TriView

Leveraging Hibernate Validator for your validation needs

By Ted Bergeron
Co-Founder
Triview, Inc.
ted@triview.com

Once in a while a tool comes along that really satisfies the goals of developers and architects alike. Such a tool allows developers to start using it in their applications the same day they first download it. Ideally, the tool allows developers to reap significant benefits long before they have invested the amount of time needed to master its usage. Such a tool pleases architects by guiding developers towards an implementation of greater academic purity. The validator component of Hibernate Annotations is just such a tool. *(Note: Usage of the Hibernate Validator does not require your application to use Hibernate for persistence.)*

JSE 5 brought many needed enhancements to the java language, none with more potential than annotations. With annotations, developers finally have a standard, first class metadata framework for their java classes. Hibernate users have been manually writing *.hml.xml files for years, or using XDoclet to automate this task. Manually creating xml files requires a developer to update two files for each persistent property needed. (The class definition and xml mapping document) Using HibernateDoclet simplifies this, but requires developers to verify that their version of HibernateDoclet supports the version of Hibernate they wish to use. The doclet information is also unavailable at runtime as it is coded into javadoc style comments. Hibernate Annotations improve on these alternatives by providing a standard, concise manner of mapping classes with the added benefit of runtime availability.

Example 1: Hibernate mapping code using HibernateDoclet.

```
/**
 * @hibernate.property column="NAME" length="60" not-null="true"
 */
public String getName() {
    return this.name;
}

/**
 * @hibernate.many-to-one column="AGENT_ID" not-null="true" cascade="none"
 *                        outer-join="false" lazy="true"
 */
public Agent getAgent() {
    return agent;
}
/**
 * @hibernate.set lazy="true" inverse="true" cascade="all" table="DEPARTMENT"
 * @hibernate.collection-one-to-many class="com.triview.model.Department"
 * @hibernate.collection-key column="DEPARTMENT_ID" not-null="true"
 */
public List<Department> getDepartment() {
    return department;
}
```

Example 2: Hibernate mapping code using Hibernate Annotations

```
@NotNull
@Column(name = "name")
@Length(min = 1, max = NAME_LENGTH) // NAME_LENGTH is a constant declared
elsewhere
public String getName() {
    return name;
}

@NotNull
@ManyToOne(cascade = {CascadeType.MERGE }, fetch = FetchType.LAZY)
@JoinColumn(name = "agent_id")
public Agent getAgent() {
    return agent;
}

@OneToMany(mappedBy = "customer", fetch = FetchType.LAZY)
public List<Department> getDepartment() {
    return department;
}
```

When using HibernateDoclet, mistakes will not be caught until generating the xml files, or until runtime. With annotations, many errors can be detected at compile time, or during editing when using a good IDE. When creating an application from scratch, developers may take advantage of the hbm2ddl utility to generate the ddl for their database from the hbm.xml files. Information such as the name property having a max length of 60 characters or that the ddl should add a not null constraint are added to the ddl from the HibernateDoclet entries. When using annotations, ddl can be generated automatically in a similar manner.

While both code mapping options are capable, annotations provides us some clear advantages. We can use constants to specify lengths or other values. We have a much faster build cycle without the need to generate xml files. The biggest advantage is that useful information such as a not null annotation or length can be accessed at runtime. In addition to the above annotations, developers can specify validation constraints. Some of the included constraints available are:

@Max(value = 100)
@Min(value = 0)
@Past
@Future
@Email

When appropriate, these annotations will cause check constraints to be generated with the ddl. (Obviously, @Future is not an appropriate case) You can also create custom constraint annotations as needed.

Validation and Application Layers

Creating validation can be a tedious, time consuming process. Often, lead developers will make the decision to forgo addressing validation in a given layer to save time. It is debatable whether or not the drawbacks of cutting corners in this area are offset by the time savings. If the time

investment needed to create and maintain validation across all application layers can be greatly reduced, the debate swings towards having validation in more layers. Suppose we have an application that lets a user create an account with a username, password and credit card.

- View – Validation via JavaScript is desirable to avoid server round trips, providing a better end user experience. Users can disable JavaScript, so while good to have, this level of validation is not reliable. Simple validations of required fields are a must.

- Controller – Validation must be processed in the server side logic. Code at this layer can be sure to handle validations in a manner appropriate for the specific use case. For example, when adding a new user, the controller may check to see if the specified username already exists before proceeding.

- Service – Relatively complex business logic validation is often best placed into the service layer. For example, once we have a Credit Card object that appears to be valid, we should verify the card information with our credit card processing service.

- Dao – By the time data reaches this layer it really "should" be valid. Even so, it would be beneficial to perform a quick check that required fields are not null, values fall into specified ranges, or adhere to specified formats, such as email. Better to catch a problem here than to cause avoidable SQLExceptions.

- DBMS – A common place to find validation being ignored. If today the application is the only client of the database that may change in the future. If the application has bugs (and most do) invalid data may reach the database. In this event, if you are lucky, you will find the invalid data and need to figure out if and how the data can be cleaned.

- Model – An ideal place for validations that do not require access to outside services or knowledge of the persistent data. For example, your business logic dictates that users must provide at least one of an email address or a phone number.

A typical approach to validation is to use Commons Validator for simple validations and write additional validations in the controller. Commons Validator has the benefit of generating JavaScript to process validations in the view. A drawback to commons validator is that it can only handle simple validations and stores the validation definition in an xml file. Commons Validator was designed to be used with Struts and does not provide an easy way to reuse the validation declarations across the application layers.

When designing your validation strategy choosing to simply handle errors as they occur is not sufficient. A good design will also aim to prevent errors from being made by generating a friendly UI. Taking a proactive approach to validation can greatly enhance a user's perception of an application. Unfortunately, Commons Validator fails to offer support for this. Suppose I want my HTML to set the maxlength attribute of text fields to match the validation, or place a % icon after text fields meant to collect % values? Typically that information is hard coded into HTML documents. If we decide to change our name property to support 75 characters instead of 60, how many places will a developer need to make changes? In many applications they will need to:

- Update the DDL to increase the database column length (Via HibernateDoclet, hbm.xml, or Hibernate Annotations)
- Update the commons validator xml file to increase the max to 75.
- Update all HTML forms related to this field to change the maxlength attribute.

A better approach is to use Hibernate Validator. Validation definitions are added to the model layer via annotations, with support for processing the validations included. If you choose to leverage all of Hibernate, the validator will help provide validation in the Dao and DBMS layers as well. We will take this a step further by using reflection and JSP 2.0 tag files, leveraging the annotations to dynamically generate code for the view layer. This will remove the practice of hard coding business logic in the view layer.

Example 3: Simple Contact mapped via Hibernate Annotations

```
/**
 * A Simplified object that stores contact information.
 *
 * @author Ted Bergeron
 * @version $Id: Contact.java,v 1.1 2006/04/24 03:39:34 ted Exp $
 */
@MappedSuperclass
@Embeddable
public class Contact implements Serializable {
    public static final int MAX_FIRST_NAME = 30;
    public static final int MAX_MIDDLE_NAME = 1;
    public static final int MAX_LAST_NAME = 30;

    private String fname;
    private String mi;
    private String lname;
    private Date dateOfBirth;
    private String emailAddress;

    private Address address;

    public Contact() {
        this.address = new Address();
    }

    @Valid
    @Embedded
    public Address getAddress() {
        return address;
    }

    public void setAddress(Address a) {
        if (a == null) {
            address = new Address();
        } else {
            address = a;
        }
    }

    @NotNull
    @Length(min = 1, max = MAX_FIRST_NAME)
    @Column(name = "fname")
    public String getFirstname() {
        return fname;
    }
```

```
    public void setFirstname(String fname) {
        this.fname = fname;
    }

    @Length(min = 1, max = MAX_MIDDLE_NAME)
    @Column(name = "mi")
    public String getMi() {
        return mi;
    }

    public void setMi(String mi) {
        this.mi = mi;
    }

    @NotNull
    @Length(min = 1, max = MAX_LAST_NAME)
    @Column(name = "lname")
    public String getLastname() {
        return lname;
    }

    public void setLastname(String lname) {
        this.lname = lname;
    }

    @NotNull
    @Past
    @Column(name = "dob")
    public Date getDateOfBirth() {
        return dateOfBirth;
    }

    public void setDateOfBirth(Date dateOfBirth) {
        this.dateOfBirth = dateOfBirth;
    }

    @NotNull
    @Email
    @Column(name = "email")
    public String getEmailAddress() {
        return emailAddress;
    }

    public void setEmailAddress(String emailAddress) {
        this.emailAddress = emailAddress;
    }
```

In example 3, we see that dateOfBirth is annotated as being NotNull and in the past. Hibernate's DDL generation will add a not null constraint to this column and a check constraint requiring the date to be in the past. The email address is also not null, and must match the email format. This will generate a not null constraint, but will not generate a check constraint matching the format.

Hibernate Dao implementations will use the validation annotations also if desired. All you need to do is specify Hibernate event-based validation in the hibernate.cfg.xml file. (See documentation)

If one really wanted to cut corners, they could just catch the `InvalidStateException` in their Service or Controller and iterate over the `InvalidValue` array.

Adding Validation to the Controller

To perform the validations, we will need to create an instance of Hibernate's `ClassValidator`. This class can be expensive to instantiate, so it is considered good practice to instantiate this once per class you wish to validate. One approach is to create a utility class or a singleton storing one ClassValidator instance per model object.

Example 4: Utility class to process validation

```
/**
 * Handles validations based on the Hibernate Annotations Validator framework.
 * @author Ted Bergeron
 * @version $Id: AnnotationValidator.java,v 1.5 2006/01/20 17:34:09 ted Exp $
 */
public class AnnotationValidator {
    private static Log log = LogFactory.getLog(AnnotationValidator.class);

    // It is considered a good practice to execute these lines once and cache the
validator instances.
    public static final ClassValidator<Customer> CUSTOMER_VALIDATOR = new
ClassValidator<Customer>(Customer.class);
    public static final ClassValidator<CreditCard> CREDIT_CARD_VALIDATOR = new
ClassValidator<CreditCard>(CreditCard.class);

    private static ClassValidator<? extends BaseObject> getValidator(Class<?
extends BaseObject> clazz) {
        if (Customer.class.equals(clazz)) {
            return CUSTOMER_VALIDATOR;
        } else if (CreditCard.class.equals(clazz)) {
            return CREDIT_CARD_VALIDATOR;
        } else {
            throw new IllegalArgumentException("Unsupported class was passed.");
        }
    }

    public static InvalidValue[] getInvalidValues(BaseObject modelObject) {
        String nullProperty = null;
        return getInvalidValues(modelObject, nullProperty);
    }

    public static InvalidValue[] getInvalidValues(BaseObject modelObject, String
property) {
        Class<? extends BaseObject>clazz = modelObject.getClass();
        ClassValidator validator = getValidator(clazz);

        InvalidValue[] validationMessages;

        if (property == null) {
            validationMessages = validator.getInvalidValues(modelObject);
        } else {
            //only get invalid values for specified property.  For example, "city"
```

```
applies to getCity() method.
            validationMessages = validator.getInvalidValues(modelObject,
property);
        }
        return validationMessages;
    }
}
```

In example 4, we create two ClassValidators, one for Customer and another for CreditCard. Classes wanting to apply validation can call `getInvalidValues(BaseObject modelObject)` returning `InvalidValue[]`. This will return an array containing the errors for the model object instance. Alternatively, the method can be called with a specific property name supplied, returning only errors pertaining to that field.

Example 5: CreditCardValidator used by Spring MVC Controller.

```
/**
 * Performs validation of a CreditCard in Spring MVC.
 *
 * @author Ted Bergeron
 * @version $Id: CreditCardValidator.java,v 1.2 2006/02/10 21:53:50 ted Exp $
 */
public class CreditCardValidator implements Validator {

    private CreditCardService creditCardService;

    public void setCreditCardService(CreditCardService service) {
        this.creditCardService = service;
    }

    public boolean supports(Class clazz) {
        return CreditCard.class.isAssignableFrom(clazz);
    }

    public void validate(Object object, Errors errors) {
        CreditCard creditCard = (CreditCard) object;

        InvalidValue[] invalids =
AnnotationValidator.getInvalidValues(creditCard);

        if (invalids == null || invalids.length == 0) { // Perform "expensive"
validation only if no simple errors found above.
            boolean validCard = creditCardService.validateCreditCard(creditCard);
            if (!validCard) {
                errors.reject("error.creditcard.invalid");
            }
        } else {
            for (InvalidValue invalidValue : invalids) {
                errors.rejectValue(invalidValue.getPropertyPath(), null,
invalidValue.getMessage());
            }
        }
    }
}
```

Creating a validator for credit cards becomes very simple when using Spring MVC and Hibernate Validator. The validate method only needs to pass the creditCard instance to the validator to have the array of InvalidValues returned. If we find one or more of these "simple" errors, we can translate Hibernate's InvalidValue array into Spring's Errors object. If the user has created this credit card without any simple errors, we can delegate a more thorough validation to the service layer. This layer may verify the card with our merchant services provider.

We have now discussed how simple model layer annotations can be leveraged into validations for the Controller, Dao and DBMS layers. Duplications in validation logic found in HibernateDoclet and Commons Validator are now consolidated into our model. While this is a very welcome improvement, the view layer has traditionally been the one most in need of better validation.

Adding Validation to the View

In my examples, I will be using Spring MVC and JSP 2.0 tag files. JSP 2.0 allows for custom functions to be registered in a tld file and called in a tag file. Tag files are like taglibs, but are written in JSP, not java. In this way, code that is best written in java can be encapsulated in a function, while code best written in JSP can be put into the tag file. In this case, processing the annotations requires reflection which will be performed by several functions. Code to bind to spring or render the XHTML will be part of the tag file.

Example 6: Creating the Form TLD

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
         http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
        version="2.0">

    <tlib-version>1.0</tlib-version>
    <short-name>form</short-name>
    <uri>formtags</uri>

    <tag-file>
        <name>text</name>
        <path>/WEB-INF/tags/form/text.tag</path>
    </tag-file>

<function>
    <description>determine if field is required from Annotations</description>
    <name>required</name>
    <function-class>com.triview.web.Utilities</function-class>
    <function-signature>Boolean
required(java.lang.Object,java.lang.String)</function-signature>
</function>

</taglib>
```

Here in the TLD excerpt, we have defined the text tagfile to be available, and the function called "required".

Example 7: Excerpt from Utilities

```
public static Boolean required(Object object, String propertyPath) {
    Method getMethod = findGetterMethod(object, propertyPath);
    if (getMethod == null) {
        return null;
    } else {
        return getMethod.isAnnotationPresent(NotNull.class);
    }
}

public static Boolean isDate(Object object, String propertyPath) {
    return java.util.Date.class.equals(getReturnType(object, propertyPath));
}

public static Class getReturnType(Object object, String propertyPath) {
    Method getMethod = findGetterMethod(object, propertyPath);
    if (getMethod == null) {
        return null;
    } else {
        return getMethod.getReturnType();
    }
}
```

In this excerpt from the Utilities class we have the required function referenced in the TLD, and another method that indicates whether or not a given property is a Date. There is a bit too much code here, but the findGetterMethod and others all perform basic reflection in addition to converting from EL method representations (customer.contact) to java representations (customer.getContact()). Here we can see exactly how easy it is to use our Validation annotations at runtime. We can simply get a reference to an object's getter method and check if that method has a given annotation associated with it.

Example 8: Simple JSP page containing a form

```
<%@ taglib tagdir="/WEB-INF/tags/form" prefix="form" %>

<form method="post" action="<c:url value="/signup/customer.edit"/>">

<form:select propertyPath="creditCard.type"
collection="${creditCardTypeCollection}" required="true"
labelKey="prompt.creditcard.type"/>

<form:text propertyPath="creditCard.number" labelKey="prompt.creditcard.number">
    <img src="<c:url value="/images/icons/help.png"/>" alt="Help" onclick="new
Effect.SlideDown('creditCardHelp')"/>
</form:text>

<form:text propertyPath="creditCard.expirationDate"/>
</form>
```

This JSP example is very simplified so we can focus on the relevant parts. Here we have a form with a select box and two input fields. All of these fields will be rendered via our tag files declared in the form.tld. The tag files are designed to use intelligent default values allowing for simply

coded JSP with the option of specifying more information if desired. The key attribute is the propertyPath, which maps the field to the model layer property using EL notation, just as with Spring MVC's bind tag.

Example 9: Excerpt of Tag file text.tag

```
<%@ attribute name="propertyPath" required="true" %>
<%@ attribute name="size" required="false" type="java.lang.Integer" %>
<%@ attribute name="maxlength" required="false" type="java.lang.Integer" %>
<%@ attribute name="required" required="false" type="java.lang.Boolean" %>

<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@ taglib uri="formtags" prefix="form" %>

<c:set var="objectPath" value="${form:getObjectPath(propertyPath)}"/>

<spring:bind path="${objectPath}">
    <c:set var="object" value="${status.value}"/>
    <c:if test="${object == null}">
<%-- Bind ignores the command object prefix, so simple properties of the command
object return null above. --%>
        <c:set var="object" value="${commandObject}"/> <%-- We depend on the
controller adding this to request. --%>
    </c:if>
</spring:bind>

<%-- If user did not specify whether this field is required, query the object for
this info. --%>
<c:if test="${required == null}">
    <c:set var="required" value="${form:required(object,propertyPath)}"/>
</c:if>

<c:choose>
    <c:when test="${required == null || required == false}">
        <c:set var="labelClass" value="optional"/>
    </c:when>
    <c:otherwise>
        <c:set var="labelClass" value="required"/>
    </c:otherwise>
</c:choose>

<c:if test="${maxlength == null}">
    <c:set var="maxlength" value="${form:maxLength(object,propertyPath)}"/>
</c:if>

<c:set var="isDate" value="${form:isDate(object,propertyPath)}"/>

<c:set var="cssClass" value="input_text"/>
<c:if test="${isDate}">
    <c:set var="cssClass" value="input_date"/>
</c:if>

<div class=”field”>
<spring:bind path="${propertyPath}">
<label for="${status.expression}" class="${labelClass}"><fmt:message
```

```
key="prompt.${propertyPath}"/></label>
<input type="text" name="${status.expression}" value="${status.value}"
id="${status.expression}"<c:if test="${size != null}"> size="${size}"</c:if>
<c:if test="${maxlength != null}"> maxlength="${maxlength}"</c:if>
class="${cssClass}"/>

<c:if test="${isDate}">
    <img id="${status.expression}_button" src="<c:url
value="/images/icons/calendar.png"/>" alt="calendar" style="cursor: pointer;"/>
    <script type="text/javascript">
        Calendar.setup(
        {
            inputField : "${status.expression}", // ID of the input field
            ifFormat : "%m/%d/%Y", // the date format
            button : "${status.expression}_button" // ID of the button
        }
        );
    </script>
</c:if>

<span class="icons"><jsp:doBody/></span>

<c:if test="${status.errorMessage != null && status.errorMessage != ''}">
    <p class="fieldError"><img id="${status.expression}_error" src="<c:url
value="/images/icons/error.png"/>" alt="error"/>${status.errorMessage}</p>
</c:if>

</spring:bind>
</div>
```

The full source for the text tag file is too large to include here, so we will review the key parts. Right away we see that `propertyPath` is the only required attribute. `Size`, `maxlength` and `required` may be optionally specified.

The `objectPath` var is set to the parent object of the property referened in the `propertyPath`. Thus if the `propertyPath` were customer.contact.fax.number, `objectPath` would be set to customer.contact.fax. We now bind to the object containing our property with spring's bind tag. This will set our object variable as a reference to the instance containing our property. Next, we check to see if the user of this tag has specified whether they want our property to required or not. It is important to allow form developers to override the value that would be returned from the annotations in case they wish to have the controller set a default value for a required field that the user may not wish to provide. If the form developer has not specified a value for required, we call the required function of the form tld. This function calls the method mapped in the tld file. The method simply checks for the `@NotNull` annotation. If it finds that the property has a `@NotNull` annotation, it sets the labelClass variable to required. We can determine the proper maxlength, and if the field is a Date in a similar manner.

Next we use Spring to bind to the `propertyPath`, as opposed to just the object containing the property as before. This allows us to use `status.expression` and `status.value` when generating the label and input html tags. The input tag also will be generated with a size, maxlength and a class as appropriate. If we determined our property to be a Date earlier, we now add in a javascript calendar. Notice how simple it is to link the label for attribute, input id and img id together as needed. This javascript calendar requires the img id to match the input field, with "_button" appended.

Finally, we wrap the `<jsp:doBody/>` in a span tag, allowing for the form developer to add additional icons, such as for help, to the page. Example 8 shows a help icon being added to the credit card number field. The last piece is to check if Spring has reported an error for this property and to display it, along with an error icon.

Example 10: CSS code to decorate required fields

```
label.required {
    color: black;
    background-image: url( /images/icons/flag_red.png );
    background-position: left;
    background-repeat: no-repeat;
}
label.required:after {
    content: '*';
}
label.optional {
    color: black;
}
```

With a little CSS, we now can have our required fields appear red, have an asterisk added to the text, or decorated with a background image. (Note, the CSS that adds the asterisk does not work in IE 6). Our date input fields automatically have a JavaScript calendar icon placed to the right. Having proper maxlength attributes set for all text fields prevents errors caused by the user entering too much text. We could extend the text tag to set the input field class for other data types as well. We could modify the text tag to use HTML instead of XHTML if that is your preference. We can receive the benefits of having proper semantic HTML forms without much of the hassle. We receive many of the benefits of a component based web framework without having to learn a component based framework.

While the tag file generates html that helps prevent errors, we do not have any code in the view layer to actually check for errors. Thanks to our use of class attributes, we can add some simple JavaScript to do this. Our JavaScript can be generic and reusable in any of our forms.

Example 11: Simple javascript validator

```
<script type="text/javascript">
    function checkRequired(form) {
        var requiredLabels = document.getElementsByClassName("required", form);
        for (i = 0; i < requiredLabels.length; i++) {

var labelText = requiredLabels[i].firstChild.nodeValue; // Get the label's text
var labelFor = requiredLabels[i].getAttribute("for"); // Grab the for attribute
var inputTag = document.getElementById(labelFor); // Get the input tag

            if (inputTag.value == null || inputTag.value == "") {
                alert("Please make sure all required fields have been entered.");
                return false; // Abort Submit
            }
        }
        return true;
    }
</script>
```

This JavaScript will be called by adding `onsubmit="return checkRequired(this);"` to the form declaration. The script simply gets all of the elements in the form that have the class "required". As our convention is to use this class in the label tags, the code then finds the input field joined to the label via its for attribute. If any of these input fields are empty, a simple alert message is generated and the form submit is aborted. This script could easily be extended to scan for several classes, validating accordingly.

For a comprehensive set of JavaScript based validations, it may be better to use an open source offering rather than roll your own. You may have noticed in example 8 the code: `onclick="new Effect.SlideDown('creditCardHelp')"/>` This function is part of the superb Script.aculo.us library, providing many advanced effects. If you are using Script.aculo.us, you are in turn using the Prototype library that it is built upon. One example of a JavaScript validation library is built on Prototype by Andrew Tetlaw. (See resources) His framework relies on classes added to the input fields:

```
<input class="required validate-number" id="field1" name="field1" />
```

The text.tag logic can easily be modified to insert several classes in the input tag. Adding class="required" to the input tag in addition to the label tag will not affect our css rules, but would break the simple JavaScript validator shown in example 10. If you are going to mix simple JavaScript code with that of a framework, it may be better to use different class names, or to be sure and inspect the tag type when searching for the elements using the classname.

Final Thoughts

This document has shown how Annotations of your model layer can be leveraged to create validations in your view, controller, dao and DBMS layers. Developers must create Service layer validations, such as our credit card validation, manually. Additional model layer validation, such as forcing property C to be required when properties A and B are in specified states also remains a manual task. However, the majority of validations can now be declared and applied with ease.

Going Further

The JavaScript validations of either the simple example or the referenced framework, can check for simple conditions such as required or matching a data type in client side code. Validations requiring server side logic may be added to your JavaScript validator using AJAX. You may have a user registration screen that lets a user pick a username. The text tag could be enhanced to check for an `@Column(unique = true)` annotation. When finding this annotation, the tag could add a class used in triggering an AJAX call.

With the time saved by not having to maintain duplicate validation logic across your application layers, just think of the enhancements you'll finally be able to add to your application.

Resources:

XDoclet / HibernateDoclet:
http://xdoclet.sourceforge.net/xdoclet/tags/hibernate-tags.html

Hibernate Validator Documentation:
http://www.hibernate.org/hib_docs/annotations/reference/en/html/validator.html

Really easy field validation with Prototype * Dexagogo
http://tetlaw.id.au/view/blog/really-easy-field-validation-with-prototype/

Script.aculo.us – web 2.0 JavaScript
http://script.aculo.us/

Prototype JavaScript Framework
http://prototype.conio.net/

The DHTML / JavaScript Calendar
http://www.dynarch.com/projects/calendar

Free Icons
http://www.famfamfam.com/lab/icons/silk

Ted Bergeron is the Co-Founder for Triview, an Enterprise Software Consulting company based in San Diego, California. Ted has been working in web based application design for over 10 years. Notable projects in that time include applications for Sybase, Orbitz, Disney and Qualcomm. Ted has also spent 3 years as a technical instructor teaching courses on web development, java development and logical database design.

You may learn more about Triview at our website http://www.triview.com, or by calling us toll free at (866)TRIVIEW.