show toc

Three Approaches for Three Client Types

Duwamish Books, Phase 4

Paul Johns
Microsoft Developer Network

August 1999

Summary: Discusses the new Web-based client strategy employed in Phase 4 of the Duwamish Books sample application. (12 printed pages)

Introduction

In this article, we'll talk about the new Web-based client strategy for the MSDN Duwamish Books sample application. This strategy includes implementing three different client types to make our application available to as many browsers as possible while also taking advantage of the latest browser technologies. We'll discuss how the architecture to serve those three client types works and why certain design decisions were made.

Moving Duwamish Books to the Web

There are a lot of changes to the Duwamish Books sample application from phase to phase, but Phase 4 is by far the most exciting. This is where the application enables Duwamish Books to go into a fundamentally new business: selling books over the Web. The previous phases of the sample application have supported Duwamish Books as its business grew from one small store to a chain of stores, but this phase enables a *new* business that simply wouldn't be possible without this technology. In other words, this is the phase that should *really* help Duwamish Books make it big.

There are more than just client changes that have to be made-we made many changes designed to ensure that Duwamish Books would scale up to whatever customer demands we might have. To get an overview of all the changes and enhancements, see "Duwamish Books, Phase 4: Welcome to the Web."

Not Just Visual Basic Anymore

In previous phases of the Duwamish Books sample application, the clients have always been Microsoft® Visual Basic® clients and the users always employees of Duwamish Books connected via private network to Duwamish Books servers. That's all well and good-but what if you want to allow your customers to connect directly to you via the Internet? Are you going to require them to download and run a Visual Basic application so they can buy from you? If you do, how many customers will you have, and how soon will you be out of business? Clearly, if you're going to do business over the Internet, you'll need to support Web browser clients.

A Browsing We Will Go?

So, the most salient change between Phase 3.5 and Phase 4 is in the client layer: Instead of clients written in Visual Basic, Phase 4 uses Web browser clients connected to Duwamish Books via the Internet.

If we were still using the Duwamish Books sample application only inside the company, we could limit the number of client types we support to one (or perhaps two). But because customers will be using a variety of browsing technologies, Duwamish Books supports three different client types:

- Microsoft Internet Explorer 5.0 using XML and XSL
- Internet Explorer 4.0 using data binding (where a data source object-either a Microsoft ActiveX® Control or Java applet that provides data via a standard interface-is connected to an HTML table)
- Older HTML 3.2 clients (without scripting)

Customers using any browsing technology have access to the same set of functionality, although the technology for expressing, formatting, and sending the data is vastly different. The three clients provide the same functionality-using different methods and somewhat different user interfaces-to their users. However, the Internet Explorer clients have better performance (both on the client and server sides). The techniques developed in this phase can be used to support other types of clients, such as clients that support dynamic HTML (DHTML) but not data source objects, earlier versions of HTML, or HTML and/or XML on handheld devices. You can even support Visual Basic clients if you want!

Layered Architecture: Making It Possible to Reuse Code

Regular readers will recall that Duwamish Books is comprised of several layers, including the data source and data access layers, business Logic and workflow layers, formatting layers, and presentation layers (see Figure 1). Although Duwamish Books supports three types of clients (and could support more!), most of the layers do not change regardless of the client being used. That means the internal data format is the same regardless of client type: Data is stored in a relational database, accessed and manipulated with COM objects running under Microsoft Transaction Server (MTS), and converted into a canonical XML format for use by the clients. By "canonical XML," we mean that there is one common XML format for representing the data in the Workflow layer regardless of what client type will eventually receive the data.

Client 1 **for** **HTML 3.2 Browsers**	**Client 2** **for** **Internet Explorer 4.0**	**Client 3** **for** **Internet Explorer 5**
	Data source object (DSO): data binding and client-side XML data cache	Formatting: XML+XSL=HTML

ASP Scripting

Stores HTML data in Cache

Formating: XML+XSL=HTML

Stores XML and HTML data in Cache

Stores XML data in Cache

Workflow Component

BLL Component

DAL Component

Legend	
	Presentation Layer
	Workflow Layer
	Business Logic Layer
	Data Layer

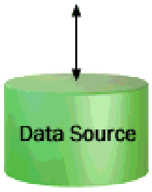

Figure 1. The application layers of the Duwamish Books sample application, Phase 4 (click a link for more information)

It’s only at the layers closer to the client (such as Formatting and Presentation) after the conversion to our canonical XML that there are differences in the architecture that depend on the client type. Because Internet Explorer 5.0 has built-in support for XML, we send the canonical XML itself over the wire to the client and let Internet Explorer worry about the formatting. (We help by providing Internet Explorer with an XSL style sheet.) Other browsers need their data "pre-digested" into HTML, so we do the appropriate conversions on the server using the canonical XML data and an XSL style sheet. With both Internet Explorer 4.0 and 5.0 we take advantage of the browser’s ability to cache data to give users a more satisfying user experience and better performance-and to reduce the round-trip burden on the servers, allowing Duwamish Books to serve more customers with less hardware.

N-tier Architecture

As you can see from Figure 1, the three client types are similar in that they use the same database and the same Business Logic, Data Access, and Workflow layers. However, they differ radically in the Workflow and Presentation layers. (See "Layered Architecture Issues" for links to discussions on this architecture in all phases of the Duwamish Books sample application.

Figure 1 shows the three client types with emphasis on the differences in the Workflow and Presentation layers. Note how some processing, such as formatting the XML output of the Workflow layer, is done on the server for some client types and on the client for one. Note also that the data being displayed is sent in XML format in the two Internet Explorer client types. Finally, note that there is extensive caching being done on the server side for all three client types to improve performance. There’s more information on the Cache component and strategy in Robert Coleridge’s article "Creating a Page Cache Object."

Client-Type Details

So without further ado, let's talk about the three types of clients Duwamish Books, Phase 4 supports: plain old HTML 3.2; Internet Explorer 4.0 using an ActiveX object as a data source for data that's sent in XML; and the all-singing, all-dancing Internet Explorer 5.0 XML/XSL approach.

Client Type 1: HTML 3.2-It's Simple!

From a client perspective, the first approach is simplest. This approach assumes only that the browser can render HTML 3.2-It doesn't even need to be able to handle client-side script! Any browser from Netscape Navigator 3.0 and Internet Explorer 3.0 to current browsers can run client type 1. In addition, many devices such as Microsoft WebTV® and Microsoft Windows® CE devices can handle HTML 3.2, although their screens may be too small to work well with the amount of data Duwamish Books displays.

Simple clients mean more work for servers

However, less complication on the client means more complication on the server. Because we're using "canonical XML" as our data representation in the Workflow layer, we'll have to translate this XML into HTML on the server before we send it to the client. This extra step means that server performance won't be as good as it would be with an Internet Explorer 5.0 client.

? and more round trips, worse performance

In addition, although browsers usually cache Web-page data, HTML 3.2 doesn't have any facilities for storing (or caching) data that the user can't see. Using HTML, we only have a visual representation of the data, not something we can manipulate by sorting or displaying only part of the data. For instance, if the user wants to sort a set of books by author rather than title, the server will have to generate the new HTML page from scratch. Similarly, the server will have to generate a new page when the user wants to see the next 20 items in their search results (once Duwamish Books has chunked results added to the API). Both of these situations involve a round trip across the Internet-that's much slower for the client (because it has to wait for the server to respond) and wasteful for the server (because the time it's spending filling these requests could be used for other customers). With the Internet Explorer solutions, which allow caching of data on the client, many round trips are eliminated, providing better performance on both sides of the wire.

Caches help performance

As you can see in Figure 1, we cache HTML pages as they're generated. This saves time on the server for frequently requested URLs-they can be generated once, and then served up out of the cache. The specific savings come in three areas: fewer database queries, fewer conversions of database results to XML, and fewer transformations from XML to HTML for rendering.

Generating HTML: how it works

The generation of the HTML from the canonical XML on the server side is easy. It's done in the ASP page by the XML DOM object. These snippets are from the ASP code in det.asp, which handles requests for item detail pages.

First, we see if the HTML page is in the cache we've created. If it is, we can just send it out.

```
’ DetKey is the cache key for this particular URL
’ Attempt to retrieve HTML from cache
  DetHTML = Application("Cache")(DetKey)
’ If HTML was not cached, generate HTML and add to cache
  If DetHTML = "" Then
   ’ generate HTML from XML storing in DetHTML; shown next
   ’ ...
  End If ’ this section will be repeated below
’ Write HTML detail to client
  Response.BinaryWrite(DetHTML)
```

We use Response.BinaryWrite rather than just Response.Write because the text, stored in DetHTML, is in UTF-8 character format, which allows us to use extended character sets (as for Asian languages). If we used Response.Write, the results would be mangled because it would be assumed that the results where ANSI.

The code to generate the HTML from XML, if it’s not already in the cache, is relatively simple. (The following four code snippets go inside the **if** statement in the preceding code example.)

First, we create objects to hold the XML and XSL documents.

```
’    Instantiate XMLDOM objects for XML and XSL
     Set XMLDoc = Server.CreateObject("Microsoft.XMLDOM")
     Set XSLDoc = Server.CreateObject("Microsoft.XMLDOM")
```

Next, we create a Workflow object and use it to obtain the XML data. We get the XSL style sheet directly from the server.

```
’    Instantiate Workflow component
     Set WFL = Server.CreateObject("d4Wflow.cWorkflow")
’    Load XMLDoc with XML representation of item from workflow
     XMLDoc.async = false
     XMLDoc.loadXML(WFL.GetItemByItemId(DetId))
’    Load XSLDoc with style sheet
     XSLDoc.async = false
     XSLDoc.load(Server.MapPath("det.xsl"))
```

Next, we call the magical **transformNode** method to cause the object containing the XML data to use the XSL style sheet to generate HTML. And, having generated this HTML, we add it to the cache so we won’t have to generate it again.

```
’    Generate HTML from XML and XSL
     DetHTML = XMLDoc.documentElement.transformNode(
                XSLDoc.documentElement)
’    Add HTML to cache
     Application("Cache").Add DetKey, DetHTML
```

Finally, we rejoin the other fork of the **if** statement (we showed this before) and write the HTML code we generated into the document that will be sent to the client.

```
    End If ’ this section is a repeat of the code above
’   Write HTML detail to client
    Response.BinaryWrite(DetHTML)
```

Why didn’t we use client-side script for the HTML 3.2 client?

You can get a little more sophisticated with version 3 browsers (although Duwamish Books' HTML 3.2 client doesn't) by using script on the client (JScript®/JavaScript or VBScript). The reason that Duwamish Books' HTML 3.2 client doesn't use client-side script is that there's little advantage to doing so: In version 3 browsers, the capabilities of script are relatively limited. (Doing client-side input validation via script is reasonable, but left as an exercise for the reader.) In addition, if we used script, browsers that are HTML 3.2-compliant but don't support script would fail, or at least have reduced functionality.

Client Type 2: Internet Explorer 4.0 with Data Binding

So, we've got a client type that will run on most any browser, but it has some problems. Because it uses only relatively simple HTML, it generates a lot of round trips to the server. That's slow for the user and expensive for Duwamish Books (when multiplied by thousands of users). And we had to convert our XML to HTML on the server-added processing time and expense there as well.

Data binding for improved performance

The major help for improving performance comes from another feature of Internet Explorer and DHTML: data binding.

Data binding allows us to create a data source object (DSO) that we can connect to various elements of the DHTML document, such as a table. In this way, the table can display the contents of the DSO. The DHTML table can even handle scrolling and sorting without needing a trip to the server.

This is a powerful feature of Duwamish Books, so we'll take advantage of it-despite one unpopular requirement: Data binding requires that a COM component (which is usually not visible on the page) be downloaded and installed on the user's machine. (It is possible to use a Java applet, but doing so exchanges a dependence on a COM component for a dependence on a Java applet and doesn't increase portability much because the data-binding feature is currently only implemented in Internet Explorer.)

Our DSO

Our client-side data source object (DSO) component, written in Microsoft Visual Basic, fetches XML from the server and parses the XML into internal arrays. We'll use this DSO to store categories, keyword searches, and natural-language search results. The DSO exposes the internal arrays via the DHTML data-binding interfaces, so search results can be displayed in an HTML table entity because the table is bound to the DSO.

More data than meets the eye

In addition to the invisible data source object, we also have a number of other invisible elements on these DHTML pages.

First, we use DHTML's ability to hide and show objects so we can send down on many pages more data than is initially shown. As the user clicks, we use script to hide the current data and show other data-all without any additional interaction with the server. This requires that more data be sent for the page originally, but because it eliminates round trips, overall performance is better. Duwamish currently does this only to allow users to quickly switch between category and advanced searches very quickly, but you could use these techniques in other areas of the application as well.

Lastly, the XML data is retrieved via a standard HTTP request into an invisible inline frame (IFrame). Some script is also written to this frame, which connects to the DSO object and sends the XML to it to be parsed.

Data binding: how it works

Recall that we have an HTML page that contains a table bound to an invisible DSO and an invisible IFrame that contains the XML data (and a little script) for the DSO. The DSO reads the XML, parses it into internal arrays, and then makes the results available via the data-binding interfaces to the data-bound table for display.

Here is the HTML code from default.htm that creates the DSO and binds it to a table (also shown). Note that the table is bound to the DSO using the "datascr="#DSC1" attribute and the spans within the table cells are bound using the "DATAFLD="field_name" attribute. Also note that there’s only one table row listed here: When the page is displayed, the table rows will repeat as often as necessary.

When the user clicks in the table, this client-side VBScript code from default.htm is called. When the user selects a category, the table1 event handler executes the **ListResultsCategory** method we wrote when we wrote the DSO.

```
Sub Table1_OnClick()
...
’   Retrieve the DSO data record associated
’      with the data bound table element
    DSC1.Recordset.absoluteposition =
         window.event.srcElement.recordNumber
...
’   If user selected a Category
    CategoryId = DSC1.Recordset("CategoryId")
...
    If CategoryId <> "" Then
’     Invoke DSO method to display selected category
      DSC1.ListResultsCategory CategoryId
...
    End If
End Sub
```

If the DSO does not have the selected category in its cache, it raises an event that is handled by the following script code (from default.htm), which fetches an XML representation of the requested category into an IFrame. The URL is something like "xmlcat.asp?CategoryId=N".

```
Sub DSC1_FetchData(URL)  ' URL something like "xmlcat.asp?CategoryId=N"
  iframe2.location.href = URL  ' navigate invisible frame to new URL
End Sub
```

When this code executes on the client, the following code (from xmlcat.asp on the server) serves up an XML representation of the category. (This code is similar to the code that generated the HTML page in the HTML 3.2 client, with the addition of some extra code at the end of the response written to the client. The following code connects with the DSO and tells it to load up the XML we just received:

```
'Get PKId of category from query string
  CatId = Clng(request.QueryString("CategoryId"))
'Convert PKId to a key as used by the cache
  CatKey = IdToCatKey(CatId)
'Attempt to retrieve XML from cache
  CatXML = Application("Cache")(CatKey)
' If XML was not cached, get it and add to cache
  If CatXML = "" Then
' Instantiate Workflow component
    Set WFL = Server.CreateObject("d4Wflow.cWorkflow")
' Get XML representation of category from workflow
    CatXML = (WFL.GetCategory(CatId))
' Add XML to cache
    Application("Cache").Add CatKey, CatXML
  End If
' Add   client-side script to data that will send XML to DSO
  CatPak = "" & scCR
  CatPak = CatPak & "" & FetchSeq & ""
  CatPak = CatPak & CatXML & scCR& " " & scCR
  CatPak = CatPak & "" & scCR
' Write XML to client
  Response.BinaryWrite(Chrw(&HFEFF) & CatPak)
```

We mentioned already why we did the binary write-so that our UTF-8 code would be preserved. In this case, we also have to add some special bytes to the beginning of the stream to indicate to the receiving computer which byte ordering was used when the stream was written. If the receiving computer reads &HFEFF, the byte ordering of the stream is the same as that computer uses. If it reads &HFFFE, the byte ordering is swapped and the computer has to switch the order of every pair of bytes.

Once the XML is loaded into the invisible IFrame, the **Results** property in the DSO is set to point to the XML code and the property handler transforms the XML representation of the category to internal arrays. (We had to write this ourselves: Because this client type is not built for Internet Explorer 5.0, the XML DOM object isn't available.) For more detailed information, including some code examples, see Steve Kirk's article, "Workflow Design for a Web Commerce Application."

After that, the data is automatically passed to the table to be displayed, and life is good.

Why didn?t we use DHTML?s fancy display capabilities?

Dynamic HTML adds additional capabilities. Because the entire document is accessible through the document object model (DOM), you can do some incredible things, including moving elements, hiding and showing elements, changing sizes and colors, and so forth.

Doing some of these things can add some sizzle to the Duwamish Books Web site, but they won't help us improve performance. So, while many developers use DHTML and even cross-browser DHTML to spice things up, we didn't spend any time on the eye candy-we leave that as an exercise for you.

Client Type 3: Internet Explorer 5.0 using XML

Our final client type is Internet Explorer 5.0. For these clients, we take advantage of DHTML features, client-side scripting, and the groundbreaking built-in XML support in the browser. This gives roughly the same performance as the Internet Explorer 4.0 solution, but the programming is considerably simpler.

XML direct to your browser

As a result of Internet Explorer 5.0 providing built-in XML support, we're able to reduce the server-side processing considerably. In the HTML 3.2 approach, the server was responsible for converting the XML into HTML. In the Internet Explorer 4.0 approach, much of the data was passed as XML (into the invisible IFrame), but we still had to deal with HTML on the server side for the pages that are sent.

For Internet Explorer 5.0 clients, the browser fetches data from the server into non-visible HTML elements, such as a DIV element or an XML data island. (An XML data island is an HTML element that contains XML data.) From there, with the help of the XSL style sheet, it's reformatted into HTML by the browser. As you see in the architecture diagram, this makes life easier for the server by moving the XML-to-HTML conversion to the browser. As a result, the server scales better without causing significant slowdowns on the client side. And because, as with the Internet Explorer 4.0 approach, we can send data that's not displayed, we can reduce round trips and improve both client and server performance.

XML on the browser: how it works

The following code examples are from default.htm. The first two show the fetch of XML data into an XML data island. The third (and last) shows the client-side transformation into HTML using XML DOM and an XSL style sheet. Note that because we use a style sheet, it's easy to change how the data looks.

First, code to fetch XML from the server and stuff it into an HTML DIV element.

```
Sub DisplayCategory(ByVal CategoryId)
   ' ...
  LoadCategoryXML(CategoryId)
  CategoryHTML = GetCategoryHTML()
   ' ...
  divResLst.innerHTML = CategoryHTML
End Sub
```

This almost couldn't be simpler. Next, the following code shows how we fetch the XML representation of Category into an XML data island. We set the **XMLCatSchDoc.async** property to false, so we'll wait for the XML to be loaded before going on.

```
Sub LoadCategoryXML(ByVal CategoryId)
  XMLCatSchDoc.async = false
```

```
  XMLCatSchDoc.load("xmlcat.asp?" & CategoryId)
End Sub
```

Now that we've got the XML, we need to format it into HTML. Here's a code example showing client-side formatting of XML into HTML using XML DOM. Again, this is about as easy as it gets.

```
Function GetCategoryHTML()
  XSLDoc.async = false
  XSLDoc.load("cat.xsl")
  GetCategoryHTML = XMLCatSchDoc.documentElement.transformNode
                      (XSLDoc.documentElement)
End Function
```

Conclusion

We've talked a bit about how Duwamish Books, Phase 4 differs from previous phases. The major difference is that clients have shifted from Visual Basic clients used only within Duwamish Books to Web-based clients used by customers anywhere in the world. This fundamental shift opens vast new business opportunities for Duwamish Books-and, if it were publicly traded, would do great things for its stock price.

Next, we looked at the architecture of this phase, which supports three different client types. Among the most interesting parts of the architecture are that the code required to support multiple client types is isolated into as few layers as possible and that this phase makes extensive use of caching to reduce database and server activity. Also notable is that the "canonical form" for data just before it's sent to the client is XML, with XML eventually rendered as HTML either on the server or client, depending on client type.

Finally, we looked at how the different client types work. Duwamish Books supports very simple HTML 3.2 clients, but doing so is expensive for the servers and slow for the users due to the very limited capabilities of HTML 3.2. So, it also supports caching some data on the client using a data source object in Internet Explorer 4.0 and using XML inserted in various HTML elements in Internet Explorer 5.0. This caching improves the user experience by reducing the need for round trips to the server. The caching also improves server performance by reducing the amount of work on the part of the server.

Comments? We welcome your feedback at duwamish@microsoft.com.