Play Family Feud Only On Uproar
How Many People Play Family Feud?
Millions and Counting 1
Join the Fun Play Now!

Put JSF to work

Build a real-world Web application with JavaServer Faces, the Spring Framework, and Hibernate

Summary
Building a real-world Web application using JavaServer Faces is not a trivial task. This article shows you how to integrate JSF, the Spring Framework, and Hibernate, and describes best practices and design guidelines for building a real-world Web application using these technologies. (*4,800 words;* **July 19, 2004**)

By Derek Yang Shen

JavaServer Faces (JSF) technology is a new user interface framework for J2EE applications. It is particularly suited, by design, for use with applications based on the MVC (Model-View-Controller) architecture. Numerous articles have introduced JSF. However, most take a highly theoretical approach that doesn't meet the challenges of real-world enterprise development. Many issues remain unsolved. For example, how does JSF fit into the overall MVC architecture? How does JSF integrate with other Java frameworks? Should business logic exist in the JSF backing beans? How do you handle security in JSF? And most importantly, how d[illegible]u build a real-world Web application using JSF?

This article addresses all those issues. It shows you how to integrate JSF with other Java frameworks—specifically, the Spring Framework and Hibernate. It demonstrates how to create the JCatalog Web application, an online product catalog system. Using the JCatalog example, this article covers each phase of Web application design, including business-requirement gathering, analysis, technology selection, high-level architecture, and implementation-level design. The article discusses the advantages and disadvantages of the technologies used in JCatalog and demonstrates approaches for designing some of the application's key aspects.

This article is written for Java architects, developers already working with J2EE-based Web applications. It is not an introduction to JSF, the Spring Framework, and Hibernate. Please see Resources if you are unfamiliar with these areas.

Functional requirements of the sample application

This article's sample application, JCatalog, is a real-world Web application, realistic enough to provide the basis for a meaningful discussion of a Web application's architectural decisions. I begin by presenting JCatalog's requirements. I refer back to this section throughout the article to address the technical decisions and architecture design.

The first phase in designing a Web application is to gather the system's functional requirements. The sample application is a typical e-business application system. Users can browse a product catalog and view product details, and administrators can manage the product catalog. Enhancements—e.g., inventory management and order processing—can be added to make the application a full-blown e-business system.

Use cases

Use-case analysis is used to access the sample application's functional requirements. Figure 1 is the application's use-case diagram.

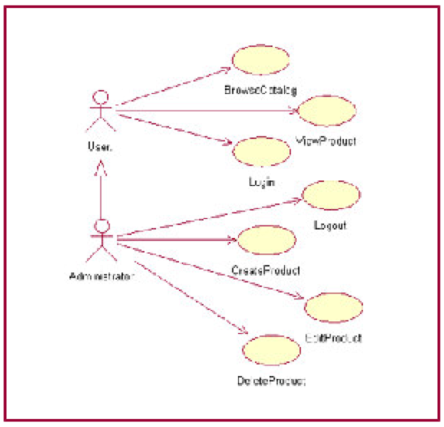

Figure 1. Use case diagram. Click on thumbnail to view full-size image.

A use-case diagram identifies the actors in a system and the operations they may perform. Seven use cases must be implemented in the sample application. Actor User can browse the product catalog and view product details. Once User logs in the system, she becomes actor Administrator, who can create new products, edit existing products, and delete old products.

Business rules
JCatalog must meet the following business rules:

- Each product has a unique product ID
- Each product belongs to at least one category
- The product ID cannot change once created

Assumptions
We make the following assumptions for the application's design and implementation:

- English is the default language; no internationalization is required
- No more than 500 products exist in the catalog
- The catalog is not updated frequently

Page flow
Figure 2 shows all of JCatalog's pages and the transitions among them.

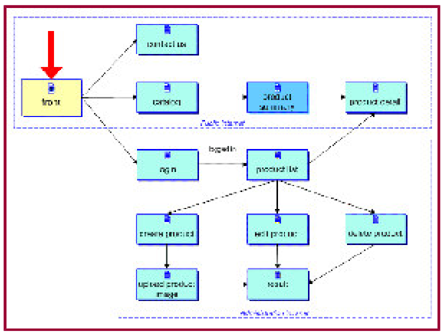

Figure 2. Page-flow diagram. Click on thumbnail to view full-size image.

The application has two groups of pages: public Internet and administration intranet. The intranet is accessible only to the users who log in the system successfully. ProductSummary is not presented to the users as a separate page. It displays in an HTML frame within the Catalog page. ProductList is a special catalog viewable only by the administrators. It contains links for creating, editing, and deleting products.

Figure 3 is a mock-up of the Catalog page. Ideally, for each page, a mock-up that details information for all the controls and the content required on the page should be included in the requirements documentation.

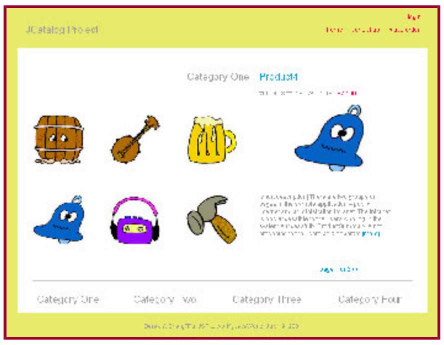

Figure 3. Mock-up of the Catalog page. Click on thumbnail to view full-size image.

High-level architecture design

The next phase in designing a Web application is the high-level architecture design. It involves subdividing the application into functional components and partitioning these components into tiers. The high-level architecture design is neutral to the technologies used.

Multitiered architecture
A multitiered architecture partitions the whole system into distinct functional units—client, presentation, business-logic, integration, and enterprise information system (EIS). This ensures a clean division of responsibility and makes the system more maintainable and extensible. Systems with three or more tiers prove more scalable and flexible than a client-server system, in which no business-logic middle tier exists.

The client tier is where the data model is consumed and presented. For a Web application, the client tier is normally the Web browser.

The browser-based thin client does not contain presentation logic; it relies on the presentation tier.

The presentation tier exposes the business-logic tier services to the users. It knows how to process a client request, how to interact with the business-logic tier, and how to select the next view to display.

The business-logic tier contains an application's business objects and business services. It receives requests from the presentation tier, processes the business logic based on the requests, and mediates access to the EIS tier's resources. Business-logic tier components benefit most from system-level services such as security management, transaction management, and resource management.

The integration tier is the bridge between the business-logic tier and the EIS tier. It encapsulates the logic to interact with the EIS tier. Sometimes, the combination of the integration tier and the business-logic tier is referred to as the *middle tier.*

Application data persists in the EIS tier. It contains relational databases, object-oriented databases, and legacy systems.

JCatalog's architecture design
Figure 4 shows JCatalog's high-level architecture design and how it fits into the multitiered architecture.

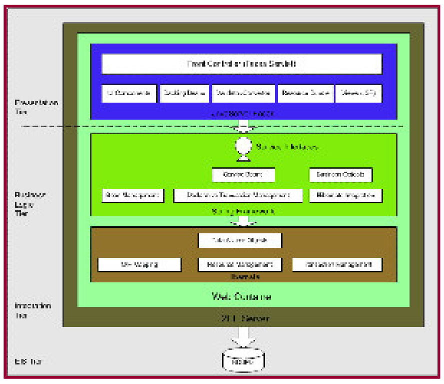

Figure 4. High-level architecture diagram. Click on thumbnail to view full-size image.

The application uses a multitiered nondistributed architecture. Figure 4 shows us the partitioning of the application tiers and the technologies chosen for each tier. It also serves as the sample application's deployment diagram. For a collocated architecture, the presentation, business-logic, and integration tiers are physically located in the same Web container. Well-defined interfaces isolate each tier's responsibility. The collocated architecture makes the application simple and scalable.

For the presentation tier, experience shows that the best practice is to choose an existing, proven Web application framework rather than designing and building a custom framework. We have several Web application frameworks to choose from, e.g., Struts, WebWork, and JSF. We use JSF for JCatalog.

Either EJB (Enterprise JavaBeans) or POJO (plain old Java objects) can be used to build the business-logic tier. EJB with remote interfaces is a better choice if the application is distributed. Since JCatalog is a typical Web application with no remote access required, POJO, with the help of the Spring Framework, is used to implement the business-logic tier.

The integration tier handles the data persistence with the relational database. Different approaches can be used to implement the integration tier:

- **Pure JDBC (Java Database Connectivity):** This is the most flexible approach; however, low-level JDBC is cumbersome to work with, and bad JDBC code does not perform well.
- **Entity beans:** An entity bean with container-managed persistence (CMP) is an expensive way to isolate data-access code and handle O/R (object-relational) mapping data persistence. It is an application-server-centric approach. An entity bean does not tie the application to a particular type of database, but does tie the application to the EJB container.
- **O/R mapping framework:** An O/R mapping framework takes an object-centric approach to implementing data persistence. An object-centric application is easy to develop and highly portable. Several frameworks exist under this domain—JDO (Java Data Objects), Hibernate, TopLink, and CocoBase are a few examples. We use Hibernate in the sample application.

Now let's discuss the design issues associated with each application tier. Since JSF is a relatively new technology, I emphasize its use.

Presentation tier and JavaServer Faces
The presentation tier collects user input, presents data, controls page navigation, and delegates user input to the business-logic tier. The presentation tier can also validate user input and maintain the application's session state. In the following sections, I discuss the presentation tier's design considerations and patterns, and the reason I chose JSF to implement JCatalog's presentation tier.

Model-View-Controller
MVC is the Java-BluePrints-recommended architectural design pattern for interactive applications. MVC separates design concerns, thereby decreasing code duplication, centralizing control, and making the application more extensible. MVC also helps developers with different skill sets focus on their core skills and collaborate through clearly defined interfaces. MVC is the architectural design pattern for the presentation tier.

JavaServer Faces
JSF is a server-side user interface component framework for Java-based Web applications. JSF contains an API for representing UI

components and managing their state; handling events, server-side validation, and data conversion; defining page navigation; supporting internationalization and accessibility; and providing extensibility for all these features. It also contains two JSP (JavaServer Pages) custom tag libraries for expressing UI components within a JSP page and for wiring components to server-side objects.

JSF and MVC
JSF fits well with the MVC-based presentation-tier architecture. It offers a clean separation between behavior and presentation. It leverages familiar UI-component and Web-tier concepts without limiting you to a particular scripting technology or markup language.

JSF backing beans are the model layer (more about backing beans in a later section). They also contain actions, which are an extension of the controller layer and delegate the user request to the business-logic tier. Please note, from the perspective of the overall application architecture, the business-logic tier can also be referred to as the model layer. JSP pages with JSF custom tags are the view layer. The Faces Servlet provides the controller's functionality.

Why JSF?
JSF is not just another Web framework. The following features differentiate JSF from other Web frameworks:

- **Swing-like object-oriented Web application development:** The server-side stateful UI component model with event listeners and handlers initiates object-oriented Web application development.

- **Backing-bean management:** Backing beans are JavaBeans components associated with UI components used in a page. Backing-bean management separates the definition of UI component objects from objects that perform application-specific processing and hold data. JSF implementation stores and manages these backing-bean instances in the proper scope.

- **Extensible UI component model:** JSF UI components are configurable, reusable elements that compose the user interfaces of JSF applications. You can extend the standard UI component and develop a more complex component, e.g., menu bar and tree component.

- **Flexible rendering model:** A renderer separates a UI component's functionality and view. Multiple renderers can be created and used to define different appearances of the same component for the same client or for different clients.

- **Extensible conversion and validation model:** Based on the standard converters and validators, you can develop customized converters and validators, which provide better model protection.

Despite its strength, JSF is not mature at its current stage. The components, converters, and validators that ship with JSF are basic. And the per-component validation model cannot handle many-to-many validation between components and validators. In addition, JSF custom tags cannot integrate with JSTL (JSP Standard Tag Library) seamlessly.

In the following sections, I discuss several key aspects and design decisions I made when implementing JCatalog with JSF. I start with a discussion of the definition and use of managed beans and backing beans in JSF. Then, I present how to handle security, pagination, caching, file upload, validation, and error-message customization in JSF.

Managed bean, backing bean, view object, and domain object model
JSF introduces two new terms: *managed bean* and *backing bean.* JSF provides a strong managed-bean facility. JavaBean objects managed by a JSF implementation are called managed beans. A managed bean describes how a bean is created and managed. It has nothing to do with the bean's functionalities.

The backing bean defines properties and handling-logics associated with the UI components used on the page. Each backing-bean property is bound to either a component instance or its value. A backing bean also defines a set of methods that perform functions for the component, such as validating the component's data, handling events that the component fires, and performing processing associated with navigation when the component activates.

A typical JSF application couples a backing bean with each page in the application. However, sometimes in the real world, forcing a one-to-one relationship between a backing bean and a page is not the ideal solution. It can cause problems like code duplication. In a real-world scenario, several pages may need to share the same backing bean behind the scenes. For example, in JCatalog, the CreateProduct and EditProduct page share the same `ProductBean` definition.

A view object is a model object used specifically in the presentation tier. It contains the data that must display in the view layer and the logic to validate user input, handle events, and interact with the business-logic tier. The backing bean is the view object in a JSF-based application. Backing bean and view object are interchangeable terms in this article.

Compared to the `ActionForm` and `Action` approach in Struts, development with backing beans in JSF follows better object-oriented design practices. A backing bean not only contains view data, but also behavior related to that data. In Struts, `Action` and `ActionForm` contain data and logic separately.

We've all heard about the domain object model. So, what's the difference between the domain object model and a view object? In a simple Web application, a domain object model can be used across all tiers, however, in a more complex Web application, a separate view object model needs to be used. Domain object model is about the business object and should belong in the business-logic tier. It contains the business data and business logic associated with the specific business object. A view object contains presentation-specific data and behavior. JCatalog's `ProductListBean` offers a good example. It contains data and logic specific to the presentation tier, e.g., pagination-related data and logic. The drawback to separating the view objects from the domain object model is that data mapping must occur between the two object models. In JCatalog, `ProductBeanBuilder` and `UserBeanBuilder` use the reflection-based Commons BeanUtils to implement the data mapping.

Security
Currently, JSF has no built-in security feature. The security requirement for the sample application is basic: only username and password-based authentication is needed for the user to log in to the administration intranet, and no authorization is required.

Several approaches have been proposed for handling user authentication in JSF:

- **Use a base backing bean:** This solution is simple. However, it ties the backing beans to a specific inheritance hierarchy.

- **Use a JSF `ViewHandler` decorator:** This way, the security logic is tightly coupled with a specific Web tier technology.

- **Use a servlet filter:** A JSF application is no different from other Java-based Web applications. It makes a filter the best place to handle authentication checking. This way, the authentication logic is decoupled from the Web application.

In the sample application, the `SecurityFilter` class handles user authentication. Currently, the protected resource contains only three pages, and their locations are hard-coded inside the `Filter` class for simplicity's sake. Enhancements can be made to externalize the security rules and protected resource to a configuration file.

Pagination
The application's Catalog page requires pagination. The presentation tier can handle pagination, which means all data must be retrieved and stored in that tier. Pagination can also be handled in the business-logic tier, integration tier, or even the EIS tier. One of JCatalog's assumptions is that no more than 500 products are in the catalog. All product information can fit into the user session. The pagination logic exists in the `ProductListBean` class. The pagination-related parameter "product per page" is configurable through the JSF managed-bean facility.

Caching
Caching is one of the most important techniques for improving performance in Web applications. Caching can be achieved in many tiers within the application architecture. It is most beneficial when one architectural tier can avoid calls to the tier beneath it. The JSF managed-bean facility makes caching in the presentation tier much easier. By changing a managed bean's scope, data contained by the managed bean can be cached within different scopes.

The sample application uses two-level caching. The first caching level exists inside the business-logic tier. The `CachedCatalogServiceImpl` class maintains a read/write cache for all products and categories. Spring manages the class as a singleton service bean. So, the first-level cache is an application-scope read/write cache.

To simplify the pagination logic and further speed up the application, products are also cached inside the presentation tier in the session scope. Each user maintains his own `ProductListBean` inside the session. The penalties of this approach are system memory and stale data. Within the duration of a user session, the user may see stale catalog data if administrator users update the catalog. However, based on the assumptions, since no more than 500 products exist in the catalog and the catalog is not updated frequently, we should be able to live with these penalties.

File upload
The current JSF Sun reference implementation does not support file upload. Struts has good file upload capabilities, however the Struts-Faces integration library is needed to use the feature. In JCatalog, an image is associated with each product. Aft[illegible] user creates a new product, she must upload the image associated with it. The image is stored inside the application server's filesystem. The product ID is the image name.

The sample application uses `<input type="file">`, Servlet and Jakarta Commons' file-upload API, to implement a simple file upload utility. The utility takes two parameters: the product image directory and the image upload result page. They are configurable through the `ApplicationBean`. Please refer to the `FileUploadServlet` class for details.

Validation
The standard validators shipped with JSF are basic and may not meet many real-world requirements. Developing your own JSF validator is easy. I developed the `SelectedItemsRange` validator with a custom tag in the sample application. It validates the number of items selected by the `UISelectMany` UI component:

```
<h:selectManyListbox value="#{productBean.selectedCategoryIds}" id="selectedCategoryIds">
   <catalog:validateSelectedItemsRange minNum="1"/>
   <f:selectItems value="#{applicationBean.categorySelectItems}" id="categories"/>
</h:selectManyListbox>
```

Please refer to the sample application for more details.

Error-message customization
In JSF, you can set up resource bundles and customize the error messages for converters and validators. A resource bundle is set up inside `faces-config.xml`:

```
<message-bundle>catalog.view.bundle.Messages</message-bundle>
```

The error message's key-value pairs are added to the `Message.properties` file:

```
#conversion error messages
javax.faces.component.UIInput.CONVERSION=Input data is not in the correct type.

#validation error messages
javax.faces.component.UIInput.REQUIRED=Required value is missing.
```

Business-logic tier and the Spring Framework
Business objects and business services exist in the business-logic tier. A business object contains not only the data, but also the logic associated with that specific object. Three business objects have been identified in the sample application: `Product`, `Category`, and `User`.

Business services interact with business objects and provide higher-level business logic. A formal business interface layer should be defined, which contains the service interfaces that the client uses directly. POJO, with the help of the Spring Framework, implements the business-logic tier in JCatalog. There are two business services: `CatalogService` contains the catalog management-related business logic, and `UserService` contains the user management logic.

Spring is based on the concept of inversion of control (IOC). Spring features used in the sample application include:

- **Bean management with application contexts:** Spring can effectively organize our middle tier objects and handles plumbing for us. Spring can eliminate the proliferation of singletons and facilitates good object-oriented programming practices, e.g., programming to interfaces.

- **Declarative transaction management:** Spring uses AOP (aspect-oriented programming) to deliver declarative transaction management without using an EJB container. This way, transaction management can be applied to any POJO. Spring transaction management is not tied to JTA (Java Transaction API) and can work with different transaction strategies. Declarative transaction management with Hibernate transaction is used in the sample application.

- **Data-access exception hierarchy:** Spring provides a meaningful exception hierarchy in place of `SQLException`. To use the Spring data-access exception hierarchy, the Spring data-access exception translator must be defined within the Spring configuration file:

```
<bean id="jdbcExceptionTranslator" class= "org.springframework.jdbc.support.SQLErrorCodeSQLExceptionTranslator">
  <property name="dataSource">
    <ref bean="dataSource"/>
  </property>
</bean>
```

  In the sample application, if a new product with a duplicate ID is inserted, a `DataIntegrityViolationException` is thrown. The exception is caught and rethrown as a `DuplicateProductIdException`. This way, the `DuplicateProductIdException` can be handled differently from other data-access exceptions.

- **Hibernate integration:** Spring does not force us to use its strong JDBC abstraction feature. It integrates well with O/R mapping frameworks, especially Hibernate. Spring offers efficient and safe handling of Hibernate sessions, handles the configuration of Hibernate `SessionFactorie`s and JDBC data sources in application contexts, and makes the application easier to test.

Integration tier and Hibernate
Hibernate is an open source O/R mapping framework that relieves the need to use the JDBC API. Hibernate supports all major SQL database management systems. The Hibernate Query Language, designed as a minimal object-oriented extension to SQL, provides an elegant bridge between the object and relational worlds. Hibernate offers facilities for data retrieval and update, transaction management, database connection pooling, programmatic and declarative queries, and declarative entity relationship management.

Hibernate is less invasive than other O/R mapping frameworks. Reflection and runtime bytecode generation are used, and SQL generation occurs at system startup. It allows us to develop persistent objects following common Java idiom—including association, inheritance, polymorphism, composition, and the Java Collections Framework. The business objects in the sample application are POJO and do not need to implement any Hibernate-specific interfaces.

Data Access Object (DAO)
The DAO pattern is used in JCatalog. This pattern abstracts and encapsulates all access to the data source. The application has two DAO interfaces: `CatalogDao` and `UserDao`. Their implementation classes, `HibernateCatalogDaoImpl` and `HibernateUserDaoImpl` contain Hibernate-specific logic to manage and persist data.

Implementation design
Now let's see how to wire everything together and implement JCatalog. You can download the application's full source code from Resources.

Database design
We create a schema named Catalog for the sample application, which consists of four tables, as shown in Figure 5:

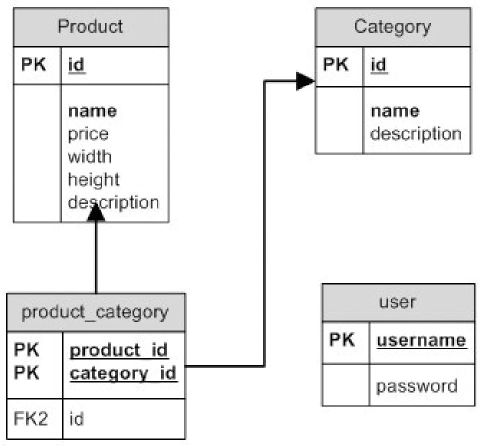

Figure 5. Database schema diagram

Class Design
Figure 6 illustrates JCatalog's class diagram.

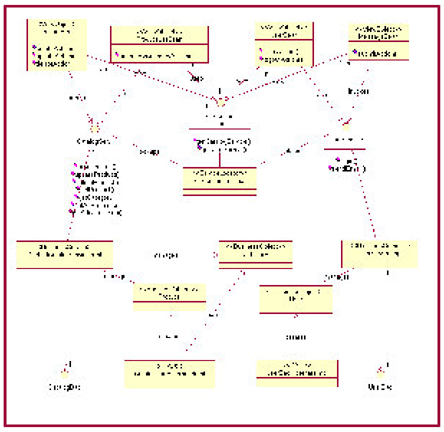

Figure 6. Class diagram. Click on thumbnail to view full-size image.

Programming against interfaces is used throughout the design. In the presentation tier, four backing beans are used: `ProductBean`, `ProductListBean`, `UserBean`, and `MessageBean`. The business-logic tier contains two business services (`CatalogService` and `UserService`) and three business objects (`Product`, `Category`, and `User`). The integration tier contains two DAO interfaces and their Hibernate implementations. The Spring application contexts wire and manage most of the object beans inside the business-logic and integration tiers. `ServiceLocator` integrates JSF with the business-logic tier.

Wire everything up
Because of this article's space limitation, we examine only one use case. The sample use case CreateProduct demonstra how to wire everything up and build the application. Before we dive into the details, let's use a sequence diagram (Figure 7) to demonstrate the end-to-end integration of all the tiers:

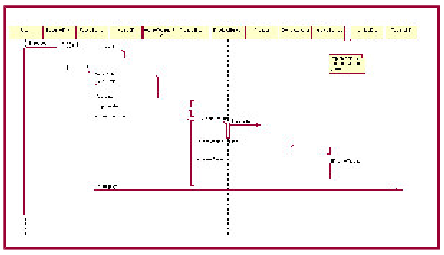

Figure 7. Sequence diagram of the CreateProduct use case. Click on thumbnail to view full-size image.

Now let's walk through each tier and discuss more details about how to implement CreateProduct.

Presentation tier
The presentation tier implementation involves creating the JSP pages, defining the page navigations, creating and configuring the backing beans, and integrating JSF with the business-logic tier.

- **JSP page:** createProduct.jsp is the page for creating a new product. It contains UI components and wires the components to the `ProductBean`. The `ValidateItemsRange` custom tag validates the number of categories the user selected. At least one category should be selected for each new product.

- **Page navigation:** Navigation for the application is defined in the application configuration file, `faces-navigation.xml`. The navigation rules defined for CreateProduct are:

```
<navigation-rule>
  <from-view-id>*</from-view-id>
  <navigation-case>
    <from-outcome>createProduct</from-outcome>
    <to-view-id>/createProduct.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
<navigation-rule>
  <from-view-id>/createProduct.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/uploadImage.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>retry</from-outcome>
```

```
      <to-view-id>/createProduct.jsp</to-view-id>
   </navigation-case>
   <navigation-case>
      <from-outcome>cancel</from-outcome>
      <to-view-id>/productList.jsp</to-view-id>
   </navigation-case>
</navigation-rule>
```

- **Backing bean:** The ProductBean contains not only the properties maps to the data for the UI components on the page, but also three actions: createAction, editAction, and deleteAction. Here's the code for the createAction() method:

```
public String createAction() {
   try {
      Product product = ProductBeanBuilder.createProduct(this);

      //Save the product.
      this.serviceLocator.getCatalogService().saveProduct(product);

      //Store the current product id inside the session bean.
      //For the use of image uploader.
      FacesUtils.getSessionBean().setCurrentProductId(this.id);

      //Remove the productList inside the cache.
      this.logger.debug("remove ProductListBean from cache");
      FacesUtils.resetManagedBean(BeanNames.PRODUCT_LIST_BEAN);
   } catch (DuplicateProductIdException de) {
      String msg = "Product id already exists";
      this.logger.info(msg);
      FacesUtils.addErrorMessage(msg);

      return NavigationResults.RETRY;
   } catch (Exception e) {
      String msg = "Could not save product";
      this.logger.error(msg, e);
      FacesUtils.addErrorMessage(msg + ": Internal Error");

      return NavigationResults.FAILURE;
   }
   String msg = "Product with id of " + this.id + " was created successfully.";
   this.logger.debug(msg);
   FacesUtils.addInfoMessage(msg);

   return NavigationResults.SUCCESS;
}
```

Inside the action, a Product business object is built based on ProductBean's properties. ServiceLocator looks up the CatalogService. Finally, the createProduct request is delegated to the CatalogService, which is in the business-logic tier.

- **Managed-bean declaration:** The ProductBean must be configured in the JSF configuration resource file faces-managed-bean.xml:

```
<managed-bean>
   <description>
      Backing bean that contains product information.
   </description>
   <managed-bean-name>productBean</managed-bean-name>
   <managed-bean-class>catalog.view.bean.ProductBean</managed-bean-class>
   <managed-bean-scope>request</managed-bean-scope>
   <managed-property>
      <property-name>id</property-name>
      <value>#{param.productId}</value>
   </managed-property>
   <managed-property>
      <property-name>serviceLocator</property-name>
      <value>#{serviceLocatorBean}</value>
   </managed-property>
</managed-bean>
```

The ProductBean is set to have a scope of request, which means the JSF implementation creates a new ProductBean instance for each request if ProductBean is referenced inside the JSP page. The ID-managed property is populated by the request parameter productId. The JSF implementation gets the parameter from the request and sets the managed property.

- **Integration between presentation and business-logic tiers:** ServiceLocator abstracts the logic to look for services. In the sample application, ServiceLocator is defined as an interface. The interface is implemented as a JSF managed bean, ServiceLocatorBean, which looks up the services from the Spring application context:

```
ServletContext context = FacesUtils.getServletContext();
this.appContext = WebApplicationContextUtils.getRequiredWebApplicationContext(context);
this.catalogService = (CatalogService)this.lookupService(CATALOG_SERVICE_BEAN_NAME);
this.userService = (UserService)this.lookupService(USER_SERVICE_BEAN_NAME);
```

The ServiceLocator is defined as a property inside the BaseBean. The JSF managed bean facility wires the ServiceLocator implementation with those managed beans that must access ServiceLocator. Inversion of control is used.

Business-logic tier
The tasks in this tier consist of defining the business objects, creating the service interfaces with their implementations, and wiring the objects with Spring.

- **Business objects:** Since Hibernate provides persistence, the `Product` and `Category` business objects need to provide getter and setter methods for all fields that they contain.

- **Business services:** The `CatalogService` interface defines all of the catalog management-related services:

  ```
  public interface CatalogService {
     public Product saveProduct(Product product) throws CatalogException;
     public void updateProduct(Product product) throws CatalogException;
     public void deleteProduct(Product product) throws CatalogException;
     public Product getProduct(String productId) throws CatalogException;
     public Category getCategory(String categoryId) throws CatalogException;
     public List getAllProducts() throws CatalogException;
     public List getAllCategories() throws CatalogException;
  }
  ```

  The `CachedCatalogServiceImpl` is the service interface implementation, which contains a setter for a `CatalogDao` object. Spring wires the `CachedCatalogServiceImpl` with the `CatalogDao` object. Because we are coding to interfaces, we do not tightly couple the implementations.

- **Spring configuration:** Here's the `CatalogService`'s Spring configuration:

  ```
  <!-- Hibernate Transaction Manager Definition -->
  <bean id="transactionManager" class="org.springframework.orm.hibernate.HibernateTransactionManager">
     <property name="sessionFactory"><ref local="sessionFactory"/></property>
  </bean>

  <!-- Cached Catalog Service Definition -->
  <bean id="catalogServiceTarget" class="catalog.model.service.impl.CachedCatalogServiceImpl" init-method="init"
     <property name="catalogDao"><ref local="catalogDao"/></property>
  </bean>

  <!-- Transactional proxy for the Catalog Service -->
  <bean id="catalogService" class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
     <property name="transactionManager"><ref local="transactionManager"/></property>
     <property name="target"><ref local="catalogServiceTarget"/></property>
     <property name="transactionAttributes">
        <props>
           <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
         <prop key="save*">PROPAGATION_REQUIRED</prop>
         <prop key="update*">PROPAGATION_REQUIRED</prop>
         <prop key="delete*">PROPAGATION_REQUIRED</prop>
        </props>
     </property>
  </bean>
  ```

  Spring declarative transaction management is set up for the `CatalogService`. `CatalogService` can be wired with a different `CatalogDao` implementation. Spring creates and manages a `CatalogService` singleton object, and no factory is needed.

  Now that the business-logic tier is ready, let's wire it to the integration tier.

- **Integration between Spring and Hibernate:** Here's the `HibernateSessionFactory`'s configuration:

  ```
  <!-- Hibernate SessionFactory Definition -->
  <bean id="sessionFactory" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
     <property name="mappingResources">
        <list>
           <value>catalog/model/businessobject/Product.hbm.xml</value>
           <value>catalog/model/businessobject/Category.hbm.xml</value>
           <value>catalog/model/businessobject/User.hbm.xml</value>
        </list>
     </property>
     <property name="hibernateProperties">
        <props>
           <prop key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop>
         <prop key="hibernate.show_sql">true</prop>
         <prop key="hibernate.cglib.use_reflection_optimizer">true</prop>
         <prop key="hibernate.cache.provider_class">net.sf.hibernate.cache.HashtableCacheProvider</prop>
        </props>
     </property>
     <property name="dataSource">
        <ref bean="dataSource"/>
     </property>
  </bean>
  ```

  CatalogDao uses `HibernateTemplate` to integrate between Hibernate and Spring. Here's the configuration for `HibernateTemplate`:

  ```
  <!-- Hibernate Template Defintion -->
  <bean id="hibernateTemplate" class="org.springframework.orm.hibernate.HibernateTemplate">
  ```

```
    <property name="sessionFactory"><ref bean="sessionFactory"/></property>
    <property name="jdbcExceptionTranslator"><ref bean="jdbcExceptionTranslator"/></property>
  </bean>
```

Integration Tier
Hibernate maps business objects to the relational database using an XML configuration file. In JCatalog, `Product.hbm.xml` expresses the mapping for the `Product` business object. `Category.hbm.xml` is used for the `Category` business object. The configuration files are in the same directory as the corresponding business objects. Here's the `Product.hbm.xml`:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping package="catalog.model.businessobject">
  <class name="Product" table="product">
    <id name="id" column="ID" unsaved-value="null">
      <generator class="assigned"/>
    </id>
    <property name="name" column="NAME" unique="true" not-null="true"/>
    <property name="price" column="PRICE"/>
    <property name="width" column="WIDTH"/>
    <property name="height" column="height"/>
    <property name="description" column="description"/>
    <set name="categoryIds" table="product_category" cascade="all">
      <key column="PRODUCT_ID"/>
      <element column="CATEGORY_ID" type="string"/>
    </set>
  </class>
</hibernate-mapping>
```

`CatalogDao` is wired with `HibernateTemplate` by Spring:

```
<!-- Catalog DAO Definition: Hibernate implementation -->
<bean id="catalogDao" class="catalog.model.dao.hibernate.CatalogDaoHibernateImpl">
  <property name="hibernateTemplate"><ref bean="hibernateTemplate"/></property>
</bean>
```

Conclusion
This article shows you how to integrate JSF with the Spring Framework and Hibernate and build a real-world Web application. The combination of these three technologies provides a solid Web application development framework. A multitiered architecture should be used as the high-level architecture for Web applications. JSF fits into the MVC design pattern very well and can be used to implement the presentation tier. The Spring Framework can be used in the business-logic tier to manage business objects, and provide declarative transaction management and resource management. Spring integrates with Hibernate very well. Hibernate is a powerful O/R mapping framework and can provide the best services inside the integration tier.

By partitioning the whole Web application into tiers and programming against interfaces, the technology used for each application tier can be replaced. For example, Struts can take the place of JSF for the presentation tier, and JDO can replace Hibernate in the integration tier. Integration between the application tiers is not trivial. The use of inversion of control and the Service Locator design pattern can make it easier. JSF provides functionalities other Web frameworks like Struts lack. However, that does not mean you should dump Struts and start using JSF right away. Whether or not JSF should be used as the Web framework for your project depends on your project's status and functional requirements, and your team's expertise.

About the author
Derek Yang Shen has been working with J2EE exclusively for the past five years, with expertise in the development of multitiered Web applications and complex B2B systems. He is a Sun Certified Enterprise Architect. Shen holds a master's degree in computer science from the University of California at Los Angeles. He currently works with a large Internet company as a J2EE architect.

Resources

- Download the JCatalog project sample application:
  http://www.javaworld.com/javaworld/jw-07-2004/jsf/jw-0719-jsf.zip
- Official JavaServer Faces site:
  http://java.sun.com/j2ee/javaserverfaces/index.jsp
- A good JSF tutorial can be found in *The J2EE 1.4 Tutorial* (Chapters 17 to 21):
  http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html
- More articles and books on JSF:
  http://jsfcentral.com/reading/index.html
- Official Spring Framework site:
  http://www.springframework.org
- Good introduction to the Spring Framework by Rod Johnson:
  http://www.theserverside.com/articles/article.tss?l=SpringFramework
- Rod Johnson's book *Expert One-on-One J2EE Design and Development* (Wrox, October 2002; ISBN: 0764543857) is the corner stone of the Spring Framework:
  http://www.wrox.com/WileyCDA/WroxTitle/productCd-0764543857.html
- Official Hibernate site:
  http://www.hibernate.org
- Online documentation of Hibernate:
  http://www.hibernate.org/hib_docs/reference/en/html/
- Introduction to the integration between the Spring Framework and Hibernate:
  http://hibernate.bluemars.net/110.html
- "Designing Enterprise Applications with the J2EE Platform, Second Edition" is a good introduction to the multitiered architecture

and MVC design pattern:
http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/index.html
- Commons BeanUtils:
http://jakarta.apache.org/commons/beanutils/
- Commons FileUpload:
http://jakarta.apache.org/commons/fileupload/
- For more on JavaServer Faces, read the following *JavaWorld* articles by David Geary:
    - "A First Look at JavaServer Faces, Part 1" (November 2002)
    - "A First Look at JavaServer Faces, Part 2" (December 2002)
    - "JavaServer Faces, Redux" (November 2003)
- Browse the **JavaServer Pages** section of *JavaWorld*'s Topical Index:
http://www.javaworld.com/channel_content/jw-jsp-index.shtml
- Browse the **Enterprise Java** section of *JavaWorld*'s Topical Index:
http://www.javaworld.com/channel_content/jw-enterprise-index.shtml

Advertisement: Support JavaWorld, click here!