

# Visual J++ Developer's Journal

March 1999

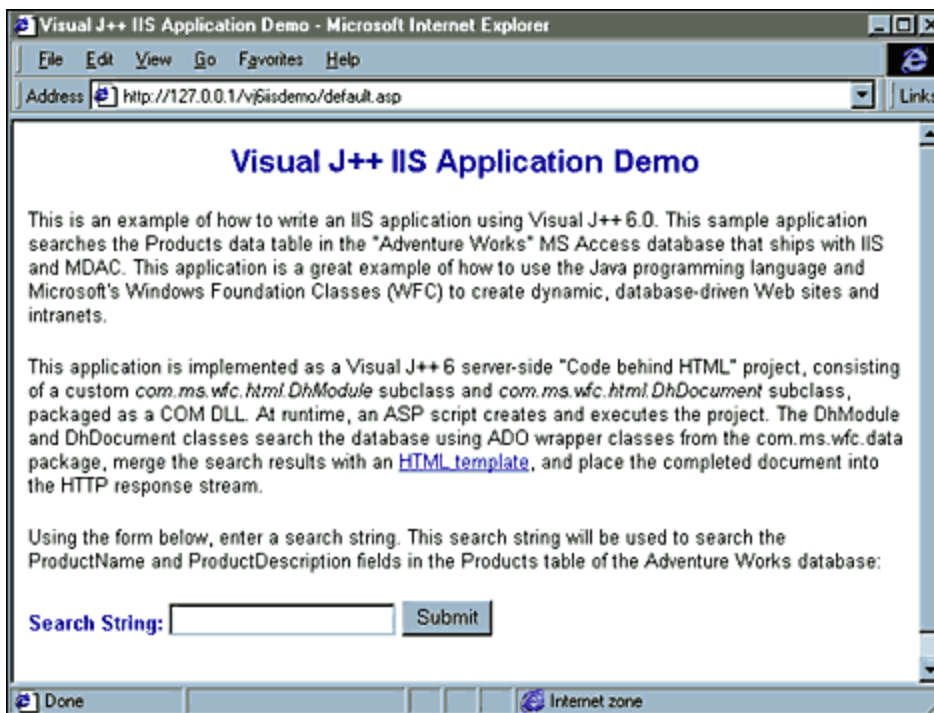
## Developing Data-Driven Web sites with Code-Behind HTML

by Andy Hoskinson

If you're a Java programmer who develops Web-based solutions for the Microsoft Active Server platform, you now have a powerful new tool at your disposal. With Visual J++ 6.0, you can use the Java programming language and Microsoft's Windows Foundation Classes (WFC) to create dynamic, database-driven Web sites and intranets. In this two-part article, we'll show you how to create a product database search engine for your Internet Information Server (IIS) 4 Web site using Visual J++.

In this first part of our article, we'll introduce *Code-behind HTML* and show you how to develop the Visual J++ code for a Code-behind HTML project. In the second part, we'll develop the HTML template and the ASP script you'll need, and show you how to build and deploy the project. Figure A shows the search tool we'll be developing.

**Figure A:** The Adventure Works product search tool is implemented as a server-side Visual J++ Code-behind HTML project.



### The Code-behind HTML programming model

Visual J++ 6.0 allows you to create Code-behind HTML projects. With this programming model, you can create a Java class that binds to and interacts with an HTML document, using the `com.ms.wfc.html` package that ships with Visual J++ 6.0 and the Microsoft VM (Virtual Machine) for Java.

The `com.ms.wfc.html` package consists of Java classes that wrap Microsoft's Dynamic HTML Document Object Model (DOM). With a Code-behind HTML project, you write a Java class that extends the `com.ms.wfc.html.DhDocument` class, and then you site it on an HTML document using an instance of the `com.ms.wfc.html.DhModule` class.

## Using Code-behind HTML on the client

A Visual J++ Code-behind HTML project can run on either the client or the server. On the client, you can implement it using the `<OBJECT>` tag in an HTML page hosted by Microsoft Internet Explorer 4 (IE4) or higher. In this context, your `DhDocument` class can programmatically add new HTML elements to its host page, bind to existing elements, and even handle events that are raised by objects on the host HTML document. For example, you can hook an event handler in your Java class to the `onClick` event of an HTML button on your page.

## Using Code-behind HTML on the server

The client-side approach, while interesting and powerful, has one drawback: it limits your end users to IE4 or higher as their choice of browsers. Fortunately, to get around this limitation, you can use the `com.ms.wfc.html` package on your IIS Web site to create dynamic Web content that's viewable by virtually any browser on the market today.

Here's how it works. On the server, an ASP script creates an instance of the `DhModule` class (or a custom subclass). Next, it passes the class name of your `com.ms.wfc.html.DhDocument` subclass, the local path to an HTML template, and any other application-specific data to the `DhModule`. The `DhModule` creates an instance of the `DhDocument` class, and merges its output with the HTML template. From there, the merged document is placed into the HTTP response stream as raw HTML. The browser receives it and renders it for the user.

The biggest advantage you gain from developing IIS applications this way is the ability to separate design from content and functionality. Your developers can focus on writing good, clean, reusable code that generates dynamic content based on your organization's business rules without having to worry about the look and feel. Conversely, your designers can focus on creating great looking Web templates without having to worry about integrating any application logic into their work. The Microsoft VM for Java brings it all together seamlessly at runtime.

## Developing a server-side Code-behind HTML project

To illustrate this, we'll create a sample server-side Code-behind HTML project. This sample application searches the Products data table in the Adventure Works MS Access database that ships with IIS and Microsoft Data Access Components (MDAC). The application, shown in Figure A, illustrates several concepts that we'll cover in this month's article, and in the second part next month:

- Creating a Code-behind HTML Visual J++ project.
- Writing custom `com.ms.wfc.html.DhModule` and `com.ms.wfc.html.DhDocument` subclasses.
- Using the `com.ms.wfc.data` classes to search an OLE DB data source.
- Using the `com.ms.wfc.html.DhElement` classes/subclasses to generate HTML dynamically, and

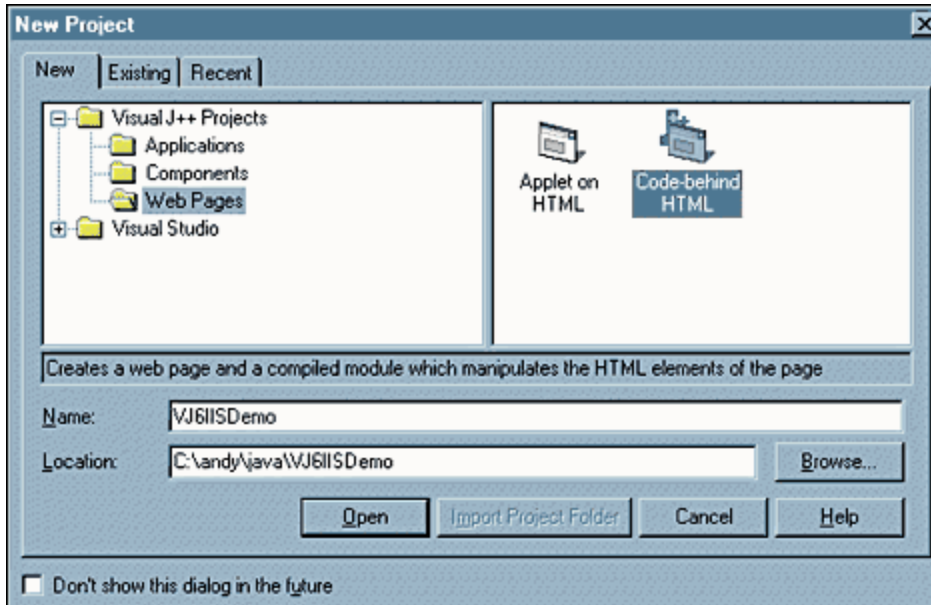
binding those elements to existing elements on an HTML template.

- Writing an ASP wrapper script to run the Code-behind HTML project on the server.
- Deploying the project to an IIS server.

### Step 1: Creating a new Code-behind HTML project

When you first open up the Visual J++ 6.0 Integrated Development Environment (IDE), it greets you with the New Project dialog box, as shown in Figure B.

**Figure B:** Start the project with Visual J++ 6.0's New Project dialog box.



To create a new project, first ensure that the New tab is selected. Then, in the Tree view control on the left side of the dialog box, select the Web Pages node. Now, on the right side, select Code-behind HTML and type in a name for your project in the Name field. We named ours *VJ6IISDemo*. Finally, browse to the location on your local file system where you want to save your project and click Open. The Visual J++ IDE creates the project, and also adds a Java class called Class1 to the project. We'll get to this class later; first, we'll create a new class.

### Step 2: Creating a custom DhModule subclass

Once you've created your project, the next step is to create a custom DhModule subclass. As I mentioned before, this subclass is used to site your DhDocument class. In other words, the DhModule subclass provides the glue between your DhDocument class and your runtime environment--either the browser (if it's running on the client), or ASP (if it's running on the server).

For our custom DhModule subclass, we'll add an instance variable and get and set accessor methods. As you'll see later, our ASP script uses these methods to pass a search string to our DhDocument subclass. We'll name our DhDocument subclass SearchModule. Listing A shows the source for SearchModule.java.

#### Listing A: SearchModule.java

```
package net.hoskinson.iis;public class SearchModule extends
=>com.ms.wfc.html.DhModule {
    private String m_SearchString = "";

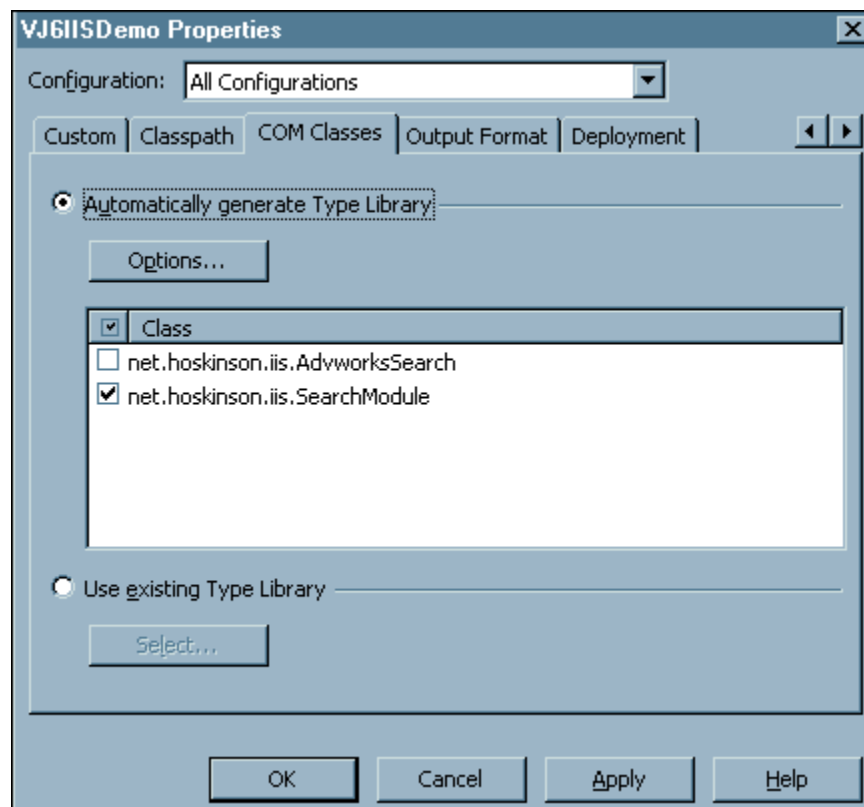
    public void setSearchString(String
newString) {
        m_SearchString = newString;
    }

    public String getSearchString() {
        return m_SearchString;
    }
}
```

### Step 3: Exposing the DhModule subclass as a COM class

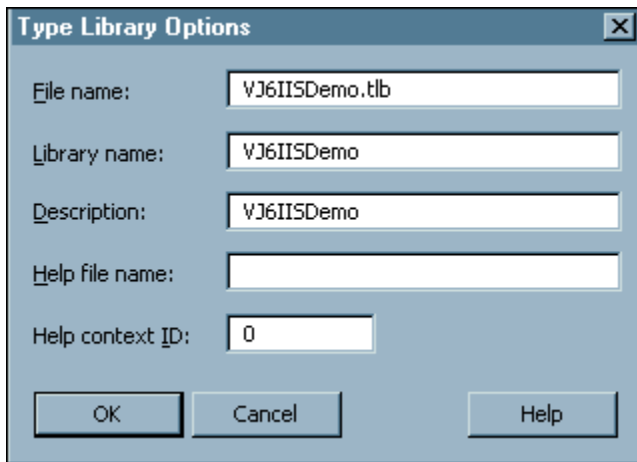
The next step is to tell Visual J++ to treat the SearchModule class as a COM class. To do this, select Project from the menu bar, and then click *MyProjectName* Properties (where *MyProjectName* is the name of the project). This invokes the Properties dialog box, as shown in Figure C. Select the COM Classes tab.

**Figure C:** Select the COM Classes tab in the Properties dialog box.



Now, select the check box for the SearchModule class, and click the Options button. This will bring up the Type Library Options dialog box, as shown in Figure D.

**Figure D:** The Type Library Options dialog box contains default values that you can change.



The File name, Library name, and Description are filled in with default values based on the project name, but you can change them, if desired. At compile time, Visual J++ generates a type library containing a COM interface for your class's public members. This allows your class to be exposed as a COM object.

After you click OK and go back to the SearchModule.java source code, you'll notice that Visual J++ added the following comment to the source code, just before the class declaration:

```
/**
 *@com.register ( clsid =
 * 4C8EFC37-70BB-11D2-87A0-D0FE09C10700,
 * typelib =
 * 4C8EFC36-70BB-11D2-87A0-D0FE09C10700 )
 */
```

The Visual J++ compiler uses this declaration to register your class as a COM object when you compile the project.

#### Step 4: Creating a custom DhDocument subclass

Next, we'll create a DhDocument subclass. Since this is the most complicated part of the project, we'll break it down into easily digestible blocks of code. See Listing B at the end of this article for the entire code listing for this subclass. As I mentioned earlier, Visual J++ created a skeleton class called Class1 when it created the Code-behind HTML project. Use this class as the boilerplate code for the DhDocument subclass.

First, rename Class1 to something more descriptive. We renamed ours *AdvworksSearch*. To do this, go to the Project Explorer frame, click Class1.java, and right-click. Select Rename from the pop-up menu, and type in a new name for your DhDocument subclass. This renames the source code file on the underlying file system.

Next, update the class declaration to reflect your new class name. You should also rename the class constructor to reflect the new name. When you're done, your class declaration should look something like this:

```
public class AdvworksSearch extends
=>DhDocument
```

See Listing B, Section 1 for the class declarations.

The next step is to provide functionality for the private void `initForm()` method. This is the meat of your `DhDocument` subclass. This method, which is called by the class's constructor, contains code that searches the database, generates the search results as an HTML table, and binds this table to a corresponding element on an existing HTML template.

The first thing to do in the `initForm()` method is to get a reference to the parent `DhModule`. Obtaining this reference allows you to get data from the parent `DhModule`. The `DhDocument` subclass provides a `getModule()` method that does exactly that. However, since we're using the `SearchModule` class (which derives from `DhModule` created in the previous step), we need to cast the instance of `com.ms.wfc.html.DhModule` returned by `getModule()` to a variable of type `SearchModule`. This allows us to call the `getSearchString()` method that we defined in the `SearchModule` class in Step 2. This is how `AdvworksSearch` gets the search string that we'll use to search the database:

```
m_Module = (SearchModule)getModule();  
m_Filter = m_Module.getSearchString();
```

Next, create an instance of the `com.ms.wfc.html.DhEdit` class, and set its value to the current search string. The `DhEdit` class is used to represent a standard HTML input text box. Later, we'll bind this instance to the existing input text box on our HTML template:

```
m_InputBox = new DhEdit();  
m_InputBox.setText(m_Filter);
```

This section of code is found in Listing B, Section 2.

Continuing on, create the table used to display the search results, and add some column headers to it. We'll use three classes from the `com.ms.wfc.html` package to do this: the `DhTable`, `DhRow`, and `DhCell` classes (Listing B, Section 3).

Now it's time to search the database. To do this, we use classes from the `com.ms.wfc.data` package. These classes are Java wrapper classes for Microsoft's ActiveX Data Objects (ADO) library. *ADO* is a COM object model that wraps OLE DB, Microsoft's universal data access interface specification. Through ADO (and, by extension, the `com.ms.wfc.data` package), you can search any data source for which an OLE DB provider has been written. This includes ODBC databases, SQL Server, Oracle, MS Access, Index Server, and a host of other data sources.

To get started, open a connection to the database. To do this, create an instance of the `Connection` class, passing the database's connection string to the constructor. In our case, we'll use the Adventure Works database's ODBC System DSN. Next, call the `open()` method (Listing B, Section 4).

Now, form the SQL statement based on the value of the search string, and execute the query by calling the `execute()` method of the `Connection` class, passing the SQL statement to it (Listing B, Section 5). This returns an instance of the `Recordset` class containing our search results rowset.

Loop through the recordset, adding a row to the table for each record (Listing B, Section 6). Then, after enumerating through the recordset, add a table footer to show the total record count (Listing B, Section 7).

If the search returned any records, we need to display those results to the user. If the search returned no

records, we want to inform the user of that. We also want to update the text-input box to display the current search string to the user.

To do this, call the `setBoundElements` method of the `DhDocument` subclass. This method takes an array of `DhElement` objects. The `DhElement` class is the superclass for most of the classes contained in the `com.ms.wfc.html` package.

The `DhElements` that we pass to this method are bound to their corresponding peer elements on the HTML template using the `setBindID` method of the `DhElement` class. To accomplish this, call this method for each of your `DhElements`, passing the string that corresponds to the ID tag of the HTML template object to which you want that particular element bound (Listing B, Section 8). If this seems a little confusing, don't worry; it will become clearer when we talk about our HTML template in Step 5 in part 2 of this article.

Finally, clean up by closing down the database connection. Then, tag the objects for garbage collection (Listing B, Section 9).

## Conclusion

In this first part of our article on developing a server-side Visual J++ 6.0 Code-behind HTML project, we looked at the steps necessary to develop the Visual J++ code. We'll finish up next month with the final steps needed to complete our project.

### Listing B: AdvworksSearch.java code

```
//***** Section 1 *****/
package net.hoskinson.iis;import com.ms.wfc.html.*;
import com.ms.wfc.core.*;
import com.ms.wfc.ui.*;
import com.ms.wfc.data.*;public class AdvworksSearch extends DhDocument {
    //The database's connection string
    public final String
    =>CONNECTION_STRING = "DSN=AdvWorks";

    //The HTML template's search string text box ID
    private final String TEXT_INPUT_ID = "filter";

    //The HTML template's search results DIV object ID
    //The search results table binds with this object
    private final String
    =>RESULTS_TABLE_ID = "search_results";

    //The calling DhModule subclass instance
    private SearchModule m_Module;

    //Our database connection object
    private com.ms.wfc.data.Connection m_Conn;

    //The recordset returned by our query
    private com.ms.wfc.data.Recordset m_RS;

    //Our SQL statement
    private String m_SQL;

    //Our search string
```

```

private String m_Filter;

//A counter to keep a running tally
//as we scroll through the recordset
private int m_RecordCount = 0;

//The search string text input box
private DhEdit m_InputBox;

//The search results HTML table
private com.ms.wfc.html.DhTable m_Table;

/**
 * The constructor calls the initForm method.
 */
public AdvworksSearch() {
    initForm();
}
/**
 * Override dispose to release any resources used.
 * Minimally, call the superclass's dispose
 * method.
 */
public void dispose()
{
    super.dispose();
}
//***** Section 2*****
/**
 * This is the "meat" of your class.
 * Add code to search your database, generate
 * HTML elements, and bind to elements on an
 * existing HTML template.
 */
private void initForm() {
    /*Get a reference to the calling DhModule. If
    you're using a custom subclass, make sure you
    cast, because getModule() returns an instance
    of com.ms.wfc.html.DhModule.*/
    m_Module = (SearchModule)getModule();

    //Get the search string from our DhModule
    m_Filter=m_Module.getSearchString();

    /*If the search string is not 'null or zero
    length, query the database and return
    results\*/
    if (!(m_Filter.equals(null) ||
    =>m_Filter.equals("")))) {
        m_InputBox = new DhEdit();

        /*Set the value of the text box to the
        current search string*/
        m_InputBox.setText(m_Filter);
    }
}
//***** Section 3 *****

    m_Table = new DhTable();

    //Create the table and add some column
headers

```

```

        m_Table.setBorder(1);
        m_Table.setCellSpacing(0);
        DhRow tableHeading = new DhRow();
tableHeading.setBackgroundColor(new
        =>Color(192,192,192));
        tableHeading.add(new DhCell("Name"));
        tableHeading.add(new DhCell("Type"));
        tableHeading.add(new DhCell("Description"));
        tableHeading.add(new DhCell(
        =>"Available Sizes"));
        tableHeading.add(new DhCell("Price"));
m_Table.add(tableHeading);//***** Section 4 *****

        //Open a connection to our database
        m_Conn = new Connection
(CONNECTION_STRING);
        m_Conn.open();//***** Section 5 *****
        //Form our SQL statement based on the value
        //of our search string
        m_SQL = "select * from Products where
        =>ProductName like \'%%\' + m_Filter +
        =>"%%\' OR ProductDescription like \'%%\' +
        =>m_Filter + "%%\' Order By ProductName";

        //Execute the query, returning a recordset
        m_RS = m_Conn.execute(m_SQL);

//***** Section 6 *****
        /*Loop through the recordset, adding a row
        to the table for each record*/
        while (!m_RS.getEOF()) {
            DhRow tableBody = new DhRow();
tableBody.add(newDhCell(
                =>ConvertNullToZeroLengthString(
                =>m_RS.getField("ProductName").
                =>getValue().toString()));
                tableBody.add(new DhCell(
                =>ConvertNullToZeroLengthString(
                =>m_RS.getField("ProductType").
                =>getValue().toString()));
                tableBody.add(new DhCell
                =>(ConvertNullToZeroLengthString(
                =>m_RS.getField
("ProductDescription").
                =>getValue().toString()));
                tableBody.add(new DhCell(
                =>ConvertNullToZeroLengthString(
                =>m_RS.getField("ProductSize").
                =>getValue().toString()));
                tableBody.add(new DhCell("$" +
                =>ConvertNullToZeroLengthString
                =>(m_RS.getField("UnitPrice").
                =>getValue().toString()));
                m_Table.add(tableBody);
                m_RS.moveNext();
                m_RecordCount++;
            }//***** Section 7 *****
        //record count
        DhRow tableFooter = new DhRow();
        tableFooter.setBackgroundColor(

```

```

=>new Color(192,192,192));
DhCell footerLabel = new DhCell(
=>"Total records meeting criteria \'' +
=>m_Filter + "\" :");
footerLabel.setColSpan(4);
tableFooter.add(footerLabel);
tableFooter.add(new DhCell(
=>" " + m_RecordCount));
m_Table.add(tableFooter);/******* Section 8 ****
//records...
if (m_RecordCount == 0) {
    setBoundElements( new DhElement[] {
=>new DhText("Your search did not
=>return any results. Please try
=>again.").setBindID(RESULTS_

TABLE_ID),
    =>m_InputBox.setBindID(TEXT_
INPUT_ID)});

/*Otherwise, bind the search results table
to the DIV object on the HTML template.
Also, bind the DhEdit object to
the text
box to display the current
search string*/
} else {
    setBoundElements( new DhElement
[] {
=>m_Table.setBindID(RESULTS_
TABLE_ID),
=>m_InputBox.setBindID(TEXT_
INPUT_ID)});
}

/******* Section 9 *****/
//Clean up our database objects
m_RS.close();
m_Conn.close();
m_RS = null;
m_Conn = null;
}

/******* Section 10 *****/
/**
 * This method returns an HTML &nbsp; if field
 * value passed to it contains a null value
 */
String ConvertNullToZeroLengthString(
=>String strAnyString) {
    if(strAnyString == null) {
        return "&nbsp;";
    } else {
        return strAnyString;
    }
}
}

```

of their respective owners.