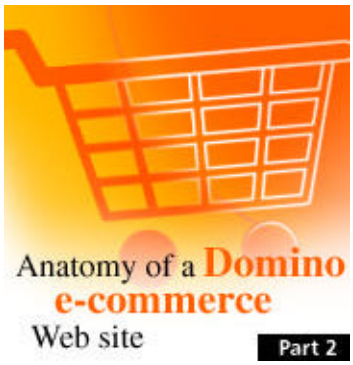
Notes.net

Iris Today



by
Michael
Patrick

Level: Intermediate
Works with: Designer 5.0
Updated: 11/01/2000

To create a successful e-commerce Web site, you have to stay focused on what's critically important—and one key component is providing customers with a simple and satisfying shopping experience. Designing an application that accomplishes this requires that you understand the big picture, but it also involves paying attention to the details that can make or break a customer's experience.

In this second article of the series, we again examine the Liberty Fund e-commerce Web site, this time exploring how it implements three important aspects of any e-commerce site: session tracking, add-to-cart shopping capabilities, and product availability notifications.

As with the first article in the series, this article references a sample database—in this case **Liberty Fund Library 2**—that you can download from the Iris Sandbox. This sample database includes the design elements discussed in Part 1 as well as those introduced here in Part 2.

This article assumes a solid understanding of how you design Notes/Domino applications using Domino Designer R5.

Understanding session tracking

E-commerce solutions of all flavors have one thing in common: through a variety of mechanisms, each customer is assigned a unique identifier so that the actions of that individual can be recognized throughout their interaction with the application. Whether this identifier is maintained at a client or server level depends on the technology behind the solution.

Domino excels at session tracking of Web users—in certain situations. Here's a typical scenario: John Doe makes a request of a database for which anonymous access is not allowed, Domino challenges Mr. Doe to authenticate, and once he has done so, Domino is then able to recognize him on successive transactions with the server. Life is good.

But e-commerce is the proverbial banana peal laying in Domino's path: public users certainly cannot be expected to authenticate prior to using your site. John Doe, while being a man of generally pleasant disposition, has come to expect instant gratification from his Web shopping experience (and why shouldn't he?) and will become quite irate if prior to filling his virtual shopping cart he must first create an account for himself and patiently wait to be added to what is by now your large and ever-expanding Domino Directory. How, then, can we keep Mr. Doe happy (hopefully "spend-happy") while simultaneously assisting Domino in remembering him whenever he initiates a server transaction?

The answer is to have the client (the Web browser in this instance) supply identification to the server with every transaction. Saddling the client with this responsibility makes for a more complex application, but only minimally so. The Liberty Fund solution makes use of two different techniques to ensure that customers continually identify themselves throughout their sessions.

Using cookies

The first technique uses cookies, which are simple to implement and

transparent to users. Cookies are small bits of information that servers can place on client machines to, among other things, identify individual users within and across sessions with that same server. In that regard, cookies have "persistence"—they hang around when the browser is shut down and are there to be accessed when it's restarted. For instance, this is what allows you to personalize content at many sites; whenever I return to such a site, they might read a cookie on my machine and from that determine who I am, heralding my return by splashing "Welcome, Mike!" across their home page.

In a similar vein, Liberty Fund takes advantage of another nice feature of cookies. When a cookie is generated for one of its customers, it is set to expire in one month. That means that for the next month, a customer is free to visit the site as often as they like, placing items in their shopping cart; and between visits their cart is maintained. When the customer returns, there is no need for them to identify themselves; we simply pull their identification out of the cookie and match it with the carts stored in our database.

So, one option available to Domino developers is to generate a unique identifier (usually a number) for each customer and store that value in a cookie on the customer's machine. Once there, each subsequent request will result in the cookie being read, allowing that customer's actions to be tracked. Again, cookies are nice because they are fairly robust. It's difficult (but by no means impossible) for users to tamper with them, and from a user's perspective, they function invisibly.

Now the bad news. Unless you've been living in a cave, you're no doubt aware of the controversy surrounding cookies. The online community seems to have divided into two camps. In one corner are the charitable souls who view cookies as a necessary evil that allows for a richer browsing experience. In the other, are those who see cookies as a means to nefarious ends and a blatant violation of privacy. That cookies are controversial isn't the main problem though; the snag lies with a user's ability to manually disable them. Poof...there goes the stored session ID.

Using JavaScript to append a session ID to links

In the absence of cookies, what other option is available? If you read the first article in this series, you're probably comfortable with appending parameters to URLs, and a session ID is no different in this regard. Once a unique ID has been generated, it can be passed from page to page via the URL. So far so good. But if you think about it for a moment, this has one unsettling implication: without exception, *every* link on *every* page will require the session ID. That's fine for links we'll build "manually" within the code itself, but what about links generated by Domino that we have no control over? Enter JavaScript. With a few short lines of code, we can cycle through all the links on a page (after the server has constructed the page, of course) and append the session ID to each.

The drawback with JavaScript is that while cookies provide us the ability to offer customers the aforementioned persistence across sessions (a shopping cart that exists for an extended period of time and so on) the JavaScript solution of appending a session ID to links does not provide anything of the sort; the session ID is good for that session only. But by using both this technique and cookies, we'll have covered the bases as much as possible.

An important caveat

And now, a caveat: Web application platforms of all flavors (and this includes Domino) rely on the client (the customer's Web browser) to provide a unique session ID to the server on successive transactions so that the server can recognize a set of actions as those of a single individual. That being the case (and Liberty Fund is no exception) cookies or JavaScript MUST be enabled by the customer's browser for this and virtually all other e-commerce applications to function as expected. If an individual has disabled both, there is no way to reliably maintain a session ID and therefore no way to distinguish one

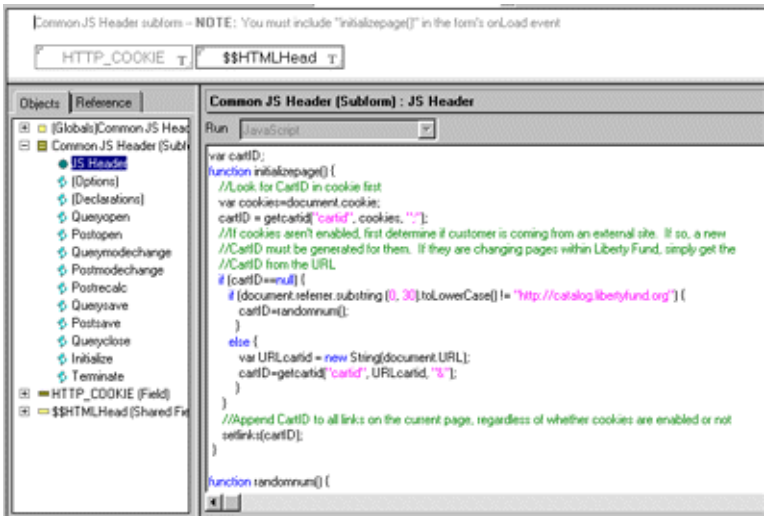
customer from another.

Unfortunately, there's no workable solution for the session ID in the case where both cookies and JavaScript are unavailable. This doesn't render an application wholly unusable, however; indeed, Liberty Fund chose to allow customers to browse their catalog even in the event they cannot maintain a session ID (and hence use a shopping cart.) We'll see in the next article how Liberty Fund handles this situation, but for now, it's important to note that while a customer's shopping experience will be curtailed without cookies and JavaScript, this does not by default leave them unable to access your catalog.

Implementing session tracking

Let's examine how session tracking is implemented in the **Liberty Fund Library 2** sample database. The heart of the solution lies with the Common JS Header subform. If you examine the database's forms in Designer, you'll notice that this subform has been included *on every form without exception.* This is done for several reasons. First, there is no guarantee as to where a customer will enter the application; if, for instance, a link was mailed to them, they might very well jump straight into a catalog entry, in which case they would still need a session ID just the same as if they had entered the site via the home page. Then there is the case I just mentioned; if the URL must transfer the session ID from page to page, that ID must be included as part of every link without fail, thus implying that the mechanism for including it on each link is part of every page/form.

Here's the Common JS Header subform in Designer:



In terms of fields, the subform is pretty simple. It consists of two computed-for-display text fields, one called HTTP_COOKIE and the other, $$HTMLHead. Both of these are marked as "Hide paragraph from Web browsers" in the Field properties box.

HTTP_COOKIE's value is simply set to HTTP_COOKIE, which is a CGI variable automatically returned by the server that contains any cookie information the browser is storing for the current site. $$HTMLHead, as always, is used for any HTML that must precede the form's BODY tag. This is typically where the HTML that creates a cookie is placed, as we'll see in a moment. Here's the formula for $$HTMLHead:

```
@If(HTTP_COOKIE=""; ""; @Return(""));

UniqueNumber := @Text(@Integer(( 1 - 16000 )*@Random + 16000)) + "-" +
@ReplaceSubstring(@Left(@Text(@Now;"D1S1");" ");":";"");
```



```
AdjDate := @Adjust(@Today;0;1;0;0;0;0);
Months:="Jan":"Feb":"Mar":"Apr":"May":"Jun":"Jul":"Aug":"Sep":"Oct":"Nov":
"Dec";
Days:="Sunday":"Monday":"Tuesday":"Wednesday":"Thursday":"Friday":
"Saturday";
Time := @Text(@Hour(@Now)) + ":" + @Text(@Minute(@Now)) + ":" +
        @Text(@Second(@Now));

ExpDate := @Subset(@Subset(Days;@Weekday(AdjDate));-1) + ", " +
@Text(@Day(AdjDate)) + "-" +
@Subset(@Subset(Months;@Month(AdjDate));-1) + "-" +
@Text(@Year(AdjDate)) + " " + Time + " GMT";

"<META HTTP-EQUIV=\"Set-Cookie\"  CONTENT=\"cartid=" + UniqueNumber
+ "; expires=" + ExpDate + "; path=/\">"
```

The first line simply checks the value returned by HTTP_COOKIE. If this value is anything other than null, we can safely assume that a cookie already exists for Liberty Fund and we won't overwrite it; the formula simply exits via the @Return(""). If no cookie was found, we'll attempt to create one.

The second line of the formula is what actually generates the unique identifier, which ends up calculating to a random number followed by a six-digit number based on the current time. When concatenated it looks something like this: 11673-025500.

Next, there are a number of lines to perform date manipulation to determine the expiration date of the cookie we'll be storing on the client machine. All cookies are required to include an expiration date, at which point they are removed from the client machine. Liberty Fund's cookies are designed to expire after one month.

The last line actually stores the cookie on the client. The cookie is created with the use of a META tag. The META tag is designed to convey "meta" information about the document in which it's included, such as keywords for search engines. It also includes the handy feature by way of its HTTP-EQUIV="Set-Cookie" property of (you guessed it) writing cookie information to a browser.

Two observations need to be made regarding the code above. First, note that while I've thus far been referring to *session tracking* and *session ID*, in an e-commerce setting these concepts are translated into the metaphor of a shopping cart, hence the "cartid=" label incorporated into the cookie. Likewise, the session identifier is referred to as the "CartID" throughout the sample database, and I'll do the same for the remainder of this article.

Second, determining client cookie support is not as simple as asking a browser, "hey, are cookies enabled?" Accordingly, aside from the check for an existing cookie (which obviates our need to add one) this code will unconditionally attempt to add a cookie via the META tag without knowing if it is successful in doing so. Why do this? Since the absence of a cookie does not necessarily imply that cookies have been disabled (an existing cookie may have expired or this may be a customer's first visit to this site), it's easiest to simply try and add one at this point. Remember, as discussed above, the solution incorporates JavaScript that compensates for instances where cookies *aren't* enabled, and there it will be simple to determine a customer's cookie status.

Speaking of which, the JavaScript is all that remains to be discussed of the Common JS Header subform. The variable declarations and functions are all included as part of the subform's JS Header object. I'll first present the code and then follow with an explanation of what it accomplishes:


```
var cartID;

function initializepage() {
	//Look for CartID in cookie first
	var cookies=document.cookie;
	cartID = getcartid("cartid", cookies, ";");
	//If cookies aren't enabled, first determine if customer is coming from an
	external site. If so, a new CartID must be generated for them. If they are
	changing pages within Liberty Fund, simply get the CartID from the URL
	if (cartID==null) {
		if (document.referrer.substring (0, 30).toLowerCase() != "
		http://catalog.libertyfund.org") {
			cartID=randomnum();
			}
		else {
			var URLcartid = new String(document.URL);
			cartID=getcartid("cartid", URLcartid, "&");
			}
	}
	//Append CartID to all links on the current page, regardless of whether
	cookies are enabled or not
	setlinks(cartID);
}

function randomnum() {
	var TodaysDate=new Date();
	var rn =
	Math.floor(16000*Math.random()+1)+"-"+TodaysDate.getHours()+
	TodaysDate.getMinutes()+TodaysDate.getSeconds();
	return rn;
}

function setlinks(v) {
	for (var i = 0; i < document.links.length; i++) {
		//Use the search property of the link object to append the CartID
		document.links[i].search=document.links[i].search + "&CartID=" +v
		}
}

function getcartid(name, inputstring, trunc) {
	//Simply parses out CartID from either the cookie or URL passed to
	function
	inputstring = inputstring + trunc;
	inputstring = inputstring.toLowerCase();
	 var start=inputstring.indexOf(name + "=");
	if (start>-1) {
		start=inputstring.indexOf("=", start)+1
	}
	var end = inputstring.indexOf(trunc, start);
	if (start==-1 || end==-1) {
		value=null
	}
	else {
		var value=unescape(inputstring.substring(start,end))
	}
	return value;
 }
```

Before examining the functions, it's important to mention that our goal with the code above is to have it execute when each page of the site loads in the browser. Of course, this will only happen when JavaScript is enabled by the browser; when it isn't, we're relying on the cookie to maintain the Cart ID. But

JavaScript functions don't call themselves; they must be explicitly executed. This being the case, in conjunction with the Common JS Header subform, each form in the database includes, as part of its onLoad object, the call:

```
initializepage()
```

You'll notice that initializepage is the first function defined in the JavaScript included above. So, whenever any Liberty Fund page loads, initializepage is called. Whether or not anything actually happens is, again, dependent on the JavaScript status of the individual browser. First, initializepage attempts to access any cookies that may exist for this site through the document.cookie property. If a cookie does exist, the CartID is parsed out via the getcartid function and then it's appended to all the links on the current page. Since the cookies exists, we could continually access the CartID from it; we don't have to append the CartID to the links, but it's a good practice to include it wherever possible. Remember, one person's paranoia is another's insurance.

When document.cookie does not return anything, the JavaScript must take control of maintaining the CartID. With one notable exception, this would be a matter of simply accessing the current URL, parsing out the CartID included there, and appending it to the current page's links. However, if Customer A sees an interesting item in the catalog and decides to mail a link to Customer B, that link includes (ominous pipe organ music, please) Customer A's CartID. This is not an issue when Customer B has cookies enabled; following the link, the application will either recognize a cookie for Customer B if one exists or it will create a new cookie, thus giving Customer B their own CartID. The problem arises when Customer B has *disabled* cookies. In this case, it's up to the JavaScript to determine the customer's CartID. This is done via the document.referrer property, which reflects the URL that called the current page. If the value returned reflects anything other than " http://catalog.libertyfund.org,"the customer has entered the catalog from another site, in which case a new CartID will be generated for them.

Before we move on, let's look at how the CartID is appended to each link on a page. This is done by the setlinks function, which is called from initializepage. The property document.links.length returns the number of links on a page, and this value is used it iterate through them with the statement:

```
for (var i = 0; i < document.links.length; i++)
```

Additionally, document.links[i] will return a single link object, and it's search property returns everything on the link following the "?". This property is editable, so we can set it to itself with the CartID appended to the end:

```
document.links[i].search=document.links[i].search + "&CartID=" +v
```

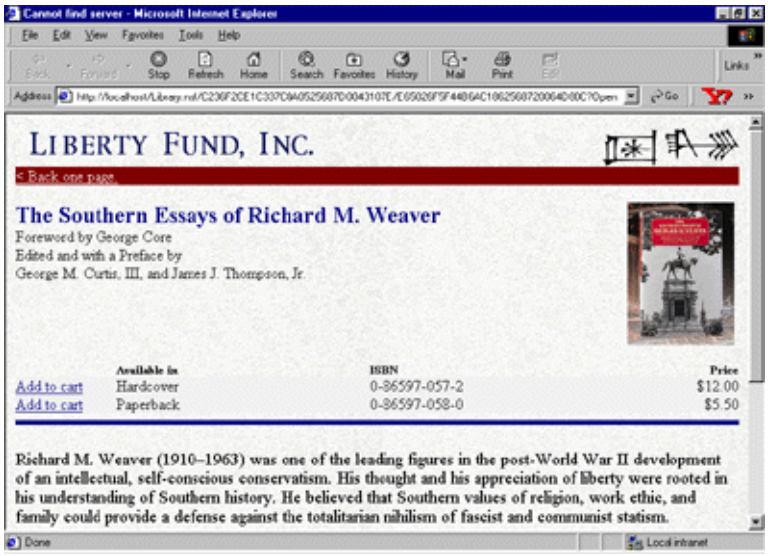
To summarize, we've just seen two distinctly independent methods that allow browsers to maintain a unique identifier in the form of a CartID. Cookies are preferable, but JavaScript is a capable substitute when cookies are not an option. Either provides Domino with the ability to perform unauthenticated session tracking—*the* crucial element in any e-commerce solution. But keep in mind that the techniques we've just examined are applicable to more than e-commerce solutions. Anytime tracking an anonymous user's actions within your site becomes necessary, cookie/JavaScript session IDs are an effective combination.

Implementing add-to-cart capabilities

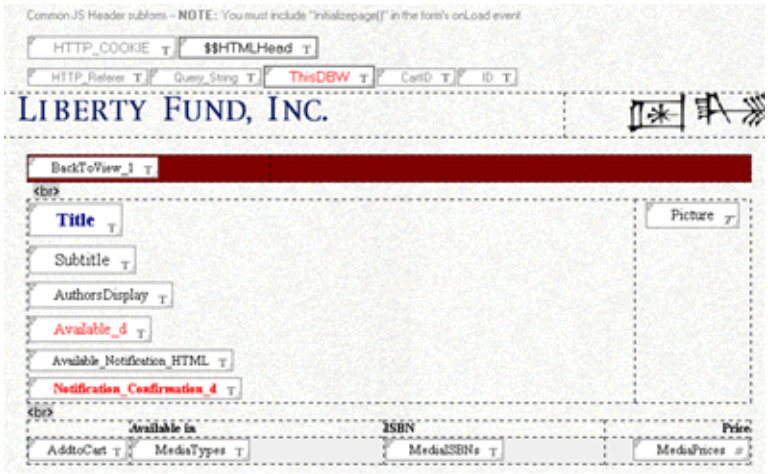
Now that we've seen how to create and maintain a Cart ID, it's time to examine the process of actually placing items into the shopping cart. In doing so, we'll see how the Cart ID is used to keep track of customer purchases.

Each item in the Liberty Fund catalog is presented via the Catalog Entry form. In the sample database there are two of these—one hidden from the Notes

client and the other hidden from the Web. Since maintenance of the catalog items is done through the Notes client and the presentation of the items is delivered to browsers, it made sense in this case to keep these two distinct functions on separate forms. Feel free to examine the Notes client version of the Catalog Entry form at your leisure; it serves to capture the details pertaining to catalog items. Its Web counterpart is where we want to focus. Before looking at the Web version in Designer, here is what a catalog item ultimately looks like via a browser:



Notice that this particular item is available in both hardcover and paperback versions, each with its own "Add to cart" link at the left. Let's look at how the links are built and how the item information is presented by taking a detailed look at the Web version of the Catalog Entry form in Designer:



Of primary interest are the four fields running the width of the form at the very bottom of the screen: AddtoCart, MediaTypes, MediaISBNs, and MediaPrices. The latter three are all multi-value, computed-for-display fields—MediaTypes and MediaISBNs are text and MediaPrices is numeric so that its format can be set to Currency—and their formula values are simply the same as the name of their respective fields. For instance, the value of MediaTypes is "MediaTypes," which is a field stored in the document for this catalog item. Since the multi-value separator for each field is "New Line," the values appear as separate line items; the publication pictured above contains the values "Hardcover" and "Paperback" in its MediaTypes field, both of which appear on

their own lines. Likewise, MediaISBNs and MediaPrices each contain two values, the first value corresponding to the hardcover item and the second adding detail to the paperback item. If there were three items under this same title, each field would contain three values, and so on.

The same holds true for the AddtoCart field, which builds the "Add to cart" links. It too, is a multi-value, computed-for-display text field whose separator is set to "New Line," but it's formula is a little more complex:

```
"[<A HREF=\"/" + ThisDBW + "/AddtoCart?OpenAgent&ISBN=" + MediaISBNs
+ "\"><FONT FACE=\"Times New Roman\" COLOR=\"#000080\" SIZE=3
onMouseover=\"this.color=\'#800000\';\"
onMouseout=\"this.color=\'#000080\';\">Add to cart</FONT></A>]"
```

Thanks to the list processing power of Notes, when a multi-value field is included as part of the value returned by a formula, an instance of that value will be returned for each of the multi-value field's values. In the example above, this results in a unique link being constructed for each of the MediaISBNs values (and, again, there are two of them in the example above).

Breaking down the formula is a straightforward process. The beginning of the link's HREF includes a reference to the ThisDBW field, which is included at the top of the Catalog Entry form and evaluates to the current database's path. ThisDBW's formula is:

```
@ReplaceSubstring (@Subset (@DbName; -1); "\\" : " "; "/" : "+")
```

This replaces Domino's "\\" with a URL-friendly "/" and does the same for spaces, replacing them with "+." Standard stuff.

It's what comes after ThisDBW that deserves our attention. You can probably guess from the "AddtoCart?OpenAgent" portion of the link that an agent called AddtoCart will be called. This agent will "place" individual items in the shopping cart. How does it know *what* to place in the cart? That's where the parameter "&ISBN=" comes in; each link will have one of the values in MediaISBNs appended to it. The AddtoCart agent will then use that value to find the item in the database.

When I indicated the AddtoCart agent would "place" an item in the cart, I used quotation marks because to the user that's exactly what appears to happen, but in reality, this agent creates an "order item" document in the database. This new document stores the Cart ID of the customer and the specifics of the item—its ISBN, media type, price, and the quantity being ordered. In the event that an order item document already exists for this customer and item, the quantity will merely be increased by one. So, for each unique item a customer orders, a separate document is created to capture details relating to that item. Web agents are typically marked to run "Manually From Agent List" and as "Run once" under "Which document(s) should it act on?" and this one is no exception. You can go to the **AddtoCart agent** sidebar to see the complete code for this agent, or check out the **Liberty Fund Library 2** sample database.

The agent's last line:

```
Print "[" + vPath(0) + "/cart?ReadForm&CartID=" + vCartID(0) + "]"
```

actually opens a customer's cart and displays the items it contains. This is accomplished by use of the LotusScript Print statement, which when combined with square brackets, tells the browser to redirect to the enclosed URL. In this case, the redirection opens the database's Cart form. This form will be covered in the third installment of the series (and will be included with the sample database for that article), so suffice to say that its function is to collect all the documents matching the customer's Cart ID and then supply the

traditional functions of a shopping cart—the display of item information, allowing changes to quantities, providing a checkout mechanism, and so on.

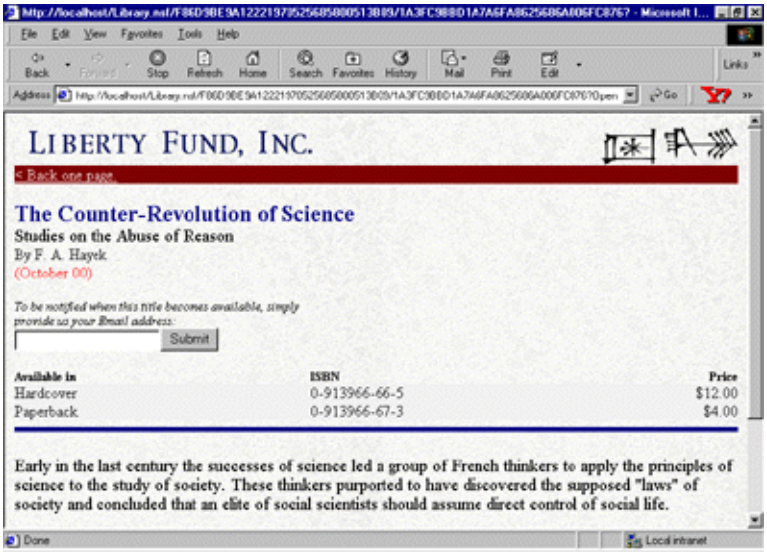
Implementing availability notifications

Before we wrap up, let's take a look at another important function the Catalog Entry form includes.

Any good e-commerce site must capitalize on opportunities to draw customers back for subsequent visits. One good way to do this is to notify customers via email when an item in which they're interested becomes available. This allows a site to preview upcoming products and generate interest in them, not to mention serving as a nice convenience for customers.

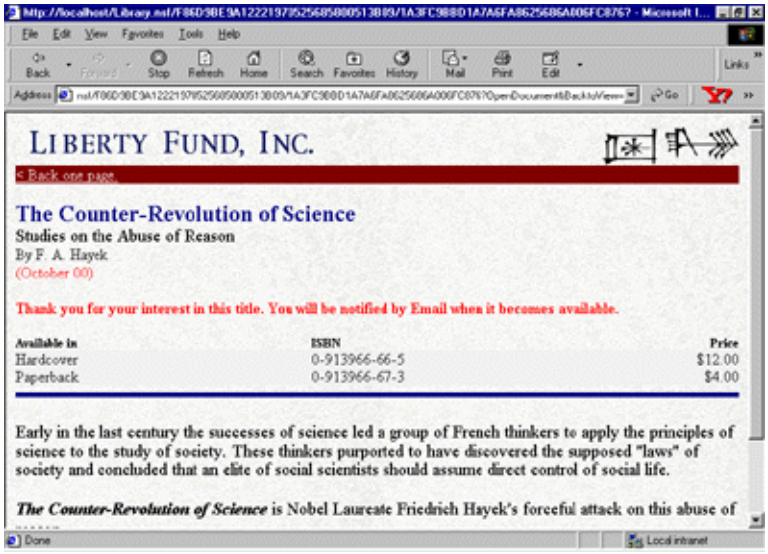
This functionality is incorporated into the Liberty Fund site through the Web version of the Catalog Entry form, but there are a number of elements that make availability requests work. In fact, you'll notice some striking similarities between this code and that of the catalog search functionality explained in **Part 1** of this series. That technique was treated in detail there, so where these two features overlap, I'll be brief in my explanation.

First take a look at an item from the catalog that is not yet available:

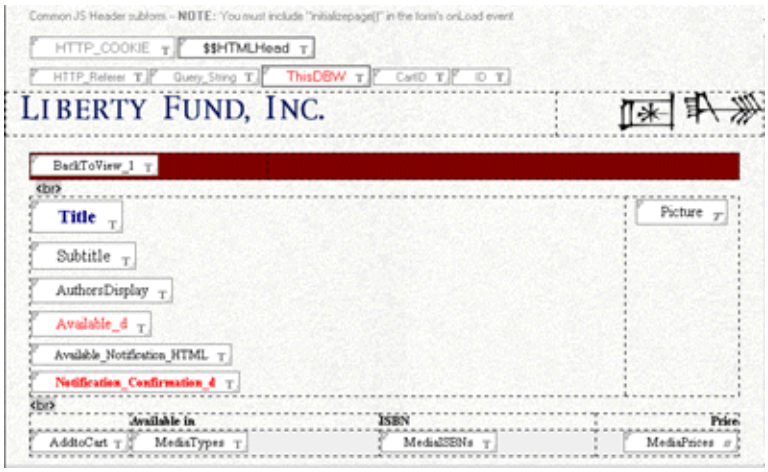


Notice that under the title and author information there is text informing customers that they can be notified when the current title becomes available. There's also an input field for their email address and a submit button. These elements only appear on those items that are marked in the catalog as not yet available. Notice too that there are no links to add this item to the cart; these are only supplied when the item is actually available for sale.

Now let's assume a customer has entered their email address and pushed the submit button. Although this action initiates quite a bit of activity behind-the-scenes, from the perspective of the customer, the item they are currently viewing simply redisplays with a confirmation message replacing the notification text, input field, and submit button:



Now that we've seen the end result, look again at the Web version of the Catalog Entry form in Designer:



The notification text, input field, and submit button are all generated by HTML contained in the Available_Notification_HTML field, which is computed-for-display text and has the following formula:

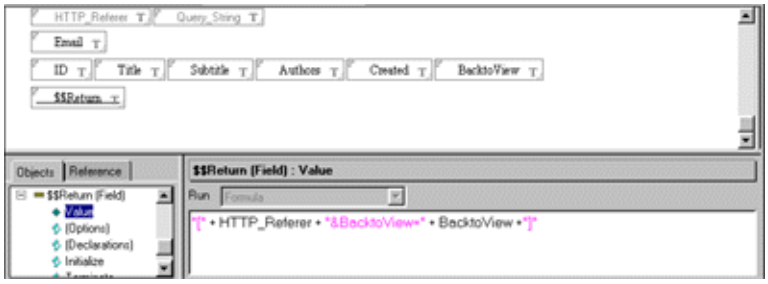
```
"[</form><form METHOD=post ACTION=\"/" + ThisDBW +
"/Notification+Query?CreateDocument\" NAME=\"_DominoForm\">
<I><FONT SIZE=2 FACE=\"Times New Roman\">To be notified when this title
becomes available, simply<br>provide us your Email address:
<br></FONT></I>
<INPUT NAME=\"Email\" VALUE=\"\">
<INPUT TYPE=submit VALUE=\"Submit\"><BR>
<INPUT NAME=\"ID\" VALUE=\"" + ID + "\" Type=\"Hidden\">
<INPUT NAME=\"Title\" VALUE=\"" + Title + "\" Type=\"Hidden\">
<INPUT NAME=\"Subtitle\" VALUE=\"" + Subtitle + "\" Type=\"Hidden\">
<INPUT NAME=\"Authors\" VALUE=\"" + @Implode(AuthorsDisplay; "!!") + "\"
Type=\"Hidden\">
<INPUT NAME=\"Created\" VALUE=\"" + @Text(@Now) + "\"
Type=\"Hidden\">
<INPUT NAME=\"BacktoView\" VALUE=\"" + HTTP_Referer + "\"
Type=\"Hidden\"></form>]"
```


If this code sets off alarm bells, that's because it's almost identical to how the Liberty Fund searches are constructed. First, a form is included in the middle of the page with its ACTION attribute set to create a new document—in this case using the Notification Query form. In other words, once the customer supplies their email address and presses the submit button, they are in fact creating a new document in the database that will capture the specifics of the item in which they are interested. Those specifics are supplied by the INPUT tags such as ID, Title, Subtitle, and so on, all of which are hidden, allowing the customer to remain oblivious to the passing of this information to the new document.

The last INPUT element, named "BacktoView," requires explanation. As a general navigation rule, we always want the ability to return a customer to the previous screen without resorting to the use of their browser's Back button. Normally, that's easy to accomplish; the CGI variable HTTP_Referer contains the URL from the previous page, which can simply be used as the basis for a "Back" link. If you'll refer to the screens of the catalog item above, notice the "Back one page" link to provide just that.

But as we'll see in a moment, when a customer submits a notification request, they are actually sent off to create a new document using the Notification Query form and then immediately redirected *back* to the Catalog Entry document from which they submitted the request. This poses a problem: when the Catalog Entry document reappears, HTTP_Referer is no longer pointing to the view or search results from which the customer entered the current catalog entry in the first place but instead returns the URL used to create the Notification Query. In short, by the time the process of creating an availability notification request is complete, we are unable to determine where the customer came from in order to view the catalog item! So, before sending them off to the Notification Query form and back to the Catalog Entry, we'll pass the URL from which they originally came (at this point, HTTP_Referer *does* still point there) to the Notification Query form.

Let's look at the sample database's Notification Query form and see what happens there. This form serves much the same purpose as the ViewSearchGeneric form (again, detailed in the **Part 1**); it captures the data being submitted as part of the availability notification request, and its $$Return field provides redirection. It's important to note that the customer never sees this happen. Here's the Notification Query form in Designer:



The form contains a matching editable text field for each value that was passed via the INPUT tags on the Catalog Entry's notification form submit. But whereas the ViewSearchGeneric's $$Return field redirected the browser to a Domino search URL, the Notification Query form's $$Return is designed to redirect the customer right back to the catalog item from which they submitted the notification request. It does so by way of the formula:

```
"[" + HTTP_Referer + "&BacktoView=" + BacktoView +"]"
```

The formula's return value is surrounded by square brackets, which tell Domino to redirect the browser to whatever is included between them. When the Notification Query is being created, HTTP_Referer contains the URL of

the Catalog Entry, and that's exactly where customers should return, making the creation of the Notification Query document seamless; *the customer never sees the document creation occur.* The final part of the redirect URL this formula builds is the appending of the BacktoView parameter, which will reflect the value passed from the Catalog Entry—and remember, this contains the URL of where the customer was *before* initially viewing the Catalog Entry. More on this in a moment.

The one remaining difference between the Notification Query form and the ViewSearchGeneric form is that while we wanted to avoid saving search documents in the database because they were simply temporary documents used to capture search criteria and redirect to the search itself, we most certainly *do* want to save documents created by the Notification Query form. Hence the absence of a SaveOptions field, which would normally be used to *prevent* a document from being saved.

Following the availability notification process to conclusion, customers will now find themselves back at the Catalog Entry they were just viewing, completely unaware that actually went anywhere else. Here is where the application needs to hide those elements created via the Available_Notification_HTML field (text, input field, and submit button) and display the confirmation message. In the Catalog Entry form, this message is represented by the Notification_Confirmation_d field. It's computed-for-display text and simply returns the string "Thank you for your interest in this title. You will be notified by Email when it becomes available."

But how are Available_Notification_HTML and Notification_Confirmation_d being hidden/unhidden? Say hello to our dear old friend, the BacktoView parameter on the URL. When a not-yet-available catalog item is first displayed, there is no "BacktoView" parameter on the URL since that is added as part of the availability notification process. Accordingly, the hide-when formula for Available_Notification_HTML field looks like:

```
Available = "1" | @Contains(Query_String; "BacktoView")
```

The first part of the OR condition above says that if the Available field on the underlying document for this catalog entry is 1, this item is currently for sale, in which case we don't want to display the availability notification information. But we also don't want to display it if the URL used to display the current item contains the BacktoView parameter—meaning a Notification Query was just created and the customer is now viewing the results of that action. This is determined through the Query_String field, which returns everything to the right of "?" in a URL. At this point, Available_Notification_HTML is hidden, and Notification_Confirmation_d is unhidden, thanks to its hide-when formula:

```
!@Contains(Query_String; "BacktoView")
```

Although negative logic is inherently evil, we'll allow it in the case of hide-when formulas. The preceding formula simply says that the confirmation message should be hidden at all times *except* when the URL contains the BacktoView parameter, at which time we know that the customer is returning from creating a Notification Query.

Just a few remaining points, I promise. There is still the matter of the "Back" links—the whole reason behind the BacktoView parameter. For convenience, there are two such links on the Catalog Entry form, one near the top and the other near the bottom of the form. Built by the BackToView and BackToView_1 fields, both are computed-for-display text and have the formula:

```
"[<a href=\"" + @If(@Contains(Query_String;
"BacktoView");@Right(Query_String; "BacktoView=");HTTP_Referer) + "\">
<FONT COLOR=\"#800000\">< Back one page.</FONT></a>"
```


Remember, if the customer is simply viewing a catalog item, HTTP_Referer will get them back to where they just came from, whether that is from a view or a set of search results. If, however, the current page's URL contains "BacktoView," we know that the customer has just created a Notification Query and, in order to send them back to the original view or search results, we've got to use the URL we stored in the BacktoView parameter at the start of the Notification Query process. Accordingly, that value is simply parsed out of the Query_String with @Right.

Of course, what you *do* with all the Notification Queries being created by your customers is another matter entirely. As you can see in the sample database, Liberty Fund limits these documents to a view called Notification Queries whose view selection formula limits the documents shown to:

```
SELECT Form = "Notification Query"
```

Then, a daily scheduled agent called Availability Notifications runs against the documents in that view, using the item ID of each to find that item in the catalog and determine if it is finally available. If so, a notification email message including a link to that item is constructed and sent to the customer. Then that Notification Query is deleted from the view. You can see the complete code for the agent that processes these queries in the **Liberty Fund Library 2** sample database or in the **Availability Notifications agent** sidebar.

With the technique we've just examined, customers can be supplied with a nice, simple interface for indicating their interest in upcoming catalog items. While there is quite a bit of action underway in the database to facilitate this, to the customer, it translates into the simple click of a button. And I'm sure it comes as no surprise that a relatively small feature equates to a rather elaborate implementation!

Where next?

So far in this series, we've seen how to address navigational challenges, track customers by unique IDs, display items and add them to a shopping cart, and create availability notifications. The next logical step is to dissect the shopping cart itself, and that's exactly what we'll do in the next month's article. In fact, we'll tackle the whole ordering process, which includes the finalization of cart contents, collecting customer information, and basic credit card processing. Along the way, we'll see a few neat little tricks that you may not be familiar with, so be sure to check back for the final installment!

ABOUT THE AUTHOR

Michael Patrick is a Senior Consultant with **Knowledge Resource Group** in Indianapolis, Indiana.

]

The AddtoCart agent

Here is the code for the AddtoCart agent:

```
Sub Initialize
	Dim s As New NotesSession
	Dim db As NotesDatabase
	Dim doc As NotesDocument, oiDoc As NotesDocument, cDoc As NotesDocument
	Set db = s.CurrentDatabase

	Set doc = s.DocumentContext

	Dim vCartID As Variant, vISBN As Variant, vItemID As Variant, vMedia As Variant, vQuantity As Variant,
	vPosn As Variant, vPath As Variant
	Dim vOrderKey(1) As String

	'Determine CartID
	vCartID = Evaluate ({ @Middle (@LowerCase(Query_String) + "&"; "&cartid="; "&") }, doc)
	If vCartID(0) = "" Then
		vCartID= Evaluate({@Middle(@LowerCase(HTTP_COOKIE) + ";";"cartid=";";")}, doc)
	End If
	'Get ISBN from URL
	vISBN =  Evaluate ({ @Middle (@LowerCase(Query_String) + "&"; "&isbn="; "&") }, doc)

	'Get the catalog document using the ISBN as the key
	Set cDoc = db.GetView ("ISBNLookup").GetDocumentByKey (vISBN(0))

	' Get the existing order item document, if it exists.
	vOrderKey(0)=vCartID(0)
	vOrderKey(1)=vISBN(0)
	Set oiDoc = db.GetView ("OrderISBNLookup").GetDocumentByKey (vOrderKey)

	' Create new order item document in the event one was not found
	If oiDoc Is Nothing Then
		Set oiDoc = db.CreateDocument
		oiDoc.Form = "OrderItem"
		oiDoc.CartID = vCartID
		oiDoc.Quantity = 1
		oiDoc.ISBN = vISBN
		oiDoc.Title = cDoc.Title
		oiDoc.Author = cDoc.Author

		'Get position of media type from catalog and retrieve
		cDoc.tempISBN = vISBN
		vPosn = Evaluate ({@Member (tempISBN; MediaISBNs)}, cDoc)  ' position of ISBN in catalog entry doc
		cDoc.tempPosn = vPosn
		vMedia = Evaluate ( {@If (tempPosn = 0; "Error"; @Subset (@Subset (MediaTypes; tempPosn); -1))},
		cDoc)
		oiDoc.Media = vMedia
	Else
		oiDoc.Quantity=oiDoc.Quantity(0) + 1
```

```
    End If

    'Even if this is an existing Order Item we can go ahead and update the information. Possibly prices have
    'changed.
    cDoc.tempISBN = vISBN
    vPosn = Evaluate ({@Member (tempISBN; @lowercase (MediaISBNs))}, cDoc)  ' position of ISBN in catalog
    entry doc
    cDoc.tempPosn = vPosn
    vPrice = Evaluate ( {@If (tempPosn = 0; "Error"; @Subset (@Subset (MediaPrices; tempPosn); -1))}, cDoc)
    oiDoc.Price = vPrice

    Call oiDoc.Save (True, True)

    'Set URL path for return
    vPath=Evaluate({@ReplaceSubstring (@Subset (@DbName; -1); "\\" : " "; "/" : "+")})
    Print "[" + vPath(0) + "/cart?ReadForm&CartID=" + vCartID(0) + "]"

End Sub
```

]

The Availability Notifications agent

Here is the code for the Availability Notifications agent:

```
Sub Initialize
	Dim session As New NotesSession
	Dim db As NotesDatabase
	Dim availabilityView, catalogView As NotesView
	Dim availabilityQuery, catalogEntry, notification, tmpDoc, profiledoc As NotesDocument
	Dim rtitem As NotesRichTextItem

	Set db = session.CurrentDatabase
	Set availabilityView = db.GetView("Notification Queries")

	Set profiledoc = db.GetProfileDocument("ApplicationSettings")
	If profiledoc Is Nothing Then
		Exit Sub
	End If

	Set catalogView = db.GetView("(CatalogByID)")

	Set availabilityQuery = availabilityView.GetFirstDocument

	While Not (availabilityQuery Is Nothing)

		' Search the catalog for the title the customer originally expressed interest in

		Set catalogEntry = catalogView.GetDocumentByKey(availabilityQuery.ID(0))

		' If we couldn't find the title, mark the notification query as 'failed' and move to the next one

		If catalogEntry Is Nothing Then

			availabilityQuery.Failed = "Failed"
			Call availabilityQuery.Save(True, False)
			Set availabilityQuery = availabilityView.GetNextDocument(availabilityQuery)

		Else

			' If the title was found in the catalog, only process notifications for those titles that are now available

			If catalogEntry.Available(0) = "1" Then
				Set notification = db.CreateDocument
				Set rtitem = notification.CreateRichTextItem( "Body" )
				Call rtitem.AppendText("The following title is now available from Liberty Fund:")
				Call rtitem.AddNewLine(2)
				Call rtitem.AppendText(availabilityQuery.Title(0))
				Call rtitem.AddNewLine(1)
				If (availabilityQuery.Subtitle(0) <> "") Then
					Call rtitem.AppendText(availabilityQuery.Subtitle(0))
					Call rtitem.AddNewLine(1)
```

```
                End If

                ' To pass a multi-value author field from the catalog entry to the availability query, we had to
                concatenate all the authors together with '!!'.  Now, we parse them apart and show them on
                separate lines of the notification email

                auths = Evaluate(|@Explode(Authors; "!!")|, availabilityQuery)
                Forall x In auths
                    Call rtitem.AppendText(x)
                    Call rtitem.AddNewLine(1)
                End Forall

                Call rtitem.AddNewLine(1)
                Call rtitem.AppendText("For more information, or to order this item, please use the following
                link:")
                Call rtitem.AddNewLine(2)
                Call rtitem.AppendText(profiledoc.WebServerURL(0) + "/" + profiledoc.LibraryDBW(0) +
                "/CatalogByID/" & availabilityQuery.ID(0))
                Call rtitem.AddNewLine(2)
                Call rtitem.AppendText("Thank you,")
                Call rtitem.AddNewLine(2)
                Call rtitem.AppendText("Liberty Fund Customer Service")
                notification.SendTo = availabilityQuery.Email(0)
                notification.Subject = "The Title You Requested Now Available From Liberty Fund"
                Call notification.Send(False)

                ' Grab the next document before we delete the one we're working on

                Set tmpDoc = availabilityView.GetNextDocument(availabilityQuery)

                ' Remove the availability query after we've processed it

                Call availabilityQuery.Remove(True)
                Set availabilityQuery = tmpDoc

            Else

                ' Item is not yet available, so grab the next one
                Set availabilityQuery = availabilityView.GetNextDocument(availabilityQuery)

            End If

        End If

    Wend

End Sub
```