Javascript E-Commerce

By Cal Henderson

In this article, we're going to be building an e-commerce system entirely in javascript. Sounds scary? Not as bad as you'd think.

The first thing we need to do is define the boundries of our system. Javascript can't communicate with the server at all (short of the redirecting-the-browser-in-hidden-frame trick) so we'll only do the work up to the point where the order needs to be logged. Our system will generate enough information to fill out hidden fields in a form which can be posted to a serverside order fulfillment system. This could be as simple as using a formmail script to send the contents of the order to the site administrator via email, or as complex as creating an order record in a database and outputing pick lists and mailing labels.

So how do we start? Well the basis of an e-commerce system is the *products* (whatever you're selling - products, services, etc. - we'll call it a product). The list of products you're selling needs to be known. We'll do this with a simple bit of javascript:

```
 var g_products = new Array();

function add_product(guid, name, price){
 var item = new Array(guid, name, price);
 g_products[g_products.length] = item;
}

add_product('item_0001', 'Product 1', 12.34);
add_product('item_0002', 'Product 2', 16.78);
add_product('item_0003', 'Product 3', 50.12);
add_product('item_0004', 'Product 4', 14.56);
add_product('item_0005', 'Product 5', 18.90);
```

So what's going on here? First we declare an array called g_products (the "g_" prefix is to remind us it's a global variable). This will have an entry for each product we have. Then we make a helper function to populate the array. The minimum that a system needs to know about each product is it's name, it's price and a unique identifier for it (I use guid here, which stands for Globally Unique IDentifier. Amazon uses an ASIN). The function makes a 3 element array of these arguments and adds it to the products array. The products array will thus consist of an array for each product. After that, we call the function a few times to populate the products array.

Before we get too bogged down in the code, we'll take a minute to think about structure. The code we're going to write will be reusable, so we'll be making a javascript file containing it all. But the product info will change from shop to shop - so it's a good idea to put this in a seperate file. This means we could generate the list of add_product() calls from a database of our products. Nice.

So now we know about the products, we'll need some way of tracking which products the user has in their basket. For this we'll use cookies, so that the basket can persist across pages. I'll assume you're familiar with using cookies in javascript, so i'll just outline the functions we'll be using:

```
 function save_cookie(name,value,days) {
function read_cookie(name) {
function delete_cookie(name) {
```

So what are we going to store in the cookies. We only need to store the quantity of each product (we can work out prcies on the fly). We could either do this as one big cookie which we parse

(cut-up) to find individual values, or we could use a single cookie for each product. Since the cookie reading code is already essentially a parser, we may as well make it easy on ourselves and use different cookies for each product.

With this in mind, we'll need two functions - one to find out the currently stored quantity for a product and one to set a new quantity. Let's call them get_qty() and set_qty():

```
var g_cookiedays = 365; //store cookies for a year

function get_qty(guid){
 var cookie = read_cookie(guid+'_qty');
 if ((cookie != null) && (cookie != '')){
  return parseInt(cookie);
 }else{
  return 0;
 }
}

function set_qty(guid, qty){
 if (qty < 0) qty = 0;
 if (!qty){
  delete_cookie(guid+'_qty');
 }else{
  save_cookie(guid+'_qty', parseInt(qty), g_cookiedays);
 }
}
```

The get_qty() function checks the cookie already exists, else returns 0. The set_qty() makes sure we have a positive quantity, then either sets the new value, or deletes the cookie - when setting the quantity to zero, we don't need a cookie, since we interpret no cookie as zero in get_qty().

So now we can store quantities of products, we need to actually be able to view how many of a product a user has. In other words, we need to construct the fabled "shopping basket" (sometimes called a "shopping cart" by those crazy americans).

Since we want it to be displayed on the page somewhere, we'll create it using a function which we'll pass the name of a parent node to create it inside (node is just a fancy term for a thing on the page). For example:

```
<body onload="build_basket('basket');">
 <div id="basket"></div>
</body>
```

So when the page loads, build_basket() is called with the id of the node we want the basket created inside.

So how do we create a basket. Well, we'll use a table since that applies itself quite well to the rows of our basket. To create a table in javascript we'll use the DOM to create a TABLE node, put a TBODY node inside it (unlike with HTML, when creating a DOM table the TBODY node is required), then several TR nodes inside that, and several TD and TH nodes inside those. To create one node and put it in another we use:

```
 var elm = document.createElement('NODE_TYPE');
parent_elm.appendChild(elm);
```

And if we want to put some text inside our new node (if it's something like a TD or TH) then we need to do this:

```
 var elm = document.createElement('NODE_TYPE');
parent_elm.appendChild(elm);
elm.appendChild(document.createTextNode('some text here'));
```

This can get tedious pretty quickly and make for huge code, so lets make some helper functions:

```
 function create_element(parent, type){
  var elm = document.createElement(type);
  parent.appendChild(elm);
  return elm;
}

function create_element_filled(parent, type, contents){
  var elm = document.createElement(type);
  parent.appendChild(elm);
  elm.appendChild(document.createTextNode(contents));
  return elm;
}
```

Just a side note here - why do we require a parent for the node straight away? IE on the mac has some strange behavoir, perhaps related to the JS garbage collector. If you add content to an un-anchored node (one which doesn't have a parent) then the node gets scrambled. To be safe, we anchor the new node before doing anything with it.

So let's use these functions to create a table...

```
 function build_basket(container_id){
  var row, cell;
  var elm_parent = document.getElementById(container_id);

  var elm_table = create_element(elm_parent, 'TABLE');
  elm_table.setAttribute('border', '1');
  elm_table.setAttribute('cellPadding', '4');
  elm_table.setAttribute('cellSpacing', '0');

  // remember we have to have a tbody!
  var elm_tbody = create_element(elm_table, 'TBODY');

  // add header row
  row = create_element(elm_tbody, 'TR');
  cell = create_element_filled(row, 'TH', 'Product');
  cell = create_element_filled(row, 'TH', 'Price');
  cell = create_element_filled(row, 'TH', 'Quantity');
  cell = create_element_filled(row, 'TH', 'Delete');
  cell = create_element_filled(row, 'TH', 'Total');
```

```
}
```

When we test this through a browser it works fine. But we're not quite there yet - we could do with showing the actual products. For a start, we can create a row for each product, regardless of the current quantity:

```
 // add product rows
for(var i=0; i<g_products.length; i++){
 var product = g_products[i];

 var guid = product[0];
 var name = product[1];
 var price = product[2];

 row = create_element(elm_tbody, 'TR');
 row.id = 'basket_row_'+guid;

 cell = create_element_filled(row, 'TD', name);

 cell = create_element_filled(row, 'TD', format_price(price));
 cell.setAttribute('align', 'right');

 cell = create_element(row, 'TD');
 cell.setAttribute('align', 'right');

 var input = create_element(cell, 'INPUT');
 input.id = 'basket_input_'+guid;
 input.value = 'QTY';
 input.size = 10;
 input.style.textAlign = 'right';
 input.onblur = update_qty;

 cell = create_element(row, 'TD');
 var link = create_element_filled(cell, 'A', 'Remove');
 link.href = "Javascript:remove_item('"+guid+"');";

 cell = create_element_filled(row, 'TD', 'TOTAL');
 cell.setAttribute('align', 'right');
}
```

This is quite a big chunk of code, so what is it doing? It loops through each element in g_products, storing the current element in a variable called product. We then extract the guid, namd and price for each product. With these in hand, we can create the row and cells for the product. The first cell just contains the name - easy. The second contains the product price. We call a function to format the price here, but we'll get to that later. The third cell contains an input box which will contain the quantity of each item. The forth cell has a link to remove that item from the basket. The fifth will contain the total for that row (the price multiplied by the quantity).

So now we have a row for each product - we'll need some way of actually showing the correct values and rows depending on the contents of the user's basket. We'll write another function for this - update_basket(). We put this in a seperate function so that we can build the basket once (calling

build_basket()) and then update it several times without refreshing the page.

```
function update_basket(){
 var sub_total = 0;

 // update product rows
 for(var i=0; i<g_products.length; i++){
  var product = g_products[i];

  var guid = product[0];
  var name = product[1];
  var price = product[2];
  var qty = get_qty(guid);

  var total = price*qty;
  sub_total += total;

  var row = document.getElementById('basket_row_'+guid);
  row.style.display = (qty > 0)?'':'none';

  // update qty
  var input = document.getElementById('basket_input_'+guid);
  input.value = qty;

  // update total
  replace_contents(row.childNodes[4],
   document.createTextNode(format_price(total)));
 }
}
```

This is pretty easy to understand. We loop through each of the products and get the quantity and calculate the total. We then get hold of the quantity input box and the total row and update the values in them. If the quantity for a row is zero then we set it's display style to 'none' to hide it, else set it blank which puts it in it's default display mode (visible). You might notice a call to replace_contents() in there - that's another helper function we'll use to update the first child of a node:

```
function replace_contents(node, newnode){
 if (node.childNodes.length > 0){
  node.replaceChild(newnode, node.childNodes[0]);
 }else{
  node.appendChild(newnode);
 }
}
```

The above routine does a little check to see if there's already a child node - if there is we replace it, else we append to the parent.

I glossed over another routine earlier which we may aswell get back to now - format_price. Because javascript doesn't have a printf style function, formatting numbers can be a real pain. The following function will format a floating point number into a price. How it works is left as an exercise for the

reader:

```
 var g_currency = '£';
var g_decimal = '.';
var g_thousand = ',';

function format_price(price){
 var s = new String(Math.round(price*100));
 var pence = s.substr(s.length-2);
 var pounds = s.substr(0, s.length-2);
 pounds = commaify(pounds);

 if (pence.length == 0) pence = '00';
 if (pence.length == 1) pence = '0'+pence;
 if (pounds.length == 0) pounds = '0';

 return g_currency + pounds + g_decimal + pence;
}

function commaify(s){
 if (s.length <= 3) return s;
 var out = s.substr(s.length-3);
 s = s.substr(0, s.length-3);
 while(s.length > 0){
  out = s.substr(s.length-3) + g_thousand + out;
  if (s.length > 3){
   s = s.substr(0, s.length-3);
  }else{
   s = '';
  }
 }
 return out;
}
```

So what else if left? Well, we have no method of adding or removing things from the basket yet. A couple of stub functions will do that for us:

```
 function add_item(guid, qty){
 set_qty(guid, get_qty(guid) + parseInt(qty));
 update_basket();
}

function remove_item(guid){
 set_qty(guid, 0);
 update_basket();
}
```

After each add/remove action we update the basket to make sure it's showing up to date values.

So now the "remove" link in the basket will remove the row from the basket (it calls remove_item). The quantity box in basket doesn't work yet though. We've set an OnBlur handler which will fire each

time the cursor leaves the box - so the handler needs to scan each box, store the new updated quantity and update the basket. This is quite easy since we have functions to do the hard bits:

```
function update_qty(){
 for(var i=0; i<g_products.length; i++){
  var product = g_products[i];

  var guid = product[0];

  var row = document.getElementById('basket_row_'+guid);
  var cell = row.childNodes[2];
  var input = cell.childNodes[0];

  set_qty(guid, input.value);
 }

 update_basket();
}
```

We loop for each product, find the input box and use the value to call set_qty(). When we're done, we call update_basket().

What else is missing? Well, it'd be nice to have a total for the basket values. We can just add another row to the basket table, then keep a running total during update_basket and store it in the new row using format_price(). And to pretty things up we'll add a "your basket is empty" row to display when there are no items in the basket. We can show and hide this in update_basket too.

So what does our page look like now?

```
<body onload="build_basket('basket');">

 <a href="Javascript:add_item('item_0001', 1)">Add product 1</a><br>
 <a href="Javascript:add_item('item_0002', 1)">Add product 2</a><br>
 <a href="Javascript:add_item('item_0003', 1)">Add product 3</a><br>
 <a href="Javascript:add_item('item_0004', 1)">Add product 4</a><br>
 <br>

 <div id="basket"></div>

</body>
```

With a bit of html and javascript magic we can also allow users to add a custom number of a product to the basket:

```
<form action="" onsubmit="add_item('item_0001', this.qty.value);
 this.qty.value=1; return false;">
 Quantity: <input type="text" name="qty" value="1" size="5">
 <input type="submit" value="Add To Basket">
</form>
```

So what else could we do? If we have the basket on a seperate page to the product listsings, then

we might want to show a mini-basket - a summary of what's in our basket. We can make another simple "inserter" function to do this:

```
function insert_basket_contents(node_name){
 var qty = 0;
 var node = document.getElementById(node_name);

 for(var i=0; i<g_products.length; i++){
  var product = g_products[i];
  var guid = product[0];
  qty += get_qty(guid);
 }

 s = (qty)?"Your basket contains "+qty+" items.":
  "Your basket is empty.";

 replace_contents(node, document.createTextNode(s));
}
```

As before, we loop through the products, summing the quantities. We then create a new text node and apend it to the given parent node. In practice, it would look like this:

```
<body onload="insert_basket_contents('basket_status');">

 <div style="background-color: #cccccc;" id="basket_status"></div>

</body>
```

So now we have a fully functioning shop - we can add things and remove things from our basket, which works out the totals for us. We can change the quantity of things in the basket. We can see a summary of the basket contents. All that's left is to have a checkout. We'll add a simple checkout link to the basket which only appears when the basket isn't empty. We can pass along a url the the build_basket() function to allow the user to choose where the checkout is.

But how will the checkout page work? This is the end of our system, as far as we're concerned. We just need to dump out the contents of the basket to a string so that it can be sent via a form. We'll create a function for that:

```
function get_order_copy(){

 var sub_total = 0;
 var items = 0;
 var buffer = '';

 // update product rows
 for(var i=0; i<g_products.length; i++){
  var product = g_products[i];

  var guid = product[0];
  var name = product[1];
  var price = product[2];
```

```
  var qty = get_qty(guid);
  items += qty;

  var total = price*qty;
  sub_total += total;

  if (qty > 0){
   buffer += name+" ("+guid+") x"+qty+" @ "+
    format_price(price)+" = "
    +format_price(total)+"\n";
  }
 }
 buffer += "\nTotal: ("+items+" items) "
  +format_price(sub_total)+"\n";

 return buffer;
}
```

And here our system ends. Well, almost. After the checkout has completed we'll want to empty the user's basket. Let's make a function for that too:

```
function empty_basket(){
 for(var i=0; i<g_products.length; i++){
  var product = g_products[i];
  var guid = product[0];
  remove_item(guid);
 }
}
```

The entire code for this article, along with a few demo pages, is available here.

(end of article)