At the Forge

JavaScript, Forms and Ajax

Reuven M. Lerner

Abstract

Dynamic user detection and registration made easy via Ajax.

Last month, we began to dip our toes into the water of Ajax, the shorthand name for asynchronous JavaScript and XML, which has taken the world of Web development by storm. Ajax applications are Web applications in every way, depending on the underlying combination of HTML, HTTP, URLs, JavaScript and CSS that provide a modern Web infrastructure. But, they also rely upon several of the features of modern JavaScript, including its ability to rewrite Web pages and also to make HTTP requests asynchronously.

And, indeed, this asynchronous behavior is what makes JavaScript—and Ajax, for that matter—so exciting for Web developers. No longer are we stuck with the modern equivalent of old-style 3270 terminals, with execution taking place on the server only when we move from one page to another. Now a Web page can be updated without having to reload the page.

Ajax is not a technical revolution, so much as a conceptual one, bringing with it new expectations from users and paradigms for developers. All of the technology behind Ajax has existed for several years, but it is only now that we are starting to take advantage of it in Web applications.

This month, we start to look at one simple example of an Ajax application, in a context that will probably be familiar to most Web developers: asking users to register with a Web site. By the end of this month's column, you will see how we can combine server-side programs, an HTML form and JavaScript to check the validity of a form before it is submitted. Next month, we will see how we can use Ajax to overcome the fatal flaws associated with this implementation, improving the efficiency, robustness and security of the application all at once.

Registering Users

If you have been developing Web sites for any length of time, you probably have needed to create a login system. Such systems come in all shapes and sizes, and require different levels of security. For our example application, we assume that each user has a user name and password. Actually, as you will see, we don't care that much about the password; the key issue here is the user name, which must be unique.

The first thing to do is create a simple table in our database to keep track of users:

```sql
CREATE TABLE Users (  
id SERIAL NOT NULL,
```
user name TEXT NOT NULL CHECK (username <> ''),
password TEXT NOT NULL CHECK (password <> ''),
email address TEXT NOT NULL CHECK (email address <> ''),

PRIMARY KEY(id),
UNIQUE(username)
);

The above table, defined using PostgreSQL syntax, keeps track of our users for us. Every user has a unique numeric ID, stored in the id column. (The special SERIAL datatype in PostgreSQL ensures that each row we INSERT into the database will have a unique value for id.) For the purposes of this column, we ignore the security issues associated with storing passwords in plain text.

Next, we define three columns of type TEXT, which PostgreSQL uses to define limitless text fields. Each of these fields is also given an integrity check to ensure that the value can be neither blank nor NULL. We also define the username column to be UNIQUE.

Now, from the standpoint of data integrity, we already have done our jobs. We can be sure that no two users will have the same user name, that each e-mail address (that is, each person) is allowed to have more than one user name in the system, and that the user name, e-mail address and password cannot be blank. Everything else is just icing on the cake, right?

Well, yes—but only if we are willing to give our users database errors. Most of us would prefer to offer a softer landing to our users, telling them not only that (for example) their chosen user name already has been taken by someone else, but also shielding them from the errors that PostgreSQL displays.

This means our application is going to need to take the user's requested user name, check for it in the database, and then either display an error message (prompting the user to try again) or INSERT a new row in the database.

Here is a simple example of how we might do this in a Web page. I use simple CGI programs written in Perl for the server-side examples this month, in no small part because they tend to be simple to understand and try on any host. Listing 1 (register.html) shows the HTML form users will see when they want to register with the site. Listing 2 (register.pl) shows the CGI program that will accept the form contents, check the user name in the database and then produce a message in response. My assumption is that the form will be in the main Web document root directory, and the CGI program will be in the cgi-bin directory. Obviously, if you are using a server-side language, such as PHP, the two can exist side by side.

Listing 1. register.html9160l1.qrk

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">

<head><title>Register</title></head>
<body>
<h2>Register</h2>
<form action="/cgi-bin/register.pl" method="post">
  <p>Username: <input type="text" name="username" /></p>
  <p>Password: <input type="password" name="password" /></p>
  <p>E-mail address: <input type="text" name="email_address" /></p>
  <p><input type="submit" value="Register" /></p>
</form>
</body>
</html>
Listing 2. register.pl

#!/usr/bin/perl
use strict;
use diagnostics;
use warnings;
use CGI;
use DBI;

# # Connect to the database
# #------------------------------------------------------------
$dbh = DBI->connect("DBI:Pg:dbname=$dbname",
                   $dbuser, $dbpassword,
                   { AutoCommit => 1, RaiseError => 1,
                   PrintError => 1, ChopBlanks => 1}) ||
                   print "<p>Error connecting: '$DBI::errstr' </p>";

# # CGI startup
# #------------------------------------------------------------
$query = new CGI;
print $query->header("text/html");
print $query->start_html(-title => "Site registration");

my $username = $query->param("username");
my $password = $query->param("password");
my $email_address = $query->param("email_address");

# # Check the parameters
# #------------------------------------------------------------
my @missing_data = ();
push @missing_data, "The username"
    unless $username;
push @missing_data, "A password"
    unless $password;
push @missing_data, "The e-mail address"
    unless $email_address;

if (@missing_data)
{
    foreach my $missing_field (@missing_data)
    {
        print "<p>Sorry, but you are missing: $missing_field</p>
        print "<ul>
        print "<li> $missing_field</li>
        print "</ul>
        print "<p>Please back up and try again.</p>
    }
exit;
}
#
# Try to register the user
my $select_sql = "SELECT COUNT(*) FROM Users WHERE username = ?";
my $select_sth = $dbh->prepare($select_sql);
$select_sth->execute($username);
my ($username_is_taken) = $select_sth->fetchrow_array();
#
# Is this username taken?  If so, give an error
if ($username_is_taken)
{
    print "<p>Sorry, but the username $username' was already taken. Please back up and try again.</p>
";
}
#
# Otherwise, insert the new trio into the database
else
{
    my $insert_sql = "INSERT INTO Users (username, password, email_address) VALUES (?, ?, ?)"
    $dbh->do($insert_sql, {}, $username, $password, $email_address);
    print "<p>Added the username $username' to the system!</p>
";
}
print $query->end_html;

The HTML form has three text fields, in which the user is requested to enter a user name, a password and an e-mail address. (I personally prefer to use e-mail addresses as user names, but I realize many people don't, so I'll add it this month.) We know from the database definition that the user name must be unique in the system, but the name and e-mail address can each exist multiple times.

The registration program (in Listing 2) is a relatively simple CGI program. It connects to the database using Perl's DBI interface, then uses the standard start of CGI programs, grabbing parameters and generally getting ready. The program then checks the database to see if the user name already exists, returning the number of rows it matches in the database. If no rows match, we can assume that the user name is available. (There is something of a race condition here, but we're not going to complicate things with transactions for this small example.)

**Making the Form Dynamic**

This is the type of registration form with which most of us are familiar. Moreover, this is the type of registration form many of us continue to implement on various sites. It's easy for the programmer to build, it's easy to understand and debug, and it's compatible with all browsers out there.

The problem is not in the technical underpinnings of the program, but rather in the user interface. From nontechnical users' perspectives, it doesn't make sense for them to enter a user name, password and e-mail address, then submit it, and only then find out that the user name is unacceptable. Surely there
must be a way to fix this!

The only way for forms to be checked before they are submitted to a server is to use a client-side language—that is, a language embedded within the Web browser, which can attach itself to browser window events. The universal standard for such a language is ECMAScript, because it was ECMA International (formerly known as the European Computer Manufacturers Association) that approved and published the standard. However, most people refer to ECMAScript by the language that inspired the standard, namely JavaScript.

JavaScript is almost always found within the pages of an HTML document. We can define and invoke functions inside of the document, triggering the invocations with event handlers. Thus, we can check the contents of a form when someone clicks the submit button, before the contents are sent to the server. We can change styles when the mouse moves on (or off) particular text and graphics. And, we can execute functions when someone enters or exits an HTML form element.

Listing 3 contains js-register.html, a modified version of register.html. The basic idea in this file is that as soon as the username text field is modified by the user, the browser executes the checkUsername function.

Listing 3. js-register.html

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>Register</title>
<script type="text/javascript">
  <!--
  var usernames = ['abc', 'def'];

  function removeText(node) { 
    if (node != null) 
      { 
        if (node.childNodes) 
        { 
          for (var i=0 ; i < node.childNodes.length ; i++) 
            { 
            var oldTextNode = node.childNodes[i];
            if (oldTextNode.nodeValue != null) 
            { 
              node.removeChild(oldTextNode);
            }
          }
        }
      }
  }

  function appendText(node, text) { 
    var newTextNode = document.createTextNode(text);
    node.appendChild(newTextNode);
  }

  function setText(node, text) { 
    removeText(node);
    appendText(node, text);
  }
```
function checkUsername() {
    var new_username = document.forms[0].username.value;
    var found = false;
    var warning = document.getElementById("warning");
    var submit_button = document.getElementById("submit-button");

    // Is this new username already taken? Iterate over
    // the list of usernames to be sure.
    for (i=0 ; i<usernames.length; i++)
    {
        if (usernames[i] == new_username)
        {
            found = true;
        }
    }

    // If we find the username, issue a warning and stop
    // the user from submitting the form.
    if (found)
    {
        setText(warning, "Warning: username "+ new_username +" was taken!");
        submit_button.disabled = true;
    }
    else
    {
        removeText(warning);
        submit_button.disabled = false;
    }
}
</script>
</head>
<body>
<h2>Register</h2>
<p id="warning"></p>
<form action="/cgi-bin/register.pl" method="post">
<p>Username: <input type="text" name="username" onchange="checkUsername()" /></p>
<p>Password: <input type="password" name="password" /></p>
<p>E-mail address: <input type="text" name="email_address" /></p>
<p><input type="submit" value="Register" id="submit-button" /></p>
</form>
</body>
</html>

This is the way that most client-side Web programs are structured. JavaScript functions do the actual work, but they are invoked by event handlers defined in the HTML. So, in Listing 3, we see:

<p>Username: <input type="text" name="username" onchange="checkUsername()" /></p>

This tells the browser that when the username text field changes, it should invoke the checkUsername() function. When this function is executed, it begins with the following:

var new_username = document.forms[0].username.value;
The new_username variable gets the value of the username text field. We do this by starting off with the document object (representing our HTML document), then taking the first element of its forms array (representing the first, and only, form in the document). The username property of the form gives us the node for the username text field, whose value we can then retrieve (as a string) with the value property.

Traversing the tree in this way is typical when working with JavaScript. However, it is also possible to jump immediately to a particular form element, assuming that element has been assigned an id attribute. IDs must be unique within a document, meaning that we can find a node with the appropriate method:

```javascript
var warning = document.getElementById("warning");
var submit_button = document.getElementById("submit-button");
```

Each of the above two lines uses document.getElementById to retrieve a node from the document tree, identified with an id attribute. (If nothing matches, the variable is set to the null value.)

The list of user names has been hard-coded in Listing 3, which is something you would never try in an actual application. I discuss this further below, and we will find production-quality, Ajax-style solutions next month.

### Dynamically Modifying the Page

Now that we have a list of user names in JavaScript, we want to force the user to choose a user name that does not clash with one already in use. We will do this by checking the proposed user name against the list that we have already collected. If the user name is already taken, we will warn the user by modifying the HTML of the current page and then by disabling the submit button. Only when the chosen user name is new and unique will the user be allowed to submit it to the server. This doesn't mean we will remove our uniqueness checks on the server or in the database, but it offloads some of that checking to the client and makes the application more immediately responsive to the user's needs.

We do this by iterating over usernames, the array containing user names:

```javascript
for (i=0 ; i<usernames.length; i++)
{
    if (usernames[i] == new_username)
    {
        found = true;
    }
}
```

If we find a match between the user's requested new user name and one in the array, then we set the found variable to be true. Otherwise, it continues to be set to false. This then tells us whether we need to warn the user about a conflict and disable the submit button or vice versa.

Warning the user consists of two steps. The first involves setting the warning text—that is, the text inside of the `<p>` tags above the form—to an appropriate message. We already set the variable warning to point to that node at the beginning of the checkUsername function, which means that we now must eliminate all children of the warning node. Actually, we don't want to eliminate all children, but merely the ones with a nodeValue property, because that is where text is stored. The removeText function does that by iterating over each of a node's children, checking to see if it contains text and removing it if it does:
if (node.childNodes)
{
    for (var i=0 ; i < node.childNodes.length ; i++)
    {
        var oldTextNode = node.childNodes[i];
        if (oldTextNode.nodeValue != null)
        {
            node.removeChild(oldTextNode);
        }
    }
}

Once we have removed text children from the warning node, we then can add a new text child to the
warning node, containing the message we want to display. This is done in the appendText function:

    function appendText(node, text) {
        var newTextNode = document.createTextNode(text);
        node.appendChild(newTextNode);
    }

As of this point, the user has received a warning about the chosen user name, indicating that it will not
be accepted because the user name was already taken. However, we cannot rely on users to read and
follow the instructions in a warning message. Rather, we should disable the form's submit button,
making it difficult for users even to send the bad user name to our server-side program. We can do this
by setting the submit button's disabled property:

    submit_button.disabled = true;

To recap—when the user enters a value in the username text field that is in the usernames array, we
remove any existing text children from the warning node. We then add a new text child node to
warning, indicating that the chosen user name already has been taken. Finally, we disable the submit
button in the HTML form.

Of course, we want the user to submit the form eventually, but only after entering a user name not in the
usernames array. This means we must remove text children from the warning node, and then re-enable
the submit button:

    removeText(warning);
    submit_button.disabled = false;

Sure enough, this combination of JavaScript functions seems to do the trick. User names that are not in
the usernames array remove any error messages and re-activate the form, allowing us to submit it to the
server-side CGI program and register with the site. User names that are in the array, however, produce a
warning and stop us from being able to submit the form. It's not Ajax just yet, but it is more responsive
to the user than our pure server-side solution.

Considerations

Of course, the program in Listing 3 is fatally flawed in several ways. The biggest, by far, is the fact that
the usernames array is hard-coded in the JavaScript. It goes without saying that hard-coding a list of
user names in this way is guaranteed to fail, because the list of users is stored in a database table, and we
have not connected the database with the program.
We could overcome this problem by generating the usernames array from the database. In other words, our server-side program would create part of our client-side JavaScript program dynamically. Thus, instead of what we see in Listing 3:

```javascript
var usernames = ['abc', 'def'];
```

we would use a server-side program to do something like the following:

```perl
my $output = "[";
my $sql = "SELECT username FROM Users";
my $sth = $dbh($sql);
$sth->execute();
while (my ($username) = $sth->fetchrow_array())
{
    $output .= "'$username', ";
}
$output .= "]";
```

We would then insert $output into the resulting HTML file, ensuring that the value of usernames would have the most complete and up-to-date list of user names in the system.

But even this is likely to cause serious security concerns in a production application, because it means that every user name in your system—including those with poorly chosen passwords—will be available to everyone visiting your registration page, simply by looking at the HTML source code. Although it is true that every user name has a password, and that someone would have to guess the password associated with a user name in order to break into your system, can you really vouch for the quality of every password? Moreover, the user names themselves might be clues as to the number or types of users on your system. In short, you really don't want a production system to list the user names for a potential attacker, secure as you might believe your system to be.

There is also an efficiency problem here. As your list of users grows, the length of the usernames array will grow as well. Can you imagine the time it would take to generate and download the JavaScript for a site with 10,000 users?

The solution to all of these problems is, of course, Ajax. Rather than checking the proposed new user name against an array in our JavaScript application, we will have JavaScript submit the proposed user name to the server, find out whether it already has been taken and act accordingly—all without forcing the user to switch to a different page of HTML! This is the underlying magic that makes Ajax applications so compelling; they keep you on the same page longer than traditional Web applications, thus providing a smoother user experience.

**Conclusion**

We're making some progress on our way to Ajax heaven. We now have an application—user registration—for which old-style Web development provides an answer, but one that feels clunky to the user. The solution we saw in this month's column works well, but requires that the JavaScript contain a usernames array with all user names on the system. For performance and security reasons, this is a bad idea, and we should look for a different solution. Next month, we will start to look at a genuine Ajax solution to this problem, making our application look and feel smoother, while increasing its security as well.
Books on Ajax and JavaScript

In working on these columns, I have found several good books on the subject of HTML, JavaScript, Ajax and related technologies.

The two most comprehensive books on the subject are O'Reilly's *JavaScript: The Definitive Guide* by David Flanagan and *Dynamic HTML: The Definitive Reference* by Danny Goodman. There is a fair amount of overlap between these two books, and they are definitely meant to be reference books rather than tutorials. That said, experienced Web developers interested in learning about client-side programming probably will learn a great deal from these books. And once you're experienced, you will undoubtedly use these two books often, checking everything from the cross-platform compatibility of various JavaScript objects to how JavaScript interfaces with the DOM.

Newer and less experienced Web developers would probably do well to start with a gentler introduction to these technologies. One of the best, and funniest, that I've seen is O'Reilly's *Head Rush Ajax* by Brett McLaughlin. My one criticism is touted as one of the book's strengths—namely, that it presents the same information in many different ways to ensure that you will remember it. The book might be a bit annoying for experienced Web developers who will want to get to the meat more quickly, and who might be frustrated by the repetition. Nevertheless, I think that this is a worthwhile read for anyone starting in the Ajax world.

A middle-of-the-road book that might appeal to more experienced Web developers, while providing a tutorial and introduction to many JavaScript concepts described here, is *Professional JavaScript for Web Developers* by Nicholas Zakas and published by Wrox. I didn't like the way in which Zakas used his own (freely available) JavaScript library throughout the book, but I did think that the examples and explanations were carefully chosen and interesting, and they also helped to illuminate some of the shadier sides of JavaScript. Zakas contributed to another Wrox book, *Professional Ajax* by Zakas, Jeremy McPeak and Joe Fawcett, which I found to be less integrated and less enjoyable than either the JavaScript book or the *Head Rush Ajax* book from O'Reilly.