Chapter 5. Implementing Database Persistence

This is Part 5 of a step-by-step account of how to develop a web application from scratch using the Spring Framework. In Part 1 we configured the environment and set up a basic application. In Part 2 we refined the application that we will build upon. Part 3 added all the business logic and unit tests and Part 4 developed the web interface. It is now time to introduce database persistence. We saw in the earlier parts how we loaded some business objects using bean definitions in a configuration file. It is obvious that this would never work in real life – whenever we re-start the server we are back to the original prices. We need to add code to actually persist these changes to a database.

5.1. Create database startup script

Before we can start developing the persistence code, we need a database. We are planning on using HSQL, which is a good open source database written in Java. This database is distributed with Spring, so we can just copy the jar file to the web application's lib directory. Copy **hsqldb.jar** from the **'spring-framework-2.5/lib/hsqldb'** directory to the **'springapp/war/WEB-INF/lib'** directory. We will use HSQL in standalone server mode. That means we will have to start up a separate database server instead of relying on an embedded database, but it gives us easier access to see changes made to the database when running the web application.

We need a script or batch file to start the database. Create a **'db'** directory under the main **'springapp'** directory. This new directory will contain the database files. Now, let's and add a startup script:

For Linux/Mac OS X add:

'springapp/db/server.sh':

```
java -classpath ../war/WEB-INF/lib/hsqldb.jar org.hsqldb.Server -database test
```

Don't forget to change the execute permission by running **'chmod +x server.sh'**.

For Windows add:

'springapp/db/server.bat':

```
java -classpath ..\war\WEB-INF\lib\hsqldb.jar org.hsqldb.Server -database test
```

Now you can open a command window, change to the **'springapp/db'** directory and start the database by running one of these startup scripts.

5.2. Create table and test data scripts

First, lets review the SQL statement needed to create the table. We create the file 'create_products.sql' in the db directory.

'springapp/db/create_products.sql':

```
CREATE TABLE products (
  id INTEGER NOT NULL PRIMARY KEY,
  description varchar(255),
  price decimal(15,2)
);
CREATE INDEX products_description ON products(description);
```

Now we need to add our test data. Create the file 'load_data.sql' in the db directory.

'springapp/db/load_data.sql':

```
INSERT INTO products (id, description, price) values(1, 'Lamp', 5.78);
```

```
INSERT INTO products (id, description, price) values(2, 'Table', 75.29);
INSERT INTO products (id, description, price) values(3, 'Chair', 22.81);
```

In the following section we will add some Ant targets to the build script so that we can run these SQL scripts.

5.3. Add Ant tasks to run scripts and load test data

We will create tables and populate them with test data using Ant's built-in "sql" task. To use this we need to add some database connection properties to the build properties file.

'springapp/build.properties':

```
# Ant properties for building the springapp

appserver.home=${user.home}/apache-tomcat-6.0.14
# for Tomcat 5 use $appserver.home}/server/lib
# for Tomcat 6 use $appserver.home}/lib
appserver.lib=${appserver.home}/lib

deploy.path=${appserver.home}/webapps

tomcat.manager.url=http://localhost:8080/manager
tomcat.manager.username=tomcat
tomcat.manager.password=s3cret

db.driver=org.hsqldb.jdbcDriver
db.url=jdbc:hsqldb:hsql://localhost
db.user=sa
db.pw=
```

Next we add the targets we need to the build script. There are targets to create and delete tables and to load and delete test data.

Add the following targets to **'springapp/build.xml'**:

```
    <target name="createTables">
        <echo message="CREATE TABLES USING: ${db.driver} ${db.url}"/>
        <sql driver="${db.driver}"
             url="${db.url}"
             userid="${db.user}"
             password="${db.pw}"
             onerror="continue"
             src="db/create_products.sql">
            <classpath refid="master-classpath"/>
        </sql>
    </target>

    <target name="dropTables">
        <echo message="DROP TABLES USING: ${db.driver} ${db.url}"/>
        <sql driver="${db.driver}"
             url="${db.url}"
             userid="${db.user}"
             password="${db.pw}"
             onerror="continue">
            <classpath refid="master-classpath"/>

        DROP TABLE products;

        </sql>
    </target>

    <target name="loadData">
        <echo message="LOAD DATA USING: ${db.driver} ${db.url}"/>
        <sql driver="${db.driver}"
             url="${db.url}"
             userid="${db.user}"
             password="${db.pw}"
```

```
            onerror="continue"
            src="db/load_data.sql">
           <classpath refid="master-classpath"/>
       </sql>
   </target>

   <target name="printData">
       <echo message="PRINT DATA USING: ${db.driver} ${db.url}"/>
       <sql driver="${db.driver}"
            url="${db.url}"
            userid="${db.user}"
            password="${db.pw}"
            onerror="continue"
            print="true">
           <classpath refid="master-classpath"/>

       SELECT * FROM products;

       </sql>
   </target>

   <target name="clearData">
       <echo message="CLEAR DATA USING: ${db.driver} ${db.url}"/>
       <sql driver="${db.driver}"
            url="${db.url}"
            userid="${db.user}"
            password="${db.pw}"
            onerror="continue">
           <classpath refid="master-classpath"/>

       DELETE FROM products;

       </sql>
   </target>

   <target name="shutdownDb">
       <echo message="SHUT DOWN DATABASE USING: ${db.driver} ${db.url}"/>
       <sql driver="${db.driver}"
            url="${db.url}"
            userid="${db.user}"
            password="${db.pw}"
            onerror="continue">
           <classpath refid="master-classpath"/>

       SHUTDOWN;

       </sql>
   </target>
```

Now you can execute **'ant createTables loadData printData'** to prepare the test data we will use later.

5.4. Create a Data Access Object (DAO) implementation for JDBC

Begin with creating a new **'springapp/src/repository'** directory to contain any classes that are used for database access. In this directory we create a new interface called ProductDao. This will be the interface that defines the functionality that the DAO implementation classes will provide – we could choose to have more than one implementation some day.

'springapp/src/springapp/repository/ProductDao.java':

```
package springapp.repository;

import java.util.List;

import springapp.domain.Product;
```

```
public interface ProductDao {

    public List<Product> getProductList();

    public void saveProduct(Product prod);

}
```

We'll follow this with a class called `JdbcProductDao` that will be the JDBC implementation of this interface. Spring provides a JDBC abstraction framework that we will make use of. The biggest difference between using straight JDBC and Spring's JDBC framework is that you don't have to worry about opening and closing the connection or any statements. It is all handled for you. Another advantage is that you won't have to catch any exceptions, unless you want to. Spring wraps all SQLExceptions in it's own unchecked exception hierarchy inheriting from DataAccessException. If you want to, you can catch this exception, but since most database exceptions are impossible to recover from anyway, you might as well just let the exception propagate up to a higher level. The class `SimpleJdbcDaoSupport` provides convenient access to an already configured `SimpleJdbcTemplate`, so we extend this class. All we will have to provide in the application context is a configured `DataSource`.

'springapp/src/springapp/repository/JdbcProductDao.java':

```
package springapp.repository;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.simple.ParameterizedRowMapper;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

import springapp.domain.Product;

public class JdbcProductDao extends SimpleJdbcDaoSupport implements ProductDao {

    /** Logger for this class and subclasses */
    protected final Log logger = LogFactory.getLog(getClass());


    public List<Product> getProductList() {
        logger.info("Getting products!");
        List<Product> products = getSimpleJdbcTemplate().query(
                "select id, description, price from products",
                new ProductMapper());
        return products;
    }

    public void saveProduct(Product prod) {
        logger.info("Saving product: " + prod.getDescription());
        int count = getSimpleJdbcTemplate().update(
            "update products set description = :description, price = :price where id = :id",
            new MapSqlParameterSource().addValue("description", prod.getDescription())
                .addValue("price", prod.getPrice())
                .addValue("id", prod.getId()));
        logger.info("Rows affected: " + count);
    }

    private static class ProductMapper implements ParameterizedRowMapper<Product> {

        public Product mapRow(ResultSet rs, int rowNum) throws SQLException {
            Product prod = new Product();
            prod.setId(rs.getInt("id"));
```

```
            prod.setDescription(rs.getString("description"));
            prod.setPrice(new Double(rs.getDouble("price")));
            return prod;
        }

    }

}
```

Let's go over the two DAO methods in this class. Since we are extending `SimpleJdbcSupport` we get a `SimpleJdbcTemplate` prepared and ready to use. This is accessed by calling the `getSimpleJdbcTemplate()` method.

The first method, `getProductList()` executes a query using the `SimpleJdbcTemplate`. We simply provide the SQL statement and a class that can handle the mapping between the `ResultSet` and the `Product` class. In our case the row mapper is a class named `ProductMapper` that we define as an inner class of the DAO. This class will so far not be used outside of the DAO so making it an inner class works well.

The ProductMapper implements the `ParameterizedRowMapper` interface that defines a single method named `mapRow` that must be implemented. This method will map the data from each row into a class that represents the entity you are retrieving in your query. Since the `RowMapper` is parameterized, the `mapRow` method returns the actual type that is created.

The second method saveProduct is also using the `SimplJdbcTemplate`. This time we are calling the `update` method passing in an SQL statement together with the parameter values in the form of a `MapSqlParameterSource`. Using a `MapSqlParameterSource` allows us to use named parameters instead of the typical "?" place holders that you are used to from writing plain JDBC. The named parameters makes your code more explicit and you avoid problems caused by parameters being set out of order etc. The update method returns the count of rows affected.

We need to store the value of the primary key for each product in the Product class. This key will be used when we persist any changes to the object back to the database. To hold this key we add a private field named 'id' complete with setters and getters to Product.java.

'springapp/src/springapp/domain/Product.java':

```
package springapp.domain;

import java.io.Serializable;

public class Product implements Serializable {

    private int id;
    private String description;
    private Double price;

    public void setId(int i) {
        id = i;
    }

    public int getId() {
        return id;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public Double getPrice() {
        return price;
```

```
    }

    public void setPrice(Double price) {
        this.price = price;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        buffer.append("Description: " + description + ";");
        buffer.append("Price: " + price);
        return buffer.toString();
    }
}
```

This completes the Simple JDBC implementation of our persistence layer.

5.5. Implement tests for JDBC DAO implementation

Time to add tests for the JDBC DAO implementation. Spring provides an extensive testing framework that supports JUnit 3.8 and 4 as well as TestNG. We can't cover all of that in this guide but we will show a simple implementation of the JUnit 3.8 specific support. We need to add the jar file containing the Spring test framework to our project. Copy **spring-test.jar** from the **'spring-framework-2.5/dist/modules'** directory to the **'springapp/war/WEB-INF/lib'** directory.

Now we can create our test class. By extending `AbstractTransactionalDataSourceSpringContextTests` we get a lot of nice features for free. We get dependency injection of any public setter from an application context. This application context is loaded by the test framework. All we need to do is specify the name for it in the `getConfigLocations` method. We also get an opportunity to prepare our database with the appropriate test data in the `onSetUpInTransaction` method. This is important, since we don't know the state of the database when we run our tests. As long as the table exists we will clear it and load what we need for our tests. Since we are extending a "Transactional" test, any changes we make will be automatically rolled back once the test finishes. The `deleteFromTables` and `executeSqlScript` methods are defined in the super class, so we don't have to implement them for each test. Just pass in the table names to be cleared and the name of the script that contains the test data.

'springapp/test/springapp/domain/JdbcProductDaoTests.java':

```
package springapp.repository;

import java.util.List;

public class JdbcProductDaoTests extends AbstractTransactionalDataSourceSpringContextTests {

    private ProductDao productDao;


    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    @Override
    protected String[] getConfigLocations() {
        return new String[] {"classpath:test-context.xml"};
    }

    @Override
    protected void onSetUpInTransaction() throws Exception {
        super.deleteFromTables(new String[] {"products"});
        super.executeSqlScript("file:db/load_data.sql", true);
    }

    public void testGetProductList() {
```

```
        List<Product> products = productDao.getProductList();

        assertEquals("wrong number of products?", 3, products.size());

    }

    public void testSaveProduct() {

        List<Product> products = productDao.getProductList();

        for (Product p : products) {
            p.setPrice(200.12);
            productDao.saveProduct(p);
        }

        List<Product> updatedProducts = productDao.getProductList();
        for (Product p : updatedProducts) {
            assertEquals("wrong price of product?", 200.12, p.getPrice());
        }

    }

}
```

We don't have the application context file that is loaded for this test yet, so let's create this file in the **'springapp/test'** directory:

'springapp/test/test-context.xml':

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

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

    <!-- the test application context definition for the jdbc based tests -->

    <bean id="productDao" class="springapp.repository.JdbcProductDao">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name="driverClassName" value="${jdbc.driverClassName}"/>
      <property name="url" value="${jdbc.url}"/>
      <property name="username"  value="${jdbc.username}"/>
      <property name="password" value="${jdbc.password}"/>
    </bean>
    <bean id="propertyConfigurer"
          class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations">
            <list>
                <value>classpath:jdbc.properties</value>
            </list>
        </property>
    </bean>

    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>

</beans>
```

We have defined a `productDao` which is the class we are testing. We have also defined a `DataSource` with place holders for the configuration values. These values are provided via a separate property file and at runtime, the

`PropertyPlaceholderConfigurer` that we have defined will read this property file and substitute the place holders with the actual values. This is convenient since this isolates the connection values into their own file. These values often need to be changed during application deployment. We put this new file in the **'war/WEB-INF/classes'** directory so it will be available when we run the application and also later when we deploy the web application. The content of this property file is:

'springapp/war/WEB-INF/classes/jdbc.properties':

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://localhost
jdbc.username=sa
jdbc.password=
```

Since we added a configuration file to the **'test'**" directory and a jdbc.properties file to the **'WEB-INF/classes'** directory, let's add a new classpath entry for our tests. It shoud go after the definition of the `'test.dir'` property:

'springapp/build.xml':

```
...
    <property name="test.dir" value="test"/>


    <path id="test-classpath">
        <fileset dir="${web.dir}/WEB-INF/lib">
            <include name="*.jar"/>
        </fileset>
        <pathelement path="${build.dir}"/>
        <pathelement path="${test.dir}"/>
        <pathelement path="${web.dir}/WEB-INF/classes"/>
    </path>

...
```

We should now have enough for our tests to run and pass but we want to make one additional change to the build script. It's a good practice to separate any integration tests that depend on a live database from the rest of the tests. So we add a separate "dbTests" target to our build script and exclude the database tests from the "tests" target.

'springapp/build.xml':

```
...

    <target name="tests" depends="build, buildtests" description="Run tests">
        <junit printsummary="on"
            fork="false"
            haltonfailure="false"
            failureproperty="tests.failed"
            showoutput="true">
            <classpath refid="test-classpath"/>
            <formatter type="brief" usefile="false"/>

            <batchtest>
                <fileset dir="${build.dir}">
                    <include name="**/*Tests.*"/>
                    <exclude name="**/Jdbc*Tests.*"/>
                </fileset>
            </batchtest>

        </junit>

        <fail if="tests.failed">
            tests.failed=${tests.failed}
            ***********************************************************
            ***********************************************************
            ****  One or more tests failed!  Check the output ...  ****
            ***********************************************************
```

```
            ******************************************************
        </fail>
    </target>

    <target name="dbTests" depends="build, buildtests,dropTables,createTables,loadData"
            description="Run db tests">
        <junit printsummary="on"
            fork="false"
            haltonfailure="false"
            failureproperty="tests.failed"
            showoutput="true">
            <classpath refid="test-classpath"/>
            <formatter type="brief" usefile="false"/>

            <batchtest>
                <fileset dir="${build.dir}">
                    <include name="**/Jdbc*Tests.*"/>
                </fileset>
            </batchtest>

        </junit>

        <fail if="tests.failed">
            tests.failed=${tests.failed}
            ******************************************************
            ******************************************************
            ****  One or more tests failed!  Check the output ...  ****
            ******************************************************
            ******************************************************
        </fail>
    </target>

...
```

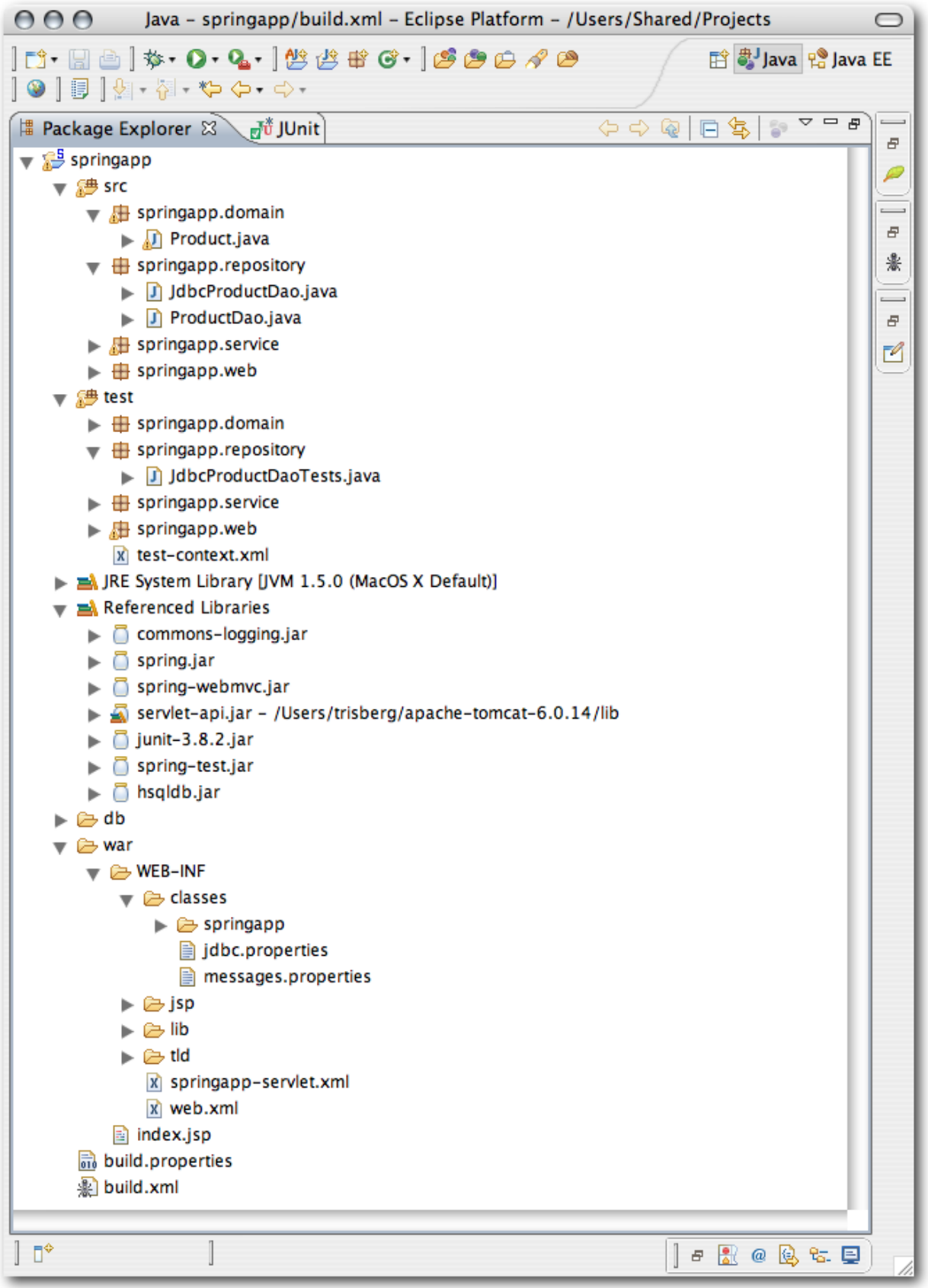
Time to run this test, execute **'ant dbTests'** to see if the tests pass.

5.6. Summary

We have now completed the persistence layer and in the next part we will integrate it with our web application. But first, lets quickly summarize hat we accomplished in this part.

- First we configured our database and created start-up scripts.
- We created scripts to use when creating the table and also to load some test data.
- Next we added some tasks to our build script to run when we needed to create or delete the table and also when we needed to add test data or delete the data.
- We created the actual DAO class that will handle the persistence work using Spring's `SimpeJdbcTemplate`.
- Finally we created unit or more accurately integration tests and corresponding ant targets to run these tests.

Below is a screen shot of what your project directory structure should look like after following the above instructions.


Java – springapp/build.xml – Eclipse Platform – /Users/Shared/Projects
Java
Java EE
Package Explorer
JUnit
springapp
src
springapp.domain
Product.java
springapp.repository
JdbcProductDao.java
ProductDao.java
springapp.service
springapp.web
test
springapp.domain
springapp.repository
JdbcProductDaoTests.java
springapp.service
springapp.web
test-context.xml
JRE System Library [JVM 1.5.0 (MacOS X Default)]
Referenced Libraries
commons-logging.jar
spring.jar
spring-webmvc.jar
servlet-api.jar – /Users/trisberg/apache-tomcat-6.0.14/lib
junit-3.8.2.jar
spring-test.jar
hsqldb.jar
db
war
WEB-INF
classes
springapp
jdbc.properties
messages.properties
jsp
lib
tld
springapp-servlet.xml
web.xml
index.jsp
build.properties
build.xml