Steve Karam
Oracle 10g Certified Master, Oracle ACE
Sr. Oracle Consultant - Burleson Consulting and Training

Putting the Express Back Into Oracle Application Express with AJAX

Introduction

Oracle Application Express (ApEx) has all the makings of a wonderful product. It provides rapid development, it's easy to learn, and best of all, it's free. The one critical thing that has been missing from the Application Express product is speed for the end user.

While anyone can jump in, and with a weekend's time make a basic page with a report, a couple branches, and a validation or two, the more experienced developer realizes that when it comes to the client, speed is of the essence. While adding more features, processes, and pages is easy and can make for stunning applications, it ends up slowing down the experience for anyone actually using the apps. The purpose of this presentation is to show that Application Express users can have their cake and eat it too. With Application Express and AJAX, the developer can create high-content pages without the need for page refreshes in order to pull data from the database.

In this presentation, we'll be talking about an application that provides:

- A drop down list of all employees in the HR system
- A text box that narrows down the employees shown in the drop down by last name

The Old Web

The application above is easy enough to make in the old way. A drop down list would be populated on page load, optionally based on any criteria from the text box that is used to narrow down employees. When a user wishes to narrow down the list, they will simply type some characters into the text box, press Submit, and the page will refresh with a new select list that reflects their entry.

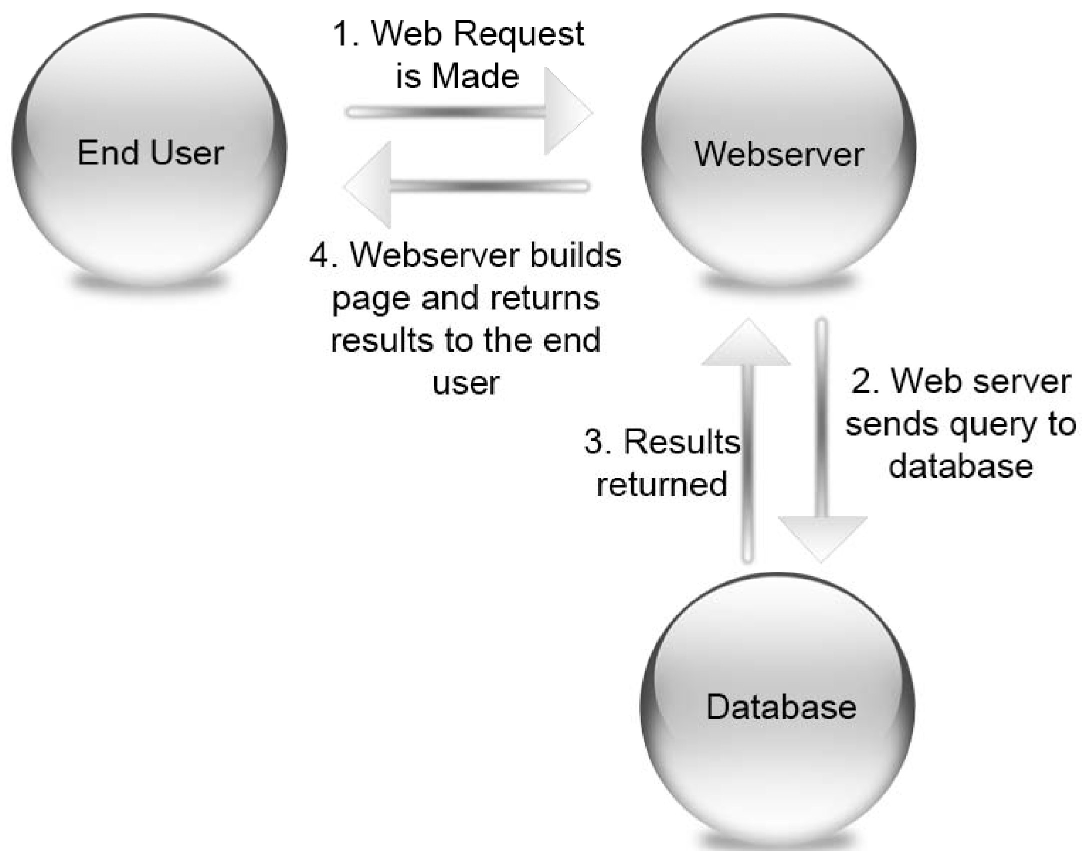
Once a user has selected an employee from the drop down list, the page would once again refresh via the onChange JavaScript call or by pressing another Submit button.

Depending on the user's selection and the way the application is designed, it will once again build a page and display the end user's choice along with any supporting information; for instance, a report on employees.

It sounds simple enough, and it works fine! We've been using this style of Internet for years, and it hasn't given us much trouble, right?

However, think of the complexities of your average web page. There are dozens of links, buttons, form items, tabs, and drop downs. A process such as buying an item from an e-store can take many pages, as can forms for complex report criteria.

Let's examine the flow of ye Olde Web:



And so we repeat again and again. Every time a button is pressed, or a user form is entered, any time data should come from the database in response to a user request, the web server must build a new page and return the result to the end user.

JavaScript

JavaScript alone can alleviate this problem to some degree. With JavaScript, developers are able to create validations to ensure that proper form elements were filled in.

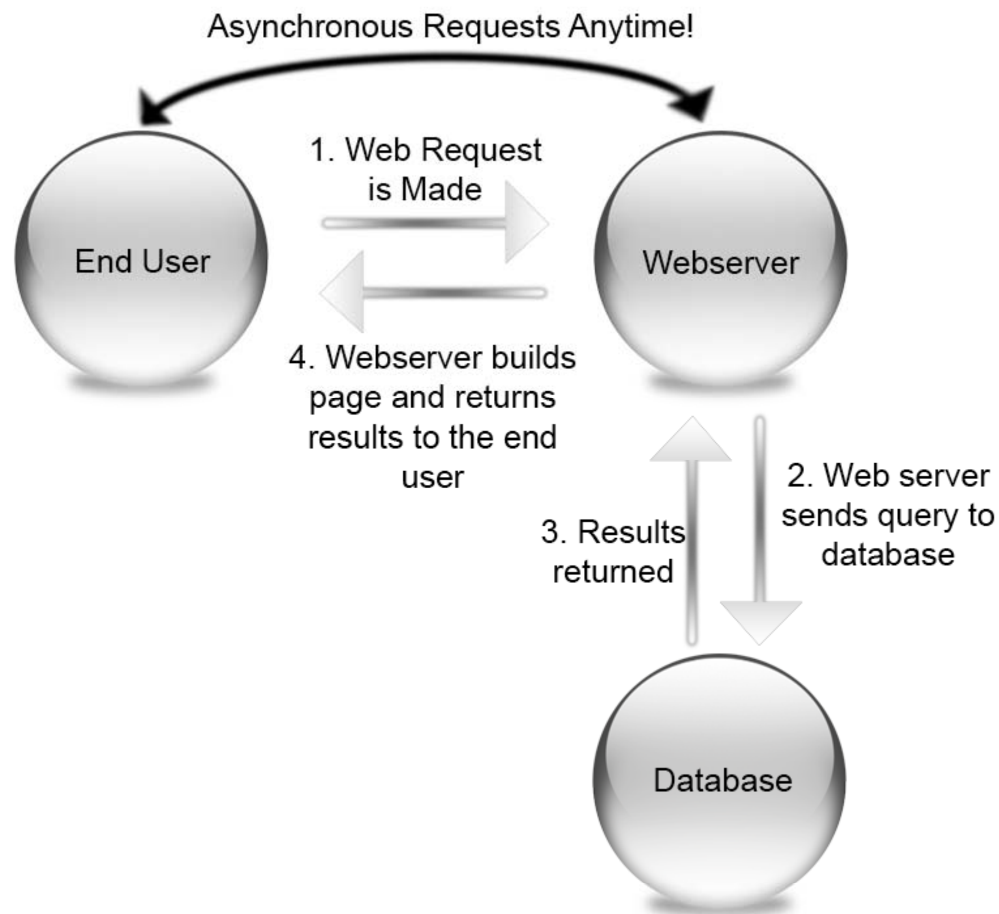
JavaScript can also be used to populate select lists dynamically, provided the data to be populated is built into the page made by the web server. It can swap images and do countless other things, but it falls short with one major fault:

JavaScript runs on the client computer.

This presents us with a major issue! We cannot dynamically pull data from the database, or read server side files, or use any other features requiring our web server or database server. If the page built by the web server does not contain everything the end user could possibly require, the page must be refreshed in order to build a new page providing additional data.

Let's Talk About AJAX

AJAX stands for Asynchronous JavaScript and XML. Asynchronous is defined as "Lack of temporal concurrence; absence of synchronism." This means we get to make our web requests and view web pages, just as we did in the old method. However, we also get to perform actions between the client and web server throughout the process! Our model will change to the following:



Note that we are still making our initial request to the web server, which calls the database and builds the page. However, the page comes packed with a very special bit of code. This code is written in JavaScript, and it allows us to make calls back to the web server to request new information while the page is already on the client's computer.

The only difficulty of this method is the data that gets returned to JavaScript. Because JavaScript is client side, it does not care what server side programming language is run: PHP, JSP, ASP, Application Express, et al. JavaScript can make a call to these languages just as it would call any other page; however, returning multiple data at a single time in an array native to JavaScript cannot always be accomplished. There are two methods of dealing with this issue:

<u>XML</u>

XML, or eXtensible Markup Language, is able to fit large amounts of data into a single stream. This steam looks like HTML, but is made up of custom nodes and attributes. A simple XML structure may look like this:

```
<EMPLOYEES>
  <EMP_ROW>
    <EMP_ID>333</EMP_ID>
    <EMP_NAME>Steve Karam</EMP_NAME>
  </EMP_ROW>
  <EMP_ROW>
    <EMP_ID>334</EMP_ID>
    <EMP_NAME>O. Racle</EMP_NAME>
  </EMP_ROW>
</EMPLOYEES>
```

This output can be returned as a single stream to the JavaScript that made the call and decoded using JavaScript's parsing components. This is where the X in AJAX comes into play.

<u>Serialization</u>

Serialization is the act of bringing data together into a single packet that can be returned to JavaScript. Technically, XML can be a form of serialization, but it is in fact a markup language and therefore is more robust. Most applications will use a serialization method such as JSON (JavaScript Object Notation) as a means of returning data to JavaScript. This method is widely used since it is very easy to both serialize (implode) and deserialize (explode). An example of JSON can be seen here:

```
“employees”: [
 {“emp_id”: “333”, “emp_name”: “Steve Karam”},
 {“emp_id”: “334”, “emp_name”: “O. Racle”}
]
```

Even though this method does not use XML, it is still called AJAX by definition. AJAS or AJAJ just doesn't have the same pizzazz.

Enough Theory! How Does Application Express Do It?

The Application Express implementation of AJAX uses a combination of JavaScript and PL/SQL for processing. The JavaScript part is performed with an object (class) built into Application Express called htmldb_Get.

The htmldb_Get object can be used to call other Application Express pages or shared processes (we'll talk about these later). Using the function, we are able to asynchronously make requests and return data to the JavaScript code. This data can be in any form we like including XML, a single string, or even generated HTML to be displayed as-is.

For instance, if we have a simple page in our application that displays a report, nothing more, like so:

Employee ID	First Name	Last Name	Email	Phone Number	Hire Date	Salary
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	17000
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	17000
114	Den	Raphaely	DRAPHEAL	515.127.4561	07-DEC-94	11000
120	Matthew	Weiss	MWEISS	650.123.1234	18-JUL-96	8000
121	Adam	Fripp	AFRIPP	650.123.2234	10-APR-97	8200
122	Payam	Kaufling	PKAUFLIN	650.123.3234	01-MAY-95	7900
123	Shanta	Vollman	SVOLLMAN	650.123.4234	10-OCT-97	6500
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	5800
145	John	Russell	JRUSSEL	011.44.1344.429268	01-OCT-96	14000
146	Karen	Partners	KPARTNER	011.44.1344.467268	05-JAN-97	13500
147	Alberto	Errazuriz	AERRAZUR	011.44.1344.429278	10-MAR-97	12000
148	Gerald	Cambrault	GCAMBRAU	011.44.1344.619268	15-OCT-99	11000
149	Eleni	Zlotkey	EZLOTKEY	011.44.1344.429018	29-JAN-00	10500
201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-96	13000
						1 - 14

We can call this page using the htmldb_Get object and pull the HTML into our JavaScript function for display. We'll see how operations such as these are accomplished soon.

First, let's cover the main components of using AJAX with APEX.

<u>The JavaScript</u>
JavaScript can be defined in many places on an APEX page. We can define it for each region in the Region Source area, or in the HTML Header area of a page's attributes.

Our JavaScript will take the form of functions that we will call during certain page actions (we'll call them Activators), which we will cover later. Take a look at this example function, used to call a shared process that returns serialized data to populate a SELECT list. The code itself is defined in a region that holds a text field called P1_EMPLOYEE_NARROW and a select list called P1_EMPLOYEES.

```
<script language="JavaScript1.1" type="text/javascript">
function getEmployeeList (narrowText, empSelect)
{
  var empSelectObj = document.getElementById(empSelect);
  var ajaxRequest = new
htmldb_Get(null,&APP_ID.,'APPLICATION_PROCESS=getEmployees',0);
  ajaxRequest.add('P1_EMPLOYEE_NARROW',narrowText.value);
  ajaxResult = ajaxRequest.get();
  if(ajaxResult)
    {
      empSelectObj.options.length = 0;
      var empArray = ajaxResult.split("~empsep~");
      for(var i=0; i < empArray.length; i++) {
        var colArray = empArray[i].split("~colsep~");
        empSelectObj.options[i] = new Option(colArray[1],
colArray[0]);
      }
    }
    else
    {
      empSelectObj.options.length = 0;
    }
  ajaxRequest = null;
}
</script>
```

This function accepts two variables, "narrowText" and "empSelect." In this case, narrowText is a textbox that will be passed along, as we shall see. empSelect is a SELECT list element. The first thing we do upon entering the function is finding and initializing the actual SELECT object itself as empSelectObj.

Once this is complete, we call the most important of JavaScript functions for AJAX within APEX: htmldb_Get.

The first argument for htmldb_Get is object_id, but this is not important for our use, and so we leave it null. The second argument is the application_id, which is contained in the &APP_ID. APEX variable. The third is the request we wish to make, which is to call an APPLICATION PROCESS called getEmployees (we will see this in detail soon). The final argument is page number, which we are not concerned with. This is used if you plan on pulling back another APEX page from the request.

Once we have initialized the htmldb_Get object as ajaxRequest, we are ready to begin. We will use the ADD method to pass variables along to the destination of the call (which

we just found out is getEmployees). We must call the "add" method with a name value pair; in this case, we pass a variable called P1_EMPLOYEE_NARROW with the value of the narrowText variable we passed into the JavaScript function.

Finally, a new object called ajaxResult is instantiated as the result of a method of ajaxRequest called get(). This method actually makes the HTTP request and pulls back the results.

The lines that follow are a small but complicated mess of splitting up the serialized object that is returned by its separators and stuffing all the values neatly into an array of options for the SELECT list.

Once you have mastered the use of htmldb_Get and its methods, you can truly do anything within APEX using AJAX. The most difficult part is figuring out just what kind of data you need to be returned, and deciding how to bring it back.

The source of the data is easy; in this case, it's the shared application process.

The Shared Application Process

These processes can be created in APEX under the Shared Components area.

Home > Application Builder > Application 19864 > Shared Components

Logic

Application Items

Application Processes

When creating a new application process, the most important part is making sure the process is set to run "On Demand: Run this application process when requested by a page process."

The beauty behind this type of process is that they are not part of any page flow. They do not require anything but basic input, and they output data via a function called htp.prn, which is simply a method of displaying data back to the process that called it. This means JavaScript can easily call it without being "attached" in any way, and asynchronously receive the return data.

In our example, we will use the following PL/SQL code for our getEmployees shared application process. If you do not remember the getEmployees process, look in the JavaScript explanation above. We supplied this name to the htmldb_Get object as the Request.

```
declare
```

```
  lv_Employee_List varchar2(32000) := '';
begin
  begin
    for i in (select employee_id, last_name || ', ' || first_name
employee_name from employees where upper(last_name || ', ' ||
first_name) like upper(:P1_EMPLOYEE_NARROW) || '%' order by
last_name, first_name) loop
      lv_Employee_List := lv_Employee_List || '~empsep~' ||
      i.employee_id || '~colsep~' ||
      i.employee_name;
    end loop;
    lv_Employee_List := substr(lv_Employee_list, 9,
length(lv_Employee_List));
  exception when no_data_found then null;
  end;
  htp.prn(lv_Employee_List);
end;
```

Here we see that our Shared Application Process is nothing more than a harmless PL/SQL block. The block begins by setting a variable called lv_Employee_List. This variable will hold our final output.

Next we begin a standard for-loop cursor, but looking at the query we can see it's not so standard after all. My horrible coding practices here are one reason (Never use functions and concatenations on the left side of a WHERE clause!), but the other is the use of the :P1_EMPLOYEE_NARROW bind variable. Where did this variable come from?

Remember in the JavaScript when we used the "add" method of htmldb_Get to define a name value pair? This is the variable we created that comes from a text box on the page itself.

For each iteration of this loop, we concatenate the query results (employee_id, last_name, first_name) into the lv_Employee_List variable using the ~empsep~ and ~colsep~ delimiters (yes, it would have been easier to just use pipes or commas). Once the loop is complete, the lv_Employee_List is cleaned up by the substr function (the first ~empsep~ is removed).

Finally, the htp.prn call is made, passing in the lv_Employee_List variable. htp.prn is nothing truly special; simply a way for a PL/SQL process to return data to "a call." In addition, there's a function called htp.p; it's just like prn except it produces a carriage return at the end of each call.

<u>The Activator</u>
All this is great, just great. However, without anything in place to call our JavaScript, we are still not getting anywhere.

This is where our activator comes in. Based on the JavaScript code and the PL/SQL Shared Application Process, you may have deduced that our code is going to use a text

field to narrow down a SELECT list. That being the case, our activator will probably be the text field (P1_EMPLOYEE_NARROW). Every time someone types text, we want to: call the JavaScript function, which in turn asynchronously calls a PL/SQL block, which returns code to the JavaScript that is put into our SELECT list without any page refreshing. Exciting, isn't it?

Do not worry; we're almost there! We already have the JavaScript in place that will take a variable and pass it on to a PL/SQL process, take the result, and populate a SELECT list. Now we just need to define the objects themselves.

We will define two objects in the same region for which I defined the JavaScript code. I called it Employee Selection, but the name is not important.

P1_EMPLOYEE_NARROW – Text Box
P1_EMPLOYEES – Select List

Don't worry about state saving for the text box or branches and redirects for the select list. Remember, we're looking for 0 refreshes!

The P1_EMPLOYEE_NARROW text box is an ordinary text box except for a single field. The "HTML Form Element Attributes" has been set to:

```
onKeyUp="javascript:getEmployeeList(this, 'P1_EMPLOYEES');"
```

The onKeyUp action says that whenever focus is on the object and we depress and release a key, the function should be called. The getEmployees function (which we defined earlier) is called, passing in "this" as the narrowText parameter and 'P1_EMPLOYEES' as the empSelect parameter. The "this" variable simply means to pass the form element making the call: in this case, the P1_EMPLOYEE_NARROW field. The P1_EMPLOYEES string is passed in, and as we saw above it will be instantiated to an object in the JavaScript with the document.getElementById method.

The Grand Finale

We now have everything we need in order to make AJAX work for us. To summarize:

- A JavaScript function that uses page elements as input and makes an asynchronous call to a remote process or page using the htmldb_Get class.
- A PL/SQL Shared Application Process that uses input from the JavaScript function and prints a serialized result back to the JavaScript code.
- An activator on a form field that calls the JavaScript when we wish for something to occur.

Using this code, we should see the following result:

Employee Selection

Narrow by Last Name

Employees: Abel, Ellen

- Abel, Ellen
- Ande, Sundar
- Atkinson, Mozhe
- Austin, David
- Baer, Hermann
- Baida, Shelli
- Banda, Amit
- Bates, Elizabeth
- Bell, Sarah
- Bernstein, David
- Bissot, Laura

Employee Selection

Narrow by Last Name: king

Employees: King, Janette

- King, Janette
- King, Steven

Employee Selection

Narrow by Last Name: king, s

Employees: King, Steven

- King, Steven

Conclusion

While other methods exist to narrow down searches, the method shown above is only a sampling of what can be produced with Application Express and AJAX. Learning how to use htmldb_Get is key; with it, you can do nearly anything.

While this walkthrough may seem complex or time consuming, it really is quite simple once you wrap your mind around it. It took roughly 10 minutes to produce the demo that I created to provide the code for this white paper. With a bit more time, you can incorporate much more, such as dynamic reports like the one below into your applications.

Employee Selection

Narrow by Last Name: king, s

Employees: King, Steven

Employee List for Managers

Employee ID	First Name	Last Name	Email	Phone Number	Hire Date	Salary
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	17000
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	17000
114	Den	Raphaely	DRAPHEAL	515.127.4561	07-DEC-94	11000
120	Matthew	Weiss	MWEISS	650.123.1234	18-JUL-96	8000
121	Adam	Fripp	AFRIPP	650.123.2234	10-APR-97	8200
122	Payam	Kaufling	PKAUFLIN	650.123.3234	01-MAY-95	7900
123	Shanta	Vollman	SVOLLMAN	650.123.4234	10-OCT-97	6500
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	5800
145	John	Russell	JRUSSEL	011.44.1344.429268	01-OCT-96	14000
146	Karen	Partners	KPARTNER	011.44.1344.467268	05-JAN-97	13500

No refreshes required! The only real change here is the htmldb_Get function. Instead of calling an application process, we called a page that contained nothing but a hidden variable for our report query and the report itself. In this case, I created a region for the report to pull into, and used this line to instantiate htmldb_Get:

```
var ajaxRequest= new htmldb_Get(null,&APP_ID.,null,2);
```

Note that in this case we do not pass an APPLICATION_PROCESS. We leave that null and instead pass a page number (2) as the final parameter. The "add" method can still be used to pass parameters to the page that's being requested.

If you ever get stuck writing AJAX code for APEX, do not despair. JavaScript can be extremely difficult to get used to if you're not accustomed to it, and some Internet software does not provide adequate resources for debugging. I would recommend Firefox if you plan on writing serious JavaScript code. In addition, if you get stuck, the folks at the Application Express forum on http://forums.oracle.com are extremely helpful. You may also want to check out Carl Backstrom's blog at http://blogs.oracle.com/carlback. His APEX demo app has plenty of great examples of the many uses of AJAX with Application Express.

About the Author

Steve Karam is a Sr. Consultant for Burleson Oracle Consulting and Training, and is responsible for troubleshooting dozens of high profile databases including RAC clusters. He is also an accomplished instructor, and teaches dozens of courses for Burleson Consulting.

Mr. Karam has the honor of being both the second Oracle 10g Certified Master in the world and an Oracle ACE. He enjoys helping others in the Oracle Community through writing, Q&A, and through meetings and events as the President of the Hampton Roads Oracle User's Group. Mr. Karam is also an avid supporter of the Oracle Academy, Oracle's high school education initiative, as he began his Oracle career while in high school. Steve also enjoys blogging on his website, http://www.oraclealchemist.com, when time permits.

Steve lives in Virginia Beach, VA with his wife and two adorable children. As a family, they enjoy the frequent travels the consulting life can sometimes demand. They also enjoy movies, outdoor fun, and driving anywhere with the top down.