Applied jQuery

DEVELOP AND DESIGN

Jay Blanchard

Applied jQuery

DEVELOP AND DESIGN

Jay Blanchard

Applied jQuery: Develop and Design
Jay Blanchard

Peachpit Press
1249 Eighth Street
Berkeley, CA 94710
510/524-2178
510/524-2221 (fax)

Find us on the Web at: www.peachpit.com
To report errors, please send a note to: errata@peachpit.com
Peachpit Press is a division of Pearson Education.

Editor: Rebecca Gulick
Development and Copy Editor: Anne Marie Walker
Technical Reviewer: Jesse R. Castro
Production Coordinator: Myrna Vladic
Compositor: Danielle Foster
Proofreader: Patricia Pane
Indexer: Valerie Haynes-Perry
Cover design: Aren Straiger
Interior design: Mimi Heft

ISBN 13: 978-0-321-77256-5
ISBN 10: 0-321-77256-3

9 8 7 6 5 4 3 2 1

Printed and bound in the United States of America

To Mom, who taught me there was magic in books,

and to Dad, who taught me there was magic in me.

ACKNOWLEDGMENTS

Projects like this are not possible without the support and understanding of a lot of people, something I really didn't understand when first embarking on the journey to create a book. Saying "thank you" isn't nearly enough, but I hope that you all understand how much I appreciate you!

Even with the blender of life roaring around us, Connie Kay, Kaitlyn, Brittany, Zach, Karla, and Morgan provided more love and support than you can imagine. I love you all!

To Rebecca Gulick, thank you for believing in me and helping a dream to come true!

To Anne Marie Walker, enough cannot be said about your gentle firmness, guidance, and subtle humor. I am eternally grateful to you!

To Jesse Castro, thanks for keeping me on the straight and narrow. Your insight, technical abilities, and encouragement blow me away!

To Larry Ullman, thanks for being the Ford Prefect to my Arthur Dent and guiding me through the galaxy! I kept my towel on my desk the whole time!

To Francis Govers, the twists and turns in my life are made all the more bearable by knowing that you are just a phone call or an e-mail away. Best friends don't get any better!

To the folks who have made up the teams of developers that I have worked with day in and day out, thank you for making me a better programmer and a better person! Your willingness to look over my shoulder and teach me something new is treasured beyond measure.

To the jQuery community, you are an amazing group of people, and I am honored to share electrons with you!

CONTENTS

INTRODUCTION

As Web designers, you are painstakingly compelled to grab Web surfers' attention as quickly as possible and then keep them on your site to absorb the content. In addition to the product, service, or information that you are providing, the site must be visually attractive and offer stimulating (and valuable) interaction. The jQuery library is the main ingredient for providing the icing on your Web-site cake. If applied well, the effects of jQuery will convince visitors and application users to click around and sample all of your content.

The trick is learning how to combine jQuery with other markup and languages effectively. You must gain knowledge in a wide range of disciplines, like HTML (HyperText Markup Language) and CSS (Cascading Style Sheets), to know how to properly mix in the right amount of jQuery. The goal of this book is to give you the knowledge to bring the HTML, CSS, and jQuery ingredients together to create compelling interactivity to your Web sites and applications.

Throughout the book, I'll also show you ways to use PHP, a popular server-side scripting language, and MySQL, a relational database product, to enhance your overall development and supercharge your applications. Both technologies translate easily to other Web development languages.

WHAT IS JQUERY?

Announced in 2006 by its creator, John Resig, jQuery quickly gained popularity and support as a new way to use JavaScript to interact with HTML and CSS. jQuery's simple selectors mimicked CSS selectors, making the library familiar and easy to learn for designers and developers alike. The jQuery library erased the worry that Web developers had suffered through when trying to create interactive sites across a wide range of browsers by handling most browser compatibility issues behind the scenes.

Topping off those two features is the shortened syntax used by jQuery. The following example shows how you would select an element based on its `id` attribute using jQuery:

```
$('#foo');
```

The jQuery selector is much shorter as opposed to the same example in old-school JavaScript:

```
document.getElementByID('foo');
```

It's no wonder that the Web-development community embraced jQuery's "write less, do more" mantra. Couple the simplicity of jQuery with its ability to support complex animations and achieve stupendous effects, and you get a JavaScript library that is flexible and capable of empowering you to provide your Web site visitors with an outstanding interactive experience.

WHO THIS BOOK IS FOR

This book is aimed at beginning to intermediate Web developers, but it doesn't matter where you are in your journey as a designer or developer. You should find examples in this book that will help you to bring your Web pages and applications to life with jQuery. It helps if you have a basic knowledge of HTML, CSS, JavaScript, and jQuery, but it is not necessary because the examples are fully baked and ready to go.

WHAT I USED

As of this writing, jQuery 1.5 had been released and is used for all of the examples in the book. You can download it at www.jquery.com. It is also available on the book's Web site at www.appliedjquery.com.

HTML, CSS, and JavaScript files are all plain-text files that you can create and edit in any plain-text editor.

Examples were all tested in Firefox 3 and Internet Explorer 8, with an occasional peek in Safari and Google Chrome.

WHERE TO FIND THE CODE

All of the code examples for the book are available from the *Applied jQuery* Web site at www.appliedjquery.com/downloads. There you can download a Zip file containing all of the examples, graphics, and other collateral needed to follow along.

The examples are arranged by chapter within the Zip file and include all of the necessary jQuery files to make the examples work right out of the box.

However, even though all of the files are available for download, I encourage you to type out each example as you progress through the book. Taking a hands-on approach will help you to learn how all of the technologies fit together and will reinforce the concepts in your brain.

LET'S GET STARTED

It's time for you to jump right in and get started learning how to use jQuery. In the first chapter I'll give you some good rules and tools to get you headed in the right direction for sweetening your Web development efforts with jQuery.

i

WELCOME TO jQUERY

WELCOME TO jQUERY

jQuery is one of the most popular JavaScript libraries in use today because it lets you build JavaScript Web pages and Web applications quickly and easily, accomplishing in a single line of code something that would have required dozens of lines of JavaScript code. Grab yourself a computer and the handful of tools outlined below, and then dig into the following six chapters.

jQUERY

jQuery, which is free to download and use, comes in the form of a single .js file that you link to from your Web page, and your code accesses the library by calling various jQuery functions. Go to jquery.com and download the jQuery library.

jQUERY UI

Next, you'll want to download the jQuery UI library from jQueryUI.com. This will equip you with some core interaction plugins as well as many UI widgets that I'll discuss later in the book.

TEXT EDITOR

You'll be doing some scripting, so get yourself a good text editor. Windows users typically opt for Microsoft Notepad or Notepad++, while Mac users often rely on BBEdit from Bare Bones Software.

BROWSER

Chances are you've already got a standards-compliant browser installed. Popular options are the latest versions of Microsoft Internet Explorer, Mozilla Firefox, Apple Safari, Google Chrome, and Opera.

TROUBLESHOOTER

I rely heavily on the Firebug Web development tool for troubleshooting. Go to http://getfirebug.com and get a version that's specific to your browser. It's 100% free and open source, and you'll be grateful you've got it installed when something goes wrong.

TESTING ENVIRONMENT

Rather than using an actual hosted Web site to test your jQuery creations, use a testing environment that's local on your own computer. I use XAMPP, which you can download from http://apachefriends.org.

4

BEING **EFFECTIVE** WITH **AJAX**

AJAX, one of the hottest technology combinations to enter the Web development landscape in years, has fueled a surge in interactive Web design with its ability to load new content into an existing DOM structure.

jQuery simplifies using AJAX with several shorthand methods for the basic AJAX methods. For most developers and designers, these shorthand methods will be all that they ever need to use. The jQuery AJAX shorthand methods `post`, `get`, and `load` are featured in this chapter. jQuery also provides a robust feature set, including callbacks, for developers who want to customize their AJAX calls to provide richer interactive experiences. I'll show you how to use several of jQuery's AJAX features to enhance Web sites and applications. Let's start by completing the form validation that you started in Chapter 3.

USING **AJAX** FOR **VALIDATION**

Simply put, AJAX (Asynchronous JavaScript and XML) lets you use JavaScript to send and receive information from the server asynchronously without page redirection or refreshes. You can use AJAX to grab information and update the Web page that your user is currently viewing with that information. Complex requests can be made to databases operating in the background.

When new users register to use the Web site, they need to have unique user names. Their user name will be associated with other information, such as photos they upload or articles they write. It will be the key that lets them update information about the photos they submit.

Make sure that you first set up the database for the Web site by running the SQL file *chap4/sql/peuser.sql* on your database. Running this script in MySQL or any other database platform will create the Web-site's database, a user for that database, and the table that will be used to store Web-site visitor registration information. You can then start building the PHP file that will respond to the actions the AJAX functions will request.

BUILDING THE PHP REGISTRATION AND VALIDATION FILE

Photographers who want to share their images and perhaps write articles on photography will need a way to register information with the site that will allow them to log in and gain access to site features not accessible to nonregistered users.

You can create an interaction for this that will appear very slick to the user. With jQuery's AJAX functionality, you can avoid page reloads or redirections to other pages (**Figure 4.1**). The AJAX engine will send the requests to the PHP scripts on the server without disruption to the user experience.

Using PHP and jQuery, you'll create the functions that will support the registration interaction.

1. Open a new text file and save it as *chap4/inc/**peRegister.php***.

NOTE: If you'd like to use the PHP file provided in the download, feel free to skip ahead to "Setting Up the jQuery Validation and Registration Functions" section. Be sure to edit the PHP file with the proper user name, password, and host name for the database connection to match what you have set up on your database server.

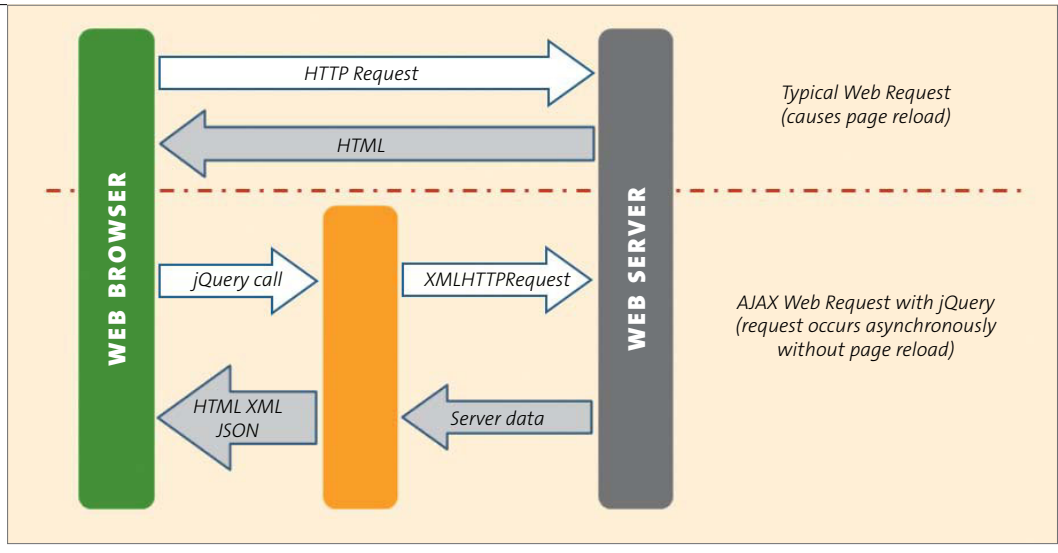


FIGURE 4.1
The difference between a typical HTTP request and the XMLHttpRequest utilized by jQuery's AJAX methods.

2. Set up the database connection for the PHP function, including a method for returning errors if no connection can be made:

```
if(!$dbc = mysql_connect('servername', 'username', 'password')){
    echo mysql_error() . "\n";
    exit();
}
```

Contained in this PHP file are three actions: one to complete registration, one to validate the user name, and a method to allow registered users to log in. The proper function will be called based on the name of the form used in the AJAX function.

3. Use PHP's `switch` method to determine which form is submitted and set up the first `case` for the registration form:

```
switch($_POST['formName']) {
    case 'register':
```

4. Check to see if the user name and password are set:

```
if(isset($_POST['penewuser']) &&
→ isset($_POST['penewpass'])) {
```

5. If the user name and password are set, use the data from the form to complete a SQL statement that will insert the new user's information into the database:

```
$peuserInsert = "INSERT INTO `photoex`.`peuser` ";
$peuserInsert .= "(`username`, `userpass`,
→ `userfirst`, `userlast`, `useremail`";
```

6. Because users can choose a number of photographic interests when they register, you must set up a loop to handle the check boxes that are selected in the registration form:

```
if(isset($_POST['interests'])){
```

7. The loop used here counts the number of interests selected and properly formats the SQL statement to name those interests. Insert commas in the correct place, and close the initial statement with a closing parenthesis:

```
$peuserInsert .= ",";
for($i = 0; $i < count($_POST['interests']);
→ $i++){
    if($i == (count($_POST['interests'])
    → - 1)){
        $peuserInsert .=
        → "`".$_POST['interests'][$i]."`";
    } else {
        $peuserInsert .=
        → "`".$_POST['interests'][$i]."`, ";
    }
}
   }
$peuserInsert .=")";
```

8. Place the values from the registration form into the SQL statement in the correct order:

```
$peuserInsert .= "VALUES (";
$peuserInsert .= "'".$_POST['penewuser']."', ";
$peuserInsert .= "'".$_POST['penewpass']."', ";
$peuserInsert .= "'".$_POST['pefirstname']."', ";
$peuserInsert .= "'".$_POST['pelastname']."', ";
$peuserInsert .= "'".$_POST['email']."' ";
```

9. Inserting the correct values includes looping through any interests selected in the form and inserting the value "yes" for those interests:

```
if(isset($_POST['interests'])){
    $peuserInsert .= ",";
        for($i = 0; $i < count($_POST
        → ['interests']); $i++){
            if($i == (count($_POST['interests'])
            → - 1)){
                $peuserInsert .= "'yes'";
            } else {
                $peuserInsert .= "'yes', ";
            }
        }
}
```

10. Close the SQL statement properly:

```
$peuserInsert .=")";
```

If you were to print out the resulting SQL statement contained in the variable `$peuserInsert`, it would look something like this:

```
INSERT INTO `photoex`.`peuser`(`username`, `userpass`,
→ `userfirst`, `userlast`, `useremail`,`landscape`,
→ `astronomy`,`wildlife`) VALUES ('Bob.Johnson','phOtOman',
→ 'Bob','Johnson','photoman@gmail.com','yes','yes','yes','yes')
```

11. Use the PHP function `mysql_query` to insert the data into the database, and the user will be registered:

```
if(!($peuInsert = mysql_query($peuserInsert,
→ $dbc))){
    echo mysql_errno();
    exit();
}
```

CHECKING THE USER NAME FOR AVAILABILITY

Because the new user will typically fill out the user name first, the password and user name will not be set, so the `else` statement will be invoked. This is the PHP code that checks the user name to see if it exists in the database.

1. Create a SQL query that selects the user name typed into the registration form from the user database:

```
   } else {
$peCheckUser = "SELECT `username` ";
$peCheckUser .= "FROM `photoex`.`peuser` ";
$peCheckUser .= "WHERE `username` =
→ '".$_POST['penewuser']."' ";
if(!($peuCheck = mysql_query($peCheckUser, $dbc))){
     echo mysql_errno();
     exit();
}
```

   If the name the user entered into the registration form is already in the database, the query will return a row count of 1. If the name is not in the database, the row count is 0.

2. Assign the count of the number of rows returned by the query to the database:

```
$userCount = mysql_num_rows($peuCheck);
```

3. Echo the count value to be returned by the AJAX function for use by jQuery to determine if the user should enter a new user name in the registration form:

```
        echo $userCount;
    }
```

4. Complete the case statement for the registration form:

```
break;
```

CREATING THE PHP FOR USER LOGIN

After registering, the user can log in to the site and begin uploading photos and writing articles. Let's complete the login section of the PHP file.

1. Set up the case statement for the login code:

```
case 'login':
```

2. Check to see if the user name and password are set:

```
    if(isset($_POST['pename']) && isset($_POST['pepass'])){
```

3. If they are set, send a query to the database with the user name and password information:

```
        $peLoginQ = "SELECT `username`, `userpass` ";
        $peLoginQ .= "FROM `photoex`.`peuser` ";
        $peLoginQ .= "WHERE `username` = '".$_POST['pename']."' ";
        $peLoginQ .= "AND `userpass` = '".$_POST['pepass']."' ";
        if(!($peLogin = mysql_query($peLoginQ, $dbc))){
            echo mysql_errno();
            exit();
        }
```

NOTE: You should always make sure that data visitors enter into forms is cleansed by checking the data rigorously before submitting it to the database.

FIGURE 4.2 The check box a user can click to be remembered. The user will not have to log in again until the cookie associated with this action expires or is removed from the computer.

4. Set the variable `$loginCount` to the number of rows returned from the database query. If the user name and password are correct, this value will be 1:

```
$loginCount = mysql_num_rows($peLogin);
```

Next, you'll set up a cookie depending on the user's preference. A *cookie* is a small file that is placed on the visitor's computer that contains information relevant to a particular Web site. If the user wants to be remembered on the computer accessing the site, the user can select the check box shown in **Figure 4.2**.

5. If the login attempt is good, determine what information should be stored in the cookie:

```
if(1 == $loginCount){
```

6. Set up a cookie containing the user's name to expire one year from the current date if the "remember me" check box was selected:

```
if(isset($_POST['remember'])){
    $peCookieValue = $_POST['pename'];
    $peCookieExpire = time()+(60*60*24*365);
    $domain = ($_SERVER['HTTP_HOST'] !=
    → 'localhost') ? $_SERVER['HTTP_HOST'] :
    → false;
```

The math for the `time()` function sets the expiration date for one year from the current date expressed in seconds, 31,536,000. A year is usually sufficient time for any cookie designed to remember the user. The information in the `$domain` variable ensures that the cookie will work on a *localhost* as well as any other proper domain.

7. Create the cookie and echo the `$loginCount` for AJAX to use:

```
                setcookie('photoex', $peCookieValue,
                → $peCookieExpire,
'/', $domain, false);
                echo $loginCount;
```

8. Set a cookie to expire when the browser closes if the user has not selected the remember option:

```
        } else {
                $peCookieValue = $_POST['pename'];
                $peCookieExpire = 0;
                $domain = ($_SERVER['HTTP_HOST'] !=
                → 'localhost') ? $_SERVER['HTTP_HOST'] :
                → false;
                setcookie('photoex', $peCookieValue,
                → $peCookieExpire,
'/', $domain, false);
                echo $loginCount;
        }
```

9. Echo out the login count if the user name and password are not set. The value should be 0:

```
    } else {
        echo $loginCount;
    }
}
break;
```

NOTE: For more on PHP and how to use it effectively with MySQL, check out Larry Ullman's book, PHP 6 and MySQL 5 for Dynamic Web Sites: Visual QuickPro Guide (Peachpit, 2008).

With the PHP file ready to go, it is time to build the jQuery AJAX functions.

SETTING UP THE JQUERY VALIDATION AND REGISTRATION FUNCTIONS

Checking the new user name should be as seamless as possible for the registrant. The form should provide immediate feedback to users and prompt them to make changes to their information prior to the form being submitted. The form input (in *chap4/4-1.php*) element for the user name will be bound to the `blur` method:

```
<label class="labelLong" for="penewuser">Please choose a user name:
→ </label><input type="text" name="penewuser" id="penewuser"
→ size="24" /><span class="error">name taken, please choose
→ another</span>
```

1. Bind the form input for the user name to jQuery's `blur` method:

```
$('#penewuser').blur(function() {
```

2. Capture the value of the user name in the `newName` variable:

```
    var newName = $(this).val();
```

Next, you'll validate with the `post` method.

1. Call the post method with the URL of the PHP script, data representing the name of the form that is being filled out, and the newName variable:

```
$.post('inc/peRegister.php', {
    formName: 'register',
    penewuser: newName
```

   Note that the data passed by the post method is in name: value pairs. The value in each pair is quoted when sending the raw data. Variables such as newName do not need the quotes.

   The results of calling the *inc/peRegister.php* script will automatically be stored for later processing in the data variable.

2. Define the callback for the post function and pass the data variable to the function, so that the results can be processed:

```
}, function(data){
```

   The PHP function returns only the row count based on the query that was used to see if the user name was in the database.

3. Set up a variable to hold the information returned in the data variable:

```
    var usernameCount = data;
```

4. Create a conditional statement that will display or hide the error message based on the data returned by the AJAX method. You'll recognize most of this conditional statement because it is similar to how validation error messages were delivered in Chapter 3:

```
    if(1 == usernameCount){
        $('#penewuser').next('.error').css('display',
        → 'inline');
    } else {
        $('#penewuser').next('.error').css('display',
        → 'none');
    }
```

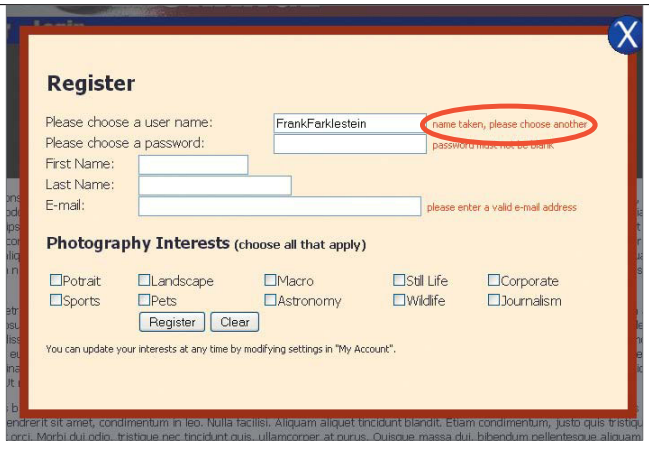


FIGURE 4.3 The user name FrankFarklestein is already in use by someone else. Who knew there were two of them?

5. Close out the `post` function by citing the data type you expect the server-side function to return:

```
    }, 'html');
});
```

   If the PHP function returns a 1, the error `span` is displayed, as illustrated in **Figure 4.3**.

The registration function needs to submit the user's data or let the user know if there are still errors with the submission. If there are errors, the user needs to be prompted to fix the registration.

1. Start the registration function by binding the registration form to the `submit` method:

```
$('#registerForm').submit(function(e) {
```

   The variable e holds information about the event object, in this case the submit event.

2. Because you will be using AJAX to submit the form, you do not want the submit event to perform as it normally would. To stop that from happening, you set the event to `preventDefault`:

```
    e.preventDefault();
```

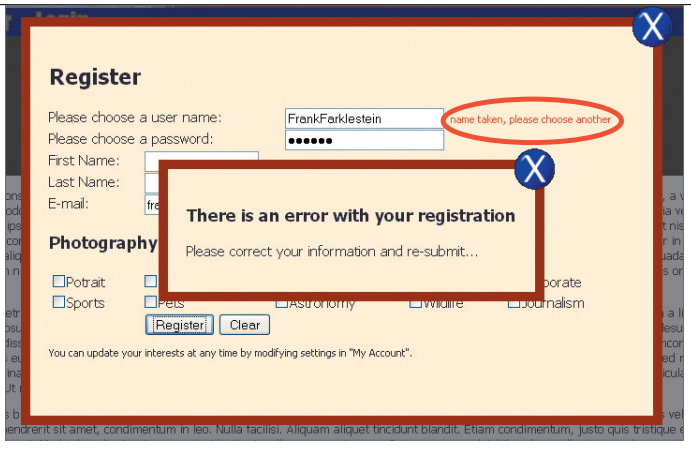


FIGURE 4.4 The modal prompt letting users know that they need to correct their registration information. In the background you can see that the user name is already taken; this must be changed.

3. Serialize the form data. The serializing creates a text string with standard URL-encoded notation. For most forms, this notation is in the form of `key=value` pairs:

```
var formData = $(this).serialize();
```

4. Now you can invoke the jQuery AJAX `post` method by providing the URL to post to and the serialized form data, and setting up a callback function:

```
$.post('inc/peRegister.php', formData, function(data) {
```

   The PHP code will return `0` if the query to add the user is successful. If not, it will return a higher number, indicating that the user could not be added.

5. Store the information returned by the AJAX function in the `mysqlErrorNum` variable:

```
var mysqlErrorNum = data;
```

   If an error is returned, you'll want to provide users with a prompt to let them know that they need to correct the information. The information is provided in a modal window as you have done before. **Figure 4.4** shows the modal window that you will set up next.

6. Test the value of the variable `mysqlErrorNum` to set up a conditional statement:

```
if(mysqlErrorNum > 0){
```

7. If `mysqlErrorNum` is greater than 0, append a modal window to the body of the Web page:

```
$('body').append('<div id="re"
→ class="errorModal"><h3>There is an error with
→ your registration</h3><p>Please correct your
→ information and re-submit...</div>');
```

8. Calculate and apply the margins for the new modal window just as you did before:

```
var modalMarginTop = ($('#re').height() + 60) / 2;
var modalMarginLeft = ($('#re').width() + 60) / 2;
$('#re').css({
    'margin-top' : -modalMarginTop,
    'margin-left' : -modalMarginLeft
});
```

9. Add the code that will fade in the modal window:

```
$('#re').fadeIn().prepend('<a href="#"
→ class="close_error"><img src=
→ "grfx/close_button.png" class="close_button"
→ title="Close Window" alt="Close" /></a>');
```

10. Provide a method to close the modal window containing the error warning:

```
$('a.close_error').live('click', function() {
    $('#re').fadeOut(function() {
        $('a.close_error, #re').remove();
    });
});
```

11. If no error was returned, fade out the registration window and clear the form:

```
        } else {
            $('#registerWindow, #modalShade').
            → fadeOut(function() {
                $('#registerForm input[input*="pe"]').val('');
            });
        }
```

12. Close the `post` method by providing the data type that you expect the PHP function to return:

```
    }, 'html');
});
```

LOGGING IN THE USER

The last step you need to do in the validation procedures is to give users a way to log in to their account.

The jQuery for the login function is nearly a duplicate of the registration, so I'll present it in its entirety:

```
$('#loginForm').submit(function(e){
    e.preventDefault();
    var formData = $(this).serialize();
    $.post('inc/peRegister.php', formData, function(data) {
        var returnValue = data;
        if(1 == returnValue){
            $('#loginWindow, #modalShade').fadeOut(function() {
                $('#loginForm input[name*="pe"]').val('');
                window.location = "4-2.php";
            });
```

```
        } else {
            $('body').append('<div id="li" class="errorModal">
            → <h3>There is an error with your login</h3><p>Please
            → try again...</div>');
            var modalMarginTop = ($('#li').height() + 60) / 2;
            var modalMarginLeft = ($('#li').width() + 60) / 2;
            $('#li').css({
                'margin-top' : -modalMarginTop,
                'margin-left' : -modalMarginLeft
            });
            $('#li').fadeIn().prepend('<a href="#"
            → class="close_error"><img src="grfx/close_button.png"
            → class="close_button" title="Close Window"
            → alt="Close" /></a>');
            $('a.close_error').live('click', function() {
                $('#li').fadeOut(function() {
                    $('a.close_error, #li').remove();
                });
            });
        }
    }, 'html');
});
```

FIGURE 4.5 The user's account page is displayed on a successful login.

If the login is successful, the browser loads *chap4/4-2.php* (**Figure 4.5**), the user's account page.

Now that you are comfortable with basic jQuery AJAX, let's move on to using the jQuery AJAX functions to update content in the browser.

USING **AJAX** TO **UPDATE CONTENT**

In many cases, you'll want to use various jQuery AJAX functions to update visible Web-site content. Some content updates may be based on the user information for the current user, other updates may be based on requests performed by any user, such as information based on a search performed by the Web-site visitor.

Let's look at some techniques for using jQuery's AJAX methods to update content.

GETTING CONTENT BASED ON THE CURRENT USER

If you have been developing Web sites even for the shortest period of time, you are likely aware of query strings in the URL. Unless Web-site developers are using methods to hide the strings, you may have seen something similar to this:

```
http://www.website.com/?user=me&date=today
```

Everything past the question mark is a query string that can be used in a GET request to the server. Each item is set up in a *name=value* pair, which can be easily parsed by scripting languages like jQuery and PHP.

NOTE: **Most forms utilize the POST method to request data from the server, but URLs are limited to the GET method. Most Web developers follow the rule of using GET when only retrieving data and using POST when sending data to the server that will invoke a change on the server.**

GET requests are not limited to the URL. You can use GET as a form method or in AJAX. jQuery provides a shorthand method call for making this kind of request to the server, and conveniently, it is called `get`.

1. Open *chap4/4-2.php* to set up a `get` function to retrieve the current user's pictures into the Web browser. Rather than storing the jQuery code in a different file and including it, let's use a slightly different technique that is very valuable when small jQuery scripts are used.

2. Locate the closing `</body>` tag. Just before that tag, the jQuery AJAX `get` method will be set up to retrieve the user's pictures. Begin by inserting the script tag:

   ```
   <script type="text/javascript">
   ```

3. Open the function by making sure that the document (the current Web page DOM information) is completely loaded:

```
$(document).ready(function() {
```

4. The first critical step in making sure that you get the right information from the database is to assign the value of the cookie set during login to a variable that can be used by jQuery. The information in the cookie is the user's name:

```
var cookieUser = '<?php echo $_COOKIE['photoex'];?>';
```

5. As stated earlier, the get method relies on *name=value* pairs to do its work properly. Make sure that the get request sends the cookie data to the server as a *name=value* pair:

```
$.get('inc/userPhoto.php', {photoUser: cookieUser},
→ function(data){
```

6. Load the information returned into the div with an id of myPhotos:

```
$('#myPhotos').html(data);
```

7. Close the get function with the data type that is expected to be returned from the PHP script. Once closed, set the closing </script> tag (the </body> tag is shown only for reference):

```
}, 'html');
});
</script>
</body>
```

8. Before you can get the photos from the database, you need to create the photo table. So, run the *pephoto.sql* file located in the *chap4/sql* folder of the code download. The SQL file will also insert default data for the photos located in the *chap4/photos* folder.

   In the PHP file *chap4/inc/userPhoto.php*, the SQL query uses the information contained in the photoUser variable:

```
$getImg = "SELECT `imgName`,`imgThumb` ";
$getImg .= "FROM `photoex`.`pephoto` ";
$getImg .= "WHERE `username` = '".$_GET['photoUser']."' ";
```

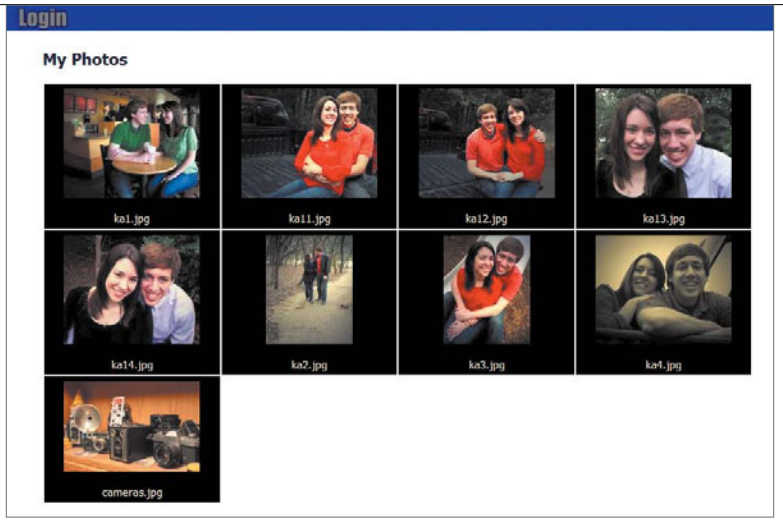


FIGURE 4.6 The user's photographs in tabular form.

The user's photographs are retrieved and placed into a table for viewing. The results are illustrated in **Figure 4.6**.

Combining user data with the `get` method is very effective for pages where data unique to the user must be displayed. What about content that is not unique to the user? The `get` method has a cool little brother called `load`.

LOADING CONTENT BASED ON REQUEST

Of the jQuery AJAX shorthand methods, `load` is the simplest and easiest method for retrieving information from the server. It is especially useful if you want to call on new information that does not need data passed to it like you would do with the `get` or `post` methods. The syntax for `load` is short and sweet as well:

```
$('a[href="writeNew"]').click(function(e){
    e.preventDefault();
    $('#newArticle').load('inc/userWrite.php');
});
```

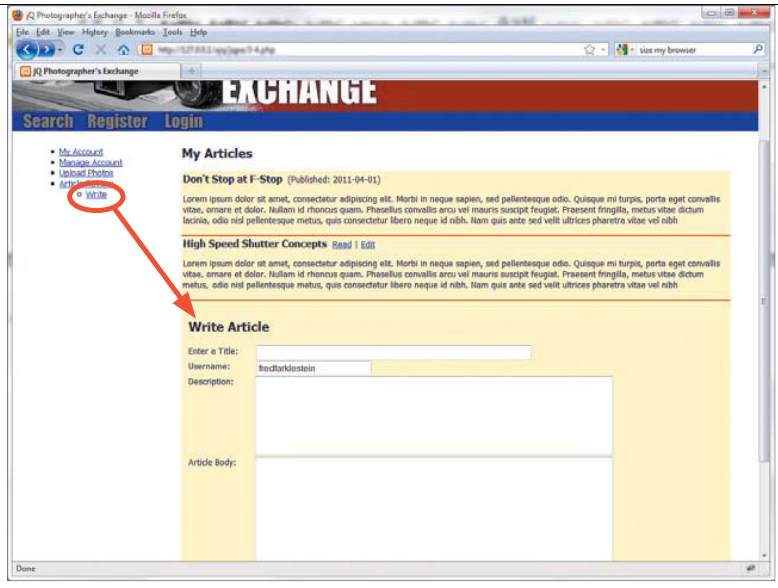


FIGURE 4.7 The form has been loaded into the page so that the user can write a new article.

Clicking on the Write link (**Figure 4.7**) invokes the `load` function, causing *chap4/inc/userWrite.php* to be loaded into the `div` with an `id` of `newArticle`.

There is one other really neat feature that `load` offers: You can use it to bring in just portions of other pages. For instance, to bring in a `div` with an `id` of `part1` from another page, the syntax is as follows:

```
$('#newArticle').load('inc/anotherPage.html #part1');
```

Having the option of loading page portions can give you a great deal of design and organizational flexibility.

> **NOTE:** In Chapter 6, "Creating Application Interfaces," you'll use an example in which several widgets will be contained in one file that will be called by `load` as needed to complete the interface.

Not every Web site can use every AJAX feature that jQuery offers, so you'll leave the Photographer's Exchange Web site behind at this point. You'll develop stand-alone examples to demonstrate some of the other features and events available in jQuery's AJAX library.

LOADING SCRIPTS DYNAMICALLY

There are some cases in which you will need to load JavaScript or jQuery scripts just for one-time use in your Web pages and applications. jQuery provides a special AJAX shorthand method to do just that, `getScript`.

For this example, you'll use the code contained in *chap3/dvdCollection*, which is a small personal Web site designed to be used as a catalog of all the DVD and Blu-ray Discs that you own.

From time to time, you'll want to know just how many DVD and Blu-ray Discs you have, but it isn't really necessary to load the script that performs the counts and displays the result every time you use the site. jQuery's `getScript` method is the perfect remedy for loading scripts that you'll use infrequently.

1. Set up a script called *dvdcount.js* and place it in the *inc* directory of the DVD collection site. This is the script that `getScript` will load when called upon to do so.

2. Include the document ready functionality:

   ```
   $(document).ready(function(){
   ```

3. Each movie is contained in a `div` with a class of `dvd`. Assign the count of those `div`'s to the variable `totalCount`:

   ```
   var totalCount = $('.dvd').length;
   ```

4. Use jQuery's `:contains` selector to help count the types of discs in the collection. The `:contains` selector is very handy for finding elements containing a specific string. Here it is used to find the text "DVD" or "Blu-ray" in the `h3` element:

   ```
   var dvdCount = $('h3:contains("DVD")').length;
   var brCount = $('h3:contains("Blu-ray")').length;
   ```

5. Set up the modal window to show the user the information. This is the same technique used in Chapter 2 and Chapter 3, so I won't cover each step in detail:

   ```
   var movieModal = '<div class="movieModal">Total Movies:
   → '+totalCount+'<br />DVD: '+dvdCount+'<br />Blu-ray:
   → '+brCount+'</div>';
   ```

```
        $('body').append(movieModal);
        var modalMarginTop = ($('.movieModal').height() + 40) / 2;
        var modalMarginLeft = ($('.movieModal').width() + 40) / 2;
        $('.movieModal').css({
            'margin-top' : -modalMarginTop,
            'margin-left' : -modalMarginLeft
        });
```

The modal will only pop up for a moment before fading out:

```
        $('.movieModal').fadeIn('slow', function(){
            $(this).fadeOut(2500, function() {
                $(this).remove();
            });
        });
    });
```

The main page for the DVD catalog site is *chap4/dvdCollection/4-5.php*. Let's take a moment to set it up.

1. Enter the information for the header:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>DVD Collection Catalog</title>
        <link rel="stylesheet" href="css/dvd.css"
        → type="text/css" />
```

2. Include the jQuery file so that all of the interactions will run properly:

```
        <script type="text/javascript"
        → src="inc/jquery-1.5.min.js"></script>
    </head>
```

3. Set up the body next:

```
    <body>
    <h2>DVD Collection Catalog</h2>
    <div class="menuContainer">
```

4. Set up the menu section carefully, because you'll use these elements to call other scripts:

```
        <ul class="menu">
            <li id="add">Add</li>
            <li id="summary">Summary</li>
        </ul>
    </div>
    <br />
```

5. Set up the div that will house the content of the page:

```
    <div class="content"></div>
```

6. Create the section containing the jQuery scripts you'll use to load information into the page along with the function that loads *chap4/dvdCollection/inc/getdvd.php*. The PHP is called by the jQuery `load` method to get the information about the DVD collection:

```
    <script type="text/javascript">
        $(document).ready(function(){
            $('.content').load('inc/getdvd.php');
```

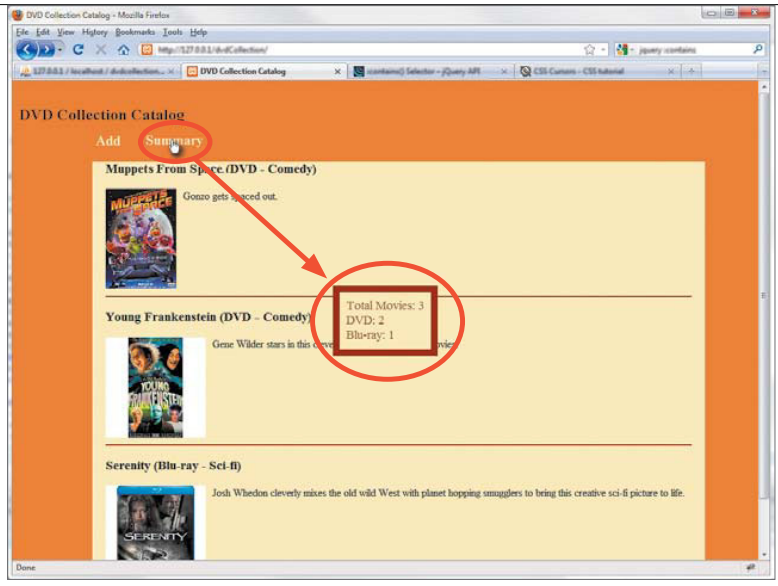


FIGURE 4.8 Clicking on the Summary element loads and runs the *dvdcount.js* script.

7. Bind the `click` method to the list item with an `id` of `summary`. This will call `getScript` to run the jQuery script created earlier, *dvdcount.js*:

```
            $('#summary').click(function() {
            $.getScript('inc/dvdcount.js');
        });
    });
</script>
```

8. Close out the HTML:

```
    </body>
</html>
```

Clicking the Summary element on the Web page causes the *dvdcount.js* script to be loaded and run, showing the modal window complete with counts (**Figure 4.8**). The modal window then slowly fades away.

You will find many cases where loading and running scripts on the fly will enhance your Web sites and applications.

Next, you'll turn your attention to many of jQuery's AJAX extras and learn how to apply them practically.

USING JQUERY'S AJAX EXTRAS

In addition to the shorthand methods, jQuery provides many useful methods and helpers to give you ways to use AJAX efficiently. These methods range from low-level interfaces to global event handlers, all of which, when applied properly, will make your programs and Web sites more effective.

Let's look at these extras, starting with the low-level interfaces.

WORKING WITH LOW-LEVEL INTERFACES

jQuery's low-level AJAX interfaces provide the most detailed approach to AJAX functions. This kind of detail makes the low-level interfaces quite flexible but introduces additional complexity due to all of the options available.

One way to combat the complexity of having an extensive choice of options is to use a method to set up options that do not change frequently. Take a look at the simplest of the low-level interfaces, `ajaxSetup`:

```
$.ajaxSetup({
    url: ajaxProcessing.php,
    type: 'POST'
});
```

The `ajaxSetup` method allows you to provide options that will be used with every AJAX request. You can set all of the AJAX options available (over 25 of them!) using `ajaxSetup`. This is very convenient if you need to make repeated AJAX requests to the same URL or use the same password each time you make a request. In many cases, developers will put all of their server-side AJAX handlers in the same file on the server. Using `ajaxSetup` shortens their AJAX calls, including the shorthand methods. Given the current example of `ajaxSetup`, your `post` method could be configured like this:

```
$.post({ data: formData });
```

The only thing you need to supply to the post function is the data to be handled by *ajaxProcessing.php*. One advantage of using the ajaxSetup method is that you can override any of the ajaxSetup options in the individual AJAX calls that you make.

The low-level interface that you will see in use most is the straight ajax method. It is the function that is wrapped by the shorthand methods and is at the very heart of all of jQuery's AJAX calls. The ajax method is capable of accepting all of the options that can be used with jQuery's AJAX requests. Perhaps the best way to understand the low-level AJAX method is to compare it to one of the shorthand methods you used earlier. Here is the post method that you used to check to make sure the user name was available:

```
$.post('inc/peRegister.php', {
    formName: 'register',
    penewuser: newName
}, function(data){
    var usernameCount = data;
    if(1 == usernameCount){
        $('#penewuser').next('.error').css('display', 'inline');
    } else {
        $('#penewuser').next('.error').css('display', 'none');
    }
}, 'html');
```

Here is the same request using jQuery's low-level ajax method:

```
$.ajax({
    type: 'POST',
    url: 'inc/peRegister.php',
    data: 'formName=register&penewuser='+newName+'',
    success: function(data){
        var usernameCount = data;
        if(1 == usernameCount){
```

```
            $('#penewuser').next('.error').css('display', 'inline');
        } else {
            $('#penewuser').next('.error').css('display', 'none');
        }
    },
    dataType: 'html'
});
```

The differences are fairly obvious, such as declaring the method that AJAX should use to convey the information to the server (`type: 'POST'`), specifying the way that raw data is formatted (`data: 'formName=register&penewuser='+newName+'',`) and ensuring that the success method is implicitly defined (`success: function(data){…`).

Take a tour of jQuery's `ajax` API at http://api.jquery.com/jQuery.ajax to see all of the options available for use with this method.

Now that you can send information to the server and receive information back from your server-side processes, you need to make sure that your users are informed that an AJAX action is taking place. jQuery provides several helper functions that make it easy for you to do just that.

TRIGGERING EVENTS BEFORE AND AFTER THE AJAX CALL

In many cases, your jQuery AJAX functions will happen so quickly that users may not even know that their actions achieved the desired result. In other cases, the AJAX process may be lengthy and require that users wait for results. jQuery provides four methods that you can use to keep users informed: `ajaxStart`, `ajaxSend`, `ajaxComplete`, and `ajaxStop`.

It is important to understand that there is an order to these four functions. You can call any number of AJAX processes during any given event. For this reason, you may want to know not only when the first AJAX function starts, but also when each subsequent AJAX method gets called and completes. Then you may want to register that all of the AJAX calls have completed. If you imagine jQuery AJAX events as a stack of items as in **Figure 4.9**, you'll see how the jQuery AJAX engine defines the order of the events and their calls.

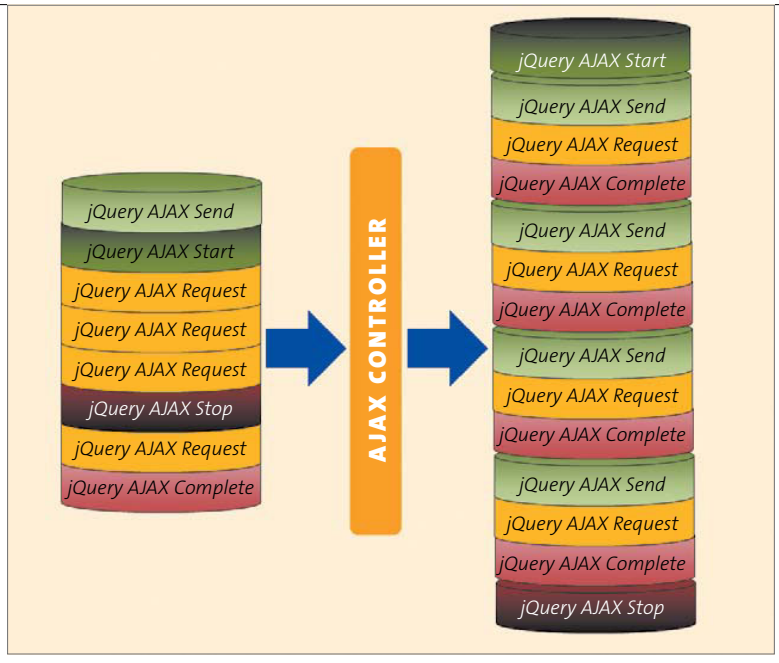


FIGURE 4.9 The initial jQuery events are stacked up by the developer and then ordered and processed by jQuery's AJAX engine.

Let's take a close look at how to use the `ajaxStart` and `ajaxStop` methods by giving users a visual queue during a data- and file-submission event in the DVD Collection Catalog.

1. Open *chap4/4-6.php*.

   In *4-6.php* you will see a form (**Figure 4.10** on the next page) that accepts user input and provides a method for uploading a file. This combination is not unusual, but it will require that you pay careful attention when writing the PHP and jQuery to handle the data transfer and file upload.

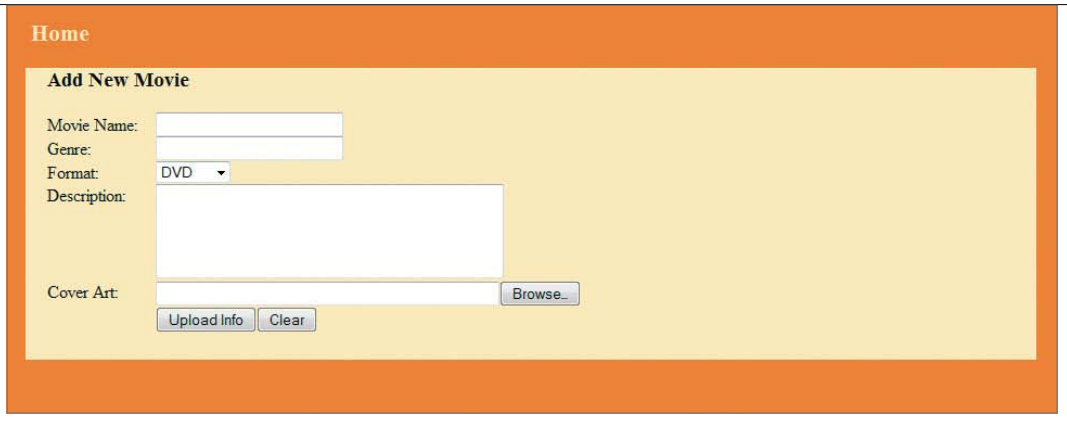


FIGURE 4.10 The form that users will fill out to add movies to their personal database.

Two PHP scripts will handle the data supplied in the form: one for the movie cover art upload (not really AJAX, remember?) and one for the data input into the form.

2. Create a file called *chap4/dvdCollection/inc/**dvdcover.php*** to set up the image upload first.

3. Set up the path for the cover art:

```
$coverPath = "../cover_art/";
```

4. Make sure that the file is submitted properly and has no errors:

```
if ($_FILES["movieCover"]["error"] == UPLOAD_ERR_OK) {
```

5. Set up the variables to hold the information about the uploaded file (this is the same technique that you used for file uploads in Chapter 3):

```
$tmpName = $_FILES["movieCover"]["tmp_name"];
$coverName = $_FILES["movieCover"]["name"];
```

6. Create the regular expression used to check the file extension of the uploaded file:

```
$regexFileExt = "/\.(jpg|jpeg|png)$/i";
```

7. Test the file extension to see if it matches one allowed by the regular expression:

```
if(preg_match($regexFileExt, $coverName)){
```

8. Check the file again by making sure it really is the right kind of file according to its first few bytes:

```
        $arrEXIFType = array(IMAGETYPE_JPEG, IMAGETYPE_PNG);
        if(in_array(exif_imagetype($tmpName), $arrEXIFType)){
```

9. Set up the file's new name and path, and place them into the variable `$newCover`:

```
            $newCover = $coverPath.$coverName;
```

10. Move the properly named file to its permanent directory:

```
            move_uploaded_file($tmpName, $newCover);
        }
    }
}
```

Now that you've completed the PHP script for the file upload, you can create the PHP script that will be called by the jQuery AJAX `post` method to update the database.

1. Create a file called ***postdvd.php*** and store it in the *chap4/dvdCollection/inc* folder.

   Only two actions are contained in *postdvd.php*: one to connect to the database and one to run the query that will perform the database update.

2. Set up the database connection first (be sure to use the user name and password that you have set up for your database):

```
if(!$dbc = mysql_connect('localhost', 'username', 'password')){
    echo mysql_error() . "\n";
    exit();
}
```

3. Introduce a little sleep timer to slow down the process. This will allow the animated loading graphic to be displayed by `ajaxStart` in the jQuery function that will be created (typically, the database operation is very fast—so fast that the user may not realize that something has occurred.):

```
sleep(2);
```

4. Create the SQL query that will accept the values from the AJAX `post` method to update the database with:

```
$insertMovie = "INSERT INTO `dvdcollection`.`dvd` ";
$insertMovie .= "(`name`,`genre`,`format`,`description`,
→ `cover`) ";
$insertMovie .= "VALUES(";
$insertMovie .= "'".$_POST['movieName']."',";
$insertMovie .= "'".$_POST['movieGenre']."',";
$insertMovie .= "'".$_POST['movieFormat']."',";
$insertMovie .= "'".$_POST['movieDescription']."',";
$insertMovie .= "'cover_art/".$_POST['movieCover']."' ";
$insertMovie .= ")";
```

NOTE: Make sure that you run the SQL chap4/dvdCollection/sql/create_collection_table.sql script in your database platform to set up and populate the table for the DVD collection.

5. Call the `mysql_query` function to run the SQL query:

```
if(!($movieInfo = mysql_query($insertMovie, $dbc))){
    echo mysql_error();
    echo mysql_errno();
    exit();
}
```

With the PHP scripts complete, you can now turn your attention to the jQuery functions. All of the jQuery functions will be placed into the file *inc/movieUp.js*.

1. Start the file by defining the `ajaxStart` method:

   ```
   $('body').ajaxStart(function(){
   ```

   The `ajaxStart` function will be called as soon as an AJAX request is made. The method can be bound to any element available in the DOM and is bound to the `body` element for use here. You can define any processes that you want within the `ajaxStart` method.

2. For this file and data upload, create a modal pop-up window to give the users a visual clue that something is occurring:

   ```
       var waitingModal = '<div class="waitingModal">
       → <img src="grfx/loading.gif" border="0" /></div>';
       $('body').append(waitingModal);
       var modalMarginTop = ($('.waitingModal').height() + 40) / 2;
       var modalMarginLeft = ($('.waitingModal').width() + 40) / 2;
       $('.waitingModal').css({
           'margin-top' : -modalMarginTop,
           'margin-left' : -modalMarginLeft
       });
       $('.waitingModal').fadeIn('slow');
   });
   ```

   The technique used to create the modal window is no different than what you have used previously in the book.

3. Bind the `ajaxStop` method to the `body` element (remember that methods like `ajaxStart` and `ajaxStop` can be bound to any element). When the AJAX request is complete, you'll want to clear the form and remove the modal from view so that the user knows the process is finished:

   ```
   $('body').ajaxStop(function(){
   ```

4. Clear the form elements so that the user can use the form to add another movie. Just like using `ajaxStart`, you can define any process within the `ajaxStop` function:

```
$('#addMovie input[name*="movie"]').val('');
$('#addMovie textarea').val('');
```

Be very specific with your jQuery selectors when choosing which form elements to clear. For example, using just `$('#addMovie input')` will also clear the form's buttons, and that would confuse the user.

5. Fade away the modal indicator and remove it from the DOM. This is the last part of the process defined in the `ajaxStop` method:

```
    $('.waitingModal').fadeOut('slow', function(){
        $(this).remove();
    });
});
```

6. Begin the form handler by binding the form `addMovie` to the `submit` method:

```
$('#addMovie').submit(function(){
```

7. Upload the image using the `iframe` method that was defined in Chapter 3:

```
    var iframeName = ('iframeUpload');
    var iframeTemp = $('<iframe name="'+iframeName+'"
    → src="about:blank" />');
    iframeTemp.css('display', 'none');
    $('body').append(iframeTemp);
    $(this).attr({
        action: 'inc/dvdcover.php',
        method: 'post',
        enctype: 'multipart/form-data',
        encoding: 'multipart/form-data',
        target: iframeName
    });
```

8. Once the image upload is complete, remove the `iframe` from the DOM:

```
setTimeout(function(){
    iframeTemp.remove();
}, 1000);
```

FIGURE 4.11 The `ajaxStart` method has called the waiting indicator.

9. Prepare the data to be used in the `post` method. Because information in a `textarea` cannot be serialized with normal jQuery methods, create a text string that sets up the `textarea` value as if it were serialized by making the information a *name=value* pair:

```
var coverData = '&movieCover=' +
→ $('input[name="movieCover"]').val();
```

10. Serialize the remainder of the form data:

```
var formData = $(this).serialize();
```

11. Once the form data has been processed by the `serialize` function, concatenate the two strings together in the `uploadData` variable:

```
var uploadData = formData + coverData;
```

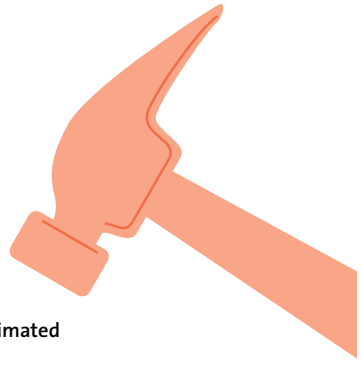
12. Call the jQuery AJAX shorthand method `post` to upload the data:

```
$.post('inc/postdvd.php', uploadData);
});
```

When the movie data form is submitted, the jQuery AJAX engine will see that there is a `post` occurring during the process, triggering the `ajaxStart` method. **Figure 4.11** shows the modal loading indicator called by `ajaxStart`.

Once the post process has completed, the `ajaxStop` method is triggered, causing the modal waiting indicator to fade out.

Now that you have learned to handle AJAX calls and the data they return, you need to learn how to handle one of the Web's fastest-growing data types, JSON.



TIP: If you need animated graphics to indicate to your users that something is occurring in the background, check out www.ajaxload.info. There you can generate several different animated graphics in a wide array of colors.

USING JSON

JSON (JavaScript Object Notation) has become a popular and lightweight way to transmit data packages for various uses over the Internet. In many ways, JSON is more popular than XML for delivering data quickly and efficiently. JSON data can be easily used with the jQuery AJAX shorthand method especially designed to handle the JSON data type, `getJSON`.

So what exactly is JSON?

To understand JSON, you need a little lesson in JavaScript's object literal notation. *Object literal notation* is an explicit way of creating an object and is the most robust way of setting up a JavaScript object. Here is an example:

```
var person = {
    name: "Jay",
    occupation: "developer",
    stats: ["blonde", "blue", "fair"],
    walk: function (){alert(this.name+ 'is walking');}
};
```

The person object has been literally defined as name: value pairs, including a nested array (stats) and a method to make the object walk. It is a very tidy way to describe an object.

The following commands interact with the person object:

```
person.walk(); //alerts 'Jay is walking'
alert(person.stats[1]); // alerts 'blue'
```

JSON is a subset of the object literal notation, essentially the *name: value* pairs that describe an object. A JSON array can contain multiple objects. The key to being successful with JSON is making sure that it is well-formed. JSON must have matching numbers of opening and closing brackets and curly braces (the braces must be in the correct order); the names and values in the *name : value* pairs must be quoted properly; and commas must separate each *name: value* pair.

To illustrate this, look at the JSON for the person object:

```
var myJSONobject = {"person":[{
    "name":"Jay",
    "occupation":"developer",
    "stats":[{
        "hair":"blonde",
        "eyes":"blue",
        "skin":"fair"
    }]
  }]
};
```

It's important to note that the JSON object does not contain any methods or functions that can be executed. JSON specifically excludes these from the notation because JSON is only meant to be a vehicle for transmitting data.

SETTING UP A JSON REQUEST

Twitter has undoubtedly become one of the most popular social media outlets since the dawn of the Internet. Twitter has made an API available for those who want to extend the use of Twitter to their own Web pages and applications. One of the most popular uses of the Twitter API is to include recent tweets in personal Web sites and blogs.

Taking advantage of the API can be as simple or as complex as you want it to be. Let's build a simple widget to obtain your last ten tweets for inclusion in a Web page.

The tweet data is returned from Twitter in the JSONP format. JSONP is known as "JSON with Padding." Under normal circumstances, you cannot make AJAX requests outside of the domain the request originates from (**Figure 4.12** on the next page). JSONP relies on a JavaScript quirk: `<script>` elements are allowed to make those cross-domain requests.

FIGURE 4.12 The only way you can make a cross-domain request is with JSONP.

To make this work, the JSON must be returned in a function. Using the JSON object created earlier, the JSONP would look like this:

```
myJSONfunction({"person":[{"name":"Jay", "occupation":"developer",
→  "stats":[{"hair":"blonde","eyes":"blue","skin":"fair"}]}]});
```

If it looks like gibberish to you now, don't worry; as you walk through the function being built to get JSON data from Twitter, it will become much clearer.

Let's build the entire file, including CSS, from scratch.

1. Create a file called ***4-7.php*** in the *chap4* folder.

2. Set up the DOCTYPE and include the basic head, title, and character set declarations:

```
<!DOCTYPE html>
<html>
    <head>
    <meta http-equiv="Content-Type" content="text/html;
    → charset=utf-8" />
    <title>Twitter Widget</title>
```

3. Provide a reference to the jQuery source that you will be using. Make sure that the path is correct; in this case the path is *inc/jquery-1.5.2.min.js*:

```
<script type="text/javascript"
→ src="inc/jquery-1.5.min.js"></script>
```

4. Create the style information for the Twitter widget:

```
<style type="text/css">
    body {
        background-color: #FFFFCC;
    }
    #tw {
        position: relative;
        width: 350px;
        left: 50%;
        margin-left: -175px;
    }
    .tweet {
        font-family: "Lucida Grande","Arial Unicode MS",
        → sans-serif;
        width: 350px;
```

```
            background-color: #99FFCC;
            padding: 5px;
            border-right: 2px solid #66CC99;
            border-bottom: 3px solid #66CC99;
            margin-bottom: 2px;
        }
    </style>
```

5. Close out the head section of the page:

```
    </head>
```

The body section for the widget is very simple: Add a `div` with an `id` of `tw` to which the tweets will be appended:

```
<body>
    <div id="tw"></div>
```

The jQuery script to get the tweets is very short but requires that you pay attention to detail. You will make the names and hash tags clickable so that they have the same functionality they have on the Twitter Web site. Any links included in a tweet will also be clickable, opening a new browser window to show the information.

1. Start the jQuery function by opening a script tag and inserting the document-ready function:

```
        <script type="text/javascript">
        $(document).ready(function() {
```

2. Create the URL to access Twitter and store the URL in the variable `twitterURL`:

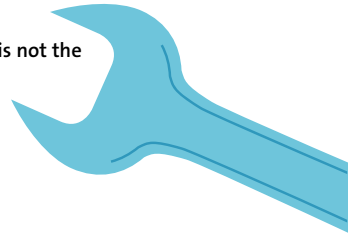
```
            var twitterURL ='http://twitter.com/statuses/
            → user_timeline.json?screen_name=
            → YOUR_TWITTER_USER_NAME&count=10&callback=?';
```

   Be sure to replace `YOUR_TWITTER_USER_NAME` with your actual Twitter user name. It is very important to make sure that the URL is formatted with the query string (*name=value* pairs) that will be used by `getJSON` during

the request. Send three options to Twitter: your Twitter `screen_name`, the `count` of the number of tweets to return, and most important, the `callback`. It is the `callback` option that lets Twitter know that you expect the return data to be JSONP.

3. Once the URL is formed, open the `getJSON` request method by sending the URL and defining the `getJSON` callback option:

```
$.getJSON(twitterURL, function(data){
```



NOTE: **The callback option for the query string is not the same as the callback for the *getJSON* request.**

4. The JSONP has been returned from Twitter at this point. Set up a loop through the data contained in the function. Treat the data as members of an array called `item`:

```
$.each(data, function(i, item){
```

5. Contain the tweet in a *name: value* pair with the *name* of `text`. Assign this item to the variable `tweetText`:

```
var tweetText = item.text;
```

6. Use regular expressions to locate URLs, @ tags, and hash(#) tags in the tweet so that you can give each the proper treatment. Look for URL's first:

```
tweetText = tweetText.replace
→ (/http:\/\/\S+/g, '<a href="$&"
→ target="_blank">$&</a>');
```

The regular expression `/http:\/\/\S+/g` matches text beginning with `http://` and ending in a space, which would typically indicate a URL. The `/g` (global) says to match all URLs in the string contained in `tweetText`. The URLs are turned into links by replacing the URL with an anchor tag containing the URL as both the `href` and the text of the link. In JavaScript the `$&` property contains the last item matched by a regular expression. Because the URL was the last item matched, it can be replaced into an anchor tag by using the `$&` property.

7. Twitter prefixes user names with the @ symbol. So, search `tweetText` for words beginning with the @ symbol:

```
tweetText = tweetText.replace(/(@)(\w+)/g,
→ ' $1<a href="http://twitter.com/$2"
→ target="_blank">$2</a>');
```

   Here, the regular expression `/(@)(\w+)/g` indicates that all words beginning with the @ symbol are replaced by the appropriate anchor tag to open a browser window for users' tweets. The `$1` and `$2` contain the information matched in each parenthesis, which is used to include those matches in the replacement text.

8. Turn your attention to the hash tags now and use a technique similar to the one you used for replacing the @ symbol:

```
tweetText = tweetText.replace(/(#)(\w+)/g,
→ ' $1<a href="http://search.twitter.com/
→ search?q=%23$2" target="_blank">$2
→ </a>');
```

9. Once the `tweetText` has been completely manipulated to insert all of the anchor tags, place it into a `div`. Then append the new `div` to the existing `div (id="tw")` that was set up as part of the original content for the page:

```
$("#tw").append('<div class="tweet">
→ '+tweetText+'</div>');
```

10. Close out the jQuery function and HTML tags for the page:

```
            });
        });
    });
    </script>
</body>
</html>
```

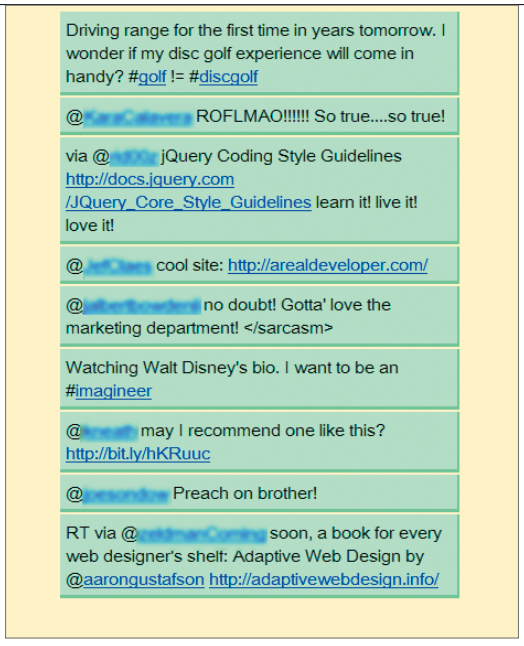


FIGURE 4.13 The Twitter widget retrieves the last few posts that you made.

11. Upload the page to a server, and load the page into a browser. You should achieve the results that you see in **Figure 4.13**.

With all of the data traveling back and forth between clients and servers, including servers not under your control, it is only natural to be concerned about the security of the information that you and your Web-site visitors send in AJAX requests. Let's address those concerns next.

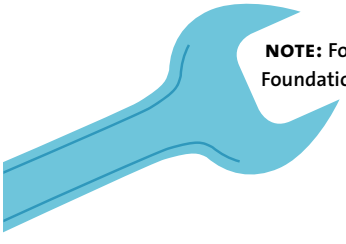
SECURING **AJAX REQUESTS**

One of the vexing problems with Web sites and applications is that users will either inadvertently or purposely submit data through your Web site that can cause harm to your databases and servers. It is important that you take as many steps as possible to guard against the input and transmission of bad or malformed data.

Several of these steps have been covered already, including using regular expressions to guide the user to input the right kind of data and making sure that cookies are set uniquely for each Web visitor. As an older, and much wiser, mentor said to me, "Locking the gate in this way only keeps the honest people from climbing the fence."

Even with regular expressions in place for form fields, you cannot stop the transmission of the data because the form can still be submitted. So, what are some of the measures you can take to prevent users from submitting potentially harmful data?

- Prevent form submission by "graying" out the Submit button on forms until all of the regular expression rules for each form field have been met.
- Use cookies to uniquely identify the user (more precisely, the user's computer) based on registration information and check cookie data against a database during transmission of user-supplied data.
- Clean user-supplied data when it arrives at the back-end process to make sure the data doesn't contain harmful statements or characters.
- Transmit the data over a secure connection (HTTPS [HyperText Transfer Protocol Secure]) to prevent outsiders from "sniffing" information traveling from and to the Web browser.



NOTE: For more information on HTTPS, visit the Electronic Frontier Foundation's Web site at www.eff.org/https-everywhere.

These techniques should be used in conjunction with each other to present the safest experience for the user and the Web-site owner. Let's walk through some of these techniques.

PREVENTING FORM SUBMISSION

Let's return to the Photographer's Exchange Web site and make some changes to the HTML file containing the registration form as well as the jQuery script that supports the form.

1. Open *chap4/4-2.php* and locate the section of the script where jQuery scripts are included. You'll find these include declarations between the head tags.

2. Change the following highlighted line to point to the updated *jqpe.js* file:

```
<script type="text/javascript"
→ src="inc/jquery-1.5.min.js"></script>

<script type="text/javascript"
→ src="inc/jquery.ez-bg-resize.js"></script>

<script type="text/javascript"
→ src="inc/spritenav.js"></script>

<script type="text/javascript"
→ src="inc/carousel.js"></script>

<script type="text/javascript"
→ src="inc/jqpe.js"></script>

<script type="text/javascript"
→ src="inc/peAjax.js"></script>
```

   After the change, the line will look like this:

```
<script type="text/javascript"
→ src="inc/jqpeUpdated.js"></script>
```

3. Save the file as ***chap4/4-8.php***.

4. Open *chap4/inc/jqpe.js* and save it as ***chap4/inc/jqpeUpdated.js***. Add the code for the error count function. Start by initializing the `$submitErrors` variable:

```
var submitErrors = 0;
```

5. Declare a function called `errorCount`:

```
function errorCount(errors) {
```

6. Set the argument variable `errors` to be equal to the `submitErrors` variable:

```
errors = submitErrors;
```

7. If the error count is zero, you want to enable the submit button. So, remove the `disabled` attribute from the button. Use the jQuery attribute selectors to select the proper button:

```
if(0 == errors){
    $('input[type="submit"][value="Register"]').
    → removeAttr('disabled');
```

8. If the error count is not zero, the submit button will be disabled. Use the same selector syntax and add the `disabled` attribute to the button:

```
} else {
    $('input[type="submit"][value="Register"]').
    → attr('disabled','disabled');
}
```

9. Close out the function :

```
}
```

Once the function is in place, you'll need to make some changes to the password and email validation functions that were created previously.

1. In *jqpeUpdated.js* locate the password validation function that begins with the comment `/*make sure password is not blank */`. Insert the two new lines of code highlighted here:

```
/* make sure that password is not blank */
    $(function() {
        var passwordLength = $('#penewpass').val().length;
        if(passwordLength == 0){
            $('#penewpass').next('.error').css('display',
            → 'inline');
            errorCount(submitErrors++);
            $('#penewpass').change(function() {
```

```
            $(this).next('.error').css('display', 'none');
            errorCount(submitErrors--);
        });
    }
});
```

If the password is blank (having a length of zero), the `errorCount` function is called and the `submitErrors` variable is incremented by a count of one.

```
errorCount(submitErrors++);
```

After a password has been entered, the error is cleared and the error count can be reduced by decrementing `submitErrors`:

```
errorCount(submitErrors--);
```

2. Locate the email validation function. It begins with the comment `/* validate e-mail address in register form */`. Add the same calls to the `errorCount` function where indicated by the following highlights:

```
/* validate e-mail address in register form */
    $(function(){
        var emailLength = $('#email').val().length;
        if(emailLength == 0){
            $('#email').next('.error').css('display',
            → 'inline');
            errorCount(submitErrors++);
            $('#email').change(function() {
            var regexEmail = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+
            → \.[a-zA-Z]{2,4}$/;
            var inputEmail = $(this).val();
            var resultEmail = regexEmail.test(inputEmail);
            if(resultEmail){
                $(this).next('.error').css('display', 'none');
                errorCount(submitErrors--);
```

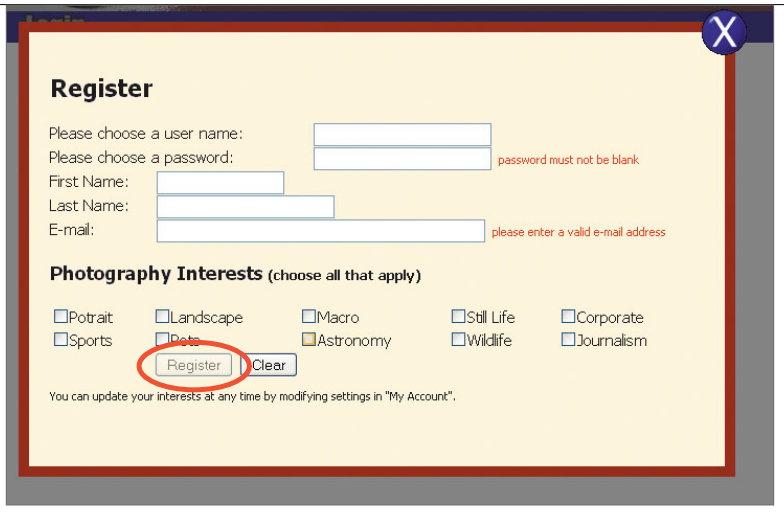


FIGURE 4.14 The Register button is grayed out. It is not available to the user until all errors are cleared.

```
                    }
                });
            }
        });
```

When the page first loads, `submitErrors` gets incremented twice—once by each of the validation functions. The total error count prior to the form being filled out is two. Because the `submitErrors` has a value of two, the submit button is disabled, as illustrated in **Figure 4.14**.

As each function is cleared of its error, the `submitErrors` variable is decremented until it finally attains a value of zero. When the value of `submitErrors` is zero, the `errorCount` function removes the disabled attribute from the submit button and the form can be submitted normally.

This technique can be applied to any number of form fields that you need to validate, but it really isn't enough to prevent malicious users from trying to hack your site. Let's take a look at another technique you can add to your Web-site application model, giving each user cookies.

USING COOKIES TO IDENTIFY USERS

Giving users cookies sounds very pleasant. But it really means that you want to identify users to make sure they are allowed to use the forms and data on your Web site. What you don't want to do is put sensitive information into cookies. Cookies can be stolen, read, and used.

Personally, I'm not a big fan of "remember me cookies" because the longer it takes a cookie to expire, the longer the potentially malicious user has to grab and use information in the cookie. I'd rather cookies expire when the user closes the browser. This would reduce the chance that someone could log in to the user's computer and visit the same sites to gain information or copy the cookies to another location.

What should you store in the cookie? One technique that you can employ that is very effective is storing a unique token in the cookie that can be matched to the user during the user's current session. Let's modify the Photographer's Exchange login process to store a token in the user's database record. The token will be changed each time the user logs in to the site, and you will use the token to retrieve other data about the user as needed.

1. Open *chap4/inc/peRegister.php* and locate the section that starts with the comment `/* if the login is good */`. You will insert new code to create and save the token into the newly created database column.

2. The first line that you need to add creates a unique value to tokenize. Concatenate the user name contained in `$_POST['pename']` with a time stamp from PHP's `time` function. PHP's `time` function returns the time in seconds since January 1, 1970. Store that in the variable `$tokenValue`, as shown in the following highlighted line:

```
/* if the login is good */
if(1 == $loginCount){
    if(isset($_POST['remember'])){
    $tokenValue = $_POST['pename'].time("now");
```

3. Modify the information to be stored in `$peCookieValue` by hashing the `$tokenValue` with an MD5 (Message Digest Algorithm) hash:

```
    $peCookieValue = hash('md5', $tokenValue);
    $peCookieExpire = time()+(60*60*24*365);
```

```
$domain = ($_SERVER['HTTP_HOST'] != 'localhost') ?
→ $_SERVER['HTTP_HOST'] : false;

setcookie('photoex', $peCookieValue, $peCookieExpire, '/',
→ $domain, false);

echo $loginCount;

} else {
```

The MD5 hash algorithm is a cryptographic hash that takes a string and converts it to a 32-bit hexadecimal number. The hexadecimal number is typically very unique and is made more so here by the use of the `time` function combined with the user's name.

4. Make the same modifications in the section of the code where no "remember me" value is set:

```
$tokenValue = $_POST['pename'].time("now");

$peCookieValue = hash('md5', $tokenValue);

$peCookieExpire = 0;

$domain = ($_SERVER['HTTP_HOST'] != 'localhost') ?
→ $_SERVER['HTTP_HOST'] : false;

setcookie('photoex', $peCookieValue, $peCookieExpire, '/',
→ $domain, false);

echo $loginCount;
```

5. Add the code that will update the database with the new value:

```
$updateUser = "UPDATE `photoex`.`peuser` ";

$updateUser .= "SET `token` = '".$peCookieValue."' ";

$updateUser .= "WHERE `username` = '".$_POST['pename']."' ";

if(!($updateData = mysql_query($updateUser, $dbc))){

    echo mysql_errno();

    exit();

}
```

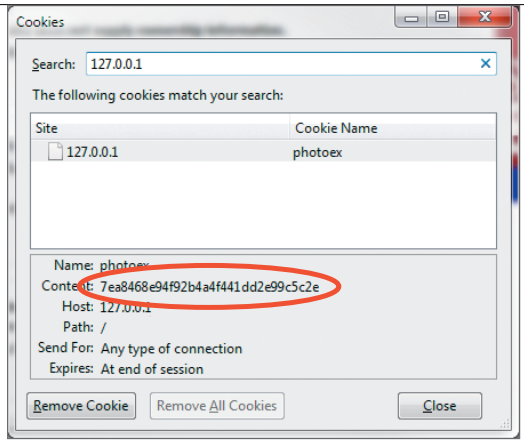


FIGURE 4.15 The content of the cookie is circled and would-be cookie thieves are foiled!

6. Open *chap4/4-8.php* and log in to the site with a known good user name and password. The cookie will be set with the token, and the token information will be set in the database. You can use your browser's built-in cookie viewer (for **Figure 4.15**, I used Tools > Page Info > Security > View Cookies in the Firefox browser) to examine the value stored in the cookie.

Using the value of the token, you can retrieve needed information about the user so that the data can be entered into forms or the appropriate photographs can be displayed. Next, let's take a look at cleaning up user-supplied data.

CLEANSING USER-SUPPLIED DATA

One additional step that you can take to make sure that user-supplied data is safe once it reaches the server is to use your client-side scripting language to ensure that the data is handled safely and securely.

A less than savory visitor may visit your site and copy your visible Web pages and functions. Once copied, modifications can be made to your jQuery scripts to remove some of the controls (regular expressions for instance) that you have placed around data. Your first line of defense against that is to replicate those controls in your server-side scripts.

1. Using email validations as an example, open *peRegister.php* (*chap4/inc/peRegister.php*) to modify it.

2. Locate the section of the code that begins with the comment `/* if the registration form has a valid username & password insert the data */` and supply this regular expression:

   ```
   $regexEmail = '/^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$/';
   ```

   This is the same regular expression used in the jQuery function to validate email addresses into the registration form.

3. Test the value posted against the regular expression with PHP's `preg_match` function:

   ```
   preg_match($regexEmail, $_POST['email'], $match);
   ```

4. The test result, a `1` if there is a match or a `0` if there isn't a match, is placed into the variable `$match` that is declared in the `preg_match function`. Use this result to modify the `$_POST['email']` variable:

   ```
   if(1 == $match){
       $_POST['email'] = $_POST['email'];
   } else {
       $_POST['email'] = 'E-MAIL ADDRESS NOT VALID';
   }
   ```

   The data from the `$_POST['email']` variable is used in the SQL query that inserts the data into the database.

Many languages, such as PHP, include specific functions for data cleansing. Let's take a look at two PHP functions that you can use to clean up data before it is entered into a database: `htmlspecialchars()` and `mysql_real_escape_string()`.

Cleaning up information submitted in HTML format is a simple matter of wrapping the data in PHP's `htmlspecialchars` function. Given a form like this:

```
<form name="search" action="inc/search.php" method="post">
    <label class="label" for="pesearch">Search For: </label>
    <input type="text" name="pesearch" id="pesearch" size="64" /><br />
    <label class="label"> </label>
    <input type="submit" value="Search" />
```

```
    <input type="reset" value="Clear" />
</form>
```

The PHP `htmlspecialchars` function replaces certain characters and returns:

```
&lt;form name=&quot;search&quot; action=&quot;inc/search.php&quot;
→ method=&quot;post&quot;&gt;&lt;label class=&quot;label&quot;
→ for=&quot;pesearch&quot;&gt;Search For: &lt;/label&gt;&lt;input
→ type=&quot;text&quot; name=&quot;pesearch&quot;
→ id=&quot;pesearch&quot; size=&quot;64&quot; /&gt;&lt;br
→ /&gt;&lt;label class=&quot;label&quot;&gt;&amp;nbsp;&lt;
→ /label&gt;&lt;input type=&quot;submit&quot;
→ value=&quot;Search&quot; /&gt;&lt;input type=&quot;reset&quot;
→ value=&quot;Clear&quot; /&gt;&lt;/form&gt;
```

The following characters have been changed:

- Ampersand (&) becomes '`&`'
- Double quote (") becomes '`&quot;`'
- Single quote (') becomes '`&#039;`'
- The less than bracket (<) becomes '`<`'
- The greater than bracket (>) becomes '`>`'

Using PHP's `htmlspecialchars` function makes user-supplied HTML data much safer to use in your Web sites and databases. PHP does provide a function to reverse the effect, which is `htmlspecialchars_decode()`.

Also just as simple is preventing possible SQL injection attacks by using PHP's `mysql_real_escape_string` function. This function works by escaping certain characters in any string. A malicious visitor may try to enter a SQL query into a form field in hopes that it will be executed. Look at the following example in which the visitor is trying to attempt to gain admin rights to the database by changing the database admin password. The hacker has also assumed some common words to try to determine table names:

```
UPDATE `user` SET `pwd`='gotcha!' WHERE `uid`='' OR `uid` LIKE
→ '%admin%'; --
```

If this SQL query was entered into the user name field, you could keep it from running by using `mysql_real_escape_string`:

```
$_POST['username'] = mysql_real_escape_string($_POST['username']);
```

This sets the value of `$_POST['username']` to:

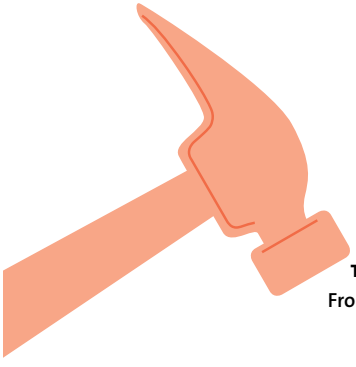
```
UPDATE `user` SET `pwd`=\'gotcha!\' WHERE `uid`=\'\' or `uid` like
→ \'%admin%\'; --
```

Because the query is properly handled and certain characters are escaped, it is inserted into the database and will do no harm.

One other technique that you can use is securing the transmission of data between the client and the server. Let's focus on that next.

TRANSMITTING DATA SECURELY

Another option that you can consider is getting a security certificate for your site or application and then putting your site into an HTTPS security protocol. This is very useful because data traveling between the client and server cannot be read by potential attackers as easily, but it can be costly.



All of the Web site data is encrypted according to keys provided by the security certificate. The Web-site information is transmitted back and forth in a Secure Sockets Layer (SSL). The SSL is a cryptographic communications protocol for the Web. Once the data reaches either end of the transmission, it is decrypted properly for use. If you have used a Web site where the URL begins with `https://` or you have seen the lock icon on your Web browser, you have used a Web site protected in this manner. Many financial institutions and business Web sites use HTTPS to ensure that their data travels securely.

TIP: You can learn more about HTTPS and SSL at the Electronic Frontier Foundation's Web site at www.eff.org/https-everywhere.

WRAPPING **UP**

In this chapter, you learned how to combine jQuery AJAX shorthand methods like .get(), .post() and .load() with server-side scripting to add responsiveness to your HTML forms. Included in this chapter were methods for getting a response back from the server that you could process with jQuery to change page content or provide meaningful messages to your Web site visitors.

You were also introduced to the jQuery low-level AJAX methods that are used for more complex interactions with Web servers. Finally, you learned about JavaScript Object Notation (JSON) and how jQuery's JSON methods can be used to retrieve data from services like Twitter or Flickr for use on the Web sites that you will build.

If the first taste of a jQuery widget has left you hungry for more, you're in luck! Chapter 5, "Applying jQuery Widgets," explores widgets of all shapes and sizes, including several from the jQuery UI project. In addition to widgets from the jQuery UI project, you'll also learn about using plugins that others have developed and how to roll (and publish) your own plugins to share with others. Read on, Macduff!

INDEX

SYMBOLS

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z