tavares_13.qxp 6/5/2006 12:35 PM Page 699

CHAPTER 13

# Hello, ATL Server

## A Modern C++ Web Platform

Throughout the previous 12 chapters, we've focused on ATL as a framework for building both COM components and user interfaces (via ATL's windowing classes and ActiveX control support). Now we look at ATL as a modern platform for building web applications. That framework is a set of classes known collectively as ATL Server.

Although most of the discussion thus far has involved ATL's COM support, ATL Server actually has very little to do with COM. Because its purpose is to handle Hypertext Transfer Protocol (HTTP) requests, the crux of ATL Server is built around Microsoft's Internet Service API (ISAPI). The library just happens to use COM inside. We start by briefly reviewing Microsoft's web platform. Then, we examine how ATL Server leverages that platform to provide a viable means of creating web applications and web services. There's much more to ATL Server than can be covered in this book, so we just hit the high points in this chapter.

## The Microsoft Web Platform (Internet Information Services)

When the World Wide Web began taking root in the early 1990s, many sites were little more than collections of Hypertext Markup Language (HTML) files and perhaps some image files. Clients with Internet browsers surfed to various sites using Universal Resource Locators (URLs) that took a form like this: http://www.example.com.

After typing the URL and sending the request through a maze of routers, the request finally ended up at a server somewhere. In the earliest days of the Web, the server was probably a UNIX box. Web technologies soon evolved to handle more elaborate requests (not just file requests). Through the development of the Common Gateway Interface and the inclusion of HTML tags representing standard GUI controls (such as combo boxes, push buttons, and text boxes), the Web became capable of handling *interactive* traffic. That is, a client could carry on a two-way conversation with web servers that extended beyond simple file requests. Of course, this capability led to the great upsurge in web sites during the late 1990s, giving rise to such enterprises as Amazon.com and Google.

tavares_13.qxp 6/5/2006 12:35 PM Page 700



The earliest means of supporting interactive capabilities over the Web were made available through the Common Gateway Interface (CGI). CGI worked by handling separate incoming requests with a new process for each request. Although the Microsoft platform supports the CGI, it doesn't work quite as well as on a UNIX box. Starting new processes on a UNIX box is not as expensive as starting new processes on a Windows box. To compensate, Microsoft introduced the Internet Services API and its own new Internet server: Internet Information Services (IIS). The IIS strategy is that it's much faster to load a DLL to respond to an HTTP request than it is to start a whole new process.

### IIS and ISAPI

The IIS component is the heart of Microsoft's web platform. Most modern web sites built around the Microsoft platform involve IIS and ISAPI DLLs. All the modern Microsoft web-based programming frameworks are extensions of the IIS/ISAPI architecture. Even classic ASP and the more modern ASP.NET rely on IIS and ISAPI at their core. And as you'll soon see, ATL Server depends upon IIS and ISAPI as well.

Regardless of the programming framework used (raw sockets programming, ASP, ASP.NET, or ATL Server), processing web requests is similar from framework to framework. A component listens to port 80 for HTTP requests. When a request, comes in, the component parses the request and figures out what to do with it. The request might vary from sending back some specific HTML, to returning a graphics file, or perhaps even to invoking a method of some sort.

When programming to the modern Microsoft web-based frameworks, IIS is the component that listens to port 80. IIS handles some requests directly and delegates others to specific ISAPI extension DLLs to execute the request. By default, IIS handles requests for standard HTML files directly. As an alternative, a custom file extension might be mapped to a handwritten ISAPI DLL that executes the request.

Classic ASP and ASP.NET integrate into IIS as ISAPI DLLs. IIS handles requests ASP files (`*.asp`) by mapping them to an ISAPI DLL named `ASP.DLL`, which handles the request by parsing the request string, loading the ASP file, parsing the contents of the file, and servicing the request according to the instructions given in the ASP file. ASP.NET files (`*.aspx`) are handled by an ISAPI DLL named `ASPNET_ISAPI.DLL`, which brings in the Common Language Runtime to help it process requests.

To set the stage for understanding ATL Server, let's take a look at how ISAPI extension DLLs work.

### ISAPI Extension DLLs

Although IIS does a perfectly fine job responding to requests for standard web file types (such as HTML and JPG), its real power lies in the capability to extend your server by writing custom DLLs to respond to requests.

tavares_13.qxp 6/5/2006 12:35 PM Page 701



The core ISAPI infrastructure is actually fairly simple. ISAPI extension DLLs implement three entry points:

```
BOOL WINAPI GetExtensionVersion(HSE_VERSION_INFO* pVer);

DWORD WINAPI HttpExtensionProc(LPEXTENSION_CONTROL_BLOCK lpECB);

BOOL WINAPI TerminateExtension(DWORD dwFlags);
```

These three methods are the hooks for writing your web site using custom DLLs. `GetExtensionVersion` is called when IIS invokes the application for the first time. `GetExtensionVersion` must set the version number in the `HSE_VERSION_INFO` structure passed in and then return `TRUE` for IIS to be capable of using your ISAPI DLL.

IIS calls the `TerminateExtension` entry point when IIS is ready to unload the DLL from its process. If you don't need to do any cleanup, the `TerminateExtension` entry point is optional.

The heart of the extension is the `HttpExtensionProc` function. Notice that `HttpExtensionProc` takes a single parameter: an `EXTENSION_CONTROL_BLOCK` structure. The structure includes everything you'd ever want to know about a request, including the kind of request made, the content of the request, the type of content, a method for getting the server variables (for example, information about the connection), a method for writing output to the client, and a method for reading data from the body of the HTTP request. Here's the `EXTENSION_CONTROL_BLOCK`:

```
typedef struct _EXTENSION_CONTROL_BLOCK {
  DWORD cbSize;             // size of this struct.
  DWORD dwVersion;          // version info of this spec
  HCONN ConnID;             // Context number not to be modified!
  DWORD dwHttpStatusCode;   // HTTP Status code
  CHAR  lpszLogData[HSE_LOG_BUFFER_LEN]; // null terminated log
                        // info specific to this Extension DLL

  LPSTR lpszMethod;         // REQUEST_METHOD
  LPSTR lpszQueryString;    // QUERY_STRING
  LPSTR lpszPathInfo;       // PATH_INFO
  LPSTR lpszPathTranslated; // PATH_TRANSLATED
  DWORD cbTotalBytes;       // Total bytes indicated from client
  DWORD cbAvailable;        // Available number of bytes
  LPBYTE lpbData;           // pointer to cbAvailable bytes
  LPSTR  lpszContentType;   // Content type of client data

  BOOL (WINAPI * GetServerVariable)(
```

tavares_13.qxp 6/5/2006 12:35 PM Page 702



```
    HCONN hConn, LPSTR lpszVariableName,
    LPVOID lpvBuffer, LPDWORD lpdwSize);

  BOOL (WINAPI * WriteClient)(HCONN ConnID, LPVOID Buffer,
    LPDWORD lpdwBytes, DWORD dwReserved);

  BOOL (WINAPI * ReadClient)(HCONN ConnID, LPVOID lpvBuffer,
    LPDWORD lpdwSize);

  BOOL (WINAPI * ServerSupportFunction)(HCONN hConn,
    DWORD dwHSERequest, LPVOID lpvBuffer, LPDWORD lpdwSize,
    LPDWORD lpdwDataType);
} EXTENSION_CONTROL_BLOCK, *LPEXTENSION_CONTROL_BLOCK;
```

When IIS receives a request, it packages the information into the `EXTENSION_CONTROL_BLOCK` and passes a pointer to the structure into the ISAPI DLL via the `HttpExtensionProc` entry point. The ISAPI extension's job is to parse the incoming request into a useable form. After that, it's completely up to the ISAPI DLL to do whatever it wants to with the request. For example, if the client makes some sort of request using query parameters (perhaps a product lookup), the ISAPI DLL might use those parameters to create a database query. The ISAPI DLL passes any results back to the client using the pointer to the `WriteClient` method passed in through the extension block.

If you've had any experience working with frameworks such as classic ASP or ASP.NET, most of this structure will appear familiar to you. For example, when you call `Write` through ASP's intrinsic `Response` object, execution eventually ends up passing through the method pointed to by `WriteClient`.

## The Simplest ISAPI Extension That Could Possibly Work[1]

Let's take a look at a simple "Hello, World" ISAPI extension. The result looks like Figure 13.1.

### Writing the Extension

Starting with a Win32 DLL project (not an ATL DLL project) named HelloISAPI, including the `HttpExt.h` system header file in `stdafx.h` brings in the ISAPI extension types, which enables us to implement the `GetExtensionVersion` function:

[1] With apologies to Kent Beck.

tavares_13.qxp 6/5/2006 12:35 PM Page 703



```
BOOL WINAPI GetExtensionVersion(HSE_VERSION_INFO *pVer) {
  pVer->dwExtensionVersion = HSE_VERSION;
  strncpy_s( pVer->lpszExtensionDesc,
    HSE_MAX_EXT_DLL_NAME_LEN,
    "Hello ISAPI Extension", _TRUNCATE );
  return TRUE;
}
```

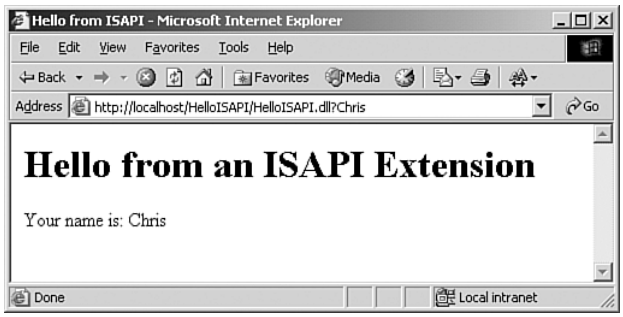


**Figure 13.1** Hello World, ISAPI style

All I did here was set the ISAPI version number (`HSE_VERSION` is a constant defined in `HttpExt.h`) and fill in the description field for use by IIS. We use the `HSE_VERSION` constant because we're reporting the version of the ISAPI API that we're using, not the version number of our extension DLL. This way, as IIS and ISAPI changes, IIS can maintain backward-compatibility with older extensions.

The meat of the extension is the `HttpExtensionProc` function:

```
DWORD WINAPI HttpExtensionProc(EXTENSION_CONTROL_BLOCK *pECB) {
  char *header =
    "<html>"
      "<head>"
        "<title>Hello from ISAPI</title>"
      "</head>"
      "<body>"
        "<h1>Hello from an ISAPI Extension</h1>";

  DWORD size = static_cast< DWORD >( strlen( header ) );
  pECB->WriteClient( pECB->ConnID, header, &size, 0 );

  char *intro = "<p>Your name is: ";
  size = static_cast< DWORD >( strlen( intro ) );
  pECB->WriteClient( pECB->ConnID, intro, &size, 0 );
```

tavares_13.qxp 6/5/2006 12:35 PM Page 704



```
  size = static_cast< DWORD >(
    strlen( pECB->lpszQueryString ) );
  pECB->WriteClient( pECB->ConnID, pECB->lpszQueryString,
    &size, 0 );

  char *footer =
        "</p>"
      "</body>"
    "</html>";

  size = static_cast< DWORD >( strlen( footer ) );
  pECB->WriteClient( pECB->ConnID, footer, &size, 0 );

  return HSE_STATUS_SUCCESS;
}
```

This method simply builds up a series of strings and then sends them out to the browser via the `pECB->WriteClient()` method. We also grab the query string (the stuff in the URL after the `?`) by using the `lpszQueryString` field in the `EXTENSION_CONTROL_BLOCK`.

In this case, the `TerminateExtension` method doesn't need to do anything and, thus, is technically optional, but adding a minimal stub for completeness looks like this:

```
BOOL WINAPI TerminateExtension(DWORD dwFlags) {
  return TRUE;
}
```

Returning `TRUE` tells IIS that it can unload this extension. If we had loaded anything that needed to be cleaned up, the call to `TerminateExtension` would be the place to do it.

There's one more thing we need to add to the project before we can deploy it: We have to export the ISAPI entry points. Normal practice is to add the `__declspec(dllexport)` declaration to each of the ISAPI functions; this tells the compiler that we want the suitably decorated function to be exported from the DLL. Unfortunately, when you do this with an ISAPI extension, you get this:

```
c:\helloisapi\helloisapi.cpp(24) : error C2375: 'GetExtensionVersion' :
redefinition; different linkage
        c:\program files\microsoft visual studio
8\vc\platformsdk\include\httpext.h(526) : see declaration of
'GetExtensionVersion'
```

tavares_13.qxp 6/5/2006 12:35 PM Page 705



```
c:\helloisapi\helloisapi.cpp(33) : error C2375: 'HttpExtensionProc' :
redefinition; different linkage
        c:\program files\microsoft visual studio
8\vc\platformsdk\include\httpext.h(527) : see declaration of
'HttpExtensionProc'
c:\helloisapi\helloisapi.cpp(65) : error C2375: 'TerminateExtension' :
redefinition; different linkage
        c:\program files\microsoft visual studio
8\vc\platformsdk\include\httpext.h(528) : see declaration of
'TerminateExtension'
```

If we look inside `HttpExt.h` on lines 526–528, we see the problem:

```
/************************************************************
 *   Function Prototypes
 *   o  for functions exported from the ISAPI Application DLL
 ************************************************************/

BOOL  WINAPI   GetExtensionVersion( HSE_VERSION_INFO  *pVer );
DWORD WINAPI   HttpExtensionProc(  EXTENSION_CONTROL_BLOCK *pECB );
BOOL  WINAPI   TerminateExtension( DWORD dwFlags );
```

The `HttpExt.h` file includes prototypes for the ISAPI entry points, but alas, the prototypes don't include `__declspec(dllexport)`. When the compiler sees the prototype without the `export` directive and the actual implementation with the directive, it gives the error I just showed you.

To work around this, we need to do exports the old-fashioned way: Add a `.def` file to the project. This is a separate file that explicitly lists the functions we want exported from our DLL. As the number of exports grows, a `.def` file gets difficult to maintain, but we've got only three exports here, and they're the same for every ISAPI extension, so it's not too much of a burden.

The `.def` file looks like this:

```
;HelloISAPI.def : Declares the module parameters for the DLL.
LIBRARY    "HelloISAPI"

EXPORTS
   HttpExtensionProc
   GetExtensionVersion
   TerminateExtension
```

tavares_13.qxp 6/5/2006 12:35 PM Page 706



### Configuring the Server

Next, we need to get the extension onto the server. Luckily, Visual Studio has some web-deployment tools built in to help, but when you're starting from scratch, they're not set up yet.

First things first: We need an IIS Virtual Directory. I recommend creating a directory that's separate from your build directory for this (you'll see why in a moment). The Internet Services Manager is a nice place to create virtual directories; the sample is called HelloISAPI.[2]

Next, we need to configure permissions on the directory so that the ISAPI extension will be executed and, more important, so we can debug it. In Internet Services Manager, right-clicking the HelloISAPI virtual directory and choosing Properties yields Figure 13.2.

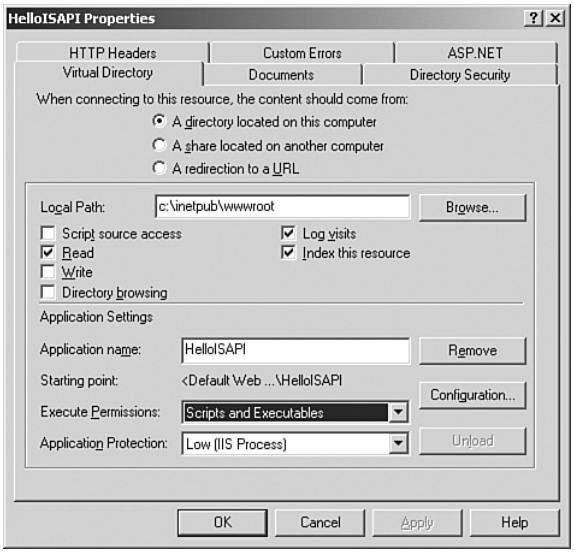


**Figure 13.2** Virtual directory properties

We need to make sure two settings are correct in this dialog box. The first is the Execute Permissions drop-down list, which must be set to Scripts and Executables. (It defaults to Scripts Only.) Without this setting, instead of loading the extension and executing it, IIS attempts to download the DLL to your browser.

[2] To create a virtual directory, first create a directory via Windows Explorer. It's easiest to create it under C:\inetpub\wwwroot because permissions will be set correctly. Next, open the Internet Services Manager. Expand the treeview on the left until you see Default Web Site. Right-click this and choose New, Virtual Directory. Follow the wizard to name the virtual directory, and then choose the directory you just created. Click Next and then Finish.

tavares_13.qxp 6/5/2006 12:35 PM Page 707



The second setting is the Application Protection drop-down list. For easy debugging, you want this set to Low. This causes IIS to load an extension directly into the IIS process. This is generally a bad thing to do on a production server because misbehaving extensions can take the IIS process down with it, but for debugging, it just makes things easier. Do this only on a development server, not on your production boxes![3]

### Configuring the Project

With a virtual directory created and configured to receive our ISAPI extension code, we want to configure Visual Studio to deploy the code on a successful build. In the project's property pages, there's an entry called Web Deployment (see Figure 13.3).

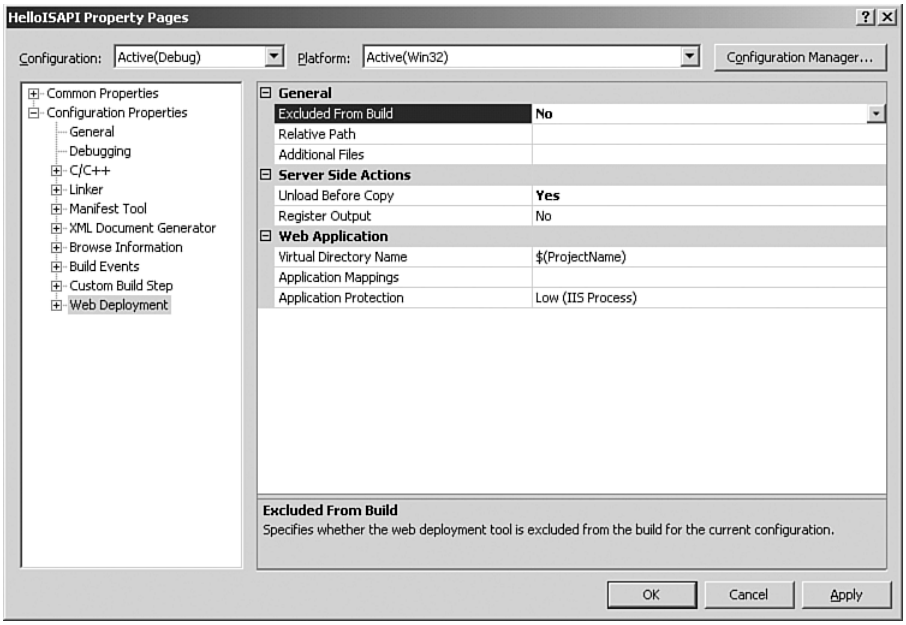


**Figure 13.3** Web Deployment settings

By default, the Excluded From Build property is set to Yes, which means that web deployment won't happen. Changing it to No enables web deployment, which, by default, uses the name of the project at the IIS virtual directory name that will receive the result of a successful build—in this example, our `HelloISAPI.DLL` file.

[3] Some recommend that you debug in IIS out-of-process mode. Although I can see the appeal, setting that up involves mucking around with COM+ settings, which is often very complicated to get right. If you want to give it a try, take a look at http://www.west-wind.com/presentations/iis5Debug.htm (http://tinysells.com/53).

tavares_13.qxp 6/5/2006 12:35 PM Page 708



At this point, compiling the project and surfing to http://localhost/HelloISAPI/HelloISAPI.dll?SomeName should get back a result that looks like Figure 13.1. If you make changes to the project and rebuild, VS is smart enough to tell ISAPI to unload our DLL so that it can be replaced. This is also the reason that we wanted a separate build directory from our virtual directory. If we compile straight into the virtual directory, IIS typically has the DLL locked. You would need to use a prebuild step to shut down IIS so that Visual Studio could regenerate the `.dll` file after it compiles. By using web deployment, you don't need to pay the compile-time speed hit for resetting IIS until the compile succeeds.

If you click F5 to start a debugging session, by default, you get the helpful Executable for Debug Session dialog box. We want to configure Visual Studio to automatically attach to IIS when debugging. To do this, navigate to the Debugging page of the project properties and set the options to match those shown in Figure 13.4.

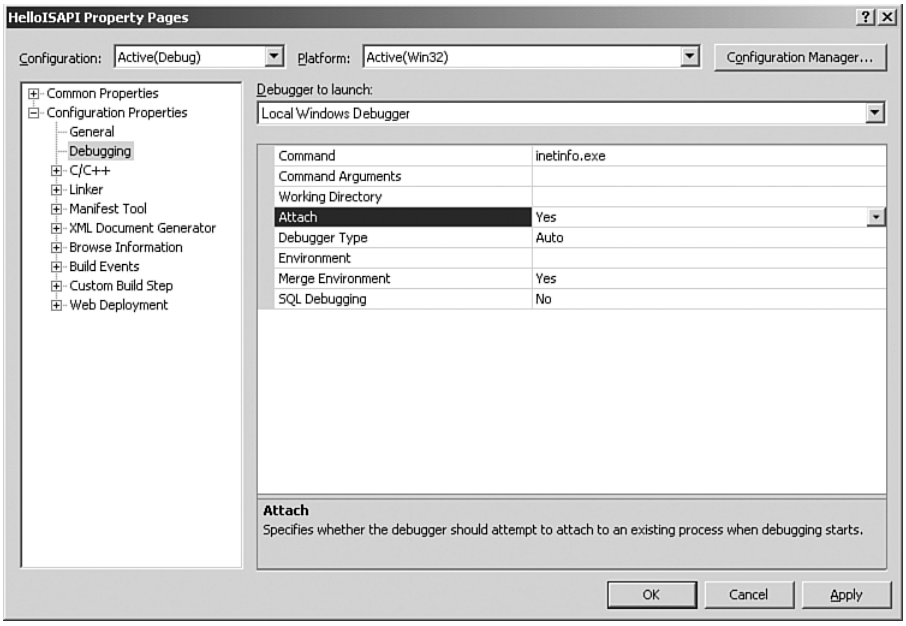


**Figure 13.4** Debugging settings

You need to change these two fields:

- **Command.** Set this to inetinfo.exe, which is the name of the IIS process.
- **Attach.** Set this to Yes. If it is set to No, the debugger attempts to start a second copy of inetinfo.exe. Yes means "Attach to already running copy."

tavares_13.qxp 6/5/2006 12:35 PM Page 709



Now, to debug your extension, set your breakpoints wherever you want and click F5 to start the debugger. Visual Studio crunches for a while after it attaches to IIS and then looks like it's doing nothing. At this point, start a browser and point it at your extension's URL; your breakpoints should fire.

## Wrapping ISAPI

You now have a simple extension written directly to ISAPI, but programming against raw ISAPI is rather awkward. It sure would be nice to have some more object-oriented wrappers. Luckily, ATL Server provides exactly that. The `CServerContext` class is a COM object that provides a wrapper around the ECB.

```
class CServerContext :
    public CComObjectRootEx<CComMultiThreadModel>,
    public IHttpServerContext {
public:
    BEGIN_COM_MAP(CServerContext)
        COM_INTERFACE_ENTRY(IHttpServerContext)
    END_COM_MAP()

    void Initialize(__in EXTENSION_CONTROL_BLOCK *pECB);

    LPCSTR GetRequestMethod();
    LPCSTR GetQueryString();
    LPCSTR GetPathInfo();
    LPCSTR GetScriptPathTranslated();
    LPCSTR GetPathTranslated();
    DWORD GetTotalBytes();
    DWORD GetAvailableBytes();
    BYTE *GetAvailableData();
    LPCSTR GetContentType();
    BOOL GetServerVariable(
        LPCSTR pszVariableName,
        LPSTR pvBuffer,
        DWORD *pdwSize);

    BOOL WriteClient(void *pvBuffer, DWORD *pdwBytes);
    BOOL AsyncWriteClient(void *pvBuffer, DWORD *pdwBytes);
    BOOL ReadClient(void *pvBuffer, DWORD *pdwSize);
    BOOL AsyncReadClient(void *pvBuffer, DWORD *pdwSize);

    BOOL SendRedirectResponse(__in LPCSTR pszRedirectUrl);
```

tavares_13.qxp 6/5/2006 12:35 PM Page 710



```
    BOOL GetImpersonationToken(__out HANDLE * pToken);

    BOOL SendResponseHeader(
        LPCSTR pszHeader = "Content-Type: text/html\r\n\r\n",
        LPCSTR pszStatusCode = "200 OK",
        BOOL fKeepConn=FALSE);

    BOOL DoneWithSession(__in DWORD dwHttpStatusCode);
    BOOL RequestIOCompletion(__in PFN_HSE_IO_COMPLETION pfn,
        DWORD *pdwContext);

    BOOL TransmitFile(
        HANDLE hFile,
        PFN_HSE_IO_COMPLETION pfn,
        void *pContext,
        LPCSTR szStatusCode,
        DWORD dwBytesToWrite,
        DWORD dwOffset,
        void *pvHead,
        DWORD dwHeadLen,
        void *pvTail,
        DWORD dwTailLen,
        DWORD dwFlags);

    BOOL AppendToLog(LPCSTR szMessage, DWORD *pdwLen);

    BOOL MapUrlToPathEx(LPCSTR szLogicalPath, DWORD dwLen,
        HSE_URL_MAPEX_INFO *pumInfo);
};
```

The `CServerContext` class provides a somewhat more convenient wrapper for the ISAPI interface. Before you use it, you need to add an ATL module object to your project and initialize it in DLL main. This code takes care of the housekeeping:

```
// Our module class definition
class CHelloISAPI2Module :
  public CAtlDllModuleT<CHelloISAPI2Module> {};

// Required global instance of module.
CHelloISAPI2Module _AtlModule;

// Initialize / shutdown module on DLL load or unload
BOOL APIENTRY DllMain( HMODULE hModule,
  DWORD  ul_reason_for_call, LPVOID lpReserved ) {
```

tavares_13.qxp 6/5/2006 12:35 PM Page 711



```
  // Set up shared data for use by CServerContext
  return _AtlModule.DllMain(ul_reason_for_call, lpReserved);
}
```

Having initialized shared data that `CServerContext` makes use of, we can now take advantage of `CServerContext`:

```
DWORD WINAPI HttpExtensionProc(EXTENSION_CONTROL_BLOCK *pECB) {
  CComObject< CServerContext > *pCtx;
  CComObject< CServerContext >::CreateInstance( &pCtx );
  pCtx->Initialize( pECB );
  // We use this smart pointer to ensure proper cleanup
  CComPtr< IUnknown > spContextUnk( pCtx );

  char *header =
    "<html>"
      "<head>"
        "<title>Hello from ISAPI</title> "
      "</head>"
      "<body>"
        "<h1>Hello from an ISAPI Extension</h1>";

  DWORD size = static_cast< DWORD >( strlen( header ) );
  pCtx->WriteClient( header, &size );

  char *intro = "<p>Your name is: ";
  size = static_cast< DWORD >( strlen( intro ) );
  pCtx->WriteClient( intro, &size );

  size = static_cast< DWORD >(
    strlen( pCtx->GetQueryString( ) ) );
  pCtx->WriteClient(
    const_cast< LPSTR >(pCtx->GetQueryString( )), &size );

  char *footer =
      "\r\n</body>\r\n"
    "</html>\r\n";

  size = static_cast< DWORD >( strlen( footer ) );
  pCtx->WriteClient( footer, &size );

  return HSE_STATUS_SUCCESS;
}
```

The bold lines show where we've replaced direct calls to the ECB with calls via the context. The `WriteClient` method is at least a little shorter, but we actually haven't gained much at this point. Luckily, there's more API wrapping to be done.

tavares_13.qxp 6/5/2006 12:35 PM Page 712



### Request and Response Wrappers

Anyone who has done ASP or ASP.NET development is familiar with the `Request` and `Response` objects. The former gives access to the contents of the HTTP request: URL, query string, cookies, and so on. The latter gives you a way to write data to the output stream. ATL Server provides similar wrappers that make both reading and writing from the ECB much more pleasant.

The `CHttpRequest` object is a wrapper object that lets you get at the contents of the HTTP request:

```
class CHttpRequest : public IHttpRequestLookup {
public:
  // Constructs and initializes the object.
  CHttpRequest(
    IHttpServerContext *pServerContext,
    DWORD dwMaxFormSize=DEFAULT_MAX_FORM_SIZE,
    DWORD dwFlags=ATL_FORM_FLAG_NONE);

  CHttpRequest(IHttpRequestLookup *pRequestLookup);

  // Access to Query String parameters as a collection
  const CHttpRequestParams& GetQueryParams() const;

  // Get the entire raw query string
  LPCSTR GetQueryString();

  ... Other methods omitted - we'll talk about them later
}; // class CHttpRequest
```

The `CHttpRequest` object gives you easy access to everything in the request. Using the `CHttpRequest` object simplifies getting at query string variables. In the raw C++ extension code, I cheated and just passed data (such as `?Chris`), to make using that data easier. However, typical web pages take named query parameters separated with the ampersand (&) and potentially containing special characters encoded in hex:

```
http://localhost/HelloISAPI2/HelloISAP2.dll?name=Chris&motto=I%20Love%20ATL
```

Unfortunately, decoding and parsing query strings in their full glory requires quite a bit of work to get right, which is where ATL Server really starts to shine via the `CHttpRequest` class:

```
DWORD WINAPI HttpExtensionProc(EXTENSION_CONTROL_BLOCK *pECB) {
  CComObject< CServerContext > *pCtx;
```

tavares_13.qxp 6/5/2006 12:35 PM Page 713



```
  CComObject< CServerContext >::CreateInstance( &pCtx );
  pCtx->Initialize( pECB );
  // We use this smart pointer to ensure proper cleanup
  CComPtr< IUnknown > spContextUnk( pCtx );

  CHttpRequest request( pCtx );

  char *header =
    "<html>"
      "<head>"
        "<title>Hello from ISAPI</title>"
      "</head>"
      "<body>"
        "<h1>Hello from an ISAPI Extension</h1>";

  DWORD size = static_cast< DWORD >( strlen( header ) );
  pCtx->WriteClient( header, &size );

  CStringA name;
  name = request.GetQueryParams( ).Lookup( "name" );
  if( name.IsEmpty( ) ) {
    char *noName = "<p>You didn't give your name";
    size = static_cast< DWORD >( strlen( noName ) );
    pCtx->WriteClient( noName, &size );
  }
  else {
    char *intro = "<p>Your name is: ";
    size = static_cast< DWORD >( strlen( intro ) );
    pCtx->WriteClient( intro, &size );
    size = name.GetLength( );
    pCtx->WriteClient( name.GetBuffer( ), &size );
  }

  char *footer =
        "\r\n</p>\r\n"
      "\r\n</body>\r\n"
    "</html>\r\n";

  size = static_cast< DWORD >( strlen( footer ) );
  pCtx->WriteClient( footer, &size );

  return HSE_STATUS_SUCCESS;
}
```