James E. Harmon

Foreword by
Dylan Schiemann, co-creator of Dojo

Dojo

Using the Dojo JavaScript Library to Build Ajax Applications

Developer's Library

Library of Congress Cataloging-in-Publication Data

Harmon, James Earl.

Using the Dojo Javascript library to build Ajax applications / James Earl Harmon.

p. cm.

Includes index.

ISBN 0-13-235804-2 (pbk. : alk. paper) 1. Ajax (Web site development technology) 2. Java (Computer program language) I. Title.

TK5105.8885.A52H37 2008

006.7'8—dc22

2008021544

ISBN-13: 978-0-132-35804-0
ISBN-10: 0-132-35804-2
Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.
First printing June 2008

Associate Publisher
Mark Taub

Acquisitions Editor
Debra Williams Cauley

Development Editor
Michael Thurston

Managing Editor
Kristy Hart

Project Editor
Chelsey Marti

Copy Editor
Language Logistics

Indexer
Lisa Stumpf

Proofreader
Kathy Ruiz

Technical Reviewer
Eric Foster-Johnson

Publishing Coordinator
Kim Boedigheimer

Cover Designer
Gary Adair

Senior Compositor
Gloria Schurick

Foreword

If there is one lesson to be learned from the Dojo Toolkit, it is "Be careful what you wish for!" When we first started Dojo, we had the modest goal of creating a JavaScript toolkit that would be useful and would prevent expert JavaScript developers from having to reinvent the wheel. With the buzz and excitement that would soon follow with the emergence of the term Ajax, we quickly found ourselves as the creators of a toolkit used by thousands and thousands of developers and millions of users in a very short time.

In the case of any project or company that grows much faster than expected, there are growing pains along the way. It has taken Dojo nearly 18 months to address and solve most of the issues caused by its rapid success: performance, comprehension, ease of use, and documentation. Open source projects are notoriously bad at both marketing and documentation, and Dojo was initially no exception to the rule. With each release from Dojo 0.9 to 1.1 and beyond, documentation and API viewing tools have improved significantly and are now something we're proud to have rather than being a blemish to the project.

Above and beyond source code documentation, demos, and great examples is the need for great books. When learning something new, the most difficult things to learn are usually the questions you don't know how to ask. The vernacular and philosophy of Dojo is very powerful and efficient but often leaves developers new to Dojo not knowing where to get started. Dojo in particular and Ajax in general also have the learning curve of basically needing to understand a wide range of technologies, from server-side programming languages to JavaScript, CSS, HTML, and the DOM, plus the browser quirks and inconsistencies across each. Toolkits such as Dojo go to great lengths to rescue developers from the most common and egregious of these issues, but developers creating something new will inevitably run into trouble along the way.

There are numerous opportunities for developers and users of Dojo to solve their problems and get up to speed, from reading this book to online community support, and the commercial support provided by companies such as SitePen.

Dojo has thrived and succeeded because of its transparent and open development process. All code is licensed under the AFL and BSD, licenses which are focused on adoption rather than control.

Contributions have been received from hundreds of individuals and from companies such as AOL, Google, IBM, Nexaweb, Renkoo, SitePen, Sun, WaveMaker, and many more. We have a strict but low-barrier contribution policy that requires all source code contributions to be made through a Contributor License Agreement, ensuring that usage of Dojo will not cause legal or IP headaches now or in the future.

And we innovate and experiment more than any other toolkit, introducing features in DojoX that are far ahead of other toolkits.

I first met James Harmon at a conference when he was giving a talk about Dojo. The great thing about James' approach was that he did an amazing job of simplifying the message. Alex Russell and I have a tendency to beat people over the head with every feature and every possibility, whereas James was able to distill complex topics down to easy-to-follow concepts that help people quickly get up to speed with Dojo.

This book takes the same simple approach of clearly explaining how to create web applications and web sites with Dojo in a manner that should make it easy, even for developers who are not JavaScript experts, to quickly get up to speed and become effective with the Dojo Toolkit.

Dylan Schiemann
CEO, SitePen
Cofounder, Dojo Toolkit

2

Using Dojo for Client-side Validation

To err is human…

—Alexander Pope (1688–1744)

We all make mistakes, so input forms must anticipate that users will inadvertently enter bad data. Identifying and correcting these mistakes is an important job of an HTML form, and this chapter describes Dojo features that allow you to easily add validation.

2.1 Validating Form Fields

Validating input data on web pages is usually a function performed by the server. The web page allows the user to enter data, and when the Submit button is pressed, the browser wraps up the data into an HTTP request and sends it to the server. The server checks each data field to make sure it is valid, and if any problems are found, a new form along with error messages is sent back to the browser. Wouldn't it be much more useful if problems could be detected in the browser before a server request is made? This approach would provide two primary advantages. It would lighten the load on the server, and, more importantly, it would notify the user of a problem with a data field almost immediately after he or she entered the bad data. This supports the truism that errors are cheapest to fix the closer the detection is to the original creation of the error. For example, if there is a problem with a zip code field and the user is notified just after he enters the bad zip code, then he is still thinking about zip code and can easily make the correction. If the user isn't notified until the server response comes back, he's already stopped

thinking about zip code—his mind has moved on to other concerns. This problem of context switching is especially difficult when the server returns errors for many different fields.

How can we drive validation closer to the entry of the data? There are two primary techniques available. The first technique involves trying to prevent the error from being entered at all. For example, if the form requires the user to enter a field that must contain a numeric value of a certain length, we can use the `size` attribute available in HTML to specify the maximum amount of characters the user can enter. So the user is prevented by the browser from entering more characters than are allowed. Following is an example from our form for the zip code field.

```
<label for="zipCode">Zip Code: </label>
<input type="text" id="zipCode" name="zipCode" size="10" /><br>
```

This initial validation markup gives us more optimism than is deserved. We might be hoping for many other attributes to provide some kind of client-side validation. Unfortunately, the `size` attribute is basically the extent of HTML-based validation techniques. There are no markup tags or attributes for minimum size or for data type. Nor is there a way in HTML to designate that a field is required.

That brings us to the second type of validation available to us in the browser. We can use JavaScript. Given the power of JavaScript, the sky is the limit in terms of types of validations we can perform. We can trigger a JavaScript function to run after the user enters a field, and that function can check to see if data is entered, check for a minimum or maximum length, or even perform sophisticated pattern matching using regular expressions.

Problem solved, correct? Not quite. The problem with depending on JavaScript as our validation technique is that we have to write lots of code to implement the checks. JavaScript code is required to perform the validation. Other JavaScript code tells the validation when to run. And even more JavaScript code is needed to display the error messages back to the user. Code, code, and more code. Suddenly, this approach doesn't seem as desirable anymore.

But this is where Dojo can come to the rescue. In this part of the tutorial, we explore how Dojo can help us with validation by combining the two techniques we've discussed. In other words, we'll be able to turn on validation by using simple HTML markup, but we'll let Dojo provide the complex JavaScript code automatically. Let's get started.

2.2 Tutorial Step 2—Adding Client-side Validation

In this step of the tutorial, we use Dojo to provide basic client-side validations. We look at a number of useful techniques within the context of making real enhancements to our form. One by one, we examine the fields that these techniques are appropriate for.

2.2.1 Validate the First Name Field

Let's look at the "First Name" field first. What are the validations that we need to apply? The data on this form feeds into our billing system, so the customer's name is very important—the field must be required. Are there any other validations? Not only do we want to get the data, but also we'd like it to be in a consistent format. Possibly the data should be stored in all capital letters. Or maybe we want to ensure that the data is *not* in all capitals. Let's choose the latter—but we'll still want to make sure that at least the first letter is capitalized. As in many of the issues related to validation, things are more complicated then they might first appear. For example, are we allowing enough room to enter long names? Will single-word names such as "Bono" be allowed? For our purposes, we'll keep it simple.

We turn on validation by using special attribute values in the HTML markup for these fields. The following code will add validation to the fields.

```
<label for="firstName">First Name: </label>
<input type="text" id="firstName" name="firstName"

    dojoType="dijit.form.ValidationTextBox"
    required="true"
    propercase="true"
    promptMessage="Enter first name."
    invalidMessage="First name is required."
    trim="true"

/><br>
```

The code is formatted to be more readable by using line breaks. To summarize what has happened: All we've done is add some new attributes to the `<input>` tag for the field. Each of the new attributes affects the validation in some way.

Notice the following line of code from the preceding example:

```
dojoType="dijit.form.ValidationTextBox"
```

This attribute is not a standard HTML `<input>` tag attribute. Depending on which editor you are using to modify the file, it may even be highlighted as an error. The `dojoType` attribute is only meaningful to the Dojo parser, which was referenced in step 1. Remember the code we needed to include the parser? It is shown here:

```
dojo.require("dojo.parser");
```

The parser reads through the HTML and looks for any tag that contains `dojoType` as one of its attributes. Then the magic happens. The parser replaces the element with the Dojo widget specified by `dojoType`. In this case, the widget `dijit.form.ValidationTextBox` is substituted for the Document Object Model (DOM) element created from the `<input>` tag.

How does Dojo know what to replace the tag with? That is determined by the specific widget. Each widget behaves a little differently. HTML markup and JavaScript code is associated with the widget in its definition, and that is how Dojo knows what to replace the original element with—which brings us to the missing piece of the puzzle. We need to tell Dojo to include the code for the widget by specifying the widget in JavaScript. To do that, we include the following JavaScript code after the link to Dojo and after the reference to the Dojo parser.

```
dojo.require("dijit.form.ValidationTextBox");
```

Notice that the name of the widget specified as the value for the `dojoType` attribute is the same as the argument for the `dojo.require` call. This is the linkage that allows Dojo to associate the HTML markup with the JavaScript code for that widget.

To emphasize this process, let's review the HTML markup specified in the original page and then compare it to the HTML markup after the parser runs. To see the original markup, we merely have to view the source of the file `form.html`. Seeing the new markup is a bit harder. The browser converts the original HTML into a DOM tree representing the various tags. The Dojo parser modifies the DOM elements using JavaScript, but the original source for the page is untouched. We need some tool that will convert the DOM (the browser's internal representation of the page) back into HTML for our review. The Firefox browser provides a DOM Inspector to do just that. An excellent add-on to Firefox, called Firebug, also allows the DOM to be inspected. Firebug also provides a number of excellent tools for developing web pages such as its DOM inspection capabilities we can use to inspect the DOM after the Dojo parser has run—so we can see exactly what it does. But before we see how the DOM changes, let's first review the original `<input>` tag for the first name field.

```
<input
    type="text"
    id="firstName"
    size="20"
    dojoType="dijit.form.ValidationTextBox"
    required="true"
    propercase="true"
    promptMessage="Enter first name."
    invalidMessage="First name is required."
    trim="true"
/>
```

The code has been reformatted to make it more readable by adding some line breaks. The attributes from `dojoType` through `trim` are not valid HTML attributes. They are meaningful only to the Dojo parser and drive some features of the Dojo widget they pertain to. Now let's see what the HTML looks like after the parser runs.

```
<input
  type="text"
  tabindex="0"
  maxlength="999999"
  size="20"
  class="dijitInputField dijitInputFieldValidationError dijitFormWidget"
  name="firstName"
  id="firstName"
  autocomplete="off"
  style=""
  value=""
  disabled="false"

  widgetid="firstName"
  dojoattachevent="onfocus,onkeyup,onkeypress:_onKeyPress"
  dojoattachpoint="textbox,focusNode"
  invalid="true"
  valuenow=""

/>
```

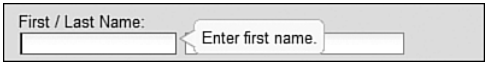
The preceding code has also been reformatted for readability, adding line breaks and changing the order of the attributes a little. Notice that a number of valid HTML attributes have been added to the `<input>` DOM element such as `tabindex`, `class`, `autocomplete`, and `disabled`. And additionally, a number of Dojo-only attributes have been added such as `widgetid`, `dojoattachevent`, `dojoattachpoint`, `invalid`, and `valuenow`. We look at these in more detail in Part II, "Dojo Widgets," but for now it's enough just to point out that the parser is rewriting our HTML. The parser is doing even more work that we can see here. It is associating various event handler functions to events that might occur on this DOM element. For instance, when the user enters or changes the value in the field, Dojo functions get called, which perform validation. And Dojo even creates objects that correspond to the HTML tags. We can't tell that this is happening just from seeing the HTML markup, but behind the scenes, that is exactly what Dojo is doing.

Let's review the other special Dojo attributes. Each Dojo widget has a set of properties that control its behavior. These properties are set by various Dojo widget attribute values.

- The `required="true"` attribute setting tells Dojo that this field must be entered.
- The `propercase="true"` attribute setting tells Dojo to reformat the field value entered by the user. In this case, the setting for `propercase` tells Dojo to make sure that the first letter is capitalized and subsequent letters are in lowercase. In other words, Dojo will put the entered value into the format for a typical proper noun.

- The `promptMessage="Enter first name."` attribute setting tells Dojo to display a message next to the field to instruct the user on what kind of data can be entered into the field. The prompt message displays while the field is in focus.
- The `invalidMessage="First name is required."` attribute setting causes Dojo to display a message next to the field if it fails the validation. In our case, if the user does not enter a value, then a message will appear.
- The `trim="true"` attribute setting tells Dojo to remove any leading or trailing spaces from the entered value before sending it to the server.

Now let's run the page and see how it behaves. Because this is the first field on the page, the field gets focus, and the cursor immediately is placed on the input area for the "First Name" field.



Notice that we get a message box that says "Enter first name." Dojo calls this a *Tool Tip*, and it has dynamic behavior. It is only displayed when the field has focus (the cursor is in the field), and once the field loses focus, the message disappears. The message appears on top of any visible element below it, so there is no need to leave room for it when designing your page.

Try entering different values in the field and then press `<tab>` to leave the field. For example, enter " joe " and watch it be transformed into "Joe" with leading and trailing spaces removed and the first letter of the name capitalized.

> **NOTE:**
>
> You might not agree with the various validations I have chosen. For example, one early review of this text pointed out that "LaToya" would be a hard name to validate. You could probably make a case for different validations, and I could probably agree with you. But I've chosen the ones I have not only to represent my example application, but also to highlight certain Dojo features—so I'm sticking to them!

2.2.2 Validating the Last Name Field

The last name field has the same validations as the first name field does. There is nothing extra to do for this field and nothing new to learn. Just replace the `<input>` tag for Last Name with the following code.

```
<input type="text" id="lastName" name="lastName"
    dojoType="dijit.form.ValidationTextBox"
    required="true"
    propercase="true"
    promptMessage="Enter last name."
```

```
    invalidMessage="Last name is required."
    trim="true"
/>
```

2.2.3 Validating the User Name Field

We are going to allow the user to manage his or her own account information in our application. To provide some security we need the user to make up a user name that he or she can use later to sign on to the system. This field will be required, and we'd like it to always be entered in lowercase. To validate this field, we'll use the same Dojo widget that we've already used, `dijit.form.ValidationTextBox`, but we'll use a new attribute called `lowercase` to force the transformation of the entered data into all lowercase letters.

There are some additional validations we'd like to do on this field. For instance, is this user name already assigned to someone else? We could check the server for existing values. However, because this validation requires interaction with the server, we'll save it for step 3 of the tutorial and focus on only the client-side validation right now.

The following HTML markup is needed to enable validation for this field.

```
<input type="text" id="userName" name="userName"
    dojoType="dijit.form.ValidationTextBox"
    required="true"
    promptMessage="Enter user name."
    trim="true"
    lowercase="true"
/>
```

2.2.4 Validating the Email Address Field

We need to communicate with our customers so we'll get their email addresses. This will be a required field. We'll also make it all lowercase for consistency. In addition, we'd like to make sure that the value entered in this field is also in the correct format for an email address. There is no way to know if it is a working email until we actually try to send something to it, but at least we can make sure that it contains a "@" character and appears to reference a valid domain.

How can we specify the desired format? By using a specialized pattern matching language known as *regular expressions*, we can specify a pattern of characters to check the value against. We need to build a regular expression to validate for email addresses. At this point in our discussions, let's not go on a long detour to discuss the building of these expressions.

> **NOTE:**
>
> Some great information on building regular expressions can be found at the Mozilla Developer Center at http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference:Global_Objects:RegExp.

The following is regular expression that can be used to validate most formats of email addresses—*most* because it is surprisingly difficult to validate for *all* possible email addresses. This is because of some of the unusual variations such as domains longer than four characters such as ".museum" or addresses consisting of a sub-domain. But the following regular expression will work for most.

```
[\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b]+
```

> **NOTE:**
>
> For more information on validating email addresses, the following link will get you to a Dojo Forum article describing a regular expression for email: http://dojotoolkit.org/forum/dijit-dijit-0-9/dijit-support/text-validation.

The `ValidationTextBox` contains a special property for validating against regular expressions. The attribute to use is `regExp`—just specify the regular expression as its value. Replace the `<input>` tag for email with the following code in "form.html" to specify validation for the email address field.

```
<input type="text" id="email" name="email" size="30"
    dojoType="dijit.form.ValidationTextBox"
    required="true"
    regExp="\b[a-zA-Z0-9._%-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}\b"
    promptMessage="Enter email address."
    invalidMessage="Invalid Email Address."
    trim="true"
/>
```

Validating email addresses is a really interesting subject. There are quite a few variants to the simple `name@company.com` format that we often see. For a really thorough discussion of email, you should review the RFC rules. The following link will get you to the Wikipedia page that describes email, from which you can link to the official RFC documents: http://en.wikipedia.org/wiki/E-mail_address.

2.2.5 Validating the Address Field

The address field will contain the first line of the user's mailing address. We'll make it required. We will use the `ValidationTextBox`, and we have seen all of the attributes already. Replace the `<input>` tag for address with the following code.

```
<input type="text" id="address" name="address" size="30"
    dojoType="dijit.form.ValidationTextBox"
    required="true"
    promptMessage="Enter address."
    invalidMessage="Address is required."
    trim="true"
/>
```

There are many additional validations that can be performed on address data, the most important being to ensure that the address is an actual address. Standard abbreviations such as "St" for "Street" could also be allowed. These additional validations could be done by a number of web services available from the U.S. Postal Service, but that is really outside the scope of this tutorial.

2.2.6 Validating the City Field

The city field will contain the value for the city in the user's mailing address. We'll make it required. We will use the `ValidationTextBox`. Replace the `<input>` tag for address with the following code.

```
<input type="text" id="city" name="city" size="30"
    dojoType="dijit.form.ValidationTextBox"
    required="true"
    promptMessage="Enter city."
    invalidMessage="City is required."
    trim="true"
/>
```

2.2.7 Validating the Zip Code Field

The zip code field is part of the mailing address and is required. There are some additional validations we can apply. Our hypothetical company is a U.S. corporation and only provides service to U.S. customers, so we'll limit our address to valid U.S. addresses, which means that the zip code must be in one of two forms. Either it is a 5-digit number, or it is a 5-digit number followed by a dash and then followed by a 4-digit number. If we can come up with a regular expression to test for either format, then we're golden!

Replace the `<input>` tag for zip code with the following to enable Dojo validation for this field.

```
<input type="text" id="zipCode" name="address" size="30"
    dojoType="dijit.form.ValidationTextBox"
    trim="true"
    required="true"
    regExp="\d{5}([\-]\d{4})?$"
    maxlength="10"
    promptMessage="Enter zip code."
    invalidMessage="Invalid zip code (NNNNN) or (NNNNN-NNNN)."
/>
```

An interesting feature of the preceding code is that we've got two overlapping validations. The `maxlength` attribute prevents the value from being over 10 digits, but so does that regular expression. What are the implications of this? One could argue that it is inefficient because both validations will be executed. But they each operate differently on the page, which might justify using both. If the user tries to enter a zip code that is 12

digits long, he will be notified as he tries to type the eleventh digit, rather than after typing all 12 digits and pressing tab to leave the field. By using both techniques, the error is detected sooner.

NOTE:

This chapter has stopped short of describing validations for the "Start Service" and "Comments" fields. This is because we will use more advanced Dojo widgets to validate these fields, which are described in Chapter 4, "Using Dojo Widgets."

Summary

The Dojo widget `dijit.form.ValidationTextBox` provides many common client-side validations. Include the `ValidationTextBox` by referencing it in the `<input>` tag for the field that needs the validation.

```
dojoType="dijit.form.ValidationTextBox"
```

Remember to tell the page that it needs the JavaScript code for the widget by coding a call to the `require` method somewhere after the call to the Dojo parser.

```
dojo.require("widget dijit.form.ValidationTextBox");
```

Additional attributes in the `<input>` tag specify behavior for the `ValidationTextBox`. A few are listed here:

`require="true"` makes the field required.

`trim="true"` removes leading blanks.

`lowercase="true"` converts field to all lower case letters.

We've now completed step 2 of the tutorial. The changes we've implemented have added client-side validation to our form. We were able to add validation almost exclusively through modifying the HTML—only a small amount of JavaScript was necessary to include the Dojo validation code. Client-side validation is an extremely powerful capability and makes our page much more usable. Yet by using Dojo, we obtain this power without the corresponding cost of writing a lot of JavaScript.

In this chapter we've focused on functionality that doesn't require a call to the server. In the next chapter the server will play a role. We'll make calls to the server using the `XMLHttpRequest` to get data and perform validations. Now that's Ajax!

A

B

C

D

E

F

G

H

I

J-K

L

M

N

O

P

R

S

T

U

V

W

X-Y

Z