
Transaction t=session.beginTransaction();
Scanner sc=new Scanner(System.in);

System.out.println("enter artist");

String artist=sc.next();

System.out.println("enter title");

String title=sc.next();

System.out.println("enter cost");

double cost=sc.nextDouble();

CD cd = new CD(artist, title,new Date(), cost);

session.save(cd);

t.commit();

session.close();

System.out.println(“Saved successfully”);

}

catch (Exception e) {

e.printStackTrace();

}
```

}

}

Explanation:

With our example application ready and the mapping established between the CDclass and the database, we can begin to incorporate Hibernate into our application.

Build Configuration Object

The first step is to build a Configuration object. The Configuration object is responsible for pulling, parsing, and compiling the mapping documents into a format that Hibernate uses internally. Here's the code:

Configuration cfg = new Configuration().configure("hibernate.cfg.xml");

Obtaining the SessionFactory

The Configurationobject handles all the parsing and internal processing of mapping documents. Most of the real work done by Hibernate occurs in a Sessionobject. To build a Sessionobject, a SessionFactoryobject needs to be instantiated from the Configurationobject. We further expand the code as:

Configuration cfg = new Configuration().configure("hibernate.cfg.xml");

sessionFactory = cfg.buildSessionFactory();

The buildSessionFactory()method builds a factory based on the specifics of the mapping documents processed by the Configurationobject. Once the SessionFactoryobject has been created,the Configurationobject can be discarded.

Creating Persistent Classes

Now that we have a SessionFactory, we need to do two things: create a new CDobject and create a session for when we want to save the object to storage. Creating the CDobject is easy. If you look back at our skeleton code, you'll see that we create a new CDobject when the Add button is clicked. As far as Hibernate is concerned, this is all we need to do to create a new object. However, once the object has been created, we should save it to permanent storage. We do this by obtaining a Sessionobject and using the save()method:

```
Scanner sc=new Scanner(System.in);

System.out.println("enter artist");

String artist=sc.next();

System.out.println("enter title");

String title=sc.next();

System.out.println("enter cost");

double cost=sc.nextDouble();

CD cd = new CD(artist, title,new Date(), cost);
```

session.save(cd);

The first step in persisting our CD object is to create a Sessionobject. The session object handles all the work involved in saving, updating, and loading objects from permanent storage. A Session should be invoked for a short time, to save or load an object. Generally, a Sessionobject isn't created when the application is started and closed when the application exits; instead, it's created when needed and closed when a persistence operation has completed. With this in mind, we create a Sessionobject using the SessionFactory that was created when the application's constructor was called. One of the most important actions performed by the Session object is creating a connection to the database. The Sessionobject includes a few methods to use when

you're persisting an object to the database, including save(), saveOrUpdate(), and update(). The methods are differentiated based on the following criteria:

❑ save(): An object hasn't been persisted before.

❑ update(): An object has been persisted before.

❑ saveOrUpdate(): The object might have been persisted before, but we don't know.

Since we just created a new CD object, clearly we want to use the save() method. To persist the new object, we call the save() method of the Sessionobject, passing in the object that needs to be persisted.

To Update CD:

cd.setId(1);

```
session.update(cd);
```

To Delete CD:

```
cd.setId(1);

session.delete(cd);
```

To select CD:

```
CD cd1=new CD();

session.load(cd1,1);

System.out.println(cd1.getArtist()+cd1.getTitle()+cd1.getCost()+cd1.getId());
```

Applying hibernate to web application

- To build our Web application, the first step is to copy the CDclass created earlier into the /classes directory.
- Since we’ll be accessing the same CDobjects in our Web application, we need the CDclass to save the objects.

- Next, we need to write the servlet.But you need to know a few key things before writing the servlet code.There should only be a single SessionFactory for your application. This means you won’t be able to put the SessionFactory instantiation code in your servlet and instantiate it every time the servlet executes.

- When a Sessionobject is created from the SessionFactory, it’s designed to handle the work at hand and then be closed. Thus, it isn’t thread-safe.

- In fact, when you’re working with servlets, you should have a specific session for each threaded instance of the servlet.

- You can use the init() method in a servlet. For example, consider Listing , which contains the traditional skeleton code for a servlet.

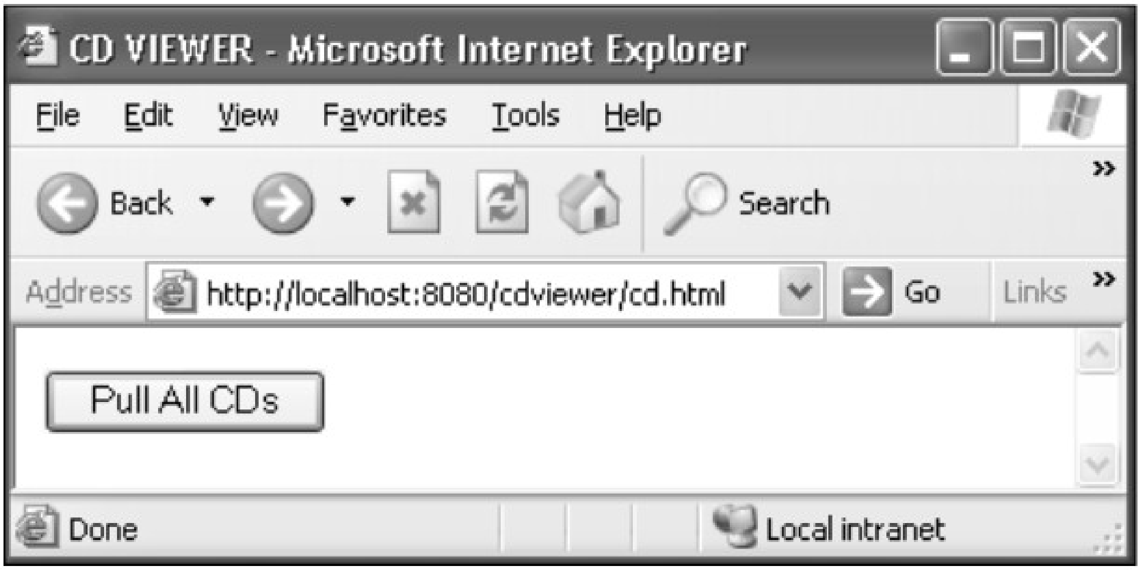
```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class CD Viewer extends HttpServlet {
private SessionFactory sessionFactory;
public void init(ServletConfig config) throws ServletException {
sessionFactory = new
Configuration().Configure(“hibernate.cfg.xml”).buildSessionFactoryO;
}
public void doGet(HttpServletRequest request,HttpServletResponse response)

throws IOException, ServletException {

response.setContentType("text/html");

PrintWriter out = response. getWriter();

}

public void doGet(HttpServletRequest request,HttpServletREsponse response)

throws IOException, ServletException {

doGet(request, response);}

}
```

- We've added an init()method to the servlet code to handle the creation of the SessionFactory object. The application server calls the init()method when the servlet is first prepared to handle incoming requests.

- For the execution lifecycle of the servlet, we're assured the init()method will be called once. We'll instantiate specific Session objects in the servlet code, since a session isn't threadsafe; thus we should obtain the session, use it, and close it like any other database connection.

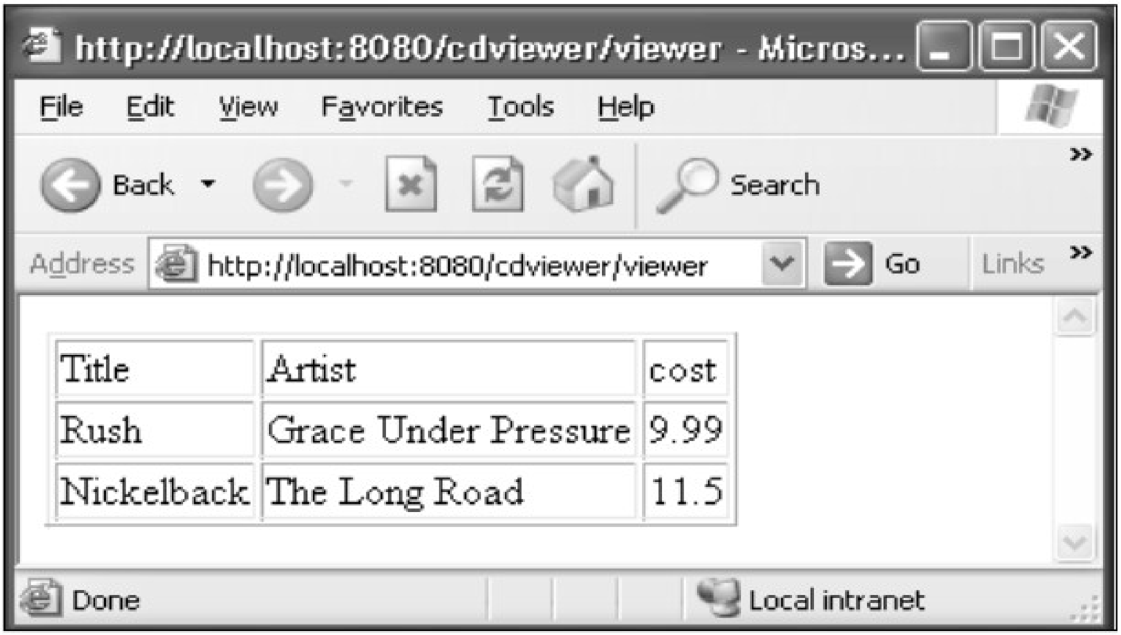
- Using this methodology, we can be sure the session is closed before the response is returned to the user Interface Page

To let outside employees access the CD objects in permanent storage, we need to have an appropriate Web page. Figure shows an example of the page presented to users.



http://localhost:8080/cdviewer/cd.html

an example of the output when the user clicks Pull All CDs.


http://localhost:8080/cdviewer/viewer - Micros...
File
Edit
View
Favorites
Tools
Help
Back
Search
Address
http://localhost:8080/cdviewer/viewer
Go
Links
Done
Local intranet

Title	Artist	cost
Rush	Grace Under Pressure	9.99
Nickelback	The Long Road	11.5

The initial Web page is created using HTML; it's shown in Listing

```
<HTML>

<HEAD>

<TITLE>CD VIEWER</TITLE>

<BODY>

<form action="/cdviewer/viewer" method="post">

<input type="submit" name="submit" value="Pull All CDs">

</form>

</BODY>
```

```
</BODY>

</HTML>
```

The most important aspect of the HTML code is found in the <form>tag, where the action specifies the /cdviewer/viewer servlet. When the use clicks the Submit button, the servlet we'll build next is executed.

Writing a Servlet to Use Hibernate

To produce the output shown in Figures, we need to begin writing the code in our servlet.

```
import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;

import java.util.*;

import org.hibernate.*;

import org.hibernate.cfg.*;

public class CDViewer extends HttpServlet {

private SessionFactory sessionFactory;

public void init(ServletConfig config) throws ServletException {

super.init(config);
```

```
try {

sessionFactory = new
Configuration().Configure(“hibernate.cfg.xml”).buildSessionFactoryO;

} catch(Exception e) {

e.printStackTrace();

}

}

private void displayAll(PrintWriter out, Session session) {

try {

out.println("<html>");

out.println("<table border='1'>");

out.println("<tr><td>Title</td><td>Artist</td><td>cost</td></tr>");

Transaction tx=session.beginTransaction();

List cds = session.find("from CD");

tx.commit();

Iterator iter = cds.iterator();

while (iter.hasNext()) {

CD cd = (CD)iter.next();
out.println("<tr><td>");

out.println(cd.getTitle());

out.println("</td><td>");

out.println(cd.getArtist());

out.println("</td><td>");

out.println(cd.getCost());

out.println("</td></tr>");

}

} catch(Exception e) {}

out.println("</table>");

out.println("</html>");

}

public void doGet(HttpServletRequest request,

HttpServletResponse response)

throws IOException, ServletException {

response.setContentType("text/html");

PrintWriter out = response. getWriter();
```

```
Session session = null;

try {

session = sessionFactory.openSession();

String action = request.getParameter("submit");

if (action.equals("Pull All CDs")) {

displayAll(out, session);

} else {

out.println("Bad Input");

}

session.flush();

session.close();

} catch (Exception e) {

e.printStackTrace();

}

}

public void doPost(HttpServletRequest request,

HttpServletResponse response) throws IOException, ServletException {
```

```
doGet(request, response);

}

}
```

When the servlet is first executed, the application server executes the init()method that instantiates the SessionFactoryobject. Since the init()method is executed only once, we get a single SessionFactoryobject. Next, the code in the doGet()method executes. This code begins by setting up for HTML output. Next,the tags for an HTML table are output along with the values for the table heading.Finally, we begin to access the data needed for the rows of the table. All the CDobjects are obtained through the find() method of the Sessionobject that was instantiated before we began to output the HTML. We haven't done anything special to access the CDobjects; we've obtained a Sessionobject, pulled the objects, and displayed their attributes for the user. Note that we close the Sessionobject before exiting the servlet. It's important to close the Sessionobject before the servlet is finished; otherwise a database connection could be left in a precarious position.

Hibernate Query Language

- In most of our previous examples, we obtained an object from the database using the object's identifier.
- Although this approach works well in our examples, it won't be so easy in an actual application. In a typical application, you'll use a username or account number to

access an object. With this information, you'll find the necessary objects that have been previously persisted to the database.

- To find the objects, you must use the Hibernate Query Language (HQL). HQL is a SQLlike language specifically designed to pull objects as well as attributes from objects.

Advantage of HQL

There are many advantages of HQL. They are as follows:

- database independent
- supports polymorphic queries
- easy to learn for Java Programmer

Query Interface

- It is an object oriented representation of Hibernate Query.
- The object of Query can be obtained by calling the createQuery() method Session interface.
- The query interface provides many methods. There is given commonly used methods:

1. public int executeUpdate() is used to execute the update or delete query.
2. public List list() returns the result of the ralation as a list.

3. public Query setFirstResult(int rowno) specifies the row number from where record will be retrieved.

4. public Query setMaxResult(int rowno) specifies the no. of records to be retrieved from the relation (table).

5. public Query setParameter(int position, Object value) it sets the value to the JDBC style query parameter.

6. public Query setParameter(String name, Object value) it sets the value to a named query parameter.

Example of HQL to get all the records

```
Query query=session.createQuery("from Emp");//here persistent class name is Emp

List list=query.list();
```

Example of HQL to get records with pagination

```
Query query=session.createQuery("from Emp");

query.setFirstResult(5);

query.setMaxResult(10);

List list=query.list();//will return the records from 5 to 10th number
```

Example of HQL update query

```
Transaction tx=session.beginTransaction();

Query q=session.createQuery("update User set name=:n where id=:i");

q.setParameter("n","Udit Kumar");

q.setParameter("i",111);

  int status=q.executeUpdate();

System.out.println(status);

tx.commit();
```

Example of HQL delete query

```
Query query=session.createQuery("delete from Emp where id=100");

//specifying class name (Emp) not tablename

query.executeUpdate();
```

HQL with Aggregate functions

You may call avg(), min(), max() etc. aggregate functions by HQL. Let's see some common examples:

Example to get total salary of all the employees

```
Query q=session.createQuery("select sum(salary) from Emp");

List<Emp> list=q.list();

    Iterator<Emp> itr=list.iterator();
    while(itr.hasNext()){

        System.out.println(itr.next());

}
```

Example to get maximum salary of employee

```
Query q=session.createQuery("select max(salary) from Emp");
```

Example to get minimum salary of employee

```
Query q=session.createQuery("select min(salary) from Emp");
```

Example to count total number of employee ID

```
Query q=session.createQuery("select count(id) from Emp");
```

Example to get average salary of each employees

```
Query q=session.createQuery("select avg(salary) from Emp");
```

Program demonstrating HQL

Main.java

```
package insertproductfromitem;

import java.util.Iterator;

import java.util.List;
import org.hibernate.*;

import org.hibernate.cfg.*;

import p1.Items;

public class Main {



  public static void main(String[] args) {

     Configuration cfg = new Configuration();

     cfg.configure("hibernate.cfg.xml");

      Transaction tx=null;

      Session session=null;

      SessionFactory factory=null;

          try

          {

         factory = cfg.buildSessionFactory();

          session = factory.openSession();

         tx=session.beginTransaction();

          Items ite=new Items();
```

```
        ite.setItemId(10);

        ite.setItemName("stationery");

        ite.setItemPrice(90.90);

        session.save(ite);

        tx.commit();

        Query qry = session.createQuery("insert into
    Product(productId,proName,price)select i.itemId,i.itemName,i.itemPrice from
    Items i where i.itemId= ?");

         qry.setParameter(0, 10);

         int res = qry.executeUpdate();



    System.out.println("Command successfully executed....");

    System.out.println("Numer of records effected...," + res);

    Main.selectProduct(session);

    qry = session.createQuery("update Product p set p.proName=?where
p.productId=1");

    qry.setParameter(0, "updated..");

    res = qry.executeUpdate();
```

```
        System.out.println("Command successfully executed....");
        System.out.println("Numer of records effected due to update query" + res);
        Main.selectProduct(session);
        qry = session.createQuery("delete from Product p where p.productId=:i");
        qry.setParameter("i", 2);
        res = qry.executeUpdate();
        System.out.println("Command successfully executed....");
        System.out.println("Numer of records effected due to delete query" + res);
         selectProduct(session);
}
catch(Exception e)
{
    tx.rollback();
    System.out.println("exception..."+e);
}
     session.close();
        factory.close();
    }

    public static void selectProduct(Session session)

    {

        Query qry = session.createQuery("select new
list(p.productId,p.proName,p.price) from Product p");

        List l =qry.list();

        System.out.println("Total Number Of Records : "+l.size());

        Iterator it = l.iterator();

        while(it.hasNext())

        {

           List i= (List)it.next();

            System.out.println("Product id : "+i.get(0));

            System.out.println("Product Name : "+i.get(1));

            System.out.println("Price"+i.get(2));

            System.out.println("---------------------------");

         }

    }

        }
```

Items.java

```
package p1;

public class Items {

   private int itemId;

   private String itemName;

   private double itemPrice;



   public int getItemId() {

      return itemId;

   }

   public void setItemId(int itemtId) {

      this.itemId = itemtId;

   }

   public String getItemName() {

      return itemName;

   }

   public void setItemName(String itemName) {
        this.itemName = itemName;
    }
    public double getItemPrice() {
        return itemPrice;
    }
    public void setItemPrice(double itemPrice) {
        this.itemPrice = itemPrice;
    }
}
```

Products.java

```
package p1;
public class Product{
    private int productId;
    private String proName;
    private double price;
    public void setProductId(int productId)
    {
```

```
   this.productId = productId;

}

public int getProductId()

{

   return productId;

}

public void setProName(String proName)

{

   this.proName = proName;

}

public String getProName()

{

   return proName;

}

public void setPrice(double price)

{

   this.price = price;
```

```
    }

    public double getPrice()

    {

        return price;

    }

}
```

Items.hbm.xml

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

<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

<class name="p1.Items" table="items">

<id name="itemId" column="ITEMID" type="int">

    <generator class="assigned"/>

    </id>

<property name="itemName"  column="ITEMNAME"  type="string"/>

<property name="itemPrice" column="ITEMPRICE" type="double"/>
```

```
</class>

</hibernate-mapping>
```

Products.hbm.xml

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

<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

<class name="p1.Product" table="products">

<id name="productId" column="pid" type="int" >

    <generator class="assigned"/>

</id>

<property name="proName" column="pname" type="string"/>

<property name="price" column="price" type="double"/>

</class>

</hibernate-mapping>
```

Struts

Understanding the MVC Design Pattern

- To gain a solid understanding of the Struts Framework, you must have a fundamental understanding of the MVC design pattern, which it is based on.
- The MVC design pattern, which originated from Smalltalk, consists of three components: a Model, a View, and a Controller. Table defines each of these components.

The Three Components of the MVC

Component	Description
Model	Represents the data objects. The Model is what is being manipulated and presented to the user.
View	Serves as the screen representation of the Model. It is the object that presents the current state of the data objects.
Controller	Defines the way the user interface reacts to the user's input. The Controller component is the object that manipulates the Model, or data object.

Some of the major benefits of using the MVC are:

- Reliability:The presentation and transaction layers have clear separation, which allows you to change the look and feel of an application without recompiling Model or Controller code.

- High reuse and adaptability: The MVC lets you use multiple types of views, all accessing the same server-side code. This includes anything from Web browsers (HTTP) to wireless browsers (WAP).

- Very low development and lifecycle costs: The MVC makes it possible to have lower-level programmers develop and maintain the user interfaces.

- Rapid deployment: Development time can be significantly reduced, because Controller programmers (Java developers) focus solely on transactions, and View programmers (HTML and JSP developers) focus solely on presentation.

- Maintainability: The separation of presentation and business logic also makes it easier to maintain and modify a Struts-based Web application.

Struts2 is a MVC based framework.

Struts 2 Overview

- Struts2 is popular and mature web application framework based on the MVC design pattern.

- Struts2 is not just the next version of Struts 1, but it is a complete rewrite of the Struts architecture.

- The WebWork framework started off with Struts framework as the basis and its goal was to offer an enhanced and improved framework built on Struts to make web development easier for the developers.

- After some time, the Webwork framework and the Struts community joined hands to create the famous Struts2 framework.

Struts 2 framework features:

Here are some of the great features that may force you to consider Struts2:

- **POJO forms and POJO actions** - Struts2 has done away with the Action Forms that were an integral part of the Struts framework. With Struts2, you can use any POJO to receive the form input. Similarly, you can now see any POJO as an Action class.

- **Tag support** - Struts2 has improved the form tags and the new tags allow the developers to write less code.

- **AJAX support** - Struts2 has recognised the take over by Web2.0 technologies, and has integrated AJAX support into the product by creating AJAX tags, that function very similar to the standard Struts2 tags.

- **Easy Integration** - Integration with other frameworks like Spring, Tiles and SiteMesh is now easier with a variety of integration available with Struts2.

- **Template Support** - Support for generating views using templates.

- **Plugin Support** - The core Struts2 behaviour can be enhanced and augmented by the use of plugins. A number of plugins are available for Struts2.

- **Profiling -** Struts2 offers integrated profiling to debug and profile the application. In addition to this, Struts also offers integrated debugging with the help of built in debugging tools.

- **Easy to modify tags** - Tag markups in Struts2 can be tweaked using Freemarker templates. This does not require JSP or java knowledge. Basic HTML, XML and CSS knowledge is enough to modify the tags.

- **Promote less configuration** - Struts2 promotes less configuration with the help of using default values for various settings. You don't have to configure something unless it deviates from the default settings set by Struts2.

- **View Technologies**: - Struts2 has a great support for multiple view options (JSP, Freemarker, Velocity and XSLT)

  The above are just the top ten features of Struts 2 that makes it an entreprise ready framework.

Struts 2 disadvantages:

Though Struts 2 comes with a list of great features but a few negative points about Struts 2 and would need lots of improvments:

- **Bigger learning curve** - To use MVC with Struts, you have to be comfortable with the standard JSP, Servlet APIs and a large & elaborate framework.

- **Poor documentation** - Compared to the standard servlet and JSP APIs, Struts has fewer online resources, and many first-time users find the online Apache documentation confusing and poorly organized.

- **Less transparent** - With Struts applications, there is a lot more going on behind the scenes than with normal Java-based Web applications which makes it difficult to understand the framework.

Final note, A good framework should provide generic behavior that many different types of applications can make use of it. Struts 2 is one of the best web framework and being highly used for the development of Rich Internet Applications (RIA).

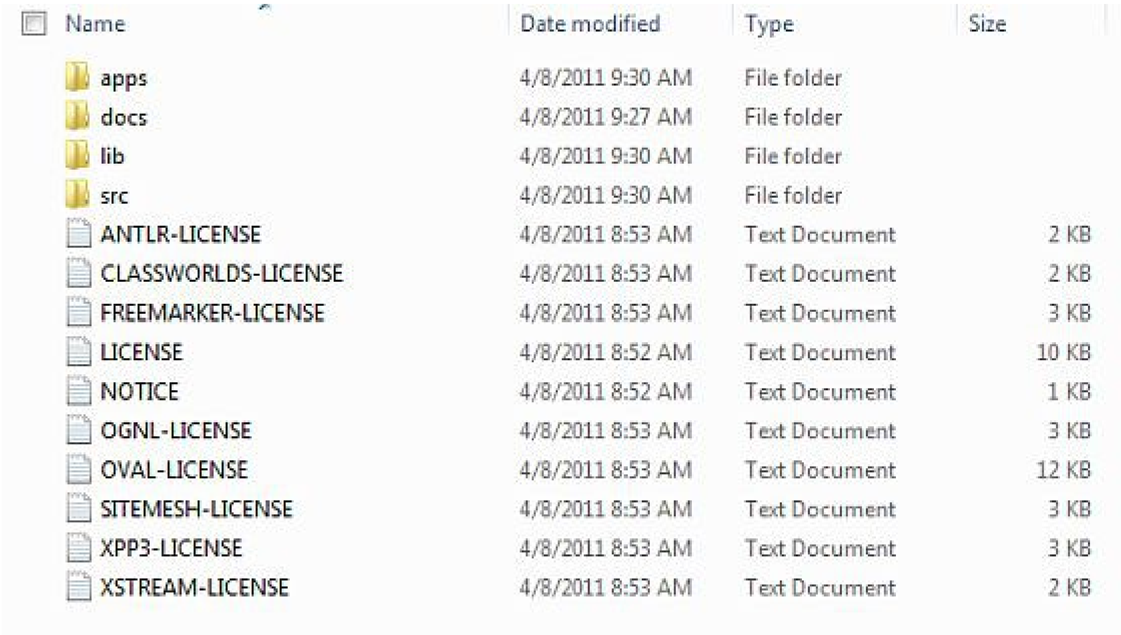
Setup Struts2 Libraries

Following are the simple steps to download and install Struts2 on your machine.

- Make a choice whether you want to install Struts2 on Windows, or Unix and then proceed to the next step to download .zip file for windows and .tz file for Unix.

- Download the latest version of Struts2 binaries from http://struts.apache.org/download.cgi.

- At the time of writing this tutorial, I downloaded struts-2.0.14-all.zip and when you unzip the downloaded file it will give you directory structure inside C:\struts-2.2.3 as follows.



Name	Date modified	Type	Size
apps	4/8/2011 9:30 AM	File folder	
docs	4/8/2011 9:27 AM	File folder	
lib	4/8/2011 9:30 AM	File folder	
src	4/8/2011 9:30 AM	File folder	
ANTLR-LICENSE	4/8/2011 8:53 AM	Text Document	2 KB
CLASSWORLDS-LICENSE	4/8/2011 8:53 AM	Text Document	2 KB
FREEMARKER-LICENSE	4/8/2011 8:53 AM	Text Document	3 KB
LICENSE	4/8/2011 8:52 AM	Text Document	10 KB
NOTICE	4/8/2011 8:52 AM	Text Document	1 KB
OGNL-LICENSE	4/8/2011 8:53 AM	Text Document	3 KB
OVAL-LICENSE	4/8/2011 8:53 AM	Text Document	12 KB
SITEMESH-LICENSE	4/8/2011 8:53 AM	Text Document	3 KB
XPP3-LICENSE	4/8/2011 8:53 AM	Text Document	3 KB
XSTREAM-LICENSE	4/8/2011 8:53 AM	Text Document	2 KB

Second step is to extract the zip file in any location, I downloaded & extracted struts-2.2.3-all.zip in c:\folder on my Windows 7 machine so that I have all the jar files into C:\struts-2.2.3\lib. Make sure you set your CLASSPATH variable properly otherwise you will face problem while running your application.

Struts 2 Architecture and Flow

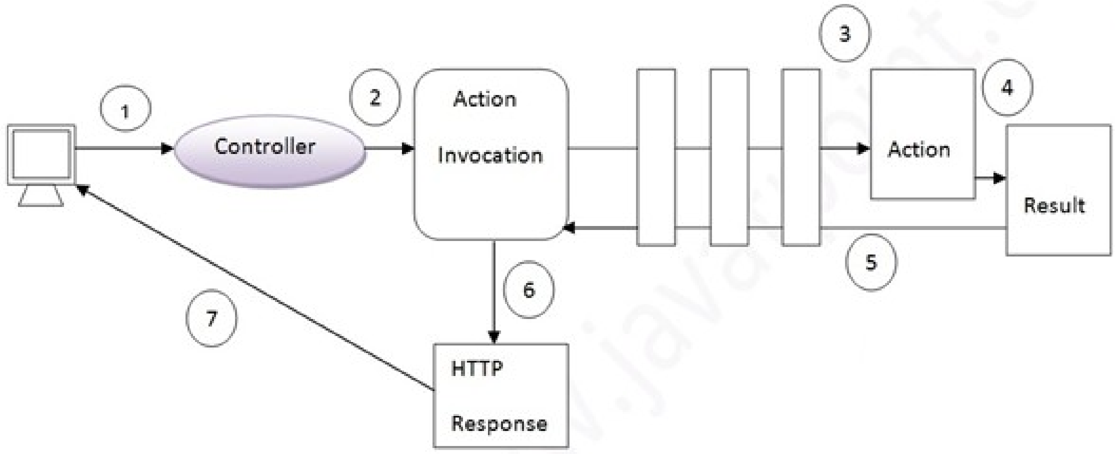
The architecture and flow of struts 2 application, is combined with many components such as Controller, ActionProxy, ActionMapper, Configuration Manager, ActionInvocation, Inerceptor, Action, Result etc.Here, we are going to understand the struts flow by 2 ways:

1. struts 2 basic flow

2. struts 2 standard architecture and flow provided by apache struts

Struts 2 basic flow

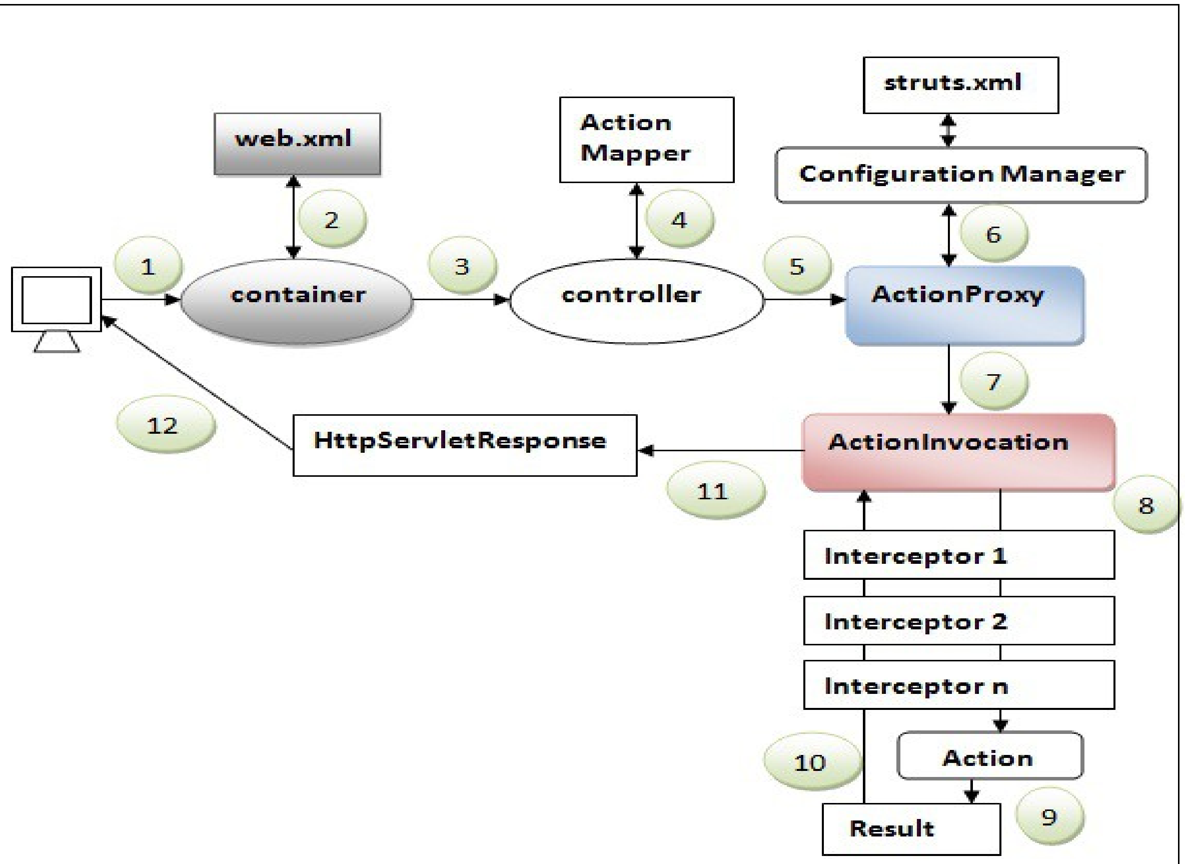
Let's try to understand the basic flow of struts 2 application by this simple figure



1. User sends a request for the action
2. Controller invokes the ActionInvocation
3. ActionInvocation invokes each interceptors and action
4. A result is generated
5. The result is sent back to the ActionInvocation
6. A HttpServletResponse is generated
7. Response is sent to the user

Struts 2 standard flow (Struts 2 architecture)

Let's try to understand the standard architecture of struts 2 application by this simple figure:



1. User sends a request for the action

2. Container maps the request in the web.xml file and gets the class name of controller.

3. Container invokes the controller (StrutsPrepareAndExecuteFilter or FilterDispatcher). Since struts2.1, it is StrutsPrepareAndExecuteFilter. Before 2.1 it was FilterDispatcher.

4. Controller gets the information for the action from the ActionMapper

5. Controller invokes the ActionProxy

6. ActionProxy gets the information of action and interceptor stack from the configuration manager which gets the information from the struts.xml file.

7. ActionProxy forwards the request to the ActionInvocation

8. ActionInvocation invokes each interceptors and action

9. A result is generated

10. The result is sent back to the ActionInvocation

11. A HttpServletResponse is generated

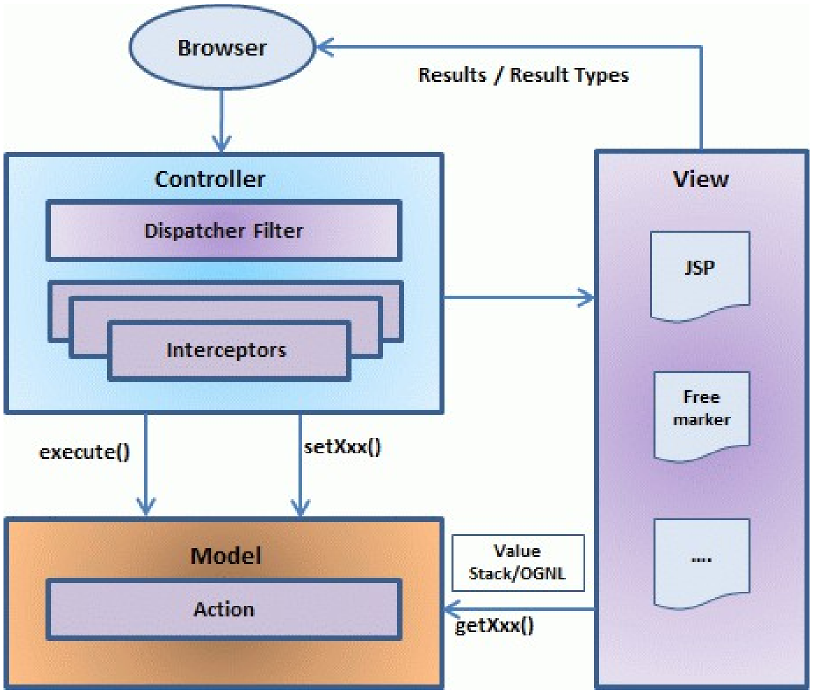
12. Response is sent to the user

Struts 2 Core Components

The Model-View-Controller pattern in Struts2 is realized with following five core components:

- Actions
- Interceptors
- Value Stack / OGNL
- Results / Result types
- View technologies

Struts 2 is slightly different from a traditional MVC framework in that the action takes the role of the model rather than the controller, although there is some overlap.



The above diagram depicts the Model, View and Controller to the Struts2 high level architecture.

- The controller is implemented with a Struts2 dispatch servlet filter as well as interceptors, the model is implemented with actions, and the view as a combination of result types and results. The value stack and OGNL provide common thread, linking and enabling integration between the other components.
- Apart from the above components, there will be a lot of information that relates to configuration. Configuration for the web application, as well as configuration for actions, interceptors, results, etc.

Request life cycle:

Based on the above digram, one can explain the user's request life cycle in Struts 2 as follows:

- User sends a request to the server for requesting for some resource (i.e pages).
- The FilterDispatcher looks at the request and then determines the appropriate Action.
- Configured interceptors functionalities applies such as validation, file upload etc.
- Selected action is executed to perform the requested operation.
- Again, configured interceptors are applied to do any post-processing if required.
- Finally the result is prepared by the view and returns the result to the user.

Struts 2 Action

- In struts 2, action class is POJO (Plain Old Java Object).POJO means you are not forced to implement any interface or extend any class.
- Generally, execute method should be specified that represents the business logic. The simple action class may look like:

Welcome.java

```
package  p1;

public class Welcome {

public String execute(){

   return "success";
} }
```

Action Interface

A convenient approach is to implement the com.opensymphony.xwork2.Action interface that defines 5 constants and one execute method.

5 Constants of Action Interface

Action interface provides 5 constants that can be returned form the action class. They are:

1.SUCCESS indicates that action execution is successful and a success result should be shown to the user.

2.ERROR indicates that action execution is failed and a error result should be shown to the user.

3.LOGIN indicates that user is not logged-in and a login result should be shown to the user.

4.INPUT indicates that validation is failed and a input result should be shown to the user again.

5.NONE indicates that action execution is successful but no result should be shown to the user.

Let's see what values are assigned to these constants:

```
public static final String SUCCESS = "success";

public static final String ERROR = "error";

public static final String LOGIN  = "login";

public static final String INPUT = "input";
public static final String NONE = "none";
```

Method of Action Interface

Action interface contains only one method execute that should be implemented overridden by the action class even if you are not forced.

```
public String execute();
```

Example of Struts Action that implements Action interface

If we implement the Action interface, we can directly use the constants instead of values.

Welcome.java

```
package p1;
import com.opensymphony.xwork2.Action;
public class Welcome implements Action{
public String execute(){
   return SUCCESS;
}
}
```

ActionSupport class

It is a convenient class that implements many interfaces such as Action, Validateable, ValidationAware, TextProvider, LocaleProvider and Serializable . So it is mostly used instead of Action.

Example of Struts Action that extends ActionSupport class

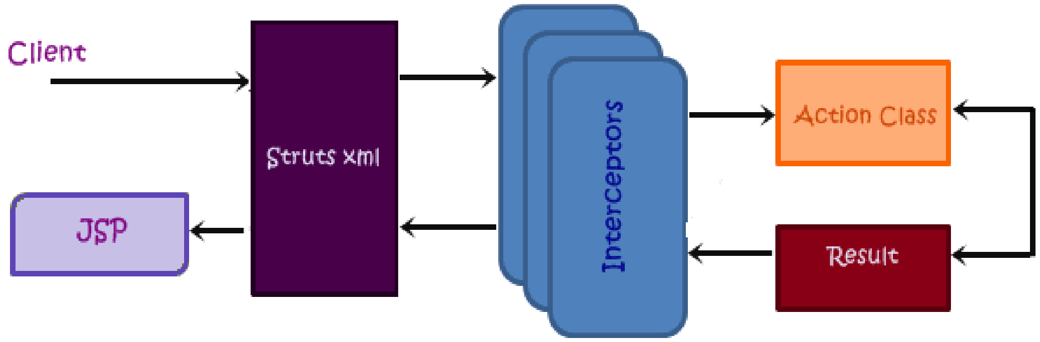
Let's see the example of Action class that extends the ActionSupport class.

Welcome.java

```
package p1;

import com.opensymphony.xwork2.ActionSupport;

public class Welcome extends ActionSupport{

public String execute(){

   return SUCCESS;  }}
```

Interceptors

- Interceptors are introduced as part of the Struts2 framework, power mechanism for controlling request and are responsible for the most of request processing.
- Interceptors are invoked by controller before and after invoking action and sits between the controller and action.
- An interceptor allows common, cross-cutting tasks to be defined into a clean, reusable component, which is different than the “action” code.



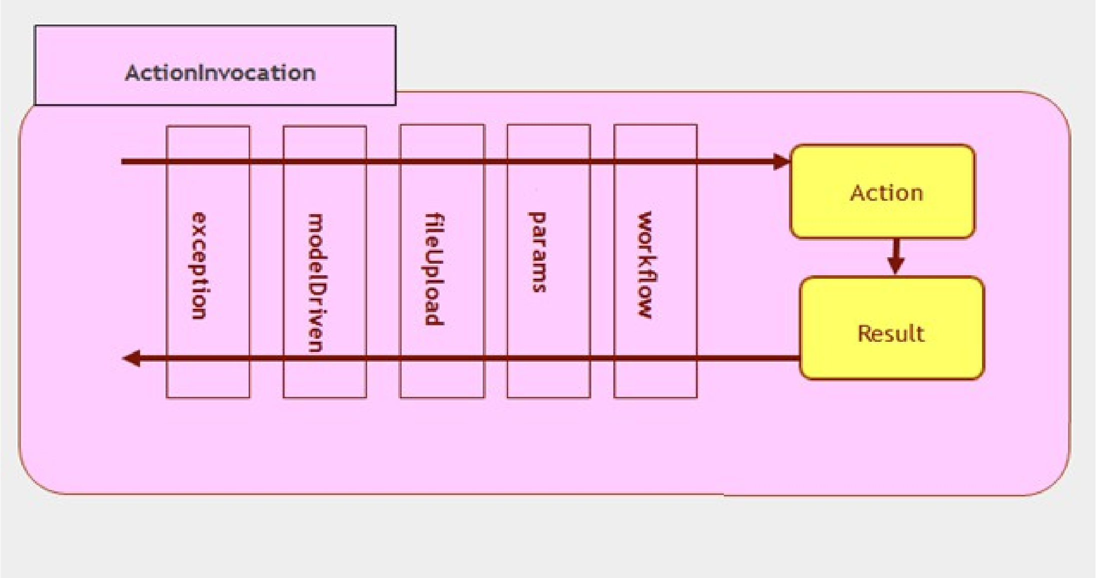
- Tasks like data validation, type conversion, and file upload are done using interceptors in Struts2.0.
- The built-in interceptors, declared in the "defaultStack" of the "struts-default package", handle most of the fundamental tasks.

Functioning of Interceptors

- From architectural point of view, interceptors improve separation of concerns in web applications.
- An example of a pre-processing task is the data transfer achieved with the "params" interceptor.
- Instead of having a simple controller that directly invokes an action, we now have a component that sits between the "controller" and the "action".
- In Struts 2, no action is invoked in isolation, the invocation of an action is a layered process. It always includes the execution of a stack of interceptors prior to and after the actual execution of the action itself.
- Rather than directly invoking the action's execute ( ) method, the framework creates an object called an "ActionInvocation". This object encapsulates the

action. Subsequently, all the interceptors that have been configured to fire before and after that “action” start their execution.

Interceptor Order



- The figure above represents the normal workflow; here none of the interceptors have diverted the invocation. This “action” will ultimately execute and return a “control string” that selects the appropriate “result”.
- After the “result” execution, each of the interceptors, in reverse order, gets a chance to do some post-processing work.
- As we will see, the interceptors have access to the “action” and other contextual values. This allows them to be aware of what is happening during the processing.
- One of the powerful functional aspects of “interceptors” is their “ability to alter the workflow of the invocation”.
- As we noted, the figure shown in the above depicts an instance where none of the interceptors has intervened in the workflow. It thus allows the “action” to execute, and determine the result that should render the view.

- Sometimes, one of the interceptors will determine that the “action should not execute”. In such cases, the interceptor can halt the workflow by itself by returning a “control string”.

ValueStack

- A valueStack is simply a stack that contains application specific objects such as action objects and other model object.
- At the execution time, action is placed on the top of the stack.
- We can put objects in the valuestack, query it and delete it.

ValueStack Interface

The struts 2 framework provides an interface to deal with valuestack. It provides many useful methods.

Methods of ValueStack interface

There are many methods in ValueStack interface. The commonly used methods are as follows:

- public String findString(String expr) finds the string by evaluating the given expression.
- public Object findValue(String expr) finds the value by evaluating the specified expression.
- public Object findValue(String expr, Class c) finds the value by evaluating the specified expression.

- public Object peek() It returns the object located on the top of the stack.
- public Object pop() It returns the object located on the top of the stack and removes it.
- public void push(Object o) It puts the object on the top of the stack.
- public void set(String key, Object value) It sets the object on the stack with the given key. It can be get by calling the findValue(key) method.
- public int size() It returns the number of objects from the stack.

Struts 2 ActionContext

- The ActionContext is a container of objects in which action is executed.
- The values stored in the ActionContext are unique per thread (i.e. ThreadLocal). So we don't need to make our action thread safe.
- We can get the reference of ActionContext by calling the getContext() method of ActionContext class. It is a static factory method. For example:

ActionContext context = ActionContext.getContext();

Struts 2 ActionInvocation

- The ActionInvocation represents the execution state of an action.
- It holds the action and interceptors objects.

ActionInvocation Interface

- The struts framework provides ActionInvocation interface to deal with ActionInvocation.
- It provides many methods, some of them can be used to get the instance of ValueStack, ActionProxy, ActionContext, Result etc.

Methods of ActionInvocation Interface

The commonly used methods of ActionInvocation interface are as follows:

No.	Method	Description
1)	public ActionContext getInvocationContext()	returns the ActionContext object associated with the ActionInvocation.
2)	public ActionProxy getProxy()	returns the ActionProxy instance holding this ActionInvocation.
3)	public ValueStack getStack()	returns the instance of ValueStack.
4)	public Action getAction()	returns the instance of Action associated with this ActionInvocation.
5)	public void invoke()	invokes the next resource in processing this ActionInvocation.
6)	public Result getResult()	returns the instance of Result.

Struts 2 OGNL

- The Object Graph Navigation Language (OGNL) is an expression language.

- It simplifies the accessibility of data stored in the ActionContext.
- The struts framework sets the ValueStack as the root object of OGNL. Notice that action object is pushed into the ValueStack.
- We can direct access the action property.

```
<s:property value="username"/>
```

Here, username is the property key.

- The struts framework places other objects in ActionContext also e.g. map representing the request, session,application scopes.
- To get these values i.e. not the action property, we need to use # notation. For example to get the data from session scope, we need to use #session as given in the following example:

```
<s:property name="#session.username"/>
```

(or)

```
<s:property name="#session['username']"/>
```

Results/Result Types

- An action method returns a String that determines what result to execute.
- An action declaration must contain result elements that each corresponds to a possible return value of the action method. If, for example, an action method

returns either Action.SUCCESS or Action.INPUT, the action declaration must have two result elements like these

```
<action ... >
<result name="success"> ... </result>
<result name="input"> ... </result>
</action>
```

A result element can have these attributes:

- **name**. The name of the result that matches the output of the action method. For example, if the value of the name attribute is "input," the result will be used if the action method returns "input." The name attribute is optional and its default value is "success."

- **type.** The result type. The default value is "dispatcher," a result type that forwards to a JSP.

The default values of both attributes help you write shorter configuration. For example, these result elements

```
<result name="success type="dispatcher">/Product.jsp</result>
<result name="input" type="dispatcher">/ProductForm.jsp</result>
```

are the same as these:

```
<result>/Product.jsp</result>
<result name="input">/ProductForm.jsp</result>
```

- The first result element does not have to contain the name an d type attributes as it uses the default values. The second result element needs the name attribute but does not need the type attribute.
- Dispatcher is the most frequently used result type, but it's not the only type available.

Table shows all standard result types. The words in brackets in the Result Type column are names used to register the result types in the configuration file. That's right, you must register a result type before you can use it.

In addition to the ones in Table, many third party developers deploy plug-ins that encapsulate new result types.Each of the result types is explained below.

Chain

- The Chain result type is there to support action chaining, whereby an action is forwarded to another action and the state of the original action is retained in the target action.(The Chaining interceptor makes action chaining possible and since this interceptor is part of defaultStack, you can use action chaining right away.)

The following declarations show an example of action chaining.

```
<package name="package1" extends="struts-default">
<action name="action1" class="...">
<result type="chain">action2</result>
```

```
</action>
<action name="action2" class="...">
<result type="chain">
<param name="actionName">action3</param>
<param name="namespace">/namespace2</param>
</result>
</action>
</package>
<package name="package2" namespace="/namespace2"
extends="struts-default">
<action name="action3" class="...">
<result>/MyView.jsp</result>
</action>
</package>
```

- action1 in package1 is chained to action2, which in turn is chained to action3 in a different package.

- Chaining to an action in a different package is allowed as long as you specify the namespace parameter of the target action.

- If action-x is chained to action-y, action-x will be pushed to the Value Stack, followed by action-y, making action-y the top object in the Object Stack. As a result, both actions can be accessed from the view.

- If action-x and action-y both have a property that shares the same name, you can access the property in action-y (the top object) using this OGNL expression:
  - [0].propertyName or propertyName
- You can access the property in action-x using this expression:
  - [1].propertyName
- Generally action chaining is not recommended as it may turn your actions into spaghetti code. If action1 needs to be forwarded to action2, for example, you need to ask yourself if there's code in action2 that needs to be pushed into a method in a utility class that can be called from both action1 and action2.

Dispatcher

- The Dispatcher result type is the most frequently used type and the default type.
- This result type has a location parameter that is the default parameter.
- Since it is the default parameter, you can either pass a value to it by using the param element like this:

```
<result name="...">
<param name="location">resource</param>
</result>
```

or by passing the value to the result element.

```
<result name="...">resource</result>
```

- Use this result type to forward to a resource, normally a JSP or an HTML file, in the same application.
- You cannot forward to an external resource and its location parameter cannot be assigned an absolute URL.
- **To direct to an external resource, use the Redirect result type.**

FreeMarker

- This result type forwards to a FreeMarker template.

HttpHeader

- This result type is used to send an HTTP status to the browser.

```
<default-action-ref name="CatchAll"/>
<action name="CatchAll">
<result type="httpheader">
<param name="status">404</param>
</result>
</action>
```

- The defaul t-action-ref element is used to specify the default action, which is the action that will be invoked if a URI does not have a matching action.

- In the example above, the CatchAll action is the default action. CatchAll uses a HttpHeader result to send a 404 status code to the browser. As a result, if there's no matching action, instead of getting Struts' error messages:

Struts Problem Report

Struts has detected an unhandled exception:

Messages: There is no Action mapped for namespace / and action name blahblah

the user will get a 404 status report and will see a default page from the container.

Redirect

- This result type redirects, instead of forward, to another resource. This result type accepts these parameters
  - **location.** Specifies the redirection target.
  - **parse**. Indicates whether or not the value of location should be parsed for OGNL expressions. The default value for parse is true.
- The main reason to use a redirect, as opposed to a forward, is to direct the user to an external resource.
- A forward using Dispatcher is preferable when directing to an internal resource because a forward is faster. Redirection would require a round trip since the client browser would be forced to re-send a new HTTP request.Having said that, there is a reason why you may want to redirect to an internal resource.

- You normally redirect if you don't want a page refresh invokes the previously invoked action. For instance, in a typical application, submitting a form invokes a Product _ save action, that adds a new product to the database. If this action forwards to a JSP, the Address box of the browser will still be showing the URL that invoked Product_save.
- If the user for some reason presses the browser's Reload or Refresh button, the same action will be invoked again, potentially adding the same product to the database. Redirection removes the association with the previous action as the redirection target has a new URL.

Here is an example of redirecting to an external resource.

```
<action name="..." class="...">
<result name="success" type="redirect">
http://www.example.com/test.html
</result>
</action>
```

And this to an internal resource:

```
<action name="..." class="...">
<result name="success" type="redirect">
/jsp/Product.jsp
</result>
</action>
```

When redirecting to an internal resource, you specify a URI for the resource. The URI can point to an action. For instance,

```
<action name="..." class="...">
<result name="success" type="redirect">
User_input.action
</result>
</action>
```

- Redirect does not care if the target is a JSP or an action, it always treat it as if the target is another page. The underlying class for the Redirect result type calls HttpServletResponse.sendRedirect.

- If you need the state of the source action available in the target destination, you can pass data through the session or request parameters. The RedirectTest action below redirects to the User_input action and passes the value of the userName property of the TestUser action class as a userName request parameter. Note that the dynamic value is enclosed in ${ and }.

```
<action name="RedirectTest" class="app03a.TestUser">
<result type="redirect">
User_input.action?userName=${userName}
</result>
</action>
```

Note also that you need to encode special characters such as & and + . For example, if the

target is http://www.test.com?user=l&site=4, you must change the & to &amp;.

```
<result name="login" type="redirect">
http://www.test.com?user=1&amp;site=4
</result>
```

Redirect Action

- This result type is similar to Redirect.
- Instead of redirecting to a different resource, however, Redirect Action redirects to another action.
- The Redirect Action result type can take these parameters:
    - **actionName.** Specifies the name of the target action. This is the default attribute.
    - **namespace.** The namespace of the target action. If no namespace parameter is present, it is assumed the target action resides in the same namespace as the enclosing action.

For example, the following Redirect Action result redirects to a User_input action .

```
<result type="redirect-action">
<param name="actionName">User_input</param>
</result>
```

And since actionName is the default parameter, you can simply write:

```
<result type="redirect-action">User_input</result>
```

- Note that the value of the redirection target is an action name. There is no . action suffix necessary as is the case with the Redirect result type.

- In addition to the two parameters, you can pass other parameters as request parameters.

For example, the following result type

```
<result type="redirect-action">
<param name="actionName">User_input</param>
<param name="userId">xyz</param>
<param name="area">ga</param>
</result>
```

will be translated into this URI:

User_input.action?userId=xyz&area=ga

Stream

This result type does not forward to a JSP. Instead, it sends an output stream to the browser.

Velocity

- This result type forwards to a Velocity template.

XSLT

- This result type uses XML/XSLT as the view technology.

PlainText

- A PlainText result is normally used for sending a JSP's source. For example, the action Source_show below displays the source of the Menu.jsp page.

```
<action name="Source_show" class="...">
<result name="success" type="plaintext">/jsp/Menu.jsp</result>
</action>
```

Struts 2 Configuration Files

- web.xml, struts.xml, struts-config.xml and struts.properties
- Honestly speaking you can survive using web.xml and struts.xml configuration files,but for your knowledge let me explain other files as well.

The web.xml file:

- The web.xml configuration file is a J2EE configuration file that determines how elements of the HTTP request are processed by the servlet container.
- It is not strictly a Struts2 configuration file, but it is a file that needs to be configured for Struts2 to work.

- This file provides an entry point for any web application. The entry point of Struts2 application will be a filter defined in deployment descriptor (web.xml). Hence we will define an entry of FilterDispatcher class in web.xml. The web.xml file needs to be created under the folder WebContent/WEB-INF.

- This is the first configuration file you will need to configure. Following is an example of web.xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://java.sun.com/xml/ns/javaee"
 xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
 http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
 id="WebApp_ID" version="3.0">
  <display-name>Struts 2</display-name>
 <welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
 </welcome-file-list>
  <filter>
  <filter-name>struts2</filter-name>
  <filter-class>
   org.apache.struts2.dispatcher.FilterDispatcher
  </filter-class>
 </filter>
 <filter-mapping>
```

```
      <filter-name>struts2</filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>
</web-app>
```

Elements of struts.xml file

struts.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts
Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">
<struts>
<package name="default" extends="struts-default">
  <action name="product" class="p1.Product">
<result name="success">welcome.jsp</result>
</action>
  </package>
</struts>
```

1) package element

- We can easily divide our struts application into sub modules. The package element specifies a module.
- You can have or or more packages in the struts.xml file.

Attributes of package element

- **name** name is must for defining any package.
- **Namespace** It is an optional attribute of package.
  - o If namespace is not present, / is assumed as the default namespace.In such case, to invoke the action class, you need this URI:

    /actionName.action

  - o If you specify any namespace, you need this URI:

    /namespacename/actionName.action

- **extends** The package element mostly extends the struts-default package where interceptors and result types are defined. If you extend struts-default, all the actions of this package can use the interceptors and result-types defined in the struts-default.xml file.

2) action element

- The action is the subelement of package and represents an action.

Attributes of action element

- **name** name is must for defining any action.
- **class** class is the optional attribute of action.

If you omit the class attribute, ActionSupport will be considered as the default action. A simple action may be as: <action name="product">

- **method** It is an optional attribute. If you don't specify method attribute, execute method will be considered as the method of action class. So this code:

<action name="product" class="p1.Product">

will be same as: <action name="product" class="p1.Product" method="execute">

If you want to invoke a particular method of the action, you need to use method attribute.

3) result element

- It is the sub element of action that specifies where to forward the request for this action.

Attributes of result element

- **name** is the optional attribute.

  If you omit the name attribute, success is assumed as the default result name.

- **type** is the optional attribute.

  If you omit the type attribute, dispatcher is assumed as the default result type.

Other elements

- There are many other elements also such as global-exception-mappings,global-results, include etc.

The struts-config.xml file:

The struts-config.xml configuration file is a link between the View and Model components in the Web Client but you would not have to touch these settings for 99.99% of your projects.

The configuration file basically contains following main elements:

Interceptor	& Description
struts-config	This is the root node of the configuration file.
form-beans	This is where you map your ActionForm subclass to a name. You use this name as an alias for your ActionForm throughout the rest of the struts-config.xml file, and even on your JSP pages.
global forwards	This section maps a page on your webapp to a name. You can use this name to refer to the actual page. This avoids hardcoding URLs on your web pages.
action-mappings	This is where you declare form handlers and they are also known as action mappings.
Controller	This section configures Struts internals and rarely used in practical situations.
plug-in	This section tells Struts where to find your properties files, which contain prompts and error messages

Following is the sample struts-config.xml file:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 1.0//EN"
"http://jakarta.apache.org/struts/dtds/struts-config_1_0.dtd">

<struts-config>

  <!-- ========== Form Bean Definitions ============ -->
  <form-beans>
    <form-bean name="login" type="test.struts.LoginForm" />
  </form-beans>

  <!-- ========== Global Forward Definitions ========= -->
  <global-forwards>
  </global-forwards>

  <!-- ========== Action Mapping Definitions ======== -->
  <action-mappings>
    <action
      path="/login"
      type="test.struts.LoginAction" >

      <forward name="valid" path="/jsp/MainMenu.jsp" />
      <forward name="invalid" path="/jsp/LoginView.jsp" />
    </action>
  </action-mappings>

  <!-- ========== Controller Definitions ======== -->
  <controller
    contentType="text/html;charset=UTF-8"
    debug="3"
    maxFileSize="1.618M"
    locale="true"
    nocache="true"/>

</struts-config>
```

For more detail on struts-config.xml file, kindly check your struts documentation.

The struts.properties file

- This configuration file provides a mechanism to change the default behavior of the framework.

- Actually all of the properties contained within the struts.properties configuration file can also be configured in the web.xml using the init-param, as well using the constant tag in the struts.xml configuration file.

- But if you like to keep the things separate and more struts specific then you can create this file under the folder WEB-INF/classes.

- The values configured in this file will override the default values configured in default.properties which is contained in the struts2-core-x.y.z.jar distribution.

- There are a couple of properties that you might consider changing using struts.properties file:

```
### When set to true, Struts will act much more friendly for developers
struts.devMode = true


### Enables reloading of internationalization files
struts.i18n.reload = true


### Enables reloading of XML configuration files
struts.configuration.xml.reload = true


### Sets the port that the server is run on
struts.url.http.port = 8080
```

Here any line starting with hash (#) will be assumed as a comment and it will be ignored by Struts 2.

Steps to create Struts 2 Application Example

- In this example, we are creating the struts 2 example without IDE.

- We can simply create the struts 2 application by following these simple steps:

1. Create the directory structure

2. Create input page (index.jsp)

3. Provide the entry of Controller in (web.xml) file

4. Create the action class (Product.java)

5. Map the request with the action in (struts.xml) file and define the view components

6. Create view components (welcome.jsp)

7. load the jar files

8. start server and deploy the project

1) Create the directory structure

- The directory structure of struts 2 is same as servlet/JSP.

- Here, struts.xml file must be located in the classes folder.

2) Create input page (index.jsp)

- This jsp page creates a form using struts UI tags.

- To use the struts UI tags, you need to specify uri /struts-tags.

- Here, we have used s:form to create a form, s:textfield to create a text field, s:submit to create a submit button.

index.jsp

```
<%@ taglib uri="/struts-tags" prefix="s" %>

<s:form action="product">
```

```
<s:textfield name="id" label="Product Id"></s:textfield>

<s:textfield name="name" label="Product Name"></s:textfield>

<s:textfield name="price" label="Product Price"></s:textfield>

<s:submit value="save"></s:submit>

</s:form>
```

3) Provide the entry of Controller in (web.xml) file

- In struts 2, StrutsPrepareAndExecuteFilter class works as the controller.
- As we know well, struts 2 uses filter for the controller.
- It is implicitly provided by the struts framework.

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
 <filter>
 <filter-name>struts2</filter-name>
  <filter-class>
   org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter
  </filter-class>
 </filter>
 <filter-mapping>
```

```
   <filter-name>struts2</filter-name>

    <url-pattern>/*</url-pattern>

  </filter-mapping>

</web-app>
```

4) Create the action class (Product.java)

This is simple bean class. In struts 2, action is POJO (Plain Old Java Object). It has one extra method execute i.e. invoked by struts framework by default.

Product.java

```
package p1;

  public class Product {

private int id;

private String name;

private float price;

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 float getPrice() {

    return price;

}

public void setPrice(float price) {

    this.price = price;

}

  public String execute(){

    return "success";

}

}
```

5) Map the request in (struts.xml) file and define the view components

struts.xml

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

<!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts

Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">

<struts>

<package name="default" extends="struts-default">
```

```
  <action name="product" class="p1.Product">
<result name="success">welcome.jsp</result>
</action>
  </package>
</struts>
```

6) Create view components (welcome.jsp)

- It is the view component the displays information of the action. Here, we are using struts tags to get the information.
- The s:property tag returns the value for the given name, stored in the action object.

welcome.jsp

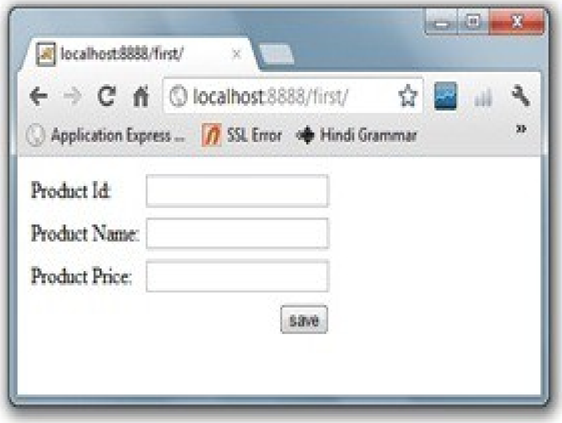
```
<%@ taglib uri="/struts-tags" prefix="s" %>
  Product Id:<s:property value="id"/><br/>
Product Name:<s:property value="name"/><br/>
Product Price:<s:property value="price"/><br/>
```

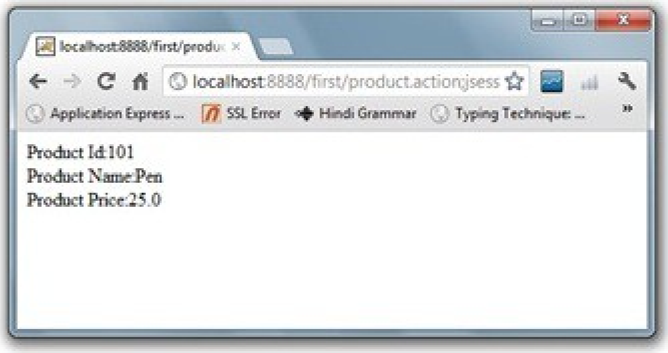
7) Load the jar files

To run this application, you need to have the struts 2 jar files. Here, we are providing all the necessary jar files for struts 2. Download it and put these jar files in the lib folder of your project.

8) start server and deploy the project

Finally, start the server and deploy the project and access it.




localhost:8888/first/produ
localhost:8888/first/product.action;jsess
Application Express ...
SSL Error
Hindi Grammar
Typing Technique: ...

```
Product Id:101
Product Name:Pen
Product Price:25.0
```

Struts 2 Registration Form Example

In this example, we are going to create a registration form using struts UI tags and store these information into the oracle database. You may use other database also such as mysql, DB2 etc. according to your requirement.

Let's see the table first that we need to create in the oracle database.

```
CREATE TABLE  "STRUTSUSER"
  (    "NAME" VARCHAR2(4000),
   "PASSWORD" VARCHAR2(4000),
   "EMAIL" VARCHAR2(4000),
   "GENDER" VARCHAR2(4000),
   "COUNTRY" VARCHAR2(4000)
  )
```

It will be better for you to create an id for each user. To simply the example, we have not alloted any id with primary key enabled. But you can do it.

The steps to create the registration application in struts2 are as follows:

1. Create input page (index.jsp)
2. Create the action class (RegisterAction.java)
3. Create the class to store data (RegisterDao.java)
4. Map the request in (struts.xml) file and define the view components
5. Create view components

1) Create input page (index.jsp)

- It is the simple jsp page that uses struts 2 UI tags to create a form to get input from the user.

index.jsp

```
<%@ taglib uri="/struts-tags" prefix="s" %>
  <s:form action="register">
```

```
<s:textfield name="name" label="UserName"></s:textfield>
<s:password name="password" label="Password"></s:password>
<s:textfield name="email" label="Email"></s:textfield>
<s:radio list="{'male','female'}" name="gender"></s:radio>
<s:select cssStyle="width:155px;" list="{'india','pakistan','other',}"
name="country" label="Country"></s:select>
  <s:submit value="register"></s:submit>
  </s:form>
```

2) Create the action class (RegisterAction.java)

- This Action class has five fields and one execute method.
- As we know, struts framework creates instance of the action class per request, we are passing this object in the save method of RegisterDao class.

RegisterAction.java

```
package p1;
  public class RegisterAction {
private String name,password,email,gender,country;
  //setters and getters
public String execute(){
    int i=RegisterDao.save(this);
    if(i>0){
```

```
    return "success";
    }
    return "error";
}
}
```

3) Create the class to store data (RegisterDao.java)

This class gets information from the object of RegisterAction class and stores these information in the strutsuser table.

RegisterDao.java

```
package p1;
import java.sql.*;
public class RegisterDao {
  public static int save(RegisterAction r){
int status=0;
try{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
  PreparedStatement ps=con.prepareStatement("insert into strutsuser values(?,?,?,?,?)");
ps.setString(1,r.getName());
ps.setString(2,r.getPassword());
ps.setString(3,r.getEmail());
ps.setString(4,r.getGender());
ps.setString(5,r.getCountry());
```

```
    status=ps.executeUpdate();
  }catch(Exception e){e.printStackTrace();}
    return status;
}
}
```

4) Map the request in (struts.xml) file and define the view components

This xml file contains information about the package, action class and view components.

struts.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD
Struts Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">
<struts>
  <package name="default" extends="struts-default">
  <action name="register" class="p1.RegisterAction">
<result name="success">register-success.jsp</result>
<result name="error">register-error.jsp</result>
</action>
  </package>
</struts>
```

5) Create view components

Here, we are creating two view components register-success.jsp and register-error.jsp.
register-success.jsp

```
<%@ taglib uri="/struts-tags" prefix="s" %>
Welcome, <s:property value="name"></s:property>
```

register-error.jsp

```
<%@ taglib uri="/struts-tags" prefix="s" %>

Sorry, some error occured!

<s:include value="index.jsp"></s:include>
```

To fetch all the records, we have stored all the records in a collection (using List), and displaying the data of the collection using the iterator tag of struts2.

Here, we assume that you have a table in oracle database named user3333 that contains records. The table query is:

```
CREATE TABLE  "USER3333"

  (    "ID" NUMBER,

   "NAME" VARCHAR2(4000),

   "PASSWORD" VARCHAR2(4000),

   "EMAIL" VARCHAR2(4000),

    CONSTRAINT "USER3333_PK" PRIMARY KEY ("ID") ENABLE

  )
```

Example to fetch all the records of the table

In this example, we are creating 5 pages :
index.jsp invoking action.

Register.java for storing data of the table in the collection.

User.java for representing table.

struts.xml for defining the action and result.

welcome.jsp for the view component to display records.

1) Create index.jsp for invoking action (optional)

This jsp page creates a link to invoke the action. But you can direct invoke the action class.

index.jsp

```
<a href="viewrecords">View All Records</a>
```

2) Create the action class

This action class contains ArrayList object as the datamember and execute method.

Register.java

```
package p1;

import java.sql.*;

import java.util.ArrayList;


public class FetchRecords {

ArrayList<User> list=new ArrayList<User>();


public ArrayList<User> getList() {

   return list;

}

public void setList(ArrayList<User> list) {

   this.list = list;

}

public String execute(){

 try{

  Class.forName("oracle.jdbc.driver.OracleDriver");

  Connection con=DriverManager.getConnection(
    "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");


  PreparedStatement ps=con.prepareStatement("select * from user3333");
  ResultSet rs=ps.executeQuery();


  while(rs.next()){
   User user=new User();
   user.setId(rs.getInt(1));
   user.setName(rs.getString(2));
   user.setPassword(rs.getString(3));
   user.setEmail(rs.getString(4));
   list.add(user);
  }
    con.close();
 }catch(Exception e){e.printStackTrace();}
      return "success";
}
}
```

3) Create the class to represent table

This is the simple bean class containing 4 fields.

User.java

```
package p1;
 public class User {
private int id;
private String name,password,email;
//getters and setters
}
```

4) Create struts.xml

This xml file defines action and result.

struts.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD
Struts Configuration 2.1//EN"
"http://struts.apache.org/dtds/struts-2.1.dtd">
<struts>
  <package name="anbc" extends="struts-default">
<action name="viewrecords" class="p1.FetchRecords">
<result name="success">displayrecords.jsp</result>
</action>
</package>
  </struts>
```

5) Create view component

It is the simple jsp file displaying the information of the user.

welcome.jsp

```
<%@ taglib uri="/struts-tags" prefix="s" %>
  <h3>All Records:</h3>
<s:iterator  value="list">
<fieldset>
<s:property value="id"/><br/>
```

```
<s:property value="name"/><br/>
<s:property value="password"/><br/>
<s:property value="email"/><br/>
</fieldset>
</s:iterator>
```

Hibernate and Struts 2 Integration

We can integrate any struts application with hibernate. There is no requirement of extra efforts.In this example, we going to use struts 2 framework with hibernate. You need to have jar files for struts 2 and hibernate.

Example of Hibernate and struts2 integration

In this example, we are creating the registration form using struts2 and storing this data into the database using Hibernate. Let's see the files that we should create to integrate the struts2 application with hibernate.

- index.jsp file to get input from the user.
- User.java A action class for handling the request. It uses the dao class to store the data.
- RegisterDao.java A java class that uses DAO design pattern to store the data using hibernate.
- user.hbm.xml A mapping file that contains information about the persistent class. In this case, action class works as the persistent class.

- hibernate.cfg.xml A configuration file that contains informations about the database and mapping file.
- struts.xml file contains information about the action class and result page to be invoked.
- welcome.jsp A jsp file that displays the welcome information with username.
- web.xml A web.xml file that contains information about the Controller of Struts framework.

index.jsp

In this page, we have created a form using the struts tags. The action name for this form is register.

```
<%@ taglib uri="/struts-tags" prefix="S" %>
  <S:form action="register">
<S:textfield name="name" label="Name"></S:textfield>
<S:submit value="register"></S:submit>
  </S:form>
```

User.java

It is a simple POJO class. Here it works as the action class for struts and persistent class for hibernate. It calls the register method of RegisterDao class and returns success as the string.

```
package p1;
```

```
public class User {
private int id;
private String name;
//getters and setters

public String execute(){
    RegisterDao.saveUser(this);
    return "success";
}

}
```

RegisterDao.java

It is a java class that saves the object of User class using the Hibernate framework.

```
package p1;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
  public class RegisterDao {
  public static int saveUser(User u){
          Session session=new Configuration(). configure("hibernate.cfg.xml").buildSession
Factory().openSession();
          Transaction t=session.beginTransaction();
          int i=(Integer)session.save(u);
         t.commit();
        session.close();
        return i;
  }
  }
```

user.hbm.xml

This mapping file contains all the information of the persitent class.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
  <hibernate-mapping>
<class name="p1.User" table="user451">
<id name="id">
<generator class="increment"></generator>
</id>
<property name="name"></property>
</class>
   </hibernate-mapping>
```

hibernate.cfg.xml

This configuration file contains informations about the database and mapping file. Here, we are using the hb2ddl.auto property, so you don't need to create the table in the database.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
```

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

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


<hibernate-configuration>


<session-factory>

<property name="hbm2ddl.auto">update</property>

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

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

<property name="connection.username">system</property>

<property name="connection.password">oracle</property>

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

<mapping resource="user.hbm.xml"/>

  </session-factory>

  </hibernate-configuration>
```

struts.xml

This files contains information about the action class to be invoked. Here the action class is User.

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

<!DOCTYPE struts PUBLIC "-//Apache Software Foundation

//DTD Struts Configuration 2.1//EN"

"http://struts.apache.org/dtds/struts-2.1.dtd">

<struts>
```

```
<package name="abc" extends="struts-default">

<action name="register" class="p1.User">

<result name="success">welcome.jsp</result>

</action>

</package>

</struts>
```

welcome.jsp

It is the welcome file, that displays the welcome message with username.

```
<%@ taglib uri="/struts-tags" prefix="S" %>
  Welcome: <S:property value="name"/>
```

web.xml

It is web.xml file that contains the information about the controller. In case of Struts2, StrutsPrepareAndExecuteFilter class works as the controller.

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

<web-app version="2.5"

    xmlns="http://java.sun.com/xml/ns/javaee"

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

    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee

    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

  <welcome-file-list>
```

```
    <welcome-file>index.jsp</welcome-file>

  </welcome-file-list>

  <filter>

    <filter-name>struts2</filter-name>

    <filter-class>

  org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter

    </filter-class>

  </filter>

  <filter-mapping>

    <filter-name>struts2</filter-name>

    <url-pattern>/*</url-pattern>

  </filter-mapping>

</web-app>
```

SPRING

Spring Framework

- Spring was developed by Rod Johnson in 2003.
- Spring framework makes the easy development of JavaEE application.
- It is helpful for beginners and experienced persons.
- Spring is a lightweight framework.
- It can be thought of as a framework of frameworks because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF etc.
- The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.
- The Spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC etc.

Advantages of Spring Framework

There are many advantages of Spring Framework. They are as follows:

1) Predefined Templates

Spring framework provides templates for JDBC, Hibernate, JPA etc. technologies. So there is no need to write too much code. It hides the basic steps of these technologies.

Let's take the example of JdbcTemplate, you don't need to write the code for exception handling, creating connection, creating statement, committing transaction, closing connection etc. You need to write the code of executing query only. Thus, it save a lot of JDBC code.

2) Loose Coupling

The Spring applications are loosely coupled because of dependency injection.

3) Easy to test

The Dependency Injection makes easier to test the application. The EJB or Struts application require server to run the application but Spring framework doesn't require server.

4) Lightweight

Spring framework is lightweight because of its POJO implementation. The Spring Framework doesn't force the programmer to inherit any class or implement any interface. That is why it is said non-invasive.

5) Fast Development

The Dependency Injection feature of Spring Framework and it support to various frameworks makes the easy development of JavaEE application.

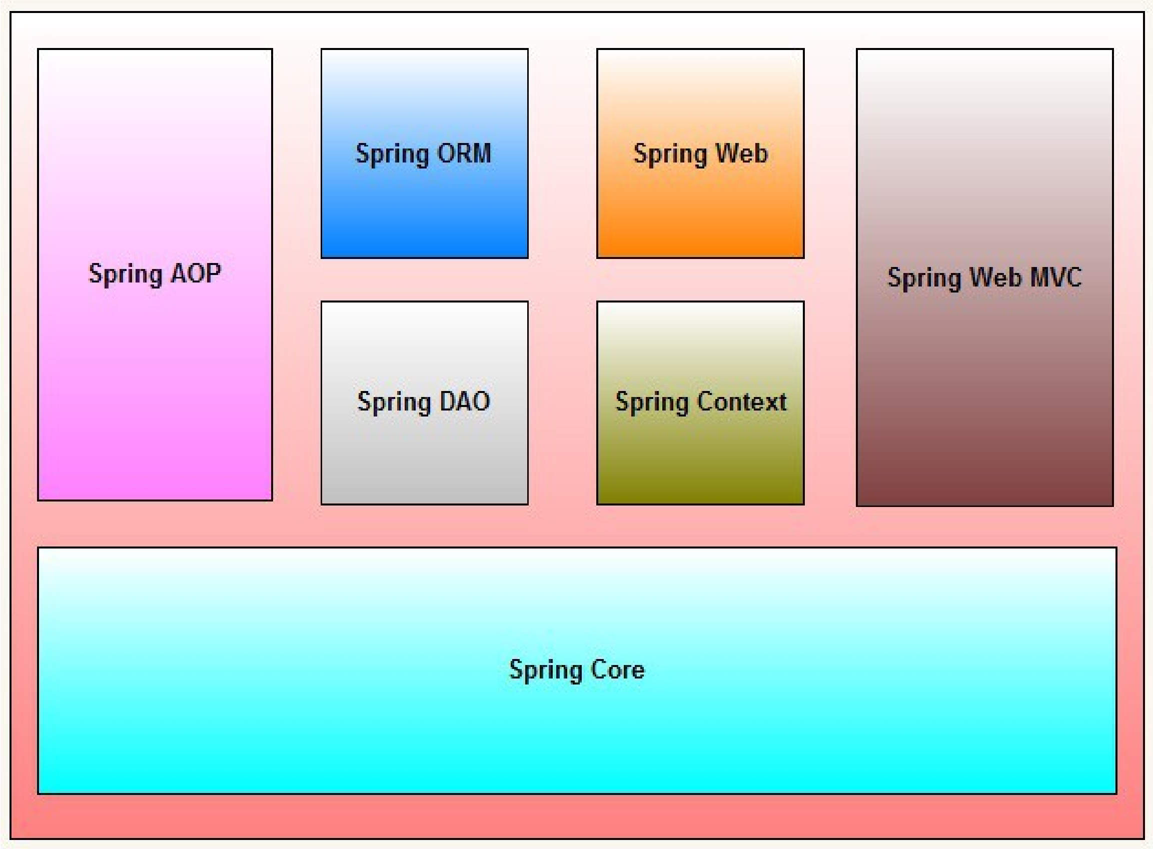
6) Powerful abstraction

It provides powerful abstraction to JavaEE specifications such as JMS, JDBC, JPA and JTA.

7) Declarative support

It provides declarative support for caching, validation, transactions and formatting.

Architecture:



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

Spring AOP:

One of the key components of Spring is the AOP framework.

AOP is used in Spring:

- To provide declarative enterprise services, especially as a replacement for EJB declarative services.

- The most important such service is declarative transaction management, which builds on Spring's transaction abstraction.To allow users to implement custom aspects, complementing their use of OOP with AOP

Spring ORM:

- The ORM package is related to the database access.
- It provides integration layers for popular object-relational mapping APIs, including JDO, Hibernate and iBatis.

Spring Web:

- The Spring Web module is part of Spring's web application development stack, which includes Spring MVC.

Spring DAO:

- The DAO (Data Access Object) support in Spring is primarily for standardizing the data access work using the technologies like JDBC, Hibernate or JDO.

Spring Context:

- This package builds on the beans package to add support for message sources and for the Observer design pattern, and the ability for application objects to obtain resources using a consistent API.

Spring Web MVC:

- This is the Module which provides the MVC implementations for the web applications.

Spring Core:

- The Core package is the most import component of the Spring Framework.
- This component provides the Dependency Injection features.
- The BeanFactory provides a factory pattern which separates the dependencies like initialization, creation and access of the objects from your actual program logic.

IoC Container

- The IoC container is responsible to instantiate, configure and assemble the objects.
- The IoC container gets informations from the XML file and works accordingly. The main tasks performed by IoC container are:
    - to instantiate the application class
    - to configure the object
    - to assemble the dependencies between the objects
    - There are two types of IoC containers. They are:
    - BeanFactory

    - ApplicationContext

Difference between BeanFactory and the ApplicationContext

- The org.springframework.beans.factory.BeanFactory and the org.springframework. context.ApplicationContext interfaces acts as the IoC container.

- The ApplicationContext interface is built on top of the BeanFactory interface.

- It adds some extra functionality than BeanFactory such as simple integration with Spring's AOP, message resource handling (for I18N), event propagation, application layer specific context (e.g. WebApplicationContext) for web application. So it is better to use ApplicationContext than BeanFactory.

Using BeanFactory

- The XmlBeanFactory is the implementation class for the BeanFactory interface. To use the BeanFactory, we need to create the instance of XmlBeanFactory class as given below:

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

- BeanFactory factory=new XmlBeanFactory(resource);

- The constructor of XmlBeanFactory class receives the Resource object so we need to pass the resource object to create the object of BeanFactory.

Using ApplicationContext

- The ClassPathXmlApplicationContext class is the implementation class of ApplicationContext interface. We need to instantiate the

ClassPathXmlApplicationContext class to use the ApplicationContext as given below:

- ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
- The constructor of ClassPathXmlApplicationContext class receives string, so we can pass the name of the xml file to create the instance of ApplicationContext.

Spring HelloWorld Example in Netbeans

After having some basic understanding of spring framework now we are ready to create Hello world example.

Now in NetBeansIDE click on File->NewProject 7

Then click on Java->Java Application

Click on Next and Write Project name .Here the name is JavaApplication14

Click on Finish and now project is created.

Now we will add the Spring 3. libraries to the project.

Right click on Libraries and then click on Add Library.

Now select Spring Framework 3.1.1 Release

Create a new package p1 to hold the java files. Right click on the source packages folder and then select New --> Package. Then provide the package name as p1 and click on the "Finish" button.

Create a new Java file **HelloWorld.java** under the package **p1** and add the following code:

Now create XML file by clicking on New->Other

Click on Other->SpringXMLConfigurationFile

Click next and enter name then click Finish

Now the file will be like this:

- Now add tags to this file.
- Here <beans> tag is for defining multiples beans.
- All beans of our program are defined in <beans> tag.
- <bean> tag is for defining a single bean.
- "id" is for providing unique identification to bean."class" is fully qualified name of class."property" is used for defining attribute of bean class.

  The below xml file declares the spring bean "HelloWorldBean" of the class **HelloWorld.**

Now add the code to main method:

In the above code we have created the instance of ApplicationContext and the retrieved the “HelloWorldBean". Then we called the printHello() method on the bean.

RUN PROGRAM:

right click on project->Run

Dependency Injection:

- The basic concept of the dependency injection (also known as Inversion of Control pattern) is that you do not create your objects but describe how they should be created.

- You don't directly connect your components and services together in code but describe which services are needed by which components in a configuration file.

- A container (in the case of the Spring framework, the IOC container) is then responsible for hooking it all up.i.e., Applying IoC, objects are given their dependencies at creation time by some external entity that coordinates each object in the system.

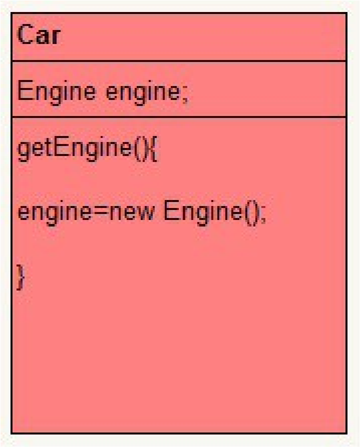
- That is, dependencies are injected into objects. So, IoC means an inversion of responsibility with regard to how an object obtains references to collaborating objects.

Example:

Lets we have two classes-**Car** and **Engine.**Car has a object of Engine.

Normal way:

There are many ways to instantiate a object. A simple and common way is with new operator.so here Car class contain object of Engine and we have it instantiated using new operator.

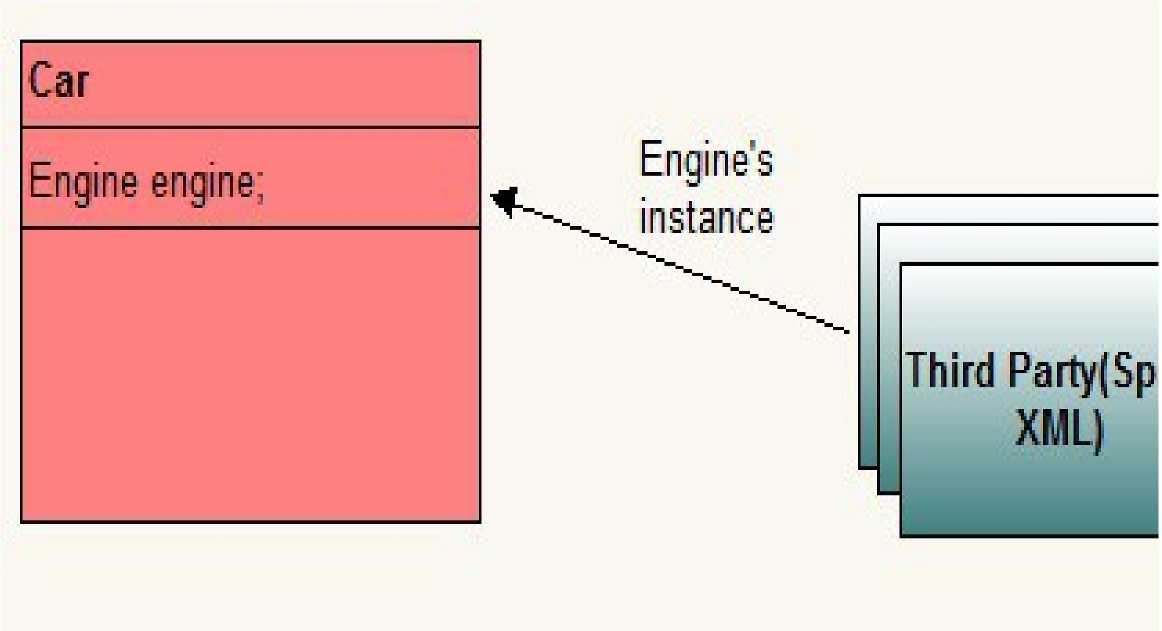


```
Engine engine;
getEngine(){
engine=new Engine();
}
```

WITHOUT DI

With help of Dependency Injection:

Now we outsource instantiation and supply job of instantiating to third party. **Car** needs object of **Engine** to operate but it outsources that job to some third party. The designated third party, decides the moment of instantiation and the type to use to create the instance. The dependency between class **Car** and class **Engine** is injected by a third party. Whole of this agreement involves some configuration information too. This whole process is called dependency injection.



WITH DI

BENIFITS OF DEPENDENCY INJECTION IN SPRING:

- Ensures configuration and uses of services are separate.
- Can switch implementations by just changing configuration.
- Enhances Testability as mock dependencies can be injected.
- Dependencies can be easily identified.
- No need to read code to see what dependencies your code talks to.

TYPES OF DEPENDENCY INJECTION:

Setter Injection:

Setter-based DI is realized by calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

Constructor Injection:

Constructor-based DI is realized by invoking a constructor with a number of arguments, each representing a collaborator.

Interface Injection:

In interface-based dependency injection, we will have an interface and on implementing it we will get the instance injected.

Dependency Injection using setter methods

COUNTRY.JAVA:

- This is simple pojo class having some attributes so here country has name and object of Capital class.
- Create Country.java under package p1.Copy following content into Country.java.

```
package p1;
 public class Country {
   String countryName;
  Capital capital;
  public String getCountryName() {
    return countryName;
  }
  public void setCountryName(String countryName) {
    this.countryName = countryName;
  }
  public Capital getCapital() {
    return capital;
  }
  public void setCapital(Capital capital) {
    this.capital = capital;
  }
}
```

2.CAPITAL.JAVA

- This is also simple pojo class having one attribute called "capitalName".
- Create Capital.java under package p1.
- Above Country class contains object of this class.Copy following content into Capital.java

```
package p1;
  public class Capital {

    String capitalName;

    public String getCapitalName() {
        return capitalName;
    }

    public void setCapitalName(String capitalName) {
        this.capitalName = capitalName;
    }
}
```

3.SETTERMEHTODMAIN.JAVA

This class contains main function.Create SetterMethodMain.java under package p1.Copy following content into SetterMethodMain.java

```
 package p1;
 import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.core.io.ClassPathResource;


public class SetterInjectionMain {
    public static void main(String[] args) {
      ApplicationContext appContext = new ClassPathXmlApplicationContext("Applicati
onContext.xml");
        Country countryObj = (Country) appContext.getBean("CountryBean");
      String countryName=countryObj.getCountryName();
      String capitalName=countryObj.getCapital().getCapitalName();
      System.out.println(capitalName+" is capital of "+countryName);


   }
}
```

You can note here that we have used ClassPathXmlApplicationContext for getting bean here.There are various ways for getting beans.

4.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: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">

   <bean id="CountryBean" class="p1.Country">

     <property name="countryName" value="India"/>

     <property name="capital" ref="CapitalBean"/>

  </bean>

  <bean id="CapitalBean" class="p1.Capital">

     <property name="capitalName" value="Delhi"/>

  </bean>

</beans>
```

Here We have declared two beans with corresponding ids.

1.Class Country with id as "CountryBean"

2.Class Capital with id as "CapitalBean"

Property's value tag is for assigning value to corresponding attribute. so In above xml file,we have assigned countryName attribute of Country class with value as india

```
<property name="Name Of Attribute" value="Value Of attribute to be assigned"/>
```

Property's ref tag is for assigning reference to corresponding attribute. so In above xml file,we have assigned reference of Capital class to capital attribute of Country class.

```
<property name="Name Of Attribute" ref="id of referencing bean"/>
```

5.RUN IT

When you will run above application,you will get following as output.

```
Delhi is capital of India
```

Dependency Injection using constructors

1.COUNTRY.JAVA:

```
package p1;


public class Country {


   String countryName;
   Capital capital;


   public Country(String countryName, Capital capital) {
      super();
      this.countryName = countryName;
      this.capital = capital;
   }
   public String getCountryName() {
      return countryName;
```

```
    }


    public Capital getCapital() {
        return capital;
    }


}
```

2.CAPITAL.JAVA

```
package p1;
  public class Capital {
      String capitalName;


    public String getCapitalName() {
        return capitalName;
    }


    public void setCapitalName(String capitalName) {
        this.capitalName = capitalName;
    }
}
```

3.CONSTRUCTORDIMAIN.JAVA

```
 package p1;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import org.springframework.core.io.ClassPathResource;


public class ConstructorDIMain{


    public static void main(String[] args) {

        ApplicationContext appContext = new ClassPathXmlApplicationContext("Applicati
onContext.xml");


        Country countryObj = (Country) appContext.getBean("CountryBean");

        String countryName=countryObj.getCountryName();

        String capitalName=countryObj.getCapital().getCapitalName();

        System.out.println(capitalName+" is capital of "+countryName);


    }

}
```

You can note here that we have used ClassPathXmlApplicationContext for getting bean here.There are various ways for getting beans.In hello world example we have used XmlBeanFactory for getting beans.

4.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: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">

   <bean id="CountryBean" class="p1.Country">
  <constructor-arg index="0" type="java.lang.String" value="India" />
  <constructor-arg index="1" ref="CaptialBean" />
 </bean>
 <bean id="CaptialBean" class="p1.Capital">
  <property name="capitalName" value="Delhi" />
 </bean>
</beans>
```

Here We have declared two beans with corresponding ids.

1.Class Country with id as "CountryBean"

2.Class Capital with id as "CapitalBean"

constructor-arg tag is used for providing argument to bean' s constructor.type is for declaring data types and index defines position in constructor's argument.

In above xml,Two arguments are passed.

1. India as string

2.CapitalBean 's reference

Property's value tag is for assigning value to corresponding attribute. so In above xml file,we have assigned capitalName attribute of Capital class with value as Delhi

```
<property name="Name Of Attribute" value="Value Of attribute to be assigned"/>
```

Property's ref tag is for assigning reference to corresponding attribute. so In above xml file,we have assigned reference of Capital class to capital attribute of Country class.

```
<property name="Name Of Attribute" value="id of referencing bean"/>
```

5.RUN IT

When you will run above application,you will get following as output.

```
Delhi is capital of India
```

Constructor Injection with Dependent Object

If there is HAS-A relationship between the classes, we create the instance of dependent object (contained object) first then pass it as an argument of the main class constructor.

Here, our scenario is Employee HAS-A Address.

The Address class object will be termed as the dependent object.

Let's see the Address class first:

Address.java

This class contains three properties, one constructor and toString() method to return the values of these object.

```
package mypack;


public class Address {

private String city;

private String state;

private String country;
public Address(String city, String state, String country) {
    super();
    this.city = city;
    this.state = state;
    this.country = country;
}

public String toString(){
    return city+" "+state+" "+country;
}
}
```

Employee.java

It contains three properties id, name and address(dependent object) ,two constructors and show() method to show the records of the current object including the depedent object.

```
package mypack;
  public class Employee {
private int id;
private String name;
private Address address;//Aggregation
  public Employee() {System.out.println("def cons");}
  public Employee(int id, String name, Address address) {
```

```java
    super();
    this.id = id;
    this.name = name;
    this.address = address;
}
void show(){
    System.out.println(id+" "+name);
    System.out.println(address.toString());
}
  }
```

applicationContext.xml

The ref attribute is used to define the reference of another object, such way we are passing the dependent object as an constructor argument.

```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="a1" class="mypack.Address">
```

```
<constructor-arg value="Kadavantra"></constructor-arg>

<constructor-arg value="Kerala"></constructor-arg>

<constructor-arg value="India"></constructor-arg>

</bean>


<bean id="e" class="mypack.Employee">

<constructor-arg value="12" type="int"></constructor-arg>

<constructor-arg value="Soften Technologies"></constructor-arg>

<constructor-arg>

<ref bean="a1"/>

</constructor-arg>

</bean>

  </beans>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the show method.

```
package mypack;

 import org.springframework.beans.factory.BeanFactory;

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

import org.springframework.core.io.*;
```

```
public class Test {

  public static void main(String[] args) {

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

      BeanFactory factory=new XmlBeanFactory(r);

      Employee s=(Employee)factory.getBean("e");

      s.show();

      }

}
```

Setter Injection with Dependent Object Example

Like Constructor Injection, we can inject the dependency of another bean using setters. In such case, we use property element. Here, our scenario is Employee HAS-A Address. The Address class object will be termed as the dependent object. Let's see the Address class first:

Address.java

This class contains four properties, setters and getters and toString() method.

```
package mypack;
```

```
  public class Address {
private String addressLine1,city,state,country;


//getters and setters


public String toString(){
   return addressLine1+" "+city+" "+state+" "+country;
}
```

Employee.java

It contains three properties id, name and address(dependent object) , setters and getters with displayInfo() method.

```
package mypack;


public class Employee {
private int id;
private String name;
private Address address;


//setters and getters


void displayInfo(){
   System.out.println(id+" "+name);
   System.out.println(address);
}
}
```

applicationContext.xml

The ref attribute of property elements is used to define the reference of another bean.

```
<?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="address1" class="mypack.Address">
<property name="addressLine1" value="51,Lohianagar"></property>
<property name="city" value="Ghaziabad"></property>
<property name="state" value="UP"></property>
<property name="country" value="India"></property>
</bean>

<bean id="obj" class="mypack.Employee">
<property name="id" value="1"></property>
<property name="name" value="Sachin Yadav"></property>
<property name="address" ref="address1"></property>
</bean>
```

```
</beans>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the displayInfo() method.

```
package mypack;


import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;


public class Test {
public static void main(String[] args) {
    Resource r=new ClassPathResource("applicationContext.xml");
    BeanFactory factory=new XmlBeanFactory(r);


    Employee e=(Employee)factory.getBean("obj");
    e.displayInfo();


}
}
```

Constructor Injection with Collection Example

We can inject collection values by constructor in spring framework. There can be used three elements inside the constructor-argelement.

It can be:

list

set

map

Each collection can have string based and non-string based values.

In this example, we are taking the example of Forum where One question can have multiple answers.In this example, we are using list that can have duplicate elements, you may use set that have only unique elements. But, you need to change list to set in the applicationContext.xml file and List to Set in the Question.java file.

Question.java

This class contains three properties, two constructors and displayInfo() method that prints the information. Here, we are using List to contain the multiple answers.

```
package mypack;
  import java.util.Iterator;
import java.util.List;
  public class Question {
private int id;
private String name;
private List<String> answers;


public Question() {}
```

```java
public Question(int id, String name, List<String> answers) {
    super();
    this.id = id;
    this.name = name;
    this.answers = answers;
}

public void displayInfo(){
    System.out.println(id+" "+name);
    System.out.println("answers are:");
    Iterator<String> itr=answers.iterator();
    while(itr.hasNext()){
        System.out.println(itr.next());
    }
}

}
```

applicationContext.xml

The list element of constructor-arg is used here to define the list.

```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="q" class="mypack.Question">
<constructor-arg value="111"></constructor-arg>
<constructor-arg value="What is java?"></constructor-arg>
<constructor-arg>
<list>
<value>Java is a programming language</value>
<value>Java is a Platform</value>
<value>Java is an Island of Indonasia</value>
</list>
</constructor-arg>
</bean>
 </beans>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the displayInfo method.

```
package mypack;
  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 r=new ClassPathResource("applicationContext.xml");
    BeanFactory factory=new XmlBeanFactory(r);


    Question q=(Question)factory.getBean("q");
    q.displayInfo();


}
}
```

Constructor Injection with Non-String Collection (having Dependent Object) Example

- If we have dependent object in the collection, we can inject these information by using the ref element inside the list, set or map.
- In this example, we are taking the example of Forum where One question can have multiple answers. But Answer has its own information such as answerId, answer and postedBy. There are four pages used in this example:
- In this example, we are using list that can have duplicate elements, you may use set that have only unique elements. But, you need to change list to set in the applicationContext.xml file and List to Set in the Question.java file.

Question.java

- This class contains three properties, two constructors and displayInfo() method that prints the information.
- Here, we are using List to contain the multiple answers.

```
package mypack;


import java.util.Iterator;
import java.util.List;


public class Question {
private int id;
private String name;
private List<Answer> answers;


public Question() {}
public Question(int id, String name, List<Answer> answers) {
    super();
    this.id = id;
    this.name = name;
    this.answers = answers;
}


public void displayInfo(){
    System.out.println(id+" "+name);
    System.out.println("answers are:");
    Iterator<Answer> itr=answers.iterator();
    while(itr.hasNext()){
        System.out.println(itr.next());
    }
}


}
```

Answer.java

This class has three properties id, name and by with constructor and toString() method.

```
package mypack;


public class Answer {
private int id;
private String name;
private String by;


public Answer() {}
public Answer(int id, String name, String by) {
    super();
    this.id = id;
    this.name = name;
    this.by = by;
}


public String toString(){
    return id+" "+name+" "+by;
}
}
```

applicationContext.xml

The ref element is used to define the reference of another bean. Here, we are using bean attribute of ref element to specify the reference of another bean.

```
<?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="ans1" class="mypack.Answer">
<constructor-arg value="1"></constructor-arg>
<constructor-arg value="Java is a programming language"></constructor-arg>
<constructor-arg value="John"></constructor-arg>
</bean>

<bean id="ans2" class="mypack.Answer">
<constructor-arg value="2"></constructor-arg>
<constructor-arg value="Java is a Platform"></constructor-arg>
<constructor-arg value="Ravi"></constructor-arg>
</bean>
  <bean id="q" class="mypack.Question">
<constructor-arg value="111"></constructor-arg>
<constructor-arg value="What is java?"></constructor-arg>
```

```
<constructor-arg>
<list>
<ref bean="ans1"/>
<ref bean="ans2"/>
</list>
</constructor-arg>
</bean>


</beans>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the displayInfo method.

```
package mypack;


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 r=new ClassPathResource("applicationContext.xml");
   BeanFactory factory=new XmlBeanFactory(r);


   Question q=(Question)factory.getBean("q");
   q.displayInfo();


}
     }
```

Constructor Injection with Non-String Map (having dependent Object) Example

In this example, we are using map as the answer that have Answer and User. Here, we are using key and value pair both as an object. Answer has its own information such as answerId, answer and postedDate, User has its own information such as userId, username, emailId.

Like previous examples, it is the example of forum where one question can have multiple answers.

Question.java

This class contains three properties, two constructors and displayInfo() method to display the information.

```
package  mypack;
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 Question() {}
public Question(int id, String name, Map<Answer, User> answers) {
    super();
    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<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);
    }
}
```

```
}
```

Answer.java

```
package mypack;
  import java.util.Date;


public class Answer {
private int id;
private String answer;
private Date postedDate;
public Answer() {}
public Answer(int id, String answer, Date postedDate) {
    super();
    this.id = id;
    this.answer = answer;
    this.postedDate = postedDate;
}


public String toString(){
    return "Id:"+id+" Answer:"+answer+" Posted Date:"+postedDate;
}
}
```

User.java

```
package mypack;
  public class User {
private int id;
private String name,email;

public User() {}

public User(int id, String name, String email) {
    super();
    this.id = id;
    this.name = name;
    this.email = email;
}


public String toString(){
    return "Id:"+id+" Name:"+name+" Email Id:"+email;
}
}
```

applicationContext.xml

The key-ref and value-ref attributes of entry element is used to define the reference of bean in the map.

```
<?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="answer1" class="mypack.Answer">
```

```
<constructor-arg value="1"></constructor-arg>
<constructor-arg value="Java is a Programming Language"></constructor-arg>
<constructor-arg value="12/12/2001"></constructor-arg>
</bean>
<bean id="answer2" class="mypack.Answer">
<constructor-arg value="2"></constructor-arg>
<constructor-arg value="Java is a Platform"></constructor-arg>
<constructor-arg value="12/12/2003"></constructor-arg>
</bean>

<bean id="user1" class="mypack.User">
<constructor-arg value="1"></constructor-arg>
<constructor-arg value="Arun Kumar"></constructor-arg>
<constructor-arg value="arun@gmail.com"></constructor-arg>
</bean>
<bean id="user2" class="mypack.User">
<constructor-arg value="2"></constructor-arg>
<constructor-arg value="Varun Kumar"></constructor-arg>
<constructor-arg value="Varun@gmail.com"></constructor-arg>
</bean>

<bean id="q" class="mypack.Question">
<constructor-arg value="1"></constructor-arg>
<constructor-arg value="What is Java?"></constructor-arg>
<constructor-arg>
```

```
<map>
<entry key-ref="answer1" value-ref="user1"></entry>
<entry key-ref="answer2" value-ref="user2"></entry>
</map>
</constructor-arg>
</bean>

</beans>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the displayInfo() method to display the information.

```
package mypack;
  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 r=new ClassPathResource("applicationContext.xml");
   BeanFactory factory=new XmlBeanFactory(r);
       Question q=(Question)factory.getBean("q");
   q.displayInfo();
     }
}
```

Inheriting Bean in Spring

- By using the parent attribute of bean, we can specify the inheritance relation between the beans.
- In such case, parent bean values will be inherited to the current bean.

Let's see the simple example to inherit the bean.

Employee.java

This class contains three properties, three constructor and show() method to display the values.

```
package mypack;


public class Employee {

private int id;

private String name;

private Address address;

public Employee() {}


public Employee(int id, String name) {

    super();

    this.id = id;

    this.name = name;

}

public Employee(int id, String name, Address address) {
    super();
    this.id = id;
    this.name = name;
    this.address = address;
}
  void show(){
    System.out.println(id+" "+name);
    System.out.println(address);
}
  }
```

Answer.java

```
package mypack;
  public class Address {
private String addressLine1,city,state,country;
  public Address(String addressLine1, String city, String state, String country) {
    super();
    this.addressLine1 = addressLine1;
    this.city = city;
    this.state = state;
    this.country = country;
}
public String toString(){
    return addressLine1+" "+city+" "+state+" "+country;
}
}
```

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-3.0.xsd">

<bean id="e1" class="mypack.Employee">
<constructor-arg value="101"></constructor-arg>
<constructor-arg  value="Sachin"></constructor-arg>
</bean>

<bean id="address1" class="mypack.Address">
<constructor-arg value="21,Lohianagar"></constructor-arg>
<constructor-arg value="Ghaziabad"></constructor-arg>
<constructor-arg value="UP"></constructor-arg>
<constructor-arg value="USA"></constructor-arg>
</bean>

<bean id="e2" class="mypack.Employee" parent="e1">
<constructor-arg ref="address1"></constructor-arg>
</bean>
```

```
 </beans>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the show method.

```
package mypack;


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 r=new ClassPathResource("applicationContext.xml");

   BeanFactory factory=new XmlBeanFactory(r);



   Employee e1=(Employee)factory.getBean("e2");

   e1.show();



}

}
```

Setter Injection with Non-String Collection (having Dependent Object) Example

If we have dependent object in the collection, we can inject these information by using the ref element inside the list, set or map. Here, we will use list, set or map element inside the propertyelement.

In this example, we are using list that can have duplicate elements, you may use set that have only unique elements. But, you need to change list to set in the applicationContext.xml file and List to Set in the Question.java file.

Question.java

This class contains three properties, two constructors and displayInfo() method that prints the information. Here, we are using List to contain the multiple answers.

Question.java

```
package mypack;

  import java.util.Iterator;

import java.util.List;
  public class Question {
private int id;
private String name;
private List<Answer> answers;


//setters and getters


public void displayInfo(){
    System.out.println(id+" "+name);
    System.out.println("answers are:");
    Iterator<Answer> itr=answers.iterator();
    while(itr.hasNext()){
        System.out.println(itr.next());
    }
}


}
```

Answer.java

```
package mypack;
  public class Answer {
private int id;
private String name;
private String by;

//setters and getters

public String toString(){
    return id+" "+name+" "+by;
}
}
```

applicationContext.xml

The ref element is used to define the reference of another bean. Here, we are using bean attribute of ref element to specify the reference of another bean.

```
<?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="answer1" class="mypack.Answer">
<property name="id" value="1"></property>
<property name="name" value="Java is a programming language"></property>
<property name="by" value="Ravi Malik"></property>
</bean>
<bean id="answer2" class="mypack.Answer">
<property name="id" value="2"></property>
<property name="name" value="Java is a platform"></property>
<property name="by" value="Sachin"></property>
</bean>

<bean id="q" class="mypack.Question">
<property name="id" value="1"></property>
<property name="name" value="What is Java?"></property>
```

```
<property name="answers">
<list>
<ref bean="answer1"/>
<ref bean="answer2"/>
</list>
</property>
</bean>
  </beans>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the displayInfo

```
package mypack;

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 r=new ClassPathResource("applicationContext.xml");
    BeanFactory factory=new XmlBeanFactory(r);

    Question q=(Question)factory.getBean("q");
    q.displayInfo();
}
}
```

Setter Injection with Non-String Map (having dependent Object) Example

- In this example, we are using map as the answer that have Answer and User. Here, we are using key and value pair both as an object.
- Answer has its own information such as answerId, answer and postedDate, User has its own information such as userId, username, emailId.

Like previous examples, it is the example of forum where one question can have multiple answers.

Question.java

This class contains three properties, getters & setters and displayInfo() method to display the information.

```
package mypack;
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;
//getters and setters


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);
    }
}
}
```

Answer.java

```
package mypack;


import java.util.Date;


public class Answer {
```

```
private int id;
private String answer;
private Date postedDate;
public Answer() {}
public Answer(int id, String answer, Date postedDate) {
    super();
    this.id = id;
    this.answer = answer;
    this.postedDate = postedDate;
}

public String toString(){
    return "Id:"+id+" Answer:"+answer+" Posted Date:"+postedDate;
}
}
```

User.java

```
package mypack;

public class User {
private int id;
private String name,email;
public User() {}
public User(int id, String name, String email) {
    super();
    this.id = id;
    this.name = name;
    this.email = email;
}

public String toString(){
    return "Id:"+id+" Name:"+name+" Email Id:"+email;
}
}
```

applicationContext.xml

The key-ref and value-ref attributes of entry element is used to define the reference of bean in the map.

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

```
<bean id="answer2" class="mypack.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="mypack.User">
<property name="id" value="1"></property>
<property name="name" value="Arun Kumar"></property>
<property name="email" value="arun@gmail.com"></property>
</bean>
<bean id="user2" class="mypack.User">
<property name="id" value="2"></property>
<property name="name" value="Varun Kumar"></property>
<property name="email" value="Varun@gmail.com"></property>
</bean>

<bean id="q" class="mypack.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

This class gets the bean from the applicationContext.xml file and calls the displayInfo() method to display the information.

```
package mypack;


  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 r=new ClassPathResource("applicationContext.xml");
    BeanFactory factory=new XmlBeanFactory(r);
          Question q=(Question)factory.getBean("q");
    q.displayInfo();


      }
}
```

Difference between constructor and setter injection

There are many key differences between constructor injection and setter injection.

- Partial dependency: can be injected using setter injection but it is not possible by constructor. Suppose there are 3 properties in a class, having 3 arg constructor and setters methods. In such case, if you want to pass information for only one property, it is possible by setter method only.
- Overriding: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.
- Changes: We can easily change the value by setter injection. It doesn't create a new bean instance always like constructor. So setter injection is flexible than constructor injection.