Best Practices with Expresso Framework Using Struts

Peter Pilgrim

Published on TheServerSide.com

February 25, 2002

Over the last two years or so, server-side Java technology has surpassed the popularity of client-side Java. Many developers started way back in the day with simple Applets, which are embedded Java programs inside a browser; then, Sun invented Java Servlets. The technology moved from the HTTP client side to the HTTP server side. Java Servlets were interesting but too cumbersome for professional web designers to create content with: they could not easily get a look inside. Thus, Java Server Pages were created to turn Servlet technology inside out. JSP solved the creative presentation problem, but introduced architectural building problems.

As far as I am concerned, the 'war of MVC' has been won in the last two years. After the arrival of Java Server Pages technology, we have seen a great leap forward in the overall architectural design of web applications. The page-centric model has proven adequate for the most basic web applications, but as soon as developers have to write more complex business applications, the design falters because the Java Server Pages are heavily coupled with each other. Of course this severely handicaps our ability to maintain software. Consequently, many developers have adopted a pattern from Smalltalk programming, the Model View Controller. We now had a choice, learn to use someone else's MVC framework or develop our own. Do we really want to write our framework from scratch? Does any one out there still have a lot of time of their hands?

One of the features of Java that originally attracted me was the abundance of libraries or APIs contained within the actual software development kit itself. It meant that as a Java developer, you didn't have to reinvent the wheel. You did not have to write your own Vector class or your own Graphics event handling. It was all there, of course, in the SDK. For web applications, until recently, there were no third-party libraries or API implementing Model View Controllers. There are now many such toolkits or frameworks, including:

- Barracuda
- Tapestry

- Enhydra XMLC
- Expresso
- Jakarta Struts

Expresso Framework, until version 3, recently had its own Model View Controller engine. However, during year 2001 a clear overall winner in the war of MVC had been declared: Jakarta Struts. The popularity of Jakarta Struts is evident by its appearance and description in several articles and at least two well-known books. There are also many people subscribed to the struts-user mailing lists, which seem to be as over-subscribed to as the JSP-interest mailing list. By adopting Struts, developers reduce the amount of code that needs to be written for a web application and since The Expresso Framework integrates Struts, it further reduces the amount of code that developers need to write. As well as having an MVC framework, Struts also has its own powerful custom tag actions. Custom tag libraries and their powerful bean introspection core further reduce the total amount of Java scriptlets in Java Server Pages.

The current economic climate is very weak and the field of information technology is suffering from widespread budget cuts and declining confidence in e-commerce. This invariably means that developers have far less time and resources to complete their projects. Unfortunately, the ready-made web application is still just beyond on the horizon. Fortunately, frameworks have been written which solve the denigrating aspects of writing web applications, such as handling user login profiles, security, MVC, and database object mapping. Each framework is obviously different and each has it's own strengths and weaknesses. There is no standard available.

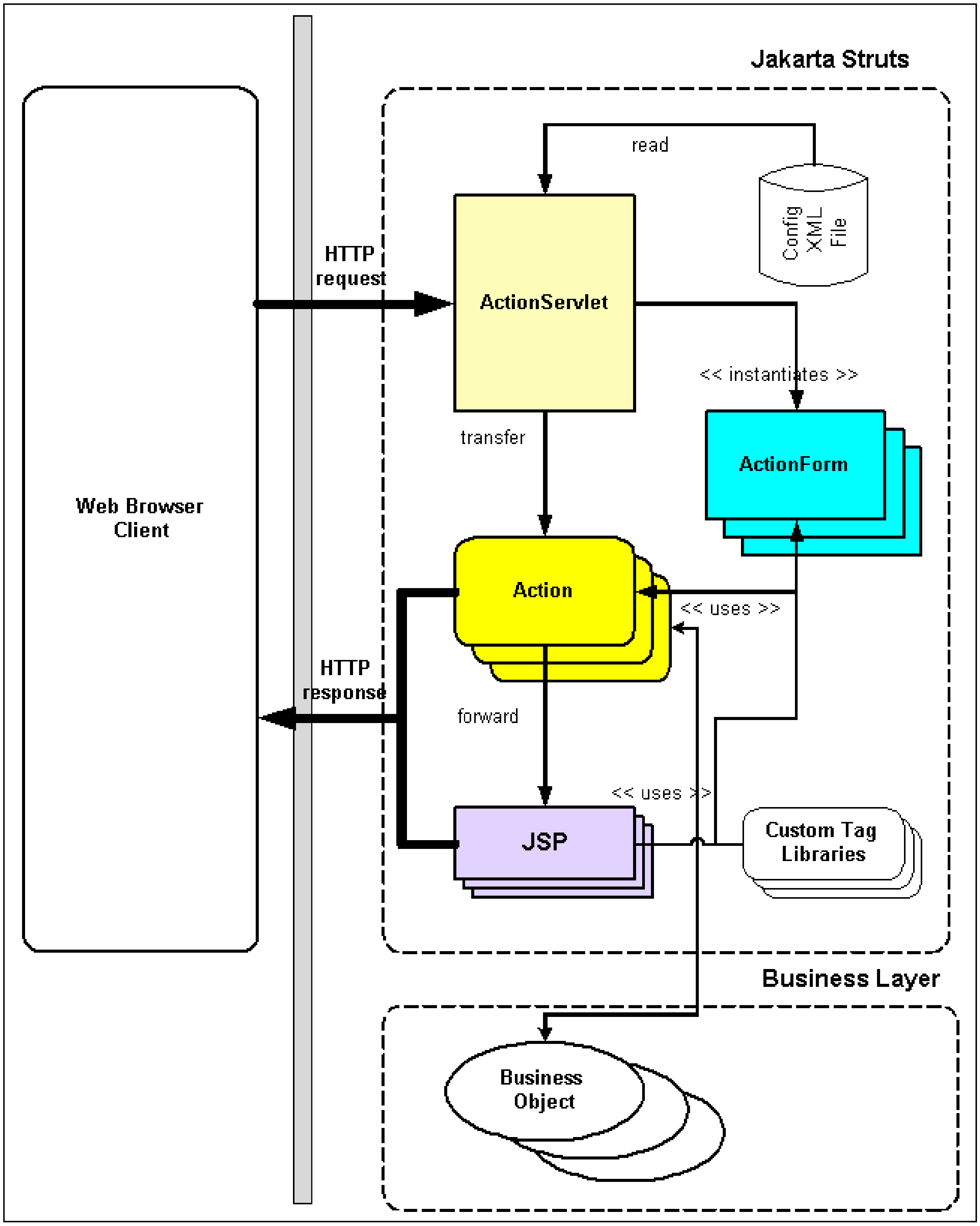
Expresso is one such framework from JCorporate. I do not work for JCorporate, or have any sort of interests with them whatsoever apart from using and developing with their open-source software. Neither am I here to start a war of words, infractions, over which one of these toolkits *is* the penultimate framework. You can find the various attractors and detractors for each one elsewhere such as in the archives of the heavily subscribed mailing list jsp-interest@java.sun.com.

Struts

Jakarta Struts is a lightweight framework that implements the Model View Controller pattern. Pure Struts [not integrated with Expresso] has, at its centre, its own action controller servlet that listens to requests from the user's browser and passes them on to special components called Actions. Each `Action` represents a specific function of the web application. An Action is associated usually with a set of forward paths and form beans. This association is called an `ActionMapping`, and thus a forward path is also known as an `ActionForward`. An action processes the incoming HTTP servlet request and then either generates the HTTP servlet response or it names a suitable Struts action forward to pass the request onto. Inside the action is where you will write the business logic of the application. The advantage of the model is that it clearly separates the presentation rules from the business rules of the task. You can make subtle changes to the overall look-and-feel of the web front end of the application, without having to alter a single line of code that manages the business logic. Conversely, you can change the business logic without affecting the presentation adversely, for example, if you change the application’s database-design. Dividing the architecture into separate tightly coupled, but less cohesive tiers is one of the goals of good software practice. I said that Struts was lightweight because it only supplies the MVC framework and the custom tag actions in the overall API. Jakarta Struts does not supply any additional features, such as user profile security, EJB, messaging, or an object to relational database mapping.

Struts uses the concept of commands associated with a Java Beans, Struts Action. All contemporary web applications provide some type of input using HTML Form elements. Struts buffers input from HTML forms in Java Beans known as `ActionForm` beans. When Struts transfers processing to an `Action`, if the action was not created already, it will cleverly instantiate the action. If the action has a form bean associated with it, it will also instantiate the form bean, and populate the attributes with values from any matching servlet request parameters. (This feature is not unlike the standard JSP action `<jsp:property name=”acmebean” property=”*” />`.)

The Java Servlet API already allows you to forward to another JSP or to include another JSP. But the problem with this is that programmers have generally hard-coded URLs into their programs, strongly binding the Servlet to a JSP. This is not what you want to do really, because everyone knows that such URLs are hard to maintain and support. Struts alleviates this problem. It offers the concept of an `ActionForward`, allowing you to forward dispatch to a *logical* name instead. A forward is essentially a logical name and URL path association.



Jakarta Struts is configured from an XML file and that is the genial concept of Struts. Action mappings have logical names and so do action forwards. This means that presentation can be separated from the implementation code. In fact, action form beans also have logical names. Action forwards can be local to the action mapping or be global to the web application. The design of Struts is the secret of its power. Enough of the theory let us now see the practice by just looking at an XML file:

```
<struts-config>
<form-beans>
    <form-bean name="loginForm"
               type="com.xenonsoft.jsp.beans.LoginForm " />
    <form-bean name="dvdOrderForm"
               type="com.xenonsoft.jsp.beans.DVDOrderForm " />
      …
</form-beans>
<global-forwards>
     <forward name="home"           path="/index.jsp" />
     <forward name="success"        path="/hello.jsp" />
     <forward name="failure"        path="/error.jsp" />
     <forward name="login"          path="/login.jsp" />
     <forward name="logout"         path="/Logout.do?action=confirmLogout" />
      …
</global-forwards>
<action-mappings>
    <action   path="/Login"
              type="com.xenonsoft.jsp.actions.Login"
              name="loginForm"     scope="request"
              validate="false" >
      <forward name="success"       path="/index.jsp"/>
    </action>
    <action   path="/DVDOrder"
              type="com.xenonsoft.jsp.actions.DVDOrder"
              name="dvdOrderForm"  scope="request"
              validate="false" >
      <forward name="promptForm"    path="/prompt-DvdOrder.jsp"/>
      <forward name="completed"     path="/completed-DvdOrder.jsp"/>
      …
    </action>
</action-mappings>
</struts-config>
```

This is a whirlwind tour of Struts. For more information of the configuration details of the preceding XML, please visit the Jakarta site directly.

Those of you who have an interest in software patterns will recognize that Jakarta Struts is based on the `COMMAND` delegation pattern, which is a special variation of the `CHAIN OF RESPONSIBILITY` pattern. Because the purpose of the command pattern is to keep the presentation front-end separate from the actions that they initiate, this makes Struts very useful for business web applications. The master switch is, of course, the Struts `ActionServlet`, which serves as a controller, which in turn invokes the referenced action mapping class. Additionally, Struts also supplies some generous in view helper classes; these and custom action tags libraries are great for JSP development. Coverage of the tag libraries is outside the of scope of this article.

For a feel of the Struts development process, it might help to look at some real simple code. Let us quickly look at a typical `ActionForm`, which is really just a simple Java Bean that knows how to reset itself and perform validation.

```
import org.apache.struts.action.*;
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DVDOrderForm extend ActionForm {
    protected String    action="", productId="", productName="";
    protected int quantity=0;

    public DVDOrderForm() { super(); }

    public String getAction() { return action; }
    public void   setAction( String a ) { this.action = a; }
    public String getProductId() { return productId; }
    public void   setProductId( String val ) { this.productId = val; }

    ... // more accessor and mutators

    public void reset( ActionMapping mapping, HttpServletRequest request )
    {
      super.reset( mapping, request );
      action=""; productId=""; productName=""; quantity=0;
    }

    public ActionErrors validate( ActionMapping mapping, HttpServletRequest
request)
    {
      ActionErrors errors = new ActionErrors();
      if ( quantity < 1 )
          errors.add( ActionErrors.GLOBAL_ERROR,
                  new ActionError("error.form.field.moreThanZero",
                              "Quantity" ) );
      ... // more form input checking

      return errors;
    }
}
```

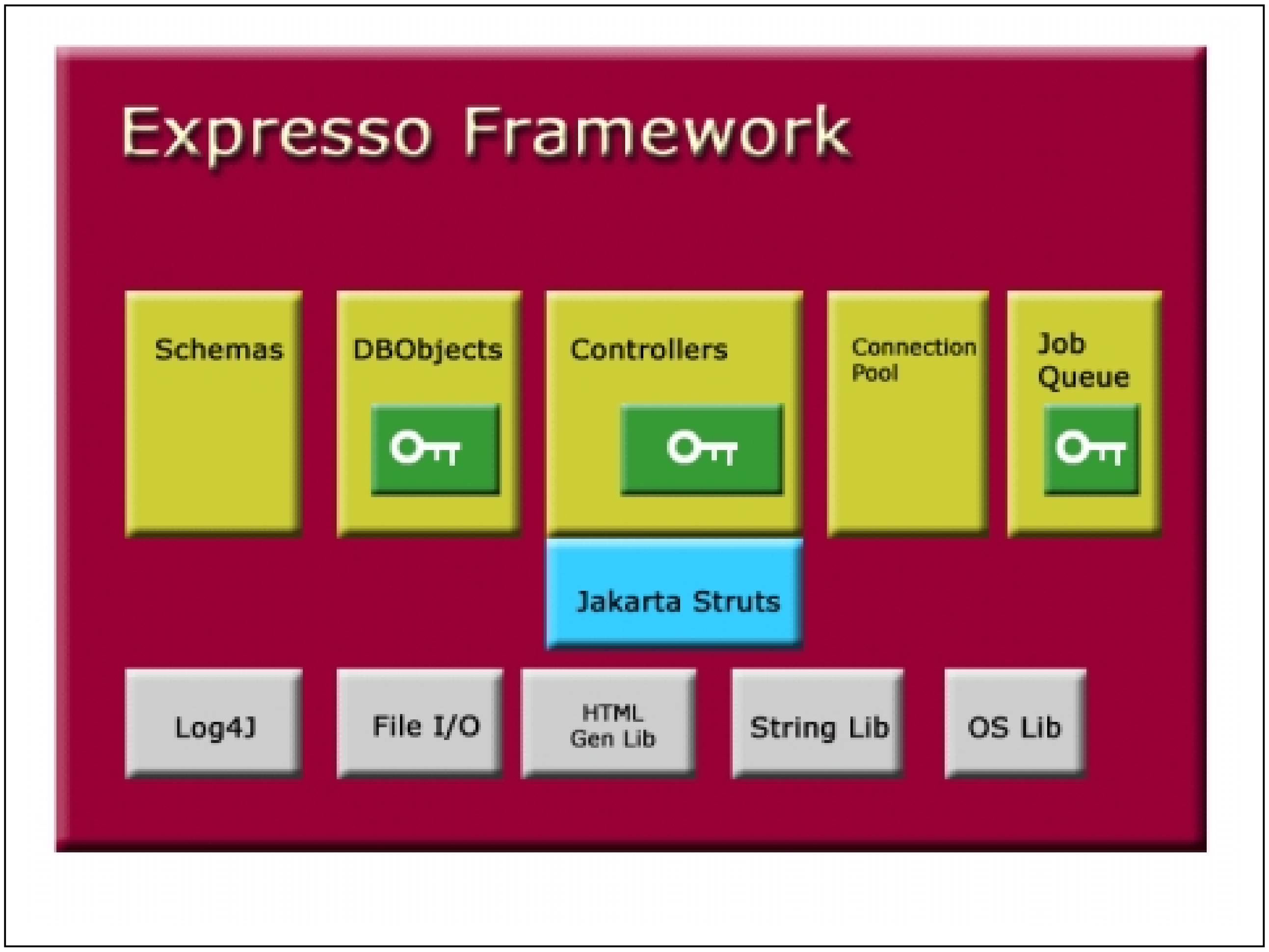
Struts instantiates the action form when the action mapping is first created for processing a request. It will call the `reset()` method to return the form to the default settings. There is also a `validate()` method, where each form attribute can be individually checked. Because Struts supports internationalisation using resource bundle properties, the error messages can be returned in the appropriate locale and territory. After populating the `ActionForm` bean, Struts passes control to the action mapping. Here is an example of a proper `Action`.

```
import org.apache.struts.action.*;
import org.apache.struts.util.*;
import java.io.*;
import java.math.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DVDOrder extends Action {
    public ActionForward perform(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException
    {
      Locale locale = getLocale(request);
      MessageResources messages = getResources();
      ActionErrors errors = new ActionErrors();
      DVDOrderForm orderForm = (DVDOrderForm)form;

      if ( action.equalsIgnoreCase("promptForm") ) {
          // Set other form attribute here.
          BigInteger price = businessLayer.getPrice(
            "The Shawshank Redemption");
          orderForm.setPrice( price );
          return mapping.findForward("promptForm");
      }
      else if ( action.equalsIgnoreCase("orderDVD") ) {
          businessLayer.orderDVD(
            orderForm.getProductId(),
            orderForm.getQuantity() );
          return mapping.findForward("completed");
      }
      else {
          errors.add( ActionErrors.GLOBAL_ERROR,
            new ActionError( "error.moarAction.action.unknown",
            "DVDOrder", action ));
          return mapping.findForward("failure");
      }
    }
}
```

Although the example is slightly contrived, it illustrates the existence of a business layer tier. The action mapping is, indeed, where the integration of the two tiers, takes place. Business processing such as calculating the tax and the surplus charge (or seasonal discount) for a quantity of DVDs should really take place in the business layer.. Also, pay notice to the fact that within pure Struts, you must *multiplex* your own *action commands*. Because Struts is a lightweight framework you have to plumb your own pipes, so to speak, in order to handle different states in the function of ordering a DVD: “promptForm”, “completed” and “failure”. As we will see, Expresso can help out substantially here and save real time and energy. All action components return either an `ActionForward` or a `null`, which signifies that the action is returning its own custom HTTP response (e.g. binary content like a GIF image or PDF file.) Action forwards can be local to the Action, global to the configuration file, or created dynamically in the action. For the benefit of illustration, I have left out the exception handling that is required for a proper business application.

Expresso Framework

Expresso Framework 4.0 is a large toolkit of classes. Getting started can be overwhelming because there are so many classes [449 classes at last count] to look at. The frameworks can be divided technically into four essential parts: schemas, database object persistence, controller design, and the rest of the kit, which involves job control, utilities, etc. It is essential that you understand the first three parts. Expresso Framework is also very much database-driven. That is, you will need to use a reliable, scalable relational database to make full use of the kit. Mysql, Sybase, Oracle, DB2, and Postgresql are all suitable candidates. I will demonstrate the best practice by describing a simple stock trading program that displays the stock trades, and allows the user to buy and sell the stocks.



Schemas

Every Expresso application must have a schema. A schema is a bit of a misnomer, because it could really be called a *Web Application Context* object. (If you search this document for the specific word *application*, it will appear at least a dozen times. Perhaps this explains why they chose to name the class something else.) The schema object defines all the database objects: controllers, servlets, job handlers, and the user-defined preferences that are used by your custom web application. The schema is a placeholder; it uniquely defines the identity of your web application. It describes the context of your web-enabled program. It also allows the Expresso core to administer a strict security matrix, in conjunction with the relational database, to prevent unauthorised access to your privileged program.

Let us now create a schema for our on-line stock trader, which is called `OrangeTrader`, simply because that is what the creative design team at said company came up with in the marketing department.

Expresso has around 449 classes spread over 75 directories. Here are my UNIX commands:

```
% cd

/opt/tomcat4.0/expresso/WEB-
INF/classes

% find com/jcorporate/expresso -
type f –name “*.java” -print | wc
–l

% find com/jcorporate/expresso -
type d -print | wc -l
```

```
import com.jcorporate.expresso.core.db.*;
import com.jcorporate.expresso.core.dbobj.Schema;
public class OrangeTraderSchema extends Schema
{
    public OrangeTraderSchema()    throws DBException
    {
      super();
      try {
          addDBObject( "com.xenonsoft.orangetrader.ot1.dbobj.StockTrade"  );
          addController(
"com.xenonsoft.orangetrader.ot1.controller.StockController" );
      }
      catch ( Throwable t ) {
          throw new DBException( t.getMessage() );
      }
    }

    public String getMessageBundlePath() {
      return "com/xenonsoft/orangetrader";
    }
}
```

This simple schema object has a default constructor that registers one single database object and one controller that we use for the orange trader program. The method `getMessageBundlePath`() returns the search path for a resource bundle, which later allows us to internationalise our application.

Database Objects

For the on-line trading application, we create a persistent data object, which maps directly to a table in our relational database. In Expresso, the object class `DBObject` provides the means to map a Java object to a relational database. If your object extends this class, then it can write and read its data persistently. Expresso provides another direct subclass of `DBObject` called `SecuredDBObject` that secures a database object against a user profile. A secured database object can be restricted to a group of users, or a single user. Database objects that extend `SecuredDBObject` are automatically secured against a user profile with no further intervention required by the developer.

The stock trade program requires a simple table to hold the details of each stock, which are the stock's name, its current price, asking price, bidding price, and the percentage change. The `StockTrade` object (see below), sets this up in a method called `setupFields`(). It is pretty self-explanatory. The `DBObject` requires a database table name which is provided by the call to `setTargetTable`(), a human-readable description provided by `setDescription`(). The names of the fields are defined by the `addField`() calls. The `addField`() method accepts for five parameters: the database column name, the scalar length of the field, the database type of the field, a Boolean that declares whether the column accepts `null` values or not, and a description. Finally, a field must be chosen to be the primary key of the table with the call to `addKey`(). As you can see, field names correspond directly to the column names of a database table.

All `DBObjects` have essential methods, `add()`, `update()`, `delete()` and a number of various search and retrieval methods. The `DBObject` class internally provides a hash table of fields and values, which you can access by using accessor and mutator methods. If you want to get the value of a particular column in the row set use the method "`String getField( String fieldName )`". If you want to update a particular column, use the method "`void setField( String fieldName, String newValue )`".

```
import com.jcorporate.expresso.core.dbobj.*;
import com.jcorporate.expresso.core.db.*;

public class StockTrade extends SecuredDBObject
{
    public StockTrade()      throws DBException
    {      super();  }

    public StockTrade(DBConnection theConnection) throws DBException
    {      super(theConnection); }

    protected synchronized void setupFields()
      throws DBException
    {
      setTargetTable("STOCKTRADE1");
      setDescription("Stock Trade I");
      addField("ST_ID", "int",       0, false, "Text Channel Autoincrement ID");
      addField("ST_TITLE",    "varchar",  40, false, "Stock Title");
      addField("ST_TRADER",   "varchar",  40, false, "Owner Trader");
      addField("ST_STATUS",   "varchar",  10, true,  "Trade Status" );
      addField("ST_PRICE",    "float",     0, true,  "Current Price" );
      addField("ST_ASK",      "float",     0, true,  "Asking Price" );
      addField("ST_BID",      "float",     0, true,  "Bid Price" );
      addField("ST_CHANGE",   "float",     0, true,  "Percentage Change" );

      addKey("ST_ID");
    }

    public DBObject getThisDBObj() throws DBException {
      return new StockTrade();    }
}
```

Whenever you write a `DBObject` subclass you *must* write the default constructor and make sure that it calls the super class constructor. At the very least, the constructor that accepts a `DBConnection` object must also be defined. A special method call `getThisDBObj()` should also be overridden. This method should simply return a new instance of the particular database object class. It is called internally by Expresso. (There are actually two more constructors deemed worthy to include, and a way of pre-populating a database table when it is created, but I have left those out to save space. See the software source code for more details.)

With this definition of the `StockTrade` object class, Expresso will be able to automatically generate the SQL statement for this database object that goes something along this line:

```
create table STOCKTRADE1 (
      ST_ID  integer not null, ST_TITLE varchar(40) not null,
      ST_TRADER varchar(40) not null, ST_STATUS varchar(10),
      ST_PRICE numeric, ST_ASK numeric, ST_BID numeric, ST_CHANGE numeric,
      PRIMARY KEY (ST_ID) )
```

I should say that the `DBObject` should not be confused with a Struts `ActionForm`. They are completely different entities. Expresso does not populate or create database objects automatically from an Http servlet request. Rather, you must create the database object from inside your Expresso controllers. In fact, Expresso supplements the use of Action forms; the toolkit has its own input, output, and transition methodology, as you will see later in the next section.

Controllers

The most confusing aspect of using Expresso Framework that developers find difficult is writing their own controllers. A controller in Expresso is much like the controller in MVC, but typically acts as both the model and controller. This confuses a lot of computer science theoreticians because they expect the controller to handle only the routing and a separate model object to handle the business logic, in exactly the way they have been taught in school. Also, many developers who stumble across the framework have not previously had any experience working with any frameworks at all. Most developers come in from the cold where they learnt directly from the *Sun Pet Store Blueprint* example. This famous example, particularly uses raw JSP, Servlets, and context path redirection. I guess what I am trying to say is that using a framework for the first time feels, well, alien.

Forthcoming Struts Features
The custom tag actions of the Struts 1.0 release supports *nested bean properties*. This powerful feature allows you to access delegated business objects in JSP using a simple dot (.) and array syntax notation. For example let's suppose we have a shopping cart that stores a delegated product and we need to get the name of the third product. We could write
"`cart.product[2].name`" which translates to Java "`getCart().getProduct(2).getName()`".

The Struts nightly build (alpha code) has several new features. One of relevance to Expresso is mapped properties. This enables a DBObject to be populated from a JSP using a new syntax. Given the follow bean properties.

```
Object getField( String key )
setField( String key, Object value )
```

The mapped property can be accessed with the syntax "`field(key)`". This is almost similar to the DBObject methods, except Strings are returned instead of Objects. Even more profound changes are expected with the JSP Standard Tag Library release 1.0.

An Expresso controller is the point of access that allows the application to move from the current state to the next state. It is a general concept known as the *Finite State Machine*. Indeed, any web application can be thought of as a complex FSM. If you think of the workflow for withdrawing money from a bank automatic teller machine, then you see the idea. Some person inserts a cash card, enters their secret personal identity number, thereafter verification takes place. If the person is authenticated then he or she must be a true customer. If the PIN does not verify, then the card is rejected or held by the teller. The authenticated customer chooses withdrawal from a menu. The customer enters the amount. The back-end system verifies that the amount does not exceed the current account or some other predefined limit. If all goes well, the cash is dispensed, the transaction is committed, the audit is written, and the teller ejects the customer's card. The teller resets itself and returns to the welcome page screen ready for the next customer. The controller is the thing that manages the process of control. Moreover, the complex FSM can be simplified. FSM can be built from a set of building block FSM's. You would not write a full-blown auto teller application inside one massive controller; rather, each important aspect of it would make a suitable controller operation. In a nutshell the Expresso controller manages the workflow or a subset of the total workflow of the application.

Expresso defines a `Controller` object class, which is a direct subclass of the Struts `Action`. It is, essentially, a collection of states as alluded to above. The basic `Controller` object is unsecured, but just like the `SecuredDBObject`, there is a secured version available called `DBController`, a direct extension of the `Controller` class which provides user profile security for a single user or a group of users. Also, `DBController` permits none, certain states, or all states, to be secured against a user profile. Now that we have the `StockTrade` database object, it is time to write a `StockController` object to manipulate it.

```
import com.jcorporate.expresso.core.controller.*;
import com.jcorporate.expresso.core.controller.session.*;
import com.jcorporate.expresso.core.dbobj.*;
import com.jcorporate.expresso.core.db.*;
import com.xenonsoft.orangetrader.ot1.dbobj.StockTrade;

public class StockController extends DBController
{
    public StockController()
    {
      super();
      addState( new State( "displayStocks", "displayStocks"));
      addState( new State( "promptBuyStock", "Prompt Buy Stock"));
      addState( new State( "promptSellStock", "Prompt Sell Stock"));
      addState( new State( "buyStock", "Buy Stock"));
      addState( new State( "sellStock", "Sell Stock"));
   }

    public String getTitle() {
      return "Stock Controller II";
    }

    // MORE CODE…
}
```

The `StockController` has a default constructor that calls the superclass default constructor, as always. It registers all of the states that the controller is going to manipulate. In contrast to pure Struts `Actions`, the latest Expresso 4.0 release handles transfer to request to different states automatically. You can, however, choose to override the mapping mechanism. Notice that there is a call to the superclass method in the first line of the default constructor. This is a requirement for security profiling. Trust me, you need it! The method `getTitle`() simply returns a readable string that names this controller. Expresso has the capability to manipulate controllers directly and generate default views of the input and outputs; this is because the toolkit has its own HTML generation package, which is similar to, but less extensive than, the Jakarta Element Construction Set. This mode of output is the default style for a controller. You can set the style to a refer to a JSP, in which case you decide how to handle the controller response. A controller can be told to generate XML content if the mode style is setting XML.

All Expresso controllers can generate responses for the purpose of form generation, or simple presentation, for example, in a JSP, as I mentioned above. There are four fundamental object classes that all immediately derive from a special `ControllerElement` abstract class.

- `Input` class is an element that a controller creates when it expects some input. It is usually an HTML Form element tag such as a text input field, but can be a drop-down select list or a set of radio boxes.
- `Output` class is an element that a controller creates just to provide output.
- `Transition` class is an element that a controller creates to cause a transition from one state to another. In our on-line stock trader application this will literally be “Buy Stock” and “Sell Stock”. This is often represented as an HTML Submit, Button, Image element in a JSP, but it could also be a complicated set of JavaScript and image maps on the client side.
- `Block` class is an element that aggregates the other three controller element types into a structured response.

Displaying Stock Trades

Let us look at the first state, the method `displayStocks`(), which reads all the stocks from the database table using the database object `StockTrade`.

```
    protected ControllerResponse runDisplayStocksState(
      ControllerRequest myRequest, ControllerResponse myResponse )
      throws ControllerException
    {
      try {
          Block blockList = new Block("StockList" );
          StockTrade stockInit = new StockTrade();
          stockInit.setDBName( myRequest.getDBName() );
          ArrayList list = stockInit.searchAndRetrieveList( "ST_TITLE" );
          for ( int k=0; k<list.size(); ++k ) {
            StockTrade stock = (StockTrade)list.get(k);
            // Create a Block for each row tuple
            Block blockRow = new Block("Stock"+k );
            blockList.add( blockRow );
            // Create an Output for the DBObject row tuple
            Output out1 = new Output();
            out1.setName( "Detail" );
            out1.setAttribute( "Title",  stock.getField("ST_TITLE") );
            out1.setAttribute( "Trader", stock.getField("ST_TRADER") );
            out1.setAttribute( "Price",  stock.getField("ST_PRICE") );
            out1.setAttribute( "Ask",    stock.getField("ST_ASK") );
            out1.setAttribute( "Bid",    stock.getField("ST_BID") );
            out1.setAttribute( "Change", stock.getField("ST_CHANGE") );
            out1.setAttribute( "Status", stock.getField("ST_STATUS") );
            blockRow.add( out1 );

            Transition buyTrx=new Transition("Buy", getClass().getName() );
            buyTrx.setName( "Buy" );
            buyTrx.addParam("state", "promptBuyStock");
            buyTrx.addParam("stock", stock.getField("ST_ID") );
            blockRow.add(buyTrx);

            Transition sellTrx=new Transition("Sell",getClass().getName() );
            sellTrx.setName( "Sell" );
            sellTrx.addParam("state", "promptSellStock");
            sellTrx.addParam("stock", stock.getField("ST_ID") );
            blockRow.add(sellTrx);
          }
          myResponse.addBlock( blockList );
      }
      catch (DBException dbe) {
          throw new ControllerException(
            myName+ " database exception: "+dbe.getMessage() );
      }
      myResponse.setStyle("listStocks" );
      return myResponse;
    }
```

Although this method looks complicated, what it is doing is very simple. If you choose to use the default state handler, each controller has to follow a simple signature, prefix the method signature with “`run`” and suffix it with “`State`”. So the old method “`displayStocks`” becomes the new method “`runDisplaysStockState`”. Additionally, the signature must accept two parameters of the type `ControllerRequest`, and `ControllerResponse`. These are similar to the `HttpServletRequest` and `HttpServletResponse` classes, but generalised so that the Expresso controllers can be used outside the Java Servlet environment, for example in a non-web application, such as a command-line utility program. If you are a seasoned Struts developer, you may be wondering what has happened to the familiar `ActionForm` and the `ActionMapping` classes. The Expresso `Controller` class implements the `performAction()` method, and stores the action form and the action mapping references in the `ControllerRequest` and `ConfigManager` objects respectively. (See the sidebar pullout: How to use DBObjects in a Pure Struts Action)

Returning back to the first state method `runPromptBuyStockState()`, first, we create a `Block` object named "StockList" which forms the root of the displayed stock list, the controller's response. We then proceed to retrieve all the stocks from the database using the database object class `StockTrade`. For each row tuple (which will be an individual stock trade), we need to create a separate block for it called "`Stock<RowNumber>`". For each row-block we create an `Output` object called "Detail" with named attributes that are filled directly from the column values from the database. For example, the attribute called "Title" for the output object has the database value for the column "`ST_ID`" from the "`STOCKTRADE1`" table. We repeat the task for each column field we eventually want to display in the JSP. Each row block also has two `Action` objects that we call "Buy" and "Sell" effectively. If you look carefully at the action objects created, you will see that they will transfer to the states "`promptBuyStock`" and "`promptSellStock`" respectively. Also note that the stock id number is set as a parameter for each action. This is because we will need to know what stock was selected when we come to buy or sell a stock. This is analogous to your e-commerce shopping cart system. So essentially the controller response is thus:

How to use DBObjects in a pure Struts Action

The key to using Expresso database objects within a pure Struts `Action` (no I dare not say legacy Struts) is to make sure that Expresso is initialised properly. The best way to achieve this is to allow the `ExpressoActionServlet` to serve as the controller.

If not then you should initialise the `com.jcorporate.expresso.core.misc.ConfigManager` class. Expresso needs to initialise itself from configuration files. Once you have set up the configuration manager, then it is a simple matter of retrieving database connection pools, and then retrieving a database connection in your Struts Action. You need to have a working database name that you supply to any created `DBObjects`. Once Expresso is initialised and configured, creating and using a database object inside a pure Struts Action is no problem at all.

```
ROOT
*--- StockList          (block)
     |
     +--Stock<n>        (block)
        |
        +-- Detail      (output)
        +-- Buy         (transition)
        +-- Sell        (transition)
```

Notice the state method must catch any database exceptions and re-throw them as controller exceptions. This is also a very good practice.

Controllers, Styles, States and Action Forwards

If you recall, Jakarta Struts associates a set of `ActionForwards` with an `Action` inside an `ActionMapping` object. Since Expresso controllers are derived directly from `Action` class, they are also associated with action forwards. In pure Struts programming, you normally explicitly write the action forward that is returned by calling the `findForward()` method of the `ActionMapping` class. Because Expresso controllers already handle the concept of internal states, Expresso maps each state automatically to an action forward that has the same name as the state. For example, the state "`displayStocks`" is mapped to the action forward "`displayStocks`". Inside the Orange Trader's Struts XML configuration file we could write something like this:

```
<forward name="displayStocks"   path="/orangetrader/stockList.jsp"
```

This means that the state method `runDisplayStocksState()` does *not* even have to specify an `ActionForward` to return. But wait a minute, what if you want to conditionally switch between two or more action forwards? You could do this in ordinary Struts but do you fix this in Expresso? And where has the Struts `ActionMapping` object disappeared to? How do I write the equivalent of `mapping.findForward("whatever")` with Expresso? One of the answers is found inside the `ControllerResponse` object that passed through the controller state method. The method is called `setStyle()` and accepts a single parameter that defines the action forward logical name, or it defines the output mode of the controller. For example "xml" to produce an XML document of the controller's response or "default" to produce an automatic HTML generation of the controller's response. In the `runDisplayStocksState()` method, the controller forwards to the action forward called "`listStocks`", which is defined in the Struts configuration XML file as:

```
</struts-config>
  </action-mappings>
    <action    path="/StockController2"
               type="com.xenonsoft.orangetrader.ot2.controller.StockController"
              scope="request"  validate="false">
       <forward name="listStocks"   path="/orangetrader/ot2/index.jsp" />
       <forward name="promptBuy"    path="/orangetrader/ot2/buy.jsp" />
       <forward name="promptSell"   path="/orangetrader/ot2/sell.jsp" />
    </action>
  </action-mappings>
</struts-config>
```

During the initialisation phase, Expresso saves the `ActionMapping` object associated with each Controller, inside a singleton global class called `ConfigManager`; there, you will find a method called `getMapping()` that accepts two parameters: the controller name and the state name. So to retrieve the mapping for the current state we call `getMapping("StockController", "displayStock")`. (Some sophisticated web applications generate dynamic `ActionForwards` and at this moment in time there is no method to set this inside Expresso's `ControllerResponse` object. A slight oversight.)

The List Stocks View

Now we look at the JSP, “`liststocks.jsp`”, that displays the view of the controller and uses custom tag actions to produce a view. I will simply present a short overview of what is going on rather than provide all the hard-details. We ask the controller to generate the list stock view by specifying a URL with the controller’s logical name. For example we add the hyperlink to JSP navigation page “`/StockController.do?state=displayStocks`”. After returning from the `runDisplayStocksState()` method, the controller response is stored in the request scope under a known public key attribute `Controller.RESPONSE_KEY`. In this way a JSP can read the controller response and build an HTML Form. Expresso toolkit has its own custom tag actions to generate HTML form elements in the documentation. Expresso has also supplied adapted Struts custom tags to work with Expresso controllers. I wrote my own custom tag actions before JCorporate published their taglibs; the central concept between both tag libraries remains the same: read the controller’s response from the public key attribute.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" %>
<%@ taglib uri="/xenon" prefix="xenon" %>
<%@ taglib uri="/xspresso" prefix="xspresso" %>

<table width="100%" border="1" cellspacing="1" cellpadding="2" >
  <tr>
       <th> <b>Title</b> </th>
       <th> <b>Trader</b> </th>
       <th> <b>Price</b> </th>
       <th> <b>Ask</b> </th>
       <th> <b>Bid</b> </th>
       <th> <b>Change</b> </th>
       <th> <b>Status</b> </th>
       <th colspan="2"> <b>Command</b> </th>
  </tr>

<xspresso:block controller="controller" blockPath="StockList"
                id="blockStocks" >
  <xenon:loop name="blockStocks" property="nested"
              loopId="block"      counterId="counter"
              className="com.jcorporate.expresso.core.controller.Block" >
    <tr>
      <xspresso:output outputPath="Detail" blockName="block" >
        <td> <xspresso:outputAttribute property="Title" /> &nbsp; </td>
        <td> <xspresso:outputAttribute property="Trader" /> &nbsp; </td>
        <td> <xspresso:outputAttribute property="Price" /> &nbsp; </td>
        <td> <xspresso:outputAttribute property="Ask" /> &nbsp; </td>
        <td> <xspresso:outputAttribute property="Bid" /> &nbsp; </td>
        <td> <xspresso:outputAttribute property="Change" /> &nbsp; </td>
        <td> <xspresso:outputAttribute property="Status" /> &nbsp; </td>
      </xspresso:output>
      <form action="<%= contextPath %>/StockController.do?cmd=button" method="POST"
>
        <td>
          <xspresso:transition  transitionName="Buy"   blockName="block" />
        </td>
        <td>
          <xspresso:transition  transitionName="Sell"  blockName="block" />
        </td>
      </form>
    </tr>
  </xenon:loop>
</xspresso:block>
</table>
```

The next action `<xspresso:block>` retrieved the named “StockList” `Block` from the controller. This is the root of the controller response. We now iterate through block rows that have been nested in the “StockList” block. What we are doing is generating an HTML table element where each table row corresponds directly to a row tuple from the database. For this task we use a custom iteration tag `<xenon:loop>`. Every loop of the custom iteration tag supplies a `Block` object for each row of the stock list. Because the `Output` object is nested in each block row, we can use the `<xspresso:output>` tag to get its reference. Within the body content of that tag we have another series of tags `<xspresso:outputAttribute>` to retrieve the named attributes. After displaying the row tuple information, we display the `Transition` objects “buy stock” and “sell stock”. The custom tag action `<xspresso:transition>` performs this task for us. It displays the relevant configured HTML Submit buttons for us.

Buying Stocks

The controller divides the task of buying a stock into two states. The first state prompts the user and the second state, if it is activated, completes the transaction of buying a stock. The method `promptBuyStock`() in the stock controller retrieves the stock id number from the controller request parameters. (Remember this was set as a parameter when the transitions were generated in `displayStocks`().) The id number is used to retrieve the corresponding stock trade from the database. An `Output` object called “SelectedStock” is created and added to the controller response, and its attributes are set up with the values from the database row tuple.

```
    protected ControllerResponse runPromptBuyStockState(
      ControllerRequest myRequest, ControllerResponse myResponse )
      throws ControllerException
    {
      try {
          String stockCodeStr = this.getParameter( "stock" );
          StockTrade stock = new StockTrade();
          stock.setDBName( this.getDBName() );
          stock.setField( "ST_ID", stockCodeStr );
          stock.retrieve();

          Output out1 = new Output();
          out1.setName( "SelectedStock" );
          out1.setAttribute( "Id",   stock.getField("ST_ID") );
          out1.setAttribute( "Title",      stock.getField("ST_TITLE") );
          out1.setAttribute( "Trader", stock.getField("ST_TRADER") );
          out1.setAttribute( "Price",  stock.getField("ST_PRICE") );
          out1.setAttribute( "Ask",    stock.getField("ST_ASK") );
          out1.setAttribute( "Bid",    stock.getField("ST_BID") );
          out1.setAttribute( "Change", stock.getField("ST_CHANGE") );
          out1.setAttribute( "Status", stock.getField("ST_STATUS") );
          this.add( out1 );

          // Create an input to get a bidding price from the client
          Input price = new Input();
          price.setName("BidPrice");
          price.setLabel("Bidding Price");
          price.setDisplayLength(12);
          price.setMaxLength(12);
          price.setType("text");
          this.add(price);

          // Create an transition to submit the form
          Transition buyTransition = new Transition("Buy", getClass().getName() );
          buyTransition.setName( "Buy" );
          buyTransition.addParam("state", "buyStock");
          buyTransition.addParam("stock", stock.getField("ST_ID") );
          buyTransition.addParam("next", getParameter("next" ) );
          this.add(buyTransition);
      }
      catch (DBException dbe) {
          throw new ControllerException(
            myName+ " database exception: "+dbe.getMessage() );
      }
      myResponse.setStyle("promptBuy" );
}
```

We continue by creating an `Input` object called “BidPrice” to allow the user to enter a bidding price. Finally, we generate a buy `Transition` object to allow the user to buy the stock. This transition will transfer to the next state “buyStock”. The transition parameter “stock” is set to the stock id. The transition parameter “next” is set to the next URL the controller will redirect to, which will be the JSP page that displays the bidding price form.

The Prompt Buy Stock View

Here is the “`buy.jsp`”, which should be fairly comprehensible by now. The controller response is found again in request scope `Controller.RESPONSE_KEY`. We display the actual stock trade details in an HTML table and we use the `<xspresso:output>` and nested `<xspresso:outputAttribute>` action tags to get the information. Now we come to the HTML Form. As soon as the form is submitted it will attempt to invoke the `ExpressoActionServlet` that handles interaction with our controller with a special CGI query parameter “`cmd=button`”. This parameter signals the action servlet to specially interpret HTML form submit buttons. The rendering of `Transition` objects generates an HTML form submit tag and also an HTML form hidden tag, which holds all the transition parameter names and values that we generated in the controller response. The hidden tag is named a specific way that associates it with the partner HTML form submit tag. The “`cmd=button`” query parameter tells the action servlet to also look out for associated query parameters defined by a specially named HTML form hidden tag. The value of the hidden tag is defined as a list of mapped parameters, for example `value=”firstName=peter&lastName=pilgrim”`. If the ExpressoAction Servlet sees the special name attribute as a query parameter, it will automatically convert the value string into individual request attributes. So for instance, `request.getAttribute( ”firstName” )` will return the string “`peter`”. (It is a bit long-winded, I know, but it really works! Use your browser’s “view source” to look at the “`buy.jsp`” to see what is going on.)

```
<table border=”1” cellpadding=”2” cellspacing=”0” >
<tr>
  <xspresso:output controller="controller" outputPath="SelectedStock" >
    <td> <xspresso:outputAttribute property="Title" /> &nbsp; </td>
    <td> <xspresso:outputAttribute property="Trader" /> &nbsp; </td>
    <td> <xspresso:outputAttribute property="Price" /> &nbsp; </td>
    <td> <xspresso:outputAttribute property="Ask" /> &nbsp; </td>
    <td> <xspresso:outputAttribute property="Bid" /> &nbsp; </td>
    <td> <xspresso:outputAttribute property="Change" /> &nbsp; </td>
    <td> <xspresso:outputAttribute property="Status" /> &nbsp; </td>
  </xspresso:output>
</tr>
</table>

<form action="<%= contextPath %>/StockController2.do?cmd=button" method="POST" >
<table border="1" cellspacing="1" cellpadding="2" >
```

```
<tr bgcolor="#E0E0FF" >
     <td>Enter Bidding Price:</td>
     <td>
      <xspresso:input controller="controller" elementName="BidPrice" />
     </td>
</tr>
<tr bgcolor="#E0E0FF" >
     <td colspan="2">
      <xspresso:transition  controller="controller"  transitionName="Buy" />
     </td>
</tr>
</table>
</form>
```

The next part of the JSP, the form itself, utilises the `<xspresso:input>` custom action to display the text field “BidPrice”. The appropriate HTML submit button is then rendered with the `<xspresso:transition>` transition “Buy” again.

The Buy Stock State

Assuming the user wants to the buy the stock, the form will eventually be submitted to the controller and it will enter the method “`runBuyStockState()`” in order to process the state. The method works by retrieving the stock code and price from the controller request. An exception is raised if the price cannot be converted from a String to a floating-point value. The stock id is used to retrieve the actual `StockTrade` from the database. We can thus interrogate the stock trade itself. If the bidding price is greater than or equal to the asking price and the asking price is not zero, then we have a sale. If we have a sale, then we set the new price of the stock trade to the input bidding price, and reset the bidding price accordingly. We also calculate the percentage change between the new price and the old price. Otherwise, with no sale, we set a new bidding price if the new bidding price was lower than the current one.

```
    protected ControllerResponse  runBuyStockState(
     ControllerRequest myRequest, ControllerResponse myResponse )
     throws ControllerException
    {
     String stockCode = myRequest.getParameter( "stock" );
     String priceStr = myRequest.getParameter( "BidPrice" );
     float price = 0.0F;
     try { price = new Float( priceStr ).floatValue(); }
     catch (NumberFormatException nfe) {
         throw new ControllerException(
           "controller cannot parse float parameter `BidPrice'
("+priceStr+")" );
     }

     try {
         StockTrade stk = new StockTrade();
         stk.setDBName( myRequest.getDBName() );
         stk.setField( "ST_ID", stockCode );
         stk.retrieve();
         float askPrice = stk.getFieldFloat( "ST_ASK" );
         float bidPrice = stk.getFieldFloat( "ST_BID" );
         float currPrice = stk.getFieldFloat( "ST_PRICE" );
         if ( price >= askPrice && askPrice > 0.0 ) {
          stk.setField( "ST_TRADER", myRequest.getUser() );
          stk.setField( "ST_STATUS", "BOUGHT" );
          stk.setField( "ST_PRICE", Float.toString(price) );
          stk.setField( "ST_BID", "0.0" );

          float change = (float)(( price - currPrice ) / currPrice * 100.0
);
          stk.setField( "ST_CHANGE", Float.toString(change) );
          stk.update();
         }
         else {
          stk.setField( "ST_STATUS", "WANTED" );
          if ( price > bidPrice ) {
              stk.setField( "ST_BID", Float.toString(price) );
          }
          stk.update();
         }
     }
     catch (DBException dbe) {
         throw new ControllerException(
             "unable to buy stock:`"+stockCode+
             "' DBException: "+dbe.getMessage() );
     }
     try {  transition( "displayStocks", myRequest, myResponse ); }
     catch (NonHandleableException nhe ) {
         nhe.printStackTrace(System.err);
     }
```

```
      return myResponse;
    }
```

Notice above that Expresso permits a controller to transfer to the next state, in the middle of the controller state. This is very useful for presenting a database-refreshed view back to the user, because you can write a state that simply reads from the database into action forms, which are then printed out to the JSP view. Typically, reading from the database involves an SQL select statement such as "`SELECT A,B FROM FOOBAR`". The update state makes the changes to the database itself, essentially performing "`INSERT INTO FOOBAR`" on the database. At this point the presentation view and the database are showing different information. It would be nice to show a refresh of the database, and this is what the method `transition()` can do. It transfers to another controller state, conveniently saving repetitious code, time and energy.

And that, ladies and gentlemen, concludes the buying stock transaction.

Where is my Action Form, Action Mapping?

Expresso controllers hide the existence of the Struts action form. If you have created an `ActionForm` and associate it with an `ActionMapping`, it can still be retrieved from inside an Expresso controller. You will find the name of it from the `ControllerRequest` object. The accessor method is called `getFormAttribute()` and there is a mutator method called `setFormAttribute()`. The action form can be found by obtaining the form attribute's name, then using it to query the request scope. In other words:

```
ActionForm form = (ActionForm)
  request.getAttribute(
     myRequest.getFormAttribute());
```

Likewise Expresso also hides the *`ActionMapping`* class, but not completely. You can retrieve the missing action mapping object from the *`ConfigManager.getMapping()`* method. See Expresso's Javadoc for more details.

Selling Stocks

Selling stocks is almost a mirror image of the buying process. The controller again has two states to progress through: the first state, "promptSellStock", and the second state, "sellStock". The first state prompts the user for the asking price this time. It supplies the controller response with the selected stock trade, an `Input`object for the user to enter a price, and an `Action` object to transfer to the "sellStock" state. The code for "`runPromptSellStockState`" and "`sell.jsp`" are so similar to the buying process that I will not show them here, saving space and potentially some trees of the world's great forests.

The `runSellStockState()` method works by retrieving the stock code, and the price from the controller request. The stock id is used to retrieve the actual `StockTrade` from the database. We can thus interrogate the selected stock trade. If the asking price is greater than or equal to the bidding price and the bidding price is not zero then we have a sale. If we have a sale, then we set the new price of the stock trade to the input asking price, and reset the asking price accordingly. We also calculate the percentage difference between the new price and the old price. Otherwise, with no sale possible, we set a new asking price if the new asking price was higher than the current one.

```
    protected ControllerResponse runSellStockState(
      ControllerRequest myRequest, ControllerResponse myResponse )
      throws ControllerException
    {
      String stockCode = myRequest.getParameter( "stock" );
      String priceStr = myRequest.getParameter( "AskPrice" );
      float price = 0.0F;
      try { price = new Float( priceStr ).floatValue(); }
      catch (NumberFormatException nfe) {
          throw new ControllerException(
            myName+"controller cannot parse float parameter `AskPrice'
("+priceStr+")" );
      }

      try {
          StockTrade stk = new StockTrade();
          stk.setDBName( myRequest.getDBName() );
          stk.setField( "ST_ID", stockCode );
          stk.retrieve();
          float askPrice = stk.getFieldFloat( "ST_ASK" );
          float bidPrice = stk.getFieldFloat( "ST_BID" );
          float currPrice = stk.getFieldFloat( "ST_PRICE" );
          if ( price <= bidPrice && bidPrice > 0.0 ) {
            stk.setField( "ST_TRADER", myRequest.getUser() );
            stk.setField( "ST_STATUS", "SOLD" );
            stk.setField( "ST_PRICE", Float.toString(price)  );
            stk.setField( "ST_ASK", "0.0" );

            float change = (float)(( price - currPrice ) / currPrice * 100.0 );
            stk.setField( "ST_CHANGE", Float.toString(change) );
            stk.update();
          }
          else {
            stk.setField( "ST_STATUS", "SELLING" );
            if ( price < askPrice || askPrice < 0.01 ) {
                stk.setField( "ST_ASK", Float.toString(price) );
            }
            stk.update();
          }
          float salePrice = stk.getFieldFloat( "ST_PRICE" );
      }
      catch (DBException dbe) {
          throw new ControllerException(
            "unable to sell stock:`"+stockCode+
            "' DBException: "+dbe.getMessage() );
      }

      // Goto back to the display stocks state
      try { transition( "displayStocks", myRequest, myResponse ); }
      catch (NonHandleableException nhe ) {
          nhe.printStackTrace(System.err);
      }

      return myResponse;
    }
```

Conclusion

Developing a web application entirely from scratch can be a massive overkill. There just isn't enough time (or money) in the world to reinvent the wheel every time for each web application. It is better to leverage an open-sourced framework and toolkit, like the Expresso Framework (or any other framework that you might prefer politically to support and use) because it will almost certainly have far more virtual Internet resources to help debug and maintain it. A good toolkit will have a strong codebase, with an emphasis on quality over quantity; it should be popular, flexible, approachable, comprehensible, and most importantly, useful. In the open-source world, it is particularly important to get enough *eyeballs* to edit, compile, test, debug, and deploy in order to make projects successful in the long run. Thus, frameworks, provided they are supported by many developers and users, will be generally more robust, productive, and efficient than 'going it alone'. I suspect that most people who develop the next generation of web software will prefer to build on something that already works out of the box. Hopefully, you can take this 'best of' practice, using a framework, Jakarta Struts, custom tag action libraries, JSP's, and Servlets, elsewhere and make it a success in your business web applications. In the future, the impact of frameworks and the forthcoming JSP *Standard Tag Library* will have a powerful affect on how we develop passive web technologies. Enjoy the rollercoaster ride because it is going to be rough!

Peter Pilgrim,

February 2002

About the author

Peter Pilgrim is a Senior Systems Analyst has been working with Java since 1998, he is Sun Certified

Java 2 Programmer, previously worked considerably with Java Swing, JFC, and GUI technology, then he made the change to the server-side. He is a member of the Association C & C++ Users group UK http://www.accu.org/ where he regularly contributes Java related articles. He contributes to open source software as a core developer with the Expresso Framework team. Peter lives in London, England, presently works professionally for Deutsche Bank AG (UK). He can be reached by sending mail to "peterp@xenonsoft.demon.co.uk" or "peter.pilgrim@db.com".

To download the latest version of this article, please visit
http://www.xenonsoft.demon.co.uk/products/java.html

Caveat Emptor

This article was written against Expresso Framework 4.02 and the software described works with this version of the toolkit. The nature of open source software often means that it is generally developed at so-called Internet speed. So I cannot guarantee the complete validity of the source code after say three weeks or even eighteen months down the line.

Resources

- Expresso Framework produced by JCorporate: http://www.JCorporate.com/
- Expresso Framework Project Page: http://www.JCorporate.com/components/internal/projframe.jsp?category=65
- Software for this article: http://www.xenonsoft.demon.co.uk/products/java.html
- Expresso Developer's Guide: http://www.jcorporate.com/doc/edg/edg.html
- Jakarta Struts Project: http://jakarta.apache.org/struts/
- Ted Husted's Struts Web Site http://www.husted.com/struts/
- J2EE Home Page at Sun Java division: http://java.sun.com/j2ee/
- Sun Java division J2EE Pet Store example: http://java.sun.com/j2ee/blueprints/jps11/archoverview.html
- Sun Java division J2EE Blueprints: http://java.sun.com/j2ee/blueprints/index.html
- Model View Controller design pattern: http://compsci.about.com/cs/mvcpattern/
- Design patterns: http://compsci.about.com/cs/designpatterns/index.htm
- Design patterns without the advertisements by Mark Grand: http://www.mindspring.com/~mgrand/pattern_synopses.htm
- Free Design Patterns Java Companion on-line book: http://www.patterndepot.com/put/8/JavaPatterns.htm

- JavaServer Page home pages:http://java.sun.com/products/jsp/
- Another good read for the Model View Controller design pattern: http://ootips.org/mvc-pattern.html
- Java Server Faces JSR 127 http://www.jcp.org/jsr/detail/127.jsp
- Element Construction Set Jakarta http://jakarta.apache.org/ecs/index.html
- Professional JSP 2nd Edition, Brown, Burdick, Falkner, Wrox Books isbn 1861005512
- JSP Site Design 1st Edition, Duffal, Goyal, Husted et al, Wrox Books isbn 1861004958

TheServerSide.com
YOUR J2EE COMMUNITY