JAVASCRIPT

Build an Advanced “Poll” jQuery Plugin

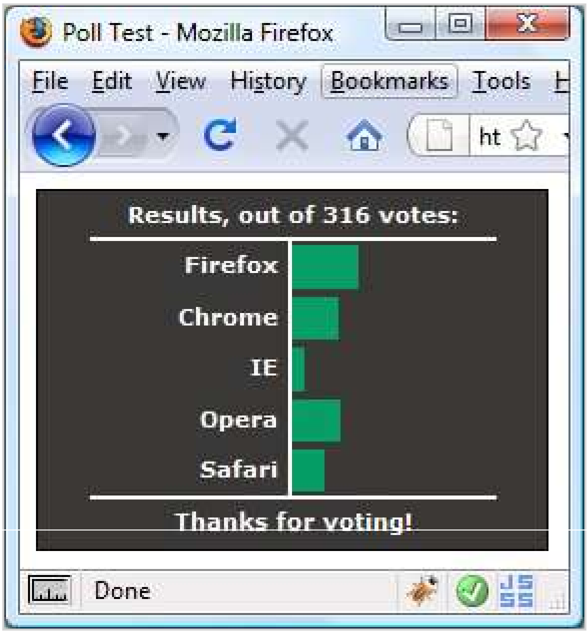
by Dan Wellman 18 Feb 2009

1

In this tutorial were going to be creating a jQuery plugin from start to finish; this plugin will allow us (or other developers) to easily add a simple poll widget to a web page or blog. By poll widget, I mean an area in which a question is posed which visitors are encouraged to answer. Once they have answered the question the results of the poll will then be displayed.

Final Product

The above video and below screenshot shows what we'll be working towards:


Poll Test - Mozilla Firefox
File Edit View History Bookmarks Tools
Done

Results, out of 316 votes:

Firefox
Chrome
IE
Opera
Safari

Thanks for voting!

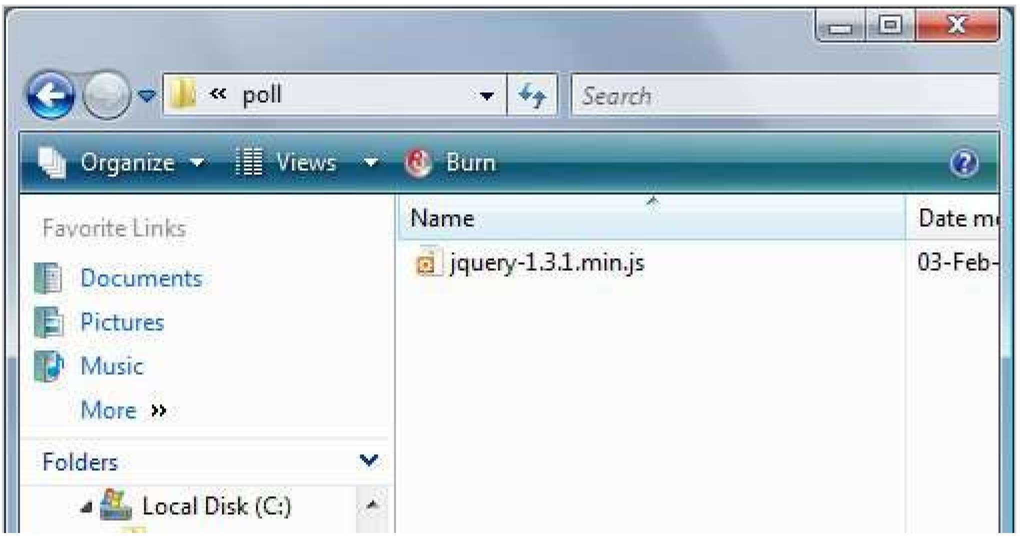
The plugin will use jQuery goodness to generate the DOM structure of the widget, as well as capture the answer to the question and pass it to the server for storage. Well use a little basic PHP to add the newest vote to a MySQL database and then echo back the new results in a JSON object. jQuery will then be used to process the response and display the results (as shown above) in the widget.

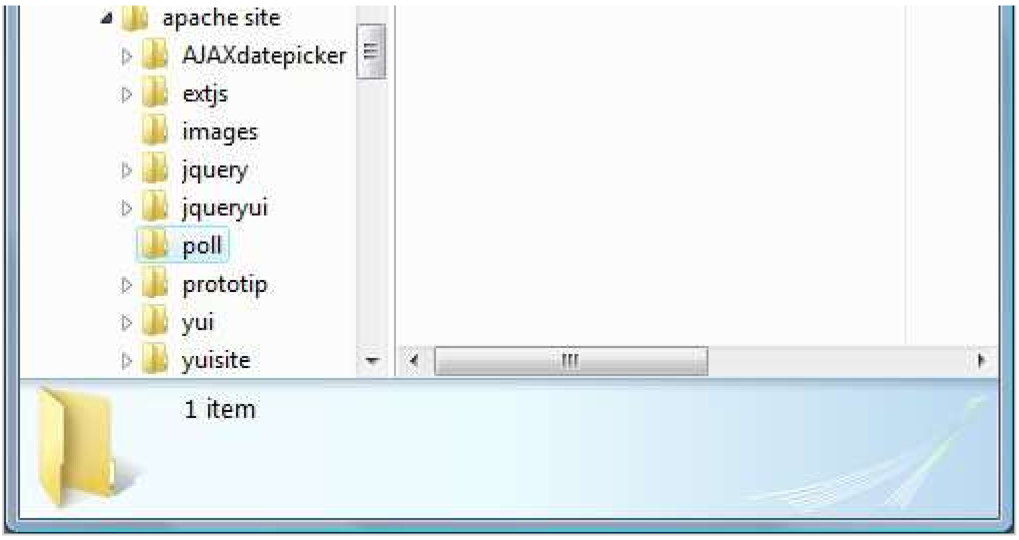
Although installing and configuring a web server, PHP and MySQL is beyond the scope of this tutorial, we will be looking at all of the steps needed including setting up the database. All in all, well be working with CSS, HTML, jQuery, PHP, MySQL and JSON during the course of this tutorial.

Prep Work

We should set up our development area first of all. To run this example on a desktop computer (for development, testing etc) well need to run the example files from a directory our web server can serve content from. I use Apache and have a folder setup on my C drive called **apache site**. This is the content-serving directory for my **localhost**. Within this folder (or the equivalent folder on your system) we should create a new folder called **poll**. This is where all of our example files will be stored.

To create a jQuery plugin, were also going to need a copy of jQuery itself; the latest version is 1.31.js and can be found at http://jquery.com. Download it to the **poll** directory we just created. So far the folder should look like this in Explorer (or equivalent file explorer application):





Next we can set up the database that will be used to store the poll results; we can do this easily enough using the MySQL Command Line Interface(CLI) easily enough, although database front-end GUIs can also be used. Open up the MySQL CLI and create a new database called poll using the following command:

```
CREATE DATABASE poll;
```

The **CREATE DATABASE** command does exactly what it says on the tin and creates a new database with the specified name. Once we have a database well need to create a new table in which to store the poll results. Before we can do this we need to select the new database; the **USE** command will do this:

```
USE poll;
```

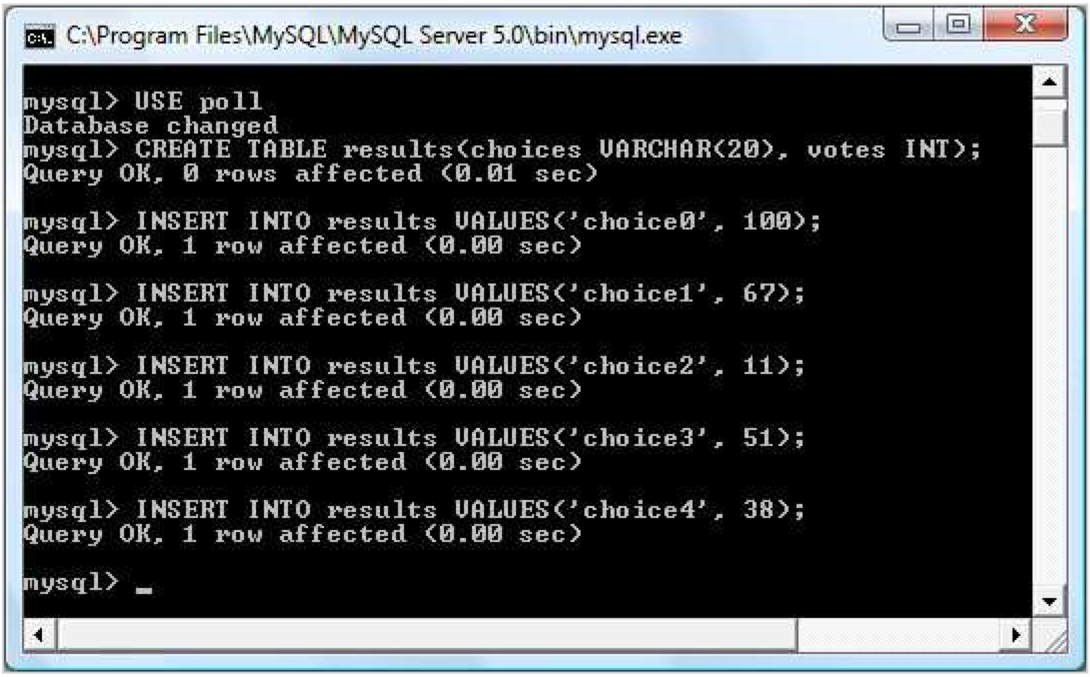
To create a new table we use the **CREATE TABLE** command, specifying the names for the columns within the table:

```
CREATE TABLE results(choices VARCHAR(20), votes INT);
```

If we were deploying this on a site wed start off with an empty table, but so that we can see some results without answering the question ourselves repeatedly, we can enter some dummy data into the table. The quickest and easiest way to do this for small sets of data (just 5 rows in this example) is to do it manually, which we can do with the following series of commands:

```sql
INSERT INTO results VALUES(choice0, 100);
INSERT INTO results VALUES(choice1, 67);
INSERT INTO results VALUES(choice2, 11);
INSERT INTO results VALUES(choice3, 51);
INSERT INTO results VALUES(choice4, 38);
```

The command should be straight-forward enough, just remember to hit enter after each line. The only point worthy of note is that the first row is **choice0** instead of **choice1** which is done to make working with the JSON object in our script easier. At this point your CLI should appear something like the following screenshot:



```
mysql> USE poll
Database changed
mysql> CREATE TABLE results(choices VARCHAR(20), votes INT);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO results VALUES('choice0', 100);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO results VALUES('choice1', 67);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO results VALUES('choice2', 11);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO results VALUES('choice3', 51);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO results VALUES('choice4', 38);
Query OK, 1 row affected (0.00 sec)

mysql> _
```

Were done with the CLI now so we can exit it and move on to our next task creating the plugin.

Building the Plugin

We have a number of tasks to complete with the plugin code; we need to create the DOM structure for the widget, add a handler that listens for submission of the selection, pass the results to the server and process the response, as well as displaying the results once processed. We can also add some sugar in the form of error messages and animated results.

Its going to take a few lines of code for sure, but it should be worth it as well get to see how easy it is to make a robust, functional and advanced plugin that provides interactivity and adds value to the page. Lets make a start; in a new file in your text editor add the following code:

```
        (function($) {

})(jquery)
```

All of our plugin code will be encapsulated within this self-executing anonymous function. Its the additional braces after the function that makes it self-executing, and these also allow us to pass arguments into our plugin. In this example, were passing the jQuery object into our plugin so that we can make use of all of jQuerys amazing functionality. The function receives **jquery** as the **$** alias which is how we can work with the library from within our plugin. Now within the function add the following code:

```
//define jPoll object with some default properties
$.jPoll = {
  defaults: {
    ajaxOpts: {
      url: "poll.php"
    },
    groupName: "choices",
    groupIDs: ["choice0", "choice1", "choice2", "choice3", "choice4"],
    pollHeading: "Please choose your favourite:",
    rowClass: "row",
    errors: true
  }
};
```

This code adds a new object to jQuery called **jPoll**. Within this object is another object called **defaults** which contains a number of different properties. Each property will be a configurable option for our plugin that implementers can change according to their requirements. Its useful to provide as many configurable properties as possible to give the plugin flexibility and robustness.

The first property we set is the **ajaxOpts** property, the value of which is another object. This object will be used to make the request to the server. We make the URL of the back-end script file a configurable property so that developers can specify their own server-side file to take the result and pass it into the database if required.

Because the values in each property are simply defaults to fall back on if implementers dont specify them explicitly, we should try to make them as generic as possible, so, for example, the **groupName** property, which will be applied to the radio buttons well be creating later, is given the value **choices**. The **groupIDs** property accepts an array of values; the array items will be used as labels for the radio buttons and indicate to the visitor the options they have when making a choice.

The **rowClass** property is used to add a class name to the row elements which will act as containers for the radio buttons and results. The errors property is set to the boolean value true. This allows implementers to switch off the automatic error message if they wish. Flexibility is the key when deciding which features of a plugin should be configurable.

Lets continue by adding the following code directly after our default configuration object:

```
//extend jquery with the plugin
$.fn.extend({
  jPoll:function(config) {

  }
});

```

The jQuery **fn.extend** method is a special construct provided by jQuery for the creation of plugins; it allows us to add additional methods to jQuery, so our plugin actually becomes part of the library. The method accepts an object which in this example contains a single property our plugin.

All of our plugin code will be placed within the anonymous function that is specified as the value of the property. This function accepts a single argument and is how implementers supply their own configuration object to alter the default options we added earlier.

Now we can add the code for our plugin that will create the initial DOM structure and render the widget. Within the anonymous function we just created add the following code:

```
//use defaults or properties supplied by user
config = $.extend({}, $.jPoll.defaults, config);

```

The **extend** jQuery method takes the object supplied to our plugin (if its provided by the implementer) and applies it to our default configuration object. Any properties supplied by the developer will overwrite the defaults. The method works in a similar way to the **fn.extend** method although on a smaller scale. Now we can start creating elements:

```
//init widget
$("<hr />
<h2>").text(config.pollHeading).appendTo($(this));
$("<form>").attr({
  id: "pollForm",
  action: config.ajaxOpts.url,
  method: config.ajaxOpts.type
}).appendTo($(this));

```

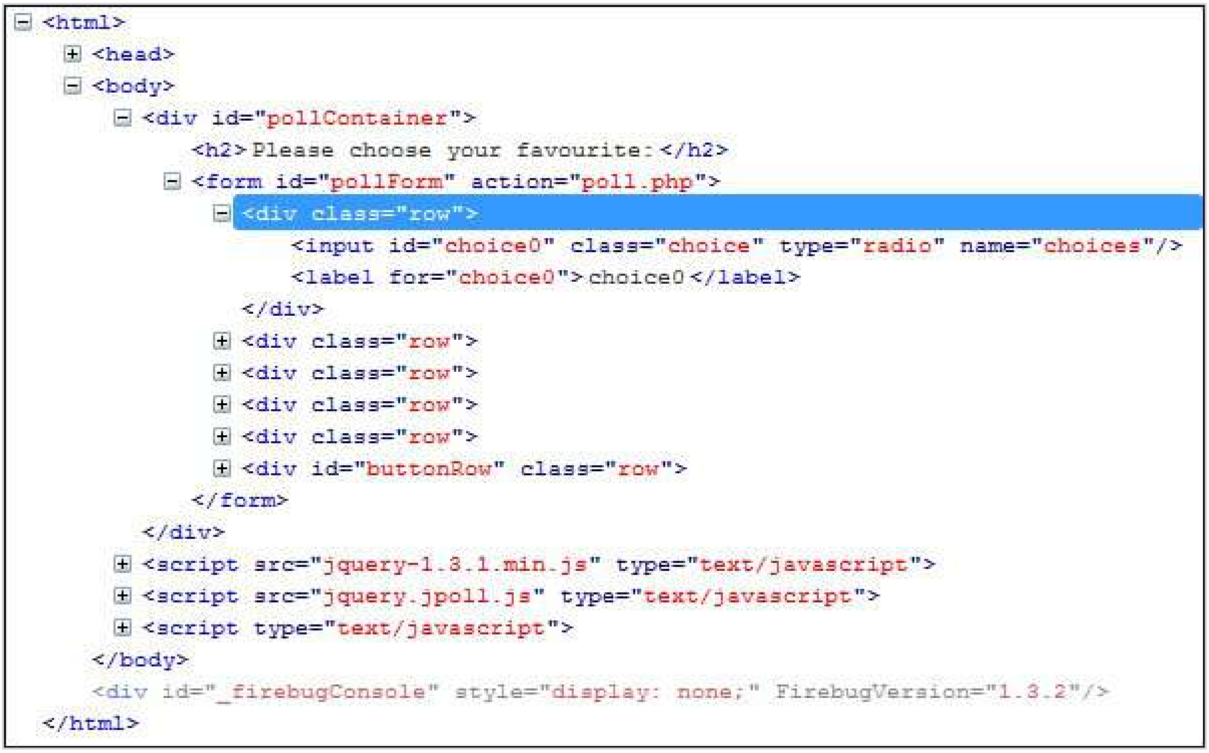
First we create a heading which can be used to display the question at the top of the widget. We can then create a form element and append it to the widget using jQuerys **appendTo** method. We can also set some of the forms attributes.

Within the context of this part of the plugin, the **$(this)** object refers to the element that our plugin is called on; the HTML page on which it is used will need a container element for the widget to be rendered into and the plugin will be called on the container. Therefore **$(this)** will refer to the container element. Next we can add the radio buttons and labels that will form the choices a visitor can make:

```
for(var x = 0; x < config.groupIDs.length; x++) {

  $("<div>").addClass(config.rowClass).appendTo($(this).find("form"));
  $("<input type='radio' name='" + config.groupName + "' id='" + config
    ($(".error").length != 0) ? $(".error").slideUp("slow") : null ;
  });

  $("<label>").text(config.groupIDs[x]).attr("for", config.groupIDs[x])
}

```

The number of choices the visitor can make is dictated by the number of items in the **groupIDs** array, which by default is five. We loop through the array, which we can access via the config object. On each iteration we create a container element and give it the class name specified by the **rowClass** property before appending it to the form.

We then create a radio button and append it to the container we added a moment ago. You might be wondering why we specify the attributes for the radio button in-line with its constructor instead of using jQuerys **attr** method as we do with other elements. The reason is simply because of a bug in IE that prevents the use of dynamic radio buttons. The structure of these elements will be as highlighted in the following Firebug screenshot:



```
⊟ <html>
    ⊞ <head>
    ⊟ <body>
        ⊟ <div id="pollContainer">
              <h2>Please choose your favourite:</h2>
            ⊟ <form id="pollForm" action="poll.php">
                ⊟ <div class="row">
                      <input id="choice0" class="choice" type="radio" name="choices"/>
                      <label for="choice0">choice0</label>
                  </div>
                ⊞ <div class="row">
                ⊞ <div class="row">
                ⊞ <div class="row">
                ⊞ <div class="row">
                ⊞ <div id="buttonRow" class="row">
              </form>
          </div>
        ⊞ <script src="jquery-1.3.1.min.js" type="text/javascript">
        ⊞ <script src="jquery.jpoll.js" type="text/javascript">
        ⊞ <script type="text/javascript">
      </body>
      <div id="_firebugConsole" style="display: none;" FirebugVersion="1.3.2"/>
  </html>
```

Additional Elements

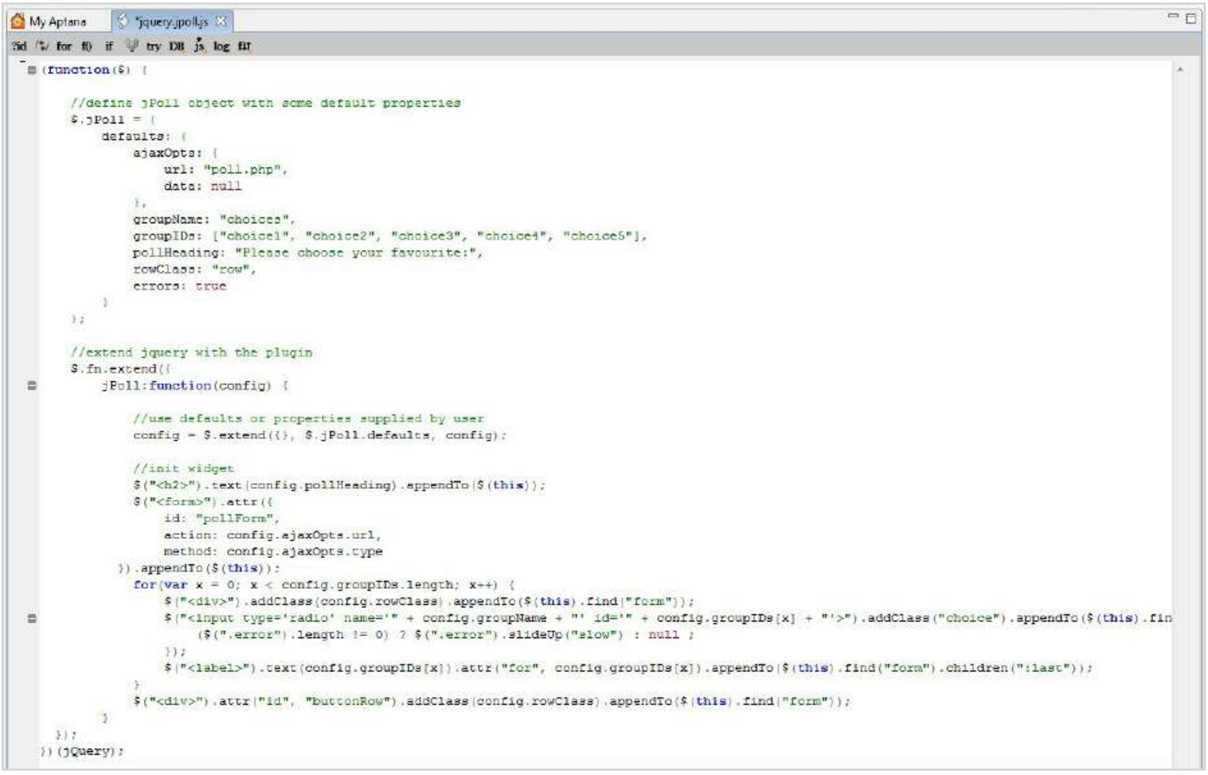
We also add a click handler for each radio which is trigged when it is selected by the visitor; all the function does is check whether there is one or more elements with the class name error and if so, hides them using the **slideUp** jQuery animation. We dont need to worry about checking whether the implementer has disabled the error property at this stage because if no errors are detected, the function will just return **false**.

Next we add a label for the radio button indicating which choice it refers to. For accessibility we add a **for** attribute so that the label is associated with the radio that it shares the container div with. Both the label and the radio are attached to the container div using the **find** and **children** methods.

To make the plugin as least restrictive as possible we want to refer to as few things as possible using a set **ID** so using **find** to return the first form (which will be the one we added ourselves) within the element in the context of **$(this)**, and then selecting the last child allows us to add the elements to the correct container without needing an **ID**. After the for loop we still need to add a couple more elements:

```
$("<div>").attr("id", "buttonRow").addClass(config.rowClass).appendTo($(
```

We add a final container to the widget which will be used to hold the submit button for the form. Well add the button a little later on. Save what weve done so far as **jquery.jpoll.js** into the poll directory. At this stage, we can move on to create the HTML page which the widget will be added to, as well as a little basic styling. We added quite a bit of code already and so far Ive said things like add this code within the function etc. The following screenshot shows how the code should appear at this point:



```
(function($) {

    //define jPoll object with some default properties
    $.jPoll = {
        defaults: {
            ajaxOpts: {
                url: "poll.php",
                data: null
            },
            groupName: "choices",
            groupIDs: ["choice1", "choice2", "choice3", "choice4", "choice5"],
            pollHeading: "Please choose your favourite:",
            rowClass: "row",
            errors: true
        }
    };

    //extend jquery with the plugin
    $.fn.extend({
        jPoll:function(config) {

            //use defaults or properties supplied by user
            config = $.extend({}, $.jPoll.defaults, config);

            //init widget
            $("<h2>").text(config.pollHeading).appendTo($(this));
            $("<form>").attr({
                id: "pollForm",
                action: config.ajaxOpts.url,
                method: config.ajaxOpts.type
           }).appendTo($(this));
            for(var x = 0; x < config.groupIDs.length; x++) {
                $("<div>").addClass(config.rowClass).appendTo($(this).find("form"));
                $("<input type='radio' name='" + config.groupName + "' id='" + config.groupIDs[x] + "'>").addClass("choice").appendTo($(this).fin
                    ($(".error").length != 0) ? $(".error").slideUp("slow") : null ;
                });
                $("<label>").text(config.groupIDs[x]).attr("for", config.groupIDs[x]).appendTo($(this).find("form").children(":last"));
            }
            $("<div>").attr("id", "buttonRow").addClass(config.rowClass).appendTo($(this).find("form"));
        }
    });
})(jQuery);
```

Some Basic HTML and CSS

In a new page in your text editor, create the following very basic page:

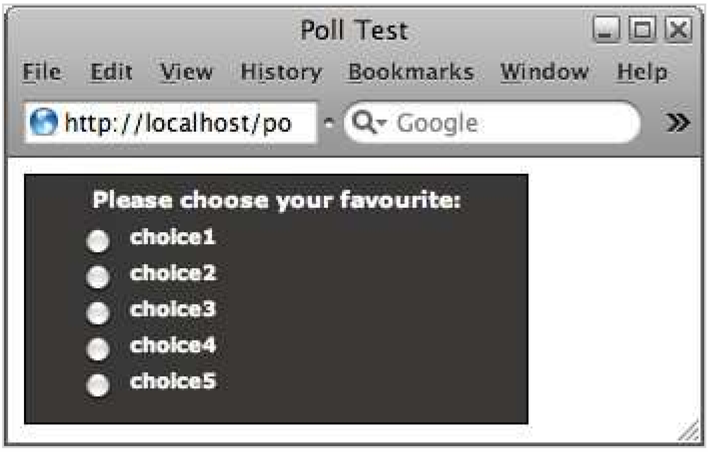
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR
<html>
  <head>
    <<meta http-equiv="Content-Type" content="text/html; charset=utf-8"
    <title>Poll Test</title>
    <link rel="stylesheet" type="text/css" href="poll.css">
  </head>
  <body>
    <div id="pollContainer"></div>
    <script type="text/javascript" src="jquery-1.3.1.min.js"></script>
    <script type="text/javascript" src="jquery.jpoll.js"></script>
    <script type="text/javascript">
      $("#pollContainer").jPoll();
    </script>
  </body>
</html>
```

Save this as **pollTest.html** in the **poll** folder we created earlier. Thats pretty much it as far as the HTML is concerned but its all we need. Its totally minimal, basically a template file that links to an external stylesheet, the jQuery library, and our plugin file. The body of the page has just the required container element present, and in a custom script block we call the **jPoll** method that weve added to jQuery.

The CSS well be using in the example is also pretty basic and is intended to present the widget in a simple skin that highlights its features. None of the CSS is required to control how the widget behaves so you can pretty much use whatever CSS you need to. The following code shows the skin used in this example:

```

#pollContainer {
  width:250px; border:1px solid #000000; margin:0; text-align:center;
  background-color:#3a3737; position:relative; padding-bottom:10px;
}
#pollContainer form, #results {
  text-align:left; margin:0 0 0 30px;
}
#pollContainer h2, #pollContainer p {
  font-family:Verdana; font-size:11px; margin:5px 0; color:#ffffff;
  font-weight:bold;
}
#pollContainer .error {
  margin:5px auto 0; background:url(images/warn.gif) no-repeat 0 0;
  padding-left:10px; width:182px;
}
#pollContainer input { margin:0 10px 0 0; }
#pollContainer label {
  font-family:Verdana; font-size:10px; font-weight:bold;
  position:relative; top:-3px; color:#ffffff;
}
#pollContainer button { margin:5px 0 0; }
#results {
  width:200px; margin:5px auto 0; border-top:2px solid #ffffff;
  border-bottom:2px solid #ffffff;
}
.row { width:200px; overflow:hidden; }
#results label {
  width:93px; font-family:Verdana; font-size:11px; font-weight:bold;
  color:#ffffff; text-align:right; border-right:2px solid #ffffff;
  padding:5px 5px 10px 0; float:left; clear:both; height:10px; top:0;
}
.result {
  background-color:#079d67; width:0; float:left; height:21px;
  margin:2px 2px 2px 0;
}
#pollContainer #thanks {
  margin:0; position:relative; width:100%; text-align:center; clear:bot
  top:4px;
}
```

Save this file as **poll.css** in the **poll** project folder. Theres nothing interesting or difficult here really, its all just presentation. Ive added an image to give the error message more weight. Its included in the accompanying code download for this tutorial and should be placed into a folder called **images** within our **poll** directory. The JavaScript, HTML and CSS that weve added so far should go together to form something like the following screenshot when run in a browser:



Please choose your favourite:

- choice1
- choice2
- choice3
- choice4
- choice5

Continuing with the Plugin

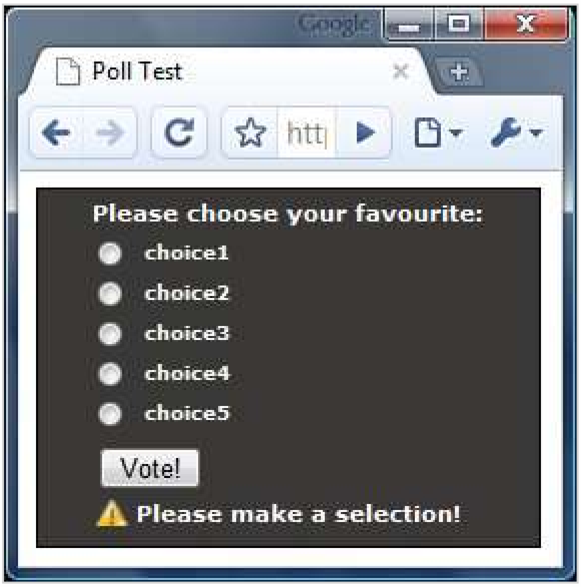
The first stage of the plugin is complete; the widget renders and the (very generic) question to the visitors is posed. Now we need to capture their response and do something with it. In **jquery.jpoll.js** add the following code directly after the code we added to create the **buttonRow** container:

```

$("<button type='submit'>").text("Vote!").appendTo("#buttonRow").click(
  e.preventDefault();

  //record which radio was selected
  var selected;
  $(".choice").each(function() {
  ($(this).attr("checked") == true) ? selected = $(this).attr("id") : n
});

//print message if no radio selected and errors enabled
if (config.errors == true) {
   (selected == null && $(".error").length == 0) ? $("<p>").addClass("er
}
```

We add a submit button to the final container within the form and set a click handling function on it. This will be quite a large function so well look at it in small chunks. The first thing we do is prevent the default behaviour of the submit button, which would be to submit the form resulting in a page reload definitely something we want to avoid in an AJAX scenario.

Then we set a new variable called **selected** which will be used to hold the **ID** of the radio button that was chosen. We loop through each radio button with jQuerys **each** method to see whether the radio has the **checked** attribute. If no radios were selected the variable will be set to **null**.

Next we check that the value of the variable isnt set to **null**. If it is, then we can add the error message after the form and show it with a nice little animation. We check that the message isnt already being displayed to prevent multiple messages being added if the button is repeatedly clicked. The error message should display something like this:



Please choose your favourite:
choice1
choice2
choice3
choice4
choice5

Please make a selection!

Now that were inside the click handling function for the button, the **$(this)** object will no longer refer to our widget container, instead it refers to the button, so at this point we have to start referring to the form using the **ID** attribute we gave it earlier on.

Once were satisfied that a choice has been made we can think about sending the details of the choice to the server. Well need to configure the AJAX request first which we can do with the following code:

```

//add additional request options
var addOpts = {
  type: "post",
  data: "&choice=" + selected,
  dataType:"json",
  success: function(data) {
```

We create a new object to hold the additional properties required for the AJAX request. We set the **type** to **POST** as well be sending data to the server and configure the data that well be sending. The format of the servers response will be in JSON so the **dataType** property is set to **json**. Finally we add a success handler to be executed when the request returns successfully. Within this anonymous function we should add the following code:

```

//add all votes to get total
var total = 0;
for (var x = 0; x < data.length; x++) {
  total += parseInt(data[x].votes);
}
```

First we work out the total number of choices that have been made so far by iterating through the response object and adding the number of votes for each choice. We use the standard JavaScript **parseInt** function to convert the JSON string into an integer.

```

  //change h2
$("div#pollContainer").find("h2").text("Results, out of " + total + " vo

//remove form
$("form#pollForm").slideUp("slow");
```

We then change the heading element so that it indicates that the widget will now show the results. For good measure we can also show how many votes have been cast so far. We then animate the form away to make space for the results.

```
//create results container
$("<div>").attr("id", "results").css({ display:"none" }).insertAfter("#

//create results
for (var x = 0; x < data.length; x++) {

  //create row elment
  $("<div>").addClass("row").attr("id", "row" + x).appendTo("#results")

  //create label and resuls
  $("<label>").text(config.groupIDs[x]).appendTo("#row" + x);
  $("<div>").attr("title", Math.round(data[x].votes / total * 100) + "%
}

```

This segment of code creates a container div for the results then in the same way that we created containers and form elements earlier on, we create container elements and results. Each row in the results container will have a label, which uses the **config.groupIDs** property to set the text of the labels. Each result will be a div element as well and we work out the percentage of votes for each choice and add this as the **title** attribute. The results container and each result div are initially hidden from view. This is so that we can show them using smart animations:

```

  //show results container
    $("#results").slideDown("slow", function() {

      //animate each result
      $(".result").each(function(i) {
        $(this).animate({ width: Math.round(data[i].votes / total * 100
      });

      //create and show thanks message
      $("<p>").attr("id", "thanks").text("Thanks for voting!").css({ di
    });
  }
};
```

Finally, we can slide the results container into view with jQuerys **slideDown** method, and then show each result with a custom animation that sets the **width** of the result div to reflect the percentage of votes for each choice. We can also add a thanks message to thank the visitor for taking part and fade this into view.

We use JavaScripts **Math.round** function to set the number as an integer, which is cleaner in this situation. We also work out the number of votes as a percentage (again using the total variable). I think for this kind of widget using a percentage is better because it will prevent the results div getting too big for the widget.

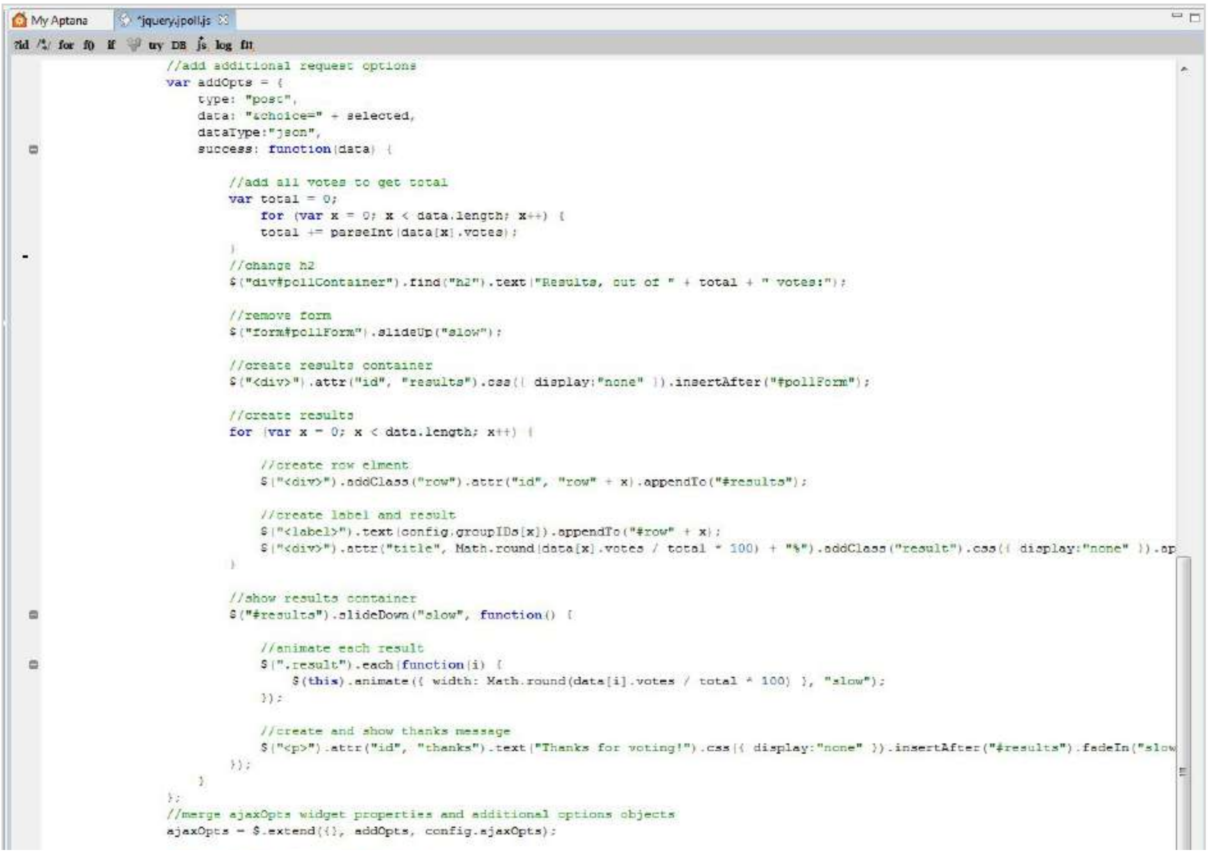
Remember however, all of this code resides within the **success** handler for our AJAX request we still need to actually make the request. Directly after the above code, add the following:

```
    //merge ajaxOpts widget properties and additional options objects
    ajaxOpts = $.extend({}, addOpts, config.ajaxOpts);

    //make request if radio selected
    return (selected == null) ? false : $.ajax(ajaxOpts) ;
  });

  //return the jquery object for chaining
  return this;

```

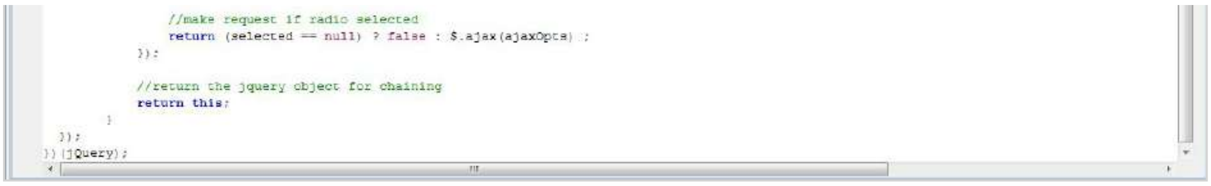
In the final section of code for our plugin we use jQuerys **extend** method once more to combine the configurable AJAX property from the developer-supplied configuration object (or the config object) with the additional options we set a little while ago.

Provided a radio button was selected we then make the request using jQuerys **ajax** method, supplying our newly merged **ajaxOpts** object as an argument. The final part of the plugin is very important; we should return this (which will now once again refer to the object representing our widgets main container div as we are outside of any inner functions) so that additional jQuery methods can be chained onto our method.

Again, heres a screenshot to show how the end of the plugin file should be looking at this point:



```
        //add additional request options
        var addOpts = {
            type: "post",
            data: "&choice=" + selected,
            dataType:"json",
            success: function(data) {

                //add all votes to get total
                var total = 0;
                    for (var x = 0; x < data.length; x++) {
                    total += parseInt(data[x].votes);
                }
                //change h2
                $("div#pollContainer").find("h2").text("Results, out of " + total + " votes:");

                //remove form
                $("form#pollForm").slideUp("slow");

                //create results container
                $("<div>").attr("id", "results").css({ display:"none" }).insertAfter("#pollForm");

                //create results
                for (var x = 0; x < data.length; x++) {

                    //create row elment
                    $("<div>").addClass("row").attr("id", "row" + x).appendTo("#results");

                    //create label and result
                    $("<label>").text(config.groupIDs[x]).appendTo("#row" + x);
                    $("<div>").attr("title", Math.round(data[x].votes / total * 100) + "%").addClass("result").css({ display:"none" }).ap
                }

                //show results container
                $("#results").slideDown("slow", function() {

                    //animate each result
                    $(".result").each(function(i) {
                        $(this).animate({ width: Math.round(data[i].votes / total * 100) }, "slow");
                    });

                    //create and show thanks message
                    $("<p>").attr("id", "thanks").text("Thanks for voting!").css({ display:"none" }).insertAfter("#results").fadeIn("slow
                });
            }
        };
        //merge ajaxOpts widget properties and additional options objects
        ajaxOpts = $.extend({}, addOpts, config.ajaxOpts);
```



```
                //make request if radio selected
                return (selected == null) ? false : $.ajax(ajaxOpts) ;
            });

            //return the jquery object for chaining
            return this;
        }
    });
})(jQuery);
```

This is now all of the code required for our plugin. We can next look at an example of the kind of PHP code needed to handle the request made by our plugin. There are many different ways in which we could achieve the same end using different techniques or server-side script languages. Ive used PHP in this example because I like it, but other languages may be equally as effective.

A Little PHP

The PHP needed for this example is relatively straight-forward, well see all of the code needed and then look at what each bit does. In a new file add the following code:

```php

<?php

  //db connection details
  $host = "localhost";
  $user = "root";
  $password = your_password_here;
  $database = "poll";

  //make connection
  $server = mysql_connect($host, $user, $password);
  $connection = mysql_select_db($database, $server);

  //get post data
  $selected = $_POST['choice'];

  //update table
  mysql_query("UPDATE results SET votes = votes + 1 WHERE choices = '$s

  //query the database
  $query = mysql_query("SELECT * FROM results");

  //loop through and return results
  for ($x = 0, $numrows = mysql_num_rows($query); $x < $numrows; $x++)
    $row = mysql_fetch_assoc($query);

    //make array
    $json[$x] = array("choice" => $row["choices"], "votes" => $row["vot
  }

  //echo results
  echo json_encode($json);

  //close connection
  mysql_close($server);

?>
```

Save this in the **poll** folder as **poll.php.** The first thing we do in the PHP is define the credentials needed to access and work with the MySQL database that we created at the start of this tutorial. The host and username will depend upon how MySQL is configured on your system and the password should be whatever password you enter when signing into the MySQL CLI. The information needed is stored in the **$connection** variable for easy access later in the script.

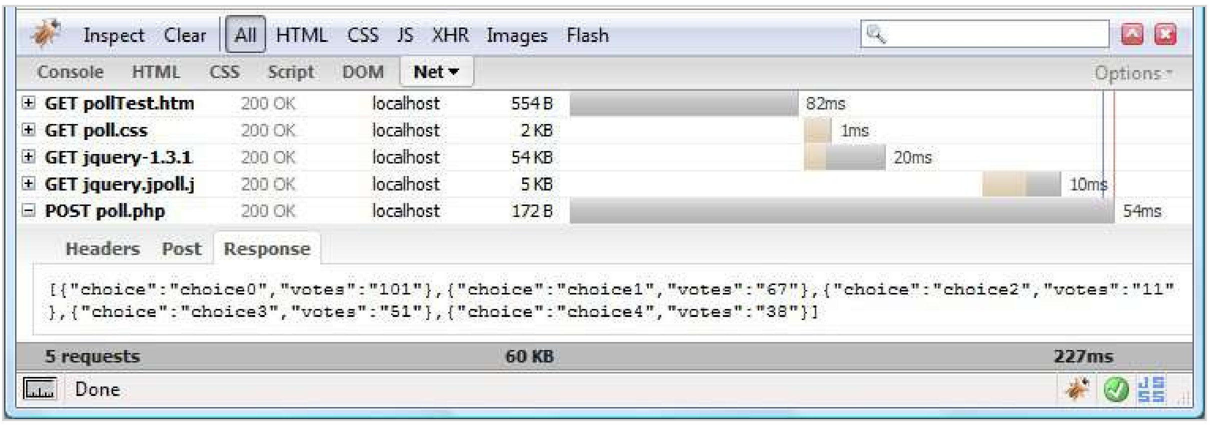
Once we have the necessary connection information we can then get the data passed to the script by our plugin, which will be accessible from the **$_POST** superglobal. Next we can update the table in our database with the new choice made by the visitor. This is really easy and simply involves looping through each row of the table until we find a row where the data in the **choices** column matches the data obtained from the superglobal. We then increment the value of the integer in the **votes** column by one.

Once the table has been updated we then query the table to get the new totals. We select all of the table data using the **SELECT * FROM** syntax and store this in the $query variable. Using a for loop we then loop through each row of data and create a new associative array item for each row containing the data from the **choices** and **votes** columns in the table.

Once weve built our multidimensional associative array (which sounds much more complex than it actually is!) we can then use PHPs native **json_encode** function build a proper JSON structured object from the array and then echo this back to our plugin. Finally we close the MySQL connection as its no longer needed and we dont want to keep it open unnecessarily.

About the JSON

Lets take a little look at the structure of the JSON object that is returned to the plugin it is a key part of how the plugin works. Once weve made a choice and hit the vote button, we can then use the Dom tab of Firebug to have a look at the data thats returned:



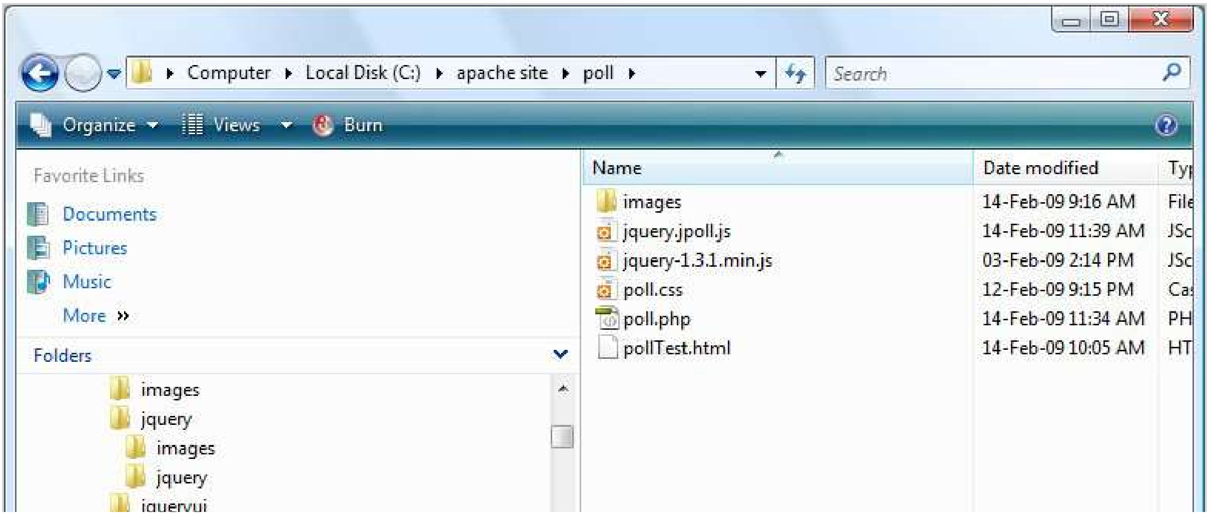
```
GET pollTest.htm    200 OK  localhost  554 B  82ms
GET poll.css        200 OK  localhost  2 KB   1ms
GET jquery-1.3.1    200 OK  localhost  54 KB  20ms
GET jquery.jpoll.j  200 OK  localhost  5 KB   10ms
POST poll.php       200 OK  localhost  172 B  54ms
Headers  Post  Response
[{"choice":"choice0","votes":"101"},{"choice":"choice1","votes":"67"},{"choice":"choice2","votes":"11"
},{"choice":"choice3","votes":"51"},{"choice":"choice4","votes":"38"}]
5 requests          60 KB          227ms
Done
```

This screenshot tells us everything we need to know about the properties and values within our object. Because the object is a standard JavaScript object pretty much like any other, we already know how we can access the different parts of it. Like a standard array and object like this will have a length property and can be access using standard bracket notation, e.g. **data[x]**. To access the first property of the first item in the object we just use **data[x].choice** which returns the value of the property. To access the second property, we just go **data[x].votes**.

This simplicity is what makes JSON so incredible, and thats without JSONP. JSONP, which we havent covered in this tutorial, is quite simply, total JavaScript nirvana; letting us access remote data, completely cross-domain if necessary, and processing it directly in the browser. Its no wonder than JSON is quicker and easier to work with even than the very flexible XML.

Playing with the Plugin

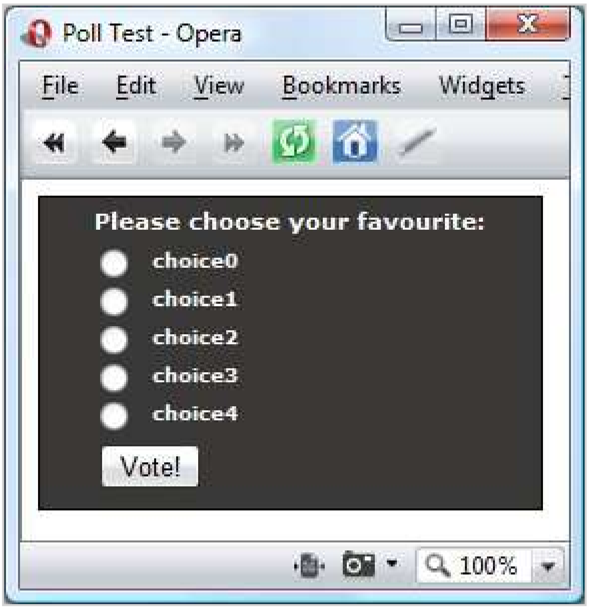
Weve gotten to the stage now where we can see the plugin in action. Our project folder should now contain all of the required files:



Name	Date modified	Ty
images	14-Feb-09 9:16 AM	File
jquery.jpoll.js	14-Feb-09 11:39 AM	JSc
jquery-1.3.1.min.js	03-Feb-09 2:14 PM	JSc
poll.css	12-Feb-09 9:15 PM	Cas
poll.php	14-Feb-09 11:34 AM	PH
pollTest.html	14-Feb-09 10:05 AM	HT

If you go in your browser of choice now and type the following URL in the address bar you should see the initial widget:

http://localhost/poll/pollTest.html


Poll Test - Opera
File Edit View Bookmarks Widgets
Please choose your favourite:
choice0
choice1
choice2
choice3
choice4
Vote!
100%

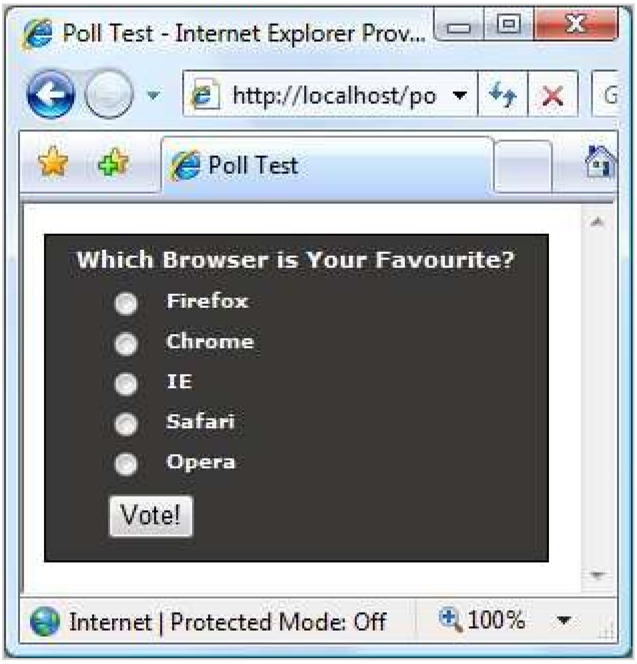
Once weve made a choice and hit the vote button, we should then see the results page and each result as it is animated into existence. It should appear something like in the screenshot at the start of the tutorial.

Configuring the Widget

In the final part of the tutorial we can see how the widget can be configured using the options we specified in the **defaults** object. Our plugin is called in the final script block in the HTML page; change this line of code so that it is as follows:

```

$("#pollContainer").jPoll({
  groupIDs: ["Firefox", "Chrome", "IE", "Safari", "Opera"],
  pollHeading: "Which Browser is Your Faviouite?"
});
```

Now when the page is run, the widget should appear like this instead:


Poll Test - Internet Explorer Prov...
http://localhost/po
Poll Test
Which Browser is Your Favourite?
Firefox
Chrome
IE
Safari
Opera
Vote!
Internet | Protected Mode: Off
100%

Summary

This brings us to the end of the tutorial; I hope its shown you how easy it is to build even an advanced plugin when we have jQuery at our side. The library provides everything we need, all our plugins will be limited only by our imaginations, never through a lack of the appropriate mechanisms for doing what we want to do.

Our plugin is as flexible as it needs to be; changing the question and the labels makes no difference to how the widget interacts with the PHP file; you could have more choices if you wanted and these two files, thanks to the generality we built in, will still work seamlessly together. The only thing we cant make generic is the MySQL database; this must be set up in the way we have specified, and if more or less choices are desired, the appropriate number of rows would need to be added or removed from the table.