Notes.net

Iris Today

by
Michael
Patrick

Level: Intermediate
Works with: Designer 5.0
Updated: 12/01/2000

One of my marketing professors from college used to give the following advice concerning sales: "It's all just power handshakes and empty platitudes unless you *close the sale*." Likewise, e-commerce solutions have to be more than just pretty pages; an intuitive shopping cart/checkout scheme is essential for both customer satisfaction and revenue generation—neither of which can live long without the other!

In this third, and final, article in a series that examines the details of the Liberty Fund Domino-powered e-commerce Web site, we tackle the shopping cart and the whole ordering process, including the finalization of cart contents, collecting customer information, and basic credit card processing. This discussion completes the exploration of the site, which began with examining site navigation in **Part 1** of the series and continued by delving into session tracking, add-to-cart shopping capabilities, and product availability notifications in **Part 2**.

As with the other articles in the series, this article references sample databases, which you can download from the **Liberty Fund Library 3** page of the Iris Sandbox. The first database is the library itself, Library3.nsf, which is the application's main database and includes all the design elements discussed in all three articles in the series. The second database, orders.nsf, is the database that tracks and maintains customer orders. (See the **Profile document** sidebar for information about the paths for the sample databases.)

Note: One of the sample databases, Library3.nsf, has its Anonymous ACL entry set to Editor. This enables the database's agent sufficient access to run correctly when you are previewing the database locally. Be aware, however, that under real-world circumstances, Anonymous in the Library database MUST be set to Author to avoid the unwanted editing of documents that users should not have access to (such as catalog entries). Also, for the sake of convenience, the Default entry in both the Library3.nsf and orders.nsf databases has been set to Manager. Keep in mind that the Default entry in both would normally be set to No Access, and all the database agents would be signed with an ID that has the proper authority to edit documents in both databases.

This article assumes a solid understanding of how you design Notes/Domino applications using Domino Designer R5. Like its predecessors, it has extensive demonstration of HTML and LotusScript, so a firm understanding of both topics will prove helpful.

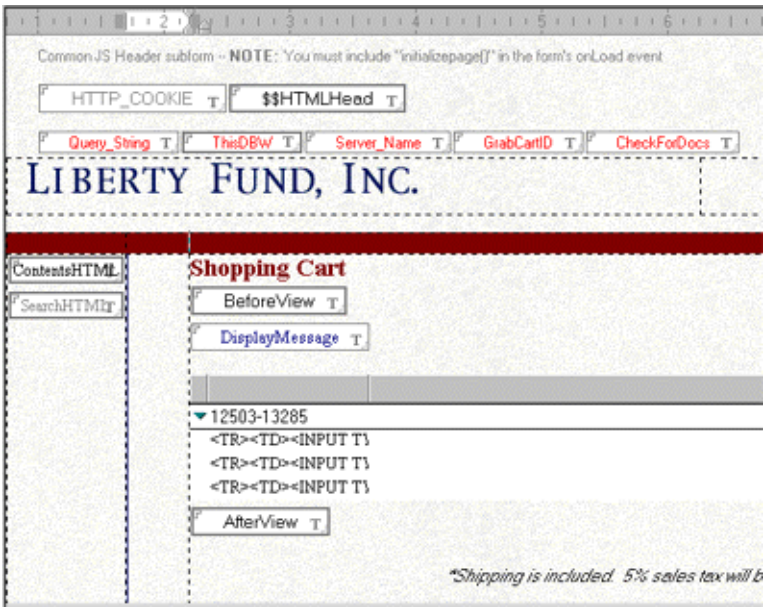
Understanding the shopping cart

Before we look at the shopping cart itself, here's a quick refresher on how the customer actually gets to the shopping cart. When viewing a catalog item, "Add to cart" links are available to call the database's AddtoCart agent (detailed in **Part 2** of this series). This agent takes the CartID of the customer and the ISBN number of the item, and creates an Order Item document in the database. The last action taken by the agent is a browser redirect to the database's Cart form.

Here's an active cart after a customer has clicked an "Add to cart" link:

In this example, the customer has added three items to the cart, which translates into three Order Item documents in the database matching the customer's CartID. In other words, the shopping cart is really an aggregation of all the Order Item documents created by a given customer. It may not be immediately obvious, but this is accomplished through the use of an embedded single-category view, (CartDetailByID), that incorporates both checkboxes *and* an editable field for the item quantity.

Let's look at the Cart form in detail:



The first element on the form is the Common JS Header subform, which we examined in **Part 2**. Immediately below that are five hidden fields. We've covered both Query_String and ThisDBW in detail in the first two articles of this series, but the last three, all of which are computed-for-display text fields, deserve attention.

The Server_Name field is a CGI variable that returns the current host name. Its value is simply Server_Name. Thus far, we've not yet bothered to derive the current host's name, opting instead to simply build relative URLs where

the host name is supplied automatically. We'll see why manually capturing this value now is important shortly.

The GrabCartID field is used for the single-category value of the embedded view. Here's its formula:

```
URL:=@Middle(@LowerCase(Query_String)+"&";"cartid=";"&");
Cookie := @Middle(@LowerCase(HTTP_COOKIE) + ";" ; "cartid="; ";");
@Trim (@If(Cookie="";URL;Cookie))
```

This formula attempts to retrieve the current CartID from a cookie (again, refer to **Part 2** in this series for details), and if that is not successful (because the customer has disabled cookies in their browser), the CartID is parsed out of the current page's URL via the Query_String field.

Finally, the CheckForDocs field ensures that there are actually items in the current cart to display:

```
Key := GrabCartID;
View := "CartDetailByID";
Docs := @DbLookup("Notes":"NoCache";@DbName;View;Key;1);
@If(Key="";"2";@IsError(Docs);"0";"1")
```

The value returned by the GrabCartID field is used for an @DbLookup against the CartDetailByID view. In the event that GrabCartID doesn't return a value (and this would only happen when a customer has *both* cookies and JavaScript disabled), the formula returns 2. If, on the other hand, the @DbLookup doesn't find any matching Order Item documents, the customer is viewing an empty cart, in which case the formula returns 0. Otherwise, 1 is returned. That's all well and good, but what do these values mean? Further down the form where the cart contents are displayed, there is a DisplayMessage field, which is computed-for-display text. Its job is to look at CheckForDocs and inform the customer of any potential problems, using the following formula:

```
NoItems := "[<BR>]You currently have no items in your shopping
cart.[<BR><BR>]To add items, go to a book description and click on the
\"Add to Cart\" link.";

NoCartID:="[<BR>]In order to use the shopping cart features of this site you
must either have \"cookies\" enabled in your browser or use a browser that
supports JavaScript. Without either one of these you may view the catalog,
but you will not be able to order online.";

@If(CheckForDocs="0";NoItems;CheckForDocs="2";NoCartID;"")
```

If no Order Item documents are found for the current customer (CheckForDocs = "0"), a message is displayed telling them how to place items in their cart.

Not being able to derive a CartID for the customer (CheckForDocs = "2") is a bad situation, since we need the CartID for the order process to function at all. If there is no derivable CartID, the customer receives a message informing them that they may browse the catalog, but until they have cookies or JavaScript enabled, they will not be able to order items. I can hear the groans already: "You mean you wait until they're in the cart before telling them they cannot order?" You can certainly make an argument that this check should be done elsewhere (and earlier) in the application. However, there's probably a greater risk of being struck by lightning than there is of having a customer coming to the site with a completely hobbled browser, so it's pretty safe to place here.

Finally, assuming that there *are* cart contents, DisplayMessage simply returns

null and is not visible when the cart is displayed.

Before we get to the meat of the cart itself, there's one more field to examine. Notice in the screen of the shopping cart viewed from the browser, there are links for Shopping Cart and Home at the upper right. These links are created by the field UserLinks, which is computed-for-display text. Here's the formula:

```
Block := "  <img src=\"/" + ThisDBW +
"/Block.gif?OpenImageResource\" border=0>  ";

ShoppingCart := Block + "<a href=\"/" + ThisDBW +
"/Cart?OpenForm\"><Font SIZE=\"2\" COLOR=\"#FFFFFF\" FACE=\"Times
New Roman\">Shopping Cart</FONT></a>";

Home := Block + "<a href=\"/" + ThisDBW + "?OpenDatabase\"><Font
SIZE=\"2\" COLOR=\"#FFFFFF\" FACE=\"Times New
Roman\">Home</FONT></a>";

"[<FONT SIZE=2>" + ShoppingCart + Home + "</FONT>]"
```

All the temporary values built by the formula rely on the field ThisDBW, which is another shared field on the form that computes to the current database's path, as explained earlier in this series.

There's one more point on the UserLinks field to consider. Since it generates a link to the shopping cart, which is detailed for the first time in this article, I deliberately left this field out of the sample databases that accompanied the first two articles in the series. This field is not unique to the Cart form, however. Because the Shopping Cart and Home links are applicable throughout the application, if you open Library3.nsf from the **Liberty Fund Library 3** page and view all the forms displayed to a customer, you'll notice that the UserLinks field is now included on each of them.

Now we come to the meat of the shopping cart. Immediately below the Shopping Cart label, is another computed-for-display text field called BeforeView. From the customer's perspective, this field provides the cart its column headings—Delete, Publication, Qty, and so on—which you can see in the screen of the active cart. Before explaining what else BeforeView accomplishes, let's look at its formula:

```
"[<FORM METHOD=POST ACTION=\"/" + THISDBW +
"/UpdateCart?OpenAgent\" NAME=\"_UpdateCart\">
<TABLE WIDTH=\"100%\"><TR><TD WIDTH=\"10%\" ><FONT
FACE=\"Times New Roman\" SIZE=2
COLOR=\"#800000\"><B>Delete</B></FONT></TD><TD WIDTH=\"50%\"
><FONT FACE=\"Times New Roman\" SIZE=2
COLOR=\"#800000\"><B>Publication</B></FONT></TD><TD
WIDTH=\"10%\"  ><FONT FACE=\"Times New Roman\" SIZE=2
COLOR=\"#800000\"><B>Qty</B></FONT></TD><TD WIDTH=\"15%\"
ALIGN=RIGHT ><FONT FACE=\"Times New Roman\" SIZE=2
COLOR=\"#800000\"><B>Unit Cost</B></FONT></TD><TD WIDTH=\"20%\"
ALIGN=RIGHT ><FONT FACE=\"Times New Roman\" SIZE=2
COLOR=\"#800000\"><B>Total Cost</B></FONT></TD></TR>]"
```

As is usually the case with HTML, there's a lot of code to do very little! All but the first line simply formats the column headings. Notice too, that the table created by the formula is not terminated. This table will be used by the embedded view and the AfterView field. BeforeView, then, simply starts the table that contains the entire cart.

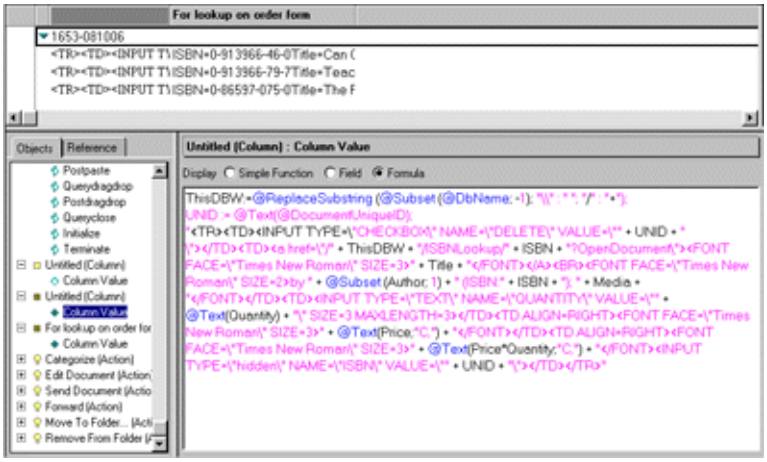
We've seen code analogous to the first line elsewhere in this series. It places an HTML form on the page, which has its own properties and behaviors. In this instance, the form's default behavior is to call the database's UpdateCart

agent. This agent makes use of some formatting included as part of the embedded (CartDetailByID) view, so once again, we'll hold off examining the code until all the pieces are in place.

As mentioned previously, right below the BeforeView and DisplayMessage fields is the single-category view, (CartDetailByID), from which the cart contents are displayed. (CartDetailByID) has a selection formula of:

```
SELECT Form="OrderItem" & Status = NULL
```

The view's "Show single category" object evaluates to the value returned by the GrabCartID field included at the top of the form. So, having derived the CartID for the current customer, only those Order Item documents that match the CartID are displayed. Let's look at the (CartDetailByID) view in Designer:



Since we'll be doing lookups by CartID against this view, the first column is sorted and categorized by CartID. That much is simple. The second column, however, is where things get interesting because it's responsible for each line of detail in the cart. Along with this, it's important to note that the view itself is set to "Treat view contents as HTML," meaning Domino will simply serve up the view contents while adding no HTML on its own; it's up to the programmer to ensure that all the HTML formatting is correct since Domino won't be supplying any for us. I'll explain why this is important after looking at the column's formula:

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

UNID := @Text(@DocumentUniqueID);

"<TR><TD><INPUT TYPE=\"CHECKBOX\" NAME=\"DELETE\" VALUE=\"" +
UNID + "\"></TD>
<TD><a href=\"/" + ThisDBW + "/ISBNLookup/" + ISBN +
"?OpenDocument\"><FONT FACE=\"Times New Roman\" SIZE=3>" + Title +
"</FONT></A><BR><FONT FACE=\"Times New Roman\" SIZE=2>by " +
@Subset (Author; 1) + " (ISBN:" + ISBN + "); " + Media + "</FONT></TD>
<TD><INPUT TYPE=\"TEXT\" NAME=\"QUANTITY\" VALUE=\"" +
@Text(Quantity) + "\" SIZE=3 MAXLENGTH=3></TD>
<TD ALIGN=RIGHT><FONT FACE=\"Times New Roman\" SIZE=3>" +
@Text(Price;"C,") + "</FONT></TD>
<TD ALIGN=RIGHT><FONT FACE=\"Times New Roman\" SIZE=3>" +
@Text(Price*Quantity;"C,") + "</FONT><INPUT TYPE=\"hidden\"
NAME=\"ISBN\" VALUE=\"" + UNID + "\"></TD></TR>"
```

With this code, we're including some features within each detail line that aren't normally part of views, most notably a checkbox that will allow for the

deletion of individual items and an editable field for the manipulation of quantities. A view automatically generated by Domino won't provide either, so the trick is to generate the HTML manually by way of the "Treat view contents as HTML" property. Also, notice that, as promised, this formula continues to add rows to the table begun by the BeforeView field.

The first line is the standard method for determining the path of the current database to be used in links later in the formula. The second line builds a temporary value called UNID and returns the 32-digit unique identifier for the current Order Item document in the database. This is actually a simpler way to look up Order Item documents than using a concatenation of CartID and ISBN number, and the UpdateCart agent will make use of this value for just that purpose. The third line uses the INPUT tag with a TYPE="CHECKBOX" to build the delete checkbox for this cart item. Also, notice that the checkbox is given the name of "DELETE" and its value is set to the temporary UNID value computed on the preceding line. This will allow the UpdateCart agent to go right to that Order Item document and remove it from the database.

Next is the HTML for everything that falls under the Publication column: the item's title in the form of a link to the title itself (through the use of the ISBNLookup view using the ISBN of the current item as the lookup key), the first author listed for the item, the ISBN, and the media of the title (Hardcover, Video, and so on). All this information is contained in the Order Item documents that are created via the (AddtoCart) agent.

This line is followed by another INPUT tag, this time of TYPE="TEXT," which supplies an input field for quantity changes. It is given the name "QUANTITY" and its value is simply the text representation of the Quantity field on the current Order Item. Next is the Unit Cost and Total Cost column values, @Text(Price;"C,") and @Text(Price*Quantity;"C,"), respectively. The "C," tells @Text to return a value formatted as currency and punctuated at the thousands. Finally, there is an INPUT tag that is hidden and named "ISBN." The name is a tad misleading because the value is actually set to the UNID computed earlier and represents the identifier for the Order Item document, not the ISBN of the item itself. The name itself is irrelevant aside from the fact that we *need* a name for this value in the UpdateCart agent.

There is a third column in the (CartDetailByID) view as well, but it is actually used later in the ordering process, so I'll hold off describing it until later.

So now that we've seen how the cart contents are displayed, all that remains is the AfterView field, which like BeforeView, is computed-for-display text:

```
List :=
@DbLookup("Notes":"NoCache";@DbName;"OrderISBNLookup";GrabCartID;
3);

Total := @If(@IsError(List);@Text(List);@Text(@Sum(List);"C,"));

orderDB := @GetProfileField ("ApplicationSettings"; "OrdersDBW");

"[<TR><TD></TD><TD></TD><TD VALIGN=BOTTOM></TD><TD
ALIGN=RIGHT><BR><FONT FACE=\"Times New Roman\" SIZE=3
COLOR=\"#800000\"><B>Subtotal:</B></FONT></TD><TD
ALIGN=RIGHT><BR><FONT FACE=\"Times New Roman\" SIZE=3><B>" +
Total + "</B></FONT>*</TD></TR></TABLE>

<INPUT TYPE=IMAGE SRC=\"/" + ThisDBW +
"/Recalculate.gif?OpenImageResource\" NAME=\"Recalculate\"
ALT=\"Recalculate\" border=0>  
<a href=\"https://" + Server_Name + "/" + orderDB +
"/Order?OpenForm\"><img src=\"/" + ThisDBW +
"/Proceed.gif?OpenImageResource\" Alt=\"Proceed to Checkout\"
```



```
border=0></A></FORM>]"
```

The first line builds the temporary value List, which is derived from an @DbLookup to the OrderISBNLookup view using the customer's CartID as the key. This lookup grabs the third column, which is defined as:

```
Price * Quantity
```

So, for each Order Item document matching the current CartID, the total price for that item is returned. These totals are then summed together by the next line and returned as Total.

The third line builds the temporary value, orderDB. This is the first time we've seen the database's Profile document in action. Due to a variety of factors, Liberty Fund decided to keep their catalog and ordering mechanism in separate databases. Since hardcoding the path and name of the Orders database is inadvisable, it is instead included on a Profile document along with various other settings. See the **Profile document** sidebar for more information about specifying the path and name of the Orders database.

In the formula's fourth line (where the HTML starts), we're closing off the table begun by the BeforeView field and placing a subtotal for the order in the last column using the temporary value of Total calculated on the second line. Next is an INPUT tag of TYPE=IMAGE, meaning instead of placing a button on the form, a graphic is substituted, but it retains the characteristics of a button. When clicked, the graphic actually submits the form. Remember, the action for the FORM in this instance is to trigger the UpdateCart agent.

Finally, the last link built by the formula is represented by the Proceed to Checkout button, which you can see in the screen of the active cart. This is where the customer jumps into the Orders database to finalize their shipping and payment information. The first thing to notice about the link is that the secured form of HyperText Transfer Protocol (HTTP) is being specified with " https://." This means that all subsequent exchanges between the server and the browser will be encrypted—as is true of virtually all credible on-line shopping sites.

Providing for a secured site has two primary implications for a Domino administrator: first, that a trusted certificate has been acquired from one of any number of companies that provide them (**Verisign** being one example); and second, that the certificate has been incorporated into the Domino server and SSL (Secure Sockets Layer) has been enabled. (See the **Domino 5 Administration Help** for more information on both of these topics.)

Note: If you install these sample databases on a workstation or a server that does not have SSL enabled, an error will result when you attempt to initiate a secure transaction. With this in mind, although the above code specifies "https://" in the link to the Orders database, the actual code I've included in Library3.nsf contains "http://" so you don't have to change it to get the sample databases to work. Be aware that if you adapt the sample code, you'll want to change this to read "https://" once your configuration can accommodate it.

Finishing our discussion of the last link, the name of the server is provided via the Server_Name field from the top of the Cart form, and the path of the Orders database (which we referenced from the application Profile document) is stored in the temporary orderDB. Once in the Orders database, the Order form is displayed for the customer via Order?OpenForm. (The Orders database is examined in detail in the last section of this article.)

Although we're at the end of the Cart form and ready to ring up the sale, I did gloss over something a little earlier. This wasn't due to its insignificance; in fact, it's so cool, it deserves its own section.

Examining the UpdateCart agent

Let's take another look at the active shopping cart:

Notice that not only does each line item in the cart have a checkbox to allow for deletion from the cart, but it also contains an editable quantity field. Before you say, "So what? That's no big deal!" remember that the shopping cart isn't a document with a form behind it containing these fields. This is an embedded view *containing editable fields*—and that *is* a big deal.

Actually, putting the checkboxes and editable fields in the view isn't all that hard. If you remember our discussion of the (CartDetailByID) view, once the property "Treat view contents as HTML" was selected, there was nothing more to it than including the proper HTML in one of the view columns in order to generate these elements. But while including checkboxes and input fields in the view is fairly trivial, *catching their values is not*—unless you know the secret.

Everything takes place in the UpdateCart agent. (You can examine the entire agent in the **UpdateCart agent** sidebar or the **sample database**.) The purpose of the agent is simple: update the quantity for each Order Item in the cart based on customer input, delete those Order Items that the customer has chosen to remove, and redisplay the updated cart.

Recall that the Order Items are separate documents in the database, so an active cart consists of one (and most likely multiple) documents matching the customer's CartID. The agent iterates over this group of documents until all have been updated or removed.

I'm only going to focus on five (nonconsecutive) lines of the agent, because this is where the customer input is captured and arranged into a useable format:

```
vContent = Evaluate({@Explode(Request_Content;"&")}, doc)
...
doc.tmpContent = vContent
vDelete=Evaluate({@Trim(@Right(tmpContent;"DELETE="))},doc)
vQuantity=Evaluate({@Trim(@Right(tmpContent;"QUANTITY="))},doc)
vISBN=Evaluate({@Trim(@Right(tmpContent;"ISBN="))},doc)
```

Starting with the first line, you'll notice the use of Request_Content. This is a CGI variable that Domino automatically returns (just like Query_String,

Server_Name, and so on) when referenced in an agent. Request_Content contains all the data posted by the HTML form when the Recalculate graphic is clicked. In other words, all the INPUT elements on the form were collected together and concatenated into Request_Content. As an example, this is the value returned by Request_Content for a cart containing three items (the values are separated by "&," which happens automatically):

```
DELETE=34CE2DAC8A85D84A0525697B00067F37&QUANTITY=1&ISBN=
34CE2DAC8A85D84A0525697B00067F37&
QUANTITY=2&ISBN=034C7AB0E481BE3F0525697B00068487&
QUANTITY=1&ISBN=AE5B2A91D0A9418F0525697C000E6B48
```

Note that while Request_Content is represented here across multiple lines, I've only done that for the purpose of clarity; Request_Content is a single string. In this instance, there are three items in the cart. Since the DELETE value is only included when an individual item's checkbox is selected, its presence with the values for the first item indicates that it has been marked for deletion. Notice too, that all three items have returned quantity and ISBN information.

With a little help from the Evaluate statement, the individual values in Request_Content will be exploded into a Variant, vContent. When this is complete, the latter's content will look like this:

```
vContent(0) = DELETE=34CE2DAC8A85D84A0525697B00067F37
vContent(1) = QUANTITY=1
vContent(2) = ISBN=34CE2DAC8A85D84A0525697B00067F37
vContent(3) = QUANTITY=2
vContent(4) = ISBN=034C7AB0E481BE3F0525697B00068487
vContent(5) = QUANTITY=1
vContent(6) = ISBN=AE5B2A91D0A9418F0525697C000E6B48
```

Now that the values are separated, they need to be grouped together by type. Once again, the Evaluate statement comes in handy, this time making use of the @Right command. Because doc.tmpContent (which is set equal to vContent on the second line) contains multiple values, @Right will find each matching value and operate on it. Taking QUANTITY as an example:

```
vQuantity=Evaluate({@Trim(@Right(tmpContent;"QUANTITY="))},doc)
```

will yield:

```
vQuantity(0) = 1
vQuantity(1) = 2
vQuantity(2) = 1
```

Not surprisingly, vISBN looks much the same, but it obviously contains document UNIDs, and vDelete has only one value since it only appears once in Request_Content.

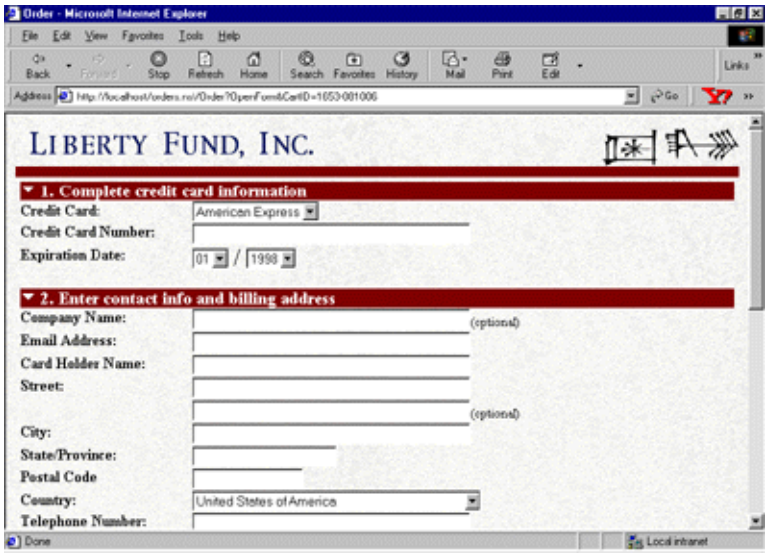
Once these Variants are established, the UpdateCart agent first updates the quantities on each Order Item document. It does so by looping through the values in vISBN, looking up each in the database (thus returning an Order Item document), and using the corresponding value in vQuantity to update its quantity information. We can do this because vISBN and vQuantity are in synch with each other; vISBN(0) contains the UNID of the document whose quantity is stored in vQuantity(0) and so on. Finally, the values in vDelete (if there are any) are used to remove Order Item documents from the database. It may seem to make more sense to do the deletes first, but the code is actually much simpler when the quantity updates are performed first and the deletes follow.

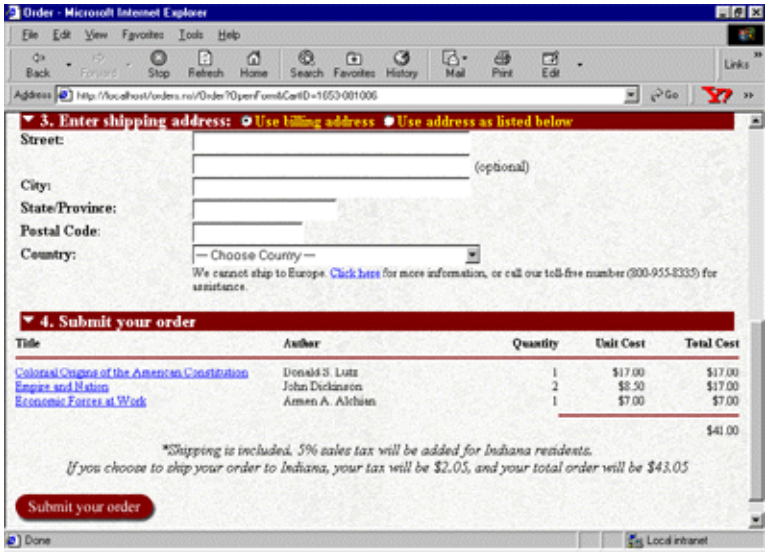
That's all there is to it! Using these techniques, not only is it simple to include

unlimited editable fields in a view, it's also easy to extract user input from those fields and process it accordingly. As is the case with just about everything we've examined in this series, this technique has far broader implications for Domino applications than just providing editable quantity fields in an e-commerce setting. Now editable fields within views are no longer a "wouldn't it be nice?" but rather a "how have I managed to live without this?"

Last but not least: order finalization

It's been a long haul getting to this point, but the end is finally in sight. We're ready to collect billing and shipping information from the customer, validate and charge their credit card (or deny the charge, if need be), and provide them with order confirmation. Remember that the Proceed to Checkout button on the Cart form passes the customer into Orders.nsf and the Order form. The following screens show the Order form as seen by the customer:





The upper portion of the form is where the customer supplies their information, and the bottom shows a summary of the order for verification.

Now let's look at the Order form in Designer. We'll examine two parts of the

form in particular: the hidden fields at the top of the form and the OrderDetail subform and additional hidden fields at the bottom. First, here's the top of the form:

The first thing you should notice is the presence of our old friend, the Common JS Header subform, which we examined in detail in **Part 2**. It's just as important to maintain the CartID in the Orders database as it was in the Library database. Given that a customer can pass back and forth between the databases, just because they are in the act of finalizing an order does not mean they are necessarily at the end of their shopping experience. Indeed, they still have the ability to jump back to individual catalog entries or return to the catalog altogether. Therefore, the CartID has lost none of its importance and must be handled here just as it was in the Library database.

Immediately below this subform are five computed-for-display text fields that are all hidden from both the Notes client and Web browsers. Both ThisDBW and Query_String have been used throughout the application and do not require further elaboration.

Server_Name is a field we've already examined on the Cart form from the Library database. There, it was used to construct a full URL that began with "https" in order to start a secure transaction with the server. It's included on the Order form for the exact opposite reason. When in the Orders database, all client/server transactions are secure, but if in the midst of an order the customer decides to return to the catalog (as the item links at the bottom of the order allow), subsequent transactions should *not* be secure for improved performance. As we'll see shortly, while most links in the Orders database default to "https," on occasions where the customer can return to the Library, we'll build links beginning with "http," thus breaking out of secure mode.

The StateCodes field is multi-value and contains abbreviations for all the states. I won't show all the values, but, in general, they are the standard two-letter abbreviations: AL, AK, AZ, AR, CA, CO, and so on. The corresponding StateNames field contains the full names of the states: Alabama, Alaska, Arizona, Arkansas, California, Colorado, and so on. These two fields simply allow us to replace a state name supplied by a customer with the state's abbreviation in the billing and shipping information sections of the Order document.

Next are the CartID and ID fields, both of which are text and editable. Although they're hidden, we want to store them on the Order document

created by this form, so instead of being hidden by the "Hide paragraph from Web browsers" setting, these fields are hidden via their respective HTML Attributes object, which is set to type=hidden. This way, both fields are known to the browser, and it can submit them to Domino with the remainder of the form's information even though a customer cannot see them. CartID's default formula is:

```
@Middle(Query_String+"&";"&CartID=";"&")
```

This formula has been used elsewhere as well. ID, on the other hand, has not been seen before now. This is the ID particular to this Order document, and differs from the CartID in that it is possible to place multiple orders while still maintaining the same CartID. ID's formula looks like this:

```
REM "Random number";
strTimeRandom := @Right (@Unique; "-");
nRandom := @Round(( 99999 - 10000 )*@Random + 1);
strTimeRandom + "-" + @Text (nRandom)
```

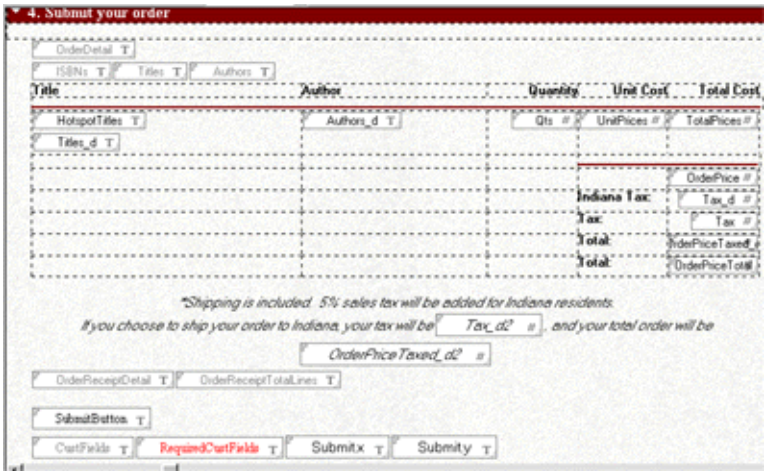
This produces a random number resembling 4QJK67-45170. We'll also see this value in use a little later.

Working our way down the form, the first section that's actually viewable begins at the Order Administrator Mode label. This section allows (with one important exception) Liberty Fund personnel to view and fulfill submitted Orders via a Notes client and is hidden from customers. The Status field is a combobox allowing for the manual update of the Order status, and Mdump is a field designed to help diagnose credit card processing errors and is populated by the (CreditCard) agent.

While both Status and Mdump are used by Liberty Fund employees, the ErrorMessage field *is* intended for customers. Once the customer has submitted an order and the (OrderWebQuerySave) agent finds that one of the required fields is in error, it redisplays the Order form, appending &ValError to the URL, which indicates that something didn't pass validation. ErrorMessage is computed-for-display and hidden as long as it remains NULL. If ValError appears in the URL, an error message is displayed; otherwise, the field is set to NULL and remains hidden. Here's the ErrorMessage field formula:

```
@If(@Contains(Query_String; "ValError"); "You are missing one or more
required fields.  All are required unless otherwise noted."; NULL)
```

Below this section are additional sections for the collection of credit card, billing, and shipping information. The fields contained in each are straightforward and don't require much explanation. But things get interesting again at the bottom of the form, for that is where the details for an order are listed for customer verification. Here's the end of the Order form in Designer:



The first field to examine is OrderDetail, which is editable text hidden via its HTML Attributes with type=hidden. This field contains all the information for the contents of the cart, and subsequent fields on the form refer to it. As such, we only want it to calculate once—when the order is first displayed. After an order is processed, the contents of the cart will be cleared, so capturing this information up front on the Order document is important, and for that, an editable field is used. Also, remember that the contents of a customer's cart are actually stored in the Library database, so in order for this detail information to appear on the Order itself, an @DbLookup must be performed back into the Library database and the (CartDetailByID) view using the customer's CartID:

```
strDB := @GetProfileField ("ApplicationSettings"; "LibraryDB");
x := @DbLookup ("Notes" : "NoCache"; @Subset (@DbName; 1) : strDB;
"(CartDetailByID)"; CartID; 3);
@If (@IsError (x); "lookup error"; x)
```

Note: Like the Library database, the Orders database also makes use of a Profile document to store the database paths. You can ensure that the path back to the Library database is correct for your installation of the sample databases by checking this Profile document. See the **Profile document** sidebar for more directions on accessing the Profile document.

Notice that the @DbLookup is grabbing the third column of the view. Although we examined this view in detail elsewhere, we didn't cover the third column since it isn't used until this point in the application. This column is marked as hidden and has the following formula:

```
"ISBN=" + ISBN +
"Title=" + Title +
"Author=" + @Subset (Author; 1) +
"Quantity=" + @Text (Quantity) +
"Price=" + @Text (Price)
```

This code simply concatenates all the information for a given Order Item into a single string. Additionally, realize that this @DbLookup returns the details for each Order Item corresponding to a given CartID.

Next there are three hidden multi-value computed fields—ISBNs, Titles, and Authors—whose job it is to parse values out of OrderDetail. The ISBNs, for instance, are derived from each value returned by the @DbLookup via:

```
@Middle (OrderDetail; "ISBN="; "Title=")
```

Suffice to say that every other detail pertaining to an Order Item is

sandwiched somewhere in the midst of the string for a given item, so they all use a formula similar to the ISBNs field. This holds true for Titles and Authors fields as well as the Qts and UnitPrices fields contained in the order detail table.

There are two fields under the Title column of the order detail table. The first, HotspotTitles, is designed to offer customers a link back to the individual items they're in the process of ordering:

```
strLibraryDB := @GetProfileField ("ApplicationSettings"; "LibraryDBW");
x := "<A HREF=\"http://" + Server_Name+ "/" + strLibraryDB + "/ISBNLookup/"
+ ISBNs + "?OpenDocument\">" + Titles + "</A>";
"[" + @Implode (x; "<br>") + "]"
```

Remember from our discussion of the Server_Name field that these links need to break the customer out of their secure session. It should also be said that these links won't be any good for Liberty Fund personnel accessing these orders with a Notes client, so this field is hidden from Notes. Conversely, the field immediately below it, Titles_d, is multi-value, computed-for-display and points to the Titles field for its value. This provides the item titles for the Notes client and the field is therefore marked as hidden from Web browsers.

The Author column of the table contains Authors_d, which is computed-for-display and multi-value and gets its values from the Authors field. Under Total Cost are a number of fields, the first of which is TotalPrices. It's a multi-value computed number field whose formula is simply, Qts*UnitPrices. These values are then summed by the OrderPrice field, which is numeric and computed as @Sum(TotalPrices).

This brings us to the tax fields, which are a little tricky. Unless an Order is displayed to a customer multiple times as the result of validation errors, they won't see these fields. Nevertheless, they are needed in order to charge a customer the correct amount. We'll look at each in succession.

The Tax_d field is numeric (currency) and computed-for-display. Like its corresponding Indiana Tax label, it is hidden when read or when the "Hide paragraph if formula is true" evaluates to:

```
!(Country = "United States of America" & State = "IN")
```

In other words, the tax information only appears when the customer specifies they are from the United States and, more specifically, from the state of Indiana. The tax is actually calculated by Tax_d's formula:

```
(@Round (OrderPrice * 0.05 * 100))/100
```

The Tax field is numeric (currency) and computed. It is hidden when the order is being edited, and its formula is:

```
strShipState := @If (ShipToSame = "1"; State; State_Ship);
@If (strShipState = "IN"; Tax_d; 0)
```

If the customer has chosen to ship their order to their billing address, then the formula takes the state information from the State field. If, on the other hand, they've specified a shipping address that differs from their billing information, State_Ship will be used. Regardless of which state field is being considered, if it specifies Indiana (IN), the tax is equal to what was calculated by Tax_d. Otherwise, the tax for the order is 0.

OrderPriceTaxed_d is numeric (currency) and computed-for-display. It's hide-when properties are identical to that of Tax_d, that is, hidden when reading or when the state information indicates Indiana. It simply sums the

following:

```
OrderPrice + Tax_d
```

The details of OrderPriceTotal matches those of Tax; it's numeric (currency), computed, and hidden when edited. It sums the following:

```
OrderPrice + Tax
```

Note also that this field contains the value that will be used during the credit card processing for the order total.

As mentioned previously, a customer who successfully fills out an order on the first try will not see any of the aforementioned tax fields. Therefore, it's important to include a summary of the tax implications on the order. To this end, notice the two lines of italicized text under the order detail table. All of it is hidden when the order is read (obviously, Liberty Fund personnel don't need to see this) and it contains two fields, Tax_d2 and OrderPriceTaxed_d2. Both are numeric (currency) and computed-for-display. Tax_d2 simply refers to Tax_d for its value, and OrderPriceTaxed_d2 refers to OrderPriceTaxed_d.

This ends the information seen by the customer on an order, but there are a few more hidden fields at the bottom of the form that still must be considered. The first two, OrderReceiptDetail and OrderReceiptTotalLines, are computed text multi-value fields that are hidden. These fields help build an order confirmation e-mail to the customer once the order has been successfully submitted and charged. We'll see the product of both fields when we examine the order confirmation e-mail, but for now let's look at how they build their values.

OrderReceiptDetail creates a single detail line for each Order Item with the formula:

```
Titles + " -- by " + @Subset (Authors; 1) + " -- " + @Text (Qts) + " copy @ " +
@Text (UnitPrices; "C,") + " = " + @Text (TotalPrices; "C,")
```

OrderReceiptTotalLines creates total information for the entire order with:

```
strTax := "Tax: " + @Text (Tax; "C,");
strTotal := "Order Total: " + @Text (OrderPriceTotal; "C,");
"----------------------------" : strTax : "----------------------------" : strTotal
```

Immediately below these two fields is the field that creates the form's Submit button. It's name is, appropriately, SubmitButton, which is computed-for-display text. Since Liberty Fund personnel don't need to see this, it's hidden from the Notes client. If you refer back to the screen of the Order form in the browser, you'll notice that once again a graphic is used in place of an actual button. This is accomplished through the following formula producing passthru HTML:

```
"[<INPUT TYPE=IMAGE SRC=\"/" + ThisDBW +
"/Submit.gif?OpenImageResource\" NAME=\"Submit\" ALT=\"Submit your
order\">]"
```

Notice too, that below SubmitButton are, among others, two fields called Submit.X and Submit.Y. These are so important that submitting an Order without them will result in an error. Both of these fields are editable text and always hidden. What are they for? When a graphic is used to submit a form (as is the case here) the browser automatically sends the server the clicked graphic's *x* and *y* coordinates. Is this information used by the application? No, but when fields are reported to Domino and there are no corresponding fields on the form, the document creation will fail. In short, these two fields are included solely for the purpose of keeping the server happy.

The only other fields left to consider are CustFields and RequiredCustFields, both of which are multi-value computed text that is hidden from Notes and browsers. The contents of RequiredCustFields controls the field validation performed by the (OrderWebQuerySave) agent. Before that can happen, *all* the form fields are first defined in the formula for CustFields:

```
 "CCType" : "CCAcctNum" : "CCExpMn" : "CCExpYr" : "CCTypeVerbose" :
"CompanyName" : "Name" : "Street1" : "Street2" : "City" : "State" : "Zip" :
"Country" : "PhoneNumber" : "Email" : "Street1_Ship" : "Street2_Ship" :
"City_Ship" : "State_Ship" : "Zip_Ship" : "Country_Ship"
```

This is not to say that all of the above fields will be required at all times. In fact, that's exactly the function of RequiredCustFields—to weed out those fields that won't be required:

```
vOptionalFields := "CompanyName" : "Street2" : "Street2_Ship";
vOptionalShippingFields :=
@If (
      ShipToSame = "1";
            "Street1_Ship" : "City_Ship" : "State_Ship" : "Zip_Ship" :
            "Country_Ship";
      NULL
);
vOptionalAll := @Trim (vOptionalFields : vOptionalShippingFields);
@Trim (@Replace (CustFields; vOptionalAll; NULL))
```

This builds a list of optional fields. In the event that the customer has chosen to ship to their billing address (ShipToSame = "1"), we don't need to worry about the fields in the Shipping Information section of the Order form, so these fields and the always-optional fields (CompanyName, Street2, and Street2_Ship) are removed from the list contained in CustFields and returned as the finalized list of required fields.

The (OrderWebQuerySave) agent

At last the Order form is ready to be submitted. When the customer clicks on the "Submit your order" graphic, an Order document is created in the Orders database. This is, of course, the job of the (OrderWebQuerySave) agent. The agent also performs the necessary field validation, and provided the order has passed validation, calls the (CreditCard) agent to actually charge the order. See the **(OrderWebQuerySave) agent** sidebar or the **sample database (orders.nsf)** for this agent's complete code.

The most important thing to realize with regard to the (OrderWebQuerySave) agent is that regardless of whether an order passes field validation or not, an Order document will be saved in the database. This is what enables the customer's previous information to be redisplayed to them. If the order fails validation, it will need to be redisplayed, which is accomplished in the agent by the line:

```
 Print "[/" + vPath(0) + "/0/" + doc.UniversalID + "?EditDocument&CartID=" +
strCartID + "&ValError]"
```

In other words, the Order document is simply reopened in Edit mode. If, on the other hand, the Order is in good shape, it is saved and the (CreditCard) agent is called, again with LotusScript's Print statement:

```
Print "[/" + vPath(0) + "/CreditCard?OpenAgent&CartID=" + strCartID + "&r="
+ strOrderID + "]"
```

This begs the question of whether the credit card processing could be done straight from the (OrderWebQuerySave) agent? Yes, it could, but there is a rather extensive set of LSXs (LotusScript Extensions) that need to be loaded

as part of the credit card processing, and if they were included as part of (OrderWebQuerySave), they would load each and every time the agent was called, whether the Order actually passed validation or not. Therefore, it makes more sense to separate the credit card processing into its own agent and call it once the Order passes validation.

The (CreditCard) agent
A word about the (CreditCard) agent: Liberty Fund's credit card processing relies on a product called Commerce Accelerator, available from **IT Factory**. This is the set of LSXs mentioned previously. It interacts with CyberCash (or any similar on-line financial transaction service company), allowing credit card validation and charging to be performed via calls from LotusScript. Note that there are other products available to perform these same functions. The credit card processing in the sample database *is not* functional for the simple reason that you would be required to have a valid copy of Commerce Accelerator installed on your machine in order to successfully complete a transaction. Nevertheless, I've included the (CreditCard) agent in an effort to demonstrate how easy it is to incorporate credit card handling into an application.

In the agent itself, notice that in addition to the regular comments, there is also quite a bit of code commented out. Every line of code that is marked as a comment applies to the initialization, execution, and resolution of tasks performed by Commerce Accelerator. This being the case, please be aware that this is a nonfunctional example supplied solely for your reference. Having said that, check out the **(CreditCard) agent** sidebar or the **sample database (orders.nsf)** to review the agent's code.

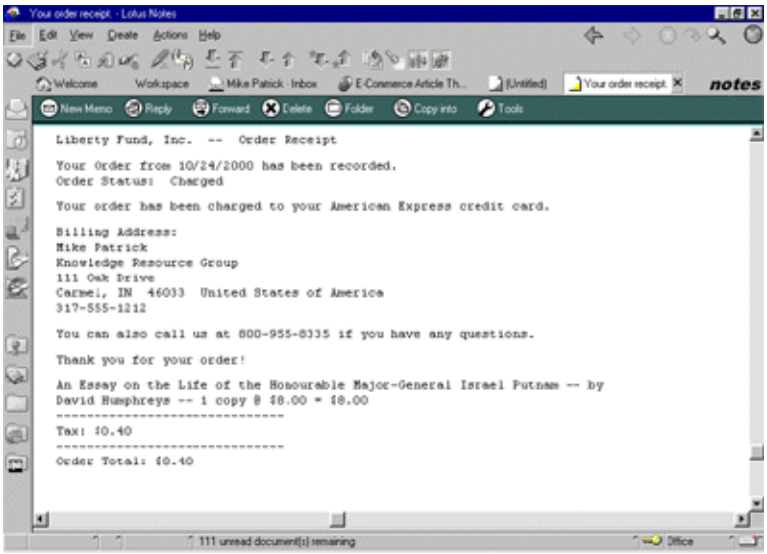
The actual portion of the agent that performs the credit card processing is fairly uneventful—it's little more than some initialization, a call to a method, and reading the response resulting from the transaction. The agent, however, provides a couple of additional subroutines of interest. Since it comes at the end of the order process, (CreditCard) also mails the customer an e-mail confirmation of their order and marks the Order Item documents in the Library database with a status that effectively cleans out the customer's cart.

Let's look first at the e-mail confirmation. Notice that the Orders database contains a form called OrderReceipt. Here is the form in Designer:

Pay no attention to the various fonts and sizes of the labels and fields on the

form; these will all render to plain text when the e-mail is sent. What is of interest is the fact that all the fields on the form have a match in the Order documents in the database. This allows the SendReceipt subroutine to pull a neat little trick. Instead of creating a new document—the receipt—and moving the fields from the Order document to the receipt, it's much simpler to "overlay" the OrderReceipt form on top of the Order document. In other words, the Order document resides in the database with its Form field set to Order. The agent simply replaces the Form field's value with OrderReceipt. Then, using the RenderToRTItem method, the Order document is essentially copied into the receipt document, visually represented as the OrderReceipt. Presto! Our Order document now looks like an OrderReceipt! Of course, once the receipt has been created, the Order document's Form field is reset to Order and saved once again. This technique is much simpler than moving various fields back and forth between documents.

After the OrderReceipt is formatted, it is of course mailed. Here's a screen of a receipt as it appears in the customer's mail program:



The second subroutine, ClearCart, does exactly that; it clears the customer's cart. This is just a matter of taking all the Order Item documents that match the current CartID, changing their Status to Ordered, creating an item called OrderID, and setting it to the Order document's ID value. Once the status of the Order Item documents has changed, they disappear from the (CartDetailByID) view, upon which the customer's cart is based. Keep in mind that it's up to you what is done with the "Ordered" Order Items; they're not needed for subsequent processing of an order, so they can either be purged via a scheduled agent or kept around for historical purposes. Just be aware that we have not examined a mechanism for their removal/archiving, so they will remain in the Library database unless otherwise dealt with.

This brings us to the last form in the Liberty Fund application, OrderThankYou. This form is displayed to indicate both success and failures of the credit card processing. The (CreditCard) agent assigns a value, strParm, with either a Yes or No, depending on whether or not the credit card transaction was successful. This value is then passed to the OrderThankYou form by (CreditCard) through the following code:

```
 Print "[" + vpath(0) +  "/OrderThankYou?ReadForm&CartID=" + vCartID(0) +
"&OrderID=" + vOrderID(0) + "&Success=" + strParm + "]"
```

We've seen the Print statement used throughout this series, so it doesn't require further explanation. Note, however, the "&Success=" parameter on

the end of the URL being constructed. This is what will tell OrderThankYou which message to display to the customer. Let's first look at the results of a successful submission:

Now, let's look at OrderThankYou in Designer:

Given the number of forms we've already looked at in this series, this one doesn't differ in any significant way from the others. Notice the presence of Server_Name again; this form, like the Order form, offers the customer links through which they can return to the Library database. Remember that everything in the Orders database is secure and that it's advisable to break out of a secure session when returning the customer to browsing the catalog. Therefore, the links on this page aren't relative as they are throughout the remainder of the application, and Server_Name helps us construct full URLs beginning with "http://."

The Success field controls which message is displayed to the customer. It's computed-for-display and numeric with the following formula:

```
@Contains (Query_String; "Success=Yes")
```


The success and failure messages then base their hide-when formulas on the value of Success. So, in order to display the message indicating success, that message's hide-when is set to hide in the event that !Success is true. Conversely, the failure message is hidden when Success is true. That is the extent of the OrderThankYou form's functionality! The customer is now at the end of the order process and is free to either camp out in front of their mailbox in anticipation of their shipment or to return to the catalog and order everything their willpower kept them from ordering the first time around!

Where to from here?

This is the end of our examination of a fully-functional Domino-powered e-commerce Web site and its catalog and shopping cart. While you can immediately build your own e-commerce site using the techniques we've discussed, they also represent just the tip of the e-commerce iceberg. To extend this thought, consider that during the course of this series you've likely compared the Liberty Fund site with that of your own on-line shopping experiences and have identified certain features whose inclusion would enhance customer satisfaction. For instance, it might be beneficial to store credit card and shipping information so that returning customers are not faced with the rather distasteful chore of re-entering it with every order. Order histories are another useful tool; customers may want a summary of past orders as well as the ability to check the status of their current orders.

These extra features imply, at minimum, the use of a basic authentication mechanism to ensure application security—a point that cannot be stressed highly enough, for storing customer information requires that the utmost attention be paid to security. The good news is that the extra measures required to provide such additional features do not fundamentally change the myriad of techniques examined in this series; in fact, they build upon these techniques. That said, consider this the foundation from which to launch your own e-commerce explorations!

ABOUT THE AUTHOR

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

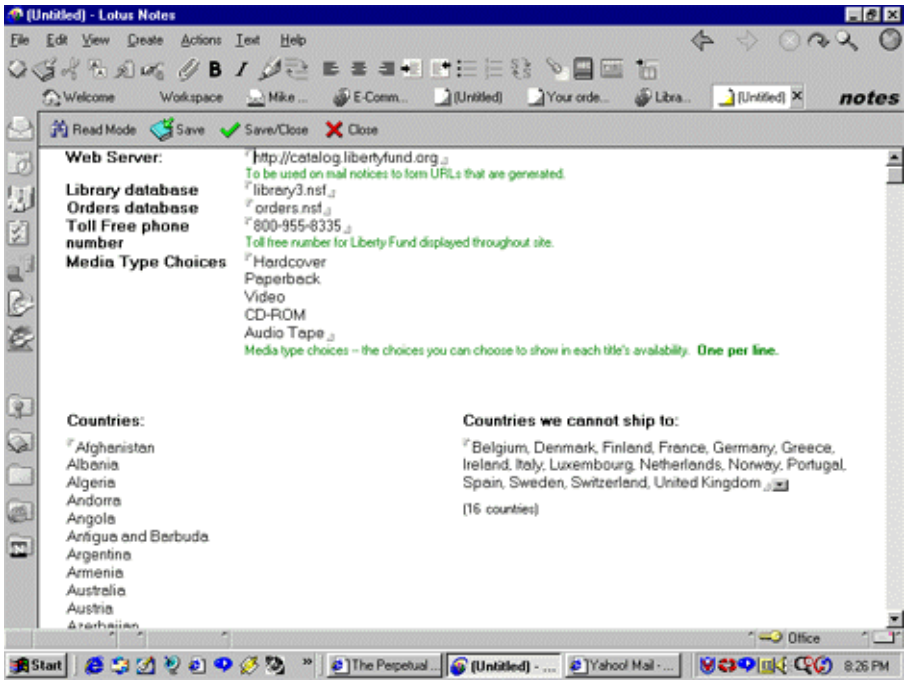
]

The Profile document

The Profile document for each database (Library3.nsf and orders.nsf) contains information needed for various processes and agents in the application. It includes the path and name of both databases. Note that these databases can be in any directory under your data directory (for example, Notes\data or Domino\data).

To see this Profile document for either database:

1. Open up the database in Notes.
2. Choose Actions - Admin - Edit Application Settings.



In the Profile document that appears, take a look at the fields labeled Library database and Orders database. These fields must include the path for the databases relative to your Notes data directory. Let's assume you've downloaded both sample databases that accompany this article and placed them in a temp subdirectory of your Notes data directory. You would reference the orders.nsf database as temp\orders.nsf.

]

The UpdateCart agent

Here is the code for the UpdateCart agent:

```
Sub Initialize
	Dim s As New NotesSession, db As NotesDatabase
	Dim doc As NotesDocument, oiDoc As NotesDocument

	Set db = s.CurrentDatabase
	Set doc = s.DocumentContext

	Dim vCartID As Variant, vPath As Variant, vContent As Variant
	Dim vDelete As Variant, vQuantity As Variant, vISBN As Variant, vSessionID As Variant
	Dim emptyQuantity As Integer

	'Establish the CartID

	vCartID= Evaluate({@Middle(@LowerCase(HTTP_COOKIE) + ";";"cartid=";";")}, doc)
	If vCartID(0) = "" Then
		vCartID = Evaluate ({@Middle (@LowerCase(HTTP_Referer) + "&"; "&cartid="; "&")}, doc)
	End If

	'Explode Request_Content and then the individual fields within it

	vContent = Evaluate({@Explode(Request_Content;"&")}, doc)
	'If the customer has accidentally blanked out any quantities, vQunatity and vISBN will get out of
	'sync when trimmed.  Thus, for each blank QUANTITY value in vContent, we'll simply blank out
	'its matching ISBN, which immediately follows the QUANTITY.  This will keep us from updating the
	'quantity on this particular Order Item doc, but the quantity is already stored there and will simply
	'redisplay when the cart is refreshed.

	Forall arg In vContent
		If emptyQuantity Then
			arg = "ISBN="
			emptyQuantity = False
		End If
		If arg = "QUANTITY=" Then
			emptyQuantity = True
		End If
	End Forall

	doc.tmpContent = vContent
	vDelete=Evaluate({@Trim(@Right(tmpContent;"DELETE="))},doc)
	vQuantity=Evaluate({@Trim(@Right(tmpContent;"QUANTITY="))},doc)
	vISBN=Evaluate({@Trim(@Right(tmpContent;"ISBN="))},doc)

	'Update quantities for all Order Item documents for this customer

	For x=0 To Ubound(vISBN)
		Set oidoc = db.GetDocumentByUNID(vISBN(x))
		If Not oidoc Is Nothing Then
```

```
                If Cint(vQuantity(x))>0 Then
                    oidoc.Quantity = Cint(vQuantity(x))
                    Call oidoc.Save(True,True)
                End If
            End If
        Next

        'Delete an Order Item documents this customer has marked for delete

        If Not vDelete(0) ="" Then
            Forall v In vDelete
                Set oidoc = db.GetDocumentByUNID(v)
                If Not oidoc Is Nothing Then
                    Call oidoc.Remove(True)
                End If
            End Forall
        End If

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

End Sub
```

]

The (OrderWebQuerySave) agent

Here is the code for the (OrderWebQuerySave) agent:

```
Sub Initialize

    Dim s As New NotesSession, db As NotesDatabase
    Dim doc As NotesDocument
    Dim strOrderID As String, strCartID As String
    Dim strDBPath As String, strErrorMessage As String, vCartID As Variant

    Set db = s.CurrentDatabase
    Set doc=s.DocumentContext

    Dim vPath As Variant

    'Set URL path for this database

    vPath=Evaluate({@ReplaceSubstring (@Subset (@DbName; -1); "\\" : " "; "/" : "+")})

    'Get the CartID and OrderID as calculated on the Order document

    strCartID = doc.CartID(0)
    strOrderID = doc.ID(0)

    'Perform field validations.  Any errors will result in this order being saved as a
    'draft and then it will be redisplayed to user with error message.

    Dim vFieldVal As Variant
    Forall ReqField In doc.RequiredCustFields
        vFieldval = doc.GetItemValue (ReqField)
        If vFieldval(0) = "" Then
            doc.Status = "Draft"
            Call doc.Save (False, True)
            Goto ReturnErrorMessage
        End If
    End Forall

    'If we make it this far there are no field validation problems.  Save the order
    'and call the CreditCard agent

    doc.Status = "ReadyToCharge"
    Call doc.Save (False, True)

    'We use a separate agent for the credit card processing so the LSX doesn't
    'have to load each time the order form saves.  Only when validation has
    'passed do we actually want to perform the credit card procesing.

    Print "[/" + vPath(0) + "/CreditCard?OpenAgent&CartID=" + strCartID + "&r=" + strOrderID + "]"

    Exit Sub
```

```
ReturnErrorMessage:
    'Since there were validatoin errors, redisplay the order
    Print "[/" + vPath(0) + "/0/" + doc.UniversalID + "?EditDocument&CartID=" + strCartID + "&ValError]"
End Sub
```

]

The (CreditCard) agent

Here is the code for the (CreditCard) agent:

```
Option Declare

'This 'Uselsx' statement would be uncommented, provided you were using Commerce Accelerator

'Uselsx "*Commerce"

Public Const  CC_ORDER_ID = "order-id"
Public Const  CC_NOTE = "note"
Public Const  CC_AMOUNT = "amount"
Public Const  CC_CARD_NAME = "card-name"
Public Const  CC_CARD_STATE =  "card-state"
Public Const  CC_CARD_COUNTRY = "card-country"
Public Const  CC_CARD_NBR = "card-number"
Public Const  CC_CARD_CITY = "card-city"
Public Const  CC_CARD_ADDR = "card-address"
Public Const  CC_CARD_EXP = "card-exp"
Public Const  CC_CARD_ZIP = "card-zip"
Public Const  CC_TXN_TYPE = "txn-type"
Public Const  CC_MSG_TYPE  = "message-type"
Public Const  CC_MERCHANT_MSG = "merchant-msg"
Public Const  CC_MERCH_TXN = "merch-txn"

Sub Initialize

    Dim s As New NotesSession, db As NotesDatabase
    Dim doc As NotesDocument, odoc As NotesDocument

    Set doc = s.DocumentContext
    Set db = doc.ParentDatabase
    Dim vCartID As Variant, vSessionID As Variant, vOrderID As Variant

    Dim vPath As Variant

    'Set URL path for return

    vPath=Evaluate({@ReplaceSubstring (@Subset (@DbName; -1); "\\" : " "; "/" : "+")})

    'Get the cart and order IDs for this doc

    vCartID = Evaluate ( {@Middle (Query_String + "&"; "&CartID="; "&") }, doc)
    vOrderID = Evaluate ( {@Middle (Query_String + "&"; "&r="; "&") }, doc)

    'Get the Order document based on OrderID from Query_String

    Set odoc = db.GetView ("(OrdersByID)").GetDocumentByKey (vOrderID(0), True)

    Dim vAmount As Variant, strParm As String
```

```
    'The commented code throughout this agent are those pieces specific to Commerce Accelerator
    'that will cause errors when the LSX is not loaded on the local machine or server.  Under regular
    'operating conditions, this code would of course be uncommented

    'Dim args As New CaNameValueList

    odoc.AgentCybercashRan = Now

    Dim num As Integer

    'args.addPair CC_ORDER_ID , "KRG-LF-"+ odoc.ID(0)
    'args.addPair CC_AMOUNT, Cstr (odoc.OrderPriceTotal(0))
    'args.addPair CC_CARD_NAME, odoc.Name(0)
    'args.addPair CC_CARD_ADDR, odoc.Street1(0)
    'args.addPair CC_CARD_CITY, odoc.City(0)
    'args.addPair CC_CARD_STATE, odoc.State(0)
    'args.addPair CC_CARD_ZIP, odoc.Zip(0)
    'args.addPair CC_CARD_COUNTRY, odoc.Country(0)
    'args.addPair CC_CARD_NBR, odoc.ccAcctNum(0)
    'args.addPair CC_CARD_EXP, odoc.CCExpMn(0) + "/" + odoc.CCExpYr(0)

    'Initialize a direct connection to CyberCash using your Cybercash ID and Merchant Key
    'You can also supply a Proxy server and Timeout here

    'Dim cc As CACybercashConnection
    'Dim result As New CaNameValueList

    If odoc.CCAcctNum(0) = "4111111111111111" Then
        ' test transaction
        'Set cc = New CaCybercashConnection("test-vvv", "asdfaeer2r4faef")  ' Test One
        'Call cc.Send("mauthcapture", args, result)
    End If

    'odoc.MStatus =  result.getval ("MStatus")
    'odoc.Mdump = result.dump

    'If the CyberCash response with "succss", finalize the Order status, email the
    'customer a receipt, and clear the Order Items out of their cart.  Otherwise, we'll
    'simply mark the Order status as failed.  strParm will be used by the OrderThankYou
    'form to display the appropriate message to the customer.  Unconditionally save
    'the Order document to reflect the Order's new status.

    If odoc.Mstatus(0) = "success" Then
        odoc.Status = "Charged"
        Call SendReceipt (s, db, odoc)
        Call ClearCart (s, db, odoc)
        strParm = "Yes"
    Else
        odoc.Status = "ChargeFailed"
        strParm = "No"
    End If

    Call odoc.save (False, True)

    'Display the OrderThankYou form

    Print "[" + vpath(0) +  "/OrderThankYou?ReadForm&CartID=" + vCartID(0) + "&OrderID=" + vOrderID(0) +
    "&Success=" + strParm + "]"

End Sub

Sub SendReceipt (s As NotesSession, db As NotesDatabase, odoc As NotesDocument)
```

```
	Dim v As NotesView

	Dim cdoc As NotesDocument, mdoc As NotesDocument

	'Build the receipt document general information

	Set mDoc = db.CreateDocument
	mDoc.Form = "Memo"
	mDoc.Principal = "Catalog"
	mDoc.SendTo =odoc.Email
	mDoc.Subject = "Your order receipt."

	'This next bit of code shows a trick.  First, a rich text field is created
	'on the receipt document.  Then, we'll actually switch the form associated
	'with the Order document, replacing the Order form with the OrderReceipt
	'form and saving it.  Next, the Order (now viewed via the OrderReceipt form) is
	'rendered to richtext onto the receipt doc, which then has all the
	'Order information, but it's displayed through the OrderReceipt form.   Before
	'sending the customer the receipt, the Order doc's form is changed back to
	'Order.

	Dim rt As New NotesRichTextItem (mdoc, "Body")
	odoc.Form = "OrderReceipt"
	Call odoc.Save (False, True)
	Call odoc.RenderToRTItem ( rt )
	odoc.Form = "Order"
	Call odoc.RemoveItem ("OrdersDBW")
	Call odoc.Save (False, True)
	Call mDoc.Send (False)
End Sub

Sub ClearCart (s As NotesSession, db As NotesDatabase, odoc As NotesDocument)
	Dim CartColl As NotesDocumentCollection
	Dim LibraryDB As NotesDatabase
	Dim strCartID As String
	Dim strDBPath As String

	'Find the Library database

	strDBPath=db.GetProfileDocument("ApplicationSettings").LibraryDB(0)
	Set LibraryDB = New NotesDatabase(db.Server,strDBPath)

	'Reach back into the Library database and get all the Order Item
	'documents for this cart.  Mark their status as 'Ordered' and also
	'give each the OrderID

	Set CartColl = LibraryDB.GetView ("(CartDetailByID)").GetAllDocumentsByKey (odoc.CartID(0))
	Call CartColl.StampAll ("Status", "Ordered")
	Call CartColl.StampAll ("OrderID", odoc.ID(0))

End Sub
```