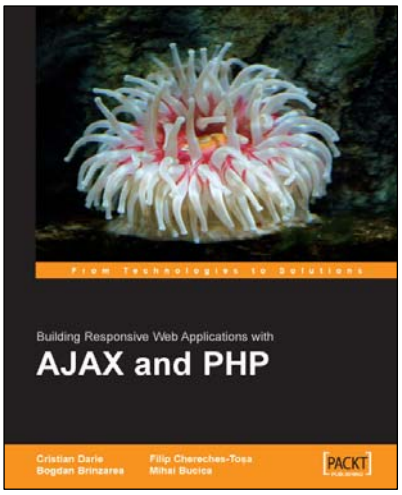
AJAX and PHP

Building Responsive Web Applications

Cristian Darie
Bogdan Brinzarea
Filip Chereches-Toşa
Mihai Bucica



Chapter 5
"AJAX chat and JSON"

In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter 5 "AJAX chat and JSON"

A synopsis of the book's content

Information on where to buy this book

About the Authors

Cristian Darie is a software engineer with experience in a wide range of modern technologies, and the author of numerous technical books, including the popular "Beginning E-Commerce" series. Having worked with computers since he was old enough to press the keyboard, he initially tasted programming success with a first prize in his first programming contest at the age of 12. From there, Cristian moved on to many other similar achievements, and now he is studying distributed application architectures for his PhD degree. He always loves hearing feedback about his books, so don't hesitate dropping a "hello" message when you have a spare moment. Cristian can be contacted through his personal website at `www.cristiandarie.ro`.

Bogdan Brinzarea has a strong background in Computer Science holding a Master and Bachelor Degree at the Automatic Control and Computers Faculty of the Politehnica University of Bucharest, Romania and also an Auditor diploma at the Computer Science department at Ecole Polytechnique, Paris, France.

His main interests cover a wide area from embedded programming, distributed and mobile computing, and new web technologies. Currently, he is employed as an Alternative Channels Specialist at Banca Romaneasca, Member of National Bank of Greece, where he is responsible for the Internet Banking project and coordinates other projects related to security applications and new technologies to be implemented in the banking area.

Filip Cherecheş-Toşa is a web developer with a firm belief in the future of web-based software. He started his career at the age of 9, when he first got a Commodore 64 with tape-drive.

Back home in Romania, Filip runs a web development company named eXigo `www.exigo.ro`, which is actively involved in web-based application development and web design. He is currently a student at the University of Oradea, studying Computer Science, and also an active member of the Romanian PHP Community `www.phpromania.net`.

Mihai Bucica started programming and competing in programming contests (winning many of them), all at age twelve. With a bachelor's degree in computer science from the Automatic Control and Computers Faculty of the Politehnica University of Bucharest, Romania, Bucica works on building communication software with various electronic markets.

Even after working with a multitude of languages and technologies, Bucica's programming language of choice remains C++, and he loves the LGPL word. Mihai also co-authored *Beginning PHP 5 and MySQL E-Commerce* and he can be contacted through his personal website, `www.valentinbucica.ro`.

5

AJAX Chat and JSON

Online chat solutions have been very popular long before AJAX was born. There are numerous reasons for this popularity, and you're probably familiar with them if you've ever used an Internet Relay Chat (IRC) client, or an Instant Messenger (IM) program, or a Java chat applet.

AJAX has pushed online chat solutions forward by making it easy to implement features that are causing trouble or are harder to implement with other technologies. First of all, an AJAX chat application inherits all the typical AJAX benefits, such as integration with existing browser features, and (if written well) cross-platform compatibility.

An additional advantage is that an AJAX chat application avoids the connectivity problems that are common with other technologies, because many firewalls block the communication ports they use. On the other hand, AJAX uses exclusively HTTP for communicating with the server.

Probably the most impressive AJAX chat application available today is **Meebo** (`http://www.meebo.com`). We are pretty sure that some of you have heard about it, and if you haven't, we recommend you have a look at it. The first and the most important feature in Meebo is that it allows you to log in into your favorite IM system by using only a web interface. At the time of writing, Meebo lets you connect to AIM or ICQ, Yahoo! Messenger, Jabber, or GTalk, and MSN. You can access all these services from a single web page with a user friendly interface, with no pop-up windows or Java applets.

Meebo isn't the only web application that offers chat functionality. Even if AJAX is very young, a quick Google search on "AJAX Chat" will reveal several other applications.

It's time to get to work. In the rest of the chapter, we'll implement our own online chat application. We'll use this occasion to learn about **JSON (JavaScript Object Notation)**, which represents an alternative to XML for representing data exchanged between the web browser and the web server.

Introducing JSON

JSON is a data format that you can use instead of XML for exchanging information between the JavaScript client and the PHP server script. Interestingly enough, JSON's popularity increased together with the AJAX phenomenon, although the AJAX acronym implies using XML.

Because XML is a more popular and more widely supported format, we've chosen to use XML for all examples in this book, except this one. However, if you like, it's fairly easy to update the other code samples to use JSON instead of XML. (As you know, this AJAX Chat chapter also has a version that uses XML, and you can compare them to see the differences.)

Perhaps the best short description of JSON is the one proposed by its official website, `http://www.json.org`: "***JSON*** *(JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate.*"

If you're new to JSON, a fair question you could ask would be: *why another data exchange format?* JSON, just as XML, is a text-based format that is easy to write and to understand for both humans and computers. The key word in the definition above is "lightweight." JSON data structures occupies less bandwidth than their XML versions.

To make an idea of how JSON compares to XML, let's take the same data structure and see how we would represent it using both standards. In the Chat application you'll write, the server composes messages for the client that would look like this, if written in XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<response>
  <clear>false</clear>
  <messages>
    <message>
      <id>1</id>
      <color>#000000</color>
      <time>2006-01-17 09:07:31</time>
      <name>Guest550</name>
      <text>Hello there! What's up?</text>
    </message>
    <message>
      <id>2</id>
      <color>#000000</color>
      <time>2006-01-17 09:21:34</time>
      <name>Guest499</name>
      <text>This is a test message</text>
    </message>
  </messages>
</response>
```

The same message, written in JSON this time, looks like this:

```
[
  {"clear":"false"},
  "messages":
    [
      {"message":
        {"id":"1",
         "color":"#000000",
         "time":"2006-01-17 09:07:31",
         "name":"Guest550",
         "text":"Hello there! What's up?"}
      },
      {"message":
        {"id":"2",
         "color":"#000000",
         "time":"2006-01-17 09:21:34",
         "name":"Guest499",
         "text":"This is a test message"}
      }
    ]
  }
]
```

As you can see, they aren't that different. If we disregard the extra formatting spaces that we added for better readability, the XML message occupies 396 bytes while the JSON message has only 274 bytes.

JSON is said to be a subset of JavaScript because it contains two basic structures: the **object**, and the **array**. An object is an unordered collection of name/value pairs, defined in this form:

```
{ name1:value1, name2:value2, ... }
```

The array is an ordered list of values defined in this form:

```
[ value1, value2, ... ]
```

The type of the value can be an object, a string, a number, an array, `true`, `false`, or `null`. A string is a collection of Unicode characters surrounded by double quotes. For escaping, we use the backslash.

For more detailed information please refer to `http://www.json.org`, which covers the theory wonderfully.

Installing JSON

It's obvious that when you plan to use JSON, you need to be able to parse and generate JSON structures in JavaScript and PHP. JSON libraries are available for most of today's programming languages: ActionScript, C, C++, C#, Delphi, E, Erlang, Java, JavaScript, Lisp,Lua, ML and Ruby, Objective CAML, OpenLazslo, Perl, PHP, Python, Rebol, Ruby, and Squeak.

For JavaScript, we'll use the library listed at `http://www.json.org/js.html`. The direct link to the small JSON library is: `http://www.json.org/json.js`. The entire installation process consists in copying this file to your application's folder, and referencing it from the files that need its functionality.

The JSON solution we're using for PHP is the library developed by Michal Migurski that can be downloaded from `http://mike.teczno.com/json.html`. Installing the library implies simply downloading the PHP class file, and copying it into the application's folder. Next we need to reference it from our application by using the `require_once` directive like this:

```
require_once('JSON.php');
```

In order to effectively start using it, we need to instantiate the class:

```
$json= new Services_JSON();
```

Then you're ready to use it. The `encode` and `decode` methods allow us to encode a PHP object into JSON format and to decode a JSON string into a PHP object.

Using JSON with JavaScript and PHP

A typical way to build a JSON structure in JavaScript:

```
var myMessages =
  {'messages' : [
    {'username': 'Guest0740',
     'message': 'This is a JSON message',
     'color': '#eeeeee'},
    {'username': 'Guest0740',
     'message': 'This is another JSON message',
     'color': '#eeeeee'} ]};
alert(myMessages.toJSONString());
```

The first command defines an object called `myMessages`, which contains a single member called `messages`, which at its turn contains an array containing two objects that are made of three members (`username`, `message`, and `color`). In order to access the "This is another JSON message" text, you could use the following syntax:

```
myMessages[0].messages[1].message
```

The following two code snippets show typical ways to read a JSON structures in JavaScript:

```
myMessagesJSON = myMessages.toJSONString();
myMessagesFromJSON = myMessagesJSON.parseJSON();
```

Or

```
myMessagesFromJSON = eval ('(' + myMessagesJSON + ') ');
```

The `eval` function is very fast but it can allow any JavaScript code to be executed, so it's generally much safer to use the `parseJSON` method.

In PHP, you encode and decode JSON messages using the `encode` and `decode` methods. Let's take the previous example and do something similar in PHP. A typical way to build a JSON structure in PHP:

```
require_once('JSON.php');
$json = new Services_JSON();
$myMessages = array ('messages' =>
  array(
    array ('username' => 'Guest0740',
           'message' => 'This is a JSON message',
           'color' => '#eeeeee'),
    array('username' => 'Guest0740',
          'message' => 'This is another JSON message',
          'color' => '#eeeeee')));
echo $json->encode($myMessages);
```

A typical way to read a JSON structure in PHP is:

```
require_once('JSON.php');
$json = new Services_JSON();
$myMessagesFromJSON = $json->decode($myMessagesJSON);
```

Implementing AJAX Chat

Now, it's time to implement the AJAX chat application. We'll keep the application simple, modular, and extensible. We won't implement a login module, support for chat rooms, the online users list, etc. By keeping it simple we try to focus on what the goal of this chapter is: posting and retrieving messages without causing any page reloads. We'll also let the user pick a color for her or his messages, because this involves an AJAX mechanism that will be another good exercise.

The chat application can be tested online at `http://ajaxphp.packtpub.com`, and it looks like in Figure 5.1.

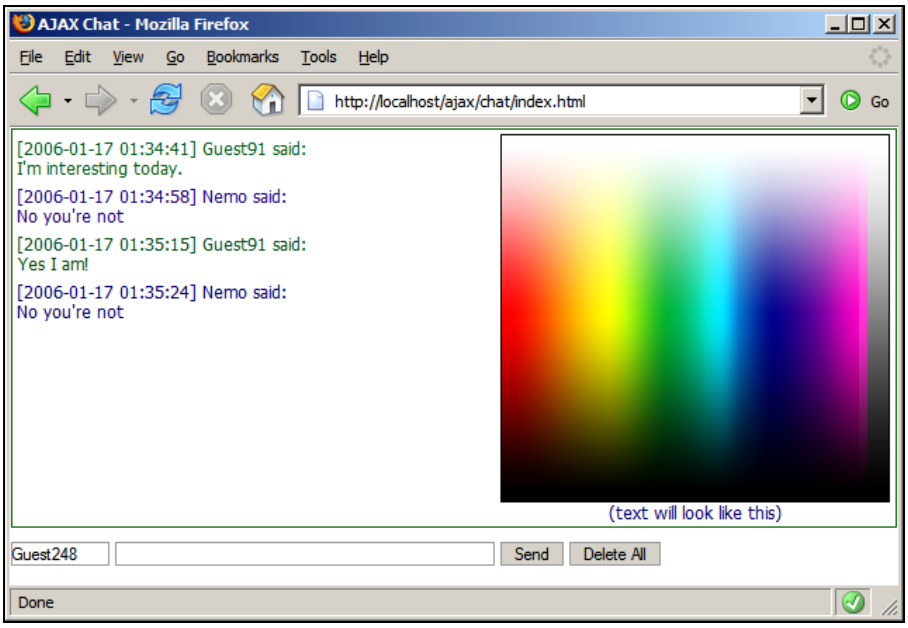


Figure 5.1: AJAX Chat

A novelty in this chapter is that you will have two `XMLHttpRequest` objects. The first one will handle updating the chat window, and the second will handle the color picker (when you click on the image, the coordinates are sent to the server, and the server replies with the color code).

The messages for the AJAX Chat are saved in a queue (a FIFO structure), whose functionality was covered in Chapter 4, so that messages are not lost even if the server is slow, and they always get to the server in the same order as you sent them. Unlike with other patterns you can find on Internet these days, we also ensure we don't load the server with any more requests until the current one is finished.

> In order to have this example working you need the **GD library**. The installation instructions in Appendix A include support for the GD library.

Time for Action—Ajax Chat

1. Connect to the `ajax` database, and create a table named `chat` with the following code:

```
CREATE TABLE chat
(
  chat_id int(11) NOT NULL auto_increment,
  posted_on datetime NOT NULL,
  user_name varchar(255) NOT NULL,
  message text NOT NULL,
  color char(7) default '#000000',
  PRIMARY KEY (chat_id)
);
```

2. In your `ajax` folder, create a new folder named `chat`.
3. Copy the `palette.png` file from the code download to the `chat` folder.
4. We will create the application starting with the server functionality. In the `chat` folder, create a file named `config.php`, and add the database configuration code to it (change these values to match your configuration):

```
<?php
// defines database connection data
define('DB_HOST', 'localhost');
define('DB_USER', 'ajaxuser');
define('DB_PASSWORD', 'practical');
define('DB_DATABASE', 'ajax');
?>
```

5. Now add the standard error handling file, `error_handler.php`:

```
<?php
// set the user error handler method to be error_handler
set_error_handler('error_handler', E_ALL);
// error handler function
function error_handler($errNo, $errStr, $errFile, $errLine)
{
  // clear any output that has already been generated
  if(ob_get_length()) ob_clean();
  // output the error message
  $error_message = 'ERRNO: ' . $errNo . chr(10) .
                   'TEXT: ' . $errStr . chr(10) .
                   'LOCATION: ' . $errFile .
                   ', line ' . $errLine;
  echo $error_message;
  // prevent processing any more PHP scripts
  exit;
}
?>
```

6. Create another file named `chat.php` and add this code to it:

```
<?php
// reference the file containing the Chat class
require_once('chat.class.php');
// reference the file containing the JSON class
require_once('JSON.php');
// create an instance of the JSON class
$json=new Services_JSON();
// retrieve the operation to be performed
$mode = $_POST['mode'];
// default the last id to 0
$id = 0;
// create a new Chat instance
$chat = new Chat();
// if the operation is SendAndRetrieve
if($mode == 'SendAndRetrieveNew')
{
  // retrieve the action parameters used to add a new message
  $name = $_POST['name'];
  $message = $_POST['message'];
  $color = $_POST['color'];
  $id = $_POST['id'];
  // check if we have valid values
  if ($name != '' && $message != '' && $color != '')
  {
    // post the message to the database
    $chat->postMessage($name, $message, $color);
```

```
  }
}
// if the operation is DeleteAndRetrieve
elseif($mode == 'DeleteAndRetrieveNew')
{
  // delete all existing messages
  $chat->deleteMessages();
}
// if the operation is Retrieve
elseif($mode == 'RetrieveNew')
{
  // get the id of the last message retrieved by the client
  $id = $_POST['id'];
}
// Clear the output
if(ob_get_length()) ob_clean();
// Headers are sent to prevent browsers from caching
header('Expires: Mon, 26 Jul 1997 05:00:00 GMT');
header('Last-Modified: ' . gmdate('D, d M Y H:i:s') . 'GMT');
header('Cache-Control: no-cache, must-revalidate');
header('Pragma: no-cache');
header('Content-Type: text/javascript');
// retrieve new messages from the server and send them in JSON format
echo $json->encode($chat->retrieveNewMessages($id));
?>
```

7. Create another file named `chat.class.php`, and add this code to it:

```
<?php
// load configuration file
require_once('config.php');
// load error handling module
require_once('error_handler.php');
// class that contains server-side chat functionality
class Chat
{
  // database handler
  private $mMysqli;
  // constructor opens database connection
  function __construct()
  {
    // connect to the database
    $this->mMysqli = new mysqli(DB_HOST, DB_USER, DB_PASSWORD,
                                                    DB_DATABASE);
  }
  // destructor closes database connection
  public function __destruct()
  {
    $this->mMysqli->close();
  }
  // truncates the table containing the messages
  public function deleteMessages()
  {
    // build the SQL query that adds a new message to the server
    $query = 'TRUNCATE TABLE chat';
    // execute the SQL query
    $result = $this->mMysqli->query($query);
  }
  /*
   The postMessages method inserts a message into the database
   - $name represents the name of the user that posted the message
   - $messsage is the posted message
   - $color contains the color chosen by the user
  */
  public function postMessage($name, $message, $color)
  {
```

```
    // escape the variable data for safely adding them to the database
    $name = $this->mMysqli->real_escape_string($name);
    $message = $this->mMysqli->real_escape_string($message);
    $color = $this->mMysqli->real_escape_string($color);
    // build the SQL query that adds a new message to the server
    $query = 'INSERT INTO chat(posted_on, user_name, message, color) ' .
             'VALUES (NOW(), "' . $name . '" , "' . $message .
             '","' . $color . '")';
    // execute the SQL query
    $result = $this->mMysqli->query($query);
  }
  /*
   The retrieveNewMessages method retrieves the new messages that have
   been posted to the server.
   - the $id parameter is sent by the client and it
   represents the id of the last message received by the client. Messages
   more recent by $id will be fetched from the database and returned to
   the client in JSON format.
  */
  public function retrieveNewMessages($id=0)
  {
    // escape the variable data
    $id = $this->mMysqli->real_escape_string($id);
    // compose the SQL query that retrieves new messages
    if($id>0)
    {
      // retrieve messages newer than $id
      $query =
      'SELECT chat_id, user_name, message, color, ' .
      '       DATE_FORMAT(posted_on, "%Y-%m-%d %H:%i:%s") ' .
      '       AS posted_on ' .
      ' FROM chat WHERE chat_id > ' . $id .
      ' ORDER BY chat_id ASC';
    }
    else
    {
      // on the first load only retrieve the last 50 messages from server
      $query =
      ' SELECT chat_id, user_name, message, color, posted_on FROM ' .
      '    (SELECT chat_id, user_name, message, color, ' .
      '       DATE_FORMAT(posted_on, "%Y-%m-%d %H:%i:%s") AS posted_on ' .
      '     FROM chat ' .
      '     ORDER BY chat_id DESC ' .
      '      LIMIT 50) AS Last50' .
      ' ORDER BY chat_id ASC';
    }
    // execute the query
    $result = $this->mMysqli->query($query);
    // initialize the response array
    $response = array();
    // add the clear flag to the response
    array_push( $response,array('clear'=> $this->isDatabaseCleared($id)));
    // initialize the results array
    $results=array();
    // check to see if we have any results
    if($result->num_rows)
    {
      // loop through all the fetched messages to build the results array
      while ($row = $result->fetch_array(MYSQLI_ASSOC))
      {
        $id = htmlspecialchars($row['chat_id']);
        $color = htmlspecialchars($row['color']);
        $userName = htmlspecialchars($row['user_name']);
        $time = htmlspecialchars($row['posted_on']);
        $message = htmlspecialchars($row['message']);
        array_push($results,array('id' => $id ,
```

```
                    'color' => $color ,
                    'time' => $time ,
                    'name' => $userName ,
                    'message' => $message ));
        }
        // close the database connection as soon as possible
        $result->close();
      }
      // add the results to the response
      array_push($response, array('results' => $results));
      return $response;
    }
    /*
      The isDatabaseCleared method checks to see if the database has been
      cleared since last call to the server
      -   the $id parameter contains the id of the last message received by
      the client
    */
    private function isDatabaseCleared($id)
    {
      if($id>0)
      {
        // by checking the number of rows with ids smaller than the client's
        // last id we check to see if a truncate operation was performed in
        // the meantime
        $check_clear = 'SELECT count(*) old FROM chat where chat_id<=' . $id;
        $result = $this->mMysqli->query($check_clear);
        $row = $result->fetch_array(MYSQLI_ASSOC);
        // if a truncate operation occured the whiteboard needs to be reset
        if($row['old']==0)
          return 'true';
      }
      return 'false';
    }
}
?>
```

8. Create another file named `get_color.php` and add this code to it:

```
<?php
// the name of the image file
$imgfile='palette.png';
// load the image file
$img=imagecreatefrompng($imgfile);
// obtain the coordinates of the point clicked by the user
$offsetx=$_GET['offsetx'];
$offsety=$_GET['offsety'];
// get the clicked color
$rgb = ImageColorAt($img, $offsetx, $offsety);
$r = ($rgb >> 16) & 0xFF;
$g = ($rgb >> 8) & 0xFF;
$b = $rgb & 0xFF;
// return the color code
printf('#%02s%02s%02s', dechex($r), dechex($g), dechex($b));
?>
```

9. Let's deal with the client now. Start by creating `chat.css` and adding this code to it:

```
body
{
  font-family: Tahoma, Helvetica, sans-serif;
  margin: 1px;
  font-size: 12px;
  text-align: left
}
```

```
#content
{
  border: DarkGreen 1px solid;
  margin-bottom: 10px
}
input
{
  border: #999 1px solid;
  font-size: 10px
}
#scroll
{
  position: relative;
  width: 340px;
  height: 270px;
  overflow: auto
}
.item
{
  margin-bottom: 6px
}
#colorpicker
{
  text-align:center
}
```

10. Create a new file named `index.html`, and add this code to it:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <title>AJAX Chat</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <link href="chat.css" rel="stylesheet" type="text/css" />
  <script type="text/javascript" src="json.js" ></script>
  <script type="text/javascript" src="chat.js" ></script>
</head>
  <body onload="init();">
    <noscript>
      Your browser does not support JavaScript!!
    </noscript>
    <table id="content">
      <tr>
        <td>
          <div id="scroll">
          </div>
        </td>
        <td id="colorpicker">
          <img src="palette.png" id="palette" alt="Color
               Palette" border="1" onclick="getColor(event);"/>
          <br />
          <input id="color" type="hidden" readonly="true" value="#000000"
/>
          <span id="sampleText">
            (text will look like this)
          </span>
        </td>
      </tr>
    </table>
    <div>
      <input type="text" id="userName" maxlength="10" size="10"
onblur="checkUsername();"/>
      <input type="text" id="messageBox" maxlength="2000" size="50"
             onkeydown="handleKey(event)"/>
```

```
      <input type="button" value="Send" onclick="sendMessage();" />
      <input type="button" value="Delete All" onclick="deleteMessages();"
/>
    </div>
  </body>
</html>
```

11. Create another file named `chat.js` and add this code to it:

```
/* chatURL - URL for updating chat messages */
var chatURL = "chat.php";
/* getColorURL - URL for retrieving the chosen RGB color */
var getColorURL = "get_color.php";
/* create XMLHttpRequest objects for updating the chat messages and
getting the selected color */
var xmlHttpGetMessages = createXmlHttpRequestObject();
var xmlHttpGetColor = createXmlHttpRequestObject();
/* variables that establish how often to access the server */
var updateInterval = 1000; // how many miliseconds to wait to get new
message
// when set to true, display detailed error messages
var debugMode = true;
/* initialize the messages cache */
var cache = new Array();
/* lastMessageID - the ID of the most recent chat message */
var lastMessageID = -1;
/* mouseX, mouseY - the event's mouse coordinates */
var mouseX,mouseY;
/* creates an XMLHttpRequest instance */
function createXmlHttpRequestObject()
{
  // will store the reference to the XMLHttpRequest object
  var xmlHttp;
  // this should work for all browsers except IE6 and older
  try
  {
    // try to create XMLHttpRequest object
    xmlHttp = new XMLHttpRequest();
  }
  catch(e)
  {
    // assume IE6 or older
    var XmlHttpVersions = new Array("MSXML2.XMLHTTP.6.0",
                                    "MSXML2.XMLHTTP.5.0",
                                    "MSXML2.XMLHTTP.4.0",
                                    "MSXML2.XMLHTTP.3.0",
                                    "MSXML2.XMLHTTP",
                                    "Microsoft.XMLHTTP");
    // try every prog id until one works
    for (var i=0; i<XmlHttpVersions.length && !xmlHttp; i++)
    {
      try
      {
        // try to create XMLHttpRequest object
        xmlHttp = new ActiveXObject(XmlHttpVersions[i]);
      }
      catch (e) {}
    }
  }
  // return the created object or display an error message
  if (!xmlHttp)
    alert("Error creating the XMLHttpRequest object.");
  else
    return xmlHttp;
}
```

```
/* this function initiates the chat; it executes when the chat page loads
*/
function init()
{
  // get a reference to the text box where the user writes new messages
  var oMessageBox = document.getElementById("messageBox");
  // prevents the autofill function from starting
  oMessageBox.setAttribute("autocomplete", "off");
  // references the "Text will look like this" message
  var oSampleText = document.getElementById("sampleText");
  // set the default color to black
  oSampleText.style.color = "black";
  // ensures our user has a default random name when the form loads
  checkUsername();
  // initiates updating the chat window
  requestNewMessages();
}
// function that ensures that the username is never empty and if so
// a random name is generated
function checkUsername()
{
  // ensures our user has a default random name when the form loads
  var oUser=document.getElementById("userName");
  if(oUser.value == "")
    oUser.value = "Guest" + Math.floor(Math.random() * 1000);
}
/* function called when the Send button is pressed */
function sendMessage()
{
  // save the message to a local variable and clear the text box
  var oCurrentMessage = document.getElementById("messageBox");
  var currentUser = document.getElementById("userName").value;
  var currentColor = document.getElementById("color").value;
  // don't send void messages
  if (trim(oCurrentMessage.value) != "" &&
      trim(currentUser) != "" && trim (currentColor) != "")
  {
    // if we need to send and retrieve messages
    params =  "mode=SendAndRetrieveNew" +
              "&id=" + encodeURIComponent(lastMessageID) +
              "&color=" + encodeURIComponent(currentColor) +
              "&name=" + encodeURIComponent(currentUser) +
              "&message=" + encodeURIComponent(oCurrentMessage.value);
    // add the message to the queue
    cache.push(params);
    // clear the text box
    oCurrentMessage.value = "";
  }
}
/* function called when the Delete Messages button is pressed */
function deleteMessages()
{
  // set the flag that specifies we're deleting the messages
  params = "mode=DeleteAndRetrieveNew";
  // add the message to the queue
  cache.push(params);
}
/* makes asynchronous request to retrieve new messages, post new messages,
delete messages */
function requestNewMessages()
{
  // retrieve the username and color from the page
  var currentUser = document.getElementById("userName").value;
  var currentColor = document.getElementById("color").value;
  // only continue if xmlHttpGetMessages isn't void
  if(xmlHttpGetMessages)
```

```
  {
    try
    {
      // don't start another server operation if such an operation
      //   is already in progress
      if (xmlHttpGetMessages.readyState == 4 ||
          xmlHttpGetMessages.readyState == 0)
      {
        // we will store the parameters used to make the server request
        var params = "";
        // if there are requests stored in queue, take the oldest one
        if (cache.length>0)
          params = cache.shift();
        // if the cache is empty, just retrieve new messages
        else
          params = "mode=RetrieveNew" +
                   "&id=" +lastMessageID;
        // call the server page to execute the server-side operation
        xmlHttpGetMessages.open("POST", chatURL, true);
        xmlHttpGetMessages.setRequestHeader("Content-Type",
                                    "application/x-www-form-urlencoded");
        xmlHttpGetMessages.onreadystatechange = handleReceivingMessages;
        xmlHttpGetMessages.send(params);
      }
      else
      {
        // we will check again for new messages
        setTimeout("requestNewMessages();", updateInterval);
      }
    }
    catch(e)
    {
      displayError(e.toString());
    }
  }
}
/* function that handles the http response when updating messages */
function handleReceivingMessages()
{
  // continue if the process is completed
  if (xmlHttpGetMessages.readyState == 4)
  {
    // continue only if HTTP status is "OK"
    if (xmlHttpGetMessages.status == 200)
    {
      try
      {
        // process the server's response
        readMessages();
      }
      catch(e)
      {
        // display the error message
        displayError(e.toString());
      }
    }
    else
    {
      // display the error message
      displayError(xmlHttpGetMessages.statusText);
    }
  }
}
/* function that processes the server's response when updating messages */
function readMessages()
{
```

```
    // retrieve the server's response
    var response = xmlHttpGetMessages.responseText;
    // server error?
    if (response.indexOf("ERRNO") >= 0
        || response.indexOf("error:") >= 0
        || response.length == 0)
      throw(response.length == 0 ? "Void server response." : response);
    // retrieve the JSON object correspondig to the responseText element
    responseJSON = xmlHttpGetMessages.responseText.parseJSON();
    // retrieve the flag that says if the chat window has been cleared
    clearChat = responseJSON[0].clear;
    // if the clear flag is set, we need to clear the chat window
    if(clearChat == "true")
    {
      // clear chat window and reset the id
      document.getElementById("scroll").innerHTML = "";
      lastMessageID = -1;
    }
    // initialize the arrays
    idArray = new Array();
    colorArray = new Array();
    nameArray = new Array();
    timeArray = new Array();
    messageArray = new Array();
    // retrieve the arrays
    for(i=0;i<responseJSON[1].results.length;i++)
    {
      // retrieve the arrays from the server's response
      idArray[i] = responseJSON[1].results[i].id;
      colorArray[i] = responseJSON[1].results[i].color;
      nameArray[i] = responseJSON[1].results[i].name;
      timeArray[i] = responseJSON[1].results[i].time;
      messageArray[i] = responseJSON[1].results[i].message;
    }
    // add the new messages to the chat window
    displayMessages(idArray, colorArray,
                    nameArray, timeArray, messageArray);
    // the ID of the last received message is stored locally
    if(idArray.length>0)
      lastMessageID = idArray[idArray.length - 1];  // restart sequence
    setTimeout("requestNewMessages();", updateInterval);
}
/* function that appends the new messages to the chat list  */
function displayMessages(idArray, colorArray, nameArray,
                         timeArray, messageArray)
{
  // each loop adds a new message
  for(var i=0; i<idArray.length; i++)
  {
    // get the message details
    var color = colorArray[i];
    var time = timeArray[i];
    var name = nameArray[i];
    var message = messageArray[i];
    // compose the HTML code that displays the message
    var htmlMessage = "";
    htmlMessage += "<div class=\"item\" style=\"color:" + color + "\">";
    htmlMessage += "[" + time + "] " + name + " said: <br/>";
    htmlMessage += message.toString();
    htmlMessage += "</div>";
    // display the message
    displayMessage (htmlMessage);
  }
}
// displays a message
function displayMessage(message)
```

```
{
  // get the scroll object
  var oScroll = document.getElementById("scroll");
  // check if the scroll is down
  var scrollDown = (oScroll.scrollHeight - oScroll.scrollTop <=
                    oScroll.offsetHeight );
  // display the message
  oScroll.innerHTML += message;
  // scroll down the scrollbar
  oScroll.scrollTop = scrollDown ? oScroll.scrollHeight :
oScroll.scrollTop;
}
// function that displays an error message
function displayError(message)
{
  // display error message, with more technical details if debugMode is true
  displayMessage("Error accessing the server! "+
                 (debugMode ? "<br/>" + message : ""));
}
/* handles keydown to detect when enter is pressed */
function handleKey(e)
{
  // get the event
  e = (!e) ? window.event : e;
  // get the code of the character that has been pressed
  code = (e.charCode) ? e.charCode :
         ((e.keyCode) ? e.keyCode :
         ((e.which) ? e.which : 0));
  // handle the keydown event
  if (e.type == "keydown")
  {
    // if enter (code 13) is pressed
    if(code == 13)
    {
      // send the current message
      sendMessage();
    }
  }
}
/* removes leading and trailing spaces from the string */
function trim(s)
{
    return s.replace(/(^\s+)|(\s+$)/g, "")
}
/* function that computes the mouse's coordinates in page */
function getMouseXY(e)
{
  // browser specific
  if(window.ActiveXObject)
  {
    mouseX = window.event.x + document.body.scrollLeft;
    mouseY = window.event.y + document.body.scrollTop;
  }
  else
  {
    mouseX = e.pageX;
    mouseY = e.pageY;
  }
}
/* makes a server call to get the RGB code of the chosen color */
function getColor(e)
{
  getMouseXY(e);
  // don't do anything if the XMLHttpRequest object is null
  if(xmlHttpGetColor)
  {
```

```
      // initialize the offset position with the mouse current position
      var offsetX = mouseX;
      var offsetY = mouseY;
      // get references
      var oPalette = document.getElementById("palette");
      var oTd = document.getElementById("colorpicker");
      // compute the offset position in our window
      if(window.ActiveXObject)
      {
        offsetX = window.event.offsetX;
        offsetY = window.event.offsetY;
      }
      else
      {
        offsetX -= oPalette.offsetLeft + oTd.offsetLeft;
        offsetY -= oPalette.offsetTop + oTd.offsetTop;
      }
      // call server asynchronously to find out the clicked color
      try
      {
        if (xmlHttpGetColor.readyState == 4 ||
            xmlHttpGetColor.readyState == 0)
        {
          params = "?offsetx=" + offsetX + "&offsety=" + offsetY;
          xmlHttpGetColor.open("GET", getColorURL+params, true);
          xmlHttpGetColor.onreadystatechange = handleGettingColor;
          xmlHttpGetColor.send(null);
        }
      }
      catch(e)
      {
        // display error message
        displayError(xmlHttp.statusText);
      }
    }
  }
  /* function that handles the http response */
  function handleGettingColor()
  {
    // if the process is completed, decide to do with the returned data
    if (xmlHttpGetColor.readyState == 4)
    {
      // only if HTTP status is "OK"
      if (xmlHttpGetColor.status == 200)
      {
        try
        {
          //change the color
          changeColor();
        }
        catch(e)
        {
          // display error message
          displayError(xmlHttpGetColor.statusText);
        }
      }
      else
      {
        // display error message
        displayError(xmlHttpGetColor.statusText);
      }
    }
  }
  /* function that changes the color used for displaying messages */
  function changeColor()
  {
```

```
    response=xmlHttpGetColor.responseText;
    // server error?
    if (response.indexOf("ERRNO") >= 0
        || response.indexOf("error:") >= 0
        || response.length == 0)
      throw(response.length == 0 ? "Can't change color!" : response);
    // change color
    var oColor = document.getElementById("color");
    var oSampleText = document.getElementById("sampleText");
    oColor.value = response;
    oSampleText.style.color = response;
  }
```

12. If you haven't already done so, copy `json.js` from `http://www.json.org/json.js` to your project's folder.

What Just Happened?

First, make sure the application works well. Load `http://localhost/ajax/chat/index.html` with a web browser, and you should get a page that looks like the one in Figure 5.1.

Technically, the application is split in two smaller applications that build the final solution:

- **The chat application**: Here we use AJAX for passing the messages between the client and the server. The chat window contacts the server periodically to send any messages that have been typed by the user, and retrieve the newly posted messages from the server.
- **The code for choosing a color**: Here we use AJAX for calling PHP script that can tell us the color that was clicked by the user on the color image. We use a `palette` containing the entire spectrum of colors to allow the user pick a color for the text he or she writes. When clicking on the palette, the mouse coordinates are sent to the server, which obtains the color code.

If you analyze the code for a bit, the details will become clear. Everything starts with `index.html`. The only part that is really interesting in `index.html` is a scroll region that can be implemented in DHTML. A little piece of information regarding scrolling can be found at `http://www.dyn-web.com/dhtml/scroll/`.

Basically, the idea for having a part of the page with a scrollbar next to it is to use nested layers. In our example, the `div scroll` and its inner layers do the trick. The outer layer is `scroll`. It has a fixed `width` and `height` and the most useful property of it is `overflow`. Generally, the content of a block box is confined to the content edges of the box. In certain cases, a box may overflow, meaning its content lies partly or entirely outside of the box. In CSS, this property specifies what happens when an element overflows its area. For more details, please see `overflow`'s specification, at `http://www.w3.org/TR/REC-CSS2/visufx.html`.

The `chat.js` script contains the JavaScript part for our application. This file can be divided in two parts: the one that handles choosing a color and the other that handles retrieving and sending chat messages.

We will start with the color choosing functionality. This part, which in the beginning might seem pretty difficult, proves to be easy to implement. Let's have a panoramic view of the entire process.

We have a palette image that contains the entire spectrum of visible colors. PHP has two functions that will help us in finding the RGB code of the chosen color: `imagecreatefrompng` and `imagecolorat`. These two functions allow us to obtain the RGB code of a pixel given the x and y position in the image. The position of the pixel is retrieved in the `getMouseXY` function in the JavaScript code.

The `getColor` function retrieves the RGB code of the color chosen by the user when clicking the palette image. First of all it retrieves the mouse coordinates from the event. Then, it computes the coordinates where the click event has been produced as relative values within the image. Using this information, `getColor` initiates an asynchronous request to the `get_color.php` script, which is supposed to return the color code associated with the clicked coordinate. The callback function `handleGettingColor` is executed when the state of request is changed.

The `handleGettingColor` function checks to see when the request to the server is completed and if no errors occurred, the `changeColor` function is called. This function changes the color of the sample text "text will look like this" with the given code.

Now, let's now see how the chat works. By default when the page initializes and the `onblur` event occurs, the `checkUsername` function is called. This function ensures that the name of the user isn't empty by generating an arbitrary username.

On pressing the `Send` button, the `sendMessage` function is called. This function adds the current message to the message queue to be sent to the server. Before adding it into the queue the function trims the message by calling the `trim` function, and we encode the message using `encodeURIComponent` to make sure it gets through successfully.

The `handleKey` function is called whenever a `keydown` event occurs. When the Enter key is pressed the `sendMessage` function is called so that both pressing the `Send` button and pressing Enter within the `messageBox` control have the same effect.

The `deleteMessages` function adds the delete message to the messages to be sent to the server.

The `requestNewMessages` function is responsible for sending chat messages. It retrieves a message from the queue and sends it to the server. The change of state of the HTTP request object is handled by the `handleReceivingMessages` function.

The `handleReceivingMessages` checks to see when the request to the server is completed and if no errors occurred then the `readMessages` function is called.

The `readMessages` function checks to see if someone else erased all the chat messages and if so the client's chat window is also emptied. In order to append new messages to the chat, we call the `displayMessages` function. This function takes as parameters the arrays that correspond to the new messages. It composes the new messages as HTML and it appends them to those already in the chat by calling the `displayMessage` function. In the beginning, the `displayMessage` function checks to see if the scroll bar is at the bottom of the list of messages. This is necessary in order to reposition it at the end of the function so that the focus is now on the last new messages.

The last function presented is the `init` function. Its role is to retrieve the chat messages, to ensure that the username is not null, to set the text's color to black, and to turn off the auto complete functionality.

For the error handling part, we use the `displayError` function, which calls the `displayMessage` function in turn with the error message as parameter.

Let's move on to the server side of the application by first presenting the `chat.php` file. The server deals with clients' requests like this:

- Retrieves the client's parameters
- Identifies the operations that need to be performed
- Performs the necessary operations
- Sends the results back to the client

The request includes the `mode` parameter that specifies one of the following operations to be performed by the server:

- `SendAndRetrieve`: First the new messages are inserted in the database and then all new messages are retrieved and sent back to the client.
- `DeleteAndRetrieve`: All messages are erased and the new messages that might exist are fetched and sent back to the client.
- `Retrieve`: The new messages are fetched and sent back to the client.

The business logic behind `chat.php` lies in the `chat.class.php` script, which contains the `Chat` class.

The `deleteMessages` method truncates the data table erasing all the information.

The `postMessages` method inserts all the new messages into the database.

The `isDatabaseCleared` method checks to see if all messages have been erased. Basically, by providing the ID of the last message retrieved from the server and by checking if it still exists, we can detect if all messages have been erased.

The `retrieveNewMessages` method gets all new messages since the last message (identified by its `id`) retrieved from the server during the last request (if a last request exists; or all messages in other cases) and also checks to see if the database has been emptied by calling the `isDatabaseCleared` method. This function composes the response for the client and sends it.

The `config.php` file contains the database configuration parameters and the `error_handler.php` file contains the module for handling errors.

Summary

At the beginning of the chapter we saw why one can face problems when communicating with other people in a dynamic way over the Internet. We saw what the solutions for these problems are and how AJAX chat solutions can bring something new, useful, and ergonomic. After seeing some other AJAX chat implementations, we started building our own solution. Step by step we have implemented our AJAX chat solution keeping it simple, easily extensible, and modular.

After reading this chapter, you can try improving the solution, by adding new features like:

- Chat rooms
- Simple command lines (joining/leaving a chat room, switching between chat room)
- Private messaging

AJAX and PHP: Building Responsive Web Applications

AJAX is a complex phenomenon that means different things to different people. Computer users appreciate that their favorite websites are now friendlier and feel more responsive. Web developers learn new skills that empower them to create sleek web applications with little effort. Indeed, everything sounds good about AJAX!

At its roots, AJAX is a mix of technologies that lets you get rid of the evil page reload, which represents the dead time when navigating from one page to another. Eliminating page reloads is just one step away from enabling more complex features into websites, such as real-time data validation, drag and drop, and other tasks that weren't traditionally associated with web applications. Although the AJAX ingredients are mature (the `XMLHttpRequest` object, which is the heart of AJAX, was created by Microsoft in 1999), their new role in the new wave of web trends is very young, and we'll witness a number of changes before these technologies will be properly used to the best benefit of the end users. At the time of writing this book, the "AJAX" name is about just one year old.

AJAX isn't, of course, the answer to all the Web's problems, as the current hype around it may suggest. As with any other technology, AJAX can be overused, or used the wrong way. AJAX also comes with problems of its own: you need to fight with browser inconsistencies, AJAX-specific pages don't work on browsers without JavaScript, they can't be easily bookmarked by users, and search engines don't always know how to parse them. Also, not everyone likes AJAX. While some are developing enterprise architectures using JavaScript, others prefer not to use it at all. When the hype is over, most will probably agree that the middle way is the wisest way to go for most scenarios.

In *AJAX and PHP: Building Responsive Web Applications*, we took a pragmatic and safe approach by teaching relevant patterns and best practices that we think any web developer will need sooner or later. We teach you how to avoid the common pitfalls, how to write efficient AJAX code, and how to achieve functionality that is easy to integrate into current and future web applications, without requiring you to rebuild the whole solution around AJAX. You'll be able to use the knowledge you learn from this book right away, into your PHP web applications.

We hope you'll find this book useful and relevant to your projects. For the latest details and updates regarding this book, please visit its mini-site at `http://ajaxphp.packtpub.com`.

The book's mini-site also contains additional free chapters and resources, which we recommend you check out when you have the time.

What This Book Covers

Chapter 1: *AJAX and the Future of Web Applications* is an initial incursion into the world of AJAX and the vast possibilities it opens up for web developers and companies, to offer a better experience to their users. In this chapter you'll also build your first AJAX-enabled web page, which will give you a first look of the component technologies.

Chapter 2: *Client-Side Techniques with Smarter JavaScript* will guide you through the technologies you'll use to build AJAX web clients, using JavaScript, the DOM, the `XMLHttpRequest` object, and XML. While not being a complete tutorial for these technologies, you'll be put on the right track for using them together to build a solid foundation for your future applications.

Chapter 3: *Server-Side Techniques with PHP and MySQL* completes the theoretical foundation by presenting how to create smart servers to interact with your AJAX client. You'll learn various techniques for implementing common tasks, including handling basic JavaScript security and error-handling problems.

Chapter 4: *AJAX Form Validation* guides you through creating a modern, responsive, and secure form validation system that implements both real-time AJAX validation and server-side validation on form submission.

Chapter 5: *AJAX Chat* presents a simple online chat that works exclusively using AJAX code, without using Java applets, Flash code, or other specialized libraries as most chat applications do these days.

Chapter 6: *AJAX Suggest and Autocomplete* builds a Google Suggest-like feature, that helps you quickly find PHP functions, and forwards you to the official help page for the chosen function.

Chapter 7: *AJAX Real-Time Charting with SVG* teaches you how to implement a real-time charting solution with AJAX and SVG. SVG (Scalable Vector Graphics) is a text-based graphics language that can be used to draw shapes and text.

Chapter 8: *AJAX Grid* teaches you how to build powerful AJAX-enabled data grids. You'll learn how to parse XML documents using XSLT to generate the output of your grid.

Chapter 9: *AJAX RSS Reader* uses the SimpleXML PHP library, XML, and XSLT to build a simple RSS aggregator.

Chapter 10: *AJAX Drag and Drop* is a demonstration of using the script.aculo.us framework to build a simple list of elements with drag-and-drop functionality.

Appendix A: *Preparing Your Working Environment* teaches you how to install and configure the required software: Apache, PHP, MySQL, phpMyAdmin. The examples in this book assume that you have set up your environment and sample database as shown here.

At the book's mini-site at `http://ajaxphp.packtpub.com`, you can find the online demos for all the book's AJAX case studies.

Where to buy this book

You can buy *AJAX and PHP: Building Responsive Web Applications* from the Packt Publishing website: `http://www.packtpub.com/ajax_php/book`.

Free shipping to the US, UK, Europe, Australia, New Zealand and India.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com