Capitulo 6. Integrando la Aplicacion Web con la Capa de Persistencia

Esta es la Parte 6 del tutorial paso a paso sobre como desarrollar una aplicacion web desde cero usando Spring Framework. En la Parte 1 hemos configurado el entorno y puesto en marcha una aplicacion basica. En la Parte 2 hemos mejorado la aplicacion que habiamos construido hasta entonces. En la Parte 3 hemos añadido toda la logica de negocio y los test de unidad, y en la Parte 4 desarrollado la interface web. En la Parte 5 hemos desarrollado la capa de persistencia. Ahora es el momento de integrarlo todo junto en una aplicacion web completa.

6.1. Modificar la Capa de Servicio

Si hemos estructurado nuestra aplicacion adecuadamente, solo tenemos que cambiar la capa de servicio para que haga uso de la persistencia en base de datos. Las clases de la vista y el controlador no tienen que ser modificadas, puesto que no deberian ser conscientes de ningun detalle de la implementacion de la capa de servicio. Asi que vamos a añadir persistencia a la implementacion de ProductManager. Modifica la clase `SimpleProductManager` y añade una referencia a la interface `ProductDao` ademas de un metodo setter para esta referencia. Que implementacion usemos debe ser irrelevante para la clase ProductManager, y podemos hacerlo mediante configuracion. Tambien vamos a cambiar el metodo `setProducts` a uno llamado `setProductDao` para que podamos inyectar una instancia de la clase DAO. El metodo `getProducts` usara ahora este DAO para recuperar la lista de productos. Finalmente, el metodo `increasePrices` recuperara la lista de productos y, despues de haber incrementado los precios, almacenara los productos de nuevo en la base de datos usando el metodo `saveProduct` definido en el DAO.

"springapp/src/springapp/service/SimpleProductManager.java":

```
package springapp.service;

import java.util.List;

import springapp.domain.Product;

public class SimpleProductManager implements ProductManager {

    privapackage springapp.service;

import java.util.List;

import springapp.domain.Product;
import springapp.repository.ProductDao;

public class SimpleProductManager implements ProductManager {

    // private List<Product> products;
    private ProductDao productDao;

    public List<Product> getProducts() {
        // return products;
        return productDao.getProductList();
    }

    public void increasePrice(int percentage) {
        List<Product> products = productDao.getProductList();
        if (products != null) {
            for (Product product : products) {
                double newPrice = product.getPrice().doubleValue() *
                                  (100 + percentage)/100;
                product.setPrice(newPrice);
                productDao.saveProduct(product);
            }
        }
    }

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

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

}
```

6.2. Resolver los tests fallidos

Hemos modificado `SimpleProductManager` y ahora evidentemente los tests fallan. Necesitamos proporcionar a `ProductManager` una implementacion en memoria de `ProductDao`. Realmente no queremos usar el verdadero DAO puesto que queremos evitar tener acceso a la base de datos por nuestros tests de unidad. Añadiremos una clase llamada `InMemoryProductDao` que almacenara una lista de productos que seran definidos en el constructor. Esta clase en memoria tiene que ser pasada a `SimpleProductManager` en el momento de ejecutar los tests.

"springapp/test/springapp/repository/InMemoryProductDao.java":

```
package springapp.repository;

import java.util.List;

import springapp.domain.Product;
```

```
public class InMemoryProductDao implements ProductDao {

    private List<Product> productList;

    public InMemoryProductDao(List<Product> productList) {
        this.productList = productList;
    }

    public List<Product> getProductList() {
        return productList;
    }

    public void saveProduct(Product prod) {
    }

}
```

Y aqui esta la version modificada de `SimpleProductManagerTests`:

"`springapp/test/springapp/service/SimpleProductManagerTests.java`":

```
package springapp.service;

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

import springapp.domain.Product;
import springapp.repository.InMemoryProductDao;
import springapp.repository.ProductDao;

import junit.framework.TestCase;

public class SimpleProductManagerTests extends TestCase {

    private SimpleProductManager productManager;

    private List<Product> products;

    private static int PRODUCT_COUNT = 2;

    private static Double CHAIR_PRICE = new Double(20.50);
    private static String CHAIR_DESCRIPTION = "Chair";

    private static String TABLE_DESCRIPTION = "Table";
    private static Double TABLE_PRICE = new Double(150.10);

    private static int POSITIVE_PRICE_INCREASE = 10;

    protected void setUp() throws Exception {
        productManager = new SimpleProductManager();
        products = new ArrayList<Product>();

        // stub up a list of products
        Product product = new Product();
        product.setDescription("Chair");
        product.setPrice(CHAIR_PRICE);
        products.add(product);

        product = new Product();
        product.setDescription("Table");
        product.setPrice(TABLE_PRICE);
        products.add(product);

        ProductDao productDao = new InMemoryProductDao(products);
        productManager.setProductDao(productDao);
        //productManager.setProducts(products);
    }

    public void testGetProductsWithNoProducts() {
        productManager = new SimpleProductManager();
        productManager.setProductDao(new InMemoryProductDao(null));
        assertNull(productManager.getProducts());
    }

    public void testGetProducts() {
        List<Product> products = productManager.getProducts();
        assertNotNull(products);
        assertEquals(PRODUCT_COUNT, productManager.getProducts().size());

        Product product = products.get(0);
        assertEquals(CHAIR_DESCRIPTION, product.getDescription());
        assertEquals(CHAIR_PRICE, product.getPrice());

        product = products.get(1);
        assertEquals(TABLE_DESCRIPTION, product.getDescription());
        assertEquals(TABLE_PRICE, product.getPrice());
    }

    public void testIncreasePriceWithNullListOfProducts() {
        try {
            productManager = new SimpleProductManager();
            productManager.setProductDao(new InMemoryProductDao(null));
            productManager.increasePrice(POSITIVE_PRICE_INCREASE);
        }
        catch(NullPointerException ex) {
            fail("Products list is null.");
        }
    }

    public void testIncreasePriceWithEmptyListOfProducts() {
        try {
            productManager = new SimpleProductManager();
            productManager.setProductDao(new InMemoryProductDao(new ArrayList<Product>()));
```

```
            //productManager.setProducts(new ArrayList<Product>());
            productManager.increasePrice(POSITIVE_PRICE_INCREASE);
        }
        catch(Exception ex) {
            fail("Products list is empty.");
        }
    }

    public void testIncreasePriceWithPositivePercentage() {
        productManager.increasePrice(POSITIVE_PRICE_INCREASE);
        double expectedChairPriceWithIncrease = 22.55;
        double expectedTablePriceWithIncrease = 165.11;

        List<Product> products = productManager.getProducts();
        Product product = products.get(0);
        assertEquals(expectedChairPriceWithIncrease, product.getPrice());

        product = products.get(1);
        assertEquals(expectedTablePriceWithIncrease, product.getPrice());
    }

}
```

Tambien necesitamos modificar `InventoryControllerTests` puesto que esta clase tambien usa `SimpleProductManager`. Aqui esta la version modificada de `InventoryControllerTests`:

"springapp/test/springapp/service/InventoryControllerTests.java":

```
package springapp.web;

import java.util.Map;

import org.springframework.web.servlet.ModelAndView;

import springapp.domain.Product;
import springapp.repository.InMemoryProductDao;
import springapp.service.SimpleProductManager;
import springapp.web.InventoryController;

import junit.framework.TestCase;

public class InventoryControllerTests extends TestCase {

    public void testHandleRequestView() throws Exception{
        InventoryController controller = new InventoryController();
        SimpleProductManager spm = new SimpleProductManager();
        spm.setProductDao(new InMemoryProductDao(new ArrayList<Product>()));
        controller.setProductManager(spm);
        //controller.setProductManager(new SimpleProductManager());
        ModelAndView modelAndView = controller.handleRequest(null, null);
        assertEquals("hello", modelAndView.getViewName());
        assertNotNull(modelAndView.getModel());
        Map modelMap = (Map) modelAndView.getModel().get("model");
        String nowValue = (String) modelMap.get("now");
        assertNotNull(nowValue);
    }
}
```

6.3. Crear un nuevo contexto de aplicacion para configurar la capa de servicio

Hemos visto antes que es tremendamente facil modificar la capa de servicio para usar persistencia en base de datos. Esto es asi porque esta despegada de la capa web. Ahora es el momento de despegar tambien la configuracion de la capa de servicio de la capa web. Eliminaremos la configuracion de productManager y la lista de productos del archivo de configuracion springapp-servlet.xml. Asi es como este archivo quedaria ahora:

"springapp/war/WEB-INF/springapp-servlet.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 application context definition for the springapp DispatcherServlet -->

    <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename" value="messages"/>
    </bean>

    <bean name="/hello.htm" class="springapp.web.InventoryController">
        <property name="productManager" ref="productManager"/>
    </bean>

    <bean name="/priceincrease.htm" class="springapp.web.PriceIncreaseFormController">
        <property name="sessionForm" value="true"/>
        <property name="commandName" value="priceIncrease"/>
        <property name="commandClass" value="springapp.service.PriceIncrease"/>
        <property name="validator">
            <bean class="springapp.service.PriceIncreaseValidator"/>
        </property>
        <property name="formView" value="priceincrease"/>
        <property name="successView" value="hello.htm"/>
        <property name="productManager" ref="productManager"/>
    </bean>

    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"></property>
        <property name="prefix" value="/WEB-INF/jsp/"></property>
```

```
        <property name="suffix" value=".jsp"></property>
    </bean>

</beans>
```

Todavia necesitamos configurar la capa de servicio y lo haremos en nuestro propio archivo de contexto de aplicacion. Este archivo se llama "applicationContext.xml" y sera cargado mediante un servlet listener que definiremos en "web.xml". Todos los bean configurados en este nuevo contexto de aplicacion estaran disponibles desde cualquier contexto del servlet.

"springapp/war/WEB-INF/web.xml":

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

<web-app version="2.4"
         xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
         http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd" >

  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>

  <servlet>
    <servlet-name>springapp</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>springapp</servlet-name>
    <url-pattern>*.htm</url-pattern>
  </servlet-mapping>

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

  <jsp-config>
    <taglib>
      <taglib-uri>/spring</taglib-uri>
      <taglib-location>/WEB-INF/tld/spring-form.tld</taglib-location>
    </taglib>
  </jsp-config>

</web-app>
```

Ahora creamos un nuevo archivo "applicationContext.xml" en el directorio "war/WEB-INF"".

"springapp/war/WEB-INF/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"
         xmlns:tx="http://www.springframework.org/schema/tx"
         xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop-2.0.xsd
           http://www.springframework.org/schema/tx
           http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">

    <!-- the parent application context definition for the springapp application -->

    <bean id="productManager" class="springapp.service.SimpleProductManager">
        <property name="productDao" ref="productDao"/>
    </bean>

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

</beans>
```

6.4.Añadir transaccion y una configuracion de pool de conexiones al contexto de la aplicacion

Siempre que persistas informacion en una base de datos es mejor usar transacciones para asegurarte que o todas o ninguna de tus actualizaciones son realizadas. Asi evitas tener la mitad de tus actualizaciones persistidas mientras la otra mitad ha fallado. Spring ofrece un extenso margen de opciones para configurar mantenimiento de transacciones. El manual de referencia cubre este tema en profundidad. Aqui haremos uso de esta caracteristica usando AOP (Aspect Oriented Programming - Programacion Orientada a Aspectos) en la forma de un advice (consejo) de transaccion y un pointcut (punto de corte) AspectJ para definir donde deben ser aplicadas las transacciones. Si estas interesado en como funciona este mecanismo mas profundamente, echale un vistazo al manual de referencia. Vamos a usar el nuevo soporte para nombres de espacio introducido en Spring 2.0. Los nombres de espacio "aop" y "tx" hacen las entradas de configuracion mucho mas concisas comparadas que el sistema tradicional, el cual usa entradas de tipo "<bean>".

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

    <aop:config>
```

```
    <aop:advisor pointcut="execution(* *..ProductManager.*(..))" advice-ref="txAdvice"/>
</aop:config>

<tx:advice id="txAdvice">
    <tx:attributes>
        <tx:method name="save*"/>
        <tx:method name="*" read-only="true"/>
    </tx:attributes>
</tx:advice>
```

El pointcut es aplicado a cualquier metodo que invoques en la interface ProductManager. El advice es un advice de transaccion, y es aplicado a metodos cuyo nombre comience con 'save'. Son aplicados los atributos de configuracion por defecto (REQUIRED) puesto que ningun otro atributo ha sido especificado. El advice tambien es aplicado a transacciones "read-only" (de solo lectura) en cualquier otro metodo que sea alcanzado mediante el pointcut.

Tambien necesitamos definir un DataSource donde configuramos los parametros de conexion a la base de datos, asi como un configurador de propiedades, el cual leera dichos parametros desde el archivo **"jdbc.properties"** que hemos creado en la parte 5. Este Datasource usara automaticamente una conexion tipo pool, en nuestro caso una conexion DBCP del proyecto Apache Jakarta.

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <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>
```

Para que todo esto funcione necesitamos copiar algunos archivos jar en el directorio **"WEB-INF/lib"**. Copia **aspectjweaver.jar** desde el directorio **"spring-framework-2.5/lib/aspectj"** y **commons-dbcp.jar** y **commons-pool.jar** desde el directorio **"spring-framework-2.5/lib/jakarta-commons"** al directorio **"springapp/war/WEB-INF/lib"**.

Aqui esta la version final de nuestro archivo **"applicationContext.xml"**:

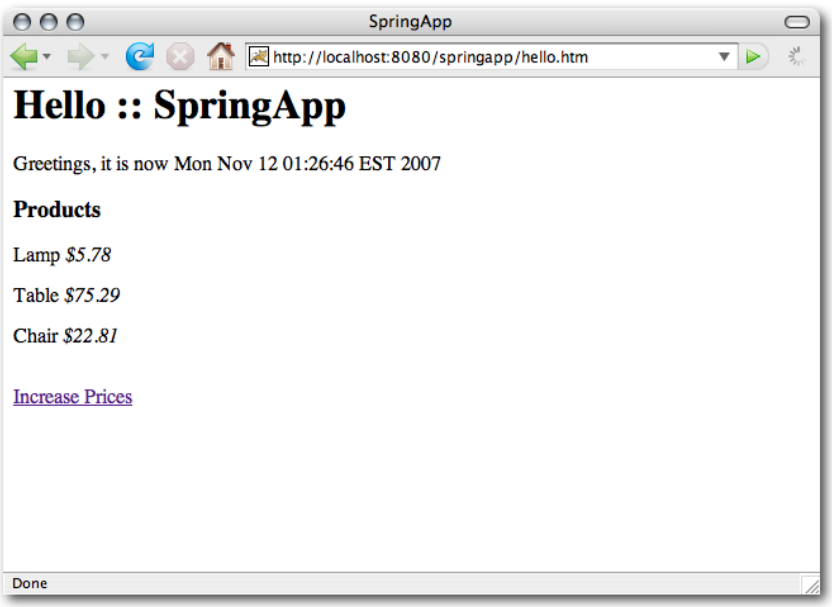
"springapp/war/WEB-INF/applicationContext.xml":

```
<?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"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
         http://www.springframework.org/schema/aop
         http://www.springframework.org/schema/aop/spring-aop-2.0.xsd
         http://www.springframework.org/schema/tx
         http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">

    <!-- the parent application context definition for the springapp application -->

    <bean id="productManager" class="springapp.service.SimpleProductManager">
        <property name="productDao" ref="productDao"/>
    </bean>

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

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
        <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>

    <aop:config>
        <aop:advisor pointcut="execution(* *..ProductManager.*(..))" advice-ref="txAdvice"/>
    </aop:config>

    <tx:advice id="txAdvice">
        <tx:attributes>
            <tx:method name="save*"/>
            <tx:method name="*" read-only="true"/>
        </tx:attributes>
    </tx:advice>
```

```
</beans>
```

6.5. Test final de la aplicacion completa

Ahora es el momento de ver si todas estas piezas funcionan juntas. Construye y despliega la aplicacion finalizada y recuerda tener la base de datos arrancada y funcionando. Esto es lo que deberias ver cuando apuntes tu navegador web a la aplicacion despues de ser recargada:



La aplicacion completa

Aparece exactamente como lo hacia antes. Sin embargo hemos añadido la persistencia en base de datos, por lo que si cierras la aplicacion tus incrementos de precio no se perderan. Ellos estaran todavia alli cuando vuelvas a cargar la aplicacion.

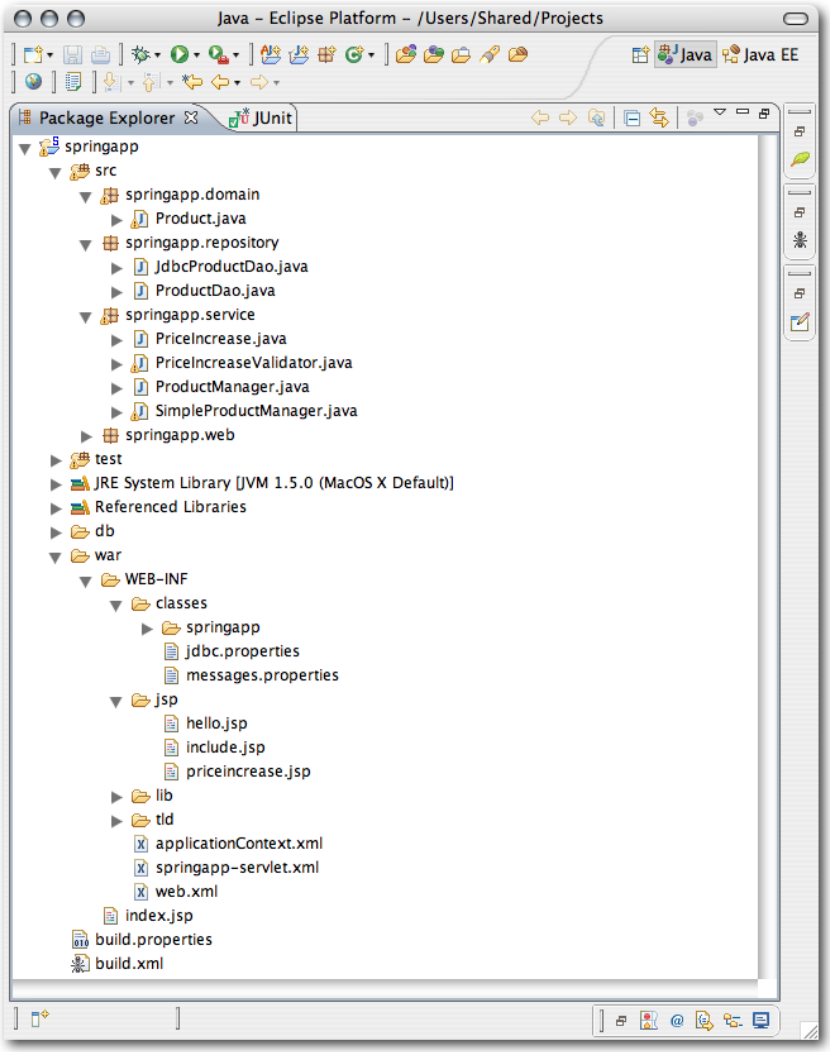
Un monton de trabajo para una aplicacion simple, pero nuestra meta nunca fue solamente escribir (¡y traducir!) la aplicacion. La meta fue mostrar como crear una aplicacion Spring MVC desde cero y ahora sabemos que las aplicaciones que tu construiras desde este momento seran mucho mas complejas. Deberas seguir los mismos pasos, y esperamos que hayas adquirido el conocimiento suficiente para hacerte mas facil comenzar a usar Spring.

6.6. Resumen

Hemos completado las tres capas de la aplicacion -- la capa web, la capa de servicio y la capa de persistencia. En esta ultima parte hemos reconfigurado la aplicacion.

- Primero hemos modificado la capa de servicio para usar la interface ProductDAO.
- Despues hemos tenido que arreglar algunos fallos en los tests de la capa de servicio y la capa web.
- A continuacion hemos introducido un nuevo applicationContext para separar la configuracion de la capa de servicio y de la capa de persistencia de la configuracion de la capa web.
- Hemos definido cierto mantenimiento de transacciones para la capa de servicio y configurado un pool de conexiones para las conexiones a la base de datos.
- Finalmente hemos construido la aplicacion y testeado que aun funciona.

Debajo puedes ver una captura de pantalla mostrando como debe aparecer tu estructura de directorios despues de seguir todas las instrucciones anteriores.


Java – 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
PriceIncrease.java
PriceIncreaseValidator.java
ProductManager.java
SimpleProductManager.java
springapp.web
test
JRE System Library [JVM 1.5.0 (MacOS X Default)]
Referenced Libraries
db
war
WEB-INF
classes
springapp
jdbc.properties
messages.properties
jsp
hello.jsp
include.jsp
priceincrease.jsp
lib
tld
applicationContext.xml
springapp-servlet.xml
web.xml
index.jsp
build.properties
build.xml