

## Developing Apps for the BlackBerry PlayBook

### Lab # 4: *Getting Started with Ajax and jQuery – Part 2*

The objective of this lab is to continue reviewing some of the concepts in communication with external data sources in a WebWorks application for the BlackBerry PlayBook. In this lab, we'll continue dealing with Ajax, jQuery and DOM.

Before attempting this lab, please be sure to download and install the BlackBerry WebWorks SDK for Tablet OS available at <https://bdsc.webapps.blackberry.com/html5/download/sdk>. Follow the steps under 'How to get up and running' to ensure you have everything you need. After completing the steps, you should have installed the Adobe AIR SDK, the BlackBerry WebWorks SDK for Tablet OS and VMware Player. You do not need to worry about signing your application as we'll be testing it on the simulator. Instructions on how to setup the simulator in VMware Player can be found here:

[http://docs.blackberry.com/en/developers/deliverables/27278/Configure\\_VM\\_BB\\_tablet\\_simulator\\_1594361\\_11.jsp](http://docs.blackberry.com/en/developers/deliverables/27278/Configure_VM_BB_tablet_simulator_1594361_11.jsp). This lab also assumes that you have knowledge of basic JavaScript.

#### Exercise 1: Continuing Communicating with External Data Sources in an HTML application

- Use your completed code from Lab 3. Open index.html in your favorite text editor.
- Open script.js in your favorite text editor as well.
- We'll continue communicating directly with Google in order to obtain information about the author which the user entered in the search box. In the previous lab, we've started the process, and now we'll continue it.
- Let's start by coding the function that will get called if there are less than 100 papers. We called this function 'doCallForLessThan100Papers' in the previous lab. Find that function in your code.
- Start by creating a variable called 'url' which will hold the URL we need to parse the results from. Your 'url' variable will use the 'resultquery' variable that was sent to the function, as well as the global 'ret\_results' variable. Remember: 'resultquery' is the user's input formatted to be used in the URL, and 'ret\_results' is the global variable which holds the number of results Google will display per page.
- Create the following 'url' variable

```
var url =  
"http://scholar.google.ca/scholar?q="+resultquery+"&num="+ret_results+  
"&hl=en&btnG=Search&as_sdt=1%2C5&as_sdt=on";
```

- Next we'll create another jQuery get function to retrieve the HTML page. Start with the following outline of the 'get' function:

```
$.get(url, function(data)  
{
```

```

        //Your code will go in here
    })
    .error(function()
    {
        alert("An error occurred");
        return;
    });

```

- Again, the jQuery 'get' function will connect to the URL sent in the variable 'url' and once the data retrieval process is complete, the result will be in the variable 'data'.
- The first step inside the get function is to check if data is equal to null. If it is, we should alert the user that there is no data, set the style of the loading icon which was triggered when the search started to 'none', and return out of the function.

```

if(data===null)
{
    alert("There is no data");
    document.getElementById("loading").style.display="none";
    return;
}

```

- After we check if the data isn't null, we'll need to perform some actions on the data.
  - We'll first need to get the number of citations for each paper on the page (We'll create a function called 'getCitationCount' for this)
  - Then we'll need to get the range of years of papers (done with the 'getYearInformation' function)
  - This will be followed by calculating the total number of citations (done with the 'totalCites' function we'll create).
- Let's add the call to the functions, and deal with implementing them afterwards.
- Since these functions only gets called if we have one page of results, we'll just need to use index 0 in the 'citesPages' array. Remember this array holds the total citations on each page. Add a call to the 'getCitationCount' function which takes in 'data' as an argument, and assign the result to 'citesPages[0]'. Afterwards, call the function for 'getYearInformation' again sending data as an argument to the function, and finally calling the 'totalCites' function with no arguments. Your 'doCallForLessThan100Papers' function should now look like the following, which is also included lab4functions.txt file included with the lab:

```

function doCallForLessThan100Papers(resultquery)
{
    var url =
    "http://scholar.google.ca/scholar?q="+resultquery+"&num="+r
    et_results+"&hl=en&btnG=Search&as_sdt=1%2C5&as_sntp=on";

    $.get(url, function(data)
    {
        if(data===null)
        {
            alert("There is no data");
            document.getElementById("loading").
            style.display="none";
            return;
        }
    });
}

```

```

        citePages[0] = getCitationCount(data);
        getYearInformation(data);
        totalCites();
    })
    .error(function()
    {
    alert("An error occurred");
    return;
    });
}

```

- Now, add the following function for 'getCitationCount' in your code. The code for this function can also be found in lab4functions.txt included with the lab.

```

function getCitationCount(responseText)
{
    if (responseText == null)
    {
        alert("There is no data.");
        document.getElementById("loading").
        style.display="none";
        return;
    }

    var cite_exists = 1;
    var resultPositionPost = 0;
    var citeArray = new Array();

    for(var i = 0; cite_exists > 0; i++)
    {
        cite_exists = responseText.indexOf("Cited by",
        resultPositionPost + 1);
        if(cite_exists == -1)
        {
        }
        else
        {
            var post = '</a>';
            var resultPositionPre = cite_exists + 'Cited
            by'.length + 1;
            resultPositionPost = responseText.indexOf(post,
            resultPositionPre);
            var resultLength = resultPositionPost -
            resultPositionPre;
            var tmp_string =
            responseText.substr(resultPositionPre,
            resultLength);
            citeArray[i] = tmp_string;
            publications++;
        }
    }
    return citeArray;
}

```

- The 'getCitationCount' function will first check to make sure the text sent is not null. If it is, it will alert the user and return out of the function. Afterwards, it will create variables for 'cite\_exists', 'resultPositionPost' and 'citeArray' and initialize them. The 'cite\_exists' variable will be used to store the location of the String 'Cited by' in the next sent to the function. If you look at the source code of a Google Scholar search result in your internet browser, you'll see that the text 'Cited by' is always followed by a number, which is the number of citations for each article.
- 'resultPostPosition' holds the location of the '</a>' HTML tag, which is always found after the number of citations in the HTML source code. The array 'citeArray' holds the number of citations for each of the articles returned from the query.
- The for-loop continues looping while 'cite\_exists' is greater than 0, which also means there is a more text that matches the String 'Cited by' in the results.
- The first step in the for-loop is to get the location of the text 'Cited by'. If the text is found (meaning that 'cite\_exists' doesn't equal -1), then we search for the text '</a>' starting from the position where we found 'Cited by'.
- We then get the length of the citation String by subtracting the beginning and end location, followed by using the 'substring' method to extract the String (the number of citations for that article) that we need. We then store this number into the 'citeArray' array at location 'i' in the for-loop. The entire process is repeated until we go through all of the publications on the page (meaning we can no longer find 'Cited by' on the page).
- The next function we need to add is the 'getYearInformation' function. This function can also be found in the lab4functions.txt file.

```
function getYearInformation(responseText)
{
    if (responseText == null)
    {
        alert("There is no data.");
        document.getElementById("loading").
            style.display="none";
        return;
    }

    var year_string_exists = 1;
    var resultPositionPost = 0;

    for(var i = 0; year_string_exists > 0; i++)
    {
        year_string_exists = responseText.indexOf('class=gs_a',
            resultPositionPost + 1);
        if(year_string_exists == -1)
        {
        }
        else
        {
            var post = '</span>';
            var resultPositionPre = year_string_exists +
                'class=gs_a'.length + 1;
            resultPositionPost = responseText.indexOf(post,
                resultPositionPre);
        }
    }
}
```



numbers we parsed from the HTML in 'citePages' array in the 'doCallForLessThan100Papers' and 'getCitationCount' function.

- The code on the next page goes thru this first step in the function

```
function totalCites()
{
  document.getElementById("loading").style.display="none";
  // Calculate the total number of citations from all fetched
  pages
  var total_citations = 0;

  for(var i = 0; i < citePages.length; i++)
  {
    var citeArray = citePages[i];
    for(var j = 0; j < citeArray.length; j++)
    {
      // Convert the string type into a numerical type
      total_citations += citeArray[j]*1;
    }
  }
  //More to be explained...
}
```

- After we sum the number of citations for the author, some calculation need to be performed to gather statistics about the author. The citations per paper can be calculated by dividing the total number of citations by the number of papers. The years of publications can be calculated by subtracting the 'maxyear' variable from the 'minyear' variable, and adding 1 to the result. Citations for year can be found by dividing the total number of citations by the number of years of publications.
- The final statistic that is calculated in the program is h-index. This is a very important calculation as it gives a number which gives a good representation of the quality of the author you searched for. It is based on the distribution of citations received by a researchers given publications. H-index works as follows:
  - Sort the number of citations from all papers. Start counting the papers from 1 to *count*. If the number of citations for paper *count* is greater than or equal to *count*, then assign the number of citations for that paper to a temporary variable (temp). If there is ever a paper count that has fewer citations than *count*, then break the look. The number that is currently stored in temp is your h-index.
- The code for these calculations can be found below. Add them to the 'totalCites' function.

```
var citeperpaper = total_citations/numOfPapers;
citeperpaper = Math.round(citeperpaper*100)/100;
var years = ((maxyear-minyear)+1);
var citeperyear = total_citations/years;
citeperyear = Math.round(citeperyear*100)/100;
var hindex = h_index();
```

- You'll also need to add a function for h-index which can be found below and also in the lab4functions.txt file. Be sure to include the 'sortNumber' function as well as the 'h\_index' function utilizes it.

```
function h_index()
{
    var hArray = new Array();
    var x = 0;
    for(var i = 0; i < citePages.length; i++){
        var citeArray = citePages[i];
        for(var j = 0; j < citeArray.length; j++){
            //Convert the string type into a numerical type
            hArray[x++] = citeArray[j]*1;
        }
    }
    hArray.sort(sortNumber);

    var hindex = 0;
    for(var i = 0; i < hArray.length; i++){
        if(hArray[i]>=(i+1))
        {
            hindex = (i+1);
        }
        else
        {
            break;
        }
    }
    return hindex;
}

function sortNumber(a,b)
{
    return b - a;
}
```

- The last piece that needs to be added into the 'totalCites' function is the code for displaying the results to the user. To do that, we'll need to use DOM (Document Object Model) to replace the text in the 'results' DIV with the calculated values. Before we do that, we'll first have to perform one pre-check. We need to check if the global 'years' variable is equal to -9999. If it is, this would mean that there was no year information on the page, and 'minyear' and 'maxyear' would have been their default variable in the calculations explained above (remember they are assigned 9000 and -1000 respectively initially). If 'years' is equal to -9999, we can just reset the 'years' variable to '-' before we show the user to make things look more professional. Add the following code to the 'totalCites' function which will create a table on the screen and display everything to the user.

```
if(years===-9999){years="-";}
var html = "<table id='resulttable'><tr><td>Total Papers: </td>
<td id='tableright'>"+numOfPapers+"</td></tr>";
html += "<tr><td id='tableleft'>Total Citations: </td> <td
id='tableright'>"+total_citations+"</td></tr>";
```

```

html += "<tr><td id='tableleft'>Citations per Paper: </td> <td
id='tableright'>"+citeperpaper+"</td></tr>";
html += "<tr><td id='tableleft'>Cited Publications: </td> <td
id='tableright'>"+publications+"</td></tr>";
html += "<tr><td id='tableleft'>h-index: </td> <td
id='tableright'>"+hindex+"</td></tr>";
html += "<tr><td id='tableleft'>Years of Publications: </td> <td
id='tableright'>"+years+"</td></tr>";
html += "<tr><td id='tableleft'>Citations per Year: </td> <td
id='tableright'>"+citeperyear+"</td></tr></table>";
html += "<br/><br/><div id='viewpapersbutton'><input
type='button' onclick='openPage();' value='View Papers by the
Author'/></div>";

```

```
document.getElementById("results").innerHTML = html;
```

- We'll also need to add some CSS code for the styling of the table. Add the following CSS code to the styles.css file to format the table properly. You should be familiar with all the CSS code that being showed. If you're unsure, refer to previous labs where CSS was covered.

```

#resulttable
{
    padding-top: 20px;
    width: 400px;
    padding-left: 5px;
    padding-right: 5px;
}

#tableright
{
    text-align: right;
    font-size: 20px;
    color: #07334e;
}

#tableleft
{
    text-align: left;
    font-size: 20px;
}

```

- We're almost finished this function. You may have noticed the following line:

```

html += "<br/><br/><div id='viewpapersbutton'><input
type='button' onclick='openPage();' value='View Papers by the
Author'/></div>";

```

- This line actually creates another DIV container, with a button inside. This button will be used to open the browser so that the user can see the results on the Google Scholar page. Let's start implementing this function after adding the CSS for this button.

```
#viewpapersbutton
{
    padding-left: 63px;
}
```

- The ‘openPage’ function will seem similar to a function we previously implemented (the ‘openCMERWebsite’ function). We’ll be performing the same action, but with a different URL. You’ll also notice the URL variable we’re using in this function is the same URL variable we’ve been saving globally in the ‘getScholarResults’ function. Add the following function into your program which is also included in the lab4functions.txt file:

```
function openPage ()
{
    var args = new
    blackberry.invoke.BrowserArguments (openpageurl) ;
    blackberry.invoke.invoke (blackberry.invoke.APP_BROWSER,
    args) ;
}
```

- Now would be a good time to test everything we’ve done in the application so far, and take a look at a result from a real query. Compile the application and load it onto the PlayBook simulator. Type in the query “KJ Rutherford” and press Search. You should get some results back for this query, as the results will have less than 100 papers. You should now see results similar to the following:

About the application Visit the CMER website

KJ Rutherford Search

This application allows you to use Google Scholar to obtain statistics about any published Author.

<b>Total Papers:</b>	<b>70</b>
<b>Total Citations:</b>	<b>1260</b>
<b>Citations per Paper:</b>	<b>18</b>
<b>Cited Publications:</b>	<b>92</b>
<b>h-index:</b>	<b>18</b>
<b>Years of Publications:</b>	<b>24</b>
<b>Citations per Year:</b>	<b>52.5</b>

[View Papers by the Author](#)

- Also try pressing the ‘View Papers by the Author’ button, and make sure that the browser opens and brings you to the Google Scholar results. If everything works, that’s great news. If not, re-trace your steps throughout the lab and make sure there aren’t any mistakes in your code.
- You may remember in lab 3 we added a call to the ‘setTimeout’ which calls a function ‘wait’ after 3000 milliseconds. The function call looked like:

```
setTimeout("wait()", 3000);
```

- Now you need to implement the 'wait' function because it will be used in the methods you implement afterwards. The 'wait' function should check if the variable 'done' is equal to true or not. If it is not equal to true, the 'wait' function should be called again after 3000 milliseconds again using the 'setTimeout' function. If 'done' is equal to true, then the 'totalCites' function should be called. Your 'wait' function should look something like the following, which is also included in the lab4functions.txt file:

```
function wait()
{
    if(done != true)
    {
        setTimeout("wait()", 3000);
    }
    else
    {
        totalCites();
    }
    return;
}
```

- Another helper function we'll need to incorporate in the program before we implement the final methods is for creating the Google Scholar URL for queries with more than one page of results. Call this function 'createUrl'. This function is also contained in the lab4functions.txt file. The function should accept two arguments. The first is the beginning result number (can either be 0, 100, 200, 300...1000). Remember that we return 100 results per page, so the beginning result can only start with one of these numbers. The second argument is the original query formatted for using with Google Scholar that was created in an earlier lab. The 'createUrl' function should look like the following, which is also in the lab4functions.txt file:

```
function createURL(begin, rq)
{
    if(begin == 0)
    {
        return
        "http://scholar.google.ca/scholar?q="+rq+"&num="+ret_
        results+"&hl=en&btnG=Search&as_sdt=1%2C5&as_sdt=on&s
        tart=0";
    }
    if(begin == 100)
    {
        return
        "http://scholar.google.ca/scholar?q="+rq+"&num="+ret_
        results+"&hl=en&btnG=Search&as_sdt=1%2C5&as_sdt=on&s
        tart=100";
    }
    if(begin == 200)
    {
        return
        "http://scholar.google.ca/scholar?q="+rq+"&num="+ret_
        results+"&hl=en&btnG=Search&as_sdt=1%2C5&as_sdt=on&s
        tart=200";
    }
}
```

```

//Code should be between here to check for 300, 400, 500,
etc.
if(begin == 1000)
{
    return
    "http://scholar.google.ca/scholar?q="+rq+"&num="+ret_
    results+"&hl=en&btnG=Search&as_sdt=1%2C5&as_sdt=on&s
    tart=1000";
}
return "";
}

```

- Notice the only difference in the return value of the IF statements is at the end of the URL where the 'start' value is changed depending on the 'begin' value sent to the function. Add this function into your code.
- Now it is time to implement the last two remaining functions, 'doCallForPagesLessThan10' and 'doCallForPagesEqualTo10'. Start with the 'doCallForPagesLessThan10' function.
- If you remember in the 'getScholarResultsPart2' function, if the number of pages in the results is less than 10, we set the global 'currentPage' variable to 0, and called the 'doCallForPagesLessThan10' function. The first thing we need to do in this function is check if 'currentPage' is less than or equal to 'pages'. 'pages' is the total number of pages returned in the query, so while we're still on a page that exists in the results, we'll need to do some work. If 'currentPage' is greater than 'pages', we should set the global variable done to true. Remember that if 'done' is equal to true, the 'wait' function will call the 'totalCites' function to complete the search.
- If 'currentPage' is less than or equal to 'pages', we should create a variable called 'begin' and set the value equal to 'currentPage \* 100'. We use this variable to call the 'createUrl' function we defined earlier.
- Call the function 'createUrl' with the 'begin' variable as the first argument and 'resultquery' as the second argument. The 'createUrl' function will return the URL we need to fetch from Google Scholar.
- Now it is time to use the jQuery 'get' function again similar to what was done before. You should also check to make sure the result returned is not null. If it is, you should again alert the user 'There was no data', hide the loading icon, and return out of the function. Your 'get' function call so far should look like the following:

```

$.get(url, function(data) {
    if(data==null)
    {
        alert("There is no data");
        document.getElementById("loading").
        style.display="none";
        return;
    }
    //Parse result from Google Scholar
})
.error(function() { alert("There was an error"); return;
});

```

- Now you'll need to call the functions you created earlier to get the number of citations as well as the minimum and maximum years for the returned articles on the page.
- After the if statement where you checked if 'data' is equal to null, make a call to 'getCitationCount' while sending 'data' as an argument to that function, and have the return value assigned to the 'citePages' array at index 'counter+'. This will allow the counter variable to increase each time we add an element to the 'citePages' array to prevent elements from being overwritten. Your function call and assignment should look like the following:

```
citePages[counter++] = getCitationCount(data);
```

- Afterwards, call the 'getYearInformation' function again sending 'data' as the argument as follows:

```
getYearInformation(data);
```

- Those two function calls will get everything we need on a single page. The next step is to move to the next page. To do that, we increase the 'currentPage' variable by 1, and call the 'doCallForPagesLessThan10' function again, with the same original argument 'resultquery'.
- Your entire 'doCallForPagesLessThan10' function should look something like the following:

```
function doCallForPagesLessThan10(resultquery)
{
    if(currentPage<=pages)
    {
        var begin = currentPage*100;
        var url = createURL(begin, resultquery);
        $.get(url, function(data) {
            if(data==null)
            {
                alert("There is no data");
                document.getElementById("loading").
                style.display="none";
                return;
            }
            citePages[counter++] = getCitationCount(data);
            getYearInformation(data);
            currentPage = currentPage + 1;
            doCallForPagesLessThan10(resultquery);
        })
        .error(function() { alert("There was an Serror");
        return; });
    }
    else
    {
        done = true;
    }
}
```

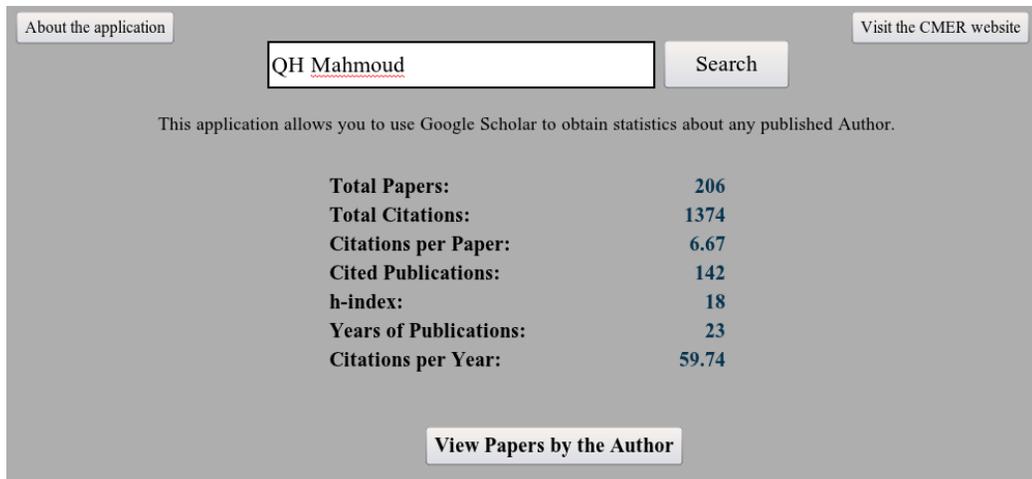
- You should be able to see how the 'doCallForPagesLessThan10' function works now. It will parse the data on the current page, increase a variable to move to the next page, and

call itself to parse the next page. This process will continue until 'currentPage' is greater than 'pages'. Once that happens, 'done' will be set to true, and the 'wait' function will then call 'totalCites' to display the results to the user.

- The process is the same for the 'doCallForPagesEqualTo10' function. The only difference is that the first IF statement will check if the 'currentPage' variable is less than or equal to 9. If it is, it will parse the page, save the results, move to the next page until all pages are completed. If not, the variable 'done' will be set to true, and the results will be displayed to the user. The 'doCallForPagesEqualTo10' function should look like the following:

```
function doCallForPagesEqualTo10(resultquery)
{
    //Starts at 0, goes to 9, therefore 10 pages
    if(currentPage<=9)
    {
        begin = currentPage*100;
        var url = createURL(begin, resultquery);
        $.get(url, function(data) {
            if(data===null)
            {
                alert("There is no data");
                document.getElementById("loading").
                style.display="none";
                return;
            }
            citePages[counter++] = getCitationCount(data);
            getYearInformation(data);
            currentPage = currentPage + 1;
            doCallForPagesLessThan10(resultquery);
        })
        .error(function() { alert("There was an error");
        return; });
    }
    else
    {
        done = true;
    }
}
```

- Now it is time to perform the final test of the application. Compile the program, and load it in the PlayBook simulator. This time, search with the query 'QH Mahmoud' which has more than one page of results. Your application should display something similar to the following:



- Again, try pressing the ‘View Papers by the Author’ button, and make sure that the browser opens and brings you to the Google Scholar results. If everything works, that’s great news. If not, re-trace your steps throughout the lab and make sure there aren’t any mistakes in your code.
- While testing the application on the simulator, you may have noticed that there is no icon for the application. If you wish, you may add an icon to your application by editing the config.xml file included in the lab. You will simply need to include a new element inside the ‘widget’ xml tags. The following XML element will set the icon of the application:

```
<icon rim:hover="false" src="images/icon.png"/>
```

- ‘src’ means source, which is referring to the source image for the icon. In the case of the example above, it is a file ‘icon.png’ which is located in the ‘images’ folder of the project. The rim:hover attribute is set to false, which specifies that we do not want the icon to change if we hover over it on the PlayBook. For simplicity, an icon.png file is already included in the images folder of the lab.
- The lab is now completed. You now have a fully functioning BlackBerry WebWorks PlayBook application which is able to communicate with an outside source on the web, parse results, and display them to the user. You can now use the skills gained in this lab create other programs which communicate with other sources to complete similar or more complex tasks.