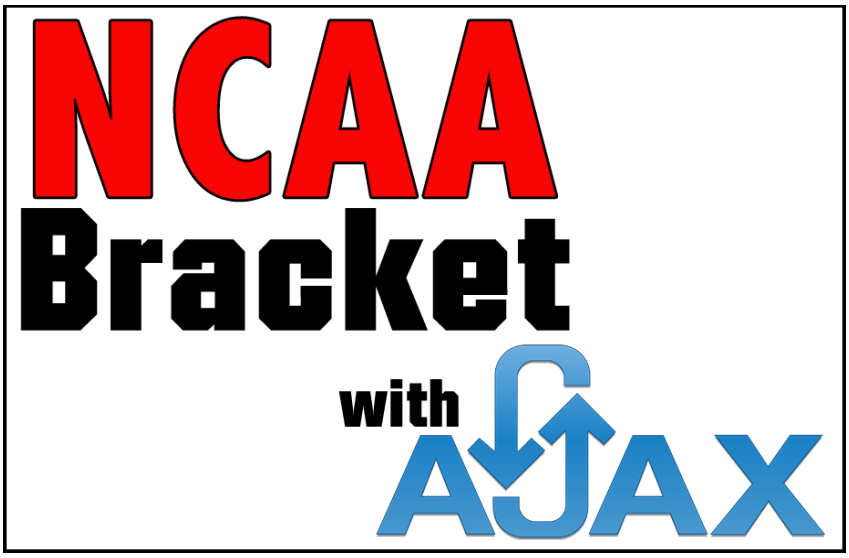


Executing Asynchronous Elements For Dynamic Data Allocation

By Chad Jordan – April 10th, 2009

Introduction

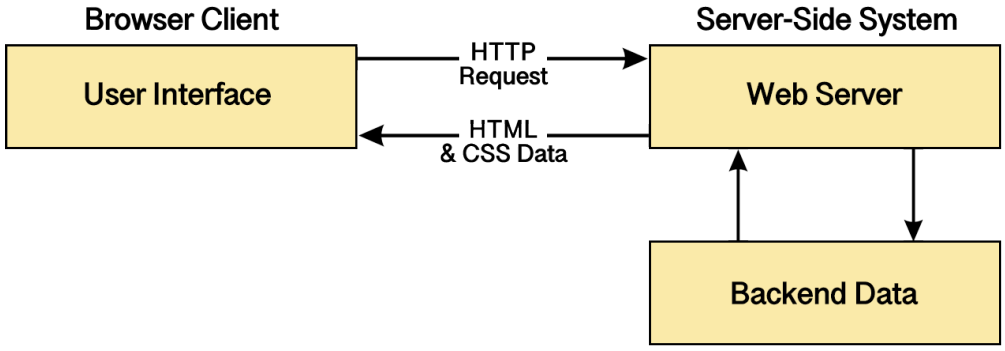
In this guide you will learn:

1. The fundamental definition and implemented uses of the AJAX technology
2. How to use AJAX to run open and send requests with Perl scripts
3. Split and replace procedures using regular expressions
4. How to implement a fully interactive NCAA bracket on a web page

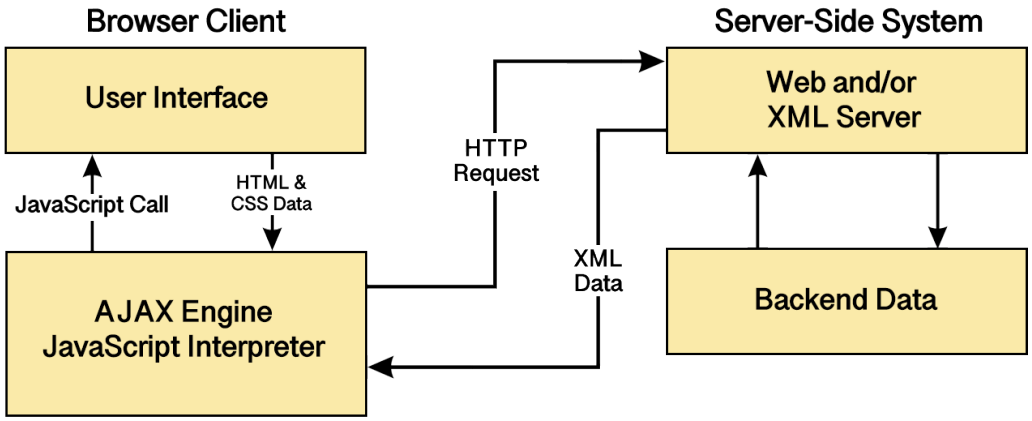
AJAX *(Asynchronous JavaScript and XML),* has currently been out for ten years, and within that span of time, it's performed faster page renders and improved response times, higher efficiency of neutrality with multi-system platforms, and architecture. AJAX is more interactive and user-friendly due to its asynchronous nature. It's been rapidly growing more in its successful UX integration throughout the web. Since AJAX allows scripts to be executed inside the user's browser to communicate with a remote server, it enables new forms of interaction for web pages and applications. Over the years, AJAX has been responsible for a lot of the technological progress on the internet. The technology has built login forms, runs auto-complete algorithms on Google searches, voting and rating functionality on Reddit, updating user content on blog pages, form submission and validation, chat rooms, instant messaging, and external widgets. In a nutshell, AJAX is known for enriching web interactions and making web applications become more like desktop applications. Because AJAX is platform-independent, we can use it beyond JavaScript with technologies such as PHP, Perl, Ruby, ASP/JSP, DOM, and of course, XHTML. In this guide, I'll be performing open and send requests with Perl, but most of the interaction is between the JavaScript, and XHTML side of the AJAX technology. For my code snippets I will be using the Vim code editor in Linux.

The AJAX Data Process

An AJAX web application has two variations from the traditional web interaction. First, the communication from the browser to the server is **asynchronous**; meaning that the browser does not need to wait for the server to respond. The user can keep using the application while AJAX connects to a script on the server which then pulls information from the database. Most AJAX code is used in conjunction with databases and the key aspect of any AJAX web app is to understand how data is passed from the frontend to the backend. The AJAX call connects to a script on your server which then pulls data from a database. This script could be written in PHP, Python, Ruby, Java, and JavaScript. In a traditional web application, HTTP requests, that are initiated by the user's interaction with the web interface, are made to a web server. The web server processes the request and returns an HTML page to the client. During HTTP transport, the user is unable to interact with the web application.



In an AJAX web application, the user is not interrupted in interactions with the web application. The AJAX engine or JavaScript interpreter enables the user to interact with the web application independent of HTTP transport to and from the server by rendering the interface and handling communications with the server on the user's behalf.



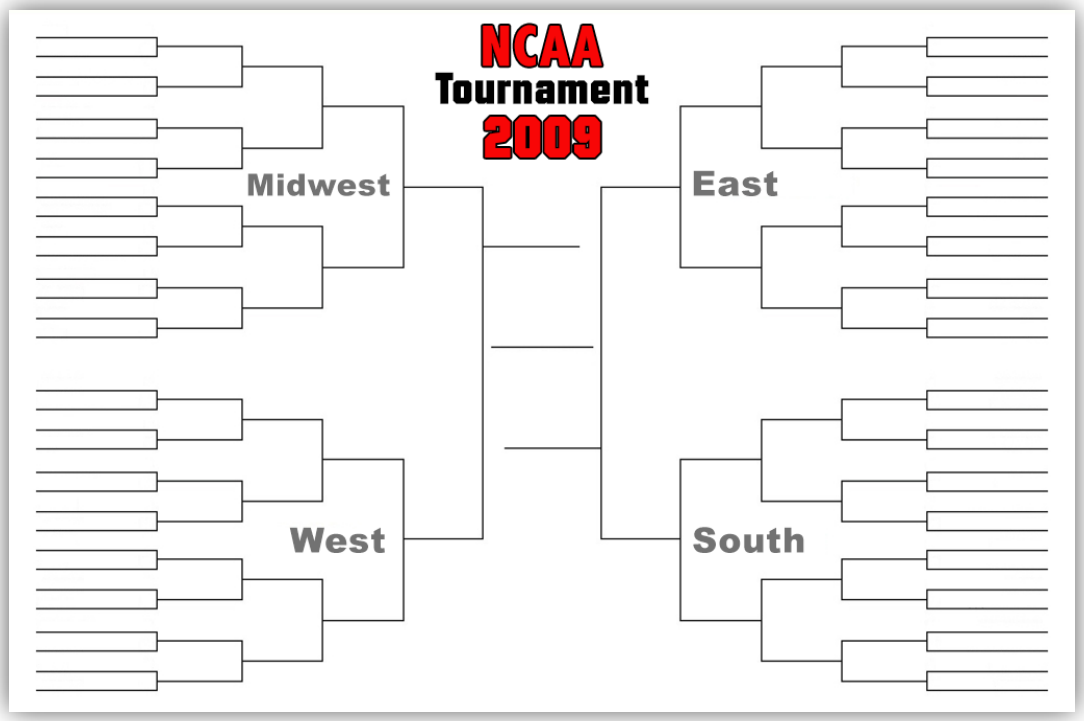
This program will gather data provided from the Perl scripts on the database, and dynamically replace the *'teams'* field with the actual team names along with the score results.

Coding the Bracket Data

I begin by creating a plain XHTML document with barely any CSS attached since the nature of this assignment is to pull data into a webpage that holds the bracket image *(bracket.jpg)*. The bracket is simply a jpeg image of an empty bracket, and it's my job to fill all of those fields with the data that has been stored on the department servers. This first block of code demonstrates the initial setup of the HTML and simply displaying the empty bracket as the background for the web page.

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
        <style>
        *{
                padding:0;
                margin:0;
        }
        body{
                background: url("Bracket.jpg")no-repeat;
        }
        p{
                position: absolute;
        }
        </style>
```

This is the image of the empty bracket that I set as the background for the app:



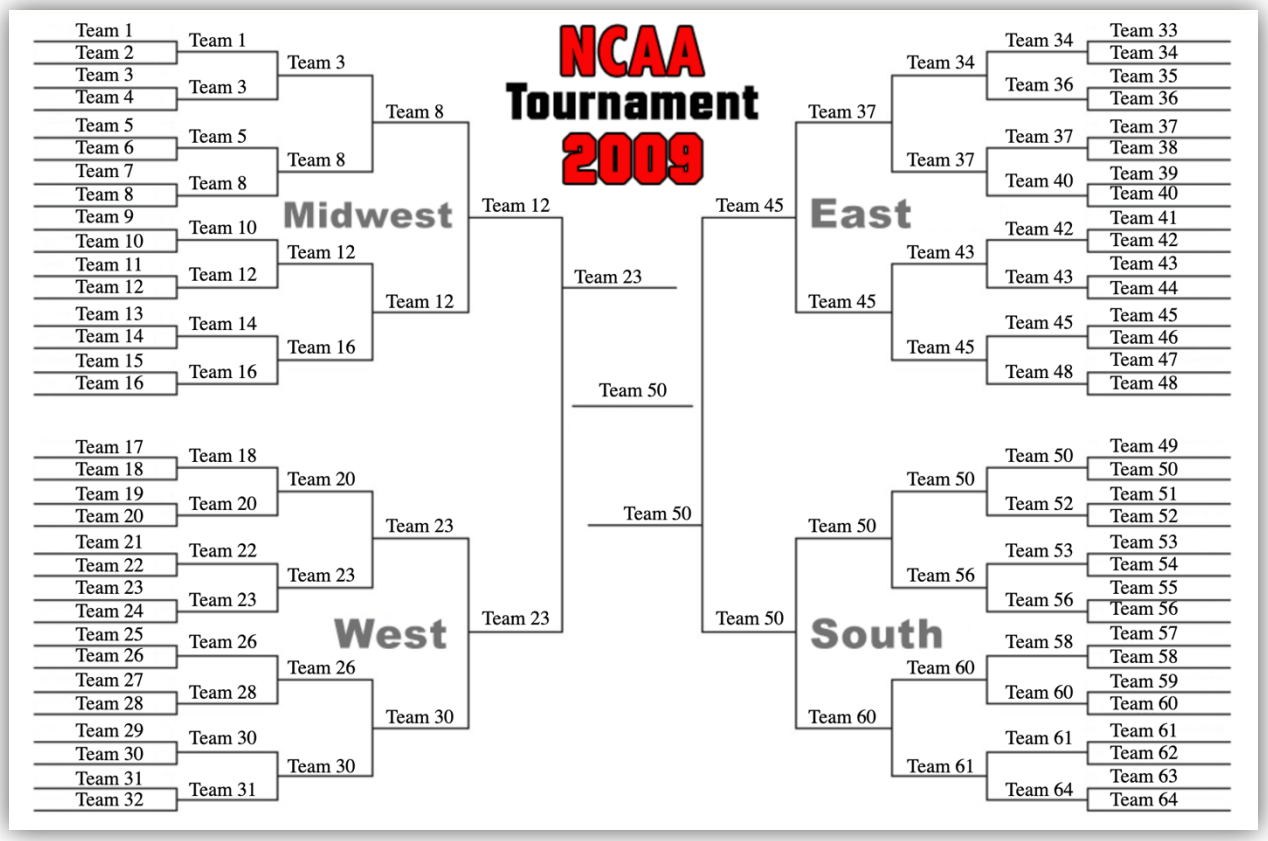
The JavaScript file, *AJAX.js* is called as a reference point for writing the bracket logic. Starting on **line 21** I do a reference call to what will be my first function in my AJAX file called *start_bracket_request*. As you can guess, upon opening the HTML file, that function will get called and run the rest of the program. Within the rest of the paragraph tags, I can provide a slot title and position for each of the temporary placeholder teams. This current block of teams covers the midwest region of the bracket.

```
18          <script type = "text/javascript" src = "AJAX.js" > </script>
19          <title> NCAA Bracket </title>
20    </head>
21    <body onload="start_bracket_request();" >
22    <p id="slot1" style="left:60px; top:8px"> Team 1 </p>
23    <p id="slot2" style="left:60px; top:26px"> Team 2 </p>
24    <p id="slot3" style="left:60px; top:44px">  Team 3 </p>
25    <p id="slot4" style="left:60px; top:62px"> Team 4 </p>
26    <p id="slot5" style="left:60px; top:85px"> Team 5 </p>
27    <p id="slot6" style="left:60px; top:103px"> Team 6 </p>
28    <p id="slot7" style="left:60px; top:122px"> Team 7 </p>
29    <p id="slot8" style="left:60px; top:141px"> Team 8 </p>
30    <p id="slot9" style="left:60px; top:159px"> Team 9 </p>
31    <p id="slot10" style="left:60px; top:179px"> Team 10 </p>
32    <p id="slot11" style="left:60px; top:198px"> Team 11 </p>
33    <p id="slot12" style="left:60px; top:216px"> Team 12 </p>
34    <p id="slot13" style="left:60px; top:238px"> Team 13 </p>
35    <p id="slot14" style="left:60px; top:256px"> Team 14 </p>
36    <p id="slot15" style="left:60px; top:276px"> Team 15 </p>
37    <p id="slot16" style="left:60px; top:294px"> Team 16 </p>
```

This next block covers the western region of the bracket so it's pretty self-explanatory what the remaining pieces of HTML will look like for the bracket.

```
41   <p id="slot17" style="left:60px; top:346px"> Team 17 </p>
42   <p id="slot18" style="left:60px; top:364px"> Team 18 </p>
43   <p id="slot19" style="left:60px; top:384px"> Team 19 </p>
44   <p id="slot20" style="left:60px; top:402px"> Team 20 </p>
45   <p id="slot21" style="left:60px; top:423px"> Team 21 </p>
46   <p id="slot22" style="left:60px; top:442px"> Team 22 </p>
47   <p id="slot23" style="left:60px; top:460px"> Team 23 </p>
48   <p id="slot24" style="left:60px; top:479px"> Team 24 </p>
49   <p id="slot25" style="left:60px; top:498px"> Team 25 </p>
50   <p id="slot26" style="left:60px; top:515px"> Team 26 </p>
51   <p id="slot27" style="left:60px; top:535px"> Team 27 </p>
52   <p id="slot28" style="left:60px; top:554px"> Team 28 </p>
53   <p id="slot29" style="left:60px; top:576px"> Team 29 </p>
54   <p id="slot30" style="left:60px; top:595px"> Team 30 </p>
55   <p id="slot31" style="left:60px; top:615px"> Team 31 </p>
56   <p id="slot32" style="left:60px; top:632px"> Team 32 </p>
```

After adding the rest of the placeholders for the eastern and southern regions, and then closing out my remaining tags, I can run the HTML file and see the following result:



Now that everything is properly aligned, I now know where my titles for each team will be positioned per the retrieved data from the Perl scripts. However, this HTML is merely displaying the text when run, and even though I now have a visual where the teams will be

displayed on the screen, I still need to program the functionality for the form, so when it runs the bracket will be empty, and thus, dynamically populating the teams as I click each empty field. With that being said, let's look at how we can create an AJAX program. The first function of this program performs the HTTP request and checks for a response to fill the bracket, and if the page detects a response from the user, then the request calls for the location of the Perl script and then places the element by tag name when there's a click.

```
1  function start_bracket_request(){
2       var request=new XMLHttpRequest();
3       request.onreadystatechange=function(){
4            if(request.readyState==4){
5                 var response=request.responseText;
6                 fill_bracket(response);
7            }
8       }
9       request.open("POST","http://cse.taylor.edu/~jgeisler/cos264/getTeams.pl",true);
10      request.send(null);
11      var ptags=document.getElementsByTagName("p");
12           for(var i=0; i<ptags.length; i++){
13                ptags[i].onclick=handle_team_click
14           }
15 }
16 function fill_bracket(bracket_data){
17      var regions=bracket_data.split(/<\/li>\s?<\/ol>\s?<\/li>\s?<li>/);
18      regions[0]=regions[0].replace(/<ul><li>/,"");
19      regions[3]=regions[3].replace(/<\/li><\/ol><\/li><\/ul>/,"");
20      for(var i=0; i<=3; i++){
21           regions[i]=regions[i].split(/<\/?..>\s?<li>/);
22      }
23      var Id=1;
24      for(var region=0; region<=3; region++){
25           for(var team=1; team<=16; team++){
26                document.getElementById("slot" + Id).innerHTML=regions[region][team];
27                     Id++;
28           }
29      }
30 }
```

If you read my previous guides on regular expressions, then you know what split and replace functions are. We have to have the ability to manipulate string data for the individual fields of each region. The regular expression on **line 17** splits between the list items in the ordered list, and then replaces the items in an unordered list in lines **18** and **19**. The for loop on **line 20** iterates through the remaining regions and performs the split procedures for the list items. The remaining code in this block iterates through the regions, and teams gets the element IDs, and places them in the appropriate slots.

This next function handles just what you would expect. Beginning on **line 32** I set a new variable to handle the team scores. *Math.ceil()* is a JavaScript rounding function that rounds a number rounded up to the nearest integer. Essentially it takes a parameter, in this case, the target id for the required string data in the slots. On the next line I set a new variable called request which will make an HTTP and XML request and then set it to the variable, *request*.

```
31 function handle_team_click(e){
32      var gamenum = Math.ceil(e.target.id.replace(/slot/,"")/2);
33      var request=new XMLHttpRequest();
34      request.onreadystatechange=function(){
35           if(request.readyState==4){
36                document.getElementById("slot" + (gamenum + 64)).innerHTML=e.target.innerHTML;
37                     if(is_full()){
38                          var request2=new XMLHttpRequest();
39                          request2.onreadystatechange=function(){
40                               if(request2.readyState==4){
41                                    var response=request2.responseText;
42                                    response=response.replace(/<p>.*<ol><li>/,"");
43                                    response=response.replace(/<\/li><\/ol><\/p>/,"");
44                                    var splitResults=response.split(/<\/li>\s*<li>/);
45                                         for(var Id=65; Id<=127; Id++){
46                                              if(splitResults[Id-65]=="true"){
```

On **line 34** the *onreadystatechange* property defines a function to be executed when the *readyState* attribute changes. The *request* property holds the status of the XMLHttpRequest object. Next, on **line 35** I have to check that the request has been sent, the server has finished returning the response and the browser has finished downloading the response content. This method is referred to as *State 4*, and is the reason why I use *request* to access the attribute, *'readyState==4'.* Doing so ensures that value 4 has executed the server HTTP request, and the AJAX call has been completed. This is why I create another variable called *request2* and check against the properties and objects for the text placement. **Line 42** begins the replace and split procedures using regular expressions for the *responseText* property. At this point, we've used split and replace functions for the data, and the text. The for loop on **line 45** iterates through all of the empty fields in the bracket and determines if the split results have properly executed for the fields. The results for the document access the data property getElementById and the winning slot is set to green, and the losing team is set to red.

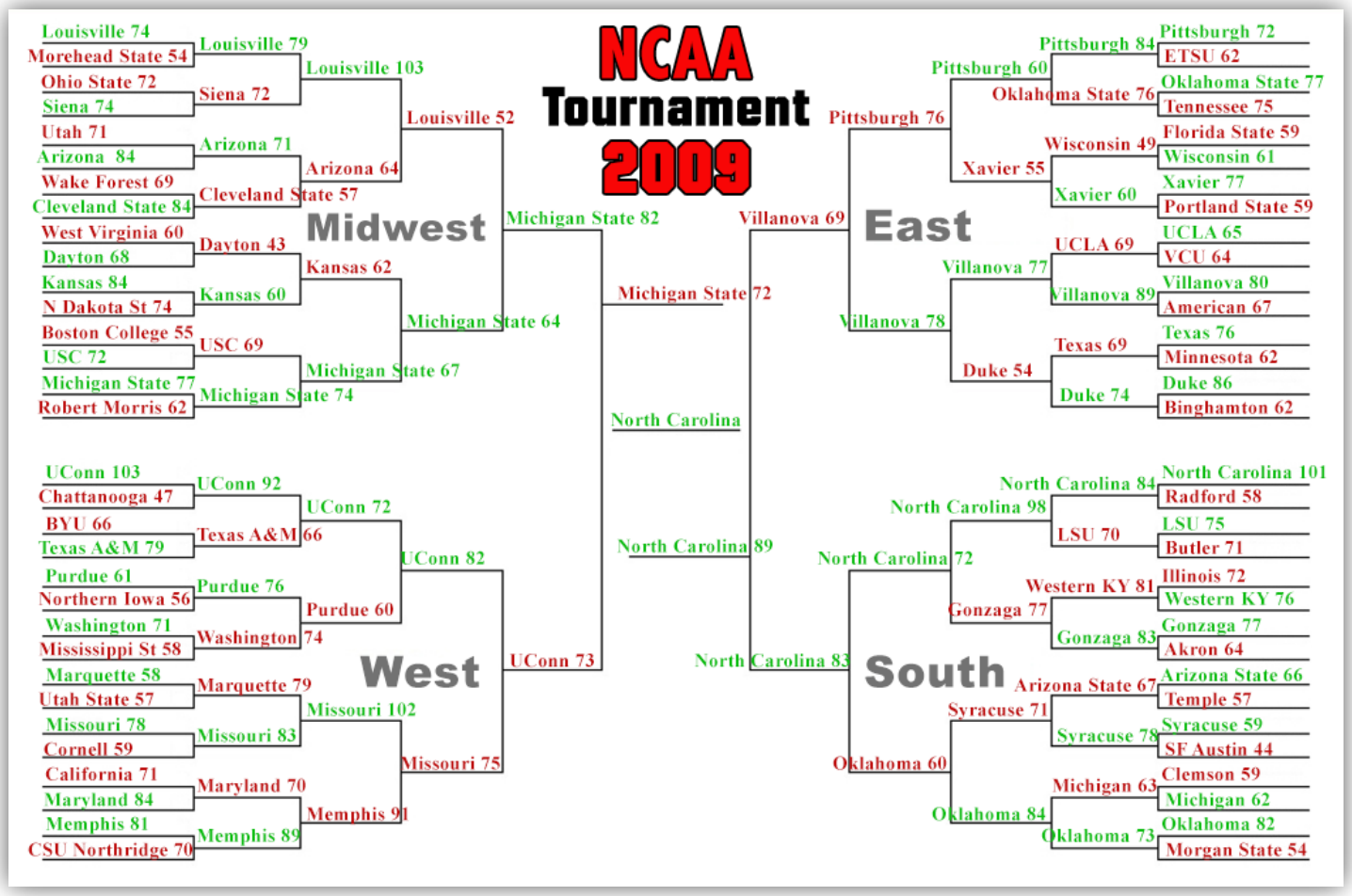
```
47                                              document.getElementById("slot" + Id).style.color="green";
48                                              }
49                                              else{
50                                                   document.getElementById("slot" + Id).style.color="red";
51                                              }
52
53                                         }
54                                    }
55                               }
56                    request2.open("POST","http://cse.taylor.edu/~jgeisler/cos264/getResults.pl",true);
57                    request2.send("user=chad");
58                    }
59          }
60     }
61        request.open("POST","http://cse.taylor.edu/~jgeisler/cos264/pickGame.pl",true);
62        request.send("user=chad&game="+gamenum+"&winner="+e.target.innerHTML);
63
64 }
```

From **lines 56** through **62** both request variables perform function calls to open and send the gathered data to the user's document.

The function, *is_full* gets the form elements by id, and adds them to the slot data strings, and places the team with the matching data attribute.

```
66 function is_full(){
67      for(var Id=65; Id<=127; Id++){
68           if(!document.getElementById("slot" + Id).innerHTML.match(/team/)){
69                return false;
70           }
71      }
72         return true;
73 }
```

This concludes the logic portion of the AJAX program, and when everything passes the JavaScript interpreter, and I run the HTML file, the final results display the following:



Looks like Christmas in more ways than one. Granted I had to make a few minor adjustments in the HTML file for pixel alignment but that's more of an issue with the supplied jpeg image. This program can no longer be tested because the Perl scripts have been pulled from the database, but essentially when the HTML file is loaded online, the fields within the bracket are entirely empty and as the user clicks each individual field, the team results auto-populate to the

respected fields of their regional locations. Other than some of the image brackets not being long enough to contain the team names, this turned out very well. The overall alignment is properly positioned, the AJAX portion of the program behaves and dynamically pulls, and loads the data from the database exactly as it's supposed to, so this is a successful AJAX implementation of an NCAA Bracket program for the web.

Conclusion

My hope is that this guide has been useful for anyone wanting to learn how AJAX works and the implementation process of integrating that technology into the web. When considering the uses of implementing more projects with AJAX, it's important to consider some of the advantages and disadvantages of using the technology. A few advantages right off the bat are AJAX applications render without data, which reduces server traffic and increases the speed with less bandwidth usage. Another is using the *XMLHttpRequest type* which seems to be a somewhat widely used technique for sending a request to AJAX pages. Another advantage is in contrast to traditional form submission, where client-side validations occur after submission, the AJAX method enables immediate form validation. These are just a few elements that improve the user's overall performance, interactions, and usability of web applications.

A few disadvantages to keep in mind, search engines like Google cannot index AJAX pages so from a users' perspective, when you click the back button on the browser, you may not return to the previous state of the page. This happens because the pages with consecutive AJAX requests are unable to register with the browser's history. Users will find it challenging to bookmark a specific state of the application due to the dynamic instance of the web page/s. Finally, even though I'm a firm believer in an open-source environment, because AJAX is open-source, this can cause security issues for the web pages that are using the technology.

Other than that, due to the ever-present changes that AJAX is helping to evolve the web and user interactions, I can see this technology continuing to be a widely-used method for web development practices everywhere. All diagrams and code snippets presented in this guide were created and written by Chad Jordan for learning purposes only. The OS that the bracket was displayed in was the Chrome browser with Ubuntu Linux 8.10, and written using the Vim code editor. For any possible inquiries such as general questions regarding this guide or other professional inquiries please feel free to email me at cjordan@wondercreationstudios.com

Resources Used:

- Sebesta, W. Robert - *Programming the World Wide Web – 4th Edition* – 2008
- W3schools.com