Chapter 6. Integrating the Web Application with the Persistence Layer

This is Part 6 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. In Part 5 we developed the persistence layer. It is now time to integrate all this into a complete web application.

6.1. Modify service layer

If we structured our application properly, we should only have to change the service layer classes to take advantage of the database persistence. The view and controller classes should not have to be modified, since they should be unaware of any implementation details of the service layer. So let's add the persistence to the ProductManager implementation. We modify the `SimpleProductManager` and add a reference to a `ProductDao` interface plus a setter method for this reference. Which implementation we actually use here should be irrelevant to the ProductManager class, and we will set this through a configuration option. We also change the `setProducts` method to a `setProductDao` method so we can inject an instance of the DAO class. The `getProducts` method will now use the DAO to retrieve a list of products. Finally, the `increasePrices` method will now get the list of products and then after the price have been increased the product will be stored in the database using the `saveProduct` method on the 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. Fix the failing tests

We rewrote the SimpleProductManager and now the tests will of course fail. We need to provide the ProductManager with an in-memory implementation of the ProductDao. We don't really want to use the real DAO here since we'd like to avoid having to access a database for our unit tests. We will add an internal class called InMemoryProductDao that will hold on to a list pf products provided in the constructor. This in-memory class has to be passed in when we create a new SimpleProductManager.

'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) {
    }

}
```

And here is the modified 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());
    }

}
```

We also need to modify the `InventoryControllerTests` since that class also uses the `SimpleProductManager`. Here is the modified `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. Create new application context for service layer configuration

We saw earlier that it was fairly easy to modify the service layer to use the database persistence. This was because it is decoupled from the web layer. It's now time to decouple or configuration of the service layer from the web layer as well. We will remove the productManager configuration and the list of products from the springapp-servlet.xml configuration file. This is what this file looks like now:

'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>
```

We still need to configure the service layer and we will do that in its own application context file. This file is called **'applicationContext.xml'** and it will be loaded via a servlet listener that we will define in **'web.xml'**. All bean configured in this new application context will be available to reference from any servlet context.

'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>
```

Now we create a new **'applicationContext.xml'** file in the **'war/WEB-INF'**" directory.

'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. Add transaction and connection pool configuration to application context

Any time you persist data in a database its best to use transactions to ensure that all your updates are perform or none are completed. You want to avoid having half your updates persisted while the other half failed. Spring provides an extensive range of options for how to provide transaction management. The reference manual covers this in depth. Here we will make use of one way of providing this using AOP (Aspect Oriented Programming) in the form of a transaction advice and an ApectJ pointcut to define where the transactions should be applied. If you are interested in how this works in more depth, take a look at the reference manual. We are using the new namespace support introduced in Spring 2.0. The "aop" and "tx" namespaces make the configuration entries much more concise compared to the traditional way using regular "<bean>" entries.

```
    <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>
```

The pointcut applies to any method called on the ProductManager interface. The advice is a transaction advice that applies to methods with a name starting with 'save'. The default transaction attributes of REQUIRED applies since no other attribute was specified. The advice also applies "read-only" transactions on any other methods that are adviced via the pointcut.

We also need to define a connection pool. We are using the DBCP connection pool from the Apache Jakarta project. We are reusing the **'jdbc.properties'** file we created in Part 5.

```
    <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>
```

For all this to work we need some additional jar files copied to the **'WEB-INF/lib'** directory. Copy **aspectjweaver.jar** from the **'spring-framework-2.5/lib/aspectj'** directory and **commons-dbcp.jar** and **commons-pool.jar** from the **'spring-framework-2.5/lib/jakarta-commons'** directory to the **'springapp/war/WEB-INF/lib'** directory.

Here is the final version of our **'applicationContext.xml'** file:

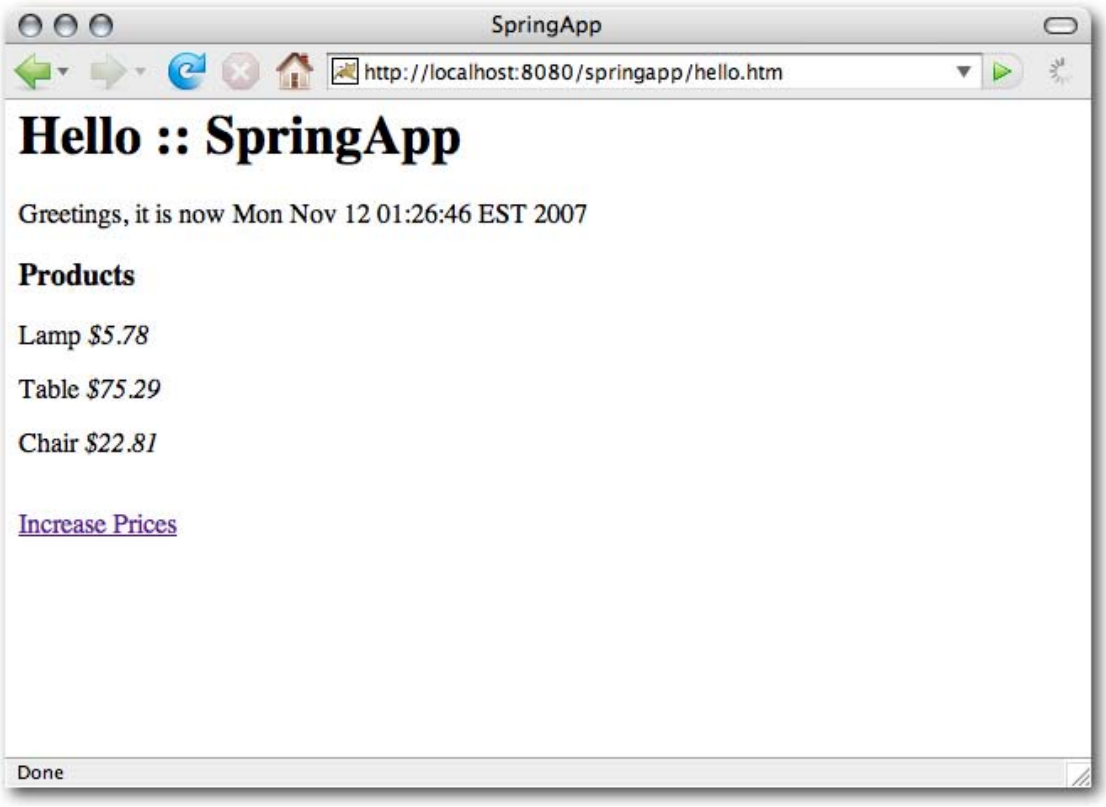
'springapp/war/WEB-INF/applicationContext.xml':

```
<?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. Final test of the complete application

Now it's finally time to see if all of these pieces will work together. Build and deploy

your finished application and remember to have the database up and running. This is what you should see when pointing the web browser at the application after it has reloaded:



The completed application

Looks just the same as it did before. We did add persistence though, so if you shut down the application your price increases will not be lost. They are still there when you start the application back up.

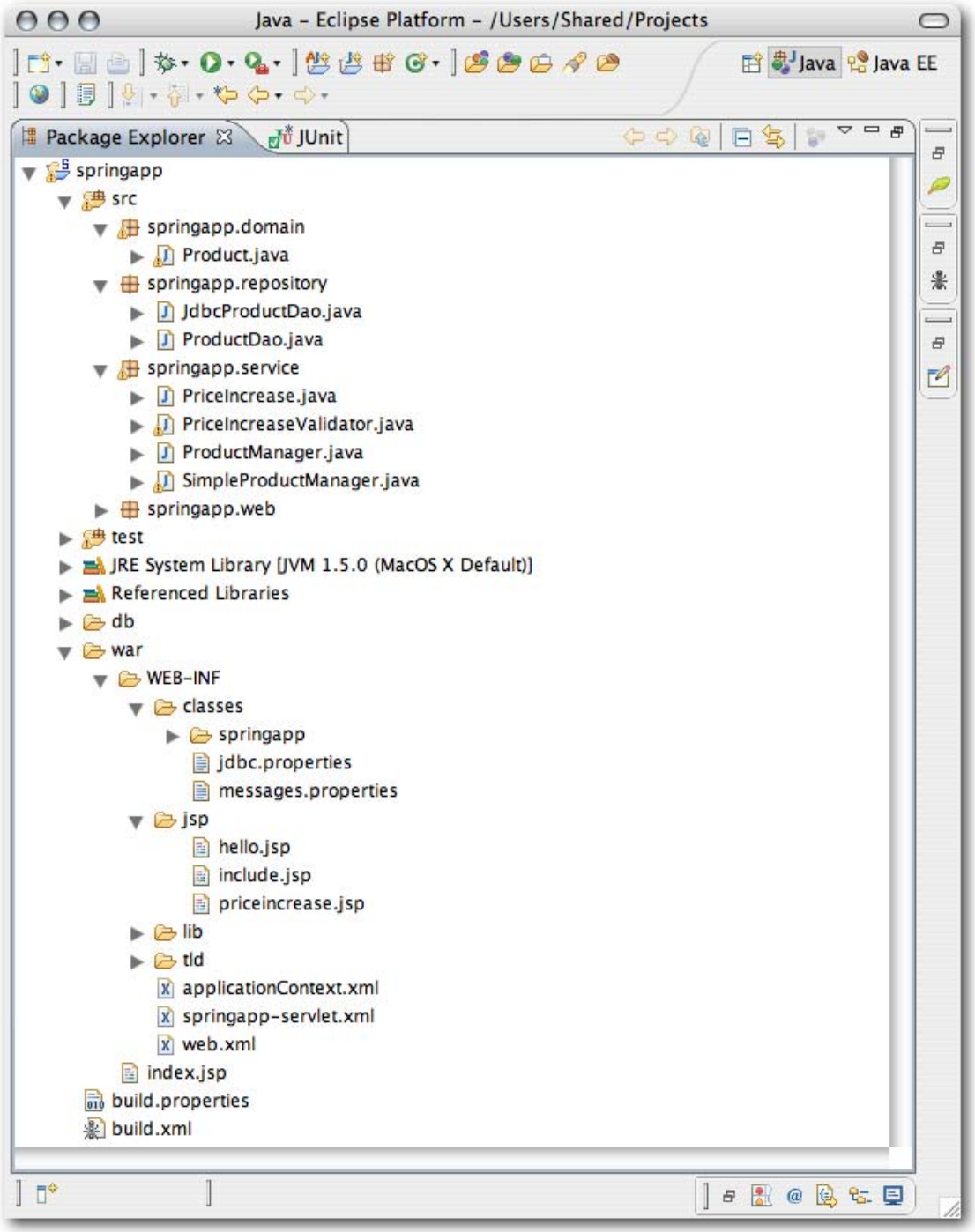
A lot of work for a very simple application, but it was never our goal to just write this application. The goal was to show how to go about creating a Spring MVC application from scratch and we know that the applications you will create are much more complex. The same steps apply though and we hope you have gained enough knowledge to make it easier getting started to use Spring.

6.6. Summary

We have completed all three layers of the application -- the web layer, the service layer and the persistence layer. In this last part we reconfigured the application.

- First we modified the service layer to use the ProductDao.
- We then had to fix some failing service and web layer tests.
- Next we introduced a new applicationContext to separate the service and persistence layer configuration from the web layer configuration.
- We also defined some transaction management for the service layer and configured a connection pool for the database connections.
- Finally we built the reconfigured application and tested that it still worked.

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



Prev
Chapter 5. Implementing Database Persistence

Home
Sponsored by SpringSource

Next
Appendix A. Build Scripts