BlueDragon™ 9.0

Integrating CFML with ASP.NET

and the Microsoft .NET Framework

NEW ATLANTA COMMUNICATIONS, LLC

BlueDragon™ 9.0

Integrating CFML with ASP.NET and the Microsoft .NET Framework

October 6, 2013
Version 9.0

100 Prospect Place • Alpharetta, Georgia 30005-5445
Phone 678.712.5075 • Fax 888.267.4020
http://www.newatlanta.com

Contents

1 INTRODUCTION 5

1.1 About This Document 5

1.2 About CFML 5

1.3 About BlueDragon 5

1.4 Technical Support 6

1.5 Other Documentation 6

2 DEPLOYING CFML ON .NET 7

2.1 Benefits of Deploying CFML on .NET 7

3 SHARING VARIABLES BETWEEN CFML AND ASP.NET 8

3.1 Sharing Session Variables between CFML and ASP.NET 8
3.1.1 Setting Support for ASP.NET Sessions 8
3.1.2 Sharing Session Variables When CFAPPLICATION Has No NAME 9
3.1.3 Sharing Session Variables When CFAPPLICATION Has a NAME 10

3.2 Sharing Application Variables Between CFML and ASP.NET 11

3.3 Sharing Request Variables Between CFML and ASP.NET 11
3.3.1 Sharing CFML Request Scope with ASP.NET 12
3.3.2 Sharing ASP.NET Context and Request Scope with CFML 12

3.4 Sharing Client Variables Between CFML and ASP.NET 13

3.5 Sharing Local (Variables Scope) Variables Between CFML and ASP.NET 14
3.5.1 Additional Features of the Variables Object 15

3.6 Sharing Complex CFML Variables 15
3.6.1 Sharing Arrays and ILists 15
3.6.2 Sharing Structures and Hashtables 17
3.6.3 Sharing Query Result Data 19

4 INTERACTION BETWEEN CFML AND ASP.NET PAGES 21

4.1 Invoking Components and Objects between CFML and ASP.NET Pages 21
4.1.1 Invoking CFML Components (CFCs) within ASP.NET 21
4.1.2 Invoking .NET Components within CFML 24
4.1.3 Invoking COM Objects from CFML 27

4.2 Including Between CFML and ASP.NET Pages 28
4.2.1 Including ASP.NET Page Output within CFML 28
4.2.2 Including CFML Page Output within ASP.NET Pages 29

4.3 Transferring Control Between CFML and ASP.NET Pages 30
4.3.1 Transferring Control from CFML to ASP.NET Pages 30
4.3.2 Forwarding from ASP.NET to CFML Pages 31

4.4 Calling CFML Custom Tags Within an ASP.NET Page 31
4.4.1 Calling a Simple Single CFML Custom Tag 31
4.4.2 Passing Attributes to a CFML Custom Tag 32
4.4.3 Simulating Caller Scope Data When Calling a CFML Custom Tag in ASP.NET 33
4.4.4 Calling Paired Custom Tags in ASP.NET 33
4.4.5 Nesting Custom Tags 34

4.5 Invoking CFX Custom Tags in .NET Languages 34

4.6 Executing CFML Code Inline Within an ASP.NET Page 34
4.6.1 Using the <cf:inline> Control 34
4.6.2 Evaluating Possible Use of CF Inline Scripting 35
4.6.3 Alternative: Using the CfInline() Method 35

4.7 Generating CFML Output within an ASP.NET Page 36
4.7.1 Using the <cf:output> Control 36
4.7.2 Alternative: Using the CfOutput() Method 36

5 ADDITIONAL INFORMATION 38

5.1 BlueDragon.CfmPage and NewAtlanta.BlueDragon 38
5.1.1 BlueDragon.CfmPage Class 38
5.1.2 NewAtlanta.BlueDragon Library 39

5.2 Using Code-Behind with BlueDragon Enhancements 39

5.3 Requirement to Use get_ Syntax For Accessing Object Properties 40

5.4 Root-relative Paths Are Relative to the Web Application 41

5.5 Executing Application.cfm When Running an ASP.NET Page 41

5.6 Converting Sample .NET Object Calls to a CFML Equivalent 41
5.6.1 Determining the Complete Class Name 42
5.6.2 Determining Available Properties and Methods: Two Ways 42

5.7 Editors for Creating/Editing CFML and ASP.NET Pages 44

1 Introduction

BlueDragon 9.0 for the Microsoft .NET Framework (referred to hereafter as BlueDragon.NET) allows CFML applications to be deployed on Windows servers running the Microsoft .NET Framework, the Microsoft IIS web server, and ASP.NET.

While most web applications on .NET are built with ASP.NET and other components of the .NET framework, BlueDragon makes it possible for the .NET framework to also process CFML applications. Indeed, it's the only way to run CFML on the .NET Framework. More than that, it's about empowering CFML to integrate with your organization's .NET development and take full advantage of the enterprise features of this strategic platform.

1.1 About This Document

This document details the many forms of integration that are possible between CFML pages and native .NET components, including ASP.NET pages. It supplements the information presented in the manual, *Deploying CFML on ASP.NET and the Microsoft .NET Framework*, which discusses implementation and deployment details, as well as outlines the many benefits of deploying CFML on .NET.

1.2 About CFML

CFML is a popular server-side scripting language for building dynamic database-driven web sites. Unlike alternatives such as ASP or PHP, CFML is based primarily on HTML-like markup tags (though CFML also contains a scripting language component). CFML is characterized by its low learning curve and ease-of-use, particularly for web developers who do not have a technical background in programming languages such as C/C++ or Java. CFML was originally developed by Allaire Corporation in the late 1990's; Allaire was acquired by Macromedia, Inc. in early 2001, which in turn was acquired by Adobe Systems Inc. in late 2005.

Over the past several years, many organizations have begun adopting standards-based application servers for their Internet and intranet web site deployments. In particular, there has been a significant migration to application servers based on the .NET Framework. This standardization on .NET (and ASP.NET) creates a problem for organizations that have legacy applications implemented in CFML: prior to the introduction of BlueDragon these applications could only be deployed on proprietary Adobe ColdFusion application servers.

1.3 About BlueDragon

The core technology of BlueDragon is a CFML runtime and execution module that, in BlueDragon.NET, is implemented as a standard .NET HTTPHandler. This allows the deployment of CFML pages onto the .NET framework and IIS without installing proprietary Adobe ColdFusion server software.

BlueDragon is highly compatible with Adobe's ColdFusion 9.0 Server, with some limitations but also many enhancements. Besides those mentioned in this guide, see the *BlueDragon 9.0 CFML Compatibility Guide* and *BlueDragon 9.0 CFML Enhancements Guide* for details:

http://www.newatlanta.com/products/bluedragon/self_help/docs/index.cfm

BlueDragon is a highly optimized, high-performance CFML runtime engine. CFML pages are compiled into an internal representation that is cached in memory and executed by the BlueDragon runtime when CFML pages are requested by client browsers.

1.4 Technical Support

If you're having difficulty installing or using BlueDragon, visit the self-help section of the New Atlanta web site for assistance:

http://www.newatlanta.com/products/bluedragon/self_help/index.cfm

Details regarding paid support options, including online-, telephone-, and pager-based support are available from the New Atlanta web site:

http://www.newatlanta.com/support/index.jsp

1.5 Other Documentation

The other relevant manuals available in the BlueDragon documentation library are:

- *Deploying CFML on ASP.NET and the Microsoft .NET Framework*
- *BlueDragon 9.0 CFML Compatibility Guide*
- *BlueDragon 9.0 CFML Enhancements Guide*
- *BlueDragon 9.0 User Guide*

Each of these documents offers useful information that may be relevant to developers, installers, and administrators using BlueDragon.NET. These are offered in PDF format in the `docs` directory where BlueDragon is installed (as discussed in *Deploying CFML on ASP.NET and the Microsoft .NET Framework*.).

All BlueDragon documents are available from New Atlanta's web site:

http://www.newatlanta.com/products/bluedragon/self_help/docs/index.cfm

2 Deploying CFML on .NET

This document supplements the information presented in the manual, *Deploying CFML on ASP.NET and the Microsoft .NET Framework*, which covers several very important points about the process of installing, managing, and troubleshooting the deployment of CFML on .NET.

Additionally, that document discusses the BlueDragon Admin Console and various configuration issues. Finally, it explains many details about the .NET Framework and offers links to many resources to assist CFML developers in making the transition to .NET. Those points are not repeated here.

Instead, this document focuses solely on the coding techniques used to integrate CFML with ASP.NET and the .NET Framework.

2.1 Benefits of Deploying CFML on .NET

The manual, *Deploying CFML on ASP.NET and the Microsoft .NET Framework*, also describes in considerable detail the many benefits of deploying CFML on .NET:

- Benefits from CFML/ASP.NET Integration
- Benefits from Running CFML on .NET Without Code Changes
- Benefits from Learning and Using ASP.NET Additional Features
- Benefits Enabled in .NET 4.0
- Benefits In Learning ASP.NET and .NET At Your Own Pace

Again, the details presented there will not be repeated here, but please read the other manual to learn more about these many benefits.

3 Sharing Variables between CFML and ASP.NET

With BlueDragon.NET, it's now possible for CFML and ASP.NET pages to share variables set in the session, application, client, request, and variables (local) scopes. The examples that follow demonstrate the means to set and get these kinds of variables in both CFML and ASP.NET.

Note that some (though not all) of the ASP.NET pages demonstrated below require use of New Atlanta-provided extensions which are enabled by "inheriting" a special cfmPage class. This is discussed in more detail in section 5.1.

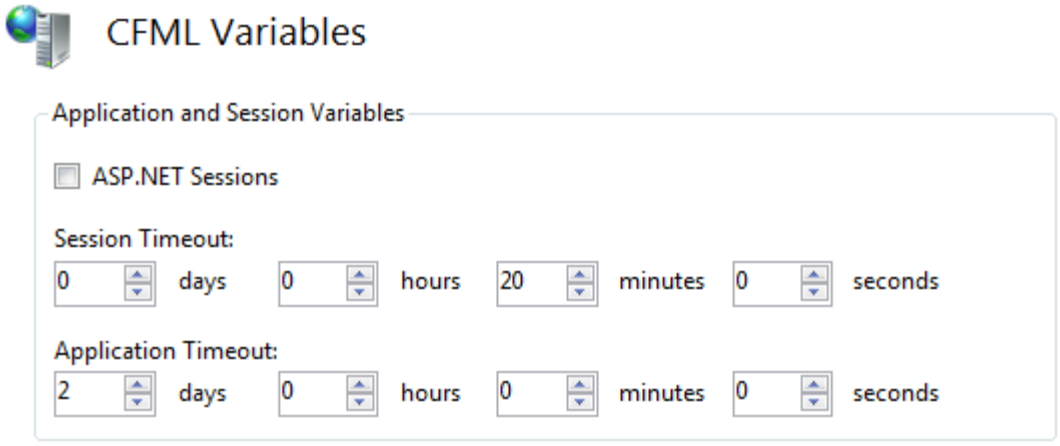
3.1 Sharing Session Variables between CFML and ASP.NET

The following examples demonstrate reading and writing session variables in ASP.NET. Note that, as in CFML, variable names are case-insensitive when using CFML scopes in ASP.NET.

Be aware also that the code in ASP.NET will differ if the CFML page with which you're integrating is using an Application name (a value for the `NAME` attribute on `CFAPPLICATION`.)

3.1.1 Setting Support for ASP.NET Sessions

In the case of session variables, it is necessary to make one configuration change in the BlueDragon admin console, in order to share session variables between CFML and ASP.NET. By selecting the CFML Variables applet in the IIS inetmgr you'll be presented a page with a dialogue that begins similarly to the following screenshot:



Note the available "ASP.NET Sessions" option. By setting this to `yes` (the default is `no`), you are telling BlueDragon to give control of session management to ASP.NET. As such, session variables set in CFML will really be managed by ASP.NET. This enables both the sharing of session variables with ASP.NET as well as the persistent sessions feature discussed in *Deploying CFML on ASP.NET and the Microsoft .NET Framework*.

3.1.2 Sharing Session Variables When CFAPPLICATION Has No NAME

When the `CFAPPLICATION` tag controlling a CFML page has no `NAME` attribute, then the ASP.NET code referring to it can refer to the session as a normal session scope in ASP.NET.

3.1.2.1 Sharing Unnamed CFML Sessions with ASP.NET

Following is a CFML page that sets a session variable when its CFAPPLICATION has no NAME:

```
<CFAPPLICATION SESSIONMANAGEMENT="yes">
<CFSET session.test = "somevalue">
```

The following ASP.NET example accesses that session variable:

```
<%@ Page language="c#" %>
Value of session variable "test" is: <%= Session["test"] %>
```

Again, note that the ASP.NET sessions feature must be enabled in the BlueDragon admin console.

A VB.NET version of the example above would be as follows (primarily changing the use of brackets to parentheses):

```
<%@ Page language="vb" %>
Value of session variable "test" is: <%= Session("test") %>
```

3.1.2.2 Sharing Unnamed ASP.NET Sessions with CFML

Conversely, the following page will set a session variable in the ASP.NET page, which could be accessed within CFML on a page that has no `NAME` in its `CFAPPLICATION`:

```
<%@ Page language="c#" %>
<% Session.Add( "test", "foo" ); %>
```

And the following CFML page accesses that session variable:

```
<CFAPPLICATION SESSIONMANAGEMENT="yes">
<CFOUTPUT>#session.test#</CFOUTPUT>
```

The only variation in a VB.NET version of the example above (other than changing the `Language` attribute) would be the removal of the closing semi-colon at the end of the script line.

3.1.2.3 Sharing Complex Data In Session Variables

There are no limitations in the data types that may be shared in the session scope. Any CFML variable may be shared between CFML and ASP.NET, including simple values like strings, numbers, and Booleans, as well as structures, arrays, and query result sets. For additional information, see section 3.6.

3.1.3 Sharing Session Variables When CFAPPLICATION Has a NAME

When the CFAPPLICATION tag controlling a CFML page has a NAME attribute, then the ASP.NET code referring to it will refer to it as a Hashtable within the ASP.NET session scope having that application name.

3.1.3.1 Sharing Named CFML Sessions with ASP.NET

Following is a CFML page that sets a session variable when its CFAPPLICATION has a NAME:

```
<CFAPPLICATION SESSIONMANAGEMENT="yes" NAME="SomeApp">
<CFSET session.test = "somevalue">
```

The ASP.NET code to access that session variable follows. Note the line declaring a variable referencing the session variables as a Hashtable within the ASP.NET session scope:

```
<%@ Page Language="c#"%>
<%
Hashtable cfmSessionScope = (Hashtable) Session["SomeApp"]; %>

Value of session variable "test", set in application "SomeApp"
is: <%= cfmSessionScope["test"] %>
```

A VB.NET version of the example follows:

```
<%@ Page Language="vb"%>
<%
DIM cfmSessionScope as Hashtable = Session( "SomeApp" )
%>
Value of session variable "test", set in application <%=
cfmSessionScope( "test" ) %>
```

3.1.3.2 Sharing Named ASP.NET Sessions with CFML

Conversely, the following page will set a session variable in the ASP.NET page, which could be accessed within CFML on a page that has a NAME of "SomeApp" in its CFAPPLICATION:

```
<%@ Page language="c#" %>
<%
// if there is no current session variable "scope"
// (as set in CFML) for the given appname, create one
if (Session["SomeApp"] == null) {
      Session.Add( "SomeApp", new Hashtable());
}
Hashtable cfmSessionScope = (Hashtable) Session["SomeApp"];
cfmSessionScope.Add( "test", "somevalue" );  %>
```

And the following CFML page accesses that session variable:

```
<CFAPPLICATION SESSIONMANAGEMENT="yes" NAME="SomeApp">
<CFOUTPUT>#session.test#</CFOUTPUT>
```

A VB.NET version of the example follows:

```
<%@ Page language="vb" %>
<%
' if there is no current application variable "scope"
' (as set in CFML) for the given appname, create one

if Application("SomeApp") is nothing Then
    Application.Item("SomeApp") = New Hashtable
End If

dim cfmApplicationScope as Hashtable =
CType(Application("SomeApp"), Hashtable)

' don't use Add method, as that does not overwrite
' key if it exists

cfmApplicationScope("test") = "somevalue"
%>
```

3.2 Sharing Application Variables Between CFML and ASP.NET

Application variables can also be set and shared in either CFML or ASP.NET pages. Indeed, in the examples above simply change any reference to “session” to instead refer to “application”, in both the CFML and ASP.NET pages (and the distinction of named and unnamed applications also applies).

3.3 Sharing Request Variables Between CFML and ASP.NET

Request scope variables can be shared between CFML and ASP.NET pages, however a difference from session and application variables is that request variables live only for the life of a single request.

In order to access a request scope variable set in a CFML page, within an ASP.NET page, the ASP.NET page must be processed as a part of the CFML page request, such as by way of a `CFINCLUDE` or `CFFORWARD` to an ASP.NET page, or the use of CFML code inline on an ASP.NET page. For more details on such integration of ASP.NET pages within CFML, see section 4.

3.3.1 Sharing CFML Request Scope with ASP.NET

The following CFML page shows setting a request scope variable and then including an ASP.NET page which accesses it:

```
<cfset request.test="somevalue">
<cfinclude page="get-request.aspx">
```

And the following page shows referring to that request scope variable in the included ASP.NET page. Note that the equivalent for the CFML request scope in ASP.Net is the built-in `Context` object and its `Items` array:

```
<%@ Page Language="c#" %>
<% if (Context.Items["test"] != null ) {
      Response.Write(Context.Items["test"]);
   }
%>
```

A simpler example of referring to the request variable would be:

```
<%= Context.Items["test"] %>
```

As discussed later in this section, there is also an available `CfmRequest` array that could be used as a convenience as an alternative to the Context.`Items` array.

A VB.NET version of the example follows:

```
<%    if Not Context.Items("test") is nothing Then
            Response.Write(Context.Items("test"))
      end if
%>
```

3.3.2 Sharing ASP.NET Context and Request Scope with CFML

Conversely, the following example shows setting a variable in ASP.NET that could be accessed in the request scope of a CFML page, then including a CFML page to process it (again, including pages between CFML and ASP.NET is discussed further in section 4.1.3, and the use of the Inherits attribute on the Page directive to call the `CfmPage` class is discussed in section 5.1):

```
<%@   Page Language="c#"
            Inherits="NewAtlanta.BlueDragon.CfmPage"%>

<%@ Register TagPrefix="cf" Namespace="NewAtlanta.BlueDragon"
      Assembly="NewAtlanta.BlueDragon" %>
```

```
<% Context.Items["setinasp"] = "somevalue"; %>
<cf:include template="get-request.cfm" runat="server"/>
```

And the CFML page included to process the request variable is:

```
<cfoutput>#request.setinasp#</cfoutput>
```

The only variation in a VB.NET version of the example above (other than changing the `Language` attribute) would be the removal of the closing semi-colon at the end of the script line.

While ASP.NET does have a notion of a `Request` object in the standard ASP.NET `System.Web.UI.Page` class, it is read-only; the following C# code will result in an ASP.NET runtime error:

```
Request[ "answer" ] = 42;
```

Instead, the available `Items` property of the `Context` object can be used to share variables in the CFML "request" scope. As a convenience, the `CfmPage` class provides the `CfmRequest` property, which is a simple wrapper around Context.Items. Thus, the following two lines of C# code are equivalent:

```
Context.Items[ "answer" ] = 42;

CfmRequest[ "answer" ] = 42;
```

Keys for the `Context.Items` property are case-sensitive; in order to insure error-free variable sharing with CFML pages, you should always use lowercase keys when setting or getting variables from the Context.Items or CfmRequest properties. (This is the only scope for which keys are case-sensitive).

3.4 Sharing Client Variables Between CFML and ASP.NET

Continuing along the lines of the previous discussion, the `Client` property of the `CfmPage` class (see section 5.1) provides access to the CFML `client` scope. The `Client` property is a subclass of the `System.Collections.Hashtable` class of the .NET Framework. Keys to the `Client` property are not case-sensitive. You must enable `Client` scope variables via the `CLIENTMANAGEMENT` attribute of the `CFAPPLICATION` tag:

```
<cfapplication sessionmanagement="true"
      clientmanagement="true">

<cfset client.browser = cgi.http_user_agent>
```

And the ASP.NET code to refer to that would be:

```
<%@ Page Language="c#"
      Inherits="NewAtlanta.BlueDragon.CfmPage"  %>
```

```
<%= Client[ "browser" ] %>
```

The only variation in a VB.NET version of the example above (other than changing the `Language` attribute) would be the use of parentheses rather than brackets.

As with CFML, you may not put complex variables (structs, arrays, or queries) into the Client scope.

3.5 Sharing Local (Variables Scope) Variables Between CFML and ASP.NET

If you will be running CFML code within an ASP.NET page, such as with `<cf:inline>` or `<cf:inline>`, that creates local variables (as would be set in the `variables` scope in CFML), you can access that data in ASP.NET using the available `Variables` object provided by the `CfmPage` class (see section 5.1 for more information on the `CfmPage` class).

The following shows the execution of a CFDIRECTORY tag on an ASP.NET page, using the `cf:inline` tags discussed in section 4.6.1. Notice how the results, which would normally be available in CFML as a `getdir` variable (or more formally, `variables.getdir`), are instead shown to be passed to and processed by an ASP.NET datagrid:

```
<%@ Page Language="c#"
      Inherits="NewAtlanta.BlueDragon.CfmPage" %>

<%@ Register TagPrefix="cf" Namespace="NewAtlanta.BlueDragon"
      Assembly="NewAtlanta.BlueDragon" %>

<cf:inline runat="server">
      <cfdirectory action="LIST" directory="." name="getdir">
</cf:inline>
<%
  grid.DataSource = Variables[ "getdir" ];
  grid.DataBind();
%>
<asp:DataGrid runat="server" id="grid"/>
```

The only variation in a VB.NET version of the example above (other than changing the `Language` attribute) would be to change the script code to remove the semi-colons and change the use of brackets to parentheses:

```
grid.DataSource = Variables( "getdir" )
grid.DataBind()
```

3.5.1 Additional Features of the Variables Object

Further, this `Variables` object is implemented similar to a structure in CFML, or what's called a `System.Collections.Hashtable` in the .NET Framework. Being a Hashtable in that it has some extra properties and methods that can be useful (such as the `count()` method below), but like traditional CFML variables, the variable names (the HashMap keys) are case-insensitive; therefore, in the following example, only one first name is created. Referring to either form leads to the same result (Dick, the last value set), and the value of `Variables.Count` equals `1`.

```
<%@   Page Language="c#"
           Inherits="NewAtlanta.BlueDragon.CfmPage" %>

<%@ Register TagPrefix="cf" Namespace="NewAtlanta.BlueDragon"
      Assembly="NewAtlanta.BlueDragon" %>

<%    Variables[ "firstname" ] = "Tom";
      Variables[ "FirstName" ] = "Dick"; %>

<p>Variables.Count = <%= Variables.Count %>
<p>Variables[ "firstname" ] =
      <%= Variables[ "firstname" ] %>
<cf:inline runat="server">
      <p>FirstName = <cfoutput>#FirstName#</cfoutput>
</cf:inline>
```

3.6 Sharing Complex CFML Variables

When sharing CFML variables with ASP.NET, whether in the session, application, or request scope, it's possible to share even complex variables (such as structures, arrays, and even query result sets). This section shows the ASP.NET code used both to read and write complex variables to be shared with CFML.

3.6.1 Sharing Arrays and ILists

CFML arrays can be accessed in ASP.NET as variables of the type System.Collections.IList. Note that CFML array indexes are 1-based, where IList array indexes are 0-based.

3.6.1.1 Sharing CFML Array Data with ASP.NET

Following is an example of creating an array in CFML:

```
<cfapplication sessionmanagement="Yes">
<cfset myArray = ArrayNew( 1 )>
<cfset myArray[ 1 ] = "one">
<cfset myArray[ 2 ] = "two">
<cfset myArray[ 3 ] = "three">
```

```
<cfset session.myArray=myArray>
```

Following is an example how to process that in ASP.NET:

```
<%@ Page Language="c#" %>
<%
      IList myArray = (IList) Session["myArray"];

      for (int  i = 0 ; i < myArray.Count ; i++) {
          Response.Write( i );
          Response.Write( " : " );
          Response.Write( myArray[ i ] );
          Response.Write( "<br>" );
      }%>
```

A VB.NET version of the example above would be as follows:

```
<%@ Page language="vb" %>
<%
  DIM MyArray as IList = Session( "myArray" )
  DIM i as Integer
  For i = 0 To MyArray.Count - 1
    Response.Write( i )
    Response.Write( " : " )
    Response.Write( MyArray( i ) )
    Response.Write( "<br>" )
  Next i
%>
```

3.6.1.2 Sharing ASP.NET IList and Array Data with CFML

Conversely, the following example shows creating an array in the session scope in ASP.NET that could be accessed by a CFML page. Note that while one could create an IList to hold array-oriented data to be shared with CFML, the resulting array when processed in CFML would be read-only. (Attempts to modify values in CFML will be ignored.)

In order to create a "normal" array for sharing with CFML, use the `ArrayNew()` method, available in the `CfmPage` class, which returns an object of class `cfArrayListData`, which implements the `System.Colletions.IList` interface. An example would be:

```
<%@ Page Language="c#" inherits="NewAtlanta.BlueDragon.CfmPage"
%>

<%    IList myArray = ArrayNew(1);
```

```
      myArray.Add( "value1" );
      myArray.Add( "value2" );
      Session.Add( "myASPArray", myArray );    %>
```

A VB.NET version of the example above would be:

```
<%
      DIM myArray as IList = ArrayNew( 1 )
      myArray.Add( "value1" )
      myArray.Add( "value2" )
      Session.Add( "myASPArray", myArray )
%>
```

And the CFML page to process the session array is:

```
<cfloop from="1" to="#arraylen(session.myASPArray)#" index="i">
      <cfoutput>#i# - #session.myASPArray[i]#<br></cfoutput>
</cfloop>
```

3.6.2 Sharing Structures and Hashtables

CFML structures can be accessed in ASP.NET as variables of the type System.Collections.Hashtable.

3.6.2.1 Sharing CFML Structures with ASP.NET

Following is an example of creating a structure in CFML:

```
<cfapplication sessionmanagement="Yes">
<cfset myStruct = structnew()>
<cfset myStruct.key1 = "one">
<cfset myStruct.key2 = "two">
<cfset session.myStruct=myStruct>
```

Following is an example how to process that in ASP.NET:

```
<%@ Page Language="c#" %>
<% if (Session["myStruct"] != null) {
      Hashtable myStruct = (Hashtable) Session["myStruct"];
      Response.Write("key1=" + myStruct["key1"] + "<br>");
      Response.Write("key2=" + myStruct["key2"]);
      }
%>
```

A VB.NET version of the example above would be:

```
<%@ Page Language="c#" %>
<% if not Session("myStruct") is nothing
           Response.Write("Value of session variable structure
'myStruct' is:<br>")
           DIM myStruct as Hashtable= Session("myStruct")
           Response.Write("key1=" + myStruct("key1") + "<br>")
           Response.Write("key2=" + myStruct("key2"))
%>
```

3.6.2.2 Sharing ASP.NET Hashtables with CFML

Conversely, the following example shows creating a structure in the session scope in ASP.NET that could be accessed by a CFML page. Note that, again, while one could create structure-oriented data by creating a new Hashtable, the resulting structure when processed in CFML would be read-only. (Attempts to modify values in CFML will be ignored.)

In order to create a "normal" structure for sharing with CFML, you can use the `StructNew()` method, available in the `CfmPage` class, which returns a `cfJSharpStructData` object, which is a subclass of `System.Collections.Hashtable`. The following example also shows two different ways to refer to the keys in the resulting Hashtable, using either the `Add()` method of the Hashtable or directly assigning the session key:

```
<%@ Page Language="c"
     Inherits="NewAtlanta.BlueDragon.CfmPage" %>
<%
     Hashtable myStruct = StructNew();
     myStruct.Add( "key1", "value1" );
     myStruct["key2"] = "value2" ;
     Session.Add( "myASPStruct", myStruct );

%>
```

A VB.NET version of the example above would be:

```
<%@ Page language="vb" Inherits="NewAtlanta.BlueDragon.CfmPage"%>
<%
     DIM myStruct as Hashtable = new Hashtable()
     myStruct.Add( "key1", "value1" )
     myStruct("key2") = "value2"
     Session.Add( "myASPStruct", myStruct )
 %>
```

And the CFML page to process the session structure is:

```
<cfloop collection="#session.myASPStruct#" item="i">
      <cfoutput>
      #i# - #session.myASPStruct[i]#<br>
      </cfoutput>
</cfloop>
```

3.6.3 Sharing Query Result Data

CFML Query resultsets (results of `CFQUERY`, `querynew()`, or other tags that return query results like `CFDIRECTORY` and others) can be accessed in ASP.NET as variables of the type System.Data.DataTable.

3.6.3.1 Sharing CFML Queries with ASP.NET

Following is an example of creating a query result in CFML:

```
<cfapplication sessionmanagement="Yes">
<cfquery datasource=”somedsn” name=”session.myQuery”>
      Select name, city from SomeTable
</cfquery>
```

Notice that the NAME attribute of CFQUERY has been set to store the query results in a session variable named myQuery. This is legal in CFML.

Of course, the preceding example presumes you have a table and database with some expected data to be processed by the corresponding ASP.NET page.

The following example generates a CFML query dynamically, with the same result as would be obtained from a CFQUERY:

```
<cfapplication sessionmanagement="Yes">
<cfscript>
myQuery = querynew("name,city");
queryaddrow(myQuery,1);
querysetcell(myQuery,"name","W.A. Mozart",1);
querysetcell(myQuery,"city","Salzburg",1);

queryaddrow(myQuery,1);
querysetcell(myQuery,"name","Nikolai Rimsky-Korsakov",2);
querysetcell(myQuery,"city","Tikhvin",2);

session.myQuery = myQuery;
</cfscript>
```

Following is an example of how to process this query resultset in ASP.NET. Note that a CFQUERY result (or that created by QueryNew and its related functions) can be treated as an

ASP.NET Datatable, and therefore can be processed the same way, including such powerful features as binding the query result to an ASP.NET Datagrid:

```
<%@ Page Language="c#" %>
<%    MyDataGrid.DataSource = Session["myQuery"];
      MyDataGrid.DataBind(); %>

<form runat="server">
      <asp:datagrid id="MyDataGrid" runat="server"/>
</form>
```

Of course, it's also possible to share a `CFQUERY` result as returned from a CFC and process it in a datagrid this same way. See section 4.1.1 for more information and examples. Similarly, a `CFQUERY` performed within cf-inline processing on an ASP.NET page, such as that shown in section 4.6, can be later processed by accessing the variables scope in which it's placed, as discussed in section 3.5.

3.6.3.2 Sharing ASP.NET Queries with CFML

Conversely, if .NET code returns a query result set in the form of a native .NET datatable, that can be processed in CFML as well, treated just as if it was a CFQUERY result. See the demonstration in section 4.1.2.1 which shows calling a .NET business object that returns a .NET datatable, and shows processing that result as a CFQUERY result set.

4 Interaction Between CFML and ASP.NET Pages

Simply being able to run your CFML on the .NET framework may be enough motivation for using BlueDragon.NET, but CFML and ASP.NET developers should also note that it enables tight integration between CFML with ASP.NET and other .NET components.

This section will discuss how it's possible for CFML pages to leverage .NET components and work done in ASP.NET pages, and how ASP.NET pages can be modified to leverage CFML components and the work done in CFML pages. Section 3 discusses sharing variables between the two environments.

Note that some (though not all) of the ASP.NET pages demonstrated below require use of New Atlanta-provided extensions which are enabled by "inheriting" a special `cfmPage` class. This is discussed in more detail in section 5.1.

4.1 Invoking Components and Objects between CFML and ASP.NET Pages

With BlueDragon.NET, it's now possible for CFML and ASP.NET pages to call upon objects created in the other environment. CFML pages can invoke .NET objects, and ASP.NET pages can invoke CFML Components (CFCs), enabling developers in either environment to leverage the work already created in the other.

4.1.1 Invoking CFML Components (CFCs) within ASP.NET

An ASP.NET page can invoke a CFC by way of a pair of methods, `CreateComponent()` and `InvokeCfc()`.

The following shows a CFC called `employees.cfc`, with an available `getemployees()` method, which returns a CFML query result set:

```
<cfcomponent>
      <cffunction name="getemployees" returntype="query">
        <cfquery datasource="cfsnippets" name="employees">
        SELECT * FROM Employees
        </cfquery>
        <cfreturn employees>
      </cffunction>
</cfcomponent>
```

For the sake of demonstration (and for those who may not be familiar with invoking CFCs in CFML), the following code shows one way to invoke a CFC from CFML:

```
<cfinvoke component="employees" method="getemployees"
returnvariable="getemp">
```

```
<cfoutput query="getemp">
      #firstname# #lastname#<br>
</cfoutput>
```

The following two subsections demonstrate invoking the CFC from ASP.NET.

4.1.1.1 Invoking a CFC Using the CreateComponent() Method

`CreateComponent()` takes a CFC name as its argument and returns an instance of the named CFC, returning a `CfComponent` object, which has an available `Invoke()` method to call a given CFC method.

The following code shows the invocation of that CFC using `CreateComponent()`.In this example, although the result returned from the CFC is a query resultset, notice that BlueDragon marshals it into a result that can be handed to the ASP.NET DataGrid:

```
<%@   Page Language="c#"
            Inherits="NewAtlanta.BlueDragon.CfmPage" %>
<%@   Register TagPrefix="cf" Namespace="NewAtlanta.BlueDragon"
            Assembly="NewAtlanta.BlueDragon" %>

<%    CfComponent Employees =
            CreateComponent( "employees" );
      grid.DataSource =
            Employees.Invoke( "getemployees" );
      grid.DataBind();
%>
<asp:DataGrid runat="server" id="grid"
            BackColor="#eeeeee" Width="85%" Font-Size="10pt"
            HorizontalAlign="Center" Font-Name="Verdana">
      <HeaderStyle BackColor="Black" ForeColor="White"
            Font-Bold="True" HorizontalAlign="Center" />
      <AlternatingItemStyle BackColor="White" />
</asp:datagrid>
```

For those who may prefer to see the .NET script portion in VB.NET instead of C#, the following could be substituted above:

```
<% DIM Employees AS CfComponent = CreateComponent( "employees" )
   grid.DataSource = Employees.Invoke( "getemployees" )
   grid.DataBind()
%>
```

Of course, the first line of the ASP.NET page must also be changed to indicate that VB.NET code is being used:

```
<%@ Page Language="vb" Inherits="NewAtlanta.BlueDragon.CfmPage"%>
```

See section 4.1.1.3 for information on passing arguments to CFC methods.

For more information on this example's use of "inheriting" the `cfmPage` class, see section 5.1.

While this example demonstrates using inline scripting, this and all BlueDragon ASP.NET extensions can be done using code-behind as well. See section 5.2 for more information.

Finally, see section 3.6 for more information on how to leverage complex CFML data types within ASP.NET.

4.1.1.2 Alternative: Invoking a CFC Using the InvokeCfc() Method

An alternative way to invoke a CFC is with the `InvokeCfc()` method, which allows you to name the CFC and method (and any arguments, as discussed in section 4.1.1.3) in a single step, like the `CFINVOKE` tag. This would also be the preferred approach when using the code-behind approach to ASP.NET page development. An example using inline scripting would be:

```
<%@ Page Language="c#"
      Inherits="NewAtlanta.BlueDragon.CfmPage" %>

<% grid.DataSource = InvokeCfc( "employees", "getemployees" );
   grid.DataBind();
%>
```

Note that the `InvokeCfc()` method does not require the page to register the `NewAtlanta.BlueDragon` library as in the previous example. In this example, the result is bound to a datagrid, such as that shown in the previous example.

4.1.1.3 Passing Arguments to CFC Methods

Both the `Invoke()` method of `CreateComponent()` and the `InvokeCfc()` method can accept positional arguments to be passed to the CFC method. Using the example in the previous section, arguments could have been passed to the GetEmployees method as:

```
InvokeCfc( "employees", "getemployees", arg1, arg2 [,argn] );
```

Optional arguments can only be omitted from the end of the list. In order for arg1 to be optional, arg2 and arg3 must be optional also; in order for arg2 to be optional, arg3 must be optional also.

4.1.1.4 Passing a Named TextWriter

Both the `CreateComponent()` and the `InvokeCfc()` methods default to writing to the current ASP.NET page (its default TextWriter). In advanced cases, you may want to have greater con-

trol, such as when any output from the CFC needs to be buffered and/or processed before being sent to the client.

You can override this default behavior with another form of the methods, passing a TextWriter as the first argument before the CFC name, as in:

```
CreateComponent( TextWriter, componentName );
```

or

```
InvokeCfc( TextWriter, componentName, methodName, arguments);
```

4.1.2 Invoking .NET Components within CFML

Just as an ASP.NET page can leverage CFML components, so also can a CFML page leverage .NET objects. Using either `CFOBJECT` and `createObject()`, with a new `type` value of `.net`, a CFML page can instantiate and then manipulate the object's properties and methods.. (Use of the `type` value `java` is maintained for backward compatibility but is simply a synonym for the `.net` value.) You can use this feature to call built-in classes/objects of the .NET Framework as well as those you (or third parties) may write.

Examples of each are described in the following sections. (Be aware that in general, calls to methods and properties using `CFOBJECT/createObject` will be case-sensitive.)

4.1.2.1 Calling a .NET Business Object

Perhaps the most compelling example of calling .NET objects would be in calling a .NET business object that you or your organization may write. A powerful benefit of BlueDragon is that, if this object returns a .NET datatable, BlueDragon will marshal the result so that it can be treated in CFML just as if it was a CFQUERY resultset. This section demonstrates an example of this.

Consider the following C# program, called `Employees.cs`, which define a class called `Employees` in a namespace called `Mycomponents`, which has a method `getEmployees()` that returns a .NET datatable:

```
using System;
using System.Data;
using System.Data.Odbc;

namespace myComponents {
public class Employees {
      public System.Data.DataTable GetEmployees() {
            OdbcConnection conn = new
OdbcConnection("DSN=cfsnippets");
            conn.Open();
            String sql = "SELECT FirstName, LastName from
Employees";
```

```
            OdbcDataAdapter adapter = new OdbcDataAdapter(sql,
conn);
            DataTable dt = new DataTable();
            adapter.Fill(dt);
            return dt;
            }
```

This example uses ODBC and assumes the CFSnippets datasource (which comes with ColdFusion) exists, but it can easily be changed to support any driver, datasource, and table. This code can be compiled using Visual C# using the following command:

```
csc /t:library Employees.cs
```

This will create a managed assembly called `Employees.dll`, which can then be placed either in the `/bin` directory of a web application or the `Global Assembly Cache`.

This object can then be called from CFML using `CFOBJECT` (or `createObject`), and its result can be processed just as if it was a CFQUERY:

```
<cfobject type=".net" class="myComponents.Employees"
name="EmpClass" action="CREATE" >
<cfset emps=EmpClass.GetEmployees()>
<cfoutput query="emps">
      #NameFirst# #Namelast#<br>
</cfoutput>
```

Finally, note that if the .NET object had instead returned a DataSet (rather than a DataTable), the following CFML could be used for the lines after the CFOBJECT call above, naming the name of the database table within the returned dataset:

```
<cfset emps=emps.get_tables().get_Item("employees")>
<cfoutput query="emps">
      #FirstName# #LastName#<br>
</cfoutput>
```

This opens the door to powerful integration of CFML with .NET business objects. Consider, as well, the option in CFML to create a CFComponent (CFC) which this example could call instead, and it can freely be changed from doing a CFQUERY today to calling such a .NET business object tomorrow.

4.1.2.2 Calling a Built-in .NET Framework Class, Example: Accessing Configuration Settings

The following example shows using a built-in .NET Framework class/object. .NET supports a notion of application configuration settings that can be configured in an XML file (the

`web.config` discussed in *Deploying CFML on ASP.NET and the Microsoft .NET Framework*). If you choose to extend that file to support .NET application configuration settings, this section shows how you can access them from within CFML.

Assume we had a `web.config` file that started with the following XML entries:

```
<?xml version="1.0" encoding="utf-8"  ?>
<configuration>
     <!-- settings to be read in CFML via CFOBJECT -->
     <appSettings>
          <add key="somekey" value="somevalue" />
     </appSettings>
```

This would define a key called `somekey` with a value of `somevalue`. Following is the code that could be used to access this value from within CFML:

```
<cfobject action="CREATE" type=".net"
class="System.Configuration.ConfigurationSettings"
name="settings">

<cfoutput>#settings.get_AppSettings().Get("somekey")#</cfoutput>
```

Or this could have been done using CFSCRIPT instead:

```
<cfscript>
settings=createobject(".net","System.Configuration.ConfigurationS
ettings");
writeoutput(settings.get_AppSettings().Get("somekey"));
</cfscript>
```

Beware of wrapping in the lines above. The line starting with `settings=` should all be on one line, and the `writeoutput` on another line.

Either of these examples will output the value of the `somekey` key, showing the value of `somevalue`.

(Some may find this notion of application settings to be a useful alternative to `application` variables, or an alternative way to load values which are then assigned to `application` variables.)

Notice also in both examples the use of `get_AppSettings()` to retrieve the `AppSettings` property. This nomenclature reflects the fact that Visual J# underlies BlueDragon.NET. See the section 5.3 for more information.

Note, as well, that any time you change the web.config file the .NET web app restarts (as explained in *Deploying CFML on ASP.NET and the Microsoft .NET Framework*), which for many is a little-known .NET behavior. If you may be frequently changing the `appSettings` values in the `web.config` file, you'll want to look at the `file` option on `appSettings` which allows you to point to another file which holds the settings. See this for more:

http://msdn.microsoft.com/en-us/library/aa903313.aspx

For still more information on the use of `appSettings`, including other ways to configure them to solve different problems, see the following:

http://www.codeproject.com/dotnet/EnhancedAppSettings.asp

You can learn more about all the built-in objects of the .NET framework in many available .NET resources. One reference is the following:

http://msdn.microsoft.com/en-us/library/aa719441.aspx

Finally, note that additional information on discovering and using the properties of a .NET class as well as converting a .NET example into a CFML equivalent is offered in section 5.6.

4.1.2.3 Invoking Your Own .NET Objects

To invoke your own objects or those written by others, you place the created DLLs in either the web application's BIN directory or in the Global Assembly Cache as appropriate to your needs, and as is standard for ASP.NET pages calling upon .NET objects. After that, the resulting .NET object can be called as discussed in the preceding sections.

4.1.2.4 Invoking Java Objects in .NET

If you're interested in accessing objects that you may have built or acquired which are written in Java (rather than C#), those must be rewritten and recompiled using C# in order to be called as .NET objects within the .NET framework.

After compiling your Java code into a .NET library, it can be processed as discussed in section 4.1.2.3.

4.1.3 Invoking COM Objects from CFML

The .NET Framework supports invocation of COM (Component Object Model) objects in two manners, using what are referred to as *late*- or *early*-binding. BlueDragon.NET supports COM objects via the more efficient early-binding process, which requires creation and use of a Runtime Callable Wrapper (RCW) for the COM object. This can be created easily within Visual Studio/.NET or via the command line (using the .NET `tlbimp.exe` program, available if you've installed Visual Studio/.NET or the .NET SDK). See information on Visual Studio, or the resources offered at the end of this section, for more information on creating an RCW class.

Once the RCW class for a COM object is created, its resulting `dll` file must be placed in the application's `/bin` directory or the Global Assembly Cache (see *Deploying CFML on ASP.NET and the Microsoft .NET Framework* for more information). The COM object must also be regis-

tered on the server (using `regsvr32.exe`). With these steps completed, the RCW DLL can be invoked like any .NET object using CFOBJECT or createObject (as described in section 4.1.2.1).

Be aware that the class name to invoke would incorporate both the wrapper name, the COM object, and a special `Class` phrase to be appended to that, so that the CFOBJECT for a COM object whose wrapper is `somewrapper.dll` and the COM object's name is `something`, would be:

```
<CFOBJECT TYPE=".net" NAME="somename"
CLASS="somewrapper.somethingClass" ACTION="create">
```

Again, the `Class` phrase appended at the end is necessary, so that if the COM object's class name was `myclass`, the `CLASS` attribute would be `somewrapper.myclassClass`.

Additional information on invoking COM objects in .NET is provided in many resources, such as:

http://www.csharpfriends.com/Articles/getArticle.aspx?articleID=43

4.2 Including Between CFML and ASP.NET Pages

With BlueDragon.NET, it's now possible for CFML and ASP.NET pages to include the output of each other.

4.2.1 Including ASP.NET Page Output within CFML

A CFML page can include the output of an ASP.NET page using either the `getpagecontext().include()` method (introduced in CFMX) or by way of `CFINCLUDE` using a `PAGE` attribute introduced in BlueDragon. As an example, the following CFML page would include an ASP.NET page named `somefile.aspx`:

```
<CFINCLUDE PAGE="somefile.aspx">
```

You can also use a relative path, such as `../somefile.aspx` to point to a file in the parent directory, or `somedir/somefile.aspx` to point to a file in a subdirectory of the current file (from which the `CFINCLUDE` is taking place.)

It's also important to understand the significance of root-relative paths in .NET and how they may represent a location other than the web site document root. See section 5.4 for more information.

Finally, although the `CFINCLUDE TEMPLATE` attribute (used to include CFML pages) can accept an absolute path, the `CFINCLUDE PAGE` attribute cannot.

Warning: if you use `CFINCLUDE TEMPLATE` instead when accessing an ASP.NET page, the source of the ASP.NET page will be included and sent to the browser.

4.2.1.1 Challenges Including Some ASPX Pages

In general, there is no limitation to what an included ASP.NET page can do when included into a CFML page. Be aware, however, that some ASP.NET controls may create HTML posting back to the included ASPX page, which could lead to unexpected results. For instance, if you include a page generating an ASP.NET calendar control, which by default has hyperlinks to move among months, the HTML generated by ASP.NET will return control (postback) to the ASP.NET page that was included, rather than the CFML page performing the include. The same is true of the ASP.NET datagrid.

This is not to say that you cannot include an ASP.NET page performing the ASP.NET calendar or datagrid control. The problem is only when that resulting generated HTML may have hyperlinks created by the .NET control (they don't always). Only in those cases where hyperlinks are created will the potential problem exist.

Indeed, one workaround for these challenges is that you can call the ASP.NET page (such as one rendering the Calendar or grid control) from within an `<iframe>` on the CFML page (you can just as well call a CFML page that includes the ASP.NET page from within the iframe). On browsers that support iframes, the postbacks would not leave the CFML page:

```
<iframe src="calendar.aspx" scrolling="No" frameborder="0">
Your browser doesn't support iframes
</iframe>
```

4.2.2 Including CFML Page Output within ASP.NET Pages

An ASP.NET page can include the output of a CFML page using either <cf:include> or the `CfInclude()` method. See the previous section for discussions about page locations and relative vs. root-relative paths.

4.2.2.1 Using the <cf:include> Control

Following is an example of an ASP.NET page including a CFML page named `somefile.cfm`:

```
<%@ Page Language="c#"
      Inherits="NewAtlanta.BlueDragon.CfmPage" %>

<%@ Register TagPrefix="cf" Namespace="NewAtlanta.BlueDragon"
      Assembly="NewAtlanta.BlueDragon" %>

<cf:include template="somefile.cfm" runat="server"/>
```

You can also use a relative path, such as `../somefile.cfm` to point to a file in the parent directory, or `somedir/somefile.cfm` to point to a file in a subdirectory of the current file (from which the `<cf:include>` is taking place.)

Again, it's important to understand the significance of root-relative paths in .NET and how they may represent a location other than the web site document root. See section 5.4 for more information.

For more information on this example's use of "inheriting" the `cfmPage` class, see section 5.1.

While this example demonstrates using inline scripting, this and all BlueDragon ASP.NET extensions can be done using code-behind as well. See section 5.2 for more information.

4.2.2.2 Alternative: Using the CfInclude() Method

As an alternative to `<cf:include>` , the `CfInclude()` method may be preferred in some instances, particularly when using the code-behind approach to ASP.NET page development. This method accepts a name of a CFML page as input and returns its output as the result. Following is an inline scripting example of including the same CFML code above using the `CfInclude()` method:

```
<%@ Page Language="c#"
      Inherits="NewAtlanta.BlueDragon.CfmPage" %>

<%= CfInclude("testquery.cfm") %>
```

Note that you must render this method within the `<%= %>` declaration in ASP.NET (rather than the `<% %>` declaration, because it renders output of HTML. (Similarly, if you were to perform the method in codebehind, you need to render it within a `Response.Write` method, as demonstrated in section 5.2).

Also, note that because you are not using a control as you were with `<cf:include>` , you do not need to register the use of the `NewAtlanta.BlueDragon` library.

The only variation in a VB.NET version of the example above would be the changing of the `Language` attribute. The script line would be identical.

4.3 Transferring Control Between CFML and ASP.NET Pages

With BlueDragon.NET, it's possible for CFML and ASP.NET pages to transfer control between each other. The examples that follow demonstrate the means to transfer control between CFML and ASP.NET.

4.3.1 Transferring Control from CFML to ASP.NET Pages

While a CFML page can transfer control to an ASP.NET page using `<CFLOCATION>`, it's also possible to perform a server-side transfer of control (or forward) to another page, whether a CFML or ASP.NET page. Server-side transfers of control were first introduced into CFML to include JSP pages in CFMX and BlueDragon's Java editions, using the CFML function, `getpagecontext().forward()`.

While you can continue to use that in BlueDragon to call ASP.NET pages (or JSP pages on the Java editions of BlueDragon), another option is the `CFFORWARD` tag which was introduced in

BlueDragon. The CFFORWARD tag takes the same `page` attribute, which has the same characteristics, as the CFINCLUDE tag discussed in section 4.2.1.

For additional information on the benefits and drawbacks of using server-side transfers of control, search for and review available documentation on the CFML `getpagecontext().forward()` function or the `Server.Transfer()` method available in ASP and ASP.NET (as discussed in the next section.)

4.3.2 Forwarding from ASP.NET to CFML Pages

While ASP.NET pages have an available `Server.Transfer()` method permitting forwards from one `.aspx` page to another, the feature does not seem to work for CFML pages. Similarly, it's not possible to use the `CFFORWARD` and `getpagecontext().forward()` methods within a `<cf:inline>` control.

It is possible, however, to transfer control from an ASP.NET page to a CFML page using the available `Response.Redirect()` method, which works just like a `CFLOCATION`. Following is an example that transfers control from an ASP.NET page to a CFML page:

```
<%@ Page Language="c#" %>

<% Response.Redirect("somefile.cfm");%>
```

The only variation in a VB.NET version of the example above (other than changing the `Language` attribute) would be the removal of the closing semi-colon at the end of the script line.

You can also use a relative path, such as `../somefile.cfm` to point to a file in the parent directory, or `somedir/somefile.cfm` to point to a file in a subdirectory of the current file (from which the `Redirect` is taking place.)

Again, it's important to understand the significance of root-relative paths in .NET and how they may represent a location other than the web site document root. See section 5.4 for more information.

4.4 Calling CFML Custom Tags Within an ASP.NET Page

With BlueDragon.NET, it's now possible for ASP.NET pages to call CFML custom tags, including both simple (single) custom tags and paired custom tags, using the `<cf:module>` control.

All the same concepts of using CFML custom tags are represented, including support for local or global custom tags by way of the `Name` and `Module` attributes (as in `CFMODULE`), support for paired tags and notions such as `CFASSOCIATE` and `thistag.ExecutionMode` used within the tag, the use of `AttributesCollection`, and more, as demonstrated below.

4.4.1 Calling a Simple Single CFML Custom Tag

Calling a CFML custom tag is done using the `Template` or `Name` attributes of `<cf:module>`, just like the CFML tag `<CFMODULE>`. An example is:

```
<%@   Page Language="c#"
            Inherits="NewAtlanta.BlueDragon.CfmPage" %>
<%@   Register TagPrefix="cf" Namespace="NewAtlanta.BlueDragon"
            Assembly="NewAtlanta.BlueDragon" %>

<cf:module template="myCustomTag.cfm" runat="server"/>
```

As in CFML, and similar to when using the more traditional "cf_" syntax for calling custom tags, when the TEMPLATE attribute is used, the current directory is searched first, and if not found then the global customtags location (and any additional custom tag paths defined in the Admin console) is searched.

The location of the global `customtags` directory depends on the installation alternative chosen, as discussed in *Deploying CFML on ASP.NET and the Microsoft .NET Framework*. If the single virtual directory option is chosen, then it will be the `bluedragon\customtags` directory within the virtual directory. If any of the other alternatives were chosen, then the global directory is in the main installation directory, such as C:`\BlueDragon.NET\customtags`.

As with CFMODULE, when specifying the NAME attribute, the `.cfm` file extension is not used and the global directory (not the local directory) is searched. An example is:

```
<%@   Page Language="c#"
            Inherits="NewAtlanta.BlueDragon.CfmPage" %>
<%@   Register TagPrefix="cf" Namespace="NewAtlanta.BlueDragon"
            Assembly="NewAtlanta.BlueDragon" %>

<cf:module name="myCustomTag" runat="server"/>
```

Note that unlike in CFML, ASP.NET requires closing tags for all custom controls, so in the examples above the custom tag is invoked only once, and in `start` mode, as if it was specified in CFML without a closing tag. See further discussion in section 4.4.4 regarding paired tags.

For more information on these examples' use of "inheriting" the `cfmPage` class, see section 5.1.

While this example demonstrates using inline scripting, this and all BlueDragon ASP.NET extensions can be done using code-behind as well. See section 5.2 for more information.

4.4.2 Passing Attributes to a CFML Custom Tag

While it is possible to pass attributes to a CFML custom tag, it is not possible to simply name the attributes on the `<cf:module>` control. Instead, you must use a combination of ASP.NET concepts which, together, achieve the desired result.

In order to be able to pass attributes to a custom tag, you must give the `<cf:module>` an `id` attribute, which is a standard .NET way to identify a control. Further, you use a newly available `Attributes` property for the `<cf:module>`, as identified with the `id` attribute. `Attributes`

is a .NET Hashtable, which works just like a CFML structure, and you simply add key-value pairs to the `Attributes` property before calling the custom tag. An example will demonstrate:

```
<%@   Page Language="c#"
           Inherits="NewAtlanta.BlueDragon.CfmPage" %>
<%@   Register TagPrefix="cf" Namespace="NewAtlanta.BlueDragon"
           Assembly="NewAtlanta.BlueDragon" %>

<%    MyTag1.Attributes[ "attr1" ] = "val1";
      MyTag1.Attributes[ "attr2" ] = "val2"; %>
<cf:module id=”MyTag1” template="myCustomTag.cfm"
      runat="server"/>
```

Note that even if you just want to pass in a single attribute-value pair, you still must use this combination of giving an `id` to the control, and passing the values to the `Attributes` property for the control using that `id`.

Indeed, the fact that the `Attributes` property is like a structure, in which multiple attribute-value pairs can be provided and are then all available within the custom tag, makes it operate just like the concept of the `AttributesCollection` in CFML. There's no need to specify that keyword, however.

4.4.3 Simulating Caller Scope Data When Calling a CFML Custom Tag in ASP.NET

If you wish to call a CFML custom tag from within ASP.NET that expects to have `caller` scope variables available (meaning variables existing in the local page scope of the caller which are then referenced within the custom tag), you can create those simply by creating `variables` (local) scope variables on the ASP.NET page before calling the custom tag.

As well, if the custom tag creates `caller` scope variables which then need to be processed in the calling ASP.NET page, you can access those using `variables` (local) scope variables on the ASP.NET page.

See section 3.5 for more information on creating and processing variables in the variables (local) scope in an ASP.NET page.

4.4.4 Calling Paired Custom Tags in ASP.NET

As in CFML, you can call a custom tag passing in some tag body between a pair of the tags (whether using the `TEMPLATE` or `NAME` attributes on `cf:module`). An example follows:

```
<cf:module template="myCustomTag.cfm" runat="server">
This is the tag body.
</cf:module>
```

As in CFML, the code within the custom tag will execute twice, the first time with `thistag.executionmode="start"` and the second time with `thistag.executionmode="end"`.

On the other hand, if there is no content between the custom tag controls, the custom tag is invoked only once, in `start` mode, as if it was specified in CFML without a closing tag. Both of these examples will be invoked only once, in `start` mode.

```
<cf:module template="myTag.cfm" runat="server"/>
<cf:module template="myTag.cfm" runat="server"></cf:module>
```

4.4.5 Nesting Custom Tags

Finally, as in CFML, it is permissible to nest custom tag calls, as this example demonstrates:

```
<cf:module name="myCustomTag" runat="server">
  <p>Within tag body before loop
  <cf:module name="exitloop" runat="server"/>
  <p>Within tag body after loop
</cf:module>
```

4.5 Invoking CFX Custom Tags in .NET Languages

BlueDragon.NET, like the other editions of BlueDragon, supports the use of CFX custom tags. Indeed, it's possible to create CFX tags using any .NET programming language, including C# and Visual Basic.NET. It is not possible, however, to simply drop Java custom tags into a BlueDragon.NET deployment and expect to call them. They must be rewritten and recompiled using C#.

For more information on creating new CFXs, or supporting existing ones you may already have, see the section of this same name in the manual, *Deploying CFML on ASP.NET and the Microsoft .NET Framework*.

BlueDragon.NET also supports "native" CFX tags implemented in C++ (note that native C++ tags must be compiled separately for x86 and x64 architectures).

4.6 Executing CFML Code Inline Within an ASP.NET Page

An interesting possibility with BlueDragon.NET is that ASP.NET pages can include CFML code inline, within the ASP.NET page itself (or in a code-behind file), using the `<cf:inline>` tag pair or available `CfInline()` method.

4.6.1 Using the <cf:inline> Control

Following is a simple example of using `<cf:inline>`:

```
<%@ Page Language="c#"
      Inherits="NewAtlanta.BlueDragon.CfmPage" %>
```

```
<%@ Register TagPrefix="cf" Namespace="NewAtlanta.BlueDragon"
      Assembly="NewAtlanta.BlueDragon" %>

<cf:inline runat="server">
      <cfoutput>#now()#</cfoutput>
</cf:inline>
```

You're not limited to such simple tags and functions, however. You can run virtually any CFML within the inline block, including CFQUERY, CFINVOKE, and more. Indeed, the CFML within the block can refer to (or create) variables in any valid CFML scope. See the discussion in section 3.5 for more information and specific examples about creating and referring to CFML scopes within ASP.NET code using `<cf:inline>`.

For more information on this example's use of "inheriting" the `cfmPage` class, see section 5.1.

While this example demonstrates using inline code, this and all BlueDragon ASP.NET extensions can be done using code-behind as well. See section 5.2 for more information.

4.6.2 Evaluating Possible Use of CF Inline Scripting

Some .NET developers may find curious (or even balk at) the notion of including CFML code on an ASP.NET page, as it may compromise a preferred separation of concerns and language. Indeed, it's worth keeping in mind, the various options in BlueDragon.NET for reusing existing CFML code from ASP.NET pages, including invoking CFCs, calling custom tags, and including CFML pages, all discussed in the preceding sections.

Still, there may be value in using `<cf:inline>`. For instance, a particular ASP.NET construct may be more difficult for you to create than the simpler CFML equivalent you may know. Perhaps more useful to some, you could quickly turn an existing CFML page into an ASP.NET page by simply rename the file, and wrapping the CFML code inside of `<cf:inline>` tags.

Another reason to consider `<cf:inline>` may be to enable an existing CFML page to leverage some benefit available to .NET pages that's not readily available within CFML pages. An example is the page caching abilities in ASP.NET, enabled by the `<%@ OutputCache>` directive.

4.6.3 Alternative: Using the CfInline() Method

As an alternative to `<cf:inline>` , the `CfInline()` method may be preferred in some instances, particularly when using the code-behind approach to ASP.NET page development. This method accepts a string of CFML as input and returns its output as the result. Following is an example of the same CFML code above executed inline using the `CfInline()` method:

```
<%@ Page Language="c#"
      Inherits="NewAtlanta.BlueDragon.CfmPage" %>

<%= CfInline("<cfoutput>#now()#</cfoutput>") %>
```

Note that you must render this method within the <%= %> declaration in ASP.NET (rather than the <% %> declaration, because it renders output of HTML. (Similarly, if you were to perform the method in codebehind, you need to render it within a Response.Write method, as demonstrated in section 5.2).

Also, note that because you are not using a control as you were with <cf:inline> , you do not need to register the use of the NewAtlanta.BlueDragon library.

The only variation in a VB.NET version of the example above would be the changing of the Language attribute. The script line would be identical.

4.7 Generating CFML Output within an ASP.NET Page

Finally, the <cf:output> control and CfOutput() method allow you to embed any CFML expression in an ASP.NET page, including use of all CFML tags and functions, and rendering its result without need to use <CFOUTPUT> (as would be required with <cf:inline> or CfInline).

4.7.1 Using the <cf:output> Control

An examples of the <cf:output> control follows:

```
<%@ Page Language="c#"
      Inherits="NewAtlanta.BlueDragon.CfmPage" %>

<%@ Register TagPrefix="cf" Namespace="NewAtlanta.BlueDragon"
      Assembly="NewAtlanta.BlueDragon" %>

<cf:output runat="server>
      <p>It is now: #Now()#
      <p>The path to the current page is:
            #GetBaseTemplatePath()#
</cf:output>
```

This is very similar to the <cf:inline> control. The only difference is that you need not use CFOUTPUT tags to output variables or other expressions. Indeed, just as with the CFOUTPUT tag in CFML, it's even possible to execute other CFML tags within the <cf:output> control.

For more information on this example's use of "inheriting" the cfmPage class, see section 5.1.

While this example demonstrates using inline scripting, this and all BlueDragon ASP.NET extensions can be done using code-behind as well. See section 5.2 for more information.

4.7.2 Alternative: Using the CfOutput() Method

As an alternative to <cf:output> , the CfOutput() method may be preferred in some instances. This method accepts a string of CFML, again, without the requirement to use

CFOUTPUT tags to output variables or other expressions. It returns its output as the result. Following is an example of the same CFML code above output using the CfOutput() method:

```
<%@ Page Language="c#"
      Inherits="NewAtlanta.BlueDragon.CfmPage" %>

<%= CfOutput("#now()#") %>
```

Note that you must render this method within the <%= %> declaration in ASP.NET (rather than the <% %> declaration, because it renders output of HTML. (Similarly, if you were to perform the method in codebehind, you need to render it within a Response.Write method, as demonstrated in section 5.2).

Also, note that because you are not using a control as you were with <cf:output> , you do not need to register the use of the NewAtlanta.BlueDragon library.

The only variation in a VB.NET version of the example above would be the changing of the Language attribute. The script line would be identical.

5 Additional Information

Various additional topics apply to many of the previous sections or are of particular importance to consider, and they are presented here.

5.1 BlueDragon.CfmPage and NewAtlanta.BlueDragon

While some aspects of integration between CFML and ASP.NET don't require anything other than use of standard .NET controls and objects, some of the new features enabled in BlueDragon require use of New Atlanta-created extensions to .NET. These are enabled by way of either or both of these, `BlueDragon.CfmPage` class and `NewAtlanta.BlueDragon` library.

This section offers some brief technical background on these classes, while the subsequent sections show using them in example code.

5.1.1 BlueDragon.CfmPage Class

The `NewAtlanta.BlueDragon.CfmPage` class is implemented on an ASP.NET page by way of the `Page` directive (or in code-behind using the `inherits` directive), and it provides various "server controls" that extend the functionality of ASP.NET to integrate tightly with CFML. It also implements additional methods to facilitate variable sharing and integration with CFML code and components (such as custom tags and CFCs).

The available methods, described elsewhere in this document, are:

- `ArrayNew()` and `StructNew()`
- `CfInclude()`, `CfInline()` and `CfOutput()`
- `InvokeCfc()` and `CreateComponent()`

The CfmPage class inherits from the standard ASP.NET `System.Web.UI.Page` class, extending it to support the BlueDragon server controls, though it does not override any of the functionality of the Page class.

Note that inheriting from `NewAtlanta.BlueDragon.CfmPage` causes execution of any `Application.cfm` file before the ASP.NET page, though this can be disabled. See section 5.5 for more information.

Inheriting from `NewAtlanta.BlueDragon.CfmPage` also automatically exposes the CFML `Variables` and `Client` scopes, as discussed in section 3.4. The `Request`, `Session`, and `Application` scopes are automatically shared, without need to extend a page to inherit from `NewAtlanta.BlueDragon.CfmPage`, as is discussed in section 3.

Additionally, note that when an ASPX page inherits from the CfmPage class (whether on the ASPX page or via code-behind), it also will display BlueDragon's debugging output at the bottom of the request, assuming debugging output is enabled.

5.1.2 NewAtlanta.BlueDragon Library

The `NewAtlanta.BlueDragon` library is registered using the `Register` directive in an ASP.NET page and is required in order to use the following controls added by BlueDragon, described later in this document:

- `<cf:include>`
- `<cf:inline>`
- `<cf:module>`
- `<cf:output>`

The `cf` prefix has been chosen as a default in these examples, but any legal ASP.NET prefix string is permissible. The `NewAtlanta.BlueDragon` library can only be used in pages that inherit from the `NewAtlanta.BlueDragon.CfmPage` class.

5.2 Using Code-Behind with BlueDragon Enhancements

The various ASP.NET enhancements and extensions offered with BlueDragon have been demonstrated in this manual using inline scripting (where the ASP.NET code is embedded in the `.aspx` page), because CFML developers are more familiar with that style of coding. More experienced ASP.NET developers may prefer to use code-behind (also known as codebehind or code behind). Also, ASP.NET 2.0 introduces still another approach called code-beside.

It is certainly possible to use these various BlueDragon extensions using code-behind (or code-beside). Any of the methods listed in section 5.1.1 may be used. Following is VB/.NET code for including a CFML page (similar to the examples shown in section 4.2.2), and this file will then itself be inherited from a sample ASP.NET page.

```
Public Class test
      Inherits NewAtlanta.BlueDragon.CfmPage
      Sub Page_Load
            Response.Write(CfInclude("somefile.cfm"))
      End Sub
End Class
```

Assuming the file above was named `cfinclude-method.vb`, and as you may notice the class name there was `test`, the ASP.NET that uses this is as follows:

```
<%@ Page Language="vb"
            inherits="test" Src="cfinclude-method.vb"%>
<html>
<head>
      <title>Include of CFML page</title>
```

```
</head>
<body>
</body>
</html>
```

An advantage of using the SRC attribute in the Page directive (the first line of the ASP.NET page above) is that you don't even need to compile the VB program first. Running the page will cause the compilation of the VB program.

Notice in this example that the ASP.NET page has nothing on it other than a title and the output is being generated from the code-behind file. There are certainly no limitations on what could be done in the ASP.NET page.

In fact, any normal ASP.NET controls (like grids, calendars, etc.) can still be used. As discussed in section 5.1.1, the `NewAtlanta.BlueDragon.CfmPage` class, where the various controls/methods are defined, itself inherits from the standard ASP.NET `System.Web.UI.Page` class.

A C# example of the same code-behind doing an include is as follows:

```
public class test : NewAtlanta.BlueDragon.CfmPage {
      void Page_Load()
      {
            Response.Write(CfInclude("somepage.cfm"));
      }
}
```

Assuming the file above was named `cfinclude-method.cs`, the ASP.NET code sample above to use this file would change only on the first line, as follows:

```
<%@ Page Language="cs" inherits="test" Src="cfinclude-method.cs"%>
```

Be aware that when using C#, the case of the available methods is significant (`Cfinclude` would fail where `CfInclude` is required.)

5.3 Requirement to Use get_ Syntax For Accessing Object Properties

When accessing properties of a .NET object, you are required to use get_XXX syntax as defined for that object, which means that properties might not be accessed by name but by using a `get_`*`property`* syntax. Consider, for example, the example in section 4.1.2.2. There, an `ApplicationSettings` property was accessed using `get_ApplicationSettings()` as a method instead.

How do you know when you must or need not use `get_` syntax to access a property? Simply use any of the approaches discussed in section 5.6.2 to determine the available properties and methods of an object, and their expected format.

5.4 Root-relative Paths Are Relative to the Web Application

An important point must be noted when trying to use a root-relative path (using a leading `/`, such as in `/somedir/somefile.aspx)` in any CFML or BlueDragon-specific .NET controls or objects. The starting point for this location may not be the docroot of your web site, as you would expect.

Instead, if the page making the request is located in a virtual directory (or directory declared in IIS to be an application, both discussed in *Deploying CFML on ASP.NET and the Microsoft .NET Framework* about .NET web applications), or in one of its subdirectories, the `/` will indicate that the path is relative to the root of that web application. That may be a subdirectory within a web site docroot, rather than the web site docroot itself.

For example, consider a `default web site` configured with a docroot of `inetpub\wwwroot`. If there is a subdirectory named `mytest` which has been defined as a virtual directory or a directory declared in IIS to be an application (as discussed in *Deploying CFML on ASP.NET and the Microsoft .NET Framework*), then a `CFINCLUDE PAGE="/somefile.aspx"` will search not in the `inetpub\wwwroot` but instead in the `inetpub\wwwroot\mytest` directory. This is an important difference in working with .NET web applications.

Finally, note that while .NET server controls support a notion of using the tilde character (~) to represent the application root while the / represents the site root, BlueDragon tags and controls do not currently follow this notion.

5.5 Executing Application.cfm When Running an ASP.NET Page

As mentioned elsewhere, using the ASP.NET `Page` directive to inherit from the `NewAtlanta.BlueDragon.CfmPage` causes any existing `Application.cfm` (and optional `Onrequestend.cfm`) file for the application to be processed for the ASP.NET page.

In order to disable this behavior, in the ASP.NET page (or code-behind) override either the `OnInit()` or `OnLoad()` methods and set the `EnableApplicationCfm` attribute to `false`; make sure to invoke `MyBase.OnInit()` or `MyBase.OnLoad()` within your method (see the `System.Web.UI.Page` documentation for additional information).

5.6 Converting Sample .NET Object Calls to a CFML Equivalent

Various examples have been demonstrated showing CFML pages calling .NET objects. If you found or were given some sample .NET code calling and using some .NET object, how would you convert that to an appropriate CFML request? This section explains the challenges and solutions.

Let's assume that you find some .NET code that showed the following:

```
ConfigurationSettings.AppSettings.Get("somekey")
```

In order to covert this to CFML, we first need to know the real, full name of the class (including any package name it's within). In the case of built-in .NET objects, like this one, there is .NET

documentation which offers several pieces of information to assist in such a conversion. But a challenge with some .NET objects is that some packages are available by default in an ASP.NET page, and so references (like that above) may not indicate the complete class name. In this particular case, the full class name is `System.Configuration.ConfigurationSettings`. How would you come to know that?

Further, there may be differences in how a CFML page would refer to a property of the class (depending on how it was written). In this particular case, the `AppSettings` property would need to be accessed in BlueDragon as a method, get_`AppSettings()`. How would you determine that?

5.6.1 Determining the Complete Class Name

The first step to creating a CFML page to use a .NET class or object is to determine the real class name (including any package). If someone has built an object internally, there may exist documentation or other resources which inform you of the class name. In the case of standard .NET classes like that used above, you could also try doing a web search (such as with Google) to find references to the string `ConfigurationSettings.AppSettings`. In one test of such a search, the very first result (since it's a built-in .NET class) showed a link to the .NET documentation about it:

> http://msdn.microsoft.com/en-us/library/system.configuration.configurationsettings.appsettings(v=vs.100).aspx

This is the first step to understanding how to use the class name. From there, you can see that this `AppSettings` property is a member of `ConfigurationSettings` (as reflected in the first "see also" link at the bottom of the page. That page:

> http://msdn.microsoft.com/en-us/library/system.configuration.configurationsettings(v=vs.100).aspx

further shows at the top of its page that this particular class (`ConfigurationSettings`) is more formally known as `System.Configuration.ConfigurationSettings`. That, then, is the name we need to use when creating an instance using `CFOBJECT` or `createObject` as in the previous section.

5.6.2 Determining Available Properties and Methods: Two Ways

Most classes have multiple methods and properties which can be very interesting to explore. There are two approaches to locate an object's available methods and properties, as outlined in the following sections.

5.6.2.1 Determining Available Properties and Methods via CFDUMP

You can use the `CFDUMP` tag to display an object's capabilities once you've created an instance of that class.

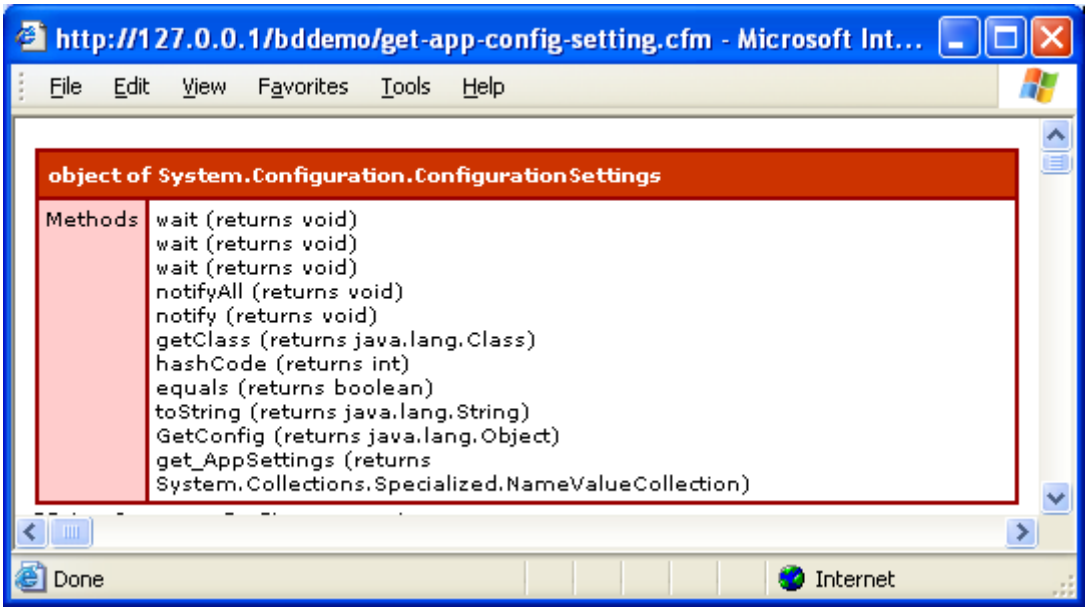
Continuing the example from section 4.1.2.2, where we had created an instance of the `ConfigurationSettings` class:

```
<cfobject action="CREATE" type=".net"
class="System.Configuration.ConfigurationSettings"
name="settings">
```

We could then use the `CFDUMP` tag as follows:

```
<CFDUMP VAR="#settings#">
```

This will dump all the object's methods and properties, as in:



Note that this shows the `appSettings` property is actually to be accessed as a method called `get_AppSettings()`. Further, we can add that method to another CFDUMP to see its available properties and methods:

```
<cfdump var="#settings.get_AppSettings()#">
```

In this case, you'd see that this object also has an AllKeys property (though represented as a get_AllKeys method), which you could further explore by using `CFDUMP` to view that method's results:

```
<cfdump var="#settings.get_AppSettings().get_allkeys()#">
```

In this case, that method returns an array of all the application settings keys defined in the web.config file (see section 4.1.2.2 for more information on application settings).

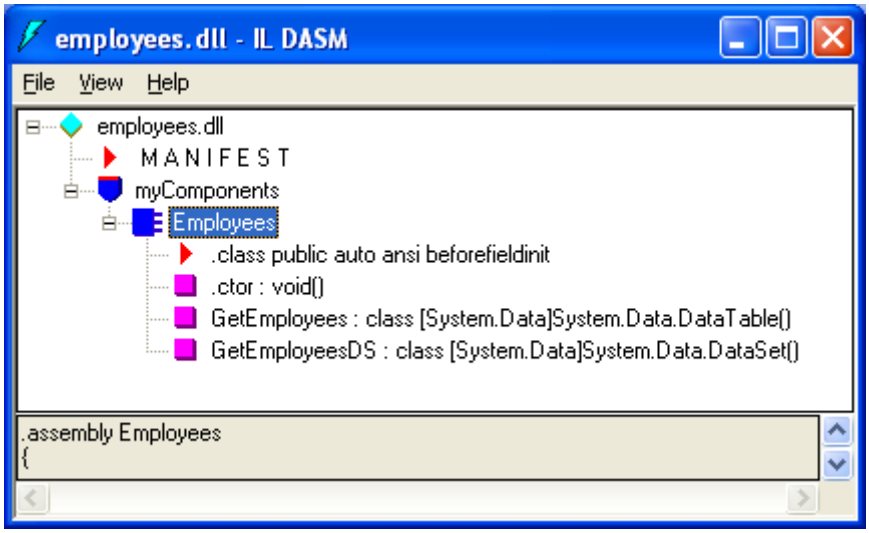
5.6.2.2 Determining Available Properties and Methods via .NET Tools

There are available .NET tools which can list methods and properties of a .NET class. First, if you have installed Visual Studio, you can use its "add reference" feature to point to a .NET object, and its explorer capability will show available properties and methods.

Another useful tool, if you've installed the .NET SDK, is the ILDASM tool (Intermediate Language Disassembler). If installed, you could open a given DLL, such as the employees.dll created in section 4.1.2.1, using this command:

```
Ildasm employees.dll
```

This will present an interface showing the available classes and their methods within the library:



You can learn more about this tool at:

http://msdn.microsoft.com/en-us/library/f7dy01k1(v=vs.100).aspx

5.7 Editors for Creating/Editing CFML and ASP.NET Pages

Note that you can continue to use traditional CFML editors to edit your CFML pages when run on BlueDragon.NET, such as Macromedia's tools ColdFusion Studio, HomeSite, HomeSite+, and Dreamweaver MX, or CFEclipse. These tools can also all edit ASP.NET pages.

It's important to note, though, that Dreamweaver MX can not only edit ASP.NET pages but also offers significant support for creating them as well as implementing ASP.NET controls on them, syntax highlighting, and more.

Additionally, a third party product, PrimalCode from Sapien Technologies, is primarily a .NET editor that also supports CFML:

http://www.sapien.com

While it's also possible to edit CFML pages in traditional ASP.NET tools such as Microsoft Visual Studio, these tools do not have specific CFML tag support, syntax highlighting, etc.

Finally, note that the Express editions of Visual Studio are offered free for one year, and while there are editions which focus on specific .NET languages, there is a new Web Developer edition that CFML developers may find very comfortable. See:

http://www.microsoft.com/express/