Developing Apps for the BlackBerry Smartphone

Lab #3

Getting Started with Ajax and jQuery – Part 1

The objective of this lab is to review some of the concepts in HTML and CSS for creating WebWorks application for the BlackBerry Smartphone. In this lab, we'll be dealing with the interface of the application. Before attempting this lab, please be sure to download and install the BlackBerry WebWorks SDK for Smartphones available at https://bdsc.webapps.blackberry.com/html5/download/sdk. Make sure you that you have Java Software Development Kit installed and configured on your computer. Please also install the Ripple Emulator available at: https://bdsc.webapps.blackberry.com/html5/download/ripple. The Ripple emulator is a mobile environment emulator that is custom-tailored for mobile HTML5 application development and testing. You do not need to worry about signing your application as we'll be testing it on the emulator. Instructions on how to setup and start using the emulator can be found here: https://bdsc.webapps.blackberry.com/html5/documentation/ww_getting_started/getting_started_with_ripple_1866966_11.html. This lab also assumes that you have knowledge of basic HTML and CSS.

Communicating with External Data Sources in an HTML application:

- Use your completed code from Lab 2. Open index.html in your favorite text editor.
- Open script.js in your favorite text editor as well.
- We'll be 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 already retrieved the name of the author, converted it into the format Google requires, and created our request URL.
- Now is the time to actually send the request and read the response. To do that, jQuery will be used. jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. The code required for with jQuery is much less than the code we would require if we used pure Ajax. The most difficult parts of communication (setting the headers, etc.) are all handled by jQuery. All that is needed to implement jQuery in any web application is to include the current release version of the jQuery JavaScript file in any web page that requires it. Then it's simply jQuery function calls that are needed to perform simple tasks that would otherwise be complex.
- More information about jQuery can be found at the jQuery homepage: http://www.jquery.com.
- The jQuery JavaScript file is already included in the scripts folder of the lab, and is already included in the header of the index.html file.

```
<script type="text/javascript" src="scripts/jquery.js"></script>
```

- Now we're ready to start communicating with Google using the 'url' variable we created in the previous lab.
- For communicating with Google, we'll use the 'GET' function from jQuery. The appearance of this function call looks a little bit different than calling regular JavaScript functions. Underneath the line where we assigned the value of 'url' to 'openpageurl', create the following call to the 'GET' jQuery function. The openpageurl variable can be found in the getScholarResults function in the script.js file we worked on in the previous lab.

```
$.get(url, function(responseText)
{
        //The code for handling the response would be in here.
})
.error(function() { alert("An error has occurred"); return; })      .complete(function() {   });
```

- The first parameter is the URL of the information you wish to retrieve. The second parameter is a function that should be called when data is returned. In this case, we've made an inner function with the parameter 'responseText' which is the actual HTML response from the URL. The error and complete inner function are called if there is an error during retrieval or when the data retrieval is complete, respectively.
- The first step we should do inside this function is check if we didn't receive a response. Add JavaScript code to check if the 'responseText' is equal to null. If it is equal to null, the user should be alerted that there was no data returned. At this point, you should also set the style of the loading image DIV to 'none' and return out of the function. Your code should look something similar to the following:

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

- If there are no articles or results returned from our query, Google will include the text 'did not match any articles' on the HTML page. You can check this for yourself by typing in a name on the Google Scholar webpage which will return no results, such as 'author:ASDF author:Jules'. Try this yourself. You'll see that the page has the exact text that we're going to search for next.
- See http://scholar.google.com
- Add an if-statement inside the inner function which searches the responseText for the string 'did not match any articles'. If this string is found, the text in the 'result' DIV container should be changed to '<div id='noresults'>Your search did not return any results.</div>'. Notice here that we're adding another container inside the result DIV container. We should also set the style of the loading image DIV to 'none' and return out of the function. To set the text inside the DIV, the innerHTML attribute of the DIV container variable in JavaScript should be used. Your function should look similar to the following:

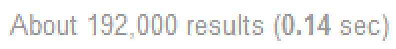
```
if(responseText.indexOf('did not match any articles')!=-1)
{
        //no results
        document.getElementById("results").innerHTML = "<div id='noresults'>Your search did
        not return any results.</div>";
        document.getElementById("loading").style.display="none";
        return;

}
```

- Now, add CSS code to adjust the position of the 'noresults' DIV container. It should have the following attributes:
    - Font size of 15px
    - Black color

    - Bold font weight
    - Top padding of 35px
    - Left padding of 5px
- Feel free to try what you've done so far in the Ripple emulator so you can see the results. Most browsers will block Ajax communication from local webpages, so you'll have to start testing from this point forward on the Ripple emulator. If you search with a query that has no results, the application should now let the user know this information.
- Before we go further with the communication, we'll need to add some more global variables for future use.
    - A global variable called 'numOfPapers' which is set to 0 by default.
    - A global variable called 'pages' which tells the application how many pages of Google Scholar results it needs to fetch. Assign this to 0 by default.
    - A global Array called 'citePages' which holds the number of citations in each paper for a specific author. Arrays are declared in JavaScript like the following:

      ```
      var citePages = new Array();
      ```

    - A Boolean global variable called 'done' which is set to false by default. This Boolean is used to tell the program when all articles are parsed. Afterwards, the total number of citations can be calculated.
    - A global variable called 'publications' which is set to 0 by default. This variable is used to calculate the total number of publications for the searched author.
    - A global variable called 'minyear' with a default value of 9000 and a variable called 'maxyear' with a default value of -1000. These two values will be used to determine the number of years of publications for an author.
    - A global variable called 'currentPage' which is used to determine which page the program is current on while parsing. This variable is also set to 0 by default.
    - A global variable called 'counter' which is set to 0 by default. This variable is used for a counter for the index of the 'citePages' Array.
- Each of these variables should be reset to their initial values at the start of the 'getScholarResults' function, so that each search starts fresh.
- Now that we've finished adding our required global variables, we can continue with the logic of our program. After the point in the jQuery get function where we checked for the text 'did not match any articles', we'll now need to find out how many results were returned from the user's search.
- If you search for an author in Google Scholar, you'll notice something like this on the upper right hand side of the web page:



About 192,000 results (**0.14** sec)

- If you look at the source code of the HTML page for that text, you'll see the following:

```
<div id="gs_ab_md">About 192,000 results (<b>0.14</b> sec)</div>
```

- This tells us how many results we have (in this case, 192000 results), and we need to extract this value for use in the program. Add the variables pre and post which searches for the HTML text before and after the number we want. Define these variables in the get function as follows:

```
var pre = <div id="gs_ab_md">About ';
var post = /\s*results\s*\(\s*<b>/;
```

- Notice that we are using regular expressions for the post variable. Now that we have our text to search for, let's find the text positions. We can find the starting point after the 'pre' text by searching for the beginning position of the text, and adding that position to the length of the 'pre' string. Similarly, we can find the starting position of the post string. Using these two values, we'll have the starting and ending position of the number we need to extract.
- A good tutorial on Regular expressions can be found at: http://www.vogella.de/articles/JavaRegularExpressions/article.html
- The length of the number we need to extract can be found by subtracting the two positional values we found, and then we can use the substring method in JavaScript to get the extract text we need. The code needed to do that can be found below:

```
var resultPositionPre = responseText.search(pre) + pre.length;
var resultPositionPost = responseText.search(post);
var resultLength = resultPositionPost - resultPositionPre;
```

- Now that we have the length and the starting position of the number we need to extract, we can use the JavaScript substring method to extract the number of results. The substring method takes two arguments, the starting position of in the string that we want to extract, and the length of what we want to extract. Since we have our starting position in the variable 'resultPositionPre' and the length of the string in the 'resultLength' variable, we can put both these in the substring method and extract the number of results.

```
var tResults = responseText.substr(resultPositionPre, resultLength);
```

- Afterwards, we need to remove the commas from the string, so that the string can be treated as a number. This occurs of we have results greater than 1000. Google Scholar will write something similar to 'Results 1 – 100 of about 1,000'. In this case, since we use a while loop to search for commas in 'tResults' and then delete them if they are found. The reason a while loop is used is because there is the possibility of more than one comma. After you used the substr method, add the following while loop:

```
while(tResults.search(',') != -1)
{
    tResults = tResults.substr(0, tResults.search(',')) + tResults.substr(tResults.search(',')+1,
    tResults.length);
}
```

- After this while loop is completed, we'll have our number of results which can be used as an integer in further calculations in the program.
- One small issue that we overlooked so far is what would occur if we only have one page of results for the author we search for. On the Google Scholar page, search with the following query: 'author: KJ author:Rutherford'. You'll see that there is about 70 results (at the time this lab was written). At the end of the url, add '&num=100'. You'll see now that all of our results are in one page. Your URL should look something like: http://scholar.google.ca/scholar?q=author%3AKJ+author%3ARutherfurd&hl=en&btnG=Search&as_sdt=1%2C5&as_sdtp=on&num=100.

- If you look at the top-right corner of the page again, you'll see that instead of 'About x results', the page now reads 'x results' instead. We need to account for this.
- There are many ways to do this, but in lab we will simply check if what we extracted from the page is a number. If it is not a number, it means that the 'search' function did not find the pre string and we need to search for a different key instead. To check if a value is not a number, we can use the JavaScript function isNaN. If the result is indeed a number, we should save the number stored in 'tResults' to the 'numOfPapers' variable, then begin parsing the HTML page (we'll make another function to do that). If the result is not a number, we should set our 'pre' variable to '<div id="gs_ab_md">', search for that string, extract the number between pre and post, save this value to the 'numOfPapers' variable then begin parsing the HTML page. Use the following IF statement to complete that task:

```
if(isNaN(tResults) == false)
{
        numOfPapers = tResults;
        getScholarResultsPart2(resultquery);
}
else
{
        //Need to search for 'of' instead of 'of about'.
        pre = '<div id="gs_ab_md">';
        resultPositionPre = responseText.search(pre) + pre.length;
        resultLength = resultPositionPost - resultPositionPre;
        tResults = responseText.substr(resultPositionPre, resultLength);
        numOfPapers = tResults;
        getScholarResultsPart2(resultquery);
}
```

- You'll see in the code provided, we call a function 'getScholarResultsPart2' with the 'resultquery' variable as our argument. Create a JavaScript function with that name which takes in a single argument called 'resultquery. If you're unsure how to do this, take a look at the function 'getScholarResults' as it also takes in one arugment.
- Now it's time to work on the 'getScholarResultsPart2' function. The first thing that should be noted is that we'll need to do slightly different things if we have more than 100 papers. If we have more than 100 papers, it also means that we have more than one page of results (because we told Google to give us 100 results per page). So, start by first checking if the number of papers is greater than 100 in the 'getScholarResultsPart2' function. You should also include an else in the IF function you created.
- Let's check our program so far to make sure everything is working. Add two alert statements in the IF statement. Alert the text 'true' if there is greater than 100 papers in the results and alert the text 'false' if there is less than 100 papers. Open your program, in the emulator and first search for 'KJ Rutherfurd'. You should see an alert that says 'false.'
    - This is correct at the time of writing, however should you not get the expected result, check through the Google Scholar webpage the number of papers.
- Now try with 'QH Mahmoud'. You should see an alert that says 'true.' If not, go through the lab again, and verify the code that you have added.
- Remove the two alert function calls you have added in the IF statement. We'll first start by working on the condition that there are more than 100 papers. We need to determine how many pages we have in the results from Google. To do that, we'll divide the number of papers for the

author (the ‘numOfPapers’ variable) by the ‘ret_results’ variable (which is the number of results per page). The result of this should be assigned to the ‘pages’ variable. Afterwards, we should set the ‘counter’ variable to 0 because that will be used in our ‘citePages’ array.

- Another IF statement will then be needed to check if ‘pages’ is less than 10. If it is, we should set the ‘currentPage’ variable to 0, and call a function which gets the results if there are less than 10 pages. Let’s call this function ‘doCallForPagesLessThan10’. This function should also take in the argument of ‘resultquery’ similar to the ‘getScholarResultsPart2’ function. For the else portion of the IF statement (which means there are 10 pages), we should again set ‘currentPage’ equal to 0, and call another function which gets the results if there are exactly 10 pages. Let’s call this function ‘doCallForPagesEqualTo10’ which again takes in the argument of ‘resultquery’.
- The ‘getScholarResultsPart2’ function should look similar to the following right now:

```
function getScholarResultsPart2(resultquery)
{
        if(numOfPapers>100)
        {
                pages = (numOfPapers)/ret_results;
                counter = 0;

            if(pages < 10)
            {
                currentPage = 0;
                doCallForPagesLessThan10(resultquery);
            }
            else
            {
                currentPage = 0;
                doCallForPagesEqualTo10(resultquery);
            }
        }
        else
        {
        }
}
```

- Create the ‘doCallForPagesLessThan10’ and ‘doCallForPagesEqualTo10’ functions with nothing inside the function bodies now.
- After the IF statement where you checked if there are less than 10 pages, we’ll need to call a function which is meant to wait until all the results are parsed before calculating the statistics to display to the user. This function call will be a little bit different than other calls we have made in the past. We’ll use the built-in JavaScript ‘setTimeout’ function which is meant to do something after a certain amount of milliseconds. Call the ‘setTimeout’ function with the parameters as follows:

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

- What this will do is call the ‘wait’ function after 3000 milliseconds, or 3 seconds. Don’t worry about implementing the ‘wait’ function now; we’ll take care of that later. Now your entire ‘getScholarResultsPart2’ function should look like the following:

```
function getScholarResultsPart2(resultquery)
{
        if(numOfPapers>100)
        {
                pages = (numOfPapers)/ret_results;
                counter = 0;
                 if(pages < 10)
                {
                        currentPage = 0;
                        doCallForPagesLessThan10(resultquery);
                }
                else
                {
                        currentPage = 0;
                        doCallForPagesEqualTo10(resultquery);
                }
                setTimeout("wait()", 3000);
        }
        else
        {
        }

}
```

- The last function call we'll add in this lab is in the else portion of the IF statement where checked if we had more than 100 papers. We'll now call a function if we have less than 100 papers. This function again will take in 'resultquery' as an argument. Let's call this function 'doCallForLessThan100Papers.' Add a call to this function in the else portion of the IF statement, and create this function in JavaScript with nothing in the function body for now. Also add another timeout to allow the 'doCallForLessThan100Papers' to finish. By the end of this lab, your entire 'getScholarResultsPart2' function should look like the following:

```
function getScholarResultsPart2(resultquery)
{
    publications = 0;
    if(numOfPapers>100)
    {
            pages = (numOfPapers)/ret_results;
            counter = 0;
     if(pages < 10)
     {
            currentPage = 0;
            doCallForPagesLessThan10(resultquery);
     }
     else
     {
            currentPage = 0;
            doCallForPagesEqualTo10(resultquery);
     }
     setTimeout("wait()", 3000);
```

```
        }
        else
        {
                doCallForLessThan100Papers(resultquery);
                setTimeout("wait()", 3000);
        }
        done =false;
}

function doCallForPagesLessThan10(resultquery){}

function doCallForPagesEqualTo10(resultquery){}

function doCallForLessThan100Papers(resultquery){}
```

- In the next lab, we'll continue implementing these functions and display the results to the user. Keep all your work from this lab for the next lab.