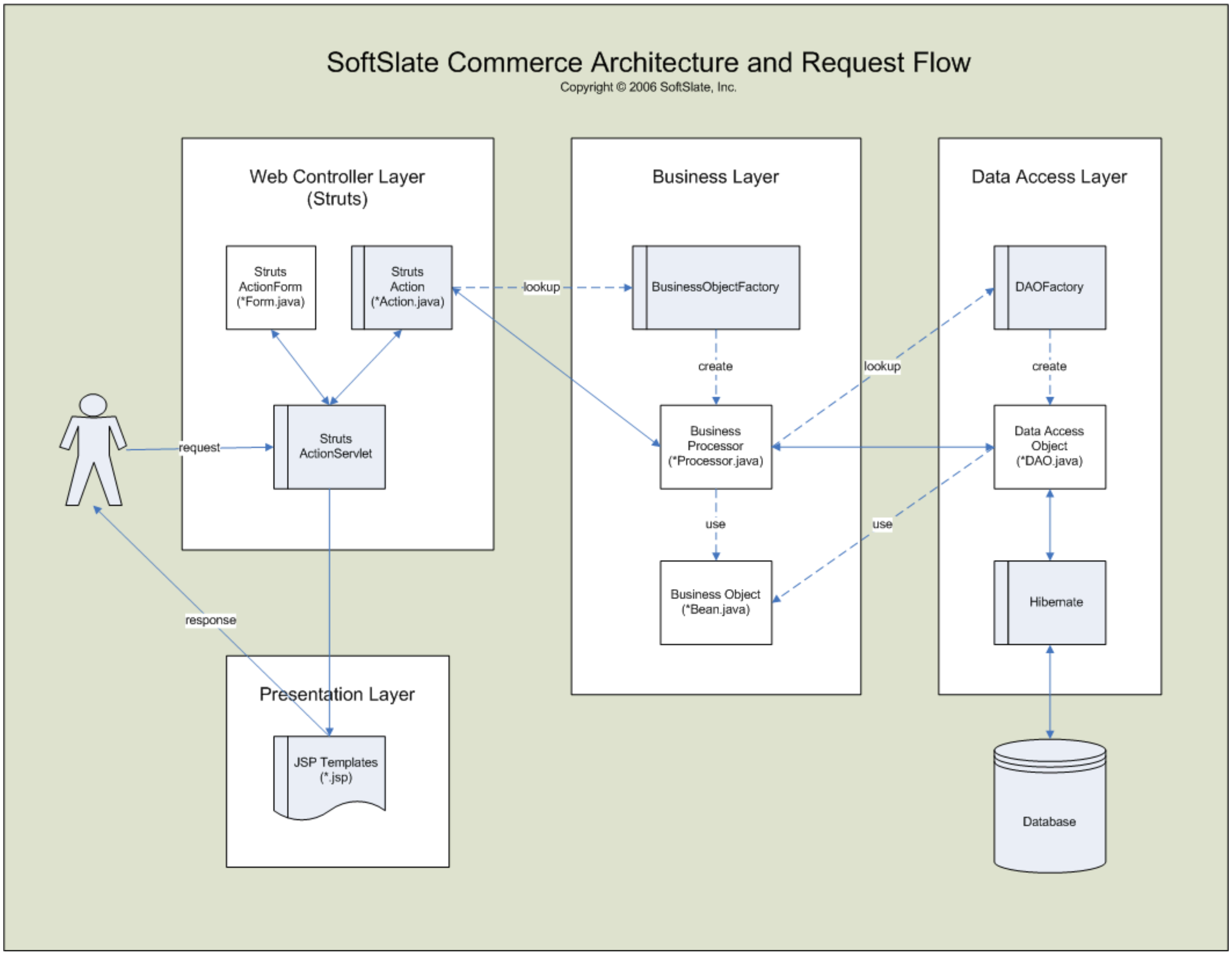
SoftSlate Commerce
Guide for Developers

rev. 6/2014

Table of Contents

1. Application Architecture

SoftSlate Commerce uses a Model, View, Controller (MVC) architecture to organize its classes and files. The Model is further separated into two sections: classes devoted to interacting with the database (DAOs), and classes devoted to processing business logic (Business Objects). The result is four distinct layers to the application, illustrated by the following flow chart:


SoftSlate Commerce Architecture and Request Flow
Copyright © 2006 SoftSlate, Inc.
Web Controller Layer (Struts)
Struts ActionForm (*Form.java)
Struts Action (*Action.java)
Struts ActionServlet
request
response
lookup
Business Layer
BusinessObjectFactory
create
Business Processor (*Processor.java)
use
Business Object (*Bean.java)
lookup
use
Data Access Layer
DAOFactory
create
Data Access Object (*DAO.java)
Hibernate
Database
Presentation Layer
JSP Templates (*.jsp)

2. The Web Controller or Struts Layer

Overview

The Web Controller or Struts layer provides the mechanisms that control the routing of each web request through the system (the "C" in MVC). In SoftSlate Commerce, for the customer and administrator interfaces, the Web Controller or Struts layer consists of the following classes and files:

- The `struts-config*.xml` files under the `WEB-INF/conf` subdirectories (for the customer interface).
- The `struts-config*.xml` files under the `WEB-INF/conf/administrator` subdirectories (for the administrator interface).
- The classes in the `com.softslate.commerce.customer` package, most of which are subclasses of `org.apache.struts.action.ActionForm` and `org.apache.struts.action.Action` (for the customer interface).
- The classes in the `com.softslate.commerce.administrator` package, most of which are subclasses of `org.apache.struts.action.ActionForm` and `org.apache.struts.action.Action` (for the administrator interface).

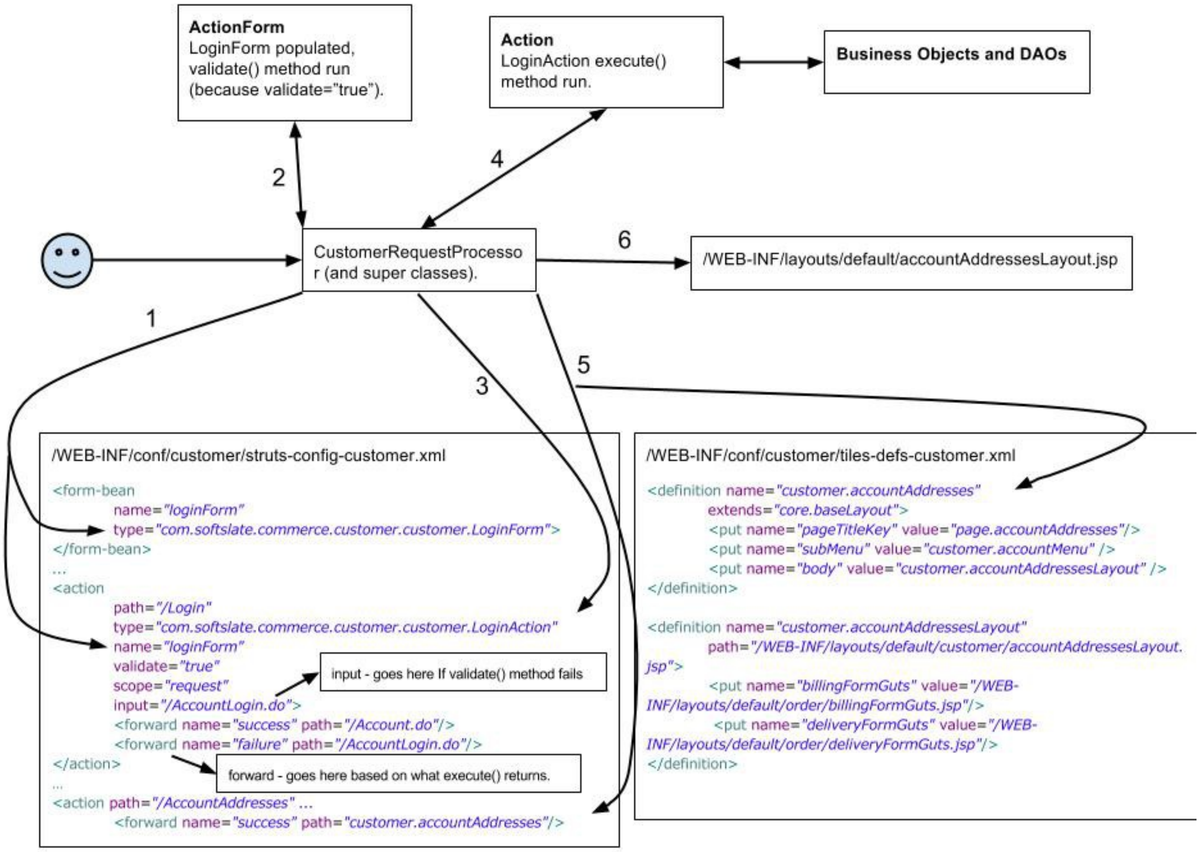
In short, the role of this layer is to take in each browser request, to validate the browser request, to invoke the classes in the Business layer responsible for processing the request, possibly to handle certain processing itself (such as recording information in the browser session), and finally to route the request to the appropriate Tiles definition in the Presentation layer.

Why separate the Web Controller into its own layer? Along with the Presentation layer, the Web Controller or Struts layer encapsulates all the processes that relate to SoftSlate Commerce being a Web application (as opposed to a GUI or some other application). For example, instances of `javax.servlet.http.HttpServletRequest` or `javax.servlet.http.HttpSession` are never referred to or used in the Business or Data Access layers. This separation makes it possible to use the same back-end processing with different front-end systems, such as legacy applications.

> **Note**
>
> Because of its use of the Struts web application framework, a familiarity with Struts is very useful when developing with SoftSlate Commerce. A great amount of information is available online and in books about Struts. For more, visit the Struts home page at http://struts.apache.org.

Components of the Web Controller (Struts) Layer

Detailed Example of Process Flow in the Web Controller (Struts) Layer:


ActionForm
LoginForm populated, validate() method run (because validate="true").
Action
LoginAction execute() method run.
Business Objects and DAOs
CustomerRequestProcessor (and super classes).
/WEB-INF/layouts/default/accountAddressesLayout.jsp
1
2
3
4
5
6
input - goes here If validate() method fails
forward - goes here based on what execute() returns.

/WEB-INF/conf/customer/struts-config-customer.xml

```
<form-bean
        name="loginForm"
        type="com.softslate.commerce.customer.customer.LoginForm">
</form-bean>
...
<action
        path="/Login"
        type="com.softslate.commerce.customer.customer.LoginAction"
        name="loginForm"
        validate="true"
        scope="request"
        input="/AccountLogin.do">
        <forward name="success" path="/Account.do"/>
        <forward name="failure" path="/AccountLogin.do"/>
</action>
...
<action path="/AccountAddresses" ...
        <forward name="success" path="customer.accountAddresses"/>
```

/WEB-INF/conf/customer/tiles-defs-customer.xml

```
<definition name="customer.accountAddresses"
        extends="core.baseLayout">
        <put name="pageTitleKey" value="page.accountAddresses"/>
        <put name="subMenu" value="customer.accountMenu" />
        <put name="body" value="customer.accountAddressesLayout" />
</definition>

<definition name="customer.accountAddressesLayout"
        path="/WEB-INF/layouts/default/customer/accountAddressesLayout.
jsp">
        <put name="billingFormGuts" value="/WEB-
INF/layouts/default/order/billingFormGuts.jsp"/>
         <put name="deliveryFormGuts" value="/WEB-
INF/layouts/default/order/deliveryFormGuts.jsp"/>
</definition>
```

ActionForms

A bean class representing the submitted parameters of a given request. ActionForms might have a validate() method where basic validation on the parameters is done.

ActionForm Example

`/WEB-INF/layouts/default-html5/customer/login.jsp`

```
...
<input type=”text” name=”userName”>
<input type=”text” name=”decryptedPassword”>
...
```

`com.softslate.commerce.customer.customer.LoginForm`

```
package com.softslate.commerce.customer.customer;
...
public class LoginForm extends BaseForm {
      private String userName = null;
      public String getUserName() {
```

```
		return (this.userName);
	}
	public void setUserName(String userName) {
		this.userName = userName;
	}
	private String decryptedPassword = null;
	public String getDecryptedPassword() {
		return (this.decryptedPassword);
	}
	public void setDecryptedPassword(String decryptedPassword) {
		this.decryptedPassword = decryptedPassword;
	}
	…
	public ActionErrors validate(ActionMapping mapping, HttpServletRequest
request) {
		if (getUserName() == null || getUserName().length() == 0) {
			String message = getMessage("customer", "customer.userName",
null);
			getErrors().add(ActionMessages.GLOBAL_MESSAGE, new
ActionMessage("errors.required", message));
		}
		…
		return getErrors();
	}
}
```

ActionForm base superclasses

All Action Forms in SoftSlate subclass either BaseForm or BaseDynaForm:

com.softslate.commerce.customer.core.BaseForm
com.softslate.commerce.customer.core.BaseDynaForm

These classes serve the same functions, and BaseDynaForm is essentially deprecated from our point of view (it uses a Struts feature where properties can be defined in the XML configuration. We find it's best to define the properties in the form class itself.)

These classes provide the following properties, which are populated by the initializeProperties() method in the base superclasses:

- `request`: The HttpServletRequest object for the current request.
- `messages`: Struts uses the ActionMessages class to store messages that should be displayed to the `customer`. This object stores success messages or informational messages that end up showing up on the screen here (admin side example):
- `errors`: Another ActionMessages object that stores error messages corresponding to the ActionForm:
- `settings`: A copy of the global application settings. (Eg. settings.getValue("databaseObjectsVersion") gets you the current version of SoftSlate).
- `user`: Representing the current user of the application (eg.

getUser().getAdministrator().getUserName() gets you the current logged in administrator's user name. getUser().getCustomer().getUserName() gets the current logged in customer's user name.)

- `businessObjectFactory`: Allowing Struts actions to create Business Objects for back-end processing.

For ActionForms in the Administrator application there is BaseAdministratorForm, which subclasses BaseForm and provides an additional `administrator` property representing the current logged in Administrator for the request.

Actions

A Struts Action class is used to process a given request. The Action class for a given request is specified in the Struts configuration file with the “path” attribute of each action mapping. Struts sends a populated ActionForm instance into the Action class's execute() method, where the Action class can retrieve everything it needs via its properties, including a BusinessObjectFactory, Settings, the request parameters, etc. This class typically calls a BusinessProcessor to process the backend of the request.

`com.softslate.commerce.customer.customer.LoginAction`

```
package com.softslate.commerce.customer.customer;
...
public class LoginAction extends BaseAction {
      public ActionForward execute(ActionMapping mapping, ActionForm form,
                  HttpServletRequest request, HttpServletResponse response)
                  throws Exception {
            log.info("Starting action.");
            LoginForm baseForm = (LoginForm) form;
            CustomerProcessor customerProcessor = baseForm
                              .getBusinessObjectFactory().createObject(
                                          CustomerProcessor.class);
                  Map result = customerProcessor.processLogin(parameters);
                  …
            }
      }
}
```

Tiles definitions

Tiles definitions are XML configurations that tell the system which JSPs to use for the output of each request. It is possible to nest Tiles definitions.

`/WEB-INF/conf/customer/tiles-defs-customer.xml`

```
    <definition name="customer.accountAddresses" extends="core.baseLayout">
      <put name="pageTitleKey" value="page.accountAddresses"/>
        <put name="subMenu" value="customer.accountMenu" />
        <put name="body" value="customer.accountAddressesLayout" />
    </definition>
...
```

```
      <definition name="customer.accountAddressesLayout" path="/WEB-
INF/layouts/default/customer/accountAddressesLayout.jsp">
             <put name="billingFormGuts" value="/WEB-
INF/layouts/default/order/billingFormGuts.jsp"/>
            <put name="deliveryFormGuts" value="/WEB-
INF/layouts/default/order/deliveryFormGuts.jsp"/>
      </definition>
```

Struts configurations

XML files that define the flow of each request, including which ActionForms correspond to which Actions, and which Tiles definitions are used for the output. If you want to know what happens with X request, start with the Struts configuration files. Find the action mapping for the request (the mapping whose path attribute corresponds to the URL minus the .do extension) and follow the processing from there.

`/WEB-INF/conf/customer/struts-config-customer.xml`

```
      <form-bean
            name="loginForm"
            type="com.softslate.commerce.customer.customer.LoginForm">
      </form-bean>
      ...
      <action
            path="/Login"
           type="com.softslate.commerce.customer.customer.LoginAction"
            name="loginForm"
            validate="true"
            scope="request"
            input="/AccountLogin.do">
            <forward name="success" path="/Account.do"/>
            <forward name="failure" path="/AccountLogin.do"/>
      </action>
```

Full Example: Customer-side Login Form and Processing

`/WEB-INF/layouts/default/customer/login.jsp`

```
<form action=”Login.do”>
      <input type=”text” name=”userName”>
      <input type=”text” name=”decryptedPassword”>
</form>
```

`/WEB-INF/conf/customer/struts-config-customer.xml`

```
      <form-bean
            name="loginForm"
            type="com.softslate.commerce.customer.customer.LoginForm">
      </form-bean>
      ...
      <action
```

```
                path="/Login"
                type="com.softslate.commerce.customer.customer.LoginAction"
                name="loginForm"
                validate="true"
                scope="request"
                input="/AccountLogin.do">
                <forward name="success" path="/Account.do"/>
                <forward name="failure" path="/AccountLogin.do"/>
        </action>
        ...
        <action
                path="/Account"
                type="com.softslate.commerce.customer.customer.AccountAction">
                <forward name="catalogMode" path="/Welcome.do"/>
                <forward name="accountLogin" path="/AccountLogin.do"/>
                <forward name="accountHome" path="/AccountAddresses.do"/>
        </action>
        …
        <action
                path="/AccountAddresses"
                type="com.softslate.commerce.customer.customer.
                AccountAddressesAction"
                name="addressForm"
                validate="false"
                scope="request">
                <forward name="success" path="customer.accountAddresses"/>
                <forward name="notLoggedIn" path="/AccountLogin.do"/>
        </action>
```

`com.softslate.commerce.customer.customer.LoginForm`

```java
package com.softslate.commerce.customer.customer;
...
public class LoginForm extends BaseForm {
        …
        public ActionErrors validate(ActionMapping mapping, HttpServletRequest
request) {
                if (getUserName() == null || getUserName().length() == 0) {
                        String message = getMessage("customer", "customer.userName",
null);
                        getErrors().add(ActionMessages.GLOBAL_MESSAGE, new
                        ActionMessage("errors.required", message));
                }
                …
                return getErrors();
        }
}
```

`com.softslate.commerce.customer.customer.LoginAction`

```
package com.softslate.commerce.customer.customer;
...
public class LoginAction extends BaseAction {
      public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response)
            throws Exception {
            log.info("Starting action.");
            LoginForm baseForm = (LoginForm) form;
            CustomerProcessor customerProcessor = baseForm
                  .getBusinessObjectFactory().createObject(CustomerProcessor.class)
;
```

```
            Map result = customerProcessor.processLogin(parameters);
            …
    }
}
```

Customizing the Web Controller (Struts) Layer

1. By convention, you should name ActionForm classes *Form.java (LoginForm.java).

2. By convention, you should name Action classes *Action.java (LoginAction.java).

3. There is a particular type of Action class that does not process a form, but rather prepares a form to be displayed to the user (the form's fields might need to be prepopulated for the user from his session, or from the database). By convention, you should name these form-preparation Action classes *FormAction.java. Eg, CartItemEditFormAction.java.

4. All new Struts configurations should go in /WEB-INF/conf/core/struts-config-custom.xml for the customer interface or /WEB-INF/conf/administrator/core/struts-config-custom.xml for the administrator interface. (These files are not overwritten during upgrades.)

5. All new Tiles definitions should go in /WEB-INF/conf/core/tiles-defs-custom.xml for the customer interface or /WEB-INF/conf/administrator/core/tiles-defs-custom.xml for the administrator interface. (Again, these files are not overwritten during upgrades.)

6. Note that if the same mapping or tiles definition exists in one of the custom configuration files, as another Struts configuration file, the custom definitions win. When searching for a given mapping or tiles definition, start with the custom files to see if they have been overwritten there.

7. By convention, all new Struts ActionForm Classes and Action class should go in a custom Java package, eg, com.yourdomainname.commerce.customer. The package name should end in “customer” for the customer interface and “administrator” for the administrator interface (eg., com.yourdomainname.commerce.administrator).

8. All new JSPs templates should go in /WEB-INF/layouts/custom. These are not overwritten by upgrades, where as the ones in /WEB-INF/layouts/default* are.

Quiz for Web Controller (Struts) Layer

1. Where are SoftSlate's Struts configuration files found in the application, for the *customer* interface? And where are the Struts config files for the *admin* interface found?
2. When customizing SoftSlate Commerce, what file should new or altered Struts configurations be placed in? Answer both for the customer interface, and for the admin interface.
3. What file should custom or altered Tiles definitions be placed in? Answer in reference to both the customer interface and the admin interface.
4. If a Struts action mapping includes a "name" attribute and specifies validate="true", what method of what class is invoked to validate the request's input parameters?
5. If the validation fails, where is the request forwarded to?
6. If the validation succeeds, where is the request forwarded to?
7. In the Struts forms and actions, we sometimes add "errors" or "messages" to a request object. What files store the text of these messages and errors? Where are they found on the file system?
8. In the Struts forms and actions, we sometimes add "errors" or "messages" to a request object. What files store the text of these messages and errors? Where are they found on the file system?
9. What file(s) would you use in order to customize one of the existing application messages or errors, or add new ones?

3. The Business Layer

Overview

The Business layer is responsible for processing the application's business logic and representing the application's state in the form of Java beans. For example, in the case of a request to add an item to the user's cart, the Business layer will initialize the user's cart if it hasn't been already, create new objects representing the items in the cart, and then communicate with the Data Access layer to store a representation of the cart in the database.

The Business layer consists of the following classes:

- The classes in the `com.softslate.commerce.businessobjects` package, which is broken down into several units loosely organized by functional area, including "product", "customer", "order", "core", etc.

Creating Business Objects

Business objects are created by Struts Actions, or subclasses of CommandLineSupport for comman-line driven jobs, or any other Java process that needs them. They are ignorant of being used within a Web application so can be called from any Java process (no references to ServletContext for example).

When creating a Business Objects the typical way is to call BusinessObjectFactory.createObject (there is also support for Google Guice injection but that is not in widespread use):

```
ProductProcessor productProcessor = baseForm.getBusinessObjectFactory()
      .createObject(ProductProcessor.class);
…
category = productProcessor.getCategoryFromCode(category,
      shouldProductsBeLoadedFromDB);
```

If the argument passed into createObject is a concrete class, BusinessObjectFactory.createObject simply instantiates it. If the argument is an interface, as it will be in all cases in the core application, it first looks up the implementing class corresponding to the argument from /WEB-INF/classes/appComponents.properties and /WEB-INF/classes/appComponents-custom.properties.

Each Business Object in the core application has a corresponding Java interface, which the rest of the application uses to manipulate it. For example, the processor used to handle various requests related to a user's cart is named `com.softslate.commerce.businessobjects.order.BasicCartProcessor`. However, throughout the rest of the application, instances of `BasicCartProcessor` are always referred to through its corresponding interface, `com.softslate.commerce.businessobjects.order.CartProcessor`. This practice provides a huge benefit in terms of customizing SoftSlate Commerce: through a simple configuration change, you can change the concrete class the application uses to something else. As long as it implements the original interface, the rest of the application can use it just as it did the original class. In your new custom class, however, you now have the ability to override any and every method to implement your own custom functionality. We'll explore the nuts and bolts of doing this in more detail later. The exact same feature is available for all the classes in the Data Access layer as well.

Beans and Processors

There are two kinds of Business Objects, which we call "Beans" and "Processors". Processors (ie. those classes named `*Processor`) contain business processing logic, whereas Beans (ie. those classes named `*Bean`) store application state.

Beans

Beans are subclasses of BaseBusinessObject and follow the Java beans pattern and we use them to store state. They are simply composed of properties, each with a getter and setting. There may be other small, supporting methods but they will have very minimal functionality. Eg. ProductBean holds the state of a given product record in the database.

`com.softslate.commerce.businessobjects.product.ProductBean`

```
public class ProductBean extends BaseBusinessObject implements Product,
            Serializable {
      private static final long serialVersionUID = 6016213050418840957L;
      static Log log = LogFactory.getLog(ProductBean.class);
      private int productID = 0;
      private String code = null;
      private String seoCode = null;
      private String name = null;
      private String keywords = null;
      private String shortDescription = null;
      private String description = null;
      private boolean isActive = true;
      …
}
```

Processors

Processors are subclasses of BaseBusinessProcessor. These objects are where the functionality lives. They are organized loosely by functional area. Eg. BasicProductProcessor has methods relating to working with products and the product catalog. BasicOrderProcessor has methods working with orders, and so on. Other Processors are more focused on a particular functional area, such as BasicCartDiscountProcessor, which has methods used to apply discounts to a user's order.

After instantiating a Processor, createObject initializes the following properties of the object. This allows each Processor to carry with it all tools it needs to perform whatever function it needs to perform.

- **`user`**: representing the current user of the application (eg. getUser().getAdministrator().getUserName() gets you the current logged in administrator's user name.
- **`settings`**: representing the global application settings (eg.

settings.getValue(“databaseObjectsVersion”) gets you the current version of SoftSlate).

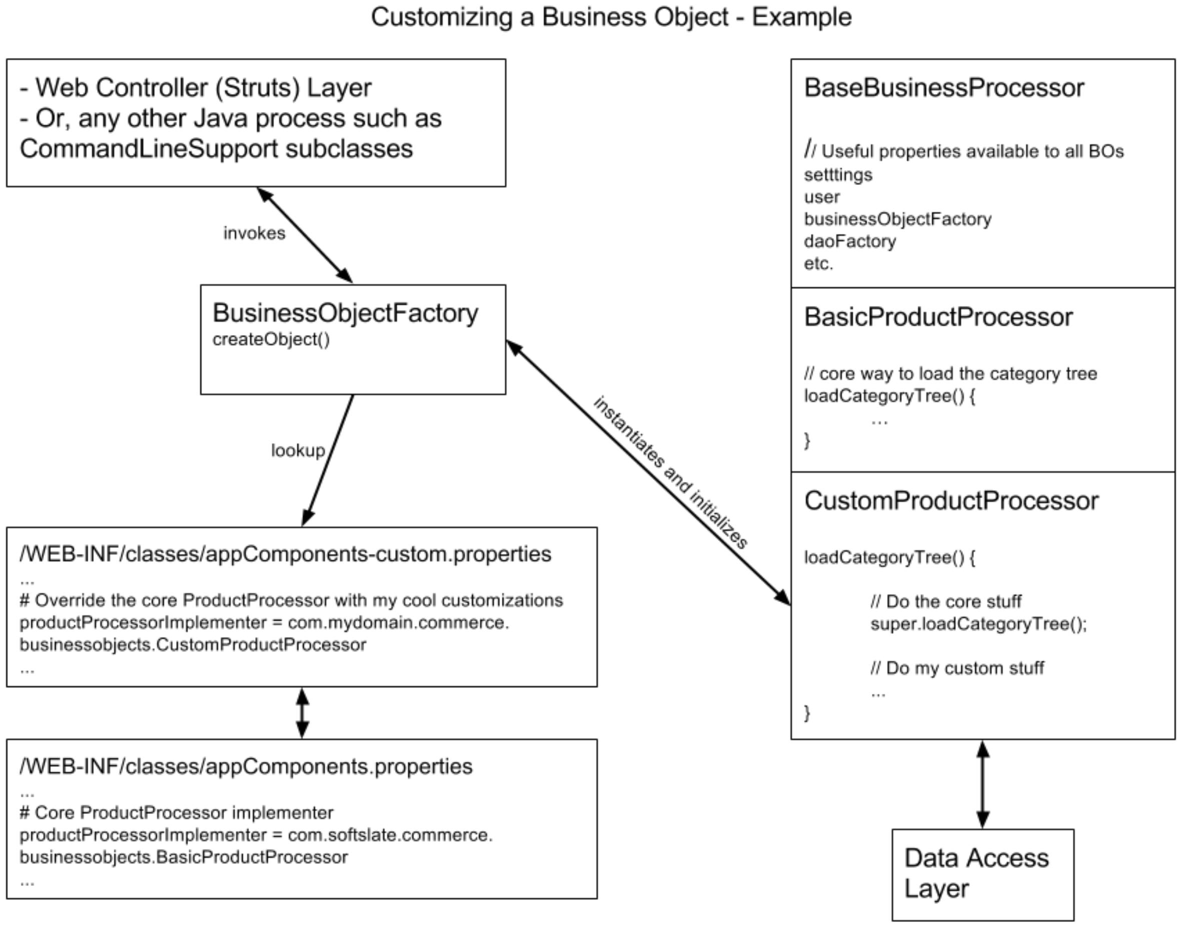
- **daoFactory**: allowing the object to create DAOs for database interaction. The BO can also do some direct database interaction via this property, which is against convention but sometimes useful.
- **businessObjectFactory**: allows the BO to create other BOs.
- **businessObjectUtils**: date formatting and other utility functions.
- **appSettings**: represents /WEB-INF/classes/appSettings.properties
- **appComponents**: /WEB-INF/classes/appComponents.properties and /WEB-INF/classes/appComponents-custom.properties
- **injector**: the Google Guice injector used for injecting other BOs (not in widespread use)
- **eventBus**: allowing the BO to post events that other registered triggers can listen and react to.

Example of a Processor

`com.softslate.commerce.businessobjects.order.BasicCartDiscountProcessor`

```
public class BasicCartDiscountProcessor extends BaseBusinessProcessor
        implements CartDiscountProcessor {
…
        public Map processDiscounts(Collection newOrderItems) throws Exception {
                if (log.isDebugEnabled())
                        log.debug("Processing discounts.");
                String applyDiscounts = (String) getSettings().getValue(
                                "applyDiscounts");
                if (getUser().getOrder() != null
                                && getUser().getOrder().firstDelivery() != null
                                && applyDiscounts != null && applyDiscounts.equals("1")) {
…
                        // Create a DAO to communicate with the database
                        OrderDAO orderDAO = (OrderDAO)
getDaoFactory().createDAO("orderDAOImplementer");
                        orderDAO.setOrder(getUser().getOrder());
                        orderDAO.updateOrder(true);
…
}
```

Customizing Business Objects


Customizing a Business Object - Example
- Web Controller (Struts) Layer
- Or, any other Java process such as CommandLineSupport subclasses
invokes
BusinessObjectFactory
createObject()
lookup
instantiates and initializes
BaseBusinessProcessor
BasicProductProcessor
CustomProductProcessor
Data Access Layer

/WEB-INF/classes/appComponents-custom.properties

```
...
# Override the core ProductProcessor with my cool customizations
productProcessorImplementer = com.mydomain.commerce.
businessobjects.CustomProductProcessor
...
```

/WEB-INF/classes/appComponents.properties

```
...
# Core ProductProcessor implementer
productProcessorImplementer = com.softslate.commerce.
businessobjects.BasicProductProcessor
...
```

BaseBusinessProcessor

```
// Useful properties available to all BOs
setttings
user
businessObjectFactory
daoFactory
etc.
```

BasicProductProcessor

```
// core way to load the category tree
loadCategoryTree() {
        …
}
```

CustomProductProcessor

```
loadCategoryTree() {

        // Do the core stuff
        super.loadCategoryTree();

        // Do my custom stuff
        ...
}
```

Quiz for Business Object Layer

1. What SoftSlate Commerce configuration file stores the names of the concrete classes that implement all of the various interfaces in the application, for both business objects and DAOs?

2. When customizing SoftSlate Commerce, what file should be used to change the concrete class used by the application to implement an interface, where you would put the reference to the custom implementing business object or DAO you created?

3. What SoftSlate Commerce class is responsible for instantiating and initializing business objects?

4. In SoftSlate Commerce, no business object or DAO will ever refer to a Struts class, or to HttpServletRequest, or any other element of the Web layer -- in order to maintain separation of concerns, and to allow non-web processes such as command line jobs to use them. In lieu of HttpSession, what business object in SoftSlate Commerce is used to store information about the current user's session?

5. In the business object layer, how would you test that the current user is a logged in customer? How would you test that he is a logged in administrator?

6. Some business objects are named like "ProductBean" and some are named like "BasicProductProcessor". Generally speaking, what's the difference between a SoftSlate Commerce bean and a processor?

7. Every BusinessProcessor has a property named settings, which an instance of SettingsBean. What does this object store, and what database table does it correspond to?

4. The Data Access Layer

Overview

The Data Access layer is responsible for writing and retrieving data to and from the database, and returning the results to the Business layer. For example, in the case of a request to add an item to the user's cart, the Business layer will invoke classes in the Data Access layer, which will do the work of inserting and updating the various tables in the database that store a representation of the user's cart. Starting with version 2.x, SoftSlate Commerce employs the Hibernate persistance framework to handle all of its database interaction.

The Data Access layer consists of the following classes and files:

- The classes in the `com.softslate.commerce.daos` package, which is broken down into several units loosely organized by functional area, including "product", "customer", "order", "core", etc.
- The `sql*.hbm.xml` files under the `WEB-INF/classes/resources` subdirectories, which contain the HQL (Hibernate Query Language) queries used by the application.
- The `*.hbm.xml` files in the `com.softslate.commerce.businessobjects` package, which contain the Hibernate mappings that map the business object Java beans to the database tables.

The majority of Beans in SoftSlate correspond to a single table in the database. SoftSlate employs Hibernate to map these beans to records in the database, and these mappings are expressed by the various *.hbm.xml configuration files. For example, take the PaymentBean:

`com.softslate.commerce.businessobjects.payment.PaymentBean`

```
public class PaymentBean extends BaseBusinessObject implements Payment,
            Serializable {
      private static final long serialVersionUID = 6172928043433996107L;
      private int paymentID = 0;
      private String status = null;
      private int orderID = 0;
      …
}
```

This Bean has a corresponding Hibernate mapping file, PaymentBean.hbm.xml.

`/WEB-INF/src/com/softslate/commerce/businessobjects/payment/PaymentBean.hbm.xml`

```
<hibernate-mapping>
      <class
            name="com.softslate.commerce.businessobjects.payment.Payment"
            table="Payment"
            lazy="false">
            <id name="paymentID">
                  <generator class="native"/>
            </id>
            <discriminator/>
            <property name="paymentID" insert="false" update="false"/>
            <property name="orderID" insert="false" update="false"/>
            <property name="status" length="100" index="IDX_sscP_status"/>
…
</hibernate-mapping>
```

And the Bean corresponds to the sscPayment database table:

```
mysql> describe sscPayment;
+--------------------------+---------------+------+-----+
| Field                    | Type          | Null | Key |
+--------------------------+---------------+------+-----+
| paymentID                | int(11)       | NO   | PRI |
| status                   | varchar(100)  | YES  |     |
| orderID                  | int(11)       | NO   | MUL |
…
```

Data Objects in SoftSlate essentially use Hibernate to convert Beans into database records and vice-versa. This is the essence of Object-Relation mapping (ORM).

Creating Data Objects

Data Objects by convention are created by BusinessObjects. As with Business objects, it is possible to inject them using Google Guice but this is not in widespread use. Typcially DAOs are created using the DaoFactory property of a Business Object (`getDaoFactory()`).

`From BasicCartDiscountProcessor`

```
      OrderDAO orderDAO = getDaoFactory().createDAO(OrderDAO.class);
      orderDAO.setOrder(getUser().getOrder());
      orderDAO.updateOrder(true);
```

DaoFactory.createDAO works much the same way as BusinessObjectFactory.createObject. If the argument passed into createDAO is a concrete class, it simply instantiates it. If the argument is an interface, it first looks up the implementing class corresponding to the argument from /WEB-INF/classes/appComponents.properties and /WEB-INF/classes/appComponents-custom.properties. In the above example, createDAO would see that OrderDAO is an interface. It would look inside appComponents-custom.properties first to see if an implementer is defined there. If none has been define in the custom configuration it will look at appComponents.properties, where it will find this:

`From /WEB-INF/classes/appComponents.properties`

```
orderDAOImplementer = com.softslate.commerce.daos.order.OrderDAOHibernate
```

This tells createDAO to instantiate an instance of OrderDAOHibernate.

Next, createDAO will initialize the new DAO with various properties so it can do everything it needs to do to communicate with the database. All DAOs subclass BaseDAO, where these properties are defined:

- **Settings**: representing the global application settings (eg.

settings.getValue("databaseObjectsVersion") gets you the current version of SoftSlate).

- **DaoFactory**: allowing the object to create other DAOs. In addition, DAO holds the Hibernate SessionFactory which is used to initiate a Hibernate session, which is what is used to run queries.
- **appSettings**: represents /WEB-INF/classes/appSettings.properties
- **appComponents**: /WEB-INF/classes/appComponents.properties and /WEB-INF/classes/appComponents-custom.properties
- **Injector**: the Google Guice injector used for injecting other DAOs (not in widespread use)

Example of a DAO

`com.softslate.commerce.daos.order.OrderDAOHibernate`

```
public class OrderDAOHibernate extends BaseDAO implements OrderDAO {
...
      public Order loadOrderFromID(int orderID) throws Exception {
            Session session = getDaoFactory().startSession();
            try {
                  Order record = (Order) session.get(Order.class, new Integer(
                              orderID));
                  return record;
            } catch (HibernateException he) {
                  if (log.isErrorEnabled())
                        log.error("Hibernate error: " + he.toString());
                  throw new DataAccessException(he);
            }
      }
…
      public void updateOrder(boolean commit) throws Exception {
            Session session = getDaoFactory().startSession();
            try {
                  getOrder().setLastModified(formatDateTime(new java.util.Date()));
                  session.update(getOrder());
                  if (commit) {
                        getDaoFactory().commitTransaction();
                  }
            } catch (HibernateException he) {
                  if (log.isErrorEnabled())
                        log.error("Hibernate error: " + he.toString());
                  throw new DataAccessException(he);
            }
            return;
      }
…
}
```

Querying for Collections of Objects

Many DAOs are like the above example and simply deal with selecting, updating, inserting and deleting a single record in the database. These types of DAOs are named *DAOHibernate.java, etc. OrderDAOHibernate, ProductDAOHibernate, CustomerDAOHibernate, etc.

The DAOs with the term "Gateway" in their name, eg OrderGatewayDAOHibernate and ProductGatewayDAOHibernate.java are used to process multiple records at once. For example, when querying all of the products in the system:

`com.softslate.commerce.daos.product.ProductGatewayDAOHibernate`

```
public class ProductGatewayDAOHibernate extends BaseAdminGatewayDAOHibernate
        implements ProductGatewayDAO {
    …
    public Collection loadAllProductsSortedByName() throws Exception {
        Collection results = null;
        Session session = getDaoFactory().startSession();
        try {
            Query q = getNamedQuery(session,"product.selectAllSortedByName");
            results = new LinkedHashSet(q.list());
        } catch (HibernateException he) {
            if (log.isErrorEnabled())
                log.error("Hibernate error: " + he.toString());
        }
        return results;
    }
    …
}
```

Named Queries (queries*.hbn.xml files)

The above example uses a "named query," which is simply a query that is defined in an XML file so it can be maintained separately from Java code. In SoftSlate, the named queries are stored in files named queries.hbm.xml (for queries distributed with the core application) and queries-custom.hbm.xml (for custom queries). Hibernate queries can be written using native SQL but all the queries distributed with the core application are written using HQL, the Hibernate Query Language, which is database independent. Much more info in the Hibernate documentation:
http://docs.jboss.org/hibernate/orm/3.2/reference/en/html/objectstate.html#objectstate-querying

`/WEB-INF/src/resources/product/queries.hbm.xml`

```
    <query name="productList.selectAllActive"><![CDATA[
        SELECT p FROM Product p
        LEFT JOIN FETCH p.primaryCategory c
        WHERE
            p.isActive = '1'
            AND (c.isActive = '1'
                OR c.isActive IS NULL)
        ORDER BY c.categoryOrder, p.productOrder
    ]]></query>
```

This query returns a Collection of Objects when the query.list() method is run for it.

Customizing the Data Access Layer

Customizing a Data Access Object is done in exactly the same way as customizing a Business Object. Please refer the to above sections for an example. The new or overriding subclass should be registered in /WEB-INF/classes/appComponents-custom.properties, as with custom Business Objects.

If you need to customize one of the core SoftSlate named queries, append “-custom” to the name of the query and place the custom query in the correponding queries-custom.properties file. Eg, to customize the above query, name your query like this:

`/WEB-INF/src/resources/product/queries-custom.hbm.xml`

```
<query name="productList.selectAllActive-custom"><![CDATA[
…
</query>
```

If named in this manner, SoftSlate will use the custom query instead of the core query.

Quiz for Data Access Layer

1. The naming convention for DAOs in SoftSlate Commerce includes classes named like "ProductDAOHibernate" and other classes named like "ProductGatewayDAOHibernate". What is the general difference between these two DAOs (what does the "Gateway" term signify)?

2. What SoftSlate Commerce object stores the Hibernate SessionFactory, and is used by each individual DAO when it needs to make use of a Hibernate Session?

3. SoftSlate Commerce follows a model of a one Hibernate Session per HTTP request. At what point is the Hibernate Session first created, and what object(s) are responsible for creating it?

4. What object is responsible for closing the Hibernate Session corresponding to each request? Where does that happen in the application?

5. What files store the HQL statements used by the DAO objects? Where are they found in the application's directory structure? How would one override one of the HQL statements with a custom HQL query?

5. The Presentation Layer

Overview

After the Web Controller layer routes the request to the Business layer and retrieves the results of the processing, it then forwards the request to the Presentation layer, which is responsible for generating the HTML sent to the user's browser. The Presentation layer is composed of the following files:

- The `tiles-defs*.xml` files under the `WEB-INF/conf` subdirectories (for the customer interface).
- The `tiles-defs*.xml` files under the `WEB-INF/conf/administrator` subdirectories (for the administrator interface).
- The JSP templates in the `WEB-INF/layouts` directory (for the customer interface).
- The JSP templates in the `WEB-INF/templates/administrator` directory (for the administrator interface).
- The `application.properties*.xml` files under the `WEB-INF/classes/resources` subdirectories, which contain various text messages used in the Presentation layer.
- The `css/style.css` and `css/style-custom.css` files, which contain the CSS stylesheet definitions the application uses.
- The `images` directory, which contains some images referred to in the application's HTML output.

How Struts Routes Requests to JSP Templates

In the case of a request to view a product detail page, the Web Controller layer will place the requested product information retrieved from the database in the request scope, (in the form of an instance of `com.softslate.commerce.businessobjects.product.Product`). It will then forward the request to the Presentation layer, to output the HTML containing the information to the user's browser.

`com.softslate.commerce.customer.product.ProductAction`

```
public class ProductAction extends BaseAction {
     public ActionForward execute(ActionMapping mapping, ActionForm form,
               HttpServletRequest request, HttpServletResponse response)
               throws Exception {
          ...
          ProductProcessor productProcessor =  baseForm
               .getBusinessObjectFactory().createObject(
               ProductProcessor.class);
          Product product =  baseForm.getBusinessObjectFactory()
               .createObject(Product.class);
          ...
          product = productProcessor.getProductFromCode(product);
          ...
          request.setAttribute("product", product);
```

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

The ActionForward returned by the Struts action is looked up in the Struts configuration to determine how to forward the request to the Presentation Layer:

`/WEB-INF/conf/product/struts-config-product.xml`

```
        <action
            path="/Product"
            type="com.softslate.commerce.customer.product.ProductAction"
            name="productForm"
            scope="request"
            validate="true"
            input="core.error">
                <forward name="success" path="product.product"/>
                <forward name="successFullLayout"
path="product.productFullLayout"/>
                <forward name="failure" path="core.error"/>
            </action>
```

SoftSlate Commerce employs the Tiles framework, which is distributed as part of Struts, to organize the JSP templates responsible for presentation. When the Web Controller layer executes a forward to the Presentation layer, it specifies one of the Tiles definitions defined in the `tiles-defs*.xml` files under the `WEB-INF/conf` subdirectories. The role of each Tiles definition is to describe which JSP templates should be used to output the response.

> **Note**
>
> There is much more information about Tiles available in books and online. Refer to the Struts Web site for more details.

In the case of a request to view a product detail page, the Web Controller layer forwards to the `product.product` Tiles definition, which is defined in the `WEB-INF/conf/product/tiles-defs-product.xml` file.

```
    <definition name="product.product" extends="core.baseLeftLayout">
      <put name="pageTitleKey" value="page.product"/>
        <put name="subMenu" value="product.breadcrumbs" />
        <put name="body" value="product.productLayout" />
    </definition>
      ...
    <definition name="product.productLayout" path="/WEB-
INF/layouts/default/product/product.jsp">
        <put name="inventoryDiscountSettings" value="/WEB-
INF/layouts/default/product/inventoryDiscountSettings.jsp" />
        <put name="inventory" value="/WEB-
```

```
INF/layouts/default/product/inventory.jsp" />
        <put name="discounts" value="/WEB-
INF/layouts/default/product/discounts.jsp" />
        <put name="promotions" value="/WEB-
INF/layouts/default/product/promotions.jsp" />
        <put name="additionalImages" value="/WEB-
INF/layouts/default/product/additionalImages.jsp" />
        <put name="productReviews" value="/WEB-
INF/layouts/default/product/productReviews.jsp" />
        <put name="attributesAndOptions" value="product.attributesAndOptions" />
        <put name="attributesAndOptionsMatrix" value="/WEB-
INF/layouts/default/product/attributesAndOptionsMatrix.jsp" />
        <put name="relatedProductList" value="product.productListProductLayout" />
        <put name="relatedProductListRows"
value="product.productListProductLayoutRows" />
        <put name="wishListProductAddForm" value="/WEB-
INF/layouts/default/product/wishListProductAddForm.jsp" />
    </definition>
```

That file tells Tiles to use the `WEB-INF/layouts/default/product/product.jsp` as the main JSP template to display the body of the product page.

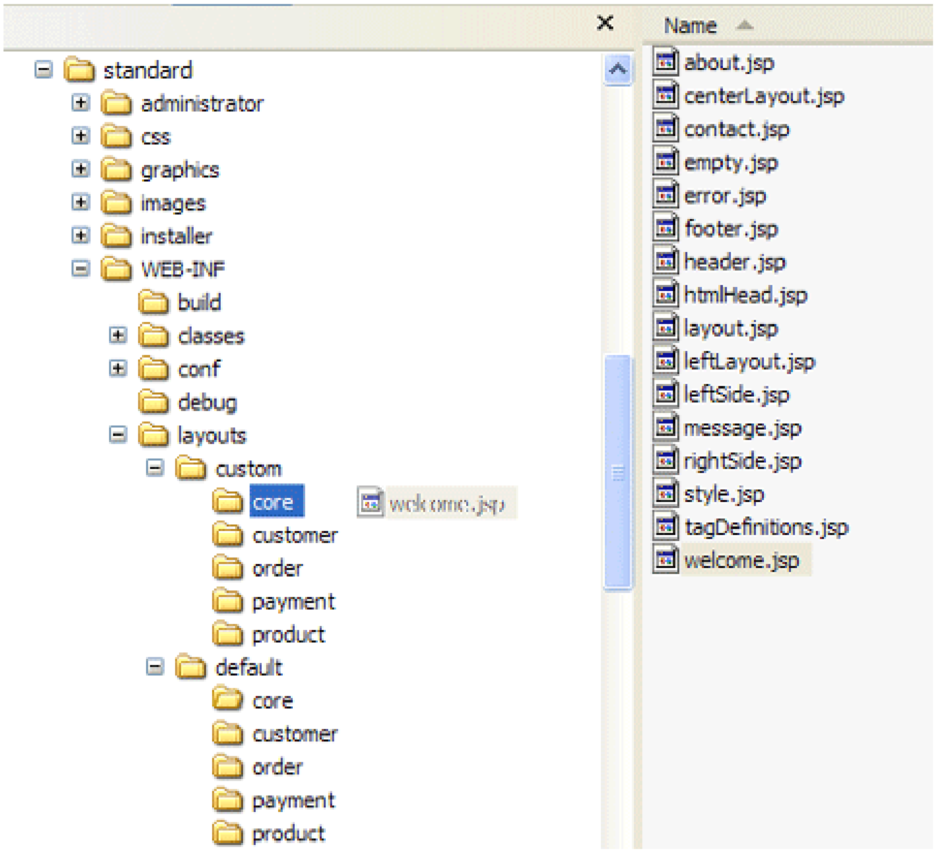
> **Note**
>
> SoftSlate Commerce extends the Tiles framework to first look inside the `custom` directory each time a given template is called for, before using the template found in the `default` directory. This is how a definition that calls for `/WEB-INF/layouts/default/core/welcome.jsp` will include `/WEB-INF/layouts/custom/core/welcome.jsp` instead, if it exists.

Customizing the Presentation Layer

The JSP templates that come with the application are all located in the `/WEB-INF/layouts/default-html5` directory (for versions between 2.3 and 3.3.2 it's the `/WEB-INF/layouts/default-xhtml` directory and for versions prior to 2.3, it's the `/WEB-INF/layouts/default` directory). Inside there, they are organized into subdirectories representing the major functional areas of your store: `core` , `product`, `order`, `customer`, and `payment` .

Custom templates that you create will go in the `/WEB-INF/layouts/custom` directory, which contains the same five subdirectories. If a JSP template exists in the `custom` directory, the store will automatically detect and use it in place of the corresponding template under the `default` directory. This system allows you to create custom templates for just those areas of the store you need to modify, while other areas can be left untouched so that they use the default templates.

It's important not to modify the templates in the `/WEB-INF/layouts/default-html5` directory. These files may be overridden when you install a future upgrade of SoftSlate Commerce. Instead, please place a copy of the template you wish to change into the corresponding subdirectory under `/WEB-INF/layouts/custom`. The store will automatically detect the new file and begin using it immediately. You can then feel free to make whatever changes you wish to the copied file.



Copying a template from the `default-html5` directory to the `custom` directory

More Information on Customizing the Presentation Layer

For more information about customizing the presentation layer, visit the Guide for Designers: http://www.softslate.com/documentation/html/guideForDesigners.html

In addition this video takes you step-by-step making a customization to the CSS and JSPs: http://screencast.com/t/AFVLzOD9JBv

Quiz for Making CSS Customizations

NOTE: Be sure to watch the video first: http://screencast.com/t/AFVLzOD9JBv

1. SoftSlate Commerce has a notion of a "CSS Theme". Where in the SoftSlate Commerce project are the files/directories corresponding to each CSS Theme?
2. What screen of the Administrator is used to select the CSS theme in effect for the store?
3. Regardless of the CSS Theme in effect, what CSS file in project is included before the CSS stylesheet corresponding to the CSS Theme is included?
4. Regardless of the CSS Theme in effect for the store, what CSS file is included *after* the stylesheet for theme's stylesheet, allowing you to override all the core styles and theme's styles?
5. What screen of the Administrator controls dynamic CSS styles for the store? And what JSP template is responsible for enacting the styles?
6. Describe how you would go about enforcing a fixed width of 1000 pixels for all screens in the store. What file(s) would you touch?

Quiz for Making JSP Customizations

NOTE: Be sure to watch the video first: http://screencast.com/t/AFVLzOD9JBv

1. Where in the SoftSlate Commerce project are all the JSP templates for the customer interface located? Where are the templates for the Administrator interface located?

2. SoftSlate Commerce has a "default" directory and a "default-xhtml" directory. What's the difference between the two? Which directory has the default customer-side templates for a fresh installation of SoftSlate Commerce?

3. What is the basic process for applying a customization to one of the JSP templates? Specifically, what would you do to apply a customization to the customer-side header.jsp file?

4. What JSP templates control the appearance of the following? (a) The customer-side Cart page. (b) The customer-side Thank You page. (c) The customer-side Order Details page under My Account -> Order History. (d) The HTML section of the customer-side order confirmation email. (e) The admin-side Order Overview screen.

5. Some SoftSlate Commerce screens on the customer side use "leftLayout.jsp" and others use "layout.jsp". Name three customer-side screens that use leftLayout.jsp. Name three customer-side screens that use layout.jsp. What is the essential difference between these two JSPs?

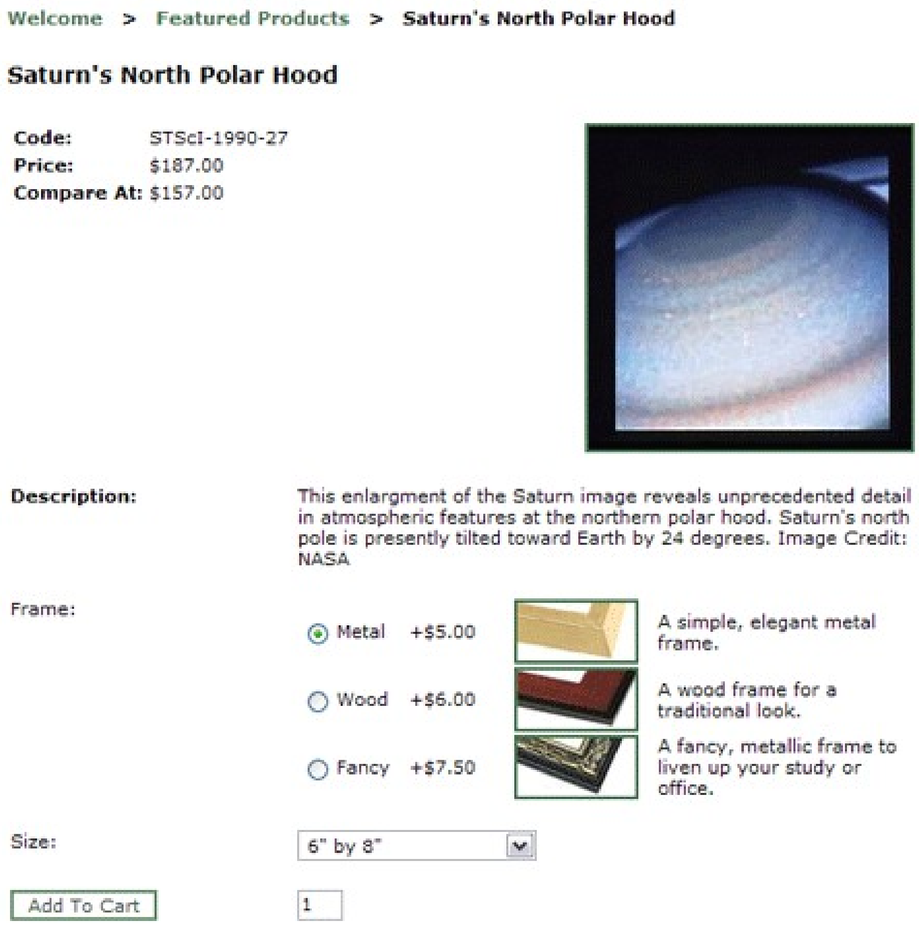
6. What JSP template would you customize in order to add an include for Javascript file inside the <head> tag of all Administrator screens?

7. What are the mechanics behind SoftSlate using the JSPs in the "custom" directory over the ones in the default directories? What tag or tags do it?

6. Full Example - Adding an Item to the Cart

There's nothing better than a detailed example to provide some insight into the concepts presented above. In the following example, we'll go step-by-step through a typical request sent to the SoftSlate Commerce application, touching on each of the configuration files, Java classes, and JSP templates used to process it. The example we'll use is a request from a user to add an item to the cart.

1. *The user submits the "Add to Cart" form from a product detail page.* As you can see, the form contains an input box for the quantity, as well as any number of form elements describing customer-selected "attributes" for the product (in this case, "Frame" and "Size" are the two attributes assigned to this product).



Welcome > Featured Products > Saturn's North Polar Hood

Saturn's North Polar Hood

Code: STScI-1990-27
Price: $187.00
Compare At: $157.00

Description: This enlargment of the Saturn image reveals unprecedented detail in atmospheric features at the northern polar hood. Saturn's north pole is presently tilted toward Earth by 24 degrees. Image Credit: NASA

Frame:

● Metal	+$5.00	A simple, elegant metal frame.
○ Wood	+$6.00	A wood frame for a traditional look.
○ Fancy	+$7.50	A fancy, metallic frame to liven up your study or office.

Size: 6" by 8"

1

The Add to Cart form.

2. *The Web Controller or Struts layer identifies the action mapping corresponding to the URL of the request.* In this case, the URL is `/CartAdd.do` . The Struts `ActionServlet` looks up the corresponding action mapping and finds it in the `WEB-INF/conf/order/struts-config-order.xml` configuration file:

```
<action
        path="/CartAdd"
        type="com.softslate.commerce.customer.order.CartAddAction"
        name="cartAddForm"
        validate="true"
        scope="request"
        input="/Product.do">
        <forward name="failure" path="/Product.do"/>
        <forward name="success" path="cart.full"/>
</action>
```

3. *The Web Controller or Struts layer identifies and invokes the Struts action form corresponding to the request.* The above action mapping refers to the "cartAddForm" form bean, which is defined at the top of the same configuration file, `WEB-INF/conf/order/struts-config-order.xml`:

   ```
   <form-bean
           name="cartAddForm"
           type="com.softslate.commerce.customer.order.CartAddForm">
   </form-bean>
   ```

   Because the action mapping's `validate` attribute is set to true, Struts knows that the request must be validated. In Struts, this means that the `validate` method of the action form `com.softslate.commerce.customer.order.CartAddForm` must be invoked.

4. *The `validate` method of `com.softslate.commerce.customer.order.CartAddForm` validates the request's parameters.* In this case, `CartAddForm` makes sure a postitive integer is present for the quantity to be added, and that if any of the product's attributes are required, they are present in the request.

   If the result of the validation is one or more errors, the request gets forwarded to the value of the action mapping's `input` attribute. If there are no errors, Struts proceeds to invoke the `execute` method of the action class `com.softslate.commerce.customer.order.CartAddAction`. We'll assume a successful result without errors.

5. *The `execute` method of `com.softslate.commerce.customer.order.CartAddAction` is invoked to process the request.* In this case, `CartAddAction` starts by creating an instance of `com.softslate.commerce.businessobjects.order.CartProcessor`, invoking its `processAddItems` method:

   ```
   // Process the items
   CartProcessor cartProcessor = (CartProcessor) baseForm
                   .getBusinessObjectFactory().createObject(
                                   "cartProcessorImplementer");
   Map results = cartProcessor.processAddItems(PropertyUtils
                   .describe(baseForm));
   ```

At this point, control of the request is passed from the Web Controller or Struts layer, to the Business layer.

6. *Now in the Business layer, the* `processAddItems` *method of* `com.softslate.commerce.businessobjects.order.BasicCartProcessor` *is invoked.* This method handles the business logic involved with the request to add an item to the cart. Namely, it instantiates the user's cart if it doesn't already exist. A user's cart is represented as an instance of `com.softslate.commerce.businessobjects.order.Order` . It then parses the incoming parameters and creates an instance of `com.softslate.commerce.businessobjects.order.OrderItem` to represent the order item.

   After performing some validations against the database, processing inventory, and processing discounts, it finally hands things off to the Data Access layer to record the results of the request in the database:

```
OrderGatewayDAO orderGatewayDAO = (OrderGatewayDAO) getDaoFactory()
                .createDAO("orderGatewayDAOImplementer");
orderGatewayDAO
                .processOrderItems(getUser(), newOrderItems, results);
```

7. *Now in the Data Access layer, the* `processOrderItems` *method of* `com.softslate.commerce.doas.order.OrderGatewayDAOHibernate` *is invoked.* This method synchronizes the state of the user's cart with the database. Namely, it will insert or update records into the `sscOrder` and related tables, storing the new order item in the `sscOrderItem` table.

   The DAO will use Hibernate mappings to synchronize the application's objects with the database. In this case, the Hibernate mapping file `WEB-INF/classes/com/softslate/commerce/businessobjects/order/Order.hbm.xml` and other mappings files next to it are used to synchronize the application's objects with the database tables.

   Some DAOs will use HQL (Hibernate Query Language) mappings to communicate with the database. In cases where data is being retrieved from the database, the DAO will employ the HQL queries in the `queries.hbm.xml` files in the `WEB-INF/classes/resources` subdirectories to query the database through Hibernate.

8. *Assuming a successful result, control passes back to* `com.softslate.commerce.customer.order.CartAddAction` *.* Assuming everything goes well, control will pass back to the Web Controller or Struts layer. In this case, `CartAddAction` will perform some additional processing of the request. In particular, depending on the application's inventory settings, the results of the processing by the Business layer might indicate that one or more messages be displayed to the user concerning inventory levels, or that one or more "low stock emails" be sent to the store's administrator.

   When it is finished processing these possibilities, the `execute` method of `CartAddAction` forwards the request to a Struts action forward:

```
return mapping.findForward(forward);
```

9. *The Web Controller or Struts layer looks up the action forward returned by the action class.* The Struts action forwards are defined back in the Struts action mapping for the request. In this case the following two action forwards can be returned:

   ```
   <forward name="failure" path="/Product.do"/>
   <forward name="success" path="cart.full"/>
   ```

   If it's a failure, control of the request is passed to the "/Product.do" URL. In other words, the user is sent back to the product page where any errors that occured are displayed.

   If on the other hand everything went well and the processing succeeded, control is passed to "cart.full", which is not a URL but rather a *Tiles definition*. Forwards that use this sort of dot notation identify Tiles definitions.

   It's at this point that control passes to the Presentation layer.

10. *The Presentation layer looks up the Tiles definition corresponding to the action forward.* In this case, it finds the "cart.full" Tiles definition in the `WEB-INF/conf/order/tiles-defs-order.xml` configuration file:

    ```
    <definition name="cart.full" extends="core.baseLeftLayout">
            <put name="pageTitleKey" value="page.cart"/>
            <put name="subMenu" value="product.breadcrumbs" />
            <put name="body" value="cart.fullLayout" />
    </definition>
    ```

    Tiles definitions can both extend other definitions, and include other definitions under them. As you can see, "cart.full" extends the "core.baseLeftLayout" definition, which is defined in the `WEB-INF/conf/core/tiles-defs.xml` file:

    ```
    <definition name="core.baseLeftLayout"
            path="/WEB-INF/layouts/default/core/leftLayout.jsp"
            controllerUrl="/LayoutAction.do">
            <put name="beforeHTML"
                    value="/WEB-INF/layouts/default/core/empty.jsp" />
            <put name="htmlHead"
                    value="/WEB-INF/layouts/default/core/htmlHead.jsp" />
            <put name="tracking"
                    value="/WEB-INF/layouts/default/core/tracking.jsp" />
            <put name="header"
                    value="/WEB-INF/layouts/default/core/header.jsp" />
            <put name="error"
                    value="/WEB-INF/layouts/default/core/error.jsp" />
            <put name="message"
                    value="/WEB-INF/layouts/default/core/message.jsp" />
    ```

```
        <put name="leftSide"
                value="core.leftSide" />
        <put name="rightSide"
                value="core.rightSide" />
        <put name="subMenu"
                value="/WEB-INF/layouts/default/core/empty.jsp" />
        <put name="body"
                value="/WEB-INF/layouts/default/core/empty.jsp" />
        <put name="footer"
                value="/WEB-INF/layouts/default/core/footer.jsp" />
        <put name="afterHTML"
                value="/WEB-INF/layouts/default/core/empty.jsp" />
</definition>
```

Here you can start to understand how Tiles organizes the JSP templates used to display the results of every request. The "cart.full" definition extends "core.baseLeftLayout", which identifies many of the JSP templates used to display the application's HTML. Note that "core.baseLeftLayout" indicates that a JSP file named `empty.jsp` should be use for the "body" of the page. Fortunately, "cart.full" overrides this attribute - otherwise, only an empty space would be displayed in the page's body.

"cart.full" identifies another Tiles definition should be used for the body of the page: "cart.fullLayout". This definition appears just below "cart.full" in the same `WEB-INF/conf/order/tiles-defs-order.xml` file:

```
<definition name="cart.fullLayout"
        path="/WEB-INF/layouts/default/order/cart.jsp">
        <put name="couponFormGuts"
                value="/WEB-INF/layouts/default/order/couponFormGuts.jsp" />
</definition>
```

With this definition we've now seen nearly all of the JSP templates used to generate the response. In particular, we can guess that `cart.jsp` is going to provide the guts of the screen for us.

11. *The Presentation layer forwards the request to JSP template correponding to the* `path` *attribute of the Tiles definition.* In this case, because "cart.full" extends "core.baseLeftLayout", the request is forwarded to `/WEB-INF/layouts/default/core/leftLayout.jsp`.
12. *The JSP templates are processed, including processing of the Tiles tags.*

    A peek at `leftLayout.jsp` will give you a good idea of how the various attributes of each Tiles definition are used. For example, this line is the point at which the value of the "body" attribute (in our case, `cart.jsp`) is included:

```
<tiles:insert attribute="body"/>
```

All of `leftLayout.jsp` is processed in this way, including each of the JSP templates referred to by their Tiles attributes. The result is an HTML page displaying the contents of the user's cart, including the new item that has just been added to it.

> **Note**
>
> SoftSlate Commerce extends the Tiles framework to first look inside the `custom` directory each time a given template is called for, before using the template found in the `default` directory. This is how a definition that calls for `/WEB-INF/layouts/default/core/welcome.jsp` will include `/WEB-INF/layouts/custom/core/welcome.jsp` instead, if it exists.

7. Overview of How to Make Customizations

Following is a list of the major areas of SoftSlate Commerce and how to customize each area. Following these rules ensures your changes to the application's core files are kept to a minimum, making future upgrades easier.

1. **Custom Java Classes.**

   Override existing Java classes by subclassing the appropriate class from the `com.softslate.commerce` package with your own new class. Then, change the implementer used by the system in the `/WEB-INF/classes/appComponents-custom.properties` file to tell the application to use your new class instead of the default one. (Requires application reload.) Do not modify any of the existing classes as they may change with an upgrade.

2. **Custom HQL and SQL Queries.**

   Override existing Hibernate queries (HQL) or create new ones in the appropriate `queries-custom.hbm.xml` file found under the subdirectories of the `/WEB-INF/classes/resources` directory. (Requires application reload.) Queries defined in the `queries-custom.hbm.xml` files will override the queries defined in the regular `queries.hbm.xml` files. Do not modify any of the `queries.hbm.xml` files as they may change with an upgrade.

3. **Custom Struts Configurations.**

   Override existing Struts action mappings and form bean definitions, or create new ones, in `/WEB-INF/conf/core/struts-config-custom.xml` . (Requires application reload.) Since this is the last file read into memory, it will override the mappings and form beans defined in the other Struts XML files. Do not modify any of the other Struts XML files as they may change with an upgrade.

4. **Custom Tiles Definitions.**

   Override existing Tiles definitions or create new ones in `/WEB-INF/conf/core/tiles-defs-custom.xml` . (Requires application reload.) Since this is the last file read into memory, it will override the definitions defined in the other Tiles XML files. Do not modify any of the other Tiles definition XML files as they may change with an upgrade.

5. **Custom JSP Templates.**

   Override existing JSP templates by copying them from the `/WEB-INF/layouts/default-html5` directory over to the `/WEB-INF/layouts/custom` directory and modifying them there. Create any new JSP templates in the `custom` directory as well. Do not modify the templates in the `default-html5` directory as they may change with an upgrade. (In a similar way, to customize templates used in the Administrator, copy templates from `/WEB-INF/templates/administrator/default` to `/WEB-INF/templates/administrator/custom` .)

6. **Custom Application Messages.**

   Override existing application messages or create new ones in the appropriate `application-custom.properties` file found under the subdirectories of the `/WEB-`

`INF/classes/resources` directory. (Requires application reload.) Do not modify any of the other `application*.properties` files as they may change with an upgrade.

7. **Custom CSS Styles.**

   Override existing CSS styles or create new ones in `/css//custom/custom.css`. Do not modify any of the other files in `/css` as they may change with an upgrade.

8. Extending SoftSlate Commerce's Business Objects

One of the most common needs in terms of customizing SoftSlate Commerce is to extend the application's business objects to include additional data beyond what's supported by default. For example, in the case of a bookstore, you may need to store additional fields like subtitle, author, and publisher with your products. In other cases, for customers, you may need to track how they first found out about your store or other idiosyncratic fields.

Fortunately, SoftSlate Commerce provides three ways to extend its built-in objects. As you will see, each of the methods has its advantages and disadvantages. We hope that the first two methods are fairly self-explanatory: they require minimal or no custom programming to use. For the last method, we'll go through a detailed example.

Using Built-In 'Extra' Fields

A number of the database tables feature 'Extra' fields named `extra1`, `extra2`, `extra3`, etc. These fields are intended to store any custom, generic data not supported by the built-in fields. To use the various 'Extra' fields, simply populate them in the database, or by using the Administrator as you add and edit your objects.

Supported Objects.

- `sscCategory`
- `sscCustomer`
- `sscCustomerAddress`
- `sscDiscount`
- `sscOrder`
- `sscOrderDelivery`
- `sscOrderDiscount`
- `sscOrderItem`
- `sscProduct`
- `sscSKU`

Advantages.

- The 'Extra' fields are placeholders meant to be used for any custom, generic data, so no database schema changes are necessary.
- The 'Extra' fields get copied from product tables into the correponding fields of the order tables (e.g. `sscProduct.extra1` -> `sscOrderItem.extra1`) with each order, so they can be used during order processing in addition to being displayed in the product catalog.
- The 'Extra' fields can be manipulated in the Administrator alongside the other fields.

Disadvantages.

- It may not be clear within programming code or in the Administrator what each of the different 'Extra' fields represents.
- The data types of the 'Extra' fields are each VARCHAR(255), so there is no flexibility in terms of the fields' data types.
- There are a limited number of 'Extra' fields in each table that supports them, so you may run out.

Storing Custom Data in Settings Tables

The two objects that most commonly need to be extended are categories and products. For these objects, you have the option to define custom settings, which are stored for each product and category respectively in the `sscProductSetting` and `sscCategorySetting` tables.

Supported Objects.

- `sscCategorySetting`
- `sscProductSetting`

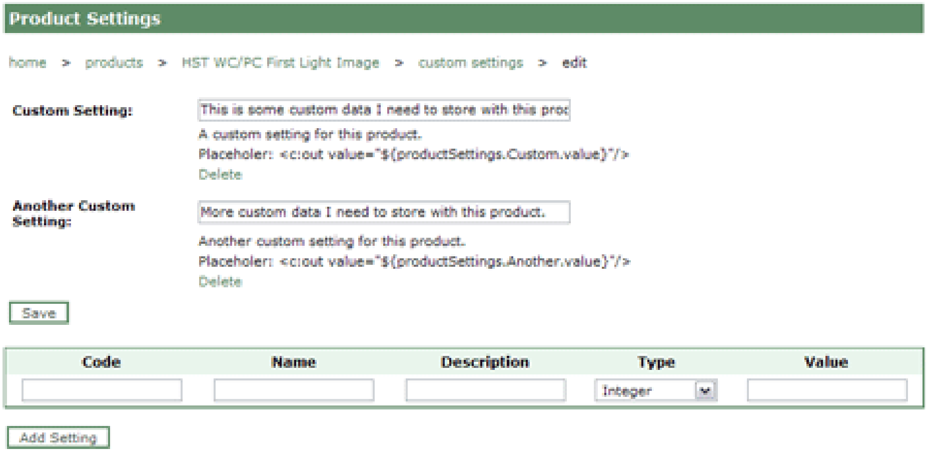
Advantages.

- Good for storing information that only needs to be used during the display of the product catalog. Product settings are loaded lazily on the product page, so no database programming is required to use them.
- The Administrator has an interface to create and modify any number of product settings for each product. (Navigate to `Product Catalog -> Products -> Details -> More -> Custom Settings`)
- A similar interface, at `Product Catalog -> Categories -> Details -> Custom Settings`, is available for categories.

Disadvantages.

- Product settings are not copied into the order item objects as an order is placed, so they cannot easily be used during order processing.
- Even if multiple products or categories have the same set of settings, you must create them individually under each individual object.

To populate `sscProductSetting` and `sscCategorySetting` with custom data related to products and categories, the easiest way may be to use the Administrator's interface. For products, this interface is found by navigating to `Product Catalog -> Products -> Details -> More -> Custom Settings`.


Product Settings
home > products > HST WC/PC First Light Image > custom settings > edit
Save
Code
Name
Description
Type
Integer
Value
Add Setting

Custom Setting: This is some custom data I need to store with this proc

A custom setting for this product.

Placeholer: <c:out value="${productSettings.Custom.value}"/>

Delete

Another Custom Setting: More custom data I need to store with this product.

Another custom setting for this product.

Placeholer: <c:out value="${productSettings.Another.value}"/>

Delete

Product Settings Form.

On this screen, you find the following text input fields:

- *Code:* A unique string identifying this setting with the product. On the product page, use the code to display the value of the setting. For example, if you create a setting with a code of

`Custom` you can display it by placing this tag in a custom `product.jsp` template: `<c:out value="${productSettings.Custom.value}"/>`

- *Name:* Used only in the Administrator, to explain to admin users what the setting is used for.
- *Description:* Also used only in the Administrator, to explain further to admin users what the setting represents.
- *Type:* Determines the data type of the setting. (Determines which field in `sscProductSetting` is used to store the value.)
- *Value:* The value of the setting for this product.

Adding Fields to the Database Tables

The method is both the most flexible and powerful, and also the one that takes the most work.

Supported Objects.

- All Objects

Advantages.

- Data can be stored as it ideally should - in the database table, with the correct data type. Queries can be designed to leverage the additional fields.
- If you add fields to one of the product tables, adding the same fields to the corresponding order table will allow the data to be copied over automatically with each order, and it can therefore be used during order processing logic. (E.g. `sscProduct.newField` -> `sscOrderItem.newField`.)

Disadvantages.

- Requires creating a number of customizations to support the additional fields, including a custom Java bean, a custom Hibernate mapping, and custom Administrator screens. (Much of the code is designed to support additional fields generically, however.)
- The table can get large if a large number of additional fields are required, possibly affecting performance or maintainability.

Let's go through an example of adding a field to a database table. In this example, we'll imagine we are running a bookstore, and that we've decided to add a field to `sscProduct` to store each book's author.

Extending the Product Object by Adding a Field to `sscProduct`

1. Create a new class that extends the built in `com.softslate.commerce.businessobjects.product.ProductBean` class. To keep things clean, we strongly recommend you place all your custom classes in a separate Java package. We'll call our class `CustomProduct` and we'll put it in a new package named `com.softslate.commerce.demo` . Here's the source code for our new class:

```
package com.softslate.commerce.demo;

import com.softslate.commerce.businessobjects.product.ProductBean;

public class CustomProduct extends ProductBean {
        private static final long serialVersionUID = 6874908321131548936L;

        String author = null;

        public String getAuthor() {
                return author;
        }

        public void setAuthor(String author) {
                this.author = author;
        }

}
```

Note that we are extending the built-in `com.softslate.commerce.businessobjects.product.ProductBean` class. Strictly speaking, you don't have to extend the built-in class, but you must implement the `com.softslate.commerce.businessobjects.product.Product` interface.

2. Create the Hibernate mapping file for the new class. The Hibernate mapping file tells Hibernate how to map a class' properties to the columns of the database table it corresponds to. Name the file `CustomProduct.hbm.xml`, and place it next to the new `CustomProduct.class` class. Put the following into the contents of the file (note the "extends" attribute):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
        <subclass
                name="com.softslate.commerce.demo.CustomProduct"

extends="com.softslate.commerce.businessobjects.product.ProductBean">
                <property name="author" length="255"/>
        </subclass>
</hibernate-mapping>
```

If you are adding more than one property to your extended class, include all of them inside the `<subclass>` tag.

As of version 3.0 there is no need to add a `lazy="false"` attribute to the Hibernate mapping (all the built in classes have the attribute set as false by default).

3. Now add the new column to the product database table:

```
ALTER TABLE sscProduct ADD author VARCHAR(255);
```

4. Update the product database table's `class` field to use the new custom class. This tells Hibernate to instantiate instances of the new class when loading data from the database:

   ```
   UPDATE sscProduct SET class = 'com.softslate.commerce.demo.CustomProduct';
   ```

5. In the `/WEB-INF/src/appComponents-custom.properties` file, add a new property for the customization. In it set the value of the productImplementer property with the fully-qualified classname of our new custom class.

   ```
   productImplementer = com.softslate.commerce.demo.CustomProduct
   ```

   This tells the application to load the `CustomProduct.hbm.xml` file as part of the Hibernate initialization process.

6. To display our new author field, first put some data into the new column of the database:

   ```
   UPDATE sscProduct SET author = 'Truman Capote' WHERE productID = 1;
   ```

7. Then, create a custom version of the `product.jsp` JSP template in the `/WEB-INF/layouts/custom/product` directory. Paste some JSP code into the custom template that displays the value of the new field if it is defined:

   ```
   <c:if test="${!empty product.author}">
   	<tr>
   		<td class="productLabel" valign="top" nowrap="nowrap">
   			Author:
   		</td>
   		<td class="productData" valign="top">
   			<bean:write name="product" property="author"/>
   		</td>
   	</tr>
   </c:if>
   ```

8. Finally, restart your application server or reload the application for your changes to take effect.

After extending the product object to add new properties, you may need to make them accessible in the Administrator so they can be viewed and edited there. The following example takes us through the steps required to add a new field to the Administrator interface.

Adding a New Product Field to the Administrator Screens

1. Create a new class that extends the built in `com.softslate.commerce.administrator.product.ProductForm` class. This will handle the processing and display of the the control screen in the administrator. We'll call our class `CustomProductForm` and we'll put it in a package named

`com.softslate.commerce.demo` . Here's the source code for our new `CustomProductForm` class:

```
package com.softslate.commerce.demo;

import java.util.Arrays;

import javax.servlet.http.HttpServletRequest;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionMessages;

import com.softslate.commerce.administrator.product.ProductForm;

/**
 * Adding author field
 *
 * @author David Tobey
 */
public class CustomProductForm extends ProductForm {

        private static final long serialVersionUID = 1571534730516598703L;

        static Log log = LogFactory.getLog(CustomProductForm.class);

        // Add the new field to the list of fields
        private String[] fields = { "productID", "code", "name", "keywords",
                        "shortDescription", "description", "isActive",
"unitCost",
                        "unitPrice", "altPrice", "weight", "header",
"footer", "isTaxed",
                        "smallImage", "mediumImage", "largeImage", "extra1",
"extra2",
                        "extra3", "extra4", "extra5", "primaryCategoryID",
                        "manufacturerID", "productOrder", "author" };

        // Add a human readable label for it
        private String[] fieldLables = { "Product ID", "Code", "Name",
"Keywords",
                        "Short Desc", "Description", "Active", "Cost",
"Price",
                        "Alt Price", "Weight", "Header", "Footer", "Taxed",
"Small Image",
                        "Medium Image", "Large Image", "Extra 1", "Extra 2",
"Extra 3",
                        "Extra 4", "Extra 5", "Pri Category", "Manufacturer",
"Order",
                        "Author" };

        // Should it be displayed on the control screen by default? Yes.
        private String[] displayFields = { "code", "name", "isActive",
"unitPrice",
                        "productOrder", "author" };

        private String[] searchableFields = { "code", "name", "keywords",
                        "shortDescription", "extra1", "extra2", "extra3",
"extra4",
                        "extra5", "author" };

        // Need a property for the control screen's power mode - a String[]
        private String[] author = null;

        public String[] getAuthor() {
                return author;
        }

        public void setAuthor(String[] author) {
                this.author = author;
        }

        public String[] getDisplayFields() {
                return displayFields;
        }

        public void setDisplayFields(String[] displayFields) {
                this.displayFields = displayFields;
        }

        public String[] getFieldLables() {
                return fieldLables;
        }

        public void setFieldLables(String[] fieldLables) {
                this.fieldLables = fieldLables;
        }

        public String[] getFields() {
                return fields;
        }

        public void setFields(String[] fields) {
                this.fields = fields;
        }
        public String[] getSearchableFields() {
                return searchableFields;
        }

        public void setSearchableFields(String[] searchableFields) {
                this.searchableFields = searchableFields;
        }

        // Let's say we must have an author - so let's add it to the
validation
        public ActionErrors validate(ActionMapping mapping,
                        HttpServletRequest request) {
                if (log.isDebugEnabled())
                        log.debug("Starting validation.");
                setProperties();
                // Run the original validations
                ActionErrors errors = super.validate(mapping, request);
                if (getProductID() != null) {
                        for (int i = 0; i < getProductID().length; i++) {
                                if
(Arrays.asList(getDisplayFields()).contains("code")) {
                                        if (getAuthor().length < i + 1 ||
getAuthor()[i] == null
                                                        || getAuthor()
[i].equals("")) {

errors.add(ActionMessages.GLOBAL_MESSAGE,
                                                                new
ActionMessage("errors.required", "Author"));
                                        }
                                }
                        }
                }
                return errors;
        }
}
```

2. Similarly, create a new class that extends the built in `com.softslate.commerce.administrator.product.ProductAddEditForm` class. This will handle the processing and display of the the product detail screen in the administrator. We'll call our class `CustomProductAddEditForm` . Here's the source code for this class:

```
package com.softslate.commerce.demo;

import javax.servlet.http.HttpServletRequest;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionMessages;

import com.softslate.commerce.administrator.product.ProductAddEditForm;

/**
 * Adding author property
 *
 * @author David Tobey
 *
 */
public class CustomProductAddEditForm extends ProductAddEditForm {

        private static final long serialVersionUID = 4589930821507654940L;

        static Log log = LogFactory.getLog(CustomProductAddEditForm.class);

        private String author = null;

        public String getAuthor() {
                return author;
        }

        public void setAuthor(String author) {
                this.author = author;
        }

        public ActionErrors validate(ActionMapping mapping,
                        HttpServletRequest request) {
                if (log.isDebugEnabled())
                        log.debug("Starting validation.");
                initializeProperties(mapping, request);
                super.validate(mapping, request);
                if (getCode() == null || getCode().equals("")) {
                        getErrors().add(ActionMessages.GLOBAL_MESSAGE,
                                        new ActionMessage("errors.required",
"Author"));
                }
                return getErrors();
        }
}
```

3. Now let's tell Struts to use our new form classes instead of the defaults. Add the following Action Form definitions to the <form-beans> section of `/WEB-INF/conf/administrator/core/struts-config-custom.xml`:

```
     <!-- Adding an author field to the product admin screens  -->
```

```
        <form-bean
              name="productAddEditForm"
              type="com.softslate.commerce.demo.CustomProductAddEditForm">
        </form-bean>
        <form-bean
              name="productForm"
              type="com.softslate.commerce.demo.CustomProductForm">
        </form-bean>
```

4. Next is to add the HTML form elements for the new field to our JSP templates. Let's start with the product detail screen. The JSP template responsible for it is `/WEB-INF/templates/administrator/default/product/productFormGuts.jsp`. We want to create a custom version of that file, so create a new file of the same name in the `custom` directory: `/WEB-INF/templates/administrator/custom/product/productFormGuts.jsp`

   In our custom version of `productFormGuts.jsp`, place the following content to display a text box for the new author field. Note we'll include the original JSP file to avoid duplication:

```
<%@ include file="/WEB-INF/layouts/default/core/tagDefinitions.jsp" %>

<!-- Start custom productFormGuts.jsp -->
<tr>
        <td class="genericLabel">Author:</td>
        <td>
                <html:text maxlength="100" name="productAddEditForm"
property="author"/>
        </td>
</tr>
<tr>
        <td>

        </td>
        <td class="genericData">
                Enter the author of this book.
                <br />
                <br />
        </td>
</tr>

<jsp:include page="/WEB-
INF/templates/administrator/default/product/productFormGuts.jsp"/>

<!-- End custom productFormGuts.jsp -->
```

5. Next is the control screen for products. The JSP template responsible for displaying the product fields on that screen is `/WEB-`

`INF/templates/administrator/default/product/productFieldColumns.jsp`. We want to use the same file, but just add some new logic for the author field. So let's place a *copy* of `productFieldColumns.jsp` in the `custom` directory: `/WEB-INF/templates/administrator/custom/product/productFieldColumns.jsp`

In our copy of `productFieldColumns.jsp`, *add* the following lines in the appropriate spots in that file to display the new author field, both when "Power Edit" is on and when it is off. (Some analysis of the JSP tags is necessary here!)

When Power Edit is on:

```
<logic:equal name="field" value="author">
        <input type="text" size="15" maxlength="100"
                name="<bean:write name="field"/>"
                value="<bean:write name="item"
                property="<%= field %>"/>"
                class="genericGridData"/>
</logic:equal>
```

When Power Edit is off:

```
<logic:equal name="field" value="partNumber">
        <bean:write name="item" property="<%= field %>"/>
</logic:equal>
```

6. Restart your application server or reload the application for your changes to take effect.