Creating Rich Web Applications by Connecting XUL with PHP

First Draft – June 30, 2004

In this multi-part series, I will explore what web applications are, and how the effective combination of XUL, JavaScript, PHP and MySQL can create a compelling and usable web application.

Overview:

- **Part I:** In the first part, I look at the past and present of web applications: what the earliest web applications were like, how they have evolved over time, and how they work.
- **Part II:** In the second part, I look at how a combination of XUL and JavaScript on the client side and PHP and MySQL on the server side can create a compelling and rich web application. I look at how the client and server sides can be bridged by using XML-RPC.
- **Part III:** In the third part, I lay out the code involved in creating a sample web application that is utilized the technologies discussed so far. The application that is created is a fortune cookie machine for the web. Users can retrieve a random cookie, they can add a cookie, and view a list of all the cookies that others have added.

Preview:

If you have a XUL-compatible (Mozilla-based) browser, you can see how a XUL + PHP application looks with this live example:

<IFRAME>

Downloads:

Please feel free to download this entire article as a PDF file, and the accompanying packaged source code (which also contains this article as a PDF file). The PDF file looks better than the HTML version.

- **PDF**: Download this article as a PDF files
- **Code + PDF**: Download the complete source code and this article as a PDF, packaged together as a ZIP file

Requirements:

- Basic knowledge of HTML, JavaScript, how web applications work, and

working knowledge of PHP
- PHP, MySQL on the server side; Mozilla on the client side

Part I : The Past and Present of Web Applications

A Brief History of Web Applications

Java, Java, Java

Today there are countless ways to create an Internet-based application; the first web application technology, however, to receive widespread prominence was **Java**. Netscape was the first to partner with Sun Microsystems in 1997 to enable its Netscape Navigator browser to run embedded Java programs called “applets”. Examples of Java applets included a stock ticker, a chat room, or a running news feed. (Netscape also introduced around this time the similarly named – but substantially different – technology called **JavaScript**, which allowed elements of an HTML document to dynamically changed.) Today Java has gained increased acceptance on the server side, but its application on the browser (or client-) side remains relatively less common. This is due to Microsoft's refusal to bundle the Java Runtime Environment in newer versions of Windows, and due to the perception that browser-side Java applets are slow in performance and unattractive in appearance.

The Rise of Microsoft Internet Explorer and ActiveX

By 1999 the hype surrounding Java gave way to Netscape Navigator's falling out of favor with the masses. By then, people had begun to get acquainted with a new product: Internet Explorer, made by Microsoft. Many websites now began to provide interactive web applications using Microsoft's ActiveX technology. Even today, prominent companies such as Intuit provide services such as a web-enabled version of QuickBooks (called QuickBooks Online) using ActiveX controls.

Mozilla: A Credible Alternative

Numerous security problems associated with the use of ActiveX controls have emerged, making technologies that compete with ActiveX compelling members of the web application marketplace. As we shall see in the following sections, Mozilla - which began its life in 1998 and has evolved rapidly since then – provides a winning set of browser-side (scripting and markup) technologies that make the creation of rich web applications secure and easy.

Mozilla's Contribution to the World of Web Applications

Mozilla began as the open source successor to Netscape with the unveiling of the Netscape 4 source code in 1998. Since then, it has grown from an ambitious rewrite of the

original Netscape product to be a powerful platform on which both desktop and web applications can be built. The original Mozilla Suite, which closely mirrored the bundled-application nature of the Netscape Communicator series has now given way to standalone applications: the Mozilla Firefox browser and the Mozilla Thunderbird email program.

XUL: the Common Thread

XUL, or the XML User Interface language, is the common thread running through all Mozilla-powered applications – both desktop and web-based. XUL is a way to describe an application's user interface using XML. XUL is similar in many way to HTML, while borrowing from, yet not exactly imitating its syntax. (It is possible to mix HTML tags into a XUL application, but we'll come to that later.)

JavaScript: the Activator

JavaScript began as something that allowed you to add interactivity to a webpage by dynamically changing the elements contained within it. So, for example, it could be used to create a pound-to-kilogram converter that resided on a webpage. It has also been used to create stock tickers, games, calculators and, yes, those annoying pop-ups.

With the advent of Mozilla and XUL, JavaScript transcended its original role as solely a *webpage activator*: It can now be considered an *application activator*. I previously mentioned that XUL was a way to describe an application's interface in XML. An XML interface is of no use if it just sits there! If I click a button, I want something to happen. If I choose a menu item, I expect something to happen. JavaScript is the activator for all XUL-based applications. It is what makes something happen when an action is performed, such as selecting a menu, or clicking a button.

PHP: the Server-Side Scripting Wonder

PHP is a server side scripting language that allows actions to be perfomed on data submitted to it by browsers.

Many of you will have used a webmail service such as Yahoo Mail. You may realize that such a service is an HTML-powered web application (and not a XUL-powered one). When you sign in to Yahoo Mail, or type an email and click on the "submit" button, something happens on the "other" side. The server, which is on the other side, processes the submitted information, and does something with it. (If you submit your username and password, it sees if it can log in you in. If you submit an email, it tries to send it.) PHP works in conjunction with several other parts on the server side: the web server itself (often Apache), and a data store (often the MySQL database). We will look at the MySQL database briefly in the following section.

MySQL: Data Storage for All Seasons

MySQL, like PHP, is a freely available and widely used piece of software. It enables information such as usernames and passwords, or received and sent emails to be stored

and retrieved. When PHP checks to see if a username and password are valid, it most often does so by checking with a MySQL database. If PHP needs to save a copy of an email you sent, it saves it to a MySQL database. PHP without MySQL is like Windows without the C drive.

Part II: Using Web Services to Connect Mozilla Applications with PHP and MySQL

Part A: Introduction (Why Rich Web Applications Are Special, How They Are Different, How You Can Implement Them Using Web Services)

In the previous part, I provided an overview of the origin and development of web applications, and the various ways you can create and deploy them. In this section, I will outline how a Mozilla application written using XUL and JavaScript can work in tandem with a server-side application written in PHP and using MySQL as a data store to create a full functional web application.

Making Web Applications Without the Refresh

In the previous section, I cited the example of Yahoo Mail as an HTML-based web application. When you send a message using Yahoo Mail, you type information into the browser, and send that information from the browser to the server where it is acted upon. In such a case, the page **changes**, or is ***refreshed***, after you attempt to send a message. The newly refreshed page indicates whether there was an error sending the message, or whether it was sent successfully. If you were to create a webmail program using XUL and JavaScript, you could make it so that if there were an error, there would be no change of page; you would simply receive a pop-up alert saying that your email could not be sent. You would then still have access to the textbox where you were typing your email. In the case of an HTML-based web application, the page would change and the email you typed would be replaced with an error message. You would then have to use the back button to return to your email message. (Depending on the browser you use, your email could be lost in this back-and-forth process.)

The Advantage of Rich Web Applications

The XUL and JavaScript based Webmail application mentioned above is an example of a rich web application. By ***rich***, I mean a web application that acts more like a program seen on your desktop, and less like a typical HTML-based web application. **A rich web application provides considerable easy-of-use improvements over a typical HTML-based one.** Because page refreshes are not necessary when data is sent from or received by a rich web application, **the chances for data loss and end-user frustration can be minimized**. A rich web application can have a more **immediate and cohesive feel** to it.

Sending and Receiving Information using HTML POST

Most HTML-based web applications send and receive information using an HTML POST. This means that **information is entered into an HTML form, and when the Submit button is pressed, that information is posted to a remote web server**. The remote server responds with some information, which then replaces the previous page in the browser. This is why the previously mentioned **page refresh** happens after an action is taken. With an HTML form, a group of input elements, such as a text input box, a drop-down box, and a group of radio buttons can all be grouped into a single HTML form. When a Submit button is attached to that HTML form, all the information typed into all the elements of that form is submitted to a remote web server for processing. **The problem many XUL developers face is that XUL does not support HTML forms.** (Well, it does, but only in a round-about and cumbersome way.)

Sending and Receiving Information Using Web Services

For XUL applications to send information to or receive information from a remote web server, they have to make use of set of technologies commonly referred to as "**web services**". **With a web service protocol, a request for information from the browser to the server can be sent as an XML document**. The server in turn sends its response back to the browser as an XML document. The advantage of using XML as a transport between the browser and server is that information contained in that XML document can contain additional information, known as **type information**.

What Type Are You?

Regardless of the programming language you have used, you probably have across the idea of types. When you store data in a variable, that data can be a string, an integer, a numerically indexed array, or an associative array. When data is sent using the traditional HTML POST method via HTML forms, there is no type information sent along with it. The information passed using such a traditional route would be mostly bits of plain text send as a request, and chunks of HTML sent as a response.

Part B: Examples (Understanding Web Services With Fortune Cookies)

In the following sections I will explore the concept of creating rich web applications through the example of a fortune cookie web application. [Some of the code samples were initially based on snippets and ideas from the XUL Channels script written by Tim Broddin and made available under the GPL. I have changed most of the code which I had previously used from XUL Channels, but some of the XUL document structures and the JavaScript element access code is based on XUL Channels code. Other code snippets on setting up an XML-RPC server in PHP and using the PEAR DB library were based on code samples available in the Essential PHP Tools book by David Sklar.]

Using JavaScript to Call a Function in PHP

The concept of remote procedure calls is central to web services. You might know

that ***usually any information displayed by JavaScript has to come from another JavaScript resource***. For example, if you used JavaScript to display a fortune cookie's text on a page, the fortune cookie's text would usually have to come from a JavaScript variable. The value of that variable could be hard-coded by doing something like:

```
var cookietext = 'You will be happy and prosperous';
```

And then you print out that information by doing something like:

```
document.write(cookietext);
```

You could also prompt the user for input -

```
var cookietext = prompt('Please enter the text of a fortune cookie', '');
```

and then similarly display on the page what the user has entered, using `document.write()`.

What if you wanted to retrieve the text of a fortune cookie from a remote web server and display it on a page without refreshing the whole page? This can be accomplished through web services, making a remote procedure call from JavaScript to a server-side PHP script.

Similarly, you can also send the text of a fortune cookie that you enter to a remote PHP server for future storage (in a MySQL database) **by using a remote procedure call from within JavaScript**. This **entered text** (along with all other entered fortune cookies) could then be **retrieved for display** at any future date via the remote PHP server.

Consider this bit of JavaScript code, which sends the text of a fortune cookie entered by user to remote web server:

```
var cookietext = 'You will find a new love for marmalade tomorrow';
var globalXmlRpcServer = "http://www.mywebsite.com/xmlrpcserver.php";
var methods = [];

try
        {
        var server = new xmlrpc.ServerProxy(globalXmlRpcServer, methods);

        // calling remote XML_RPC function with addCookie()
        var ourresult = server.addCookie(cookietext);
        // automatic type conversion from PHP int to JS int for ourresult
        if (ourresult == 1)
                {
                        clearC();
                        alert('Your new cookie has been added');
                }
        else
                {
                        alert('Error: could not add new cookie');
```

```
        }
}

catch(e)
{
alert (e);
return 'An error has occured.';
}
```

In the above example, a function on a remote PHP server is called from within JavaScript. The remote function that is called saves a bit of text ('*You will be happy today and tomorrow*') to the remote server. The JavaScript code calls the **addCookie()** function residing on the PHP script located at http://www.mywebsite.com/xmlrpcserver.php. The PHP script receives that information, and saves it to a MySQL database for future use. If the PHP script could successfully save the submitted information, it passes back the response of "1" whose type is an integer. Similarly, if the PHP script could not save the submitted information, it passes back a response of "0" whose type is an integer. The JavaScript code reads the response, and make a decision as to which message should be presented to the user as a browser pop-up alert: a "success!" message, or an "Error!" message. [You may have also noticed the try ... catch sequence. If an error occurred on the server side (such as the PHP script no being able to connect to the MySQL server), such an error is returned as an XML-RPC fault, and is made available when the exception is caught using catch {}. XML-RPC is the specific web services protocol that we are using here, and it will be explained in more detail shortly.]

In this example, the information sent via the **addCookie()** function is encoded as XML before it is sent from browser to server, and the response of "1" or "0" is also encoded as XML before it is sent from server to browser. As programmers, we do not have to worry about encoding the information as XML, and decoding it from XML to a native (JavaScript or PHP) data type. This is because on both on the browser and server end, t**he remote procedure libraries take care of this for us**.

XML-RPC: An Easy-To-Use Web Services Protocol

The specific web service protocol used above was XML-RPC, which was introduced by Userland Software in 1998. There are many other more complicated protocols (such as SOAP) which can do the same thing – but for the sake of simplicity, I have used XML-RPC here. The JavaScript XML-RPC library that I used (which took care of encoding the information from and to XML and sending and receiving XML-encoded information) was written by Jan Kolhoff, and is freely available at his website under an open source license: http://xmlrpc.kollhof.net/jsolait.xmlrpc/index.xhtml. (His XML-RPC library is part of his larger "JavaScript o lait" library.)

The Server-Side of XML-RPC based Web Services

So far we have seen how a small bit of JavaScript code can send information to a remote PHP web server using XML-RPC. But what we have not seen it what happens on the

server side when that information is received by the PHP script. How does the PHP script process the incoming information? Which libraries does it need to process data encoded as XML ?

What are PEAR Libraries? Why do We Need Them? Where Can We Find Them?

PEAR libraries are classes written in PHP and made available through the PHP Extension and Application Repository (PEAR) at http://pear.php.net/. There are many PEAR libraries available, but the most important ones are called the "**core libraries**" and are **bundled with PHP as of version 4.3.0**. There is a core PHP library called XML-RPC, which enables a PHP script to act an XML-RPC server, processing XML data sent to it by another XML-RPC client (such as the mini-JavaScript client mentioned above). There is another core PEAR class called DB, which aids in connecting a PHP script to a database server. I will be using both these PEAR libraries in the code examples that follow.

Creating a Simple PHP XML-RPC Server

Consider the following PHP code snippet:

```
require('prefs.php'); // Load preferences (database connection info)
require 'XML/RPC/Server.php'; // PEAR XML-RPC library
require 'DB.php'; // PEAR DB library

function addcookie($xmlrpcdata)
{
    $cookietext = XML_RPC_decode($xmlrpcdata->params[0]);

    global $username,$password,$server,$database,$table;
    global $XML_RPC_erruser;


$dbh = DB::connect("mysql://$username:$password@$server/$database");
   // Establish MySQL DB Connection via PEAR DB

  if ( DB::isError($dbh) )
  // CHECK PEAR DB CONNECTION ERRORS
  {
  // Return XML-RPC fault
   return new XML_RPC_Response(0, $XML_RPC_erruser + 1, "DB Error: Could not
connect to server.");
  }
  else
  {
   // Insert fortune cookie text into database
   $query = "INSERT INTO $table (fortunecookie) VALUES ('$cookietext')";
```

```
    $sth = $dbh->query($query);
    if ( $dbh->affectedRows() == 1) // Was the text entered successfully?
        {
         $retval = 1;// If so, return 1
        }
    else
        {
         $retval = 0; // Oops, an error occured. Return 0.
        }
        $retval = XML_RPC_encode($retval);
        return new XML_RPC_Response($retval);
    }
}

/* Set up a dispatch map which maps public, XML_RPC function names to the private,
PHP functions we have just written. */
$dispatch_map = array
(
 'addCookie' => array('function' => 'addcookie')
);

// Start the server
$server = new XML_RPC_Server($dispatch_map);
```

In the code example above, we set up an XML-RPC server using PHP and some PEAR libraries. In the beginning a preferences file (prefs.php) in included, which contains information that the PHP script needs to connect to a MySQL server. Then the PEAR XML-RPC and DB libraries are included. Then the addcookie() function is defined. This function is called by the JavaScript XML-RPC client, as illustrated in the previous example. The addcookie() function takes the string passed to it by the JavaScript client and enters it into the MySQL database. We also set up a dispatch map that maps the publicly available function name [addCookie()] to the real PHP one [addcookie()]. Thus, when the following JavaScript statement is run -

```
var ourresult = server.addCookie(cookietext);
```

- in actuality the PHP addcookie() function is being called through the public alias of addCookie().

Part III – Building a Fully Functional Fortune Cookie using XUL, JavaScript, PHP, MySQL and the XML-RPC Web Services Protocol

So far I have given you a rough overview of how to build a JavaScript XML-RPC client that communicates with a PHP XML-RPC server. But I have not shown you the specific XUL code required to invoke the JavaScript, and I have also left out some other details regarding related functions on both the browser and client side.

In this section I will now give more concrete examples and the complete code required to build a fully functional XUL + JavaScript + PHP + MySQL + XML-RPC fortune cookie web application that does the following:

- **Gets A Random Cookie**: Retrieves a random cookie from the remote server, and dynamically adds it to the XUL document
- **Lists All Cookies**: Retrieves a list of all cookies stored on the remote server and dynamically adds this list to the XUL document
- **Add a Cookie**: Adds a cookie to the remote server after text is typed into a textbox and the submit button is pressed.

Database Setup

Before I go any further into how the interface will look, I will go over setting up the MySQL database which will store the fortune cookies required for this application.

First create a MySQL database if don't already have one, using PHPMyAdmin, or the MySQL client of your choice. Then similarly use the following SQL to create a table for the fortune cookies and insert the beginning content.

```
#
# Create table `fortunecookies`
#

CREATE TABLE fortunecookies (
  id mediumint(9) NOT NULL auto_increment,
  fortunecookie varchar(255) NOT NULL default '',
  PRIMARY KEY  (id)
) TYPE=MyISAM;


#
# Insert data into table `fortunecookies`
#

INSERT INTO fortunecookies VALUES (1, 'Many receive advice, only the wise profit by it.');
INSERT INTO fortunecookies VALUES (2, 'Be assertive when decisive action is needed');
INSERT INTO fortunecookies VALUES (3, 'Our first love and last love is self-love');
INSERT INTO fortunecookies VALUES (4, 'Good food brings good health and longevity');
INSERT INTO fortunecookies VALUES (5, 'Rest has a peaceful effect on your physical and emotional health');
INSERT INTO fortunecookies VALUES (6, 'The star of riches is shining upon you');
INSERT INTO fortunecookies VALUES (7, 'Happiness begins with facing life with a
```

smile and a wink.');

Setting up Preferences

Then create a file called **prefs.php** with the following variables. Edit this file to match your MySQL setup (and the name of the table which you just created).

```
<?php
/*
Preferences File (for server-side settings)
*/

$username = "username";
$password = "password";
$database = "xulwebservices";
$server = "localhost";
$dsn = "mysql://$username:$password@$server/$database"; //
used by PEAR DB class
$table = "fortunecookies"; // main articles table
?>
```

Then create a file called **prefs_client.php** which looks like the following:

```
<?php
//prefs_client.php
$xmlrpcserver =
"http://localhost/fortunecookies/fortunecookieserver.php";
?>
```

Make sure you modify the value of $xmlrpcserver to match where you have the fortunecookie script installed. The example assumes you have it installed on your local development box (hence "http://localhost).

Finally, create a file called **prefs_client_js.php** which looks like the following:

```
<?php
require 'prefs_client.php';
echo "var globalXmlRpcServer = '$xmlrpcserver';";
?>
```

How it Looks

Here is how this application will look:

The Main (index) File

The main XUL document for this application will be called index.php and will need to set the correct Server MIME type through PHP's header function so that Mozilla knows that it is a XUL file.

We will make three XUL tabs for each of the tasks outlined above (getting a random cookie, listing all cookies, and add a cookie).

Here is what the index.php file should look like:

```
<?
header ("Content-type: application/vnd.mozilla.xul+xml; charset=iso-8859-15");
echo '<' . '?xml version="1.0" encoding="iso-8859-15" ?' . '>';
echo '<' . '?xml-stylesheet href="chrome://global/skin/" type="text/css"?' . '>' . "\n";
?>
 <window title="Add a new channel"
xmlns:html="http://www.w3.org/1999/xhtml"
xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
style="margin: 10px;"
id="fortuneCookies">

        <script type="application/x-javascript" src="fortunecookies.js" />
        <script type="application/x-javascript" src="prefs_client_js.php" />
        <script type="application/x-javascript" src="load_xmlrpc/init.js" />
        <script type="application/x-javascript" src="load_xmlrpc/tester.js" />
```

```
<tabbox>
  <tabs>
    <tab label="Get a Cookie"/>
    <tab label="List All Cookies"/>
    <tab label="Add a Cookie"/>
  </tabs>

  <tabpanels>

    <tabpanel id="getacookie">
    <vbox>
        <hbox>
                              <button id="btnLoadFortuneCookie" label="Get Random Cookie"
oncommand="retrieveFortuneCookie();"/>
          </hbox>

          <hbox>
                              <vbox>
                              <spacer flex="1"/>
                              <groupbox style="width:400px;margin-top:15px;margin-
bottom:15px;">
                                        <caption label="Cookie:" />
                                        <description id="descCookieBox">A cookie will appear
here.</description>

                              </groupbox>
                              </vbox>
          </hbox>

          <hbox>
                              <button id="btnLoadText" label="Move Cookie Below"
oncommand="moveCookies();" style="margin-top:15px;margin-bottom:15px;" />
          </hbox>

          <hbox>

                              <textbox  style="margin-bottom: 5px;width:400px;"
id="txtCookieBox" multiline="true" rows="5" cols="10" />

          </hbox>
    </vbox>
    </tabpanel>

    <tabpanel id="listallcookies">
    <vbox>
            <hbox>
            <description>To show all the cookies that have been entered so far, press the
```

```
button below.</description>
             </hbox>

             <hbox>
             <button id="btnShowAll" label="List All Fortune Cookies"
oncommand="getAllCs();"/>
             </hbox>

             <hbox>
             <listbox id="listAllCookies" style="width:400px;">

             </listbox>
             </hbox>
    </vbox>

   </tabpanel>

   <tabpanel id="addacookie">
   <vbox>
      <hbox>

                   <vbox>
                             <description style="margin-bottom: 5px;">Please enter a fortune
cookie's message:</description>
                             <textbox  style="margin-bottom: 5px;" id="txtCookie"
multiline="true" rows="5" cols="10" />
                   </vbox>
          </hbox>

          <hbox>
                             <button id="btnSubmit" label="Submit" oncommand="addC();"/>
          </hbox>

          <hbox>
                             <button id="btnEchoText" label="Echo Your Words"
oncommand="var whatyousay = prompt('What\'s on your mind?','');echoMe(whatyousay);
" style="margin-top:25px;"/>
          </hbox>
   </vbox>
   </tabpanel>

  </tabpanels>

</tabbox>

</window>
```

What Just Happened?

In the example above, we first **used the PHP header function** to send the current Server MIME type for index.php (Content-type: application/vnd.mozilla.xul+xml; charset=iso-8859-15). Then we used PHP to display the document type declaration as that of XML. If we had directly done this outside of PHP tags, **we would have received a PHP parse error**, as the **delimiters for the beginning and end of the XML document type declaration are the same as those of PHP** ("<?" and "?>"). Thus, just entering the following outside of the PHP tags would have caused a parse error:

```
<?xml version="1.0" encoding="iso-8859-15" ?>
```

The rest of the document is a XUL document with embedded JavaScript function calls, and CSS styling instructions.

A XUL file is an XML document and its very first tags are the **<window> </window>** tags. The window tag contains various attributes so that this XML document is established as a XUL file.

Immediately after the first window tag is opened, **various JavaScript files are imported**. These include the fortunecookies.js file, the prefs_client_js.php file, and two other files which load the XML-RPC JavaScript library. The fortunecookies.js file is the main JavaScript file for the XML-RPC functions that we will be writing. The prefs_client_js.php file is a JavaScript file that is generated by PHP. It reads information about the location of the remote XML-RPC from the prefs_client.php file, and makes it available through a global JavaScript variable.

The rest of the document contains the XUL tags required to set three different tabbed sections for the three separate tasks listed above. To start a set of tabs, we use the parent <tabbox> </tabbox> tag. Contained within this tag are two other tags: <tabs> </tabs> and <tabpanels> </tabpanels>. Contained within the <tabs> </tabs> section is a list of the individual tab titles (otherwise called tab boxes). And the <tabpanels> </tabpanels> section contains a list of the sections that are loaded when a tab title is selected (otherwise called tab panels). The basic structure of a tabbed XUL document looks as follows:

```
<tabbox>

  <tabs>
    <tab label="Get a Cookie"/>
    <tab label="List All Cookies"/>
    <tab label="Add a Cookie"/>
  </tabs>

<tabpanels>
     <tabpanel id="getacookie"> </tabpanel>
     <tabpanel id="listallcookies"> </tabpanel>
     <tabpanel id="addacookie"> </tabpanel>
```

</tabpanels>

</tabbox>

Each <tabpanel> </tabpanel> tag contains further tags for each panel's functionality. If you are still confused about how XUL tabs work, please consult XULPlanet section on tabs.

The tab panel for the "**Get a Cookie**" section contains various tags:

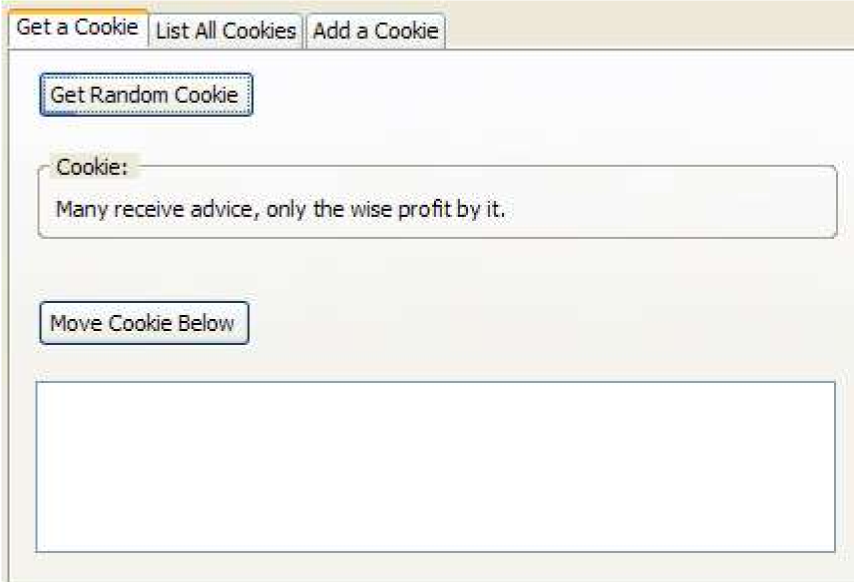
- a button that reads "Get Random Cookie"
- a description box that reads "A cookie will appear here"
- a button that reads "Move Cookie below"
- a textbox that is empty

Get a Cookie Section

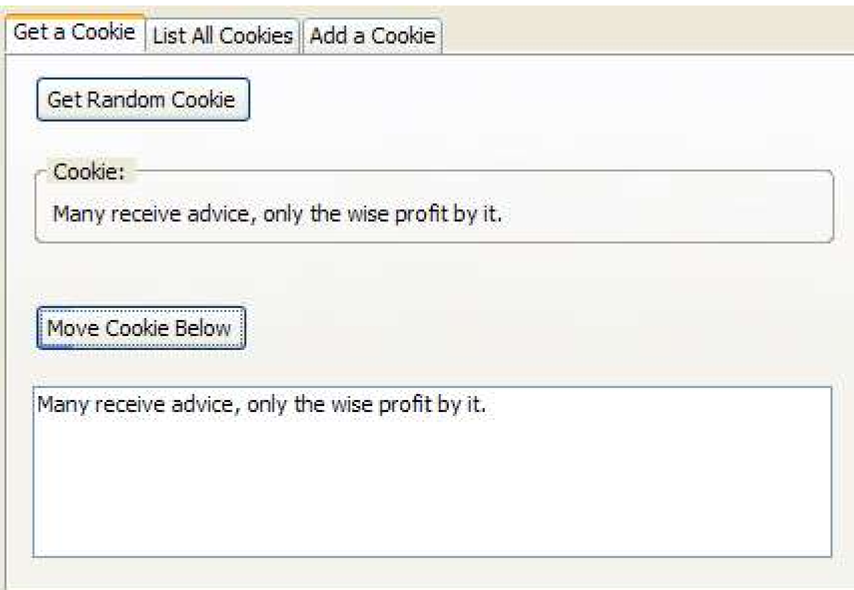
Here is how the "Get a Cookie" section looks:

When the "Get Random Cookie" button is pressed, the description box is filled with the text of a randomly selected cookie. When the "Move Cookie below" button is pressed, the empty textbox below is filled with the text of the cookie that was just retrieved and placed in to the description box.

Here is how this tab looks after the "Get Random Cookie" button has been pressed:



And here is how it looks after the “Move Cookie Below” button has been pressed:



The text in the description box cannot be selected, but the text in the textbox can.

There are various <vbox> </vbox> and <hbox> </hbox> tags inside of each tabpanel section. These take some time to get used to; for now, it should suffice to say that they are used to determine how the elements contained within them show up: either stacked on top of each other vertically, or nestled next to each other horizontally. I have omitted these vbox and hbox tags in the excerpt below, and instead list the buttons, description box and textbox tags that were mentioned in the screens above.

Get Random Cookie button:

```
<button id="btnLoadFortuneCookie" label="Get Random Cookie"
oncommand="retrieveFortuneCookie();"/>
```

Description Box:

```
<groupbox style="width:400px;margin-top:15px;margin-bottom:15px;">
<caption label="Cookie:" />
<description id="descCookieBox">A cookie will appear here.</description>
</groupbox>
```

Move Cookie Below button:

```
<button id="btnLoadText" label="Move Cookie Below" oncommand="moveCookies();"
style="margin-top:15px;margin-bottom:15px;" />
```

Textbox:

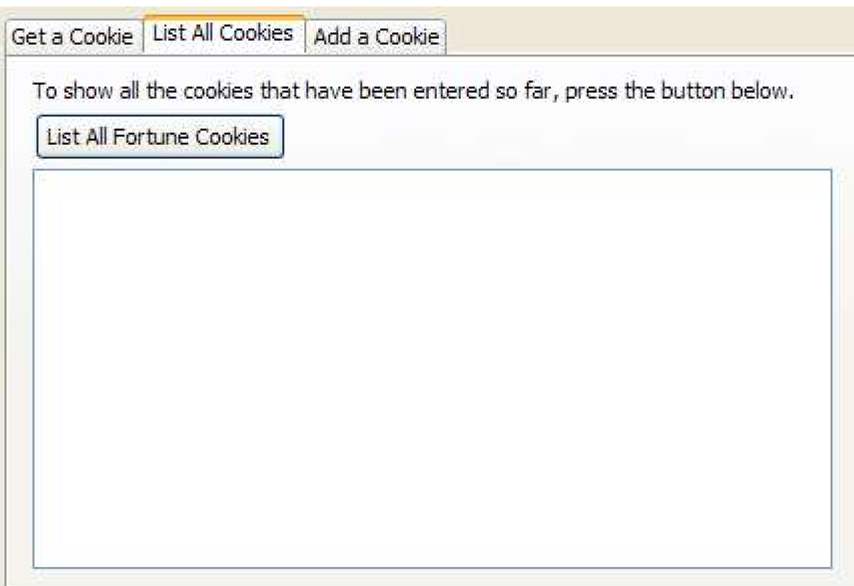
```
<textbox  style="margin-bottom: 5px;width:400px;" id="txtCookieBox" multiline="true"
rows="5" cols="10" />
```

The description tag does not have an equivalent in HTML, and the textbox tag is most similar to either the HTML input element or the HTML textarea element, depending on whether the **multiline=”true”** attribute is used. A JavaScript action is associated with each button by using the **oncommand** attribute. This is similar to the onmouseover (and other) attributes used to associate JavaScript actions to HTML document elements.

When the “Get Random Cookie” button is pressed, the **retrieveFortuneCookie()** function in fortunecookies.js is called. **The retrieveFortuneCookies() function makes an XML-RPC function call to the remote PHP server, retrieves a random fortune cookie in the form of XML, decodes it, and inserts it into the XUL document.** When the “Move Cookie Below” button is pressed, the **moveCookies()** function in fortunecookies.js is called.

List All Cookies Section

The “list all cookies” section is displayed in the following image:



It consists of a "**List All Fortune Cookies**" **button** and a **listbox** tag that is filled with a list of all the cookies in the MySQL database after this button is clicked.

A listbox has no direct equivalent in HTML, but perhaps an approximate equivalent would be the <ul> tag and its <li> children.

The code for these two elements is as follows:

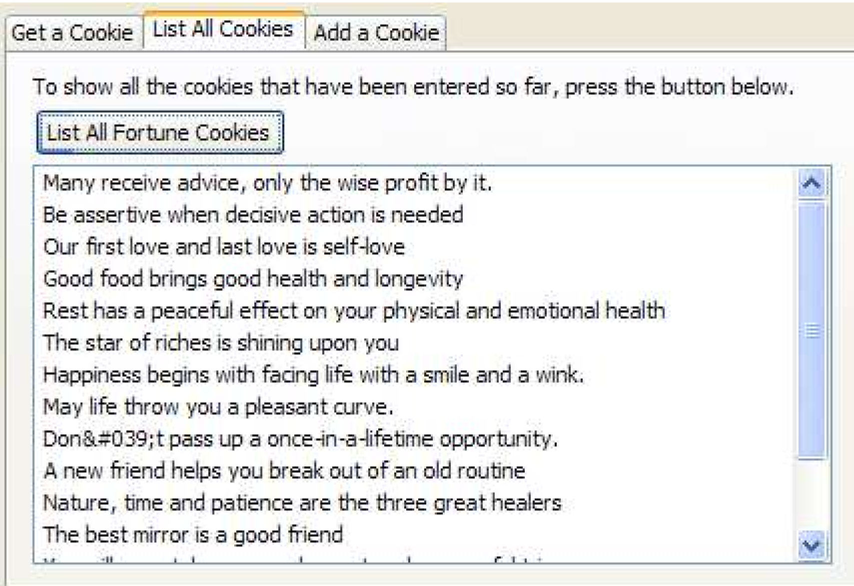
```
<button id="btnShowAll" label="List All Fortune Cookies" oncommand="getAllCs();"/>
```

and

```
<listbox id="listAllCookies" style="width:400px;"> </listbox>
```

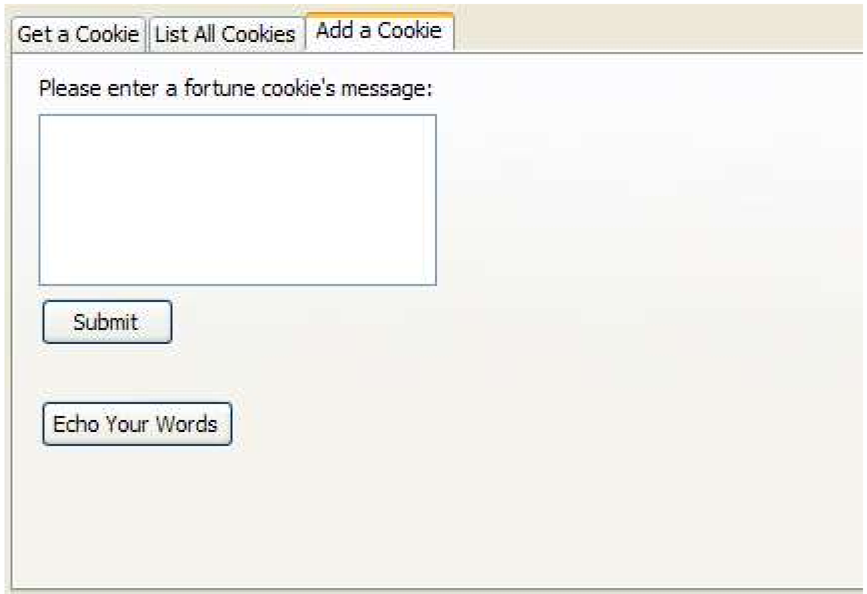
When the "**List All Fortune Cookies**" button is pressed, the **getAllCs()** function is called. Again, this function resides in the fortunecookies.js file.

After the "List All Fortune Cookies" button is pressed, this is how this tab panel should look:



Add a Cookie Section

The “add a cookie” section is displayed in the following image:



This section contains the following XUL tags:

Description Box:
<description style="margin-bottom: 5px;">Please enter a fortune cookie's message:</description>

Textbox:
<textbox style="margin-bottom: 5px;" id="txtCookie" multiline="true" rows="5"

cols="10" />

Submit Button:
<button id="btnSubmit" label="Submit" oncommand="addC();"/>

Echo Input Button:
<button id="btnEchoText" label="Echo Your Words" oncommand="var whatyousay = prompt('What\'s on your mind?','');echoMe(whatyousay);" style="margin-top:25px;"/>

When text is typed into the textbox, and the **Submit button** is pressed, the entered text is submitted via the **addC()** JavaScript function to the remote server. The **addC()** function makes an XML-RPC function call to the remote PHP script, which in turn adds the submitted text to the database.

The **Echo Input button** prompts the user to enter some text when it is pressed. The echoMe() function then takes that inputted text, and submits it to the remote server using XML-RPC. The PHP script at the server takes this text, adds some more text in front of it, and returns it to the echoMe() function. The echoMe() function then adds this text to the textbox above it. This functionality was written in order to demonstrate how a simple echo function could created using JavaScript on the client side and PHP on the server side and XML-RPC as the transport.

The fortunecookies.js file:

Here is what the fortunecookies.js file looks like:

```
function setTxtVal(what)
{
	var txtVal = document.getElementById('txtCookieBox');
	txtVal.value = what;
}

function moveCookies()
{
	var descCB = document.getElementById('descCookieBox');
	var txtCB = document.getElementById('txtCookieBox');
	txtCB.value = descCB.value;
}

function clearC()
{
	var txtVal = document.getElementById('txtCookie');
	txtVal.value = '';
}
```

```
function addC()
{
	var txtVal = document.getElementById('txtCookie');
	msg = txtVal.value;
	addFortuneCookie(msg);
}

function addFortuneCookie(cookietext)
{
	var methods = [];

	try
	{
	var server = new xmlrpc.ServerProxy(globalXmlRpcServer, methods);

	var ourresult = server.addCookie(cookietext); // calling remote XML_RPC
function here!
	// automatic type conversion here from PHP int to JS int
	if (ourresult == 1)
		{
			clearC();
			alert('Your new cookie has been added');
		}
	else
		{
			alert('Error: could not add new cookie');
		}
	}

	catch(e)
	{
	alert (e);
	return 'An error has occured.';
	}
}

function getRandCookie()
{
	var methods = [];

	try
	{
	var server = new xmlrpc.ServerProxy(globalXmlRpcServer, methods);

	var ourresult = server.getRandomCookie(); // calling remote XML_RPC function
here!
	// automatic type conversion here from PHP assoc array to JS object
```

```
        var cookietext = ourresult['fortunecookie']; // articlename is now a string
        return cookietext;
        }

        catch(e)
        {
        alert(e);
        return 'An error has occured.';
        }
}

function getAllCs()
{
        var methods = [];

        try
        {
        var server = new xmlrpc.ServerProxy(globalXmlRpcServer, methods);

        var allcookies = server.getAllCookies(); // calling remote XML_RPC function
here!
        // automatic type conversion here from PHP assoc array to JS object
        cookies_list = "";
        var listVal = document.getElementById('listAllCookies');

                for (var i in allcookies)
                {
                        acookie = allcookies[i];
                        fc = acookie['fortunecookie'];
                        id = acookie['id'];
                        if (fc)
                        // Mystery bug:
                        // if this is not done, an "undefined" item is the last item added to
the list
                        {
                                listVal.appendItem(fc, id);
                        }
                }
        }
        catch(e)
        {
        alert(e);
        }
}

function echoMe(what)
{
```

```
	var methods = [];

	try
	{
	var server = new xmlrpc.ServerProxy(globalXmlRpcServer, methods);
	var ourresult = server.echoEcho(what); // calling remote XML_RPC function
here!
	var txtVal = document.getElementById('txtCookie');
	txtVal.value = ourresult;
	}

	catch(e)
	{
	alert(e);
	}
}

function retrieveFortuneCookie()
{
	var txtVal = document.getElementById('descCookieBox');
	var what = getRandCookie();
	// alert('I got: ' + what);
	txtVal.value = what;
}
```

Most of the principles involved with using JavaScript to create an XML-RPC client were explained earlier on in this tutorial. Depending on user feedback, I will elaborate on what the above code does.

Creating The PHP XML-RPC Server File

The main concepts of the PHP-based XML-RPC server were explained earlier on this tutorial. Depending on user feedback, I will explain in further detail what all the functions in the follow code do. For now, here is the source code for the **fortunecookieserver.php** file.

```
<?php

require('prefs.php');
// Load the class and function definitions
/*
The following libraries are part of the core PEAR libraries and are installed by default
with PHP versions of 4.3 or higher.
*/
require 'XML/RPC/Server.php'; // PEAR XML-RPC library
require 'DB.php'; // PEAR DB library
```

```
function echothis($xmlrpcdata)
{
   $hellostr = XML_RPC_decode($xmlrpcdata->params[0]);
   $hellostr = 'The text you submitted said nicely: ' . $hellostr;
   $retval = XML_RPC_encode($hellostr);
   return new XML_RPC_Response($retval);
}

function getrandomcookie()
{
   global $username, $password, $server, $database, $table;
   global $XML_RPC_erruser;
   $dbh = DB::connect("mysql://$username:$password@$server/$database");
   if ( DB::isError($dbh) )
   // CHECK PEAR DB CONNECTION ERRORS
   {
      error_log("PEAR DB Error - could not run connect to server: $server ", 0);
      // Return XML-RPC fault
      return new XML_RPC_Response(0, $XML_RPC_erruser + 1, "DB Error: Could not
connect to server.");
   }
   else
   {
      // Retrieve list of ids and cookies corresponding to fortune cookies in the table
      $dbh->setFetchMode(DB_FETCHMODE_ASSOC);
      $query = "SELECT id, fortunecookie FROM $table";
      $fcookies = $dbh->getAll($query);

      /*
      $fcookies should be a numerically indexed array
      Each numeric index of this array should hold another array
      whose keys are the column names of a row
      i.e., something like:
      Array
      (
         [0] => Array
         (
            [id] => 5
            [fortunecookie] => Rest has a peaceful effect on your physical and emotional
health
         )
      )
      */
      if ( DB::isError($sth) )
      // CHECK FOR PEAR DB / QUERY RUN ERRORS
      {
```

```
            error_log("PEAR DB Error - could not run query $query ", 0);
            return new XML_RPC_Response(0, $XML_RPC_erruser + 2, "DB Error: Could
not run query.");
        }
        else
        {
            $numelements = count($fcookies);
            $rand = rand(0, $numelements - 1);
            $acookie = $fcookies[$rand]; // $acookie is an associative array
            // $acookietext = $acookie['fortunecookie'];

            // escape quotes and stuff to prevent errors in XML transport
            $acookie['fortunecookie'] = htmlentities($acookie['fortunecookie'],
ENT_QUOTES);
            $acookie['fortunecookie'] = str_replace('’', '&#039;', $acookie['fortunecookie']);

            /*
            $acookie is an associative array that looks something like:
            Array
            (
                [id] => 5
                [fortunecookie] => Rest has a peaceful effect on your physical and emotional
health
            )
            */

            $retval = XML_RPC_encode($acookie);
            return new XML_RPC_Response($retval);
        }

    }
}



function getallcookies()
{
    global $username, $password, $server, $database, $table;
    global $XML_RPC_erruser;
    $dbh = DB::connect("mysql://$username:$password@$server/$database");
    if ( DB::isError($dbh) )
    // CHECK PEAR DB CONNECTION ERRORS
    {
        error_log("PEAR DB Error - could not run connect to server: $server ", 0);
        // Return XML-RPC fault
        return new XML_RPC_Response(0, $XML_RPC_erruser + 1, "DB Error: Could not
connect to server.");
```

```
    }
    else
    {
        // Retrieve list of ids and cookies corresponding to fortune cookies in the table
        $dbh->setFetchMode(DB_FETCHMODE_ASSOC);
        $query = "SELECT id, fortunecookie FROM $table";
        $fcookies = $dbh->getAll($query);

        if ( DB::isError($sth) )
        // CHECK FOR PEAR DB / QUERY RUN ERRORS
        {
            error_log("PEAR DB Error - could not run query $query ", 0);
            return new XML_RPC_Response(0, $XML_RPC_erruser + 2, "DB Error: Could
not run query.");
        }
        else
        {
            $retval = XML_RPC_encode($fcookies);
            return new XML_RPC_Response($retval);
        }

    }
}

function addcookie($xmlrpcdata)
{
    $cookietext = XML_RPC_decode($xmlrpcdata->params[0]);
    $cookietext = htmlentities($cookietext, ENT_QUOTES);
    $cookietext = str_replace("’", '&#039;', $cookietext);
    global $username, $password, $server, $database, $table;
    global $XML_RPC_erruser;
    $dbh = DB::connect("mysql://$username:$password@$server/$database");
    if ( DB::isError($dbh) )
    // CHECK PEAR DB CONNECTION ERRORS
    {
        error_log("PEAR DB Error - could not run connect to server: $server ", 0);
        // Return XML-RPC fault
        return new XML_RPC_Response(0, $XML_RPC_erruser + 1, "DB Error: Could not
connect to server.");
    }
    else
    {
        $query  = "INSERT INTO $table (fortunecookie) VALUES ('$cookietext')";
        $sth = $dbh->query($query);
        if ( $dbh->affectedRows() == 1)
        {
            $retval = 1;
```

```
        }
        else
        {
           $retval = 0;
        }
        $retval = XML_RPC_encode($retval);
        return new XML_RPC_Response($retval);
    }
}

/* Set up a dispatch map which maps public, XML_RPC function names to the private,
PHP functions we have just written. */
$dispatch_map = array(
                  'getRandomCookie' =>
                  array('function' => 'getrandomcookie'),

           'getAllCookies' =>
                  array('function' => 'getallcookies'),

           'addCookie' =>
                  array('function' => 'addcookie'),

            'echoEcho' =>
                  array('function' => 'echothis')

           );

// Start the server
$server = new XML_RPC_Server($dispatch_map);
?>
```

Summary and Conclusion:

In this three-part series, I have attempted to provide an overview of the past and present of web applications, the advantage of combining XUL with PHP to create a rich web application, and the specific code involved with doing so. Please send feedback for future versions of this three part series to jay [at] moztips.com.