

The author has made every effort in the preparation of this book to ensure the accuracy of the information. However, information in this book is sold without warranty either expressed or implied. The author will not be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Hibernate, Spring, Eclipse, HSQL Database & Maven tutorial

Hibernate is a very popular ORM (Object to Relational Mapping) tool and Spring is a very popular IOC (Inversion Of Control) container with support for AOP, Hibernate etc.

by

K. Arulkumaran

&

A. Sivayini

Website: <http://www.lulu.com/java-success>

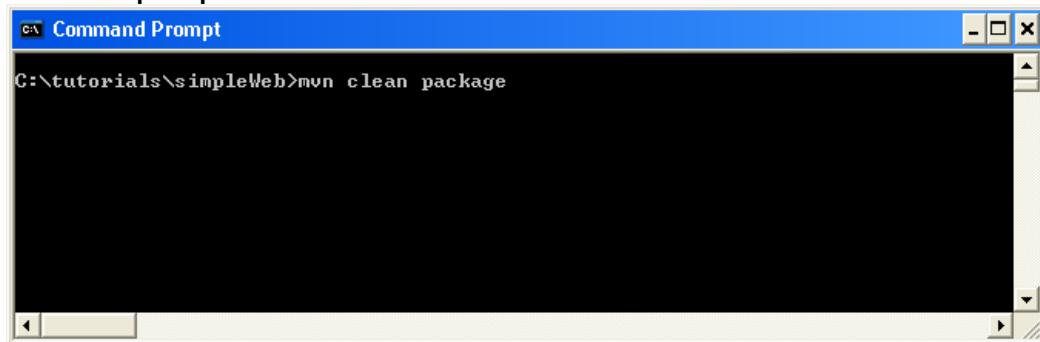
Feedback email: java-interview@hotmail.com

Table Of Contents

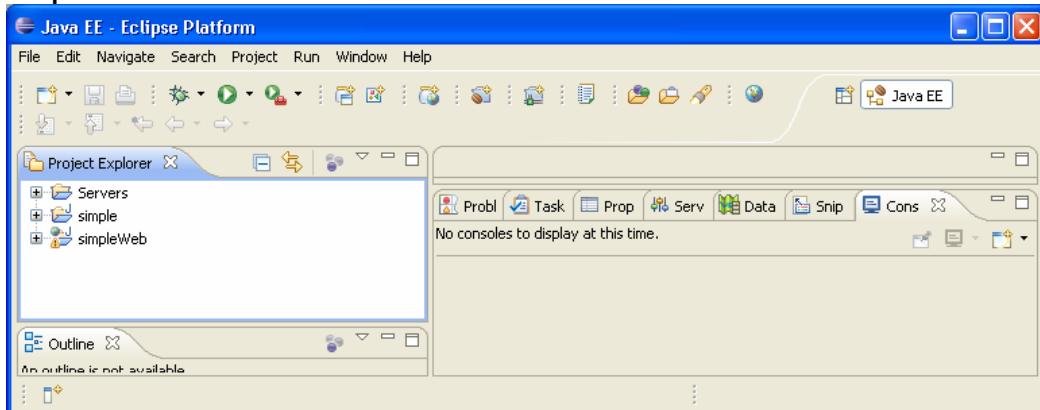
Notations	3
Tutorial 4 – Hibernate, HSQL Database, Maven and Eclipse	4
Tutorial 5 – Spring, Hibernate, Maven and Eclipse.....	20
Tutorial 6 – Spring AOP.....	31

Notations

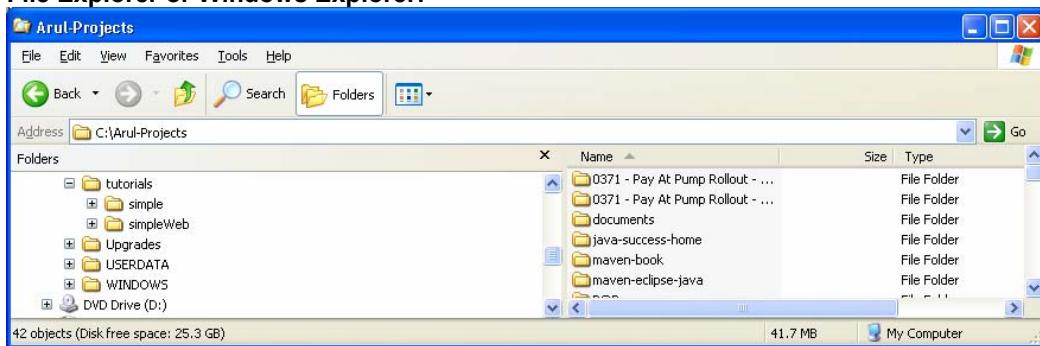
Command prompt:



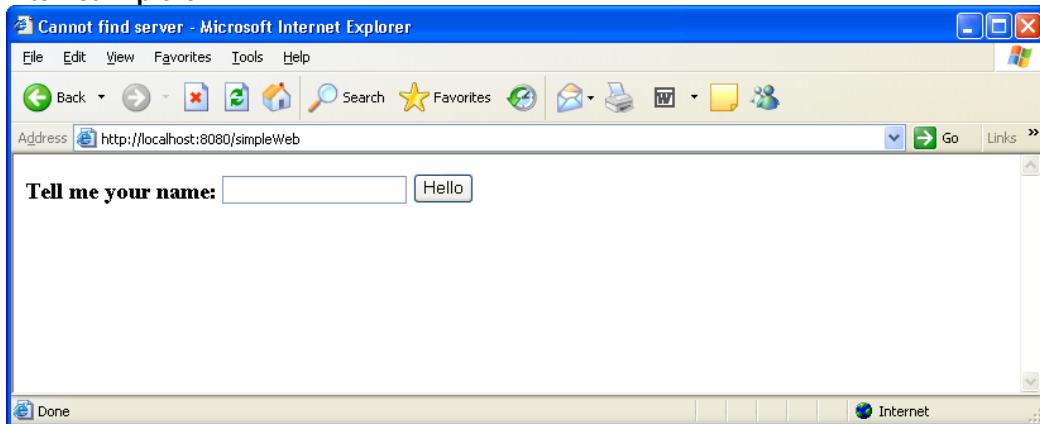
Eclipse:



File Explorer or Windows Explorer:



Internet Explorer:



Tutorial 4 – Hibernate, HSQL Database, Maven and Eclipse

This tutorial assumes that you are familiar with Java, Eclipse and Maven. If not please refer Tutorials 1-3 at <http://www.lulu.com/content/1080910>. This tutorial is a continuation of Tutorial 1 (**Java, Eclipse and Maven**).

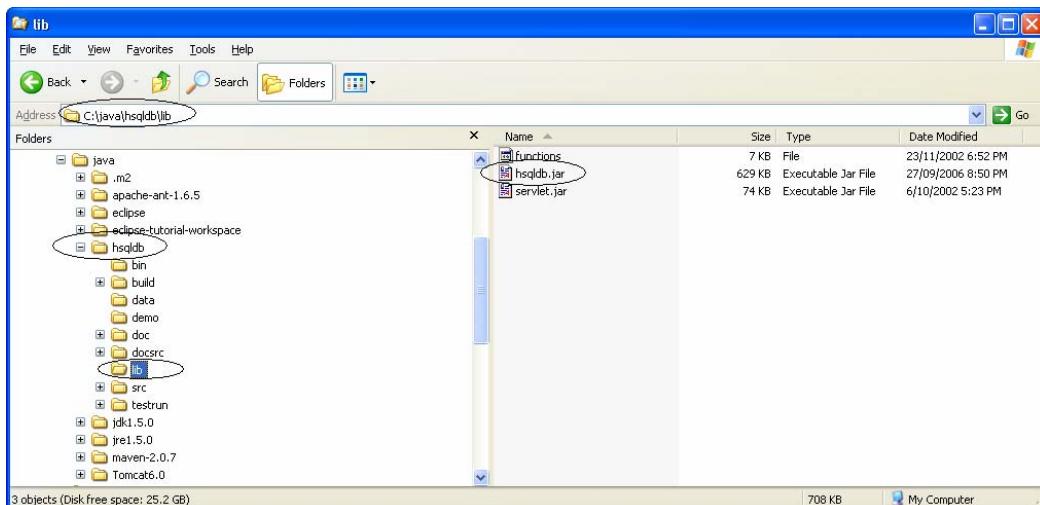
Hibernate is an ORM (Object to Relational Mapping) tool, so we need a relational database. To keep things simple, I will be using **HypersonicSQL** (aka **HSQL**) database, which is easy to use. This is an open source database, which can be found at <http://hsqldb.sourceforge.net/>. Also check <http://hsqldb.org/>.



The three types of persistent tables are **MEMORY** tables, **CACHED** tables and **TEXT** tables.

I will be using the default **MEMORY** tables where data is held entirely in memory but any change to their structure or contents is written to the `<dbname>.script` file. The script file is read the next time the database is opened, and the **MEMORY** tables are recreated with all their contents. So **MEMORY** tables are persistent. It is important to remember that the data in memory is written to the `<dbname>.script` file when you shutdown your database **properly/naturally** by executing SQL “**SHUTDOWN (COMPACT | IMMEDIATELY**” . The saved `<dbname>.script` file will load the data into memory the next time the HSQLDB server starts up. But if you stop the HSQLDB server **abruptly** in the command line by pressing [Ctrl] + [C] the data will not be written to the script file and consequently lost. Refer documentation for **CACHED & TEXT** tables.

- Install **HSQL** database into **c:\java** folder from <http://hsqldb.sourceforge.net/>.
Download the `hsqldb_1_8_0_7.zip` and unpack it into your **c:/java** folder.



- Start the HSQL database server by executing the following command in a command prompt as shown below:

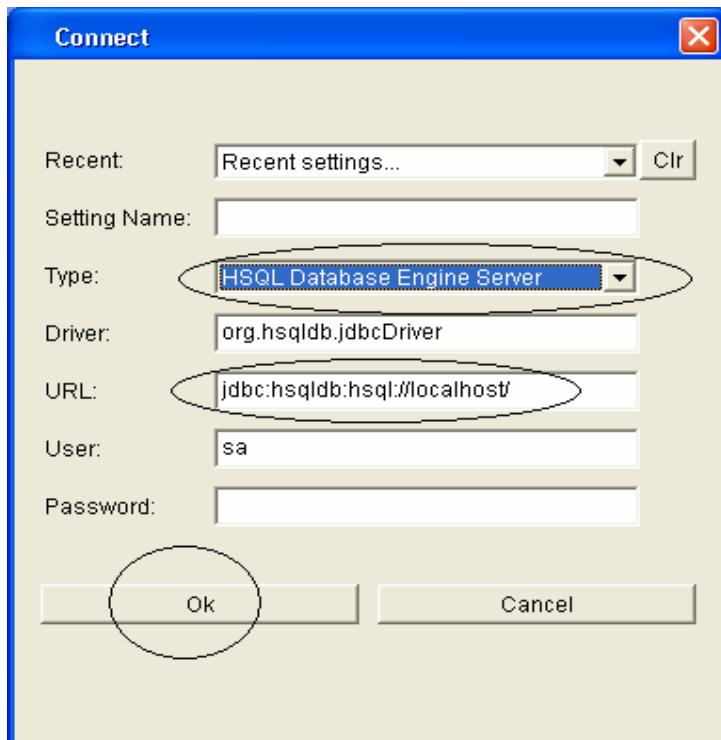
```
C:\java\hsqldb>java -cp ./lib/hsqldb.jar org.hsqldb.Server
[1] Command Prompt - java -cp ./lib/hsqldb.jar org.hsqldb.Server
C:\java\hsqldb>java -cp ./lib/hsqldb.jar org.hsqldb.Server
[Server@1a758cb]: [ThreadMain$main]: checkRunning(false) entered
[Server@1a758cb]: [ThreadMain$main]: checkRunning(false) exited
[Server@1a758cb]: Startup sequence initiated from main() method
[Server@1a758cb]: Loaded properties from [C:\java\hsqldb\server.properties]
[Server@1a758cb]: Initiating startup sequence...
[Server@1a758cb]: Server socket opened successfully in 0 ms.
[Server@1a758cb]: Database [index=0, id=0, db=file:test, alias=test] opened successfully in 406 ms.
[Server@1a758cb]: Startup sequence completed in 406 ms.
[Server@1a758cb]: 2007-08-07 10:59:49.193 HSQLDB server 1.8.0 is online
[Server@1a758cb]: To close normally, connect and execute SHUTDOWN SQL
[Server@1a758cb]: From command line, use [Ctrl]+[C] to abort abruptly
```

Since I have not given any database name and or alias (refer HSQLDB document and/or type C:\java\hsqldb>java -cp ./lib/hsqldb.jar org.hsqldb.Server -? **For more details**), it defaults to “**test**” as the database name & alias. After starting the HSQL database server the following files are created under “C:\java\hsqldb” → **test.lck**, **test.log**, **test.properties**.

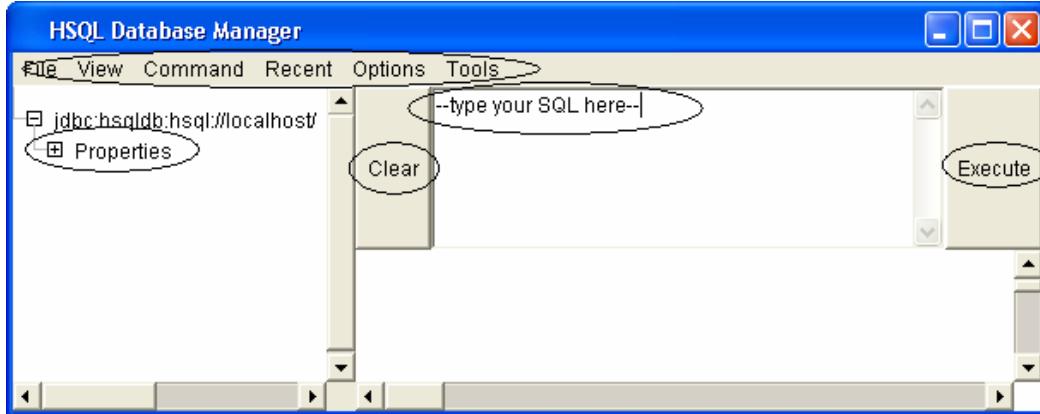
- Open up another command prompt and start the **DatabaseManager**, with which you can execute **SQLs**. If you are new to **SQL**, then this is handy to practice and gain some SQL skills. **Note:** You need to have the HSQLDB server running before you can open the **DatabaseManager**.

```
C:\java\hsqldb>java -cp ./lib/hsqldb.jar org.hsqldb.util.DatabaseManager
[1] Command Prompt - java -cp ./lib/hsqldb.jar org.hsqldb.util.DatabaseManager
C:\java\hsqldb>java -cp ./lib/hsqldb.jar org.hsqldb.util.DatabaseManager
```

This will spawn a new window as shown:



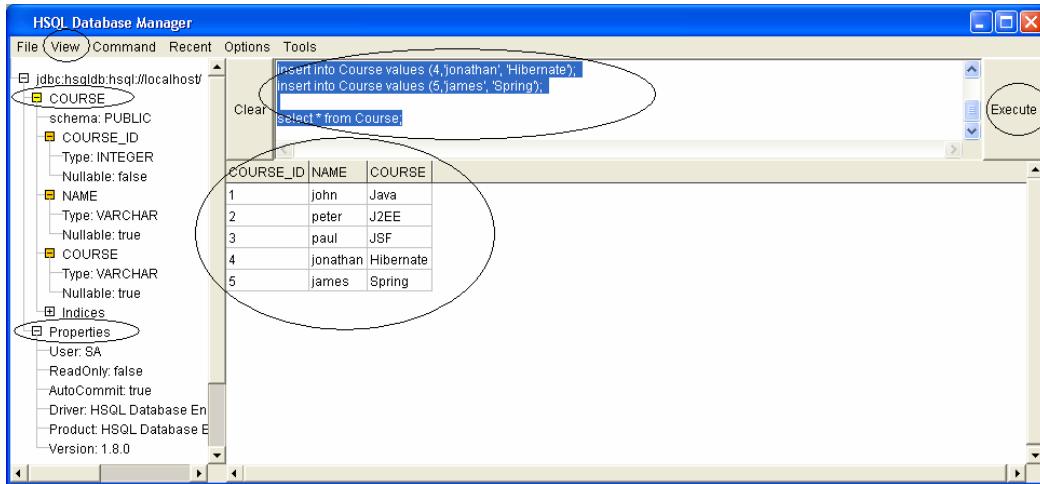
Select “**HSQL Database Engine Server**” and click “**Ok**”. Now you should have the following window opened, where you can type in your SQL and execute it by clicking on the “**Execute**” button. The results will be shown at the bottom. You can clear the “SQL” with the “**Clear**” button.



- Let's try executing the following SQL, which creates a table named “Course” and inserts some values and then select those values.

```
create table Course (course_id integer, name varchar, course varchar, PRIMARY KEY (course_id));  
  
insert into Course values (1,'Sam', 'Java');  
insert into Course values (2,'peter', 'J2EE');  
insert into Course values (3,'paul', 'JSF');  
insert into Course values (4,'jonathan', 'Hibernate');  
insert into Course values (5,'james', 'Spring');  
  
select * from Course;
```

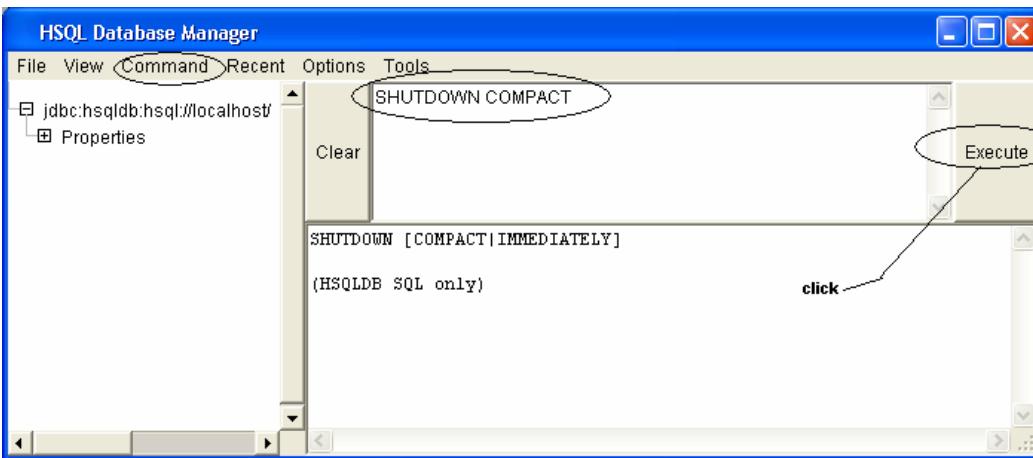
Copy and paste the above SQL into where it says --type your SQL here -- and press the “Execute” button to see the results as shown below:



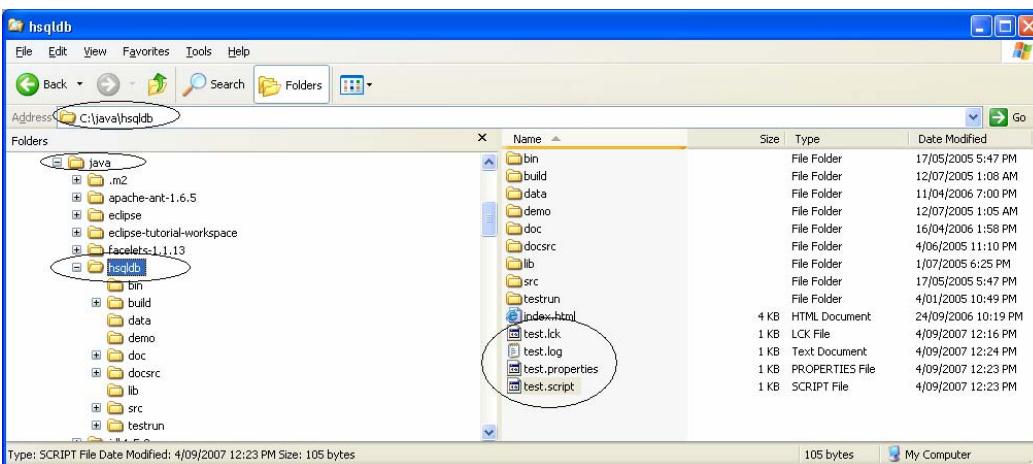
Note: After executing select **View → “Refresh Tree”** from the menu to see the “**COURSE**” on the left window. Also note the files generated under **c:\java\hsqldb**.

Note: The syntax for these files are <database>.properties, < database>.lck etc. Since we have not specified any database name, it defaults to "test". Refer HSQL documentation for starting the server with database name. e.g. **java -cp ./lib/hsqldb.jar org.hsqldb.Server -database.0 file:mydb -dbname.0 xdb**, where **xdb** is the alias.

Note: To persist your data from memory to “**test.script**”, you need to execute the SHUTDOWN SQL command as shown below.



Now you should have the "test.script" file created under "C:\java\hsqldb".



That's all to it on **HSQL**. Let's now move on to **Hibernate**. As said before this is the continuation of tutorial 1 at <http://www.lulu.com/content/1080910>. So you should have the java project "simple" under c:\tutorials.

You need to start the HSQLDB server again:

```
C:\java\hsqldb>java -cp ./lib/hsqldb.jar org.hsqldb.Server
```

Also open up the **DatabaseManager**:

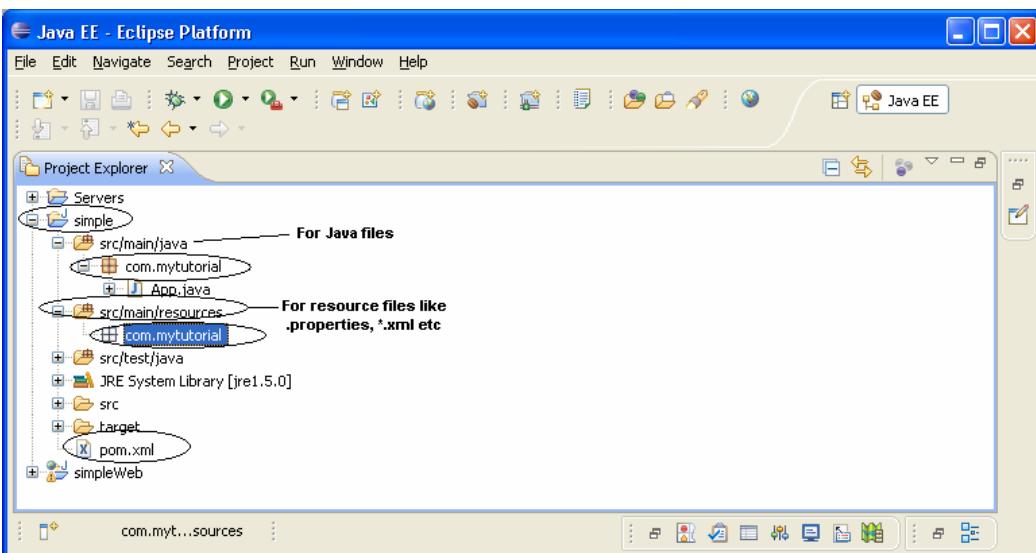
```
C:\java\hsqldb>java -cp ./lib/hsqldb.jar org.hsqldb.util.DatabaseManager
```

Firstly, we will set up the maven related **pom.xml** file configurations:

- Open your eclipse with the workspace "**C:\java\Tutorial-WorkSpace**".
- Create a "**resources**" folder under "**simple/src/main**"
- Run the following maven command from a new command prompt:

```
C:\tutorials\simple>mvn eclipse:clean eclipse:eclipse
```
- Refresh the "**simple**" project in eclipse. Now create a java package named "**com.mytutorial**" under "**simple/src/main/resources**".

Your eclipse workbench should look something like:



- Open the "**pom.xml**" file under "**simple**" to add dependencies like Hibernate and HSQLDB java driver. Add the following in bold. To [learn how to identify dependency library coordinates](#) refer [Tutorial 3 on "JSF, Maven and Eclipse"](#) from <http://www.lulu.com/content/1080910>.

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mytutorial</groupId>
  <artifactId>simple</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>simple</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
      <version>3.2.4.ga</version>
    </dependency>
    <dependency>
      <groupId>hsqldb</groupId>
      <artifactId>hsqldb</artifactId>
      <version>1.8.0.7</version>
    </dependency>
  </dependencies>

  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>2.0.2</version>
          <configuration>
            <source>1.5</source>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>

```

Hibernate library and its transitive dependencies. I.e. Maven will look at Hibernate's **.pom** file and bring in all its dependency jars. That's power of Maven.

HSQL database JDBC driver

In maven everything is done through plug-ins. This is the maven java compiler plugin. I am using **Java 5** (i.e. JDK 1.5)

```

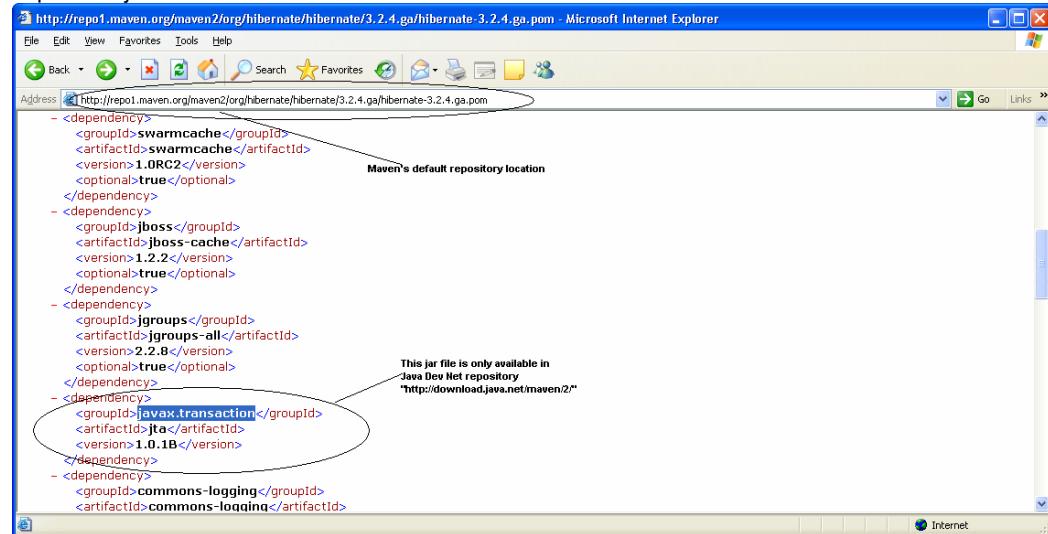
        <target>1.5</target>
    </configuration>
</plugin>
</plugins>
</pluginManagement>
</build>

<repositories>
    <repository>
        <id>maven-repository.dev.java.net</id>
        <name>Java Dev Net Repository</name>
        <url>http://download.java.net/maven/2/</url>
        <releases>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
        </releases>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
</repositories>

</project>

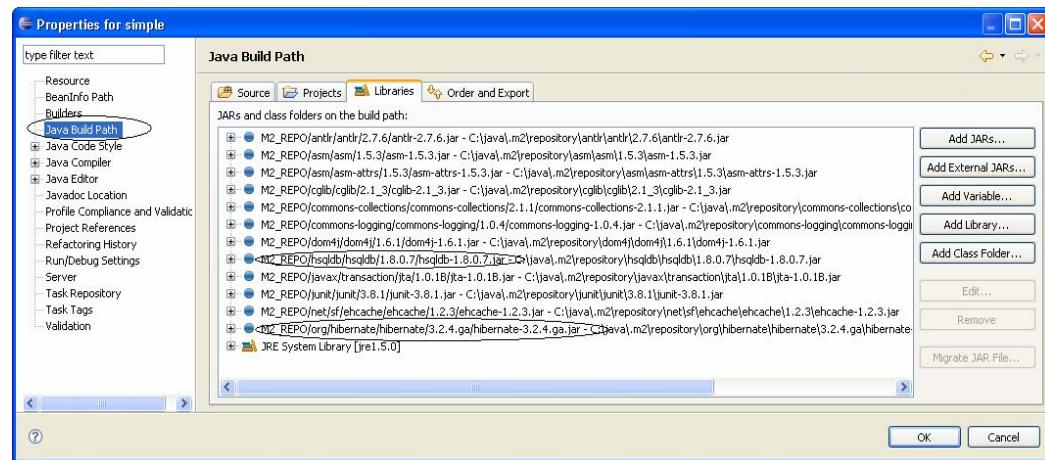
```

Note: You can add any number of repositories and Maven 2 will look through them one by one (including local repository c:\java\m2\repository & <http://repo1.maven.org/maven2/>) until the dependent jar file is found.

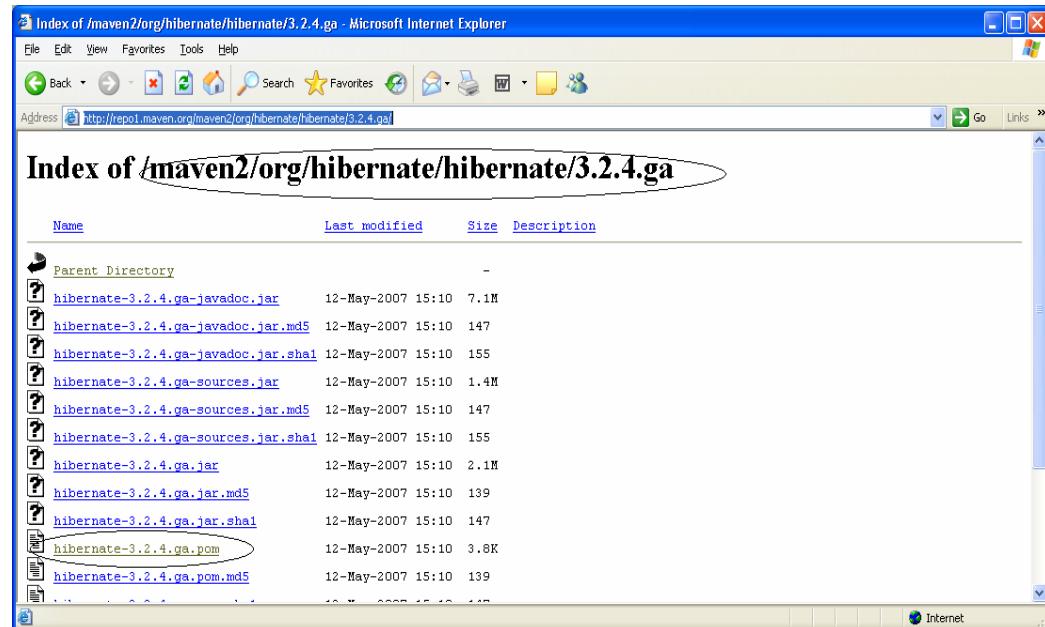


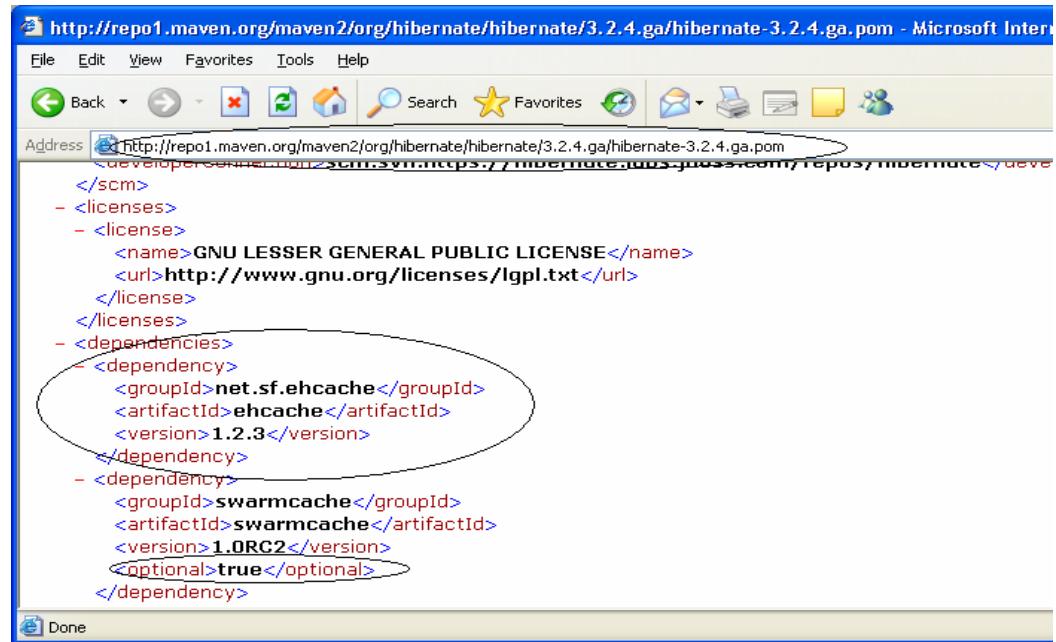
After adding the above dependencies remember to **save** the “**pom.xml**” and run the following command from the previously opened command prompt to construct the eclipse build path with dependency jars.

```
C:\tutorials\simple>mvn eclipse:clean eclipse:eclipse
```



You can open the properties window on “**simple**” project by right clicking and selecting “**properties**”. You can see the jar files for hibernate and its dependency jars like asm, cgilib, ehcache etc based on hibernate’s *.pom file where dependencies are defined. Open hibernate’s pom file shown below and check the dependencies. Also you can find the hsqldb java driver.





Next we will move on to creating some java classes and interfaces under “simple/src/main/java/com/mytutorial”.

- Let's create the java domain class “Course.java” under “simple/src/main/java/com/mytutorial”, which gets mapped to the table “Course”. For example the Course table looks like:

The screenshot shows the HSQL Database Manager interface. On the left, the database structure is shown with a tree view. A red oval highlights the 'COURSE' table node. The main pane displays the table structure with columns: COURSE_ID, NAME, and COURSE. Below is a data grid showing the following rows:

COURSE_ID	NAME	COURSE
1	john	Java
2	peter	J2EE
3	paul	JSF
4	jonathan	Hibernate

```

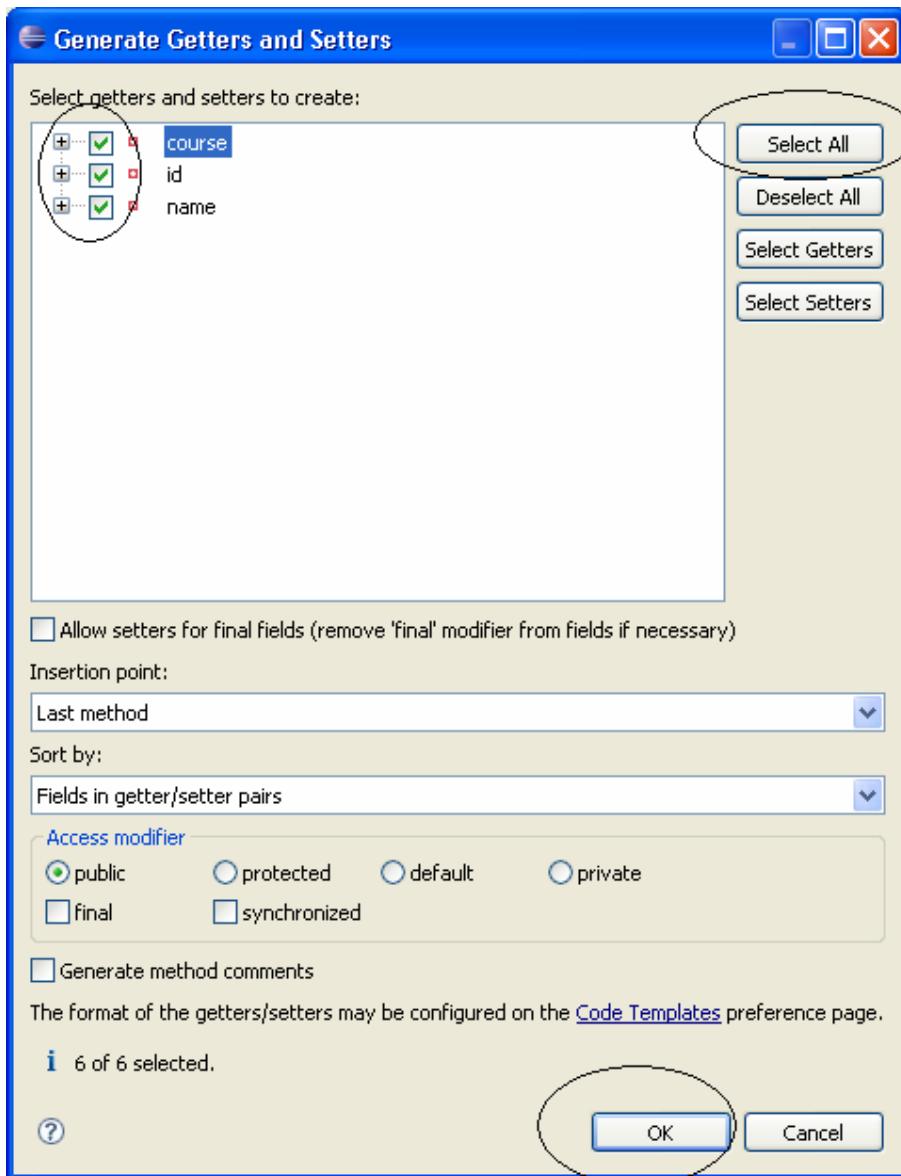
package com.mytutorial;

public class Course {

    private Long id;
    private String name;
    private String course;
}

```

Right click on “Course.java” and select “Source” and then “Generate Getters and Setters” to generate getters/setters.



The “Course.java” should look like

```
package com.mytutorial;

public class Course {

    private Long id;
    private String name;
    private String course;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getCourse() {
        return course;
    }
}
```

```

        public void setCourse(String course) {
            this.course = course;
        }
    }

Java EE - simple/src/main/java/com/mytutorial/Course.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Project Explorer Course.java
package com.mytutorial;

public class Course {

    private Long id;
    private String name;
    private String course;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getCourse() {
        return course;
    }
    public void setCourse(String course) {
        this.course = course;
    }
}

```

- No we need to create the Hibernate mapping file “**Course.hbm.xml**” under “**simple/src/main/resources/com/mytutorial**” to map the domain class “**Course.java**” to the HSQL database table “**Course**”.

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

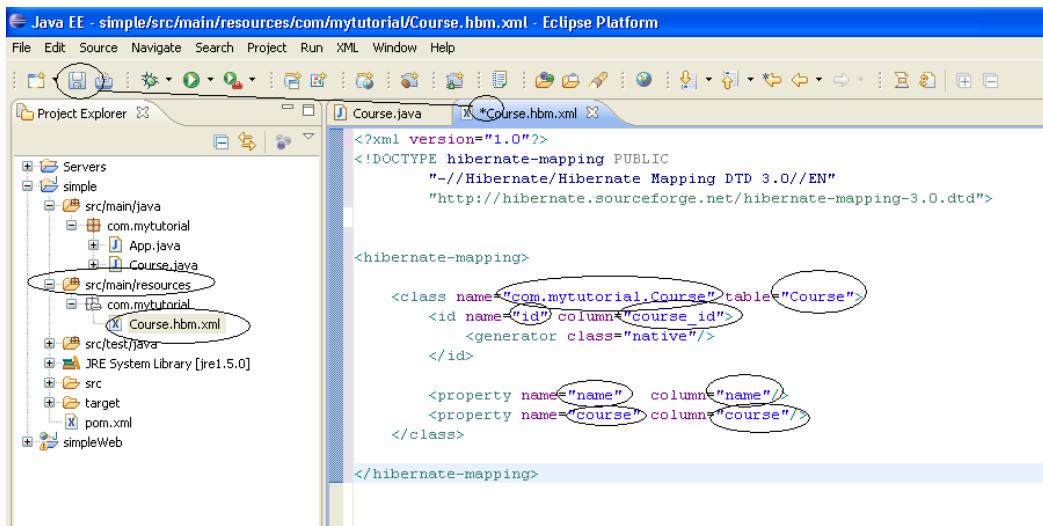
    <class name="com.mytutorial.Course" table="Course">
        <id name="id" column="course_id">
            <generator class="native"/>
        </id>

        <property name="name" column="name"/>
        <property name="course" column="course"/>
    </class>

</hibernate-mapping>

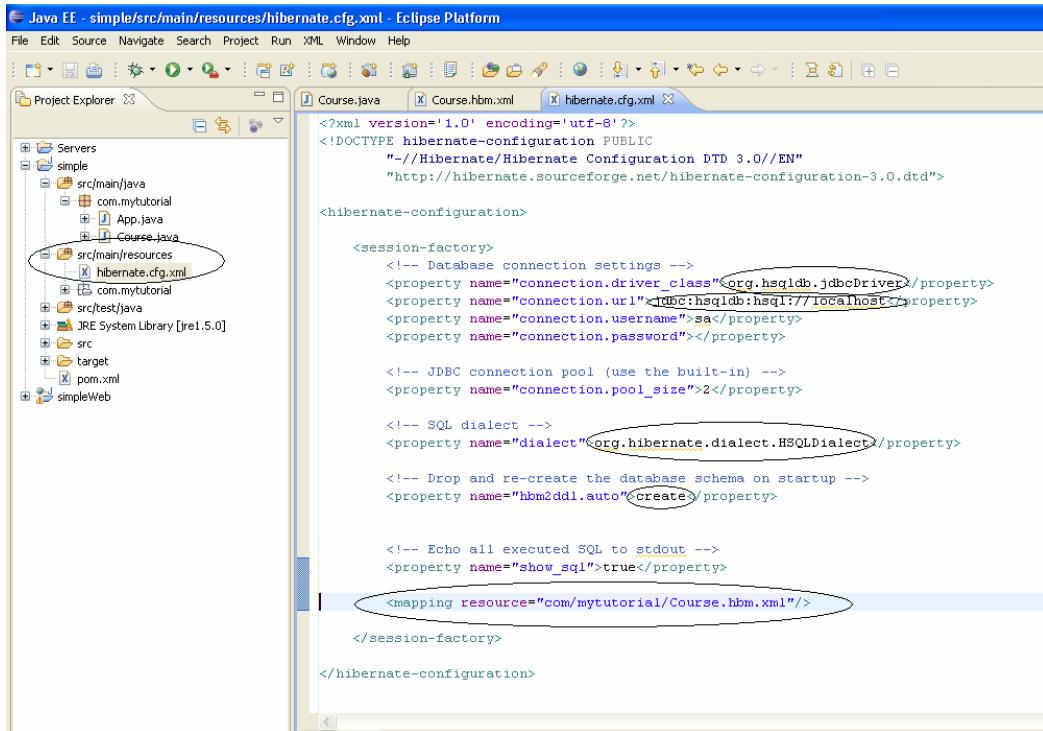
```

Mapping of domain class
“**Course.java**” to the HSQL
DB table “**Course**”



Remember to save the “Course.hbm.xml”.

- Next step is to create the Hibernate configuration file “**hibernate.cfg.xml**” under “**simple/src/main/resources**”. To configure the HSQL database connection details, dialect, bind the mapping file Course.hbm.xml etc via session factory.



```

<?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>
    <!-- Database connection settings -->
    <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
    <property name="connection.url">jdbc:hsqldb:hsqldb://localhost</property>
    <property name="connection.username">sa</property>
    <property name="connection.password"></property>
  </session-factory>
</hibernate-configuration>

```

```

<!-- JDBC connection pool (use the built-in) -->
<property name="connection.pool_size">2</property>

<!-- SQL dialect -->
<property name="dialect">org.hibernate.dialect.HSQLDialect</property>

<!-- Drop and re-create the database schema on start-up, also try with "update" to keep the
     previous values -->
<property name="hbm2ddl.auto">create</property>

<!-- Echo all executed SQL to stdout -->
<property name="show_sql">true</property>

<mapping resource="com/mytutorial/Course.hbm.xml"/>

</session-factory>

</hibernate-configuration>

```

- Next step is to create the Data Access Objects (DAO) and the Business Service classes/interfaces. Firstly create an interface **CourseDao.java** under “**simple/src/main/java/com/mytutorial**”.

```

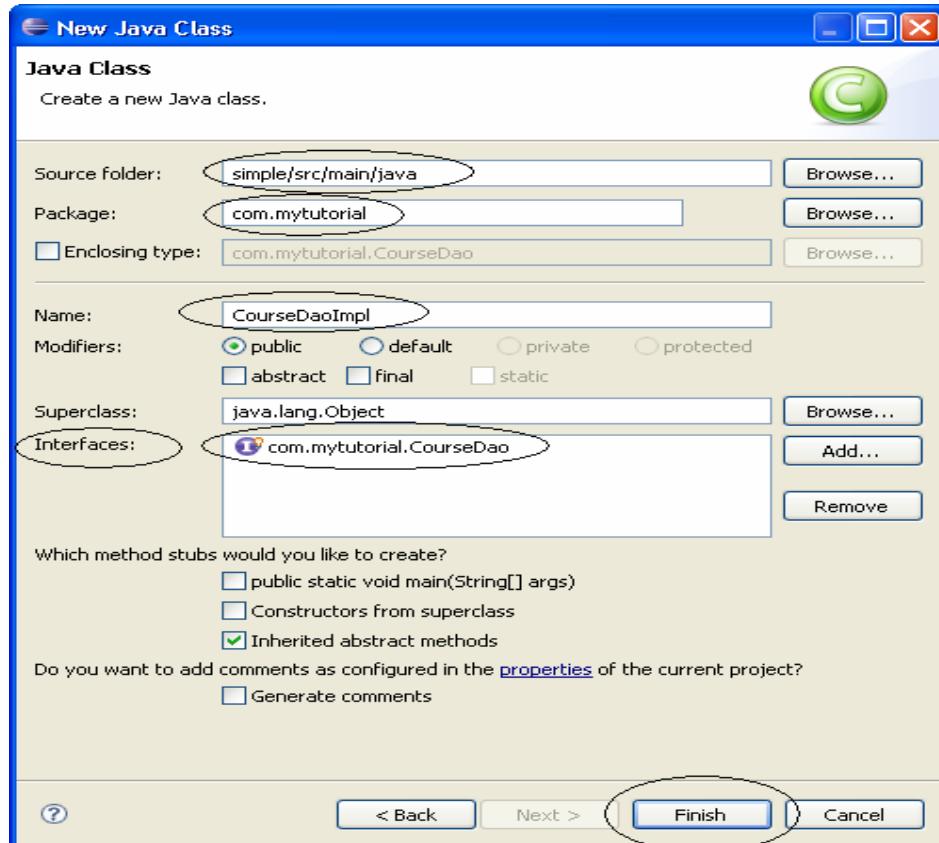
package com.mytutorial;

import java.util.List;

public interface CourseDao {
    public abstract void create(List<Course> listCourses);
    public abstract List findAll();
}

```

Now create the DAO implementation class **CourseDaoImpl.java** under “**simple/src/main/java/com/mytutorial**”.



```

package com.mytutorial;

import java.util.List;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class CourseDaoImpl implements {

    private static final SessionFactory sessionFactory;
    static {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            sessionFactory = new Configuration().configure()
                .buildSessionFactory();
        } catch (Throwable ex) {
            // Make sure you log the exception, as it might be swallowed
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public void create(List<Course> listCourses) {
        Session session = sessionFactory.openSession();
        session.beginTransaction();

        for (Course course : listCourses) {
            session.save(course);
        }
        session.getTransaction().commit();
    }

    public List findAll() {
        Session session = sessionFactory.openSession();
        List<Course> list = session.createQuery("From Course").list();
        return list;
    }
}

```

Tip: Some of the eclipse features you need to be aware of are like mouse right click on the code and then select “Source” → “Organize Imports” to create the import statements.

- Next step is to create the Business Service classes/interfaces under “simple/src/main/java/com/mytutorial”. Firstly create an interface named “CourseService.java”.

```

package com.mytutorial;

import java.util.List;

public interface CourseService {
    public abstract void processCourse(List<Course> courses);
}

```

Now, we can create the implementation class **CourseServiceImpl.java**.

Tip: Always code to interface not implementation. That is why we created an interface and an implementation class.

```

package com.mytutorial;

import java.util.List;

public class CourseServiceImpl implements CourseService {
    public void processCourse(List<Course> courses) {
        CourseDao dao = new CourseDaoImpl(); // tightly coupled
    }
}

```

```

        dao.create(courses);

        List<Course> list = dao.findAll();

        System.out.println("The saved courses are --> " + list);
    }
}

```

Finally modify our class which has the main method “**App.java**” under **simple/src/main/java/com/mytutorial**.

```

package com.mytutorial;

import java.util.ArrayList;
import java.util.List;

public class App {

    public static void main(String[] args) {

        List<Course> courses = new ArrayList<Course>(10);

        Course c1 = new Course();
        c1.setName("John");
        c1.setCourse("Java");

        courses.add(c1);

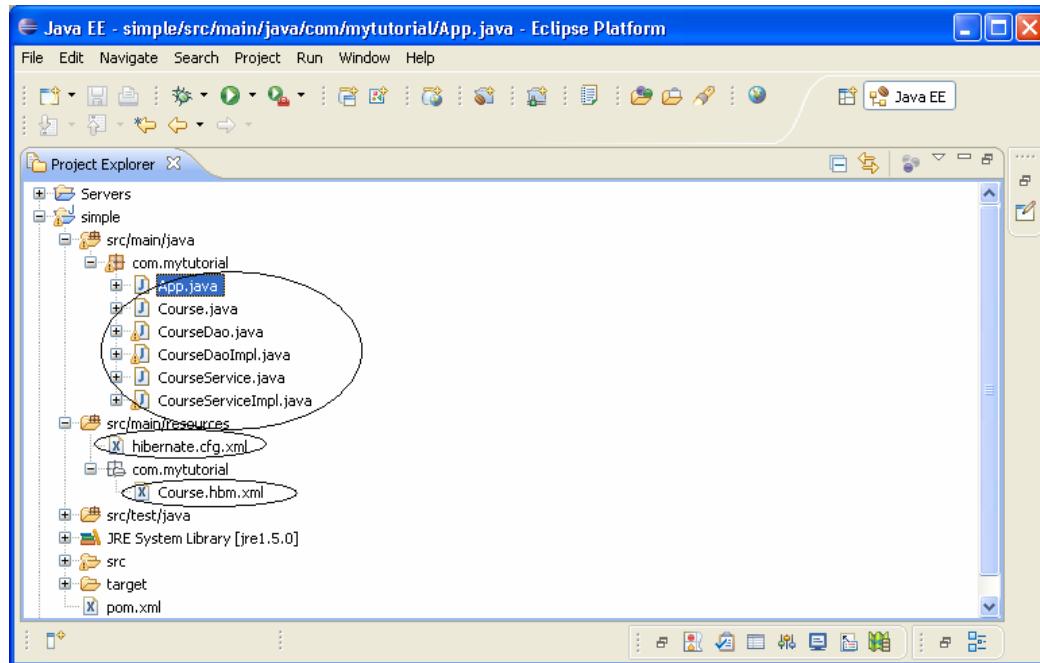
        Course c2 = new Course();
        c2.setName("Peter");
        c2.setCourse("Hibernate");

        courses.add(c2);

        CourseService service = new CourseServiceImpl(); // tightly coupled
        service.processCourse(courses);
    }
}

```

Now you should have all the source code required to run the **App.java**.



- Now run the **App.java** by right clicking and “Run As” → “Java Application”. You should get an output of:

```
Hibernate: insert into Course (name, course, course_id) values (?, ?, ?)
Hibernate: call identity()
Hibernate: insert into Course (name, course, course_id) values (?, ?, ?)
Hibernate: call identity()
Hibernate: select course0_.course_id as course1_0_, course0_.name as name0_, course0_.course
as course0_ from Course course0_
The saved courses are --> [com.mytutorial.Course@145c859, com.mytutorial.Course@64883c]
```

Tip: Why are we getting → com.mytutorial.Course@145c859? Because we do not have a `toString()` method in `Course.java`. Let's go and add a `toString()` method and try again.

```
@Override
public String toString() {
    return new StringBuffer().append("id=" + id).append(",name=" + name)
        .append(",course=" + course).toString();
}
```

The whole class should look like:

```
package com.mytutorial;

public class Course {

    private Long id;
    private String name;
    private String course;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

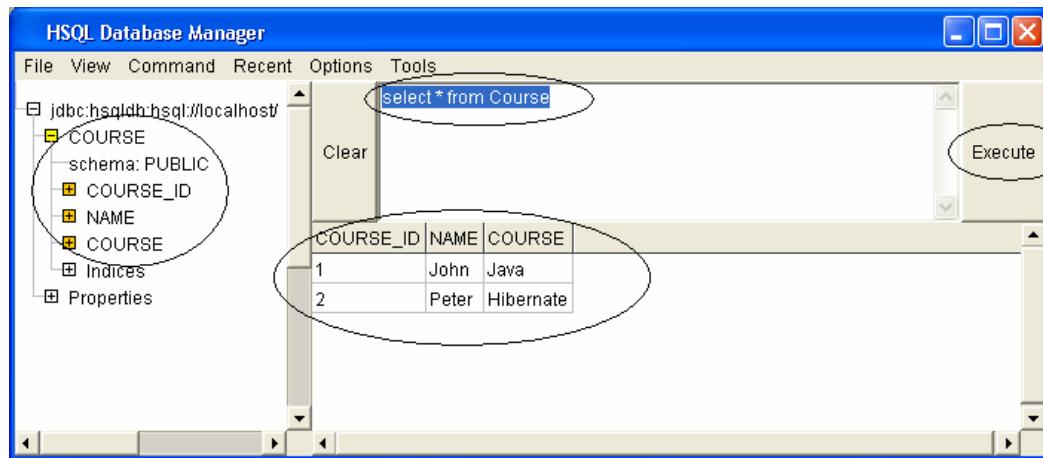
    public String getCourse() {
        return course;
    }

    public void setCourse(String course) {
        this.course = course;
    }

    @Override
    public String toString() {
        return new StringBuffer().append("id=" + id).append(",name=" + name)
            .append(",course=" + course).toString();
    }
}
```

Now run the `App.java` again and you should get an output:

```
Hibernate: insert into Course (name, course, course_id) values (?, ?, ?)
Hibernate: call identity()
Hibernate: insert into Course (name, course, course_id) values (?, ?, ?)
Hibernate: call identity()
Hibernate: select course0_.course_id as course1_0_, course0_.name as name0_, course0_.course
as course0_ from Course course0_
The saved courses are --> [id=1,name=John,course=Java,
id=2,name=Peter,course=Hibernate]
```



That's all to it.

You can find some fundamental Questions & Answers relating to Hibernate/Spring under "Emerging technologies/framework" section in **Java/J2EE Job Interview Companion**

<http://www.lulu.com/content/192463>

Please feel free to email any errors to java-interview@hotmail.com. Also stay tuned at <http://www.lulu.com/java-success> for more tutorials and Java/J2EE interview resources.

Tutorial 5 – Spring, Hibernate, Maven and Eclipse

It has been good so far that we followed the “**code to interface** not implementation” design principle. For example:

In **CourseServiceImpl.java**:

```
CourseDao dao = new CourseDaoImpl(); // tightly coupled
```

In **App.java**:

```
CourseService service = new CourseServiceImpl(); // tightly coupled
```

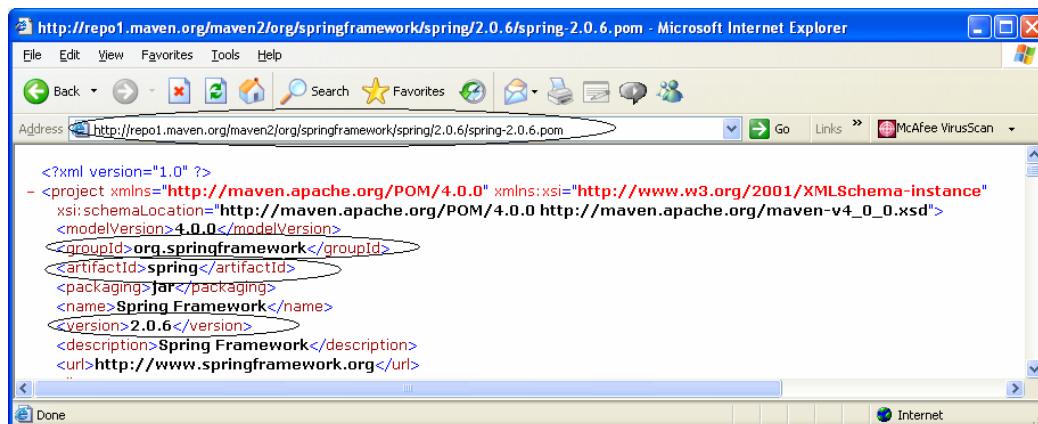
Why it is tightly coupled? If you have above code snippet scattered across a number of places throughout your code and if you want to change the implementation class for example “**CourseServiceImpl.java**” to say “**AdvancedCourseServiceImpl.java**” then you need to change your code in number of places with the following code.

```
CourseService service = new AdvancedCourseServiceImpl(); // tightly coupled
```

Q. How to improve on this by more loosely coupling? One way to do this is to introduce the “factory design pattern” where your implementation gets assigned inside a factory class. So, if you need to change the implementation, you just change it inside the factory class. With this approach you may end up having so many factory classes in your whole application. Alternatively we can use “**Dependency Injection**” (DI) using an “**Inversion Of Control**” (IOC) container like **Spring**.

In this tutorial, we will Spring enable the code we generated in **Tutorial 4**. So this is a continuation of Tutorial 4.

- Firstly add the Spring framework dependency to the **pom.xml** file under the project “**simple**” by looking up the coordinates at maven repository.



After adding the **Spring** dependency, your **pom.xml** file should look like:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mytutorial</groupId>
  <artifactId>simple</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>simple</name>
  <url>http://maven.apache.org</url>
  <dependencies>
```

```

<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifactId>
    <version>3.2.4.ga</version>
</dependency>

<dependency>
    <groupId>hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <version>1.8.0.7</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring</artifactId>
    <version>2.0.6</version>
</dependency>

</dependencies>

<build>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.0.2</version>
                <configuration>
                    <source>1.5</source>
                    <target>1.5</target>
                </configuration>
            </plugin>
        </plugins>
    </pluginManagement>
</build>

<repositories>
    <repository>
        <id>maven-repository.dev.java.net</id>
        <name>Java Dev Net Repository</name>
        <url>http://download.java.net/maven/2/</url>
        <releases>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
        </releases>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
</repositories>

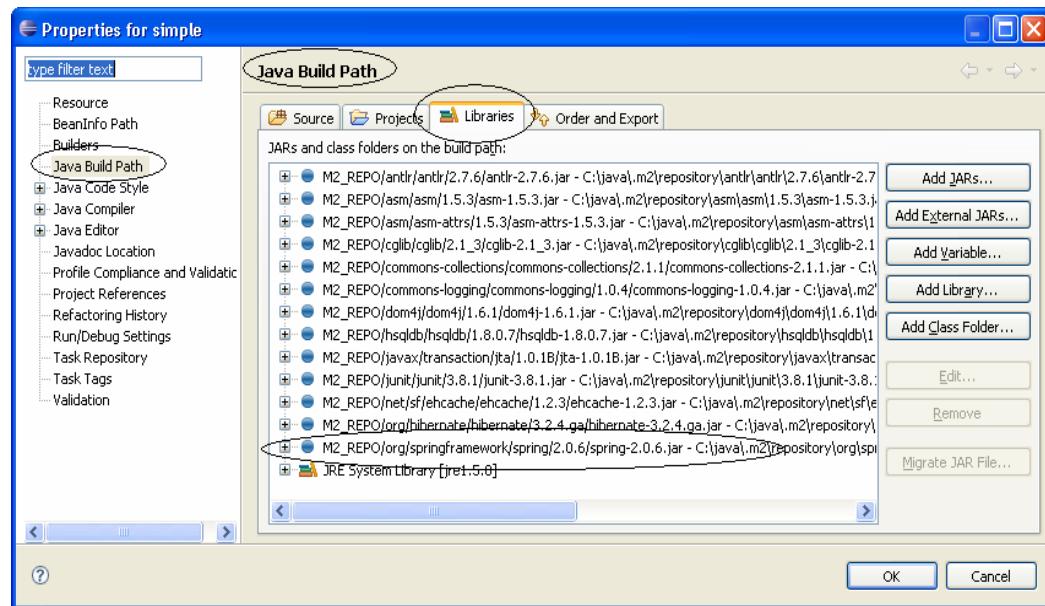
</project>

```

After saving this file, run the following maven command in a command prompt.

```
C:\tutorials\simple>mvn eclipse:clean eclipse:eclipse
```

After executing the above command, if you go back to your eclipse and refresh the project “**simple**” and check the eclipse build path dependency and should look like shown below with the **Spring** jar added:



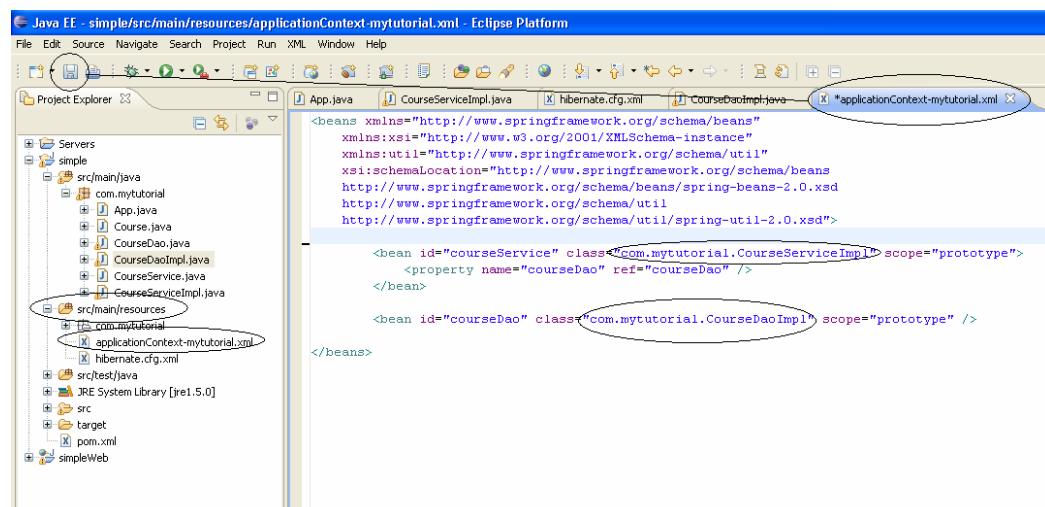
- Next step is to write Spring configuration file, which wires up your beans. This file is named “**applicationContext-mytutorial.xml**” under “**simple/src/main/resources**”.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                           http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util-2.0.xsd">

    <bean id="courseService" class="com.mytutorial.CourseServiceImpl" scope="prototype">
        <property name="courseDao" ref="courseDao" />
    </bean>
    <bean id="courseDao" class="com.mytutorial.CourseDaoImpl" scope="prototype" />
</beans>
```

Attribute name in CourseServiceImpl.java

Setter injection of dao into service



- Next we need to add “**courseDao**” attribute and the corresponding getter/setter methods inside “**CourseServiceImpl.java**” so that the “**courseDao**” can be injected using the setter method.

```
package com.mytutorial;

import java.util.List;

public class CourseServiceImpl implements CourseService {

    private CourseDao courseDao; // attribute name in " applicationContext-mytutorial.xml"

    public CourseDao getCourseDao() {
        return courseDao;
    }

    public void setCourseDao(CourseDao courseDao) {
        this.courseDao = courseDao;
    }

    public void processCourse(List<Course> courses) {
        // CourseDao dao = new CourseDaoImpl();←
        courseDao.create(courses);

        List<Course> list = getCourseDao().findAll();
        System.out.println("The saved courses are --> " + list);
    }
}
```

Don't need this, Spring will inject this.

- Next step is to inject the “**courseService**” into the **App.java** class.

```
package com.mytutorial;

import java.util.ArrayList;
import java.util.List;

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

public class App {
    public static void main(String[] args) {
        ApplicationContext ctx =←
        new ClassPathXmlApplicationContext("applicationContext-mytutorial.xml");

        List<Course> courses = new ArrayList<Course>(10);

        Course c1 = new Course();
        c1.setName("John");
        c1.setCourse("Java");

        courses.add(c1);

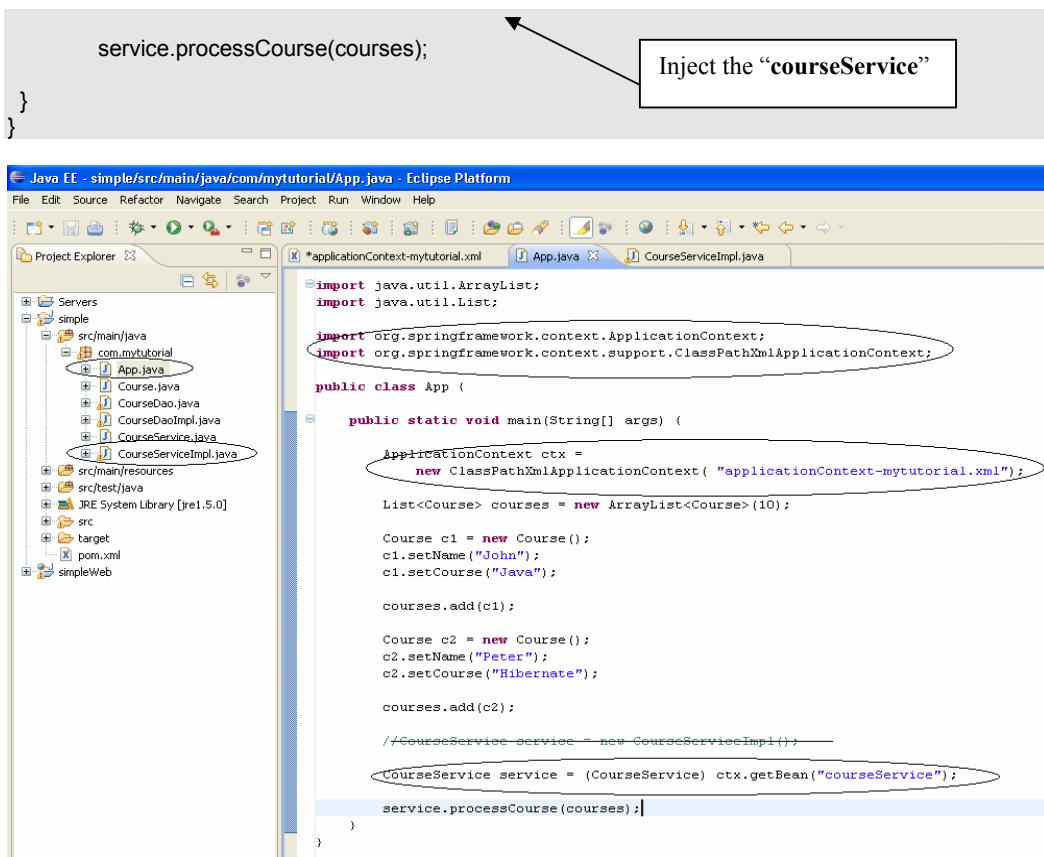
        Course c2 = new Course();
        c2.setName("Peter");
        c2.setCourse("Hibernate");

        courses.add(c2);

        // CourseService service = new CourseServiceImpl();←
        CourseService service = (CourseService) ctx.getBean("courseService");
    }
}
```

Load the application context file.

Don't need this, Spring will inject this.



- Now run the **App.java** and you should get the same output results as before in Tutorial 4. This time with Spring more loosely coupling our classes.

```

Hibernate: insert into Course (name, course, course_id) values (?, ?, ?)
Hibernate: call identity()
Hibernate: insert into Course (name, course, course_id) values (?, ?, ?)
Hibernate: call identity()
Hibernate: select course0_.course_id as course1_0_, course0_.name as name0_, course0_.course as course0_
from Course course0_
The saved courses are --> [id=1,name=John,course=Java, id=2,name=Peter,course=Hibernate]

```

- Spring provides support for Hibernate to improve your code quality . We could also inject the **sessionFactory** using Spring as shown below:

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                           http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util-2.0.xsd">

    <bean id="courseService" class="com.mytutorial.CourseServiceImpl" scope="prototype">
        <property name="courseDao" ref="courseDao" />
    </bean>

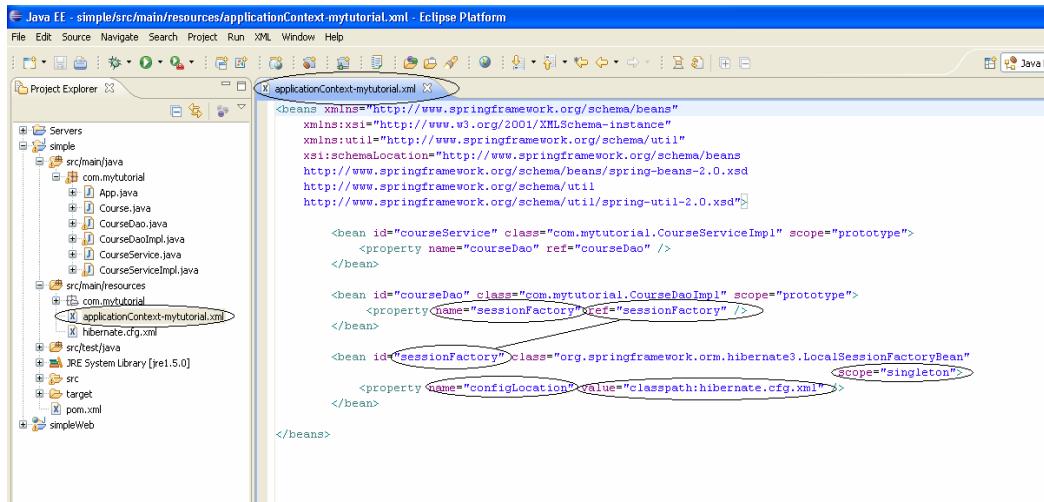
    <bean id="courseDao" class="com.mytutorial.CourseDaoImpl" scope="prototype">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean"
          scope="singleton">
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />
    
```

Attribute in
CourseServiceImpl.java

```
</bean>

</beans>
```



Now we need to provide the member variable “**sessionFactory**” and its setter/getter methods in “**CourseDaoImpl.java**” as shown below.

```
package com.mytutorial;

import java.util.List;

import org.hibernate.Session;
import org.hibernate.SessionFactory;

public class CourseDaoImpl implements CourseDao {

    private SessionFactory sessionFactory = null;

    /**
     * 
     private static final SessionFactory sessionFactory;
     static {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            sessionFactory = new Configuration().configure()
                .buildSessionFactory();
        } catch (Throwable ex) {
            // Make sure you log the exception, as it might be swallowed
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }
    **/


    public void create(List<Course> listCourses) {
        Session session = sessionFactory.openSession();
        session.beginTransaction();

        for (Course course : listCourses) {
            session.save(course);
        }
        session.getTransaction().commit();
    }

    public List findAll() {
        Session session = sessionFactory.openSession();
        List<Course> list = session.createQuery("From Course").list();
        return list;
    }
}
```

Not required, because now done through Spring.

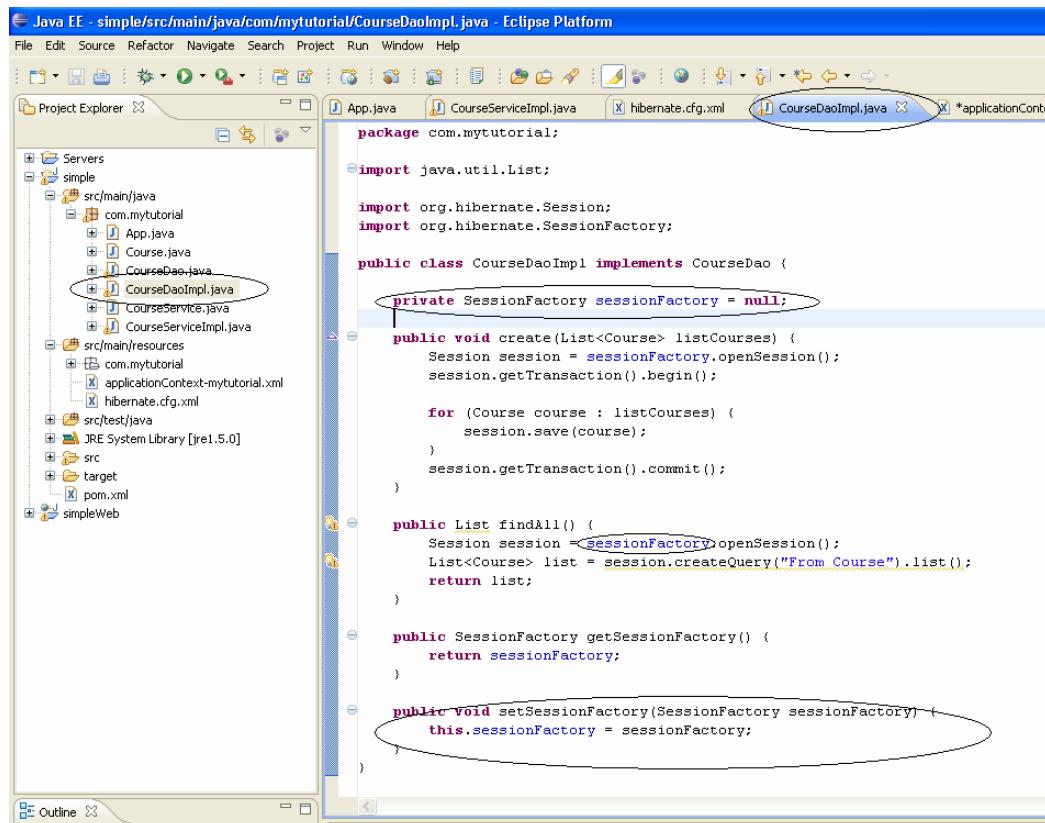
```

public SessionFactory getSessionFactory() {
    return sessionFactory;
}

public void setSessionFactory(SessionFactory sessionFactory) {
    this.sessionFactory = sessionFactory;
}
}

```

Spring will inject invoking this method



- Spring provides template support (e.g. JdbcTemplate, HibernateTemplate, JmsTempalte etc) to minimise the amount of code you have to write. Let's look at a simplified example using "HibernateTemplateSupport" in "CourseDaoImpl.java"

```

package com.mytutorial;

import java.util.List;
import org.hibernate.SessionFactory;
import org.springframework.orm.hibernate3.HibernateTemplate;

public class CourseDaoImpl implements CourseDao {

    private SessionFactory sessionFactory = null;

    public void create(List<Course> listCourses) {
        HibernateTemplate ht = new HibernateTemplate(sessionFactory);

        for (Course course : listCourses) {
            ht.save(course);
        }
    }

    public List findAll() {
        HibernateTemplate ht = new HibernateTemplate(sessionFactory);
        return ht.find("From Course");
    }
}

```

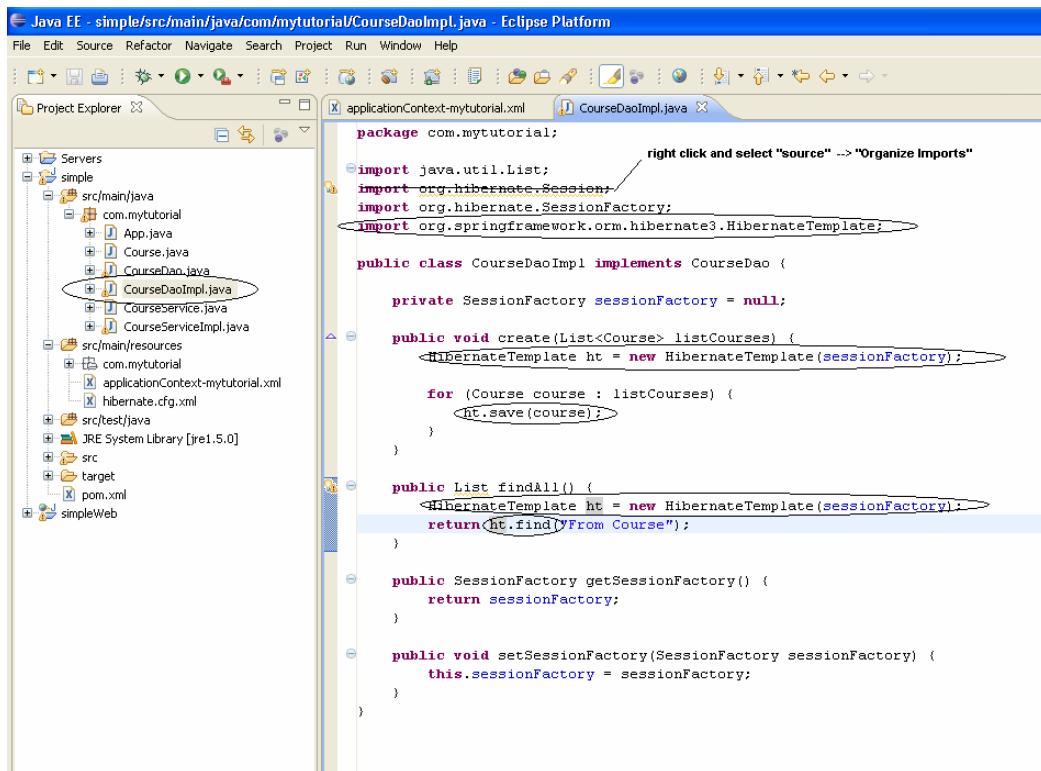
Less code with "HibernateTemplate"

```

public SessionFactory getSessionFactory() {
    return sessionFactory;
}

public void setSessionFactory(SessionFactory sessionFactory) {
    this.sessionFactory = sessionFactory;
}
}

```



- Try running the **App.java** again and you should get the same results with a more simplified (i.e. less) code. But your data will not be committed to the database because we have not added the Transaction Support (By default “autoCommit” is set to false). Let's add the declarative transaction support to the service layer i.e. the **CourseService**. Unlike in the Tutorial-4 where transaction was done via code like “session.getTransaction().begin();”. We need to make the following changes to enable declarative transaction demarcation via Spring.

applicationContext-mytutorial.xml

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                           http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util-2.0.xsd">

    <bean id="courseService" parent="txProxyTemplate" >
        <property name="target">
            <bean class="com.mytutorial.CourseServiceImpl" scope="prototype">
                <property name="courseDao" ref="courseDao" />
            </bean>
        </property>
    </bean>

```

```

<bean id="courseDao" class="com.mytutorial.CourseDaoImpl" scope="prototype">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>

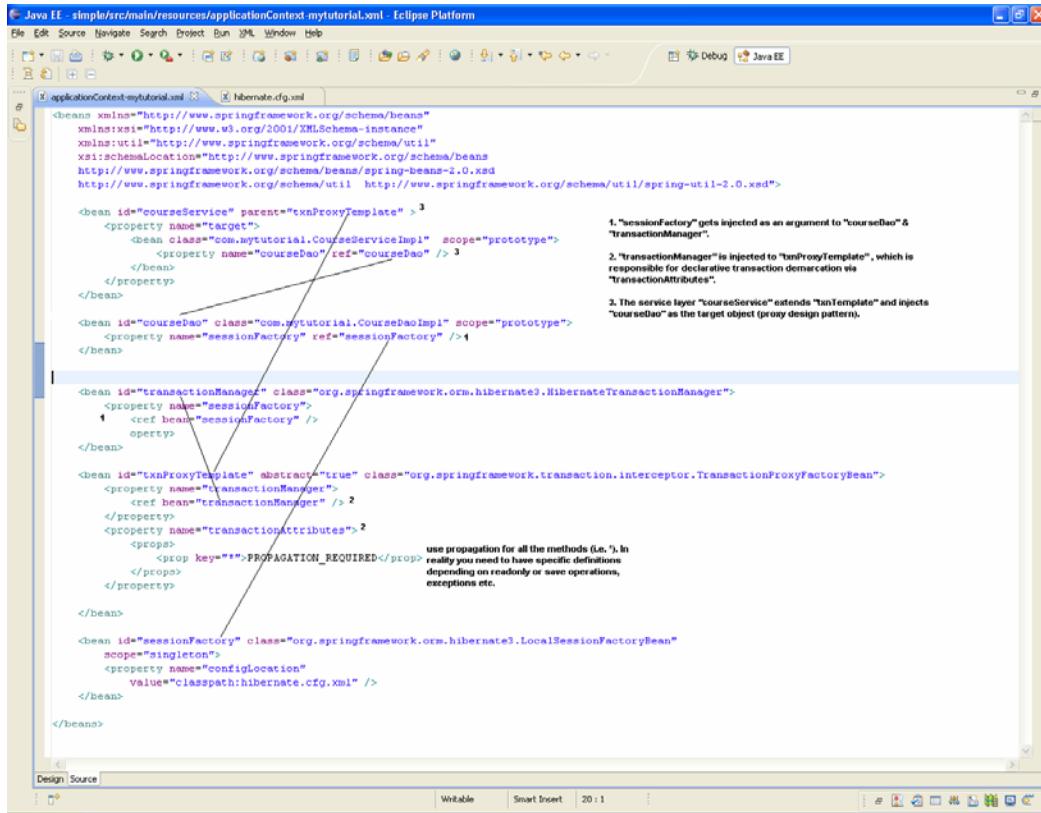
<bean id="transactionManager"
    class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory">
        <ref bean="sessionFactory" />
    </property>
</bean>

<bean id="txProxyTemplate" abstract="true"
    class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager">
        <ref bean="transactionManager" />
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="*">>PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>

<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean"
    scope="singleton">
    <property name="configLocation" value="classpath:hibernate.cfg.xml" />
</bean>

</beans>

```



CourseDaoImpl.java

```

package com.mytutorial;

import java.util.List;

import org.springframework.orm.hibernate3.HibernateTemplate;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;

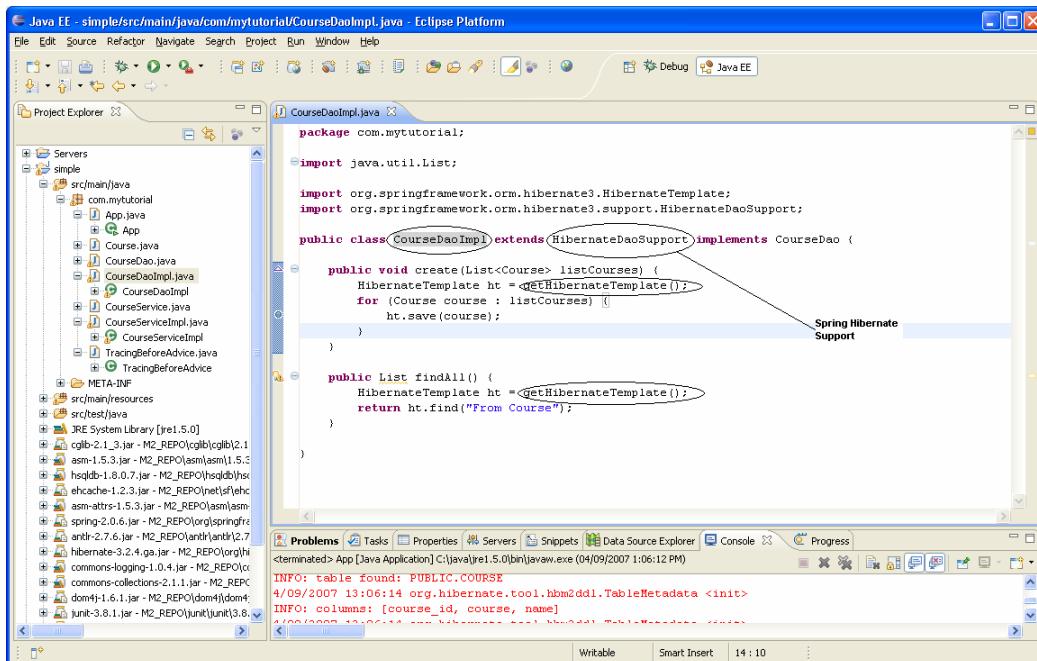
public class CourseDaoImpl extends HibernateDaoSupport implements
    CourseDao {

    public void create(List<Course> listCourses) {
        HibernateTemplate ht = getHibernateTemplate();
        for (Course course : listCourses) {
            ht.save(course);
        }
    }

    public List findAll() {
        HibernateTemplate ht = getHibernateTemplate();
        return ht.find("From Course");
    }

}

```



CourseServiceImpl.java

```

package com.mytutorial;

import java.util.List;

public class CourseServiceImpl implements CourseService {

    private CourseDao courseDao;

```

```

public CourseDao getCourseDao() {
    return courseDao;
}

public void setCourseDao(CourseDao courseDao) {
    this.courseDao = courseDao;
}

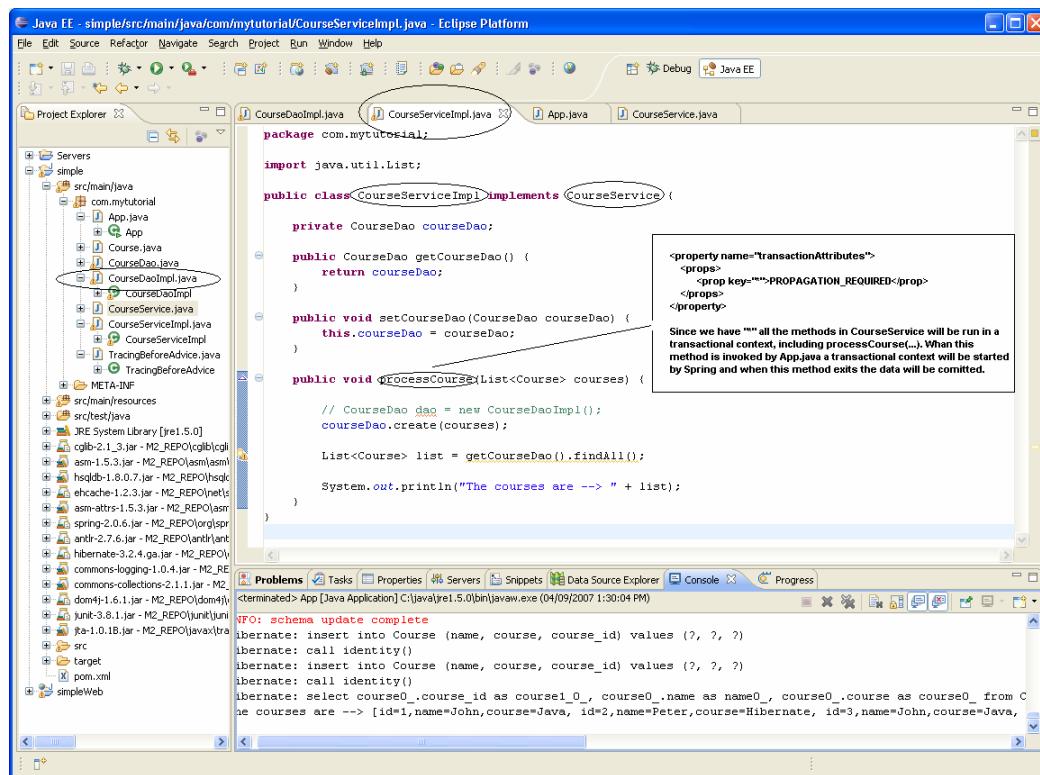
public void processCourse(List<Course> courses) {

    // CourseDao dao = new CourseDaoImpl();
    courseDao.create(courses);

    List<Course> list = getCourseDao().findAll();

    System.out.println("The courses are --> " + list);
}
}

```



Now try running the `App.java` again and the data should be committed into the database and should be able to query the values using the `DatabaseManager`.

Note: Spring uses **AOP** for its transaction demarcation. Let's look at a simple example in the next tutorial how to use **Spring AOP** in your application.

Tutorial 6 – Spring AOP

Finally let's look at Spring's support for **Aspect Oriented Programming** (AOP). We will do a simplified example. Firstly create an “**Advice**” to print “**Just before method call...**” before a method executes for all our service classes (e.g. **CourseServiceImpl.java**) with the method starting with **processXXXXX**.

- Create an Advice named “**TracingBeforeAdvice.java**” that gets executed before a method call.

```
package com.mytutorial;

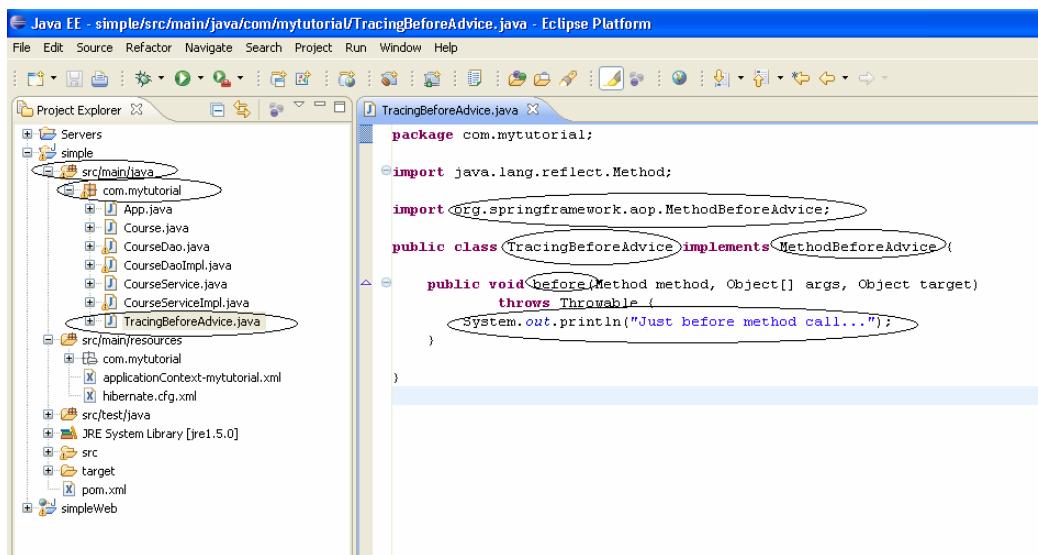
import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;

public class TracingBeforeAdvice implements MethodBeforeAdvice {

    public void before(Method method, Object[] args, Object target)
        throws Throwable {
        System.out.println("Just before method call...");
    }

}
```



- Now we need to wire up this advice via “**applicationContext-mytutorial.xml**”.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                           http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util-2.0.xsd">

    <bean id="courseService" parent="txProxyTemplate" >
        <property name="target">
            <bean class="com.mytutorial.CourseServiceImpl" scope="prototype">
                <property name="courseDao" ref="courseDao" />
            </bean>
        </property>

        <property name="preInterceptors">
            <list>
```

```

        <ref bean="traceBeforeAdvisor" />
    </list>
</property>
</bean>

<bean id="courseDao" class="com.mytutorial.CourseDaoImpl" scope="prototype">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>

<bean id="transactionManager"
      class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory">
        <ref bean="sessionFactory" />
    </property>
</bean>

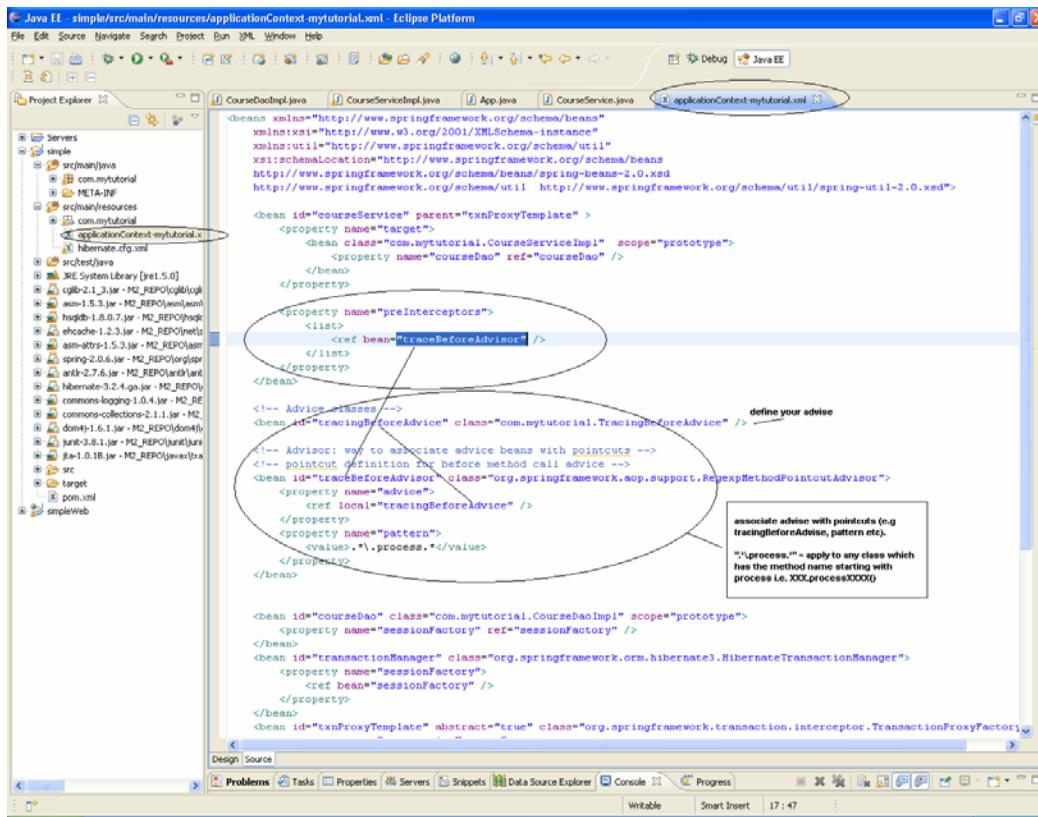
<bean id="txProxyTemplate" abstract="true"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager">
        <ref bean="transactionManager" />
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="*">>PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>

<bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean"
      scope="singleton">
    <property name="configLocation"
              value="classpath:hibernate.cfg.xml" />
</bean>

<!-- Advice classes -->
<bean id="tracingBeforeAdvice" class="com.mytutorial.TracingBeforeAdvice" />

<!-- Advisor: way to associate advice beans with pointcuts -->
<!-- pointcut definition for before method call advice -->
<bean id="traceBeforeAdvisor"
      class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
        <ref local="tracingBeforeAdvice" />
    </property>
    <property name="pattern">
        <value>.*\process.*</value>
    </property>
</bean>

</beans>
```



- Now, run the **App.java** and you should get the output as follows:

Just before method call...

Hibernate: insert into Course (name, course, course_id) values (?, ?, ?)

Hibernate: call identity()

Hibernate: insert into Course (name, course, course_id) values (?, ?, ?)

Hibernate: call identity()

Hibernate: select course0_.course_id as course1_0_, course0_.name as name0_, course0_.course as course0_ from Course course0_

The saved courses are --> [id=1,name=John,course=Java, id=2,name=Peter,course=Hibernate]

That's all to it.

You can find some fundamental Questions & Answers relating to Hibernate/Spring under "Emerging technologies/framework" section in **Java/J2EE Job Interview Companion** at <http://www.lulu.com/content/192463>

Please feel free to email any errors to java-interview@hotmail.com. Also stay tuned at <http://www.lulu.com/java-success> for more tutorials and Java/J2EE interview resources.