Paper 61-28

DHTML – GUI on the Cheap!

Richard Phillips, The Open University, UK

ABSTRACT

Any sort of interactive SAS/IntrNet application is going to have form inputs -- a place where your users select columns to chart, row and column headings for tables, variables to subset by, and so forth. This data is passed to the back end – the SAS/IntrNet application broker.

You can bet that it doesn't appreciate having its time wasted with bogus information, and chances are the user doesn't appreciate it either. If the data the user sends contains an error, there will be a noticeable lag -- typically several seconds -- before the information travels over the Internet to the server, a SAS session is started up, and then returns an irritating failure report to the user.

With a little preliminary validation of the form content before the form is submitted, there will be no wait time. Client-side validation is instantaneous because it doesn't have to transmit any data. JavaScript catches any erroneous data entered before it goes anywhere.

But the step beyond this – to present a user with a dynamic set of valid options only – takes a little more thought…

INTRODUCTION

The basis of almost any interactive web-based application, on the client side at least, is the HTML form. In even its most basic form it allows us to collect information from the user using a number of widgets, and send them to the server for processing.

What it doesn't do, however, is to validate any user input, or provide any but the most basic interactivity within the client.

Validation can, of course, be done on the server side of an application – but this often takes time and increases network traffic. Incorporating validation in the client allows us to reduce the load on both the network and the server, and decreases user frustration by providing, for example, immediate feedback, or controlling the user's input to aid their navigation through often complex data structures.

The solution we're going to look at can be, broadly, called Dynamic HTML. Note that this isn't a language, or a standard, no RFCs exist for it. DHTML is a combination of techniques (often cited as Style, Content and Scripting) that allow us to have influence over the client-side of the web transaction.

A LITTLE HISTORY

Human/machine interfaces over the last few decades have described an arc through dumb terminals, desktop machines, client server architecture and back to dumb terminals via the web.

The increasing sophistication and maturity of the 'web-i-verse' now provides us with the ability to implement powerful, goal-oriented, and above all user friendly interfaces via this pervasive medium. The introduction of scripting capabilities within the browser allows us to 'fatten up' the traditionally thin client.

Let's consider a full-fat client development tool for a moment. In SAS/AF (may it rest in peace) a lot of effort was devoted to valid values lists, validation methods, custom access methods and event trapping. All these techniques were dedicated to input validation processing, in order to squeeze every last drop of use out of the client.

Suddenly, with the advent of HTML forms and server-side processing, the world was full of quick and dirty data gathering web tools – HTML pages that submitted their entire content to the server for processing, willy-nilly. Validation was seen as a techie job – the server side had to cope with any and all input and notify the user if there was a mistake, assuming the user hadn't moved on to another page…

One of the simplest ways to guarantee valid input and control the users' interaction with our application is to shift the client side of the application to – well, the client.

THE WEB APPLICATION

A traditional application is an enclosed user interface experience: even though window systems allow application-switching and make multiple applications visible simultaneously, the user is fundamentally "in" a single application at any given time and only that application's commands and interaction conventions are active. Users spend relatively long periods of time in each application and become familiar with its features and design.

On the Web, users move between sites at a very rapid pace and the borders between different designs (i.e., sites) are very fluid. It is rare for users to spend more than a few minutes at a time at any given site, and users' navigation frequently takes them from site to site to site as they follow the hyperlinks. Because of this rapid movement, users feel that they are using the Web as a whole rather than any specific site: users don't want to read any manuals or help information for individual sites but demand the ability to use a site on the basis of the Web conventions they have picked up as an aggregate of their experience using other sites.

The users' experience of a web *application* is something of a blend of these; they know at some level that they are in our site to obtain (for example) business information, but the whole experience is 'wrapped' in their experience of the web as a whole – they are, after all, accessing our information via their web browser.

What we really need to know is that our interface should make a user's task easier, not impede them. There is a legacy of successful (non-web) GUI design that we cannot ignore if we are to avoid putting our users through an unnecessary learning curve.

THE WEB INTERFACE

THE DOCUMENT OBJECT MODEL

We know that an HTML page isn't always rendered the same by different browsers. Fortunately for us, the objects that make up a web page all eventually break down into a common hierarchy – the Document Object Model (DOM).

In brief, the DOM is a standard representation of a document (for example HTML), which can be used by a scripting language (such as Javascript) to access values and manipulate the content of that document.

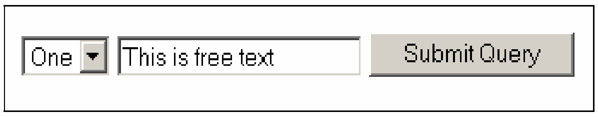
The DOM view of an HTML document is treelike. Any document has a number of 'nodes' that correspond to the elements of the document – the objects that are rendered by the browser. Technically it is a 'taxonomy' – a systematic way of referring to objects and collections of objects that shows where they lie in the structure.

AND THAT MEANS?

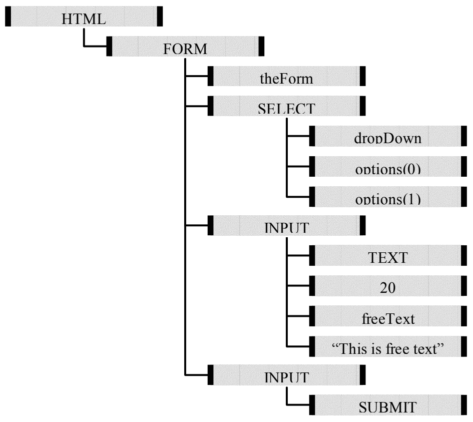
Consider this simple HTML document:

```
<HTML>
  <BODY>
    <FORM NAME='theForm'>
      <SELECT NAME='dropDown'>
        <OPTION VALUE='1'>One
        <OPTION VALUE='2'>Two
      </SELECT>
      <INPUT TYPE='TEXT'
             SIZE=20
             NAME='freeText'
             VALUE='This is free text'>
      <INPUT TYPE=SUBMIT>
    </FORM>
  </BODY>
</HTML>
```

This document has a couple of form elements to capture user input. It is rendered by a browser something like this:



To the DOM, the document is rendered as a 'tree' of objects:



This is important to us because it provides a standard way for scripts to access and manipulate the elements of an HTML document, whatever the browser. Now the page may still actually *look* different in different browsers, but we have a standard model hierarchy that allows us to use a scripting language to control elements of the interface.

SO WHERE ARE THESE VALUES?

Each form element has a number of attributes that can be accessed and/or updated by a scripting language. Dot notation is used to reference these attributes.
For example:

Reference	Return value
document.theForm.freeText.value	"This is free text"
document.theForm.dropDown.options(0).value	"1"
document.theForm.dropDown.options(1).value	"2"

In fact, we can reproduce the 'tree' above to show how the variable references are built up with dot notation (figure below).

This provides a handy way for a script to access the contents of a form. Note that the syntax for assigning values to form elements is identical – this allows the content of a document to be altered 'on the fly'…

SO WHAT'S WRONG WITH THIS FORM?

Well, nothing. When a user interacts with the form and presses 'submit', the browser will send the form contents to the location specified in the FORM tag's ACTION property. The server will process according to the form contents it receives, and return a result to the browser.

A problem arises if the user gives us garbage via the form. We *could* code our server in such a way that it traps such errors and asks for better data from the user – but by that time, the user may have been waiting around for a while. The whole transaction can add up to a pretty frustrating experience at best, and let's not forget that every time we request the server to do something, we're eating up resources.

Individually these transactions may not be a heavy load on the server – but remember a web application may be accessible from anywhere, and that fact alone means that there could be thousands of requests coming in *every second*. If we don't try and direct users properly at the browser/client end, a high proportion of our traffic could be wasted on bad requests and error reports…

EVENTS AND EVENT HANDLERS

The DOM also provides for events – these are actually methods **associated with specific objects** and allow us to capture particular parts of user interaction, and write our own functions to act as event handlers.

Types of events that can be captured include page loading, mouse click and double click, drag and drop, window operations such as resizing – all the events you would expect to be able to capture in a GUI application. They also include the `onMouseOver` and `onMouseOut` events that provide the rollover effects seen on so many web pages.

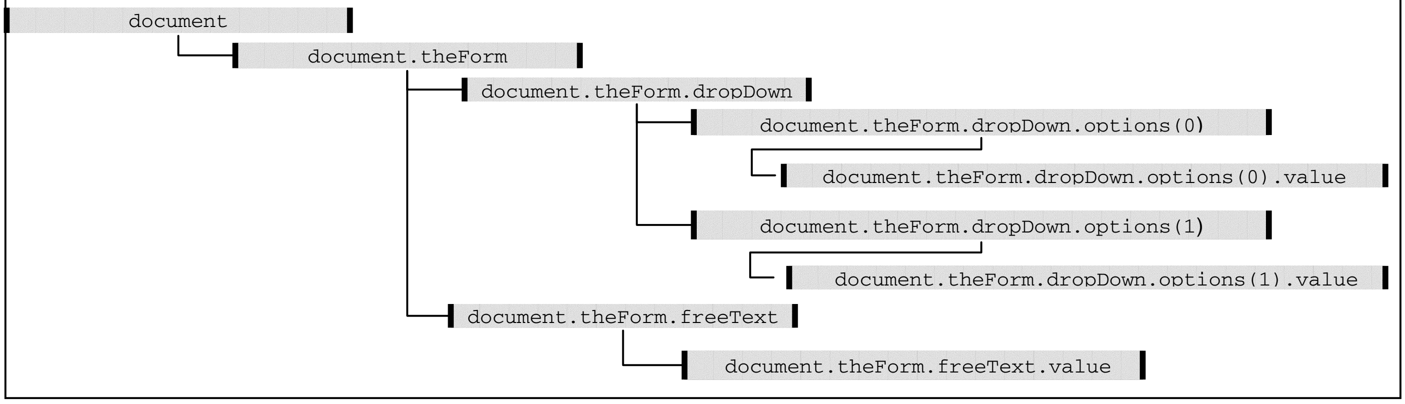
For our purposes, some of the most useful events are shown below.

Event	Triggered when
onChange	associated form element is edited by the user
onClick	associated form element receives focus
onSubmit	form is submitted to the server, but before the values are passed across

The syntax of coding for events is discussed later on.

We'll stop right there. We're not interested here in the species of DHTML that gives you whizzy effects via style sheet

Figure: dot notation for FORM values


document
document.theForm
document.theForm.dropDown
document.theForm.dropDown.options(0)
document.theForm.dropDown.options(0).value
document.theForm.dropDown.options(1)
document.theForm.dropDown.options(1).value
document.theForm.freeText
document.theForm.freeText.value

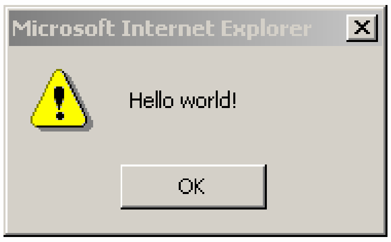
manipulation, for example – we're after the kind of enhanced GUI that is provided by client development tools such as SAS/AF, allowing us to specify valid values lists, validation methods, custom access methods and event trapping.

SCRIPTING

We've seen above how we can potentially access the 'pieces' of an HTML document. What we need now is a way to control the content and values of those 'pieces' at will.
There are actually several ways to achieve this level of control, either on the client or the server. We're only going to look at one – Javascript – for a couple of reasons.
Firstly, Javascript is non-proprietary, and in theory at least allows us to write code for all the common browsers. Other scripting languages tend towards the proprietary.
Secondly is the fact that Javascript is a *client-side* scripting language. The importance of this is that our script can 'plug in' to a document and be served to the browser right alongside the content we want to control. Also, we can rely on the browser to handle routine tasks (making buttons look like buttons – rendering list boxes and the like). This makes our job *much* easier, and also makes for compact code that hardly dents the size of the document.
Javascript is written as a series of functions with a straightforward and simple syntax. The function code is embedded within SCRIPT tags either within the HTML document or as an external file on the web server.
The language is so simple, in fact, that it is best illustrated by example. It's traditional at this point to show a 'Hello world' routine:

```
<HTML>
  <HEAD>
    <SCRIPT LANGUAGE='JavaScript'>
      function hiThere() {
        alert('Hello world!');
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <FORM>
      <INPUT TYPE='BUTTON' VALUE='Click Me'
onClick='javascript:hiThere()'>
    </FORM>
  </BODY>
</HTML>
```

This document tells the browser to render a button labeled 'Click Me', which causes a script to run when it is clicked. The script throws a dialogue box:



We can only shiver with horror at the thought of how much code would be required to do this in VB or Java.

AN EXAMPLE

Let's take a look at an old 'friend' – SASHELP.PRDSAL2.
Imagine we've been given a requirement to provide reporting of actual and predicted sales over time for individual products. We'd like to provide a simple web interface to allow users to choose a specific product and get a tabular report for a specific sales area within a country.

COUNTRY	STATE
Canada	British Columbia
	Ontario
	Quebec
	Saskatchewan
Mexico	Baja California Norte
	Campeche
	Michoacan
	Nuevo Leon
U.S.A.	California
	Colorado
	Florida
	Illinois
	New York
	North Carolina
	Texas
	Washington

Our problem in terms of the interface is that sales areas are individual states or provinces, which of course are different within each country. So a list of all possibilities would be so long as to be unusable. Also, a long list takes up too much room on our tiny screens, forcing users to scroll around to view all the possible options.
We need to control the interface so that the user can select from a list of countries, and then select from a list of areas within the selected country.

DESIRED BEHAVIOUR

When a user requests a report by loading our page, we need to interrogate them for their requirements – in this case, which sales area (`STATE`) they want the report for. We will assist their navigation by asking them to select the appropriate country, and **then** present them with a list of valid states within that country. **When a state is selected** we pass the request to the server and await the response.
From time to time the lists need to change, so some automatic method for updating the interface has to be provided for.
It can be helpful to 'storyboard' the sequence of events:

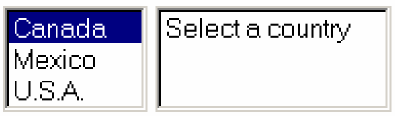
1. Present user with a list of countries.
2. When a country is selected, present the user with a list of states within the selected country.
3. When the user selects a country, pass the request to the server (*i.e.* submit the form)

STAGE 1: THE CONTENT

This is pretty straightforward as far as the values of `COUNTRY` go. We can even write a SAS program to get the distinct values of `COUNTRY` and write our HTML for us. In any case the result should look like this:

```
<HTML>
<BODY>
  <FORM name='theForm'>
    <SELECT name='countryList' size=3>
      <OPTION SELECTED>Canada
      <OPTION>Mexico
      <OPTION>U.S.A.
    </SELECT>
    <SELECT name='stateList' SIZE=3>
      <OPTION>Select a country</OPTION>
    </SELECT>
  </FORM>
</BODY>
</HTML>
```

The browser rendition:



Note that we haven't manually populated the `stateList` box, we'll see why later.

STAGE 2: THE SCRIPTING

We need to supply our form list box (named `stateList` – **Javascript is case sensitive**) with the appropriate values of `STATE` for the selected `COUNTRY`.

So, our script needs to determine which value is selected in the list box named `countryList`, and push the right values of `STATE` to `stateList`.

Remember the dot notation? We access the individual elements of `countryList` like this:

```
document.theForm.countryList.options[option-
number].text
```

Note that the first item in a list box is item 0 (zero). Fortunately for us, list boxes have another property that allows us to determine which item is selected:

```
document.theForm.countryList.selectedIndex
```

So, the text value of the item in the list box that is currently selected can be returned to our script thus:

```
function changeStates() {
var selCountry =
document.theForm.countryList.selectedIndex ;
var country =
document.theForm.countryList.options[selCountry]
.text ;
}
```

So now we know which `COUNTRY` our user is interested in. You might think it's a simple matter to write some tests for `COUNTRY` and set the `stateList` entries to the appropriate values:

```
if (country == 'Canada') {
document.theForm.stateList.options[0].text =
'British Columbia' ;
document.theForm.stateList.options[1].text =
'Ontario' ;
document.theForm.stateList.options[2].text =
'Quebec' ;
document.theForm.stateList.options[3].text =
'Saskatchewan' ;
}
```

Unfortunately, this code fails rather badly. The status line will tell us there's an error in the document:

```
document.theForm.stateList.options[1] has no
properties.
```

The reason for this is quite simple, despite the obscurity of the error message. The list box, as we created it in the original HTML, has an `options[0]` (with text 'Select a country') so we can manipulate this as we please. But later entries in the list box do not yet exist – they too are objects which need to be instantiated. The error message is telling us that `options[1]` has no properties, because there is *no* `options[1]`!

Fortunately, Javascript allows us to create new objects on the fly. Any new object we want to create has a specific method associated with it, and requires specific arguments. In this case the `Option()` method is supplied with the list item's **text**, **value**, whether the row is **selected by default**, and whether it is **currently selected**.

The correct way to manipulate our list box is shown below:

```
if (country == 'Canada') {
document.theForm.stateList.options[0]
  = new Option('British Columbia', null, false,
false);
document.theForm.stateList.options[1]
  = new Option('Ontario', null, false, false);
document.theForm.stateList.options[2]
  = new Option('Quebec', null, false, false);
document.theForm.stateList.options[3]
  = new Option('Saskatchewan', null, false,
false);
}
```

There is one pitfall with our approach so far. 'Canada' and 'Mexico' both have four states. 'U.S.A.', on the other hand, has eight. We'll see the problem with this (and its solution!) a little later.

STAGE 3: THE GLUE

Now we actually have a functioning script, we need to tie it in to the HTML so that it is run at the appropriate time. To do this we need to link the script to the document, and supply event handlers to the document to run our function.

One way to link Javascript to an HTML document is to define the function in-line, writing code between SCRIPT tags, typically in the HEAD section of the document.

```
<HTML>
<HEAD>
  <SCRIPT>
    function changeState() {
      ...
      Javascript code...
      ...
    }
  </SCRIPT>
</HEAD>
<BODY>
  <FORM name='theForm'>
      ...
      FORM elements...
      ...
  </FORM>
</BODY>
</HTML>
```

But it would be easier for our purposes of *updating* the script if it existed in a separate file. HTML allows a script file to be linked to a document:

```
<HTML>
<HEAD>
  <SCRIPT SRC='SUGI.js'></SCRIPT>
</HEAD>
<BODY>
  <FORM name='theForm'>
      ...
      FORM elements...
      ...
  </FORM>
</BODY>
</HTML>
```

This also means that the same script can be shared by many HTML documents…

So now we have the script and the document nicely tied together – except for the last, vital, part. The script may exist within the document, but it will never be called unless we specify where and when it is to run. This is where our events come in.

The specification we made earlier on says "When a country is selected, present the user with a list of states within the selected country". We have dealt with the presentation part - our script writes to the `stateList` list box. The script needs to fire when the user selects an item from the `countryList` box, so this is where we need to put our event.

```
<SELECT name='countryList' size=3
        onClick='javascript:changeStates()'>
```

The `onClick` event fires when the object with which it is associated *receives focus*. This effectively means that every time the user clicks on the `countryList` list box, our script will run.

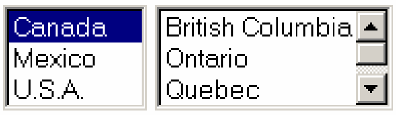
We can take the concept one stage further. Remember we didn't populate the `stateList` box in the initial HTML. We can make the box populate (because `countryList` *does* have an initial value, in this case 'Canada') by using the `onLoad` event to run our script:

```
<BODY onLoad='javascript:changeStates()'>
```

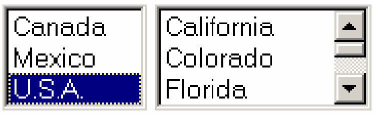
The `onLoad` event fires when the browser completes loading a document. This means that every time a user loads the HTML, the script will run and our list boxes will be populated.

A CAVEAT: TESTING AND TIDYING UP

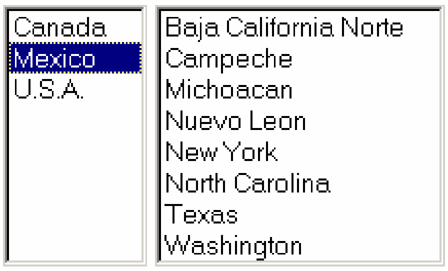
There's an unexpected twist in all this. Our document and script as described so far work fine, and when the document is loaded we see this:



All well and good. These are the Canadian states, as expected. And selecting U.S.A., we get the following result:



Great, these are the states that belong to the USA in our list. Now we click 'Mexico':

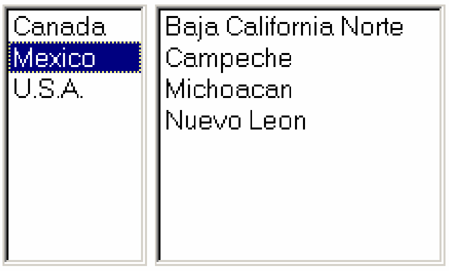


(The list boxes have been expanded to show all the values here). The first four states are our Mexican examples, but what's happened? The U.S.A. states still occupy the last four slots of the list box. The clue here is really in the question. Our script creates new rows in the list box in specific positions, regardless of the previous state of the list box. So when the user selects 'Mexico' *after* 'U.S.A.', only the first four positions are updated, and the last four remain in their current state. The easiest way to guarantee that this will not happen is to clear the stateList box of all entries *before we do anything else*:

```
document.theForm.stateList.options.length = 0 ;
```

The `length` property of the `options` object is simply the number of items/rows that exist in the list box. So, by setting this property to zero, we can clear a list box of all its entries.

Now when we select 'U.S.A.' followed by 'Mexico', we get this result (box sizes expanded again):



So far, so good.

The code we have so far does exactly what we intended. The user is presented with a list of countries, and when one is selected they are presented with a list of states within that country.

But, we have one more requirement that we haven't yet addressed. Being SAS practitioners we need this interface to be up to date with any and all updates to the data. It would be great if we could link our script in some way to the source data, so that manual updates to the script are kept to a minimum.

It would be a relatively simple matter to use `DATA _NULL_` to write the `.js` file using values from the dataset, but a glance at the script tells us that this would be fiddly at best, and prone to error.

ARRAY PROCESSING IN JAVASCRIPT

One way to divorce the data from the scripting (and therefore make the job of updating the interface in batch, using SAS, much easier) is to store all of our `COUNTRY` and `STATE` options in arrays, and use Javascript to process these arrays in order to populate our list boxes. This way we can also remove the hard-coded countries from our HTML.

Javascript array syntax is straightforward. We can declare an array of countries thus:

```
countries= new Array('Canada',
                     'Mexico',
                     'U.S.A.') ;
```

and the `STATE`s in each `COUNTRY` in the same order:

```
country1 = new Array('British Columbia',
                     'Ontario',
                     'Quebec',
                     'Saskatchewan') ;
country2 = new Array('Baja California Norte',
                     'Campeche',
                     'Michoacan',
                     'Nuevo Leon') ;
country3 = new Array('California',
                     'Colorado',
                     'Florida',
                     'Illinois',
                     'New York',
                     'North Carolina',
                     'Texas',
                     'Washington') ;
```

These arrays are declared as the first thing in our external script file.

Using the `countries` array we can now remove the hard coded options from the `countryList` list box in the HTML. Then we can write a new function to populate the list box with the array data, which we will call with the `onLoad` event handler. This new function also introduces the syntax for iterative processing in Javascript.

```
function showCountries() {
  for (var Current=0;
       Current < countries.length;
       Current++) {
    if(Current == 0) {var Selected=true}
    else {Selected=false}
  document.theForm.countryList.options[Current]
= new Option(countries[Current], null, false,
Selected) ;
        }
}
```

The 'for loop' construct takes three arguments: In order, they are the **starting value**, the **ending value**, and the **increment**. We declare a variable (`Current`) within our `for()` function so we can use the value of the iteration counter within our script. Also, this introduces the `length` property of an array – this is simply the number of elements a specified array contains, and it is used here to provide the end point of the loop.

For each iteration of the loop, this script creates a new option in the `countryList` list box, setting the `text` property of the option to the value contained in the corresponding array element.

We have to do one sneaky additional test, so that the first (top) item in the list box is selected when the box is populated. This has to be done for the population of the `stateList` list box to work, as the `changeStates()` function tests the **selected value** of `countryList()`. We select the top item in the list programmatically by testing the iteration counter, and if it is zero (*i.e.* the first time through the loop) we select the item by setting the last argument in the `Option()` method to `true`. Javascript has a number of special values at its disposal, such as `true`, `false`, `null`, and (scarily) `infinity`.

```
<HTML>
<HEAD>
<SCRIPT SRC='SUGIv3.js'></SCRIPT>
</HEAD>
<BODY
onLoad='javascript:showCountries();changeStates(
);'>
  <FORM name='theForm'>
    <SELECT name='countryList' size=8
            onClick='javascript:changeStates()'>
      <OPTION>automatic :-)</OPTION>
    </SELECT>
    <SELECT name='stateList' SIZE=8>
      <OPTION>Select a country</OPTION>
    </SELECT>
  </FORM>
</BODY>
</HTML>
```

Using the same `for()` construct, we can exploit the arrays of `STATE` values within our script. We can also remove the hard coded values of `COUNTRY`, which is A Good Thing™. We can achieve this by using just the `selectedIndex` property of the `countryList` box, since we don't actually need to compare the text strings if our values of `COUNTRY` are in the right order – which, of course, they are.

```
function changeStates() {
  var selCountry =
    document.theForm.countryList.selectedIndex ;
  document.theForm.stateList.options.length = 0;
  if (selCountry == 0) {
    for (var Current=0;
               Current < country1.length;
               Current++) {
document.theForm.stateList.options[Current] =
new Option(country1[Current], null, false,
false) ;
      }
    }
...
same for other countries
...
}
```

OBLIGATORY SAS CONTENT

This really wouldn't be a SUGI paper unless there was some SAS code. Fortunately there's an ideal opportunity here. Let's finish the job with a batch program to create a file of Javascript arrays from the SAS data set SASHELP.PRDSAL2. Then we can separate the array declarations (which will change) from the script (which won't) and the HTML (which is also static).

All our code needs to do is to extract the distinct values of `COUNTRY` and `STATE` from the data set, and write the appropriate values (surrounded by Javascript) to a file.

One way of doing this can be seen in **Appendix 1: SAS Component**.

This program creates a file (SUGIardef.js), containing our array definitions as shown above. See **Appendix 2: Array Definitions** for the output from the SAS program.

We should now remove the array definitions from our script file. For the purposes of this paper we call it SUGIscripts.js. The complete script is in **Appendix 3: Script Definitions**.

We have to make sure that our HTML includes the link to the array definitions as well. The final version of the HTML file can be seen in **Appendix 4: Content**.

CONCLUSION

Javascript is a relatively simple and powerful tool for manipulating the user interface in a web application. We can use it to reproduce many of the sophisticated aspects of a full blown client server application, both to enhance the user experience and to minimize network traffic.

Furthermore, because the scripting language is interpreted by the browser and is therefore text based, data dependent updates can be easily generated using a regular SAS program, automating as far as possible the maintenance of the system.

REFERENCES

"Building Dynamic HTML GUIs", Steven Champeon and David S. Fox, ISBN 0-7645-3267-7

www.webreference.com

developer.netscape.com/docs/manuals/js/client/jsref/

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Richard Phillips
The Open University
Walton Hall
Milton Keynes, United Kingdom MK7 6AA
+44 (1) 908 858432
R.H.Phillips@open.ac.uk
www.open.ac.uk

APPENDIX 1: SAS COMPONENT

```
proc sort data=sashelp.prdsal2 out=countries nodupkey ;
  by country ;
run ;

proc sort data=sashelp.prdsal2 out=states nodupkey ;
  by country state ;
run ;

filename script 'path-to-your-server\SUGIardef.js' ;

data _null_ ;
  set countries end=last ;
  file script ;
  if _n_ eq 1 then put "countries= new Array(" @ ;
  put "'" country +(-1) "'" @ ;
  if last then put ");" ;
  else put "," @ ;
run ;

data _null_ ;
  set states ;
  by country state ;
  file script mod ;
  if _n_ eq 1 then countryID=1 ;
  if first.country then put "country" countryID "= new Array(" @ ;
  put "'" state +(-1) "'" @ ;
  if last.country then do ;
    countryID+1 ;
     put ");" ;
  end ;
  else put "," @ ;
  retain countryID ;
run ;
```

APPENDIX 2: ARRAY DEFINITIONS

```
countries= new Array('Canada','Mexico','U.S.A.');

country1 = new Array('British Columbia','Ontario','Quebec','Saskatchewan');
country2 = new Array('Baja California Norte','Campeche','Michoacan','Nuevo Leon');
country3 = new Array('California','Colorado','Florida','Illinois','New York','North
Carolina','Texas','Washington');
```

APPENDIX 3: SCRIPT DEFINITIONS

```
function showCountries() {

     for (var Current=0;Current < countries.length;Current++) {
            if(Current == 0) {var Selected=true}
            else {Selected=false}
            document.theForm.countryList.options[Current] =
                                                   new Option(countries[Current], null, false, Selected) ;
     }
}

function changeStates() {

     var selCountry = document.theForm.countryList.selectedIndex ;
     document.theForm.stateList.options.length = 0 ;

     if (selCountry == 0) {
            for (var Current=0;Current < country1.length;Current++) {
                   document.theForm.stateList.options[Current] =
                                                   new Option(country1[Current], null, false, false) ;
            }
     }

     if (selCountry == 1) {
            for (var Current=0;Current < country2.length;Current++) {
                   document.theForm.stateList.options[Current] =
                                                   new Option(country2[Current], null, false, false) ;
            }
     }

     if (selCountry == 2) {
            for (var Current=0;Current < country3.length;Current++) {
                   document.theForm.stateList.options[Current] =
                                                   new Option(country3[Current], null, false, false) ;
            }
     }
}
```

APPENDIX 4: CONTENT

```
<HTML>
<HEAD>
     <SCRIPT SRC='SUGIardef.js'></SCRIPT>
     <SCRIPT SRC='SUGIscripts.js'></SCRIPT>
</HEAD>
<BODY onLoad='javascript:showCountries();changeStates();'>
     <FORM name='theForm'>
            <SELECT name='countryList' size=8 onClick='javascript:changeStates()'>
                   <OPTION>automatic :-)</OPTION>
            </SELECT>
            <SELECT name='stateList' SIZE=8>
                   <OPTION>Select a country</OPTION>
            </SELECT>
     </FORM>
</BODY>
</HTML>
```