Ajax and XML: Ajax for chat

Use Ajax and PHP to build a chat application

Jack D. Herrington

December 04, 2007

Learn to build a chat system into your Web application with Asynchronous JavaScript™ + XML (Ajax) and PHP. Your customers can talk to you and to each other about the content of the site without having to download or install any special instant-messaging software.

View more content in this series

Developers talk a lot about community when the term Web 2.0 comes up. And whether you think it's hype or not, the idea of engaging your customers or readers in an instantaneous conversation about the topics at hand or the products you're selling is pretty compelling. But how do you get there? Can you put a chat box on the same page as your product listing so that your customers don't have to install special software or even Adobe Flash Player? Sure! Turns out, you can do it all with free, off-the-shelf tools such as PHP, MySQL, dynamic HTML (DHTML), Ajax, and the Prototype.js library.

Without further ado, let's jump right into the implementation.

Logging in

The first step in chatting is to have an identity. That requires a rudimentary login page, such as the one shown in Listing 1.

Listing 1. index.html

```html
<html>
<head><title>Chat Login</title></head>
<body>
<form action="chat.php" method="post">
Username: <input type="text" name="username">
<input type="submit" value="Login">
</form>
</body>
</html>
```

In Figure 1, you can see a screenshot of this page.
Figure 1. The login window for the chat

![Login Window](image)

Note: I only need this for the example, because I want to determine who's saying what. For your application, you probably already have a login, so you can use your existing user name.

A basic chat system

A chat system is really just a table of strings in which each string is attributed to an individual. The most basic version of the schema is shown in Listing 2.

Listing 2. chat.sql

```
DROP TABLE IF EXISTS messages;
CREATE TABLE messages (
    message_id INTEGER NOT NULL AUTO_INCREMENT,
    username VARCHAR(255) NOT NULL,
    message TEXT,
    PRIMARY KEY ( message_id )
);
```

The script contains an auto-incrementing message ID, the user name, and the message. If you think it's important, you can also add a timestamp to each message to track when it was sent.

If you want to manage multiple conversations on different topics, you must have another table to track the topics and include a related topic_id in the messages table. I wanted to keep this example simple, so I used the simplest possible schema.

To set up the database and load the schema, I used these instructions:

```
% mysqladmin create chat
% mysql chat < chat.sql
```

Depending on the setup of your MySQL server and its security settings and passwords, your command might vary a bit.

The basic user interface (UI) of the chat is shown in Listing 3.
Listing 3. chat.php

```php
<?php
if ( array_key_exists( 'username', $_POST ) ) {
    $_SESSION['user'] = $_POST['username'];
}
$user = $_SESSION['user'];
?>
<html>
<head><title><?php echo( $user ) ?> - Chatting</title>
<script src="prototype.js"></script>
</head>
<body>
<div id="chat" style="height:400px;overflow:auto;"> </div>
<script>
function addmessage() {
    new Ajax.Updater( 'chat', 'add.php',
    {
        method: 'post',
        parameters: $('chatmessage').serialize(),
        onSuccess: function() {
            $('messagetext').value = '';
        }
    });
}
</script>
<form id="chatmessage">
<textarea name="message" id="messagetext">
</textarea>
</form>
<button onclick="addmessage()">Add</button>
<script>
function getMessages() {
    new Ajax.Updater( 'chat', 'messages.php', {
        onSuccess: function() { window.setTimeout( getMessages, 1000 ); }
    });
} getMessages();
</script>
</body>
</html>
```

At the top of the script, you get the user name from the posted arguments of the login page and store it in the session. The page then continues to load the invaluable Prototype.js JavaScript library, which will handle all the Ajax work for you.

After that, the page has a spot in which the chat messages go. This area is populated by the `getMessages()` JavaScript function located at the bottom of the file.

Below the messages area is a form and a textarea in which users type their message text. You also have a button labeled Add that adds the message to the chat.

The page looks like Figure 2.
Figure 2. The simple chat window

Careful inspection of the `getMessages()` function shows that the page is indeed polling the server every 1,000 milliseconds (1 second) to check for new messages and put the output of the call into the messages area at the top of the page. I talk more about polling later in the article, but for the moment, I want to finish the basic implementation of the chat by showing the messages.php page, which returns the current set of messages. This page is shown in Listing 4.

Listing 4. messages.php

```php
<table>
<?php
// Install the DB module using 'pear install DB'
require_once("DB.php");

$db =& DB::Connect( 'mysql://root@localhost/chat', array() );
if (PEAR::isError($db)) { die($db->getMessage()); }

$res = $db->query('SELECT * FROM messages');
while( $res->fetchInto( $row ) )
{
<?php echo($row[1]) ?></td>
<td><?php echo($row[2]) ?></td></tr>
<?php
}
</table>
```

At the top of the script, I connect to the database with the DB library, which is available from PEAR (see Related topics). If you haven't installed that library already, you can do so with the following command:

```
% pear install DB
```

With PEAR installed, the script can query the current messages, fetch each row, and output the user name and the comment text.
The final script is the add.php script, which is called from the Prototype.js Ajax code in the addmessage() function on the page. This script takes the message text and the user name from the session and inserts a new row into the messages table. This code is shown in Listing 5.

Listing 5. add.php

```php
<?php
// Install the DB module using 'pear install DB'
require_once("DB.php");
$db =& DB::Connect( 'mysql://root@localhost/chat', array() );
if (PEAR::isError($db)) { die($db->getMessage()); }

$sth = $db->prepare( 'INSERT INTO messages VALUES ( null, ?, ? )' );
$db->execute( $sth, array( $_SESSION['user'], $_POST['message'] ) );
?
<table>
<?php
$res = $db->query('SELECT * FROM messages' ,
while( $res->fetchInto( $row ) )
{
  ?
  <tr><td><?php echo($row[1]) ?></td>
  <td><?php echo($row[2]) ?></td></tr>
<?php
}
</table>
```

The add.php script also returns the current list of messages, because the Ajax code on the original page updates chat messages from the returned HTML code. This behavior gives users immediate feedback that they added a comment to the conversation.

These are the basics of the chat system. The next section covers how to make the polling a bit more efficient.

Making a better chat

With the original chat system, the page requests all the chat messages for the conversation every second. While not too bad for a short conversation, for something longer, it will be a real performance problem. Thankfully, there's a fairly easy solution. A message_id is associated with each message, and that number increases incrementally. So, if you know that you have messages to a certain ID, then you simply ask for any messages that occur after that ID. That keeps the message traffic down in a big way. For most requests, you're likely get no new messages, which is a very small packet.

Changing this setup to the more efficient design requires a few changes to the chat.php page, as shown in Listing 6.

Listing 6. chat.php (revised)

```php
<?php
if ( array_key_exists( 'username', $_POST ) ) {
  $_SESSION['user'] = $_POST['username'];
}
$user = $_SESSION['user'];
?>
```
<html>
<head>
<title><?php echo( $user ) ?> - Chatting</title>
<script src="prototype.js"></script>
</head>
<body>
<div style="height:400px;overflow:auto;">
<table id="chat">
</table>
</div>
<script>
function addmessage()
{
    new Ajax.Request( 'add.php', {
        method: 'post',
        parameters: $('chatmessage').serialize(),
        onSuccess: function( transport ) {
            $('messagetext').value = '';
        }
    });
}
</script>
<form id="chatmessage">
<textarea name="message" id="messagetext"></textarea>
</form>
<button onclick="addmessage()">Add</button>
<script>
var lastid = 0;
function getMessages()
{
    new Ajax.Request( 'messages.php?id='+lastid, {
        onSuccess: function( transport ) {
            var messages = transport.responseXML.getElementsByTagName( 'message' );
            for( var i = 0; i < messages.length; i++ )
            {
                var message = messages[i].firstChild.nodeValue;
                var user = messages[i].getAttribute('user');
                var id = parseInt( messages[i].getAttribute('id') );

                if ( id > lastid )
                {
                    var elTR = $('chat').insertRow( -1 );
                    var elTD1 = elTR.insertCell( -1 );
                    elTD1.appendChild( document.createTextNode( user ) );
                    var elTD2 = elTD1.insertCell( -1 );
                    elTD2.appendChild( document.createTextNode( message ) );

                    lastid = id;
                }
            }
            window.setTimeout( getMessages, 1000 );
        }
    });
}
getMessages();
</script>
</body>
</html>
Instead of a "chat" `<div>` tag that held all the messages, you now have a `<table>` tag; you add rows to that tag dynamically for each new message as it comes in. You can see the change in the `getMessages()` function, which has gained a little weight since its first version.

This new version of `getMessages()` expects the results of the messages.php page to be an XML block with the new messages. The messages.php page now also takes a parameter called `id`, which is the `message_id` of the last message the page has seen. The first time out, this ID is 0, so that the messages.php page returns everything it has. After that, it's sent the ID of the last message seen so far.

The XML response is broken up by the `onSuccess` handler, and each element is added to the table using standard DHTML document object model (DOM) functions such as `insertRow()`, `insertCell()`, and `appendChild()`.

The upgraded version of the messages.php file that returns XML instead of HTML is shown in **Listing 7**.

**Listing 7. messages.php**

```php
<?php
// Install the DB module using 'pear install DB'
require_once("DB.php");

header( 'Content-type: text/xml' );

$id = 0;
if ( array_key_exists( 'id', $_GET ) ) { $id = $_GET['id']; }

$db =& DB::Connect( 'mysql://root@localhost/chat', array() );
if (PEAR::isError($db)) { die($db->getMessage()); }

<messages>
    <?php
        $res = $db->query( 'SELECT * FROM messages WHERE message_id > ?', $id );
        while( $res->fetchInto( $row ) )
        {
            ?><message id="<?php echo($row[0]) ?>" user="<?php echo($row[1]) ?>"<?php }
        ?></message>
    <?php
    }?
</messages>
```

**Figure 3** shows the new, enhanced version.
Figure 3. The optimized chat window

![Chat Window](image)

It hasn't changed at all in terms of its look and feel. But it's far more efficient than the original.

**The myth of "real time"

If you're new to Ajax or really just a reasonable programmer who has worked in the field long enough to know better, the idea of "polling" probably sends shivers up your spine. Unfortunately, polling is all you have. There is no cross-platform, cross-browser way to create a continuous pipe between the client and the server without special software installed on both ends of the line. And even then, you might need a special firewall configuration to make it all happen. So, if you want an easy solution that everyone can use, Ajax and polling are what you have.

But what about marketing and the insistence on "real time"? Polling cannot be real time. Or can it? I think it depends on what the definition of the term real time is to you. When I used to write code for electrophysiology data acquisition, real time meant microseconds. I'm sure geologists might happily consider minutes, days, or even years in some cases as real time.

If I look on Wikipedia, I find that a human's average reaction time is somewhere between 200 and 270 milliseconds. And that's just the time to do something like hit a ball. The time to read a message and to start formulating a reply has to be a lot longer, even if you are fully engaged in the conversation. So really, any amount of time in the 200-millisecond range (and probably a little longer) is just fine when waiting for these chat messages. I put it at a second, and it doesn't feel too bad to me.

As the moderator of the Ajax forum on developerWorks (see Related topics), the issue of polling and real time comes up at least once each month. Hopefully, I have debunked the polling problem and the issue of real time when it comes to Ajax. My suggestion is to try polling before you attempt to create some incredibly complex real-time solution. At the very least, know what can be done with off-the-shelf tools before you attempt a custom solution.
Moving on from here

Hopefully, what I have given you here is a good starting point for your own implementation of a chat system within your application. Here are some ideas about where to go next:

- **Track users:** Put a list of the people actively engaged in the conversation alongside the chat. Doing so lets people know who is at the party and when they come and go.
- **Allow multiple conversations:** Allow multiple conversations on different topics to go on simultaneously.
- **Allow emoticons:** Translate character groupings such as :-( into the appropriate image of a smiley face.
- **Use URL parsing:** Use regular expressions in the client-side JavaScript code to find URLs and turn them into hyperlinks.
- **Handle the Enter key:** Instead of having an **Add** button, watch for users pressing the Enter or Return key by hooking onto the `onkeydown` event in the `textarea`.
- **Show when a user is typing:** Alert the server when a user has started to type, so that other participants can see that a reply is pending. This lessens the perception that the conversation has died off to a minimum if you have slow typists.
- **Limit the size of a posted message:** Another way to keep the conversation going is to keep messages small. Limit the maximum number of characters in the `textarea`—again by trapping `onkeydown`—to help speed up conversations.

These are just a few ideas for enhancements that you can make to this code. If you do so and you want to share your changes with the community, please let me know, and I can make them part of the source code available from [Download](#).

**Conclusion**

I admit that I'm really not much of a chatter. I never have my chat client up. I only use text messages once in a long while. My chat handle is idratheryouemail. Seriously. But I have found that contextual chat, such as what is shown in this article, is actually compelling. Why? Because it's focused on the subject matter that the site addresses, which keeps distracting talk about the most recent "TomKat" news to a minimum.

Give this example code a try in your Web application. See whether you can engage your readers and customers in a real-time conversation, and let me know how it goes on the developerWorks Ajax forum site. I hope you will be pleasantly surprised.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code for chat application</td>
<td>x-ajaxxml8-chat.zip</td>
<td>38KB</td>
</tr>
</tbody>
</table>
Related topics

- **PHP home page**: Visit an invaluable resource for PHP programmers.
- **Prototype library**: Explore this JavaScript Framework designed to ease development of dynamic Web applications.
- **Scriptaculous JavaScript library**: Find display helpers and effects to make your Web site fly with this Prototype-based framework.
- **Prototype.js documentation page**: Get additional information on the Prototype JavaScript library with links to the official Prototype blog and many other resources.
- **jQuery**: Explore another JavaScript library that provides similar functionality to Prototype.js.
- **Yahoo! User Interface Library**: Check out Yahoo!’s toolkit for Ajax.
- **DB library**: Get the DB library, available from PEAR.
- **IBM certification**: Find out how you can become an IBM-Certified Developer.
- **XML technical library**: See the developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.