Published on **ONJava.com** (http://www.onjava.com/)
http://www.onjava.com/pub/a/onjava/2004/04/07/wiringwebapps.html
See this if you're having trouble printing code examples

Wiring Your Web Application with Open Source Java

by Mark Eagle
04/07/2004

Building non-trivial web applications with Java is no trivial task. There are many things to consider when structuring an architecture to house an application. From a high-level, developers are faced with decisions about how they are going to construct user interfaces, where the business logic will reside, and how to persist application data. Each of these three layers has their own questions to be answered. What technologies should be implemented across each layer? How can the application be designed so that it is loosely coupled and flexible to change? Does the architecture allow layers to be replaced without affecting other layers? How will the application handle container level services such as transactions?

There are definitely a number of questions that need to be addressed when creating an architecture for your web application. Fortunately, there have been developers that have run into these reoccurring problems and built frameworks to address these issues. A good framework relieves developers from attempting to reinvent the wheel for complex problems; it is extensible for internal customization; and it has a strong user community to support it. Frameworks generally address one problem well. However, your application will have several layers that might require their own framework. Just solving your UI problem does not mean that you should couple your business logic and persistence logic into a UI component. For example, you should not have business logic with JDBC code inside of a controller. This is not the functionality that a controller was intended to provide. A UI controller should be a lightweight component that delegates calls to other application layers for services outside the UI scope. Good frameworks naturally form guidelines where code should be placed. More importantly, frameworks alleviate developers from building code such as persistence from scratch and allow them to concentrate on the application logic that is important to a client.

This article will discuss how to combine several well-known frameworks to achieve loose coupling, how to structure your architecture, and how to enforce a consistent design across all application layers. The challenge is combining frameworks so that each layer is exposed to each other in a loosely coupled manner, regardless of the underlying technologies. This article will discuss one strategy for combining frameworks using three popular open source frameworks. For the presentation layer we will use Struts; for our business layer we will use Spring; and for our persistence layer we will use Hibernate. You should be able to substitute any one of these frameworks in your application and get the same effect. Figure 1 shows what this looks like from a high level when the frameworks are combined.

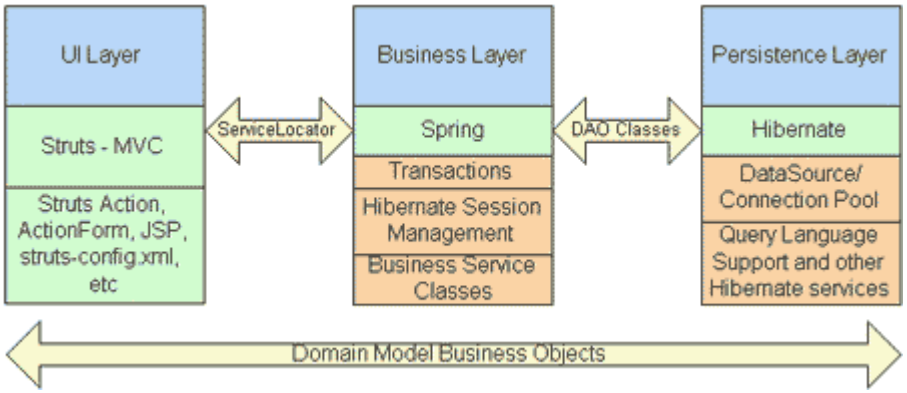

Figure 1. Overview of framework architecture with Struts, Spring, and Hibernate.

Application Layering

Most non-trivial web applications can be divided into at least four layers of responsibility. These layers are the presentation, persistence, business, and domain model layers. Each layer has a distinct responsibility in the application and should not mix functionality with other layers. Each application layer should be isolated from other layers but allow an interface for communication between them. Let's start by inspecting each of these layers and discuss what these layers should provide and what they should not provide.

The Presentation Layer

At one end of a typical web application is the presentation layer. Many Java developers understand what Struts provides. However, too often, coupled code such as business logic is placed into an `org.apache.struts.Action`. So, let's agree on what a framework like Struts should provide. Here is what Struts is responsible for:

- Managing requests and responses for a user.
- Providing a controller to delegate calls to business logic and other upstream processes.
- Handling exceptions from other tiers that throw exceptions to a Struts Action.
- Assembling a model that can be presented in a view.
- Performing UI validation.

Here are some items that are often coded using Struts but should not be associated with the presentation layer:

- Direct communication with the database, such as JDBC calls.
- Business logic and validation related to your application.
- Transaction management.

Introducing this type of code in the presentation layer leads to type coupling and cumbersome maintenance.

The Persistence Layer

At the other end of a typical web application is the persistence layer. This is usually where things get out of control fast. Developers underestimate the challenges in building their own persistence frameworks. A custom, in-house persistence layer not only requires a great amount of development time, but also often lacks functionality and becomes unmanageable. There are several open source object-to-relational mapping (ORM) frameworks that solve much of this problem. In particular, the Hibernate framework allows object-to-relational persistence and query service for Java. Hibernate has a medium learning curve for Java developers who are already

familiar with SQL and the JDBC API. Hibernate persistent objects are based on plain-old Java objects and Java collections. Furthermore, using Hibernate does not interfere with your IDE. The following list contains the type of code that you would write inside a persistence framework:

- Querying relational information into objects. Hibernate does this through an OO query language called HQL, or by using an expressive criteria API. HQL is very similar to SQL except you use objects instead of tables and fields instead of columns. There are some new specific HQL language elements to learn; however, they are easy to understand and well documented. HQL is a natural language to use for querying objects that require a small learning curve.
- Saving, updating, and deleting information stored in a database.
- Advanced object-to-relational mapping frameworks like Hibernate have support for most major SQL databases, and they support parent/child relationships, transactions, inheritance, and polymorphism.

Here are some items that should be avoided in the persistence layer:

- Business logic should be in a higher layer of your application. Only data access operations should be permitted.
- You should not have persistence logic coupled with your presentation logic. Avoid logic in presentation components such as JSPs or servlet-based classes that communicate with data access directly. By isolating persistence logic into its own layer, the application becomes flexible to change without affecting code in other layers. For example, Hibernate could be replaced with another persistence framework or API without modification to the code in any other layer.

The Business Layer

The middle component of a typical web application is the business or service layer. This service layer is often the most ignored layer from a coding perspective. It is not uncommon to find this type of code scattered around in the UI layer or in the persistence layer. This is not the correct place because it leads to tightly coupled applications and code that can be hard to maintain over time. Fortunately, several frameworks exist that address these issues. Two of the most popular frameworks in this space are Spring and PicoContainer. These are referred to as microcontainers that have a very small footprint and determine how you wire your objects together. Both of these frameworks work on a simple concept of dependency injection (also known as inversion of control). This article will focus on Spring's use of setter injection through bean properties for named configuration parameters. Spring also allows a sophisticated form of constructor injection as an alternative to setter injection as well. The objects are wired together by a simple XML file that contains references to objects such as the transaction management handler, object factories, service objects that contain business logic, and data access objects (DAO).

The way Spring uses these concepts will be made clearer with examples later in this article. The business layer should be responsible for the following:

- Handling application business logic and business validation
- Managing transactions
- Allowing interfaces for interaction with other layers
- Managing dependencies between business level objects
- Adding flexibility between the presentation and the persistence layer so they do not directly communicate with each other
- Exposing a context to the business layer from the presentation layer to obtain business services
- Managing implementations from the business logic to the persistence layer

The Domain Model Layer

Finally, since we are addressing non-trivial, web-based applications we need a set of objects that can move between the different layers. The domain object layer consists of objects that represent real-world business objects such as an `Order`, `OrderLineItem`, `Product`, and so on. This layer allows developers to stop building and maintaining unnecessary data transfer objects, or DTOs, to match their domain objects. For example, Hibernate allows you to read database information into an object graph of domain objects, so that you can present it to your UI layer in a disconnected manner. Those objects can be updated and sent back across to the persistence layer and updated within the database. Furthermore, you do not have to transform objects into DTOs, which can get lost in translation as they are moved between

different application layers. This model allows Java developers to work with objects naturally in an OO fashion without additional coding.

Wiring Together a Simple Example

Now that we understand the components from a high level, let's put this into practice. Again, for this example, we will combine the Struts, Spring, and Hibernate frameworks. Each one of these frameworks has too much detail to cover in one article. Instead of going into many details about each framework, this article will show how to wire them together with simple example code. The sample application will demonstrate how a request is serviced across each layer. A user of this sample application can save a new order to the database and view an existing order in the database. Further enhancements might allow the user to update or delete an existing order.

You can download the source code of the application.

First, we will create our domain objects since they will interoperate with each layer. These objects will allow us to define what should be persisted, what business logic should be provided, and what type of presentation interface should be designed. Next, we will configure the persistence layer and define object-to-relational mappings with Hibernate for our domain objects. Then we will define and configure our business objects. After we have these components we can discuss wiring these layers using Spring. Finally, we will provide a presentation layer that knows how to communicate with the business service layer and knows how to handle exceptions that arise from other layers.

Domain Object Layer

Since these objects will interoperate across all layers this might be a good place to start coding. This simple domain model will contain an object that represents an order and an object that represents a line item for an order. The order object will have a one-to-many relationship to a collection of line item objects. The example code has two simple objects in the domain layer:

- `com.meagle.bo.Order.java`: contains the header-level information for an order.
- `com.meagle.bo.OrderLineItem.java`: contains the detail-level information for an order.

Consider choosing package names for your objects that reflect how your application is layered. For example, the domain objects in the sample application can be located in the `com.meagle.bo` package. More specialized domain objects would be located in subpackages under the `com.meagle.bo` package. The business logic begins in the `com.meagle.service` package and DAO objects are located in the `com.meagle.service.dao.hibernate` package. The presentation classes for forms and actions reside in `com.meagle.action` and `com.meagle.forms`, respectively. Accurate package naming provides a clear separation for the functionality that your classes provide, allows for easier maintenance when troubleshooting, and provides consistency when adding new classes or packages to the application.

Persistence Layer Configuration

There are several steps involved in setting up the persistence layer with Hibernate. The first step is to configure our domain business objects to be persisted. Since Hibernate works with POJOs we will use our domain objects for persistence. Therefore the `Order` and `OrderLineItem` objects will need to provide getter and setter methods for all fields that they contain. The `Order` object would contain setter and getter methods such as `ID`, `UserName`, `Total`, and `OrderLineItems` in a standard JavaBean format. The `OrderLineItem` would similarly follow the JavaBean format for its fields.

Hibernate maps domain objects-to-relational databases in XML files. For our `Order` and `OrderLineItem` objects there will be two mapping files to express this. There are tools such as XDoclet to assist with this mapping. Hibernate will map the domain objects to these files:

- *Order.hbm.xml*
- *OrderLineItem.hbm.xml*

You will find these generated files in the *WebContent/WEB-INF/classes/com/meagle/bo* directory. The Hibernate `SessionFactory` is configured to know which database it is communicating with, the `DataSource` or connection pool to use, and what persistent objects are available for persistence. Session objects provided by the

`SessionFactory` are the interface used to translate between Java objects and persistence functions such as selecting, saving, updating, and deleting objects. We will discuss configuring the `SessionFactory` that Hibernate requires to handle Session objects in a later section.

Business Layer Configuration

Now that we have our domain objects we need to have business service objects that perform application logic, make calls to the persistence layer, take requests from the UI layer, deal with transactions, and handle exceptions. To wire all of this together and make this easy to manage we will use the bean management aspect of the Spring framework. Spring uses inversion of control (IoC), or setter dependency injection, to wire up objects that are referenced in an external XML file. Inversion of control is a simple concept that allows objects to accept other objects that are created at a higher level. This way your object is free from having to create objects and reduces object coupling.

Here is an example of an object creating its dependencies without IoC, which leads to tight object coupling:

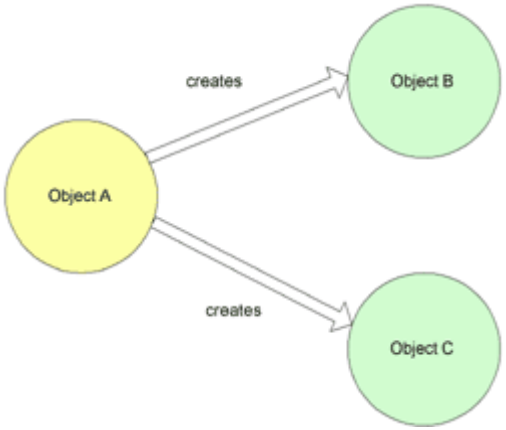

Figure 2. Objects arranged without IoC. Object A creates objects B and C.

And here is an example with IoC that allows objects to be created at higher levels and passed into objects so that they can use the implementations directly:

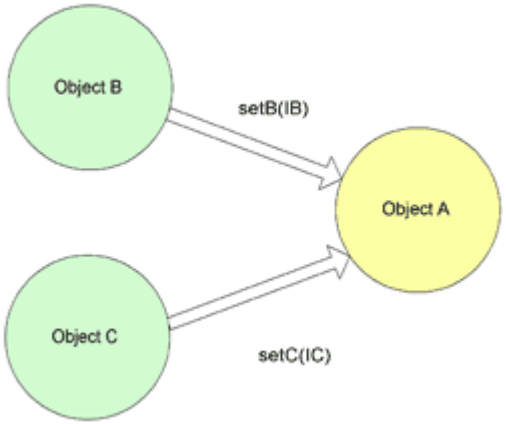

Figure 3. Objects arranged with IoC. Object A contains setter methods that accept interfaces to objects B and C. This could have also been achieved with constructors in object A that accepts objects B and C.

Building Our Business Service Objects

The setters we will use in our business objects accept interfaces that allow loosely defined implementations of the objects that will be set, or injected. In our case we will allow our business service object to accept a DAO to handle the persistence of our domain objects. While the examples in this article use Hibernate, we can easily switch implementations to a different persistence framework and inform Spring of the new implementation DAO object to use. You can see how programming to interfaces and using the dependency injection pattern loosely couples your business logic from your persistence mechanism.

Here is the interface for the business service object that is stubbed for a DAO object dependency:

```
public interface IOrderService {
  public abstract Order saveNewOrder(Order order)
    throws OrderException,
           OrderMinimumAmountException;

  public abstract List findOrderByUser(
                                    String user)
                            throws OrderException;

  public abstract Order findOrderById(int id)
                            throws OrderException;

  public abstract void setOrderDAO(
                              IOrderDAO orderDAO);
}
```

Notice that the code above has a setter for a DAO object. There is not a `getOrderDAO` method because it is not necessary since there is often no need to access the wired `OrderDAO` object from the outside. The DAO object will be used to communicate with our persistence layer. We will wire the business service object and the DAO object together with Spring. Because we are coding to interfaces, we do not tightly couple the implementation.

The next step is to code our DAO implementation object. Since Spring has built-in support for Hibernate this example DAO will extend the `HibernateDaoSupport` class, which allows us to easily get a reference to a `HibernateTemplate`, which is a helper class that simplifies coding with a Hibernate Session and handles `HibernateExceptions`. Here is the interface for the DAO:

```
public interface IOrderDAO {

  public abstract Order findOrderById(
                                    final int id);

  public abstract List findOrdersPlaceByUser(
                            final String placedBy);

  public abstract Order saveOrder(
                               final Order order);
}
```

We still have a couple more objects to wire together for our business layer. This includes the `HibernateSessionFactory` and a `TransactionManager` object. This is done directly in the Spring configuration file. Spring provides a `HibernateTransactionManager`, which will bind a Hibernate Session from the factory to a thread to support transactions (see `ThreadLocal` for more information). Here is the Spring configuration of the `HibernateSessionFactory` and the `HibernateTransactionManager`:

```
<bean id="mySessionFactory"
       class="org.springframework.orm.hibernate.
              LocalSessionFactoryBean">
  <property name="mappingResources">
    <list>
      <value>
        com/meagle/bo/Order.hbm.xml
      </value>
      <value>
        com/meagle/bo/OrderLineItem.hbm.xml
      </value>
    </list>
  </property>

  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">
        net.sf.hibernate.dialect.MySQLDialect
      </prop>
      <prop key="hibernate.show_sql">
        false
      </prop>
      <prop key="hibernate.proxool.xml">
        C:/MyWebApps/.../WEB-INF/proxool.xml
      </prop>
      <prop key="hibernate.proxool.pool_alias">
          spring
      </prop>
    </props>
  </property>
</bean>

<!-- Transaction manager for a single Hibernate
SessionFactory (alternative to JTA) -->
<bean id="myTransactionManager"
         class="org.
                springframework.
                orm.
                hibernate.
                HibernateTransactionManager">
  <property name="sessionFactory">
    <ref local="mySessionFactory"/>
  </property>
  </bean>
```

Each object can be referenced in the Spring configuration within a `<bean>` tag. In this case the bean `mySessionFactory` represents a `HibernateSessionFactory` and the bean `myTransactionManager` represents a Hibernate transaction manager. Notice that the `transactionManger` bean has a property element called `sessionFactory`. The `HibernateTransactionManager` class has a setter and getter for `sessionFactory`, which is used for dependency injection when the Spring container starts. The `sessionFactory` property references the `mySessionFactory` bean. These two objects will now be wired together when the Spring container initializes. This wiring relieves you from creating singleton objects and factories for referencing and creating these objects, which reduces code maintenance in your application. The `mySessionFactory` bean has two property elements, which translate to setters for `mappingResources` and `hibernatePropertes`. Normally, this configuration would be stored in the *hibernate.cfg.xml* file if you were using Hibernate outside of Spring. However, Spring provides an easy way to incorporate the Hibernate configuration within the Spring configuration file. For more information see the Spring API.

Now that we have our container service beans configured and wired together we need to wire our business service object and our DAO object together. Then we need to wire these objects to the transaction manager.

Here is what this looks like in the Spring configuration file:

```
<!-- ORDER SERVICE -->
<bean id="orderService"
  class="org.
         springframework.
         transaction.
         interceptor.
         TransactionProxyFactoryBean">
  <property name="transactionManager">
    <ref local="myTransactionManager"/>
  </property>
  <property name="target">
    <ref local="orderTarget"/>
  </property>
  <property name="transactionAttributes">
    <props>
      <prop key="find*">
     PROPAGATION_REQUIRED,readOnly,-OrderException
      </prop>
      <prop key="save*">
     PROPAGATION_REQUIRED,-OrderException
      </prop>
    </props>
  </property>
</bean>

<!-- ORDER TARGET PRIMARY BUSINESS OBJECT:
Hibernate implementation -->
<bean id="orderTarget"
         class="com.
                meagle.
                service.
                spring.
                OrderServiceSpringImpl">
  <property name="orderDAO">
    <ref local="orderDAO"/>
  </property>
</bean>

<!-- ORDER DAO OBJECT -->
<bean id="orderDAO"
         class="com.
                meagle.
                service.
                dao.
                hibernate.
                OrderHibernateDAO">
  <property name="sessionFactory">
    <ref local="mySessionFactory"/>
  </property>
</bean>
```

Figure 4 is an overview of what we have wired together. This shows how each object is related and set into other objects by Spring. Compare this with the Spring configuration file in the sample application to see these relationships.

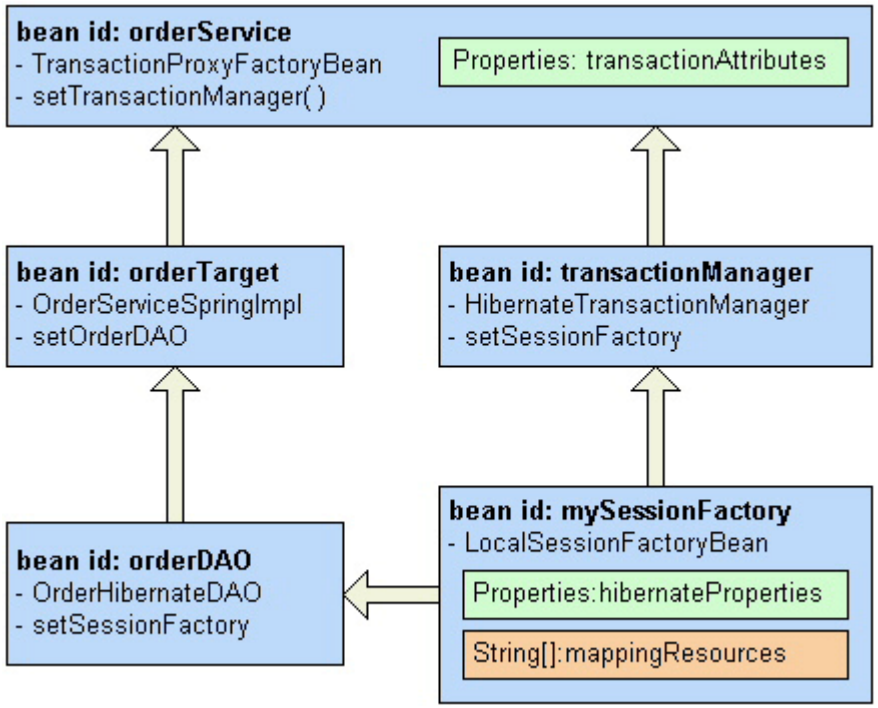

Figure 4. This is how Spring will assemble the beans based on this configuration.

This example uses a `TransactionProxyFactoryBean`, which has a setter for a transaction manager that we have already defined. This is a convenience object that knows how to deal with declarative transaction handling and your service objects. You can define how transactions are handled through the `transactionAttributes` property, which defines patterns for method names, and how they participate in a transaction. For more information about configuring isolation levels and commits or rollbacks on a transaction see `TransactionAttributeEditor`.

The class `TransactionProxyFactoryBean` also has a setter for a target, which will be a reference to our business service object called `orderTarget`. The `orderTarget` bean defines which business service class to use and it has a property which refers to `setOrderDAO()`. This property will populate the `orderDAO` bean that is our DAO object to communicate with our persistence layer.

One more note about Spring and beans is that beans can operate in two modes. These are defined as singleton and prototype. The default mode for a bean is singleton that means that one shared instance of the bean will be managed. This is used for stateless operations like a stateless session bean would provide. The prototype mode allows new instances of the bean to be create when the bean is served through Spring. You should only use prototype mode when each user needs their own copy of the bean.

Providing a Service Locator

Now that we have wired up our services with our DAO we need to expose our services to other layers. This is generally used from code in a layer such as UI that uses Struts or Swing. An easy way to handle this is with a service locator patterned class to return resources from a Spring context. This can also be done directly through Spring by referencing the bean ID.

Here is an example of how a service locator can be configured in a Struts Action:

```
public abstract class BaseAction extends Action {

  private IOrderService orderService;

  public void setServlet(ActionServlet
                                actionServlet) {
    super.setServlet(actionServlet);
    ServletContext servletContext =
              actionServlet.getServletContext();

    WebApplicationContext wac =
      WebApplicationContextUtils.
         getRequiredWebApplicationContext(
                                servletContext);

      this.orderService = (IOrderService)
                    wac.getBean("orderService");
  }

  protected IOrderService getOrderService() {
    return orderService;
  }
}
```

UI Layer Configuration

The UI Layer for the example application uses the Struts framework. Here we will discuss what is related to Struts when layering an application. Let's begin by examining an `Action` configuration within the `struts-config.xml` file.

```
<action path="/SaveNewOrder"
    type="com.meagle.action.SaveOrderAction"
    name="OrderForm"
    scope="request"
    validate="true"
    input="/NewOrder.jsp">
  <display-name>Save New Order</display-name>
  <exception key="error.order.save"
    path="/NewOrder.jsp"
    scope="request"
    type="com.meagle.exception.OrderException"/>
  <exception key="error.order.not.enough.money"
    path="/NewOrder.jsp"
    scope="request"
    type="com.
          meagle.
          exception.
          OrderMinimumAmountException"/>
  <forward name="success" path="/ViewOrder.jsp"/>
  <forward name="failure" path="/NewOrder.jsp"/>
</action>
```

The `SaveNewOrder` Action is used to persist an order that the user submitted from the UI layer. This is a typical Struts Action; however, notice the exception configuration for this action. These exceptions are also configured in the Spring configuration file, *applicationContext-hibernate.xml*, for our business service objects in

the `transactionAttributes` property. When these exceptions get thrown back from the business layer we can handle them appropriately in our UI. The first exception, `OrderException`, will be used by this action when there is a failure saving the order object in the persistence layer. This will cause the transaction to rollback and propagate the exception back through the business object to the Struts layer. The `OrderMinimumAmountException` will also be handled in a transaction within the business object logic that fails when the order placed does not meet the minimum order amount. Again, the transaction will rollback and this exception can be handled properly by the UI layer.

The last wiring step is to allow our presentation layer to interact with our business layer. This is done by using the service locator that was previously discussed. The service layer acts as an interface to our business logic and persistence layer. Here is how the `SaveNewOrder Action` in Struts might use a service locator to invoke a business method:

```
public ActionForward execute(
  ActionMapping mapping,
  ActionForm form,
  javax.servlet.http.HttpServletRequest request,
  javax.servlet.http.HttpServletResponse response)
  throws java.lang.Exception {

  OrderForm oForm = (OrderForm) form;

  // Use the form to build an Order object that
  // can be saved in the persistence layer.
  // See the full source code in the sample app.

  // Obtain the wired business service object
  // from the service locator configuration
  // in BaseAction.
  // Delegate the save to the service layer and
  // further upstream to save the Order object.
  getOrderService().saveNewOrder(order);

  oForm.setOrder(order);

  ActionMessages messages = new ActionMessages();
  messages.add(
      ActionMessages.GLOBAL_MESSAGE,
            new ActionMessage(
      "message.order.saved.successfully"));

  saveMessages(request, messages);

  return mapping.findForward("success");
}
```

Conclusion

This article covers a lot of ground in terms of technology and architecture. The main concept to take away is how to better separate your application, user interface, persistence logic, and any other application layer you require. Doing this will decouple your code, allow new code components to be added, and make your application more maintainable in the future. The technologies covered here address specific problems well. However, by using this type of architecture you can replace application layers with other technologies. For example, you might not want to use Hibernate for persistence. Since you are coding to interfaces in your DAO objects it should be apparent to you how you might use another technology or framework, such as iBATIS, as a substitute. Or you might want to replace your UI layer with a different framework than Struts. Switching UI layer implementations should not directly affect your business logic or your persistence layer. Replacing your persistence layer should not affect your UI logic or business service layer. Wiring a web application is not a trivial task but it can be made easier to deal with by decoupling your

application layers and wiring it together with suitable frameworks.

Mark Eagle is a Senior Software Engineer at MATRIX Resources, Inc. in Atlanta, GA.

Return to ONJava.com.