appendix A: Working with Struts

From among the many Java-based web server frameworks available, we settled on a Struts/Hibernate/MySQL solution as our representative framework for developing enterprise-class Java EE web applications. Because Laszlo only requires XML documents as its input, this dramatically simplifies the presentation services, thus reducing the differences between alternative frameworks. As a result, past criteria for selecting a particular framework are irrelevant with Laszlo.

In this appendix, we'll examine Struts's overall architecture and how to repurpose existing Struts applications to support Laszlo applications.

A.1 Creating our Struts application

Struts is an open source framework implementing the model-view-controller (MVC) architecture to develop Java EE web applications. The central purpose of every web application is to receive an HTTP request for processing and return an HTTP response. When a web container receives an incoming URI, it checks for a .do extension indicating that the request is to be forwarded to Struts's *front controller* servlet (known as the *ActionServlet*). A front controller is a singleton servlet that fields all requests for subsidiary controllers. Although the ActionServlet performs several services, its main task is to route URI requests to these other controllers, known as *Actions*.

The struts-config.xml file contains a roadmap for our Struts application that maps the URI, known as *ActionMappings*, to their Actions. The ActionServlet looks through this file for an entry with a matching URI path. In our case, there is only a single matching URI: /products.do. The ActionServlet executes this `Action` with its set of `ActionForward` objects that represent the various processing outcomes. The eventual outcome—which `ActionForward` object is selected—is determined by processing the request.

Let's next look at how to create a struts-config roadmap.

A.1.1 Creating the struts-config roadmap

A struts-config.xml file is composed of three collections of objects:

- `global-forwards` containing `ActionForward` objects
- `form-beans` containing `ActionForm` objects
- `action-mappings` containing `Actions` and their `forward` objects

Listing A.1 shows the struts-config file that supplies products to the Laszlo Market.

Listing A.1 struts-config.xml to support Laszlo

```
<struts-config>
   <global-forwards>
      <forward name="success" path="/jsp/results.jsp"/>
      <forward name="failure" path="/jsp/error.jsp"/>
   </global-forwards>

   <form-beans>
      <form-bean name="productForm"
                 type="com.laszlomarket.ProductForm"/>
   </form-beans>
   <action-mappings>
      <action path="/products"
              type="com.laszlomarket.ProductAction"
              name="productForm" validate="true">
         <forward name="status" path="/pages/status.jsp"/>
         <forward name="add_product"
                  path="/jsp/add_product.jsp"/>
         <forward name="display_products"
                  path="/jsp/display_products.jsp"/>
      </action>
   </action-mappings>
</struts-config>
```

Our struts-config.xml file contains a single mapping action with a /products path, which specifies

- The name of the `ActionForm` object to use—a `ProductForm`
- The `Action` object to use—a `ProductAction`
- A list of possible forward destinations

Figure A.1 illustrates the overall architecture of our Struts application. The application is accessed through the /product.do URI, which matches the `Action` object identified by the product path. Both our `Action` and `ActionForm` will have a product prefix, so we'll have a `productAction` and a `productForm`. Also associated with this action are two sets of `ActionForward` objects, *global* and *local*. A global `ActionForward` is used by all `Actions`, while a local `ActionForward` is only used by our `ProductAction`.

Rather than diving straight in and creating all of these software modules, let's start by looking at how to retrofit an existing Struts application to interface to Laszlo.

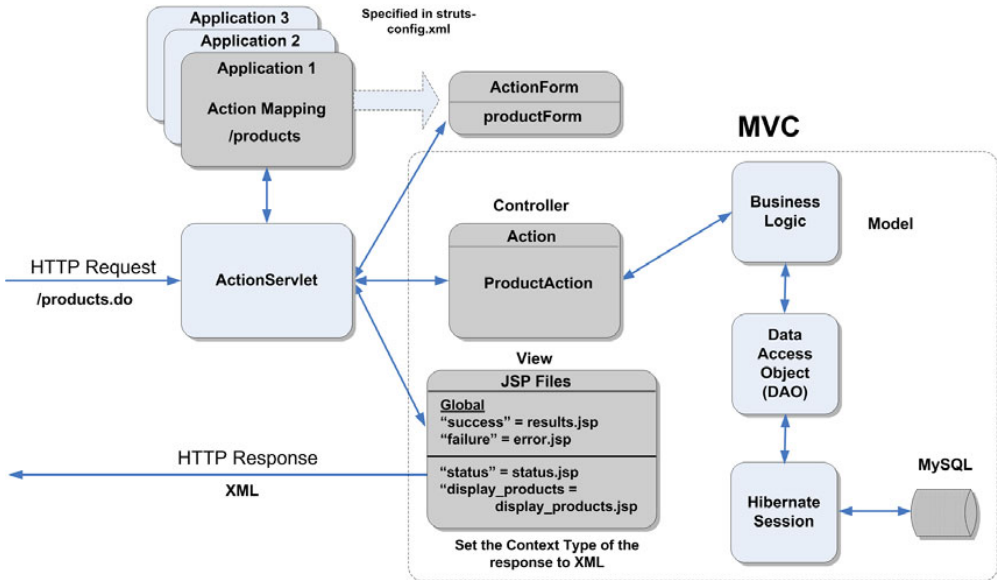

Figure A.1 The shaded objects within the dotted area are the objects defined by the matching ActionMapping. These objects are called by the ActionServlet.

A.1.2 Retrofitting Struts applications

The Struts MVC architecture confines the generation of XML or HTML output to its (JSP code) view layer. So a Struts application's existing `Action`, `ActionForm`, and business objects don't need to be modified, but rather can be made to work with Laszlo just by adding another set of JSP files. These JSP files generate an XML document instead of HTML. Your struts-config.xml file will need to be updated to provide an additional Action Mapping path. Figure A.2 shows the additional Action Mapping `/loginlzx` and results-xml.jsp file for adding Laszlo support for an existing `/login` action.

The mapping capability of the struts-config.xml file is used to support both HTML and XML output. We'll add a new action mapping, `/loginlzx`, to generate an XML response. This new mapping will use the existing `LoginForm` and `LoginAction` objects, but it forwards the request to a different JSP file, result-xml.jsp, to generate an XML response. The affected areas of the struts-config.xml file are shown in bold.

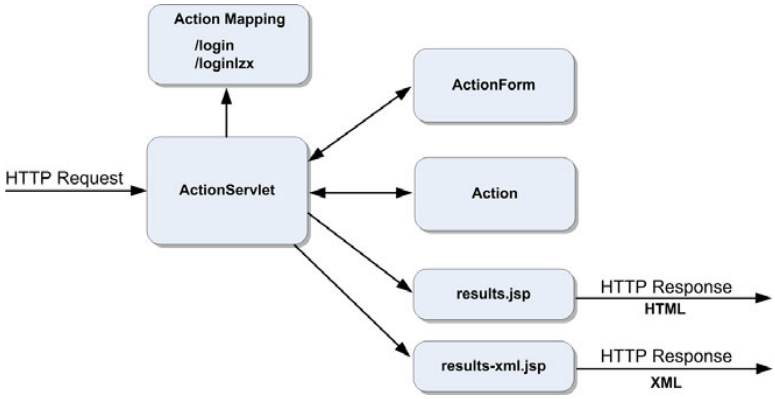

Figure A.2 Existing Struts applications can be modified to support Laszlo applications by adding another set of JSP files to generate XML rather than HTML.

```
<action-mappings>
   <action path="/login" type="com.laszlomarket.LoginAction"
                   name="loginForm" validate="true">
      <forward name="success" path="/result.jsp"/>
   </action>
   <action path="/loginlzx" type="com.laszlomarket.LoginAction"
                      name="loginForm" validate="true">
      <forward name="success" path="/result-xml.jsp"/>
   </action>
</action-mappings>
```

The original JSP files can be used to get a jumpstart on these new XML-based JSP files, since most of the iterative tag logic can be reused.

A.1.3 Creating our Struts controller

Let's start with a technical overview of the entire progression through Struts. Once an `ActionMapping` match is found for our URI, another search begins for a matching `form-bean` within the `form-beans` tag. This matching entry contains the location of the `ActionForm` class, which is instantiated as a `ProductForm` object. Before the `ProductForm` is populated, Struts calls its `reset` method to clear its data fields. Afterward, the `ProductForm` object is populated by the request parameters.

Next, the `ActionServlet` invokes the `execute` method of the `Action` object passing the `ActionMapping`, `ActionForm`, and the HTTP request and response objects as arguments. Depending on the result of the action, an `ActionForward` object is used to inform the `ActionServlet` controller where the request object should be forwarded to next. This destination could be either a JSP file to produce

an XML response or another `Action` for further processing. The JSP file sets the HTTP response's context type to XML to ensure that the body of the response contains XML, and this response is sent back to Laszlo.

So let's take the first step by creating our `ProductForm`.

Creating our ProductForm

An `ActionForm` provides a buffer to validate the input data. If the `validate` attribute of the action is true, which is the default, the `validate` method is called to perform field and form validation by checking relationships between fields. Before a data form is submitted to the server, a Laszlo application should perform its own validation to ensure that all data is valid. But it is still necessary to perform validation on the server to ensure that data fields are not corrupted in transit.

Creating a validate method

A `validate` method must be strict about validating all data inputs. For demonstration purposes, we provide only validation for the `action` parameter. In a real-world situation, every input parameter should be checked. The `action` parameter specifies the action to be performed by the `Action` controller. Listing A.2 shows our example `validate` method.

Listing A.2 A validate method for input data validation

```
public ActionErrors validate(ActionMapping mapping,
                             HttpServletRequest request) {
ActionErrors errors = new ActionErrors();      (1)
String action = request.getParameter("action");

   if ("add".equals(action)) {                          (2)
      if (id == null || id.intValue() < 0) {
         errors.add(ActionErrors.GLOBAL_ERROR,
                new ActionError("errors.required", "id"));
         return errors;
      }
   }
   else if ("delete".equals(action)) {
      if (id == null || id.intValue() < 0) {
            errors.add(ActionErrors.GLOBAL_ERROR,
                   new ActionError("errors.required", "id"));
            return errors;
      }
   }
   else if ("get".equals(action)) {
      if (id == null || id.intValue() < 0) {
         errors.add(ActionErrors.GLOBAL_ERROR,
                 new ActionError("errors.required", "id"));
```

```
            return errors;
        }
    }
    else if ("update".equals(action)) {
        if (id == null || id.intValue() < 0) {
            errors.add(ActionErrors.GLOBAL_ERROR,
                 new ActionError("errors.required", "id"));
            return errors;
        }
    }
    else {
        errors.add(ActionErrors.GLOBAL_ERROR,
                  new ActionError("errors.unknownaction"));
        return errors;
    }
    return errors;
}
```

Listing A.2 first creates an `ActionErrors` object ❶ to hold any errors. We'll require that each action must have an `id` ❷ that has a positive value. If the `validate` method returns null or an empty `ActionErrors` collection, then we continue on to our `ProductAction`'s `execute` method. Otherwise, we're routed to a JSP file to display an error message. Now we'll start creating our `ProductAction`.

A.1.4 Creating the ProductAction

An `Action` is a coordinator between the model and the view layers to dictate the flow of an application. The request's `action` parameter specifies an operation—add, delete, and update for updating and replacing—that maps to one of the CRUD operations. If the CRUD operation succeeds, then the `ActionForward` is set to `success`; otherwise an exception is thrown and the `ActionForward` object is set to `failure`. Listing A.3 shows the code for this action, where each of the CRUD operations is represented by stubs.

Listing A.3 Overview of the Struts `Action`

```
public ActionForward execute(ActionMapping mapping, ActionForm form,
      HttpServletRequest request, HttpServletResponse response)
   throws Exception {
      ActionErrors errors = new ActionErrors();

      String action =
         request.getParameter("action");
      try {
         if ("add".equals(action)) { … }
         else if ("delete".equals(action)) {  … }
         else if ("update".equals(action)) { … }
```

Gets action for processing

```
      else if ("get".equals(action)) { … }
      else if (action == null || "list".equals(action)) { … } }
   catch (Exception e) {
      log.warn("productAction exception: " + e.getMessage());
      errors.add(ActionErrors.GLOBAL_ERROR, new
                 ActionError("errors.invalidDBOp"));
      saveErrors(request, errors);
      return mapping.findForward("failure"); }
   return mapping.findForward("success"); } }
```

Puts errors into request

Let's next look at the model layer, since it returns the value that controls which `ActionForward` is returned by the controller.

A.2 Creating the model

Hibernate is an open source object/relational mapping (ORM) framework for Java, supporting the persistence of Java objects in a relational database. Once a mapping between a Java class and a database table has been created, complex objects, such as our class instances, can be saved and retrieved from a database without having to perform low-level SQL operations.

Hibernate acts as a buffer between these two worlds, providing persistent classes to support object-oriented idioms like inheritance and polymorphism. This allows developers to work with objects in a completely object-oriented way and to ignore the persistence issues of databases.

Next, we'll see the steps to configure Hibernate.

A.2.1 Configuring Hibernate

Configuring Hibernate requires updating its two configuration files: the hibernate.properties file that contains database settings and a Hibernate mapping file that contains class and database table-mapping files to support persistence. We'll start by updating Hibernate's properties file.

Creating the hibernate.properties file

This hibernate.properties file contains all the configuration parameters to a MySQL database. This includes specifying the database to use, its driver's name and location, and the name and password of the user:

```
hibernate.dialect                org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class  org.gjt.mm.mysql.Driver
hibernate.connection.url          jdbc:mysql://localhost:3306/market
hibernate.connection.username      laszlo
hibernate.connection.password      laszlo
```

We'll assume that we have a product database table that is described by this schema:

```
create table products (
    id int not null auto_increment,
    sku   varchar(12) not null,
    title varchar(100) not null,
    description text not null,
    specs text not null,
    image varchar(200) not null,
    video varchar(200) not null,
    category varchar(100) not null,
    price decimal(10,2) not null,
    primary key (id)
);
```

Now that Hibernate is connected to a database, we'll examine how it creates its mappings.

Creating the Hibernate mapping file

Hibernate's mapping file for our `Product` object is called Product.hbm.xml. This file maps the properties of our `Product` class to the fields of a `product` database table. Whenever either the Java object or the database table is updated, this file must also be updated:

```
<hibernate-mapping package="com.laszlomarket">
   <class name="Product" table="product">      ❶
      <id name="id" column="id" type="java.lang.Long"
          unsaved-value="0">      ❷
         <generator class="identity"/>
      </id>
      <property name="title" column="title" type="java.lang.String"/>
      <property name="image" column="image" type="java.lang.String"/>
      <property name="video" column="video" type="java.lang.String"/>
      <property name="price" column="price" type="java.lang.Double"/>
      <property name="sku" column="sku" type="java.lang.String"/>
      <property name="description" column="description"
                type="java.lang.String"/>      ❸
      <property name="specs" column="specs" type="java.lang.String"/>
   </class>
</hibernate-mapping>
```

First, we specify ❶ that a class with the name `com.laszlomarket.Product` is associated with the database table `product`. The name `id` ❷ is a special property that represents the database identifier—the primary key—of this class. A simple mapping ❸ can be used between each of the `property` names and their database fields.

Now that we have Hibernate configured, we can start using it. But rather than scattering Hibernate calls throughout the application, a better approach is to encapsulate them in a special object. The data access objects (DAO) design pattern was created for just this purpose.

A.2.2 Creating the DAO methods

Data access objects (DAO) is a core design pattern in Java EE, providing an interface between the controller and model layers. The purpose of the DAO pattern is to locate all the code dealing with Hibernate persistence in one place, and to free business objects from data persistence issues. In general, a DAO is created for each class. Since our DAO works with the `Product` class, it is called `ProductDAO`. For easy access, it is implemented as a global singleton service. Before any objects can be made persistent, a DAO instance is obtained with the `getInstance` method:

```
public class ProductDAO {
  private static ProductDAO instance = null;
  private ProductDAO() { }      ❶

  public static synchronized ProductDAO getInstance() {
     if (instance == null) {
        instance = new ProductDAO(); }      ❷
     return instance; } }
```

The `private` access modifier ❶ precludes this constructor from being invoked outside this class. If there is no instance, then a `ProductDAO` object is instantiated ❷. Otherwise, the instance is returned. This ensures that there is only a single instance of this object.

But before this `ProductDAO` class can perform any persistence operations, Hibernate needs to perform runtime initialization for the class.

Runtime initialization for Hibernate

Every Hibernate request results in the creation and destruction of a *session*. To allow `session` objects to be inexpensively created and destroyed, we build an instance of the `SessionFactory` class. But first we need a `Configuration` instance.

A `Configuration` object contains all the classes to be persisted by Hibernate. Configuring a class to be persisted requires that we add this class to the `Configuration` declaration through its `addClass` method. Multiple classes can be declared by concatenating their declarations, like this:

```
Configuration cfg = new Configuration().addClass(apple.class)
                                       .addClass(berry.class)
                                       .addClass(cherry.class);
```

The purpose of the `Configuration` object is to build a `SessionFactory`, which must be provided for each database. Since we have only a MySQL database, only a single `SessionFactory` object is needed:

```
private static ConnectionFactory instance = null;
private SessionFactory sessionFactory = null;

private ConnectionFactory() {
   try {Configuration cfg = new
      Configuration().addClass(Product.class);
      sessionFactory =
           cfg.buildSessionFactory(); }
    catch (MappingException e) {
      log.warn("Mapping Exception" + e.getMessage());
      throw new RuntimeException(e); }
    catch (HibernateException e) {
      log.warn("Hibernate Exception" + e.getMessage());
      throw new RuntimeException(e); } }
```

Establishes sessionFactory

The `sessionFactory` needs to be built only once and should be easily accessible. Consequently, it is also executed within the `ConnectionFactory` constructor that is implemented as a singleton:

```
public static synchronized ConnectionFactory
       getInstance() {      ❶
   if (instance == null) {
      instance =
           new ConnectionFactory(); }      ❷
   return instance; }

public Session getSession() {
   try {
       Session s =
           sessionFactory.openSession();      ❸
      return s; }
   catch (HibernateException e) {
       log.warn("Hibernate Exception" + e.getMessage());
       throw new RuntimeException(e); } }
```

In the first line, `getInstance` ❶ returns the instance of the `ConnectionFactory` singleton. If an instance of the singleton ❷ has not been created, one is created and returned. At ❸ the Hibernate `sessionFactory` returns a `session` to the caller.

All operations in Hibernate start with a `session`. Before we can do anything, we must retrieve an open `session` through the `getInstance` singleton method:

```
Session session = ConnectionFactory.getInstance().getSession();
```

Now that Hibernate's runtime configuration is complete, we're ready to implement CRUD functionality with our `DAOProduct` class.

A.2.3 Building our ProductDAO Object

We'll use the same coding outline for all our DAO operations. All Hibernate operations must be enclosed within a set of `try` and `catch` brackets, to capture Hibernate exceptions. Whenever a `HibernateException` occurs, the error message is directed to a system log and an uncatchable `RuntimeException` is thrown to propagate the exception up one level, where a more contextually specific error message can be generated. When the Hibernate operation has successfully completed, a `finally` clause ensures that the Hibernate `session` is closed. Failure to close the `session` correctly causes a Hibernate exception to be generated, which is handled as before.

Listing A.4 shows a complete example. In subsequent examples, we provide only an abridged version of the Hibernate calls.

Adding a product

Since a `Product` object is transient, its data would be lost if the server shut down. To make this data persistent, the `Product` object is passed to the Hibernate `session`, which generates SQL statements to save it. The Hibernate `session` uses transparent *write-behind*, which combines many changes into a smaller number of database requests, thus increasing efficiency. This also generates the SQL statements asynchronously. During development, we prefer to see the generated SQL statements, so a `flush` statement is added to force their immediate generation. These `flush` statements can easily be commented out later.

Listing A.4 Adding a `Product` with Hibernate

```
public void addProduct(Product product) {
   Session session = ConnectionFactory.
                getInstance().getSession();
   try {
       session.save(product);
       session.flush(); }
   catch (HibernateException e) {
       log.warn("Hibernate Exception" + e.getMessage());
       throw new RuntimeException(e); }
   finally {
      if (session != null) {
          try {
             session.close(); }
          catch (HibernateException e) {
             log.warn("Hibernate Exception" + e.getMessage());
              throw new RuntimeException(e); } } } }
```

Gets open session

Persists transient instance

Synchronizes persistent store with memory

Logs warning, throws runtime exception

Ends session

Next, we'll look at implementing each of the DAO operations for products.

Listing products

Hibernate features HQL, an object-oriented variant of the relational query language SQL, to simplify object retrieval. Queries are created with HQL to retrieve objects of a particular class. Selections can be filtered by adding a `where` clause that restricts selections to match certain class properties. To list a single `Product`, we retrieve it by a matching `id` value. This is a multistep procedure; the first step creates the query, then the identifier value is bound to a named parameter, and, finally, the result is returned. An advantage of using named parameters is that it is not necessary to know the numeric index of each parameter:

```
public List getProduct(Product product) {
   ...
   try {
       Query query = session.createQuery(
                   "select from Product i where i.id = :id");
       query.setParameter("id", product.getId(), Hibernate.LONG);
       return query.list(); }
   ... }
```

This `select` query returns all the products. When multiple products are listed, they should be ordered by a field. Here they are ordered by the `name` field:

```
public List getProducts(Product product) {
   ...
   try {
       Query query =
             session.createQuery("from Product i order by i.name");
       return query.list(); }
   ... }
```

In both examples, a list of the results is returned. There is no need for a `flush` statement, since results are being read from the database. The `Action` is responsible for storing these results in a list-based object, so it can be iterated for display in the view.

Updating a product

The `update` method updates the persistent state of the object in the database with the contents of the transient object. Hibernate executes a SQL `UPDATE` to perform this operation:

```
public void updateProduct(Product product) {
   ...
   try {
      session.update(product);
      session.flush (); }
   ... }
```

This results in the database record corresponding to this object being updated with the contents of its fields.

Deleting a product

The `get` method is used to retrieve a `Product` instance by its identifier. Once we have the `Product` instance, it is passed to the `delete` method to schedule a SQL `delete` operation:

```
public void deleteProduct(Product product) {
 ...
 try {
     Product product = (Product) session.get(Product.class, id);
     session.delete (product);
     session.flush(); }
 ... }
```

Now that we've defined all of the DAO methods, we can begin putting all of the pieces together within the `Action` controller. The values returned from the DAO methods are used to update the request, which is later accessed by the view's JSP pages for display.

A.2.4 Using the ProductDAO with our ProductAction

The `Action` controller performs some setup operations before beginning its main task of routing the `action` parameter to invoke the appropriate DAO method. If the DAO returns a value, the request object must be updated with this value. This makes these values available to the JSP files for display within an XML file. Finally, the `ActionForward` is set to cause the invocation of a JSP file to display this information. Listing A.5 shows the `Action` controller.

Listing A.5 Updated `ProductAction` using the `ProductDAO`

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
    HttpServletRequest request, HttpServletResponse response)
    throws Exception {
 ActionErrors errors = new ActionErrors();
 ProductDAO dao = ProductDAO.getInstance();

 String action =
       request.getParameter("action");
 Product product = new Product();
 BeanUtils.copyProperties(product, form);
 try {
    if ("add".equals(action)) {
       dao.addProduct(product);
       return mapping.findForward(
                  "status"); }
```

Gets DAO instance

Determines CRUD operation

Copies parameters to product object

Adds product

```
   else if ("delete".equals(action)) {
      dao.deleteProduct(product);
      return mapping.findForward(
                "status"); }
   else if ("update".equals(action)) {
      dao.updateProduct(product);
      return mapping.findForward(
                 "status"); }
   else if ("get".equals(action)) {
      List products = new ArrayList();
      products.add(
                dao.getProducts(product));
      request.setAttribute(
                "products", products);
      return mapping.findForward(
                "display_products"); }
   else if (action == null ||
               "list".equals(action)) {
      List products =
                dao.getProducts(product);
      request.setAttribute(
                "products", products);
      return mapping.findForward(
                "display_products"); } }
catch (Exception e) {
   log.warn("productAction exception: " + e.getMessage());
   errors.add(ActionErrors.GLOBAL_ERROR,
             new ActionError("errors.invalidDBOp"));
   saveErrors(request, errors);
   return mapping.findForward("failure"); }
return mapping.findForward("success"); } }
```

Deletes product

Updates product

Gets product

Lists all products

In each case, after calling the `ProductDAO` to complete each CRUD-related operation `ActionForward` directs the `Action` controller to display a JSP page. This page contains either a status or the `product` listing.

Now the only task left is to write the JSP files to read the information stored in the `request` and `product` objects and create a response containing an XML document.

A.3 Outputting XML content

Laszlo requires that its HTTP response contain an XML document. This XML output can be easily created by setting the response's MIME type to `text/xml`. We do this by updating the page directive's `contentType` attribute within the JSP file:

```
<%@ page contentType="text/xml" %>
```

When returning a successful status, a JSP can simply contain a hard-coded XML response like this:

```
<%@ page contentType="text/xml" %>
<result>
   <status>SUCCESS</status>
</result>
```

When `products` are returned by the product.jsp file, the Struts `iterate` tag is used to display both single and multiple product listings:

```
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>
<%@ page contentType="text/xml" %>
<response>
   <products>
      <logic:iterate id="product"                          Iterates over
            name="products"                                products array
            scope="request"
            type ="com.laszlomarket.Product">
         <product sku="<bean:write name='product' property='sku'/>"
                  id="<bean:write name='product' property='id'/>"
                  title="<bean:write name='product'
                                     property='title'/>"
                  price="<bean:write name='product'
                                     property='price'/>"
                  image="<bean:write name='product'
                                     property='image'/>">
                  video="<bean:write name='product'
                                     property='video'/>">
            <description><![CDATA[
               <bean:write name="product" property="description"/>
            ]]></description>
            <outline><![CDATA[
               <bean:write name="product" property="outline"/>
            ]]></outline>
            <specs><![CDATA[
               <bean:write name="product" property="specs"/>
            ]]></specs>
         </product>
      </logic:iterate>
   </products>
</response>
```

When the following code is executed, it produces the identical XML output to the sample data used in our local dataset, but now it's wrapped within a `response` node:

```
<response>
   <products>
      <product sku="SKU-001" title="The Unfolding"
         price="3.99" image="dvd/unfold.png" id="1"
         video="video/unfold.flv" category="new">
         <description><![CDATA[A man calling himself "Simon"
            ...]]>
         </description>
         <outline><![CDATA[This film deals with
            ...]]>
         </outline>
         <specs><![CDATA[<p>Regional Code: 2 (Japan, Europe,
            ...</p>]]>
         </specs>
      </product>
      <product sku="SKU-002" ...>
         ...
      </product>
   </products>
</response>
```

We've now completed our Struts web application to support the product listing as required by the Laszlo Market application. Chapter 16 contains further information on how a Laszlo application can generate an HTTP request to invoke this web service and how it will process the returned response.