

Hibernate can meet your validation needs

Annotations make easier data validation possible

Level: Intermediate

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

12 Sep 2006

While it's important to build data validation into as many layers of a Web application as possible, it's traditionally been very time-consuming to do so, leading many developers to just skip it -- which can lead to a host of problems down the road. But with the introduction of annotations in the latest version of the Java™ platform, validation got a lot easier. In this article, Ted Bergeron shows you how to use the Validator component of Hibernate Annotations to build and maintain validation logic easily in your Web apps.

Once in a while, a tool comes along that really satisfies the goals of developers and architects alike. Developers can start using such a tool 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.

Java SE 5 brought many needed enhancements to the Java language, none with more potential than *annotations*. With annotations, you finally have a standard, first-class metadata framework for your Java classes. Hibernate users have been manually writing *.hbm.xml files for years (or using XDoclet to automate this task). If you manually create XML files, you must update two files (the class definition and the XML mapping document) for each persistent property needed. Using HibernateDoclet simplifies this (see Listing 1 for an example) but requires you to verify that your version of HibernateDoclet supports the version of Hibernate you wish to use. The doclet information is also unavailable at run time, as it is coded into Javadoc-style comments. Hibernate Annotations, illustrated in Listing 2, improve on these alternatives by providing a standard, concise manner of mapping classes with the added benefit of run-time availability.

What you need to know before you start

Before you read this article, you should have a basic understanding of version 5 of the Java platform (annotations specifically), JSP 2.0 (because I create tag files and define functions in a TLD, both new features of JSP 2.0), and the Hibernate and Spring frameworks. Please note that you can use Hibernate Validator in your application even if you don't use Hibernate for persistence.

Listing 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;
}

```

Listing 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;
}

```

If you use HibernateDoclet, you won't be able to catch mistakes until you generate the XML files or until run time. With annotations, you can detect many errors at compile time, or, if you're using a good IDE, during editing. When creating an application from scratch, you can take advantage of the hbm2ddl utility to generate the DDL for your database from the hbm.xml files. Important information -- that the `name` property must have a maximum length of 60 characters, say, or that the DDL should add a not null constraint -- is added to the DDL from the HibernateDoclet entries. When you use annotations, you can generate the DDL automatically in a similar manner.

While both code mapping options are serviceable, annotations offer some clear advantages. With annotations, you can use constants to specify lengths or other values. You have a much faster build cycle without the need to generate XML files. The biggest advantage is that you can access useful information such as a not null annotation or length at run time. In addition to the annotations illustrated in Listing 2, you 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

Writing validation code can be a tedious, time-consuming process. Often, lead developers will forgo addressing validation in a given layer to save time, but 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 toward having validation in more layers. Suppose you have an application that lets a user create an account with a username, password, and credit card number. The application components into which you would ideally like to incorporate validation are as follows:

- **View:** Validation through JavaScript is desirable to avoid server round trips, providing a better user experience. Users can disable JavaScript, so while this level of validation is good to have, it's 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 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 you have a credit card object that appears to be valid, you should verify the card information with your 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 to make sure that required fields are not null and that values fall into specified ranges or adhere to specified formats -- for instance, an e-mail address field should contain a valid e-mail address. It's better to catch a problem here than to cause avoidable `SQLExceptions`.
- **DBMS:** This is a common place to find validation being ignored. Even if the application you're building is the only client of the database today, other clients may be added 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 it can be cleaned up.
- **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 may dictate that users provide at least one form of contact information, either a phone number or an e-mail address; you can use model layer validation to make sure that they do.

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. But Commons Validator does have its drawbacks: it can only handle simple validations and it 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 application layers.

When planning your validation strategy, choosing to simply handle errors as they occur is not sufficient. A good design also aims to prevent errors by generating a friendly user interface. 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 you want your HTML to set the `maxLength` attribute of text fields to match the validation or place a percent sign (%) after text fields meant to collect percentage values. Typically, that information is hard-coded into HTML documents. If you decide to change your `name` property to support 75 characters instead of 60, in how many places will you need to make changes? In many applications, you 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 through annotations, with support for processing the validations included. If you choose to leverage all of Hibernate, the validator helps provide validation in the DAO and DBMS layers as well. In the code samples that follow, you 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.

In Listing 3, `dateOfBirth` is annotated as being `NotNull` and in the past. Hibernate's DDL generation adds a not null constraint to this column and a check constraint requiring the date to be in the past. The e-mail address is also not null and must match the e-mail format. This generates a not null constraint but does not generate a check constraint matching the format.

Listing 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;
}

```

Hibernate DAO implementations use the validation annotations as well, if you want them to. All you need to do is specify Hibernate event-based validation in the hibernate.cfg.xml file. (See the Hibernate Validator documentation for more information; you can find a link in the [Resources](#) section.) If you really wanted to cut corners, you could just catch the `InvalidStateException` in the service or controller and iterate over the `InvalidValue` array.

Adding validation to the controller

To perform the validations, you 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 it once for each class you wish to validate. One approach is to create a utility class or a singleton storing one `ClassValidator` instance per model object, as illustrated in Listing 4:

Listing 4. Utility class to process validation

Sample application

In the [Download](#) section, you can download a sample application that demonstrates the ideas and code established in the article. Because it is a working application, the code is more extensive than the excerpts discussed in the article. For example, Listing 9 is an excerpt from the tag file `text.tag`; the sample app has all of the code for the tag file, along with code for three additional similar tag files (for a select, hidden, and checkbox HTML elements). Because the sample app is a working app, it contains a skeleton of what would be found in almost any application of its type. There is an Ant build file, Spring and Hibernate XML wiring code, and a log4j configuration. None of that is the focus of the article, but you may find perusing

```

/**
 * 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 Listing 4, you create two `ClassValidators`, one for `Customer` and another for `CreditCard`. Classes wanting to apply validation can call `getInvalidValues(BaseObject modelObject)`, returning `InvalidValue[]`. This returns 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.

Creating a validator for credit cards becomes very simple when using Spring MVC and Hibernate Validator, as illustrated in Listing 5:

Listing 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);

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

```

The `validate()` method only needs to pass the `creditCard` instance to the validator to have the array of `InvalidValues` returned. If you find one or more of these simple errors, you can translate Hibernate's `InvalidValue` array into Spring's `Errors` object. If the user has created this credit card without any simple errors, you can delegate a more thorough validation to the service layer. This layer may verify the card with your merchant services provider.

Now you see 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 your 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 the examples that follow, I use 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 code, not Java code. In this way, code that is best written in the Java language can be encapsulated in functions, while code best written in JSP code 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.

The TLD excerpt in Listing 6 defines the `text.tag` file to be available and a function called `required`:

Listing 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>
```

Listing 7 is an excerpt from the `Utilities` class, which contains all of the functions used by the tag file. Above, I noted that the code that is best written as Java code is placed in several functions that TLD then maps so that the tag file can use them; these functions are coded in `Utilities`. Therefore, you need three pieces: the TLD file defining everything, the functions in `Utilities`, and the tag file itself, which uses those functions. (The fourth piece would be the JSP page that uses the tag file.)

In Listing 7, you 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 in this class to cover it all without the length of this article getting really out of hand, but note that the `findGetterMethod()` and others all perform basic reflection in addition to converting from Expression Language (EL) method representations (`customer.contact`) to Java representations (`customer.getContact()`).

Listing 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();
    }
}
```


Here you can see exactly how easy it is to use Validation annotations at runtime. You can simply get a reference to an object's getter method and check to see if that method has a given annotation associated with it.

The JSP example in Listing 8 is simplified so you can focus on the relevant parts. Here, you have a form with a select box and two input fields. All of these fields are rendered through our tag files declared in the form.tld file. The tag files are designed to use intelligent default values, allowing for simply coded JSPs 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.

Listing 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">
  " alt="Help"
    onclick="new Effect.SlideDown('creditCardHelp')"/>
</form:text>

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

The full source for the text.tag file is too large to include here, so Listing 9 reviews the key parts:

Listing 9. Excerpt from the 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="${isDate}">
    " alt="calendar"
    style="cursor: pointer;"/>
    <script type="text/javascript">
        Calendar.setup(
        {
            inputField : "${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="${status.errorMessage != null && status.errorMessage != ''}">
    <p class="fieldError">"
    alt="error"/>${status.errorMessage}</p>
</c:if>

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

Right away, you can 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 referenced in the `propertyPath`. Therefore, if the `propertyPath` were `customer.contact.fax.number`, `objectPath` would be set to `customer.contact.fax`. You now bind to the object containing your property with Spring's `bind` tag. This sets your object variable as a reference to the instance containing your property. Next, you check to see if the user of this tag has specified

whether they want your property to be 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`, you 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 this annotation, it sets the `labelClass` variable to be required. You can determine the proper `maxLength` and whether or not the field is a `Date`, in a similar manner.

Next, you use Spring to bind to the `propertyPath`, as opposed to just the object containing the property as you did before. This allows you 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 `class` as appropriate. If you determined your property to be a `Date` earlier, you now add in a JavaScript calendar. (You can find a link to an excellent calendar component in the [Resources](#) section below.) Notice how simple it is to link the label for attribute, input ID, and image ID together as needed. This JavaScript calendar requires the image ID to match the input field, with `_button` appended.

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

With a little CSS, you can decorate your required fields -- for instance, by having them appear in red, have an asterisk added to their text, or be decorated with a background image. In Listing 10, you set the label for required fields to be black with a red asterisk added after in Firefox and other standards-compliant browsers, while adding a flag background image to the left in Internet Explorer:

Listing 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;
}
```

The 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. You could extend the `text` tag to set the input field class for other data types as well. You could modify the `text` tag to use HTML instead of XHTML if that is your preference. You can reap the benefits of having proper semantic HTML forms without much of the hassle, and you 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, you do not have any code in the view layer to actually check for errors yet. Thanks to the use of class attributes, you can add some simple JavaScript to do this, shown in Listing 11. Your JavaScript can be generic and reusable in any of your forms.

Listing 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 is 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 your convention is to use this class in the label tags, the code then finds the input field joined to the label through its `for` attribute. If any of these input fields are empty, a simple alert message is generated and the form submit is aborted. You could easily extend this script 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 to roll your own. You may have noticed the following code in Listing 8:

```
onclick="new Effect.SlideDown('creditCardHelp')"
```

This function is part of the superb Script.aculo.us library, which provides many advanced effects. If you are using Script.aculo.us, you are in turn using the Prototype library upon which it is built. One example of a JavaScript validation library is built on Prototype by Andrew Tetlaw. (See the [Resources](#) section for a link.) His framework relies on classes added to the input fields:

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

You can easily modify the text.tag logic to insert several classes in the `input` tag. Adding `class="required"` to the input tag in addition to the `label` tag will not affect your CSS rules but would break the simple JavaScript validator shown in Listing 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 be sure and inspect the tag type when searching for the elements using the class name.

Final thoughts

This article has shown how annotations of your model layer can be leveraged to create validations in your view, controller, DAO, and DBMS layers. You must create service-layer validations, such as the 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, with the Validator component of Hibernate Annotations, you can declare and apply the majority of validations with ease.

Going further


The JavaScript validations of either the simple example or the referenced framework can check for simple conditions, such as a field being required or a data type in client-side code matching the expected type. 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 a `@Column(unique = true)` annotation. When finding this annotation, the tag could add a class used in triggering an Ajax call.

Now that you don't need to maintain duplicate validation logic across your application layers, you'll free up a lot of development time. Just think of the enhancements you'll finally be able to add to your applications!

Download

Description	Name	Size	Download method
Sample application	j-hibval-source.zip	8MB	HTTP

→ [Information about download methods](#)

 [Get Adobe® Reader®](#)

Resources

Learn

- [XDoclet @hibernate tag reference](#): Learn more about XDoclet and HibernateDoclet.
- [Hibernate Validator documentation](#): Learn the ins and outs of Hibernate Validator.
- [Really easy field validation with Prototype](#): From Andrew Tetlaw's blog.
- [The Java technology zone](#): Hundreds of articles about every aspect of Java programming.

Get products and technologies

- [Hibernate](#) and [Spring](#): These frameworks form the basis for this article's examples.
- [Script.aculo.us](#): JavaScript for Web 2.0 apps.
- [Prototype](#): Check out this JavaScript framework.
- [The DHTML / JavaScript Calendar](#): Integrate it into your projects.
- [Silk Icons](#): These free icons are good for helping identify fields in Web forms.

Discuss

- [developerWorks blogs](#): Get involved in the developerWorks community.
-

About the author



Ted Bergeron is the co-founder of 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 three years as a technical instructor teaching courses on Web development, Java development, and logical database design. You can learn more about Triview at its [Web site](#) or by calling the company's toll-free number, (866)TRIVIEW.
