Chapter 25

ISAPI

CONTENTS

Web servers need the capability to expand their horizons. They occupy a unique niche in technology, allowing people to view information you want them to see. As such, each situation is unique, and it would be near impossible to pick any one set of functions or methods and say, "This is it. This is how everyone will deal with information." That would be like making cars in only one color. People might try to sell you on the idea that generic is good, but individuality (on either a personal or a corporate level) needs to find a way to express itself.

CGI programs are great at what they do-gather information, process it, and generate output. But they also have disadvantages, such as needing to create a new instance of themselves every time someone runs the script. If five people are using the function, there are five copies of that process in memory. If your site gets thousands of hits, and most of them are going to be starting CGI processes... you get the picture. Lots of wasted memory space and processing time, all to do some simple (or maybe not so simple) functions.

The Internet Server API (ISAPI) is a different method of dealing with informational functions. It applies high-level programming to give you the most efficient combination of power and flexibility, including information that is almost impossible to get through CGI. By developing a program in a certain manner, and by meeting certain requirements, you gain access to this hidden world of additional power, information, and speed. Be warned, however, that prog-ramming ISAPI functions is not something to be approached lightly, or by the faint of programming-heart. It can be a jungle out there.

This chapter does not assume much background in the realm of C or C++ programming.

The primary focus here is not the actual writing of ISAPI code, but understanding the concepts behind it. Among these concepts are

- What ISAPI is all about
- ISAPI's two schools-Applications and Filters
- How does it all work?
- Implementation Complications
- Future directions for ISAPI

With these concepts firmly in hand, and some examples of code sneaked in here and there, you'll have a place to start planning your own functions.

What Is ISAPI All About?

When you're working with a Web server, it is useful to gain more information and be able to deal with it faster. You want ways of getting at details that normal CGI can't give you, as well as ways to modify those bits of information. You want a method of doing it faster and more efficiently, so that the only lag time that exists is the user sorting through the cool stuff you can do for themin an almost instantaneous manner. You want an API, and you want it now.

An Application Programming Interface (API) exists so that you can do fun things with it. You gain access to the inner workings of the programitself, giving you more freedom and power to do things with that information. In the case of a Web server, there are all sorts of hidden things you might want to get hold of, such as user authorization information, the ability to manipulate how errors are handled, and how information is logged on the system In addition, APIs are faster than normal CGI programs during execution, and take upless resources while running. This means more power, more users and fewer problems.

Note
In theory, API functions are supposed to be faster, and thus better in general for use. Some people even say that CGI will become obsolete. Later in the chapter we'll cover some of the problems that API functions can present later on in this chapter during "Implementation Complications," and you'll get a sense of why API programming isn't for everyone, no matter what benefits it may have.

To take advantage of all this freedomand power, Microsoft and Process Software teamed upto create an API standard for their Web servers (and for anyone who wants to adopt it). The aptly-named Internet Server API (ISAPI) is a whole collection of

functions that allow you to extend the capability of your web server in a nearly unlimited number of ways. There are actually two very distinct components present in the ISAPI standard that will be discussed separately in the next section, "ISAPI Background and Functionality." I will later discuss ISAPI as a whole in the "Implementation Complications" section later in the chapter.

ISAPI Background and Functionality

The ISAPI standard is a very recent, but natural, invention. Microsoft has long been providing Windows developers with access to Windows' inner workings through the Windows Software Development Kit (SDK), while Process software has been providing people with Web servers. When Microsoft began development of its new Internet Information Server (IIS), it was expected that they would allow developers the opportunity to get down and dirty with IIS' functionality: they didn't disappoint anyone with the release of the ISAPI.

The two branches of the ISAPI, Internet Server Applications (ISAs) and ISAPI Filters, comprise two different schools of thought on how programmers can approach additional functionality. ISAs are the more traditional of the two, leading programmers to develop something that's more of an external component with special links back into the server's workings. ISAPI Filters are closer to building blocks, which can be attached directly to the server, providing a seamless component that carefully monitors the HTTP requests being directed at the server. Since each has its own particular way of being dealt with by the server, I'll look at them as separate entities, and tie together the common points where they conveniently overlap.

Internet Server Applications (ISAs)

Internet Server Applications (ISAs), which can also be called ISAPI DLLs, are the first step in extending a server's functionality. Much like a traditional CGI program, an ISA might find itself referenced in a form entry like the following:

```
<form method=POST action=/scripts/function.dll>
```

> **Note**
>
> See Chapter 8, "Forms and How to Handle Them," for more details on CGI used with form elements.

An ISA performs the same task of gathering the POSTed form data, parsing it out, and doing something with it, but there the similarities stop. Although the surface elements look exactly the same, what occurs once the form in question gets submitted (or whatever other action triggers the ISA to execute) is a completely different matter.

Figure 25.1 shows the typical path of processes in ISAPI and CGI requests.

Figure 25.1: *Request processes for ISAPI in CGI.*

Figure 25.1 shows an example of how communication works between various entities in the land of the server. Requests are routed to the main HTTP server. When the server receives instructions to start a typical CGI program, it needs to make a separate process for that request. It sends the data out to the CGI program through the environment variables and Standard Input (STDIN). The CGI program, in turn, processes that information from the environment variables and STDIN, then sends output (normally through Standard Output (STDOUT)) back to the server, which responds to the request. This action takes place far from home so there's going to be some delay. In addition, there's some information that the server can't export past its own boundaries.

Requests that go to an ISA, on the other hand, stay within the boundaries of the server's process territory. The data is handled using Extension Control Blocks (ECBs). There's much less work involved in getting the data to the ISAs. Also, because it's closer to home, it also allows for more detailed exchanges of information, even changes to the server based on that information. There's a lot more going on than might meet the eye.

What happens when an ISA function is called? There are a number of internal steps:

- Server receives call
- Server checks function and loads it, if not already in memory
- Function reads data from Extension Control Blocks
- Data gets processed
- Function sends output back to client
- Server terminates function and unloads it, if desired

When the server receives a request to start the ISA, one of the first things it does is check to see if the ISA is already in the memory. This is called Run-Time Dynamic Linking. While the program is running, it hooks up with other components that it needs and recognizes that it already has them on board when other requests come in for those components' functions. These components are commonly referred to as Dynamic Linked Libraries, or DLLs. Just as the name might imply, DLLs are libraries of functions that an application can dynamically link to and use during its normal execution. Anyone who uses the Windows operating system, in any version, has encountered DLLs before-Windows is a whole compilation of mutually cooperative DLL functions. Each function can call out to another to do whatever needs to be done. When the server needs to load the DLL, it calls into a special entry point that defines an ISAPI function, as opposed to some other DLL that might not be safe to use.

The primary entry point that the server looks for in an ISA is the `GetExtensionVersion()` function. If the server calls out to that function, and nobody answers, it knows that it's not a usable function. Therefore, the attempt to load the

DLL into memory will fail. If, on the other hand, the function is there, then it will let the server know what version of the API it conforms to. Microsoft's recommended implementation of a `GetExtensionVersion()` definition is

```
BOOL WINAPI GetExtensionVersion(HSE_VERSION_INFO  *version )
  {
   version->dwExtensionVersion = MAKELONG(HSE_VERSION_MAJOR,
HSE_VERSION_MINOR);
   lstrcpyn( version->lpszExtensionDesc, "This is a sample
Extension",
             HSE_MAX_EXT_DLL_NAME_LEN);
   return TRUE;
}
```

The `GetExtensionVersion()` entry point is really just a way for the server to ensure that the DLL is conforming to the specification that the server itself conforms to. It could be that the server or function is too old (or too new), and so they wouldn't work well together. It's also possible that future changes will need to know past versions to accommodate for special changes, or use it for some other compatibility purpose.

The actual startup of the function occurs at the `HttpExtensionProc()` entry point. Similar to the `main()` function declaration in a standard C program, it accepts data inside an Extension Control Block. This block is made available to the function to figure out what to do with the incoming data before composing a response. Here is the declaration for the `HttpExtensionProc()` entry point:

```
DWORD WINAPI HttpExtensionProc( LPEXTENSION_CONTROL_BLOCK *lpEcb);
```

Whatever happens, you can't keep the client waiting; you have to tell them something. In addition, it has to be something that the server understands and can properly deal with. To properly create a response, the ISA can call on either the `ServerSupportFunction()` or the `WriteClient()` function (These functions are defined and explained in "Callback Functions."). Within that response, it will want to return one of the valid return values shown in Table 25.1.

Table 25.1. Acceptable return values for an ISA application.

Return Value	*Meaning*
`HSE_STATUS_SUccESS`	The ISA successfully completed its task, and the server can disconnect and clean up
`HSE_STATUS_SUccESS_ AND_KEEP_CONN`	The ISA successfully completed its task, but the server shouldn't disconnect just yet, if it supports persistent connections. The application hopes it will wait for another HTTP request.

HSE_STATUS_PENDING	The ISA is still working and will let the server know when it's done by sending an HSE_REQ_DONE_WITH_SESSION message through the ServerSupportFunction call.
HSE_STATUS_ERROR	Whoops, something has gone wrong in the ISA. The server should end the connection and free up space.

All of these extension processes have to interact with something to get their data, and, as shown before, there's a group of intermediaries called Extension Control Blocks (ECBs) that handle that particular duty. They're nothing more than a C structure that is designed to hold specific blocks of data and allow a few functions to make use of that data. Here is the setup of an ECB:

Listing 25.1. Extension Control Block structure.

```
// To be passed to extension procedure on a new request
//
typedef struct _EXTENSION_CONTROL_BLOCK {
    DWORD    cbSize;         //Size of this struct
    DWORD    dwVersion;     //Version info for this spec
    HCONN    ConnID;         //Connection Handle/ContextNumber(don't
modify!)
    DWORD    dwHttpStatusCode;    //Http Status code for request
    chAR        lpszLogData[HSE_LOG_BUFFER_LEN];
                        //Log info for this specific  request (null
terminated)
    LPSTR    lpszMethod;         // REQUEST_METHOD
    LPSTR    lpszQueryString;     // QUERY_STRING
    LPSTR    lpszPathInfo;         // PATH_INFO
    LPSTR    lpszPathTranslated;     // PATH_TRANSLATED
    DWORD    cbTotalBytes;     // Total Bytes from    client
    DWORD    cbAvailable;     // Available Bytes
    LPBYTE     lpbData;         // Pointer to client Data (cbAvailable
bytes worth)
    LPSTR    lpszContentType;     // Client Data Content  Type
    BOOL     (  WINAPI * GetServerVariable)
        (    HCONN    ConnID,
            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    ConnID,
            DWORD    dwHSERRequest,
            LPVOID    lpvBuffer,
            LPDWORD    lpdwSize,
            LPDWORD    lpdwDataType);
}
```

Table 25.2 goes into detail about each individual component of an Extension Control Block.

Table 25.2. Explanation of fields in the Extension Control Block.

Field	*Data Direction*	*Comments*
`cbSize`	IN	The size of the structure itself (shouldn't be changed).
`dwVersion`	IN	Version information of this specification. The formfor this information is `HIWORD` for major version number, and `LOWORD` for minor version number.
`ConnID`	IN	A connection handle uniquely assigned by the server (DO NOT chANGE).
`dwHttpStatusCode`	OU	The status of the current transaction once completed.
`lpszLogData`	OU	Contains a null-terminated string for log information of the size (`HSE_LOG_BUFFER_LEN`).
`lpszMethod`	IN	Equivalent of the environment variable `REQUEST_METHOD`.
`lpszQueryString`	IN	Equivalent of the environment variable `QUERY_STRING`.
`lpszPathInfo`	IN	Equivalent of the environment variable `PATH_INFO`.
`lpszPathTranslated`	IN	Equivalent of the environment variable `PATH_TRANSLATED`.

cbTotalBytes	IN	Equivalent of the environment variable CONTENT_LENGTH.
cbAvailable	IN	Available number of bytes (out of cbTotalBytes) in the lpbData buffer. See the explanation of the Data Buffer that follows this table.
lpbData	IN	Pointer to a buffer, of size cbAvailable, which holds the client data.
lpszContentType	IN	Equivalent of the environment variable CONTENT_TYPE.

Note
For items that are referred to as "Equivalent of the environment variable...," a more detailed explanation of the particular environment variables can be found in this chapter in Table 25.6, "Variable Names and Purposes," and also in Chapter 3, "Crash Course in CGI."

The Data Buffer

Sometimes there's a lot of information sent to a program, and sometimes there's not. By default, lpbData will hold a 48K chunk of data from the client. If cbTotalBytes (the number of bytes sent by the client) is equal to cbAvailable, then the program is telling you that all the information that was sent is available within the lpbData buffer. If cbTotalBytes is greater than cbAvailableBytes, lpbData is only holding part of the client's data, and you'll have to ferret out the rest of the data with the ReadClient() callback function.

An additional possibility is that cbTotalBytes has a value of 0xFFFFFFFF. This means that there's at least four gigabytes of client data sitting out there, waiting to be read. What are the chances that you're going to receive that much data? If you do expect it, you'll be happy to know that you can keep calling ReadClient() until you get everything that's there. It's a good thing API functions are fast.

The Callback Functions

Throughout some of these other functions, there have been references to callback functions. These are the nice little hooks that let you get information that the server supplies you with. The functions are listed in Table 25.3, along with a brief description

of each. They are `GetServerVariable()`, `ReadClient()`, `WriteClient()` and `ServerSupportFunction()`.

Table 25.3. ISAPI DLL callback functions and purposes.

Function	*Purpose*
`GetServerVariable`	Retrieves connection information or server details
`ReadClient`	Reads data from the client's HTTP request
`WriteClient`	Sends data back to the client
`ServerSupportFunction`	Provides access to general and server-specific functions

GetServerVariable

`GetServerVariable` is used to return information that the server has in regards to a specific HTTP connection, as well as server-specific information. This is done by specifying the name of the server variable that contains the data in `lpszVariableName`. On the way to the server, the Size indicator (`lpdwSize`) specifies how much space it has available in its buffer (`lpvBuffer`). On the way back, the Size indicator (`lpdwSize`) is set to the amount of bytes now contained in that buffer. If the Size (`lpdwSize`), going in, is larger than the number of bytes left for reading, the Size indicator (`lpdwSize`) will be set to the number of bytes that were placed in the buffer (`lpvBuffer`). Otherwise, the number should be the same going in and coming out. Here are the details on how the `GetServerVariable()` function is defined:

```
BOOL WINAPI GetServerVariable (
    HCONN    hConn,
    LPSTR    lpszVariableName,
    LPVOID    lpvBuffer,
    LPDWORD    lpdwSize);
```

Tables 25.4 and 25.5 show the accepted parameters and possible error returns for `GetServerVariable()`, respectively.

Table 25.4. Accepted parameters for `GetServerVariable` callback function.

Parameter	*Direction*	*Purpose*
`hConn`	IN	Connection handle (if request pertains to a connection), otherwise any non-NULL
`lpszVariableName`	IN	Name of the variable being requested (See Table 25.6 for a list)

lpvBuffer	OUT	Pointer to buffer that will receive the requested information
lpdwSize	IN/OUT	Indicates the size of the lpvBuffer buffer on execution; on completion is set to the resultant number of bytes transferred into lpvBuffer

Table 25.5. Possible error codes returned if GetServerVariable returns FALSE.

Error Code	*Meaning*
ERROR_INVALID_INDEX	Bad or unsupported variable identifier
ERROR_INVALID_PARAMETER	Bad connection handle
ERROR_INSUFFICIENT_BUFFER	More data than what is allowed for lpvBuffer; necessary size now set in lpdwSize
ERROR_MORE_DATA	More data than what is allowed for lpvBuffer, but total size of data is unknown
ERROR_NO_DATA	The requested data isn't available

Use of the GetServerVariable() callback function requires knowing exactly what variables are available, and why you might want to get them A brief description of each of these variables can be found in Table 25.6 "Variable Names and Purposes," but a more detailed explanation can be found in Chapter 3, as they are also commonly encountered as CGI environment variables.

Table 25.6. Variable names and purposes.

Name	*Data Type*	*Purpose*
AUTH_TYPE	String	Type of authentication in use; normally either none or basic
CONTENT_LENGTH	Dword	Number of bytes contained in STDIN from a POST request
CONTENT_TYPE	String	Type of Content contained in STDIN
GATEWAY_INTERFACE	string	Revision of CGI spec that the server complies to

PATH_INFO	string	Additional path information, if any, which comes before the QUERY_STRING but after the script name
PATH_TRANSLATED	string	PATH_INFO with any virtual path names expanded
QUERY_STRING	string	Information following the ? in the URL
REMOTE_ADDR	string	IP address of the client (or client gateway/proxy)
REMOTE_HOST	string	Hostname of client (or client gateway/proxy)
REMOTE_USER	string	Username, if any, supplied by the client and authenticated by the server
REQUEST_METHOD	string	The HTTP request method, normally either GET or POST
UNMappeD_REMOTE_USER	string	Username before any ISAPI filter mapped the user to an account name (the mapped name appears in REMOTE_USER)
SCRIPT_NAME	string	Name of the ISAPI DLL being executed
SERVER_NAME	string	Name or IP address of server
SERVER_PORT	string	TCP/IP port that received the request
SERVER_PORT_SECURE	string	If the request is handled by a secure port, the string value will be one. Otherwise it will be zero.
SERVER_PROTOCOL	string	Name and version of information retrieval protocol (usually HTTP/1.0)
SERVER_SOFTWARE	string	Name and version of the Web server running the ISAPI DLL

ALL_HTTP	string	All HTTP headers that are not placed in a previous variable. The format for these is http_<header field name>, and are contained within a null-terminated string, each separated by a line feed.
HTTP_AccEPT	string	Semi-colon (;) concatenated list of all AccEPT statements from the client
URL	string	Provides the base portion of the URL (Version 2.0 only)

ReadClient

ReadClient does what you'd expect it to-it keeps reading data fromthe body of the client's HTTP request and placing it into a storage buffer. Just as GetServerVariable() and the other callback functions do, it uses the Buffer Size Indicator (lpdwSize) as an indicator to show how big the buffer (lpvBuffer) initially was, and how big it is after it's finished. ReadClient has only two possible return values, and no specific associated error codes; however, if the return of ReadClient() is True, but lpdwSize is 0, the socket closes prematurely. The following code shows how ReadClient() would be defined:

```
BOOL ReadClient (
    HCONN    hConn,
    LPVOID    lpvBuffer,
    LPDWORD    lpdwSize);
```

Table 25.7 details the expected parameters of the ReadClient() function.

Table 25.7. Expected parameters for ReadClient callback function.

Parameter	*Data Direction*	*Purpose*
hConn	IN	Connection Handle (cannot be NULL)
lpvBuffer	OUT	Pointer to buffer for receiving client data
lpdwSize	IN/OUT	Size of available buffer/number of bytes placed in buffer

WriteClient

WriteClient writes information back to the client frominformation stored in the buffer. The Buffer Size indicator (lpdwSize), in this case, functions as a record of how many bytes are supposed to be written to the client fromthe buffer, and how many were.

Since this might be used for binary data, it does not assume that the data will be in the form of a null-terminated string like the `ServerSupportFunction` does. A sample definition of the `WriteClient` function follows.

```
BOOL WriteClient(
    HCONN        hConn,
    LPVOID     lpvBuffer,
    LPDWORD     lpdwSize,
    DWORD     dwReserved);
```

Table 25.8 shows what parameters are accepted for the `WriteClient` callback function. (An additional reserved parameter is set aside for changes in the function's behavior that might be implemented in the future.)

Table 25.8. Expected parameters for the `WriteClient` callback function.

Parameter	*Data Direction*	*Purpose*
`hConn`	IN	Connection Handle (cannot be NULL)
`lpvBuffer`	IN	Pointer to data being written to client
`lpdwSize`	IN/OUT	Number of bytes being sent; number of bytes actually sent
`dwReserved`		Unspecified-reserved for future use

`ServerSupportFunction`

The final callback function, `ServerSupportFunction`, is one of the most powerful. It sends a Service Request Code to the server itself, which is a value that the server translates into a request to execute an internal function. An example of such a function would be redirecting the client's browser to a new URL, something that the server knows how to do without any help. This allows some standard operations to be performed, but it also gives server manufacturers a method for allowing developers to have easy access to a specialized internal function. Be it a built-in search routine, an update of user databases, or anything else, this function can call anything the server will allow. The actual list of what each server will allow varies, of course, but the definitions for the individual functions have a fixed order. Any service request code with a value of 1000 or less is a reserved value, used for mandatory `ServerSupportFunction` codes and defined in the `HttpExt.h` file. Anything with a Service Request Code of 1001 or greater is a general purpose server function, and should be able to be found in the servers own `*Ext.H` file. For example, Process Software's Purveyor maintains a `Prvr_Ext.h` file listing some additional supported functions, which have been included in Table 25.11. The `ServerSupportFunction` definition follows.

```
BOOL ServerSupportFunction (
    HCONN    hConn,
    DWORD    dwHSERequest,
    LPVOID     lpvBuffer,
```

```
    LPDWORD     lpdwSize,
    LPDWORD     lpdwDataType);
```

Tables 25.9 and 25.10 list the acceptable parameters and the standard defined values for service request codes for the `ServerSupportFunction`, respectively.

Table 25.9. Expected parameters in the `ServerSupportFunction` callback function.

Parameter	*Data Direction*	*Purpose*
`hconn`	IN	Connection Handle (cannot be null)
`dwHSERequest`	IN	Service Request code (See Table 25.10 for default values)
`lpvBuffer`	IN	Buffer for optional status string or other information passing
`lpdwSize`	IN/OUT	Size of optional status string when sent; bytes of status string sent, including NULL term
`lpdwDataType`	IN	Optional null-terminated string with headers or data to be appended and sent with the header generated by the service request (If NULL, header is terminated by `\r\n`)

Table 25.10. Defined values for standard service requests.

Service Request	*Action*
`HSE_REQ_SEND_URL_REDIRECT_RESP`	Sends a URL Redirect (302) message to the client. The buffer (`lpvBuffer`) should contain the null-terminated URL, which does not need to be resident on the server. No further processing is needed after this call.
`HSE_REQ_SEND_URL`	Sends the data to the client specified by the null-terminated buffer (`lpvBuffer`), as if the client had requested that URL. The URL cannot specify any protocol information (for example, it must be `/document.htm` instead of `http://server.com/document.htm`).

HSE_REQ_SEND_RESPONSE_HEADER	Sends a complete HTTP server response header, which includes the status code, server version, message time, and MIME version. The DataType buffer (lpdwDataType) should contain additional headers such as content type and length, along with a CRLF (Carriage Return/Line Feed (\r\n)) combination and any data. It will read text data only, and it will stop at the first \0 termination.
HSE_REQ_MAP_URL_TO_PATH	The buffer (lpvBuffer) points to the logical path to the URL on entry, and it returns with the physical path. The Size buffer contains the number of bytes being sent in, and is adjusted to the number of bytes sent back on return.
HSE_REQ_DONE_WITH_SESSION	If the server has previously been sent an HSE_STATUS_PENDING response, this request informs the server that the session is no longer needed, and it can feel free to clean up the previous session and its structures. All parameters are ignored in this case, except for the connection handle (hConn).

Table 25.11. Examples of server-defined acceptable service requests (Purveyor 1.1).

Service Request	*Action*
HSE_GET_COUNTER_FOR_GET_METHOD	Accepts the SERVER_NAME system variable in the Buffer (lpvBuffer), the length of SERVER_NAME in the Size buffer (lpdwSize), and returns the total number of GET requests served since initiation of the server, storing them in the DataType buffer (lpdwDataType)

HSE_GET_COUNTER_FOR_POST_METHOD	Except DataType (lpdwDataType), will hold the number of POST requests since startup of the server
HSE_GET_COUNTER_FOR_HEAD_METHOD	Except DataType(lpdwDataType), will hold the number of HEAD requests since startup of the server
HSE_GET_COUNTER_FOR_ALL_METHODS	Except DataType (lpdwDataType), will hold the total number of all requests since server startup

Note
Currently, there aren't many extra defined server functions in the public arena. But, given the rate of server expansion, and Microsoft's race to expand it's Internet Information Server, it wouldn't be surprising to see a large number of very useful functions show up in the near future.

As you've seen, an ISA is much like a traditional program that has a couple of added advantages, such as being able to get at the server program's insides, and taking advantage of things that might otherwise require either lots of file reading, or not be able to be accomplished at all. Next, however, you're going to take a look at something that's one step beyond that-adding onto the server functionality itself, to make it do whatever tricks you want it to.

Internet Server API Filter

ISAPI filters are quite different from a traditional CGI program If ISAPI DLLs make a server more flexible, ISAPI filters turn a server into a true contortionist, able to flex whatever way they need to. They're not just resident with the server, they're part of the server itself, having been loaded into memory and the server's configuration ahead of time. They're direct extensions of the server, allowing them to do tasks that no CGI program could think of doing, such as enhancing the local logging of file transfers, building in pre-defined methods of handling forms or searches, and even doing customized local authentication for requests. You're making the server evolve into something more powerful, instead of adding pieces that the server can call for help

When you create an ISAPI filter, you're creating a linked DLL that's being examined every time the server processes an HTTP request. The goal is to filter out specific notifications that you're interested in, and remove the duties of handling that

particular process from the core functionality of the server. You're essentially saying, "Oh, don't worry about that when you see it; that's what this filter is for." This scalability allows you to take a basic server and customize it to meet whatever needs you might have. You're adding scalability to the server so that it meets your needs, even if the original manufacturer didn't anticipate them

Like an ISAPI DLL, an ISAPI filter has two entry points that must be present in order to verify that the function meets the current specification, and that it has a place to receive information from the server. Unlike an ISAPI DLL, though, an ISAPI filter isn't something that's spur-of-the-moment in its use-any filters you define have to be entered in the system registry so that they are literally part of the server's configuration. Since they're intercepting things as they happen, the server needs to know about them In this case, it's convinced that it always had the ability to do these functions, it just never used them before.

Entry Point-`GetFilterVersion`

The first entry point to define in the ISAPI filter is the one that tells the server that it does in fact correspond to the correct specification version, so it should be able to run without difficulty. This is the `GetFilterVersion` function, which takes only one argument-a structure that will hold the data that the server uses to find out the version, the description, and the events that the filter wants to process. Here's how `GetFilterVersion` would normally be defined:

```
BOOL WINAPI GetFilterVersion(PHTTP_FILTER_VERSION pVer);
```

This points to a data structure of the `HTTP_FILTER_VERSION` type, which contains all the little niceties that the server is looking for. Since pointing to a structure that you don't know anything about isn't necessarily a good idea, look at what's inside an `HTTP_FILTER_VERSION` structure to understand what data the server really needs. A typical definition of an `HTTP_FILTER_VERSION` structure follows:

```
typedef struct _HTTP_FILTER_VERSION
{
    DWORD    dwServerFilterVersion;
    DWORD    dwFilterVersion;
    chAR        lpszFilterDesc[SF_MAX_FILTER_DESC_LEN+1];
    DWORD    dwFlags;
} HTTP_FILTER_VERSION, *PHTTP_FILTER_VERSION;
```

Table 25.12 points out the details of what those structure components are.

Table 25.12. Expected parameters for an `HTTP_FILTER_VERSION` structure.

Parameter	*Data Direction*	*Purpose*

dwServerFilterVersion	IN	Version of specification used by the server (Currently server-defined to HTTP_FILTER_revISION)
dwFilterVersion	OUT	Version of the specification used by the filter (Currently, server defined to be HTTP_FILTER_revISION)
lpszFilterDesc	OUT	A string containing a short description of the function
dwFlags	OUT	Combined list of notification flags (SW_NOTIFY_*) that inform the server what kind of events this filter is interested in knowing about; see Table 25.15 for the complete list of SW_NOTIFY_* flags available

Note

dwFlags is a parameter to be careful with-if you don't specify what you need, the filter won't do you any good. If you specify everything, your filter will drag down the server's performance and possibly cause other problems. Be picky with what you place inside.

Once the GetFilterVersion information has been transferred, it's time to call the main filter process itself-HttpFilterProc(). Just like the main() function in a C program, or the HttpExtensionProc used by an ISAPI DLL, HttpFilterProc() serves as the gateway to all that your filter is designed to do. Here's how the HttpFilterProc() would normally be defined:

```
DWORD WINAPI HttpFilterProc(
    PHTTP_FILTER_CONTEXT     pfc,
    DWORD            notificationType,
    LPVOID            pvNotification);
```

Table 25.13 explains the HttpFilterProc() function parameters.

Table 25.13. Expected Parameters for the HttpFilterProc function.

Parameter	*Purpose*

pfc	Pointer to a data structure of the HTTP_FILTER_CONTEXT type, which contains context information about the HTTP request.
notificationType	The type of notification being processed, as defined in the list of notification flags (SW_NOTIFY_*). See Table 25.15 for a complete list of SW_NOTIFY_* flags available.
pvNotification	The data structure pointed to as a result of the type of notification. See Table 25.15 (SW_NOTIFY_* flags) for the relationship between notification types and specific structures.

Based on the event, and what was done by the custom processes within the filter, HttpFilterProc() can yield a variety of different return codes, from "All Set," to "Keep Going," to "Whoops." Table 25.14 lists the accepted return codes and their explanations.

Table 25.14. Acceptable return codes for the HttpFilterProc function.

Return Code	*Meaning*
SF_STATUS_REQ_FINISHED	Successfully handled the HTTP request, and the server can now disconnect
SF_STATUS_REQ_FINISHED_KEEP_CONN	Successfully handled the HTTP request, but the server shouldn't necessarily disconnect
SF_STATUS_REQ_NEXT_NOTIFICATION	The next filter in the notification order should be called
SF_STATUS_REQ_HANDLED_NOTIFICATION	Successfully handled the HTTP request, and no other filters should be called for this notification type
SF_STATUS_REQ_ERROR	An error happened and should be evaluated
SF_STATUS_REQ_READ_NEXT	(Used for raw data only) Session parameters are being negotiated by the filter

One thing you might have noticed from Table 25.14 (Acceptable Returns) is that one of the possible returns is for the server to call the next filter in the notification order. You can have a whole sequence of filters all looking for the same event to occur, and

all standing in a line waiting to do something with the data. So, the first function might be an authorization log, while the next might be a document conversion, and the last would be some other customlogging function. As long as one of the earlier functions doesn't return a code saying "OK, shut it all down," then everyone else who's waiting for data will get their turn at it.

Since they have already been mentioned several times, now is probably a good point to examine the Notification flags (`SW_NOTIFY_*`). These serve a combination of purposes: they let the server know what kind of specific events are being looked for, and they can also specify that the filter should be loaded at a certain priority level. Depending on the type of notification, there are specific data structures that these functions mapto that hold the additional information the server might need upon receiving the specific notification. Table 25.15 shows the notification flags and their descriptions, as well as what specific data structures each notification corresponds to, where appropriate.

Table 25.15. Acceptable notification flags for `GetFilterVersion` and `HttpFilterProc`.

Notification	*Meaning*
`SF_NOTIFY_ORDER_DEFAULT`	(GFV only) Load the filter at the default priority (this is the recommended priority level)
`SF_NOTIFY_ORDER_LOW`	(GFV only) Load the filter at a low priority
`SF_NOTIFY_ORDER_MEDIUM`	(GFV only) Load the filter at a mediumpriority
`SF_NOTIFY_ORDER_HIGH`	(GFV only) Load the filter at a high priority
`SF_NOTIFY_SECURE_PORT`	Looking for sessions over secured ports
`SF_NOTIFY_NONSECURE_PORT`	Looking for sessions over nonsecured ports
`SF_NOTIFY_READ_RAW_DATA`	Let the filter see the raw data coming in; returned data consists of both data and headers Structure: `HTTP_FILTER_RAW_DATA`
`SF_NOTIFY_PREPROC_HEADERS`	Looking for instances where the server has processed the headers
	Structure: `HTTP_FILTER_PREPROC_HEADERS`
`SF_NOTIFY_AUTHENTICATION`	Looking for instances where the Client is being authenticated by the server

	Structure: HTTP_FILTER_AUTHENTICATION
SF_NOTIFY_URL_MAP	Looking for instances where the server is mapping a logical URL to a physical path
	Structure: HTTP_FILTER_URL_MAP
SF_NOTIFY_SEND_RAW_DATA	Let the filter know when the server is sending back raw data to the client
	Structure: HTTP_FILTER_RAW_DATA
SF_NOTIFY_LOG	Looking for instances where the server's saving information to its log
	Structure: HTTP_FILTER_LOG
SF_NOTIFY_END_OF_NET_SESSION	Looking for the termination of the client session
SF_NOTIFY_AccESS_DENIED	(New for version 2) Looking for any instance where the server is about to send back an Access Denied (401) status message. Intercepts this instance to allow for special feedback
	Structure: HTTP_FILTER_AccESS_DENIED

As you can see fromTable 25.15, there are a lot of possibilities for the filter to want to do something. There are also lots of pointers to Data Structures.

The two components of the ISAPI standard, one being Applications and the other Filters, make sense in a number of ways. CGI is a wide open field, with dozens of tools and languages available to help people create whatever they need. Since what's normally needed is more functionality, ISAs take CGI to that next step by providing the direct link between the server's internal functions and an external program Filters allow the programmers to modify the heart of the server programitself, and make it react differently for their unique situation. The power behind these methods is obvious, the pain of implementation tends to come out later.

Implementation Complications

The point that tends to detract fromall of the wonders of an API are the difficulties in creating an API-based function. You might have to use specific tools, or be limited to specific platforms once you build what you want. You need to know what you're doing

because a casual programmer is not going to write a very efficient or safe API function. Why? Because, unlike traditional CGI, where a new instance of the CGI programis run for each user, an API function is a shared resource. This is why it doesn't take upas much space in memory-there's only one of thematt a time.

To imagine just one of the possible myriad of problems that could cause an unprepared program, compare it to a small company with one phone. The phone rings occasionally, the receptionist picks it upand does what's necessary, then hangs up. If the phone calls don't come in too fast, there's no problem. As soon as the pace starts picking up, however, there had better be a systemin place to deal with multiple callers, or the "one person at a time" syndrome is going to grind the company into the ground.

How does that relate to an API-based function? Well, when you design a traditional CGI program, it follows a very linear path. It accepts data, and then goes down a one-way street. It calls subroutines and other organized functions, but it's only dealing with one user, or one streamof data.

Programming Considerations

The biggest caveat about an ISAPI function, be it a filter or a DLL, is that it has to be multithread safe. A multithreaded environment is one where lots of things are running at once, requests come in bunches, and every function has to be careful not to stepon another function's toes while they use a file or take data froma block of memory. Coding something to be multithreaded can be a challenge, especially for more complex functions. If you're an advanced C programmer, then you've probably already dealt with this issue before, and you've been anticipating it ever since you started reading this chapter. If you're a novice programmer, or even an intermediate programmer, you'll want to delve deeper into the many resources out there that cover multithreaded DLL programming. Microsoft maintains a great deal of these resources on their Web site (`www.microsoft.com`), and they're also available on their Development Library, and in general programming references.

Caution
If you write an ISAPI function that's not multithread safe (or safe in general), you're risking a lot more than just some errant data. As mentioned in the beginning of this chapter, ISAPI functions are resident within the same process space as the server itself. That's right-they all live together in harmony. If one crashes and burns, they other is toast as well. The likelihood of a commercial server product like IIS or Purveyor going ker-blooey isn't very high, due to the amount of testing performed on them, so in most cases the weak link can be your ISAPI function. Don't make your server suffer-test early, test often, and get helpto do it if necessary.

There is hope for developers in that there are a number of ways to create an ISAPI function, including one that might not be thought of at first glance-Visual Basic. Microsoft has announced that upcoming releases of Visual Basic will support a number of functions, including the ability to create shared DLLs. Talk about making your life easier...

Debugging ISAPI DLL programs

Like any programming endeavor, you're going to hit the debugging phase. ISAPI functions could place your server in danger if not well-tested. Therefore, chances are pretty good you'll be looking at those functions in debug mode for quite a while. The problems that Web servers present in debugging some of these components, however, can cause a bit of frustration.

To cover the standard CGI debugging methods using an ISAPI DLL, you'll want to review Chapter 6, "Testing and Debugging." This covers some of the methods available, but it's important to note that one of the focuses of debugging a regular CGI application is setting the Environment Variables to control the data. Since the ISAPI DLLs use Extension Control Blocks, Environment Variables are no longer in the picture. You can't just set themwith a shell script or regular executable. In fact, you really can't set themat all. But wait, there's more.

Web servers on the NT platformnormally run as a background service-they start up when the systemboots, and go on their merry little way behind the scenes. Unless something goes horribly wrong, they just sit there and process, and things magically work. Unfortunately, to be behind the scenes means that there's no window that they toss messages and other output out of. Therefore, you don't get nice visual recognition of things going on as they process. Before you get nervous about that, though, look at what can be done to helpyour debugging efforts.

When most servers go ahead and load your ISAPI DLL, they're going to cache it when they're through. If you're making changes to your DLL, and wondering why in the world they're not showing up, this can certainly be a cause of it. By editing the registry, you can disable caching for the DLLs, making sure the current copy is loaded each time. This makes the execution of those functions slower, so make sure you turn it back on once you're done. Here's how you can disable caching under Microsoft's IIS:

Disabling ISAPI DLL caching with Microsoft's IIS

1. Run the 32-bit registry editor (`REGEDT32.EXE`)
2. Select the `HKEY_LOCAL_MAchINE` window
3. Select the systemfolder
4. Select the `CurrentControlSet` folder

5. Select the services folder
6. Select the `W3SVC` folder
7. Select the parameters folder
8. Set `CacheExtensions` to 0

To re-enable caching, go through the same process, but set `CacheExtensions` to one. Note that this only applies to ISAPI DLL's with Microsoft's Internet Information Server (IIS). Process Software's Purveyor 1.2 and previous versions do not have a similar option for cache manipulation, and ISAPI Filters can't be controlled in this manner. If you want to replace a filter, you're going to have to shut your server down, replace the filter, and start the server up again. Good thing servers don't (normally) take a long time to come up.

For real-time debugging, you'll probably first turn to the lowly message box. Although not the most elegant method in the world, it's simple. You place a message box within your code to show you data from the ECBs, or as the result of a function call. Then slowly wind your way through your program's execution. Remember, though, that your program is running as a service, so it doesn't have its own window to play with. What you'll want to take advantage of are the `MB_SERVICE_NOTIFICATION` and `MB_TOPMOST` flags for the message box, which are defined in the Win32 SDK. These are the flags that let those messages come through when the service has something horrible to report to you and needs its own place on your screen.

Tip
Don't leave Message boxes in the final version of your program! They're only for debugging purposes. You're not supposed to have any bugs left when you're done, so why keep them around? If one does pop up, it's better to use the built-in logging functions to write it out to a file and continue on, rather than pop up a message box that no one might be around to see.

Log files are your friends as well. Without a lot of preparation you can insert logging functions in either your ISAPI DLL or ISAPI Filter, and look at a nice printout of what happened. Standard File I/O will create the files so you don't have to rely on building an extended logging function within the ISAPI code set itself. Standard `ReadFile` and `WriteFile` will work just fine as long as you've got written permissions for the directory in question (the system root). Logging is great for verifying data coming in and going out as part of your testing process. If someone else is doing the testing for you it might be more suitable to have a written testing record generated this way, rather than relying on message boxes or the next method.

The next method is a slightly more involved method for debugging. You can use the

`OutputDebugString()` function, which sends a string to your specified Debug monitor. If you don't have a Debug monitor, or don't know what one is, this might not be the best choice. If it sounds like fun, though, Microsoft includes a `DBMON` example with the Win32 SDK to show how this is generally done in code.

Tip
According to Microsoft, early versions of the Win32 SDK (for instance, prior to the Win32 SDK for NT 4.0 Beta 2) need a code change in DBMON to account for user privileges. If you have one of these earlier versions, you'll want to take a look at the code change they specify in Knowledge Base Article Q152054, page 2, under "`OutputDebugString()`." This is available from Microsoft's Web site.

Other methods of debugging ISAPI DLLs and Filters are constantly evolving, as more and more people work on making them For some further details, check Microsoft's Online Knowledge Base during your development cycle, and check for any updates regarding helpful additions (or new problems) to the ISAPI debugging process. One of the current Knowledge Base articles on this topic is Q152054, which provides the details described above, as well as a few other options that you might like to pursue.

Platform Considerations

Currently, ISAPI is not extremely cross-platform capable. It will work with all flavors of NT, but only ISAPI DLLs can, at this point, go to any other platform In this case, that additional platform is OpenVMS, made possible by Process Software's architecture. ISAPI filters are NT-only, and right now they're only to be found with Microsoft's Internet Information Server (IIS) package. If you're thinking about going to UNIX, you might want to talk to the folks at Microsoft-a recent document published by them on their Internet strategy says:

"In addition, Microsoft continues to work closely with its key UNIX partners (Software AG, Bristol, Spyglass, and Mainsoft) to deliver the server and client technologies for the various UNIX platforms."

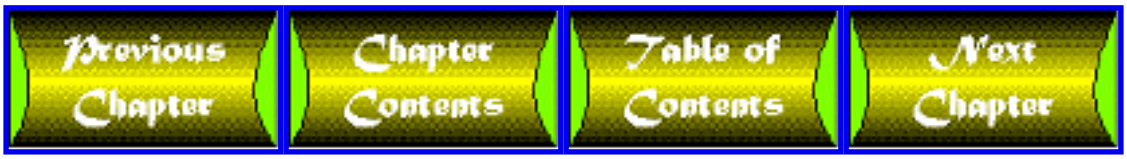
Of course this doesn't say when, but, with the aggressive efforts Microsoft is making to take a big stake in the Internet market, you can bet that they're not going to sit idly by while other standards, like the Netscape Server API (NSAPI), hold dominance in Internet-intensive markets like UNIX.

Note

> See Chapter 26, "NSAPI," for details on the Netscape Server API (NSAPI) and its cross-platform capabilities.

Summary

You can do almost anything with the ISAPI. The big questions to ask yourself are: Does my Server support ISAPI in either form? Do I have the programming experience to create an API function? Is this a function I could do with standard CGI programming and, if so, what are the real benefits I gain from doing it this way? In many cases you might discover that something limits you from using ISAPI for your needs, but as Web sites continue to evolve, some of the more sophisticated functions within servers will be accessible only to API calls. In addition, building value-added functions for established server packages is an area that has not yet begun to reach saturation of developers. The power and flexibility to expand and exceed what you're doing now with CGI exists. All you have to do is want to take advantage of it.



Chapter 26

NSAPI

CONTENTS

What do you do if your server can't do what you want it to? Buy another server? Not do what you were planning on? Even CGI programs can't really change the way a server works; they can only add on specific functions that need to be called in a specific way. Another way does exist, however.

The Netscape Server API, or NSAPI, enables you to add functions to your server, like CGI, but it also enables you to change the way the server works at the very core of its

functionality. Don't like the way errors are handled? Change the error handling. Not enough information being logged? Change the logging system Want to add your own customauthorization mechanism? Go right ahead. The power exists for you to change almost any function that the server performs, as well as add whatever pieces you want.

In this chapter, you explore the world of the NSAPI and see what it means to Netscape as well as to you, the developer/user. The primary focus areas are as follows:

- Why NSAPI?
- NSAPI versus CGI
- NSAPI and the server's processes
- Functions and features
- Implementation complications
- Future directions

Let me give you advance warning: creating and using NSAPI functions require a thorough knowledge of C programming. Some of the information contained in this chapter may be of more use if you have a programming background, as I cover data structures, function calls, and general programming issues. The real focus of this chapter, though, is to provide you with a detailed overview of the NSAPI itself so that you have a starting point if you want to pursue it further.

Caution
Before you spend too much time learning about the NSAPI standard, make sure that your server supports it. It is native only to Netscape servers, and there are certain other server packages and versions that have added, or are adding, NSAPI support. You wouldn't want to create a great function only to find out it can't be of any use to you.

Why NSAPI?

When building their servers, Netscape couldn't possibly anticipate exactly how you would want to work. They could and have built in a great deal of flexibility to enable you to performmost of the tasks you want, but you will always have a set of functions or a way of doing business that is unique to your situation. Support for the open CGI standard allows easy access to functionality to extend the server's reach, but in a world where things need to be faster, more flexible, and more seamlessly integrated, easy access is often not enough.

Netscape has a vision, and that vision is the Internet Application Framework. Sounds all-encompassing, doesn't it? Well, it is. The overall focus of Netscape's efforts is to create a

set of open standards and protocols that any developer can use to get the functionality that he or she needs for Internet and intranet applications. As part of Netscape's *Open Network Environment (ONE)* philosophy, the Internet Application Framework and NSAPI, specifically, play pivotal roles in defining the next generation of applications. Figure 26.1 shows Netscape's representation of this Internet Application Framework. The figure also shows that in Netscape's vision NSAPI and CGI (along with other technologies) reside in the category of Server APIs (Application Programming Interfaces).

Figure 26.1: *Netscape's representation of their Internes Application Framework.*

Server APIs are the methods that you can use to extend your server's functionality. An API is nothing more than a way to get at special bits of data and exert an amount of control over the server itself, through a variety of methods.

Unlike CGI, whose open standard is supported by almost every server in existence, no open standard currently exists for Web servers. This lack of standard presents a problem, and it goes rather deep-each server is different, and its insides operate in different ways. Without forcing every server developer to adopt the same methodology in how he or she processes data internally, no common ground can be had for every server to take advantage of. Although this situation might present a problemfor some companies, it's just another opportunity for Netscape.

With its broad base of users, Netscape can influence Internet trends, including HTML, security issues, and servers themselves. Most of this influence is through just adopting what Netscape feels is a good method of doing something and going with it. In a few cases, arbitrary additions meet with unanimous approval (frames, for instance), but Netscape has enough momentumto avoid being bogged down by a need for everyone to endorse what they've designed. Much like Microsoft often says, "This is the standard that software must conformto." Because they control a great percentage of the operating systems market, Netscape has a similar kind of power in the Internet environment, and it is due to their setting the trends, not following them

The NSAPI is Netscape's next venture into trend-setting standards, though it will be a much more difficult task. The first reason is, as I mentioned before, that it involves having the server work in a specific way. The second is that it requires a reasonably high level of expertise to create a useful and robust API function. The final reason is the one that may be the biggest obstacle: Netscape is not alone in their proposal for a server API standard. The challengers are numerous, but the biggest competition comes fromone source-the combination of Microsoft and Process Software to create the Internet Server API (ISAPI).

Note

> Later in this chapter I cover in more detail the future of the NSAPI amidst these challenges. You can also review the ISAPI in Chapter 25, "ISAPI."

Why, then, with all these challenges, would anyone pursue the creation of NSAPI functions? To begin to understand that question, you must start by comparing NSAPI functions to the CGI standard to understand where they're similar and where they part company.

As you can see from the diagrams shown in Figure 26.1, many other components fit naturally into the creation of Internet applications, but the role of NSAPI itself is the focus of attention here. Both CGI and NSAPI provide additional server-side functionality; that is, they isolate data processing from the client for convenience, security, and/or speed. The key difference is that a CGI program is built to take advantage of data gathered by the server and operate outside the boundaries of the HTTP server's environment, whereas NSAPI functions are built directly into the server itself, extending its core functionality to meet individual or corporate needs.

NSAPI versus CGI

Of all the server software produced anywhere in the world, only a small fraction provides no support for the CGI standard or some closely related independent extension. The reason is that CGI makes sense. You can't expect the server manufacturer to know everything you want to do with the software, and often you wouldn't want the manufacturer to cramp your style even if it could predict some form of what you wanted.

As you've seen throughout this book, using CGI can be an easy and powerful way to extend the server's functionality without too much fuss.

Performance

API functions are faster than CGI functions. How much faster? They are anywhere from two to five times as fast, depending on what you're doing. Knowing how important it is to get work done as fast as possible on the Internet, this speed is a good thing.

Process Space

API functions share process space with the server itself. As a result, these functions are faster. Every time a server executes a CGI program, on the other hand, a new process is started up for that program, and it doesn't like to share with anybody. That's one process per CGI execution. Imagine how much server effort that process causes if your

server gets tens of thousands of CGI hits a day. And people wonder why server's need so much memory.

If you look through a CGI script, you'll see that it's just not designed to be nice to other applications that may need extra memory or processor time. Each script wants to do its task as fast as possible and get out of there. Although this capability may not be bad for most functions, think of database functions. Say you have a database with 400,000 records in it, and you're doing a search off a subset within your CGI script. Every time you want to do some Inter-Process Communications to whatever manages your data source, you're in for a long haul. API functions, on the other hand, share resources of any kind and can coexist peacefully whether they're dealing with an external or an internal resource.

Data and Function Access

The largest advantage that an API function has over a CGI program is the amount of data and number of functions that the server has available but can't be accessed from "outside the loop" of its own process space. CGI is designed to receive data and send data in a limited fashion through intermediaries (environment variables and STDOUT, normally). API functions, however, are part of the server itself and can cause the server to take some action or intercept some action that the server might normally do.

Suppose that you have a page that requires authorization, and someone within your company who isn't authorized tries to access it. You could intercept the error message that would normally go back, identify who the user is, and then present more appropriate feedback, such as "Sorry, Bob, you don't currently have access to the Technical Specifications for that product. The contact person for your department's questions is Janet, who can be reached at extension 58. Your regional contact is Joe, who can be reached at 555-0101 for any questions when Janet isn't available." As you can see, this message is a lot more helpful than the `Access Denied` message. By building this functionality as an additional server function, all your company's servers can easily make use of similar functionality, and it will be transparent to the users and even to some of the administrators.

The preceding example is just a sampling of what you can do, and without having much impact on your server's performance at all. You can create customized logging entries, security functions, reporting, automatic document conversion, and even "cookie"-like information for maintaining user states while they're accessing your pages. The functionality you add is all up to you, your needs, and your imagination.

To help you better understand the ways in which the NSAPI enables you to get better acquainted with your server, I delve into some of the functions and structures that make the NSAPI what it is in the following sections.

NSAPI and the Server's Processes

NSAPI works by acting in place of, or in addition to, specific server functions. By crawling around inside your server's configuration files and changing what things are being done, you can rebuild the server in any way you want. Not only are you customizing it to meet your needs, but you're also learning how it worked in the first place.

HTTP Request/Response Process

The server functions that you're adjusting to your own purposes take place in a specific order, starting as soon as the client sends a message that says, "I'd like this file." This process is called an *HTTP Request/Response process*; it's the series of tasks that occurs once the client has sent data, and the duty rests with the server to complete the exchange of information.

To come to grips with how Netscape servers treat this whole process, look at Netscape's own definition of what the HTTP Request/Response process looks like, as shown in Table 26.1.

Table 26.1. Netscape's HTTP Request/Response process.

Process	*Purpose*
Authorization Translation	Any client authorization data converted into user and group for server
Name Translation	URL translated or modified, if necessary
Path Checks	Local access tests to ensure document can be safely retrieved
Object Type Check	Evaluate the MIME type for the given object (document)
Response to Request	Generate appropriate feedback
Logging of Transaction	Save information about the transaction to logging files

Server Application Functions

To get from the beginning to the end of the whole Request/Response process, each one of these steps has its own internal server functions, called *server application functions*. These internal functions, which are known to the server, help it do its job. These functions can be separated into classes, called *function classes*, to best describe what each internal function relates to and to help with organizing the design process. Each function class relates to one or more of the processes that take place when the server answers a client's call for data. Though Netscape's breakdown of the HTTP Request/Response process has six steps, only five classes are used because one serves double duty. Table

26.2 lists these classes in order of execution and tells to which of the six steps shown in Table 26.1 they map.

Table 26.2. NSAPI function classes.

Class	*Function*
`AuthTrans`	Performs Authorization Translation
`NameTrans`	Performs Name Translation
`PathCheck`	Performs Path Checks
`ObjectType`	Performs Object Type Check
`Service`	Performs Response to Request and Logging of Transaction

Note
For more detailed information on each of these functions, including what types of response codes and errors they can generate, see "Functions, Variables, and Their Responses" later in this chapter.

Controlling Function Use

At every step along the way, the server needs to know what functions take priority and what additional functions are available. Although the function classes have their own special order of working, control within each class of function is administered by the server's configuration files. To make use of your own function, you need to know how to define the function in these configuration files and what the configuration files themselves are.

Function Declaration

All server application functions, regardless of what they do, get referenced in the same way. They let the server know what function class they belong to, what the name of the function is, and which values need to be passed to the function itself. An example of this kind of function declaration is as follows:

```
class fn=functionname value1=data1 .. valueN=dataN
```

You may also see `directive` substituted for `class`. You say potato, they say potato.

Configuration: UNIX

When you configure a UNIX-based Netscape server to use a function, you do so through the use of two files: magnus.conf and obj.conf.

magnus.conf

The magnus.conf file is the server's master controller. It contains the instructions and directives that the server has to take into account when it first starts up. The magnus.conf file sets up, for example, the server's name, port number, and what functions to load into memory. When you develop an NSAPI function, magnus.conf is the starting point for making sure that your function is available for use. You specify that you want the server, on initialization, to load the module that you've created which holds your function. You also give it any aliases that might be used for your function, as in the following example:

```
Init fn=load-module shlib=/usr/me/mymodule.so funcs=myfunction
```

`Init` specifies that this process is to be performed upon initialization. Next, you instruct the server to use its default `load-modules` function to load your module, and you provide the full path to that module within the `shlib` (shared library) parameter. The `mymodule.so` file is a shared object, which I discuss in the "Functions and Features" section. Finally, you provide the alias (or aliases) for your function so that you can reference it later.

obj.conf

When the server is running, obj.conf is in command. All requests that come to the server get analyzed through the order and method specified in obj.conf to determine what should happen. The breakdown order of the HTTP Request/Response process that you examined in Table 26.1 occurs here. Your particular function is specified somewhere in the obj.conf, depending on what class of function it is and what you want it to do, and the server goes through each function in class order and then function order to see what should happen.

One example of this process is a function supplied by Netscape; it takes a URL path as a value and then maps that value to a hard-coded path on your system. Essentially, it creates a symbolic path link. This function's entry in the obj.conf file follows the basic function declaration:

```
NameTrans fn=makepath path=/usr/mine/stuff
```

Here the function is in the `NameTrans` function class, because it should occur as the server is mapping file locations. The function itself is called `makepath`; it accepts only one value, a hard-coded path.

Configuration: NT

Windows NT relies on a different mechanism for controlling server processes, but the basic premise is still the same. Different sections in the Windows NT registry correspond to the purposes of the magnus.conf and obj.conf files, so you just make the entries there instead.

Caution
Be extremely careful when doing anything to the Windows NT registry. Whereas messing up an obj.conf or a magnus.conf file would only create problems for your server, the NT registry controls all actions on your system. You could ruin your whole week if you accidentally delete or modify something. Before you modify anything, make a backup copy of the registry for safekeeping, or make a system disk that can restore your current configuration.

You modify the Windows NT registry by running the regedt32.exe program. You should look for the `HKEY_LOCAL_MAchINE\Software\Netscape\Httpd-80` section. Remember, this is your entire system configuration, so be careful!

Start Up Key

Under the `Httpd-80` section is `CurrentVersion\StartUp`, which controls the startup processes of the Netscape server. This controls what functions are loaded into memory when the Netscape server runs as an NT service. To add your own entry, you create a new key in the StartUp folder by choosing Edit | Add Key and then entering `InitFunction01` (or `InitFunction02` if `InitFunction01` is already there) in the Key Name value, leaving the Class entry blank for now. You then add values to the key, as shown here:

```
fn: load-modules
shlib: c:\netscape\stuff\mymodule.dll
funcs: myfunction
```

Here you're specifying that the server use its own `load-modules` function to place the shared library (`shlib`) of `c:\netscape\stuff\mymodule.dll` into memory. Files with a .dll extension are Windows Dynamic Linked Libraries (DLLs), which allow their functions to be shared by other processes on the system. You're also telling the server that the alias for this particular function is `myfunction`.

Directive Keys

To convince the server that your function should be called, now that you've specified that it should be loaded, you need to determine where your function fits into the general scheme of things. To start with, go into the `CurrentVersion\Objects` registry folder and check through the objects listed to see which one of themhas the `name: default` value.

Next, you need to look under that object, in its directive keys, to find the Directive (class) under which your application falls. If your function is supposed to take place as a logging request, for example, you should see if you can find a directive key with a value of `Directive: Addlog`. If one doesn't exist, you can add it.

After you find (or create, which is less likely) the directive key your function should be part of, you need to add a new function underneath that Directive folder. This time, you just specify the function and any necessary values. A simple case would be just the function itself, as shown here:

```
fn: myfunction
```

Here you're just saying, "When you run through this groupof functions, be sure to call `myfunction` as well!"

Initializing the Server

After you make any changes, regardless of which platformyou make themon, you need to shut down the server and restart so that it loads the new functions. Sending the server a Restart signal isn't enough because the signal is not going to cause the server to load the new functions specified in either magnus.conf or the StartUpregistry key. Make sure that you completely shut down the server and then start it back upagain.

One thing to keepin mind is that certain modules may need to be explicitly instructed to stopwhat they're doing before the server starts upagain; otherwise, you could end up with two instances of a process running, wasting space and even causing conflicts. To clean upthose processes, you can use the atrestart() function, defined as

```
void magnus atrestart(void (*fn) (void *), void *data)
```

This is called by the server during restart, not during termination, with the data pointer being passed in as the argument for the function call.

Functions and Features

To create an NSAPI function, you need to know what it's built of-what data structures and general functions are necessary and available to make your idea for a function

become reality.

Server Application Function Prototype

Just as all server application functions are defined the same way in the configuration files for how they're accessed, all the functions are defined the same way in your actual code. This ensures that they are compatible with the other processes the server is performing and that they can access the server's data in a timely manner. The prototype that follows is Netscape's required definition for each function:

```
int function(pblock *pb, Session *sn, Request *rq);
```

Response Codes

To determine the outcome of the function, the integer return from the function must correspond to the available response code, as listed in Table 26.3.

Table 26.3. Acceptable server application function response codes.

Code	*Value*	*Definition*
`REQ_PROCEED`	`0`	The function has performed its task; proceed with the remainder of the request.
`REQ_ABORTED`	`-1`	An error occurred; the entire request should be aborted at this point.
`REQ_NOACTION`	`-2`	The function didn't accomplish what it wanted to do, but the request should proceed.
`REQ_EXIT`	`-3`	Close the session and exit.

Depending on the class of function, different interpretations for the returns are available, as you learn in the section "Functions, Variables, and Their Responses."

Parameter Blocks

The parameter block (`pblock`) data structure is the amino acid of NSAPI functions. Because servers deal with information based on `name=value` pairs, `pblock` is a hash table that is keyed on the name string, which then allows the function to map names to values.

The data structures used when dealing with parameter blocks are shown in Listing 26.1. The `name=value` pairs are stored in the `pb_param` structures, which are in turn used inside `pb_entry` to create linked lists of the `pb_param` structures. The hash table itself (`pblock`) is defined in Listing 26.2, though it is subject to change by Netscape and is

normally transparent to most functions.

Listing 26.1. Parameter block structure definition.

```
#include "base/pblock.h"

typedef struct {
    char *name,*value;
} pb_param;

struct pb_entry {
    pb_param *param;
    struct pb_entry *next;
};

typedef struct {
    int hsize;
    struct pb_entry **ht;
} pblock;
```

Listing 26.2. `pblock` hash table sample definition.

```
#include "base/pblock.h"

/* Create parameter with given name and value */
pb_param *param_create(char *name, char *value);

/* Free Parameter if not null, Return 1 if non-null, 0 if null*/
int param_free(pb_param *pp);

/* Create new pblock of Hash Table size 'n' */
pblock *pblock_create(int n);

/* Free defined pblock and entries it contains */
void pblock_free(pblock *pb);

/* Find entry with given name in pblock */
pblock *pblock_find(char *name, pblock *pb);

/* Return value of pblock with given name found in it */
char *pblock_findval(char *name, pblock *pb);

/* Find entry containing name in pblock, and remove it */
pblock *pblock_remove(char *name, pblock *pb);

/* Create new parameter, insert into specified pblock */
pb_param *pblock_nvinsert(char *name, char *value, pblock *pb);
```

```
/* Insert a pb_param into a pblock */
void pblock_pinsert)pb_param *pp, pblock *pb);

/* Scan the string for name=value pairs */
int pblock_str2pblock(char *str, pblock *pb);

/* Place all the parameters in pblock into string */
char *pblock_pblock2str(pblock *pb, char *str);
```

Not all the functions are needed for server application functions, but the preceding listings show which functions are currently defined. Before implementing any of these functions, however, check the latest version of your server documentation, which contains more specific details on each one of these functions, their use, and their current state.

Sessions

A *session*, by Netscape's definition for the NSAPI, is the time between the opening and the closing of the client connection. To hold the data associated with the session in question and make it available system-wide, a `Session` data structure is needed, as outlined in Listing 26.3.

Listing 26.3. Sample `Session` data structure.

```
#include "base/session.h"

typdef struct {
    pblock *client;

    SYS_NETFD csd;

    netbuf *inbuf;

    struct in_addr iaddr;
} Session;
```

The `client` parameter block points to two more pieces of information to help identify the session uniquely: the IP address of the client machine and the resolved DNS name of the client machine. Information in `csd` and `inbuf` is relevant to the socket descriptor for the connection, whereas the `iaddr` structure is for internal use and contains raw socket information.

Request Structure

When the client makes a request, various HTTP-related information is stored, just as it is during normal CGI operations. All this information is accessible through the `Request` data structure, as outlined in Listing 26.4.

Listing 26.4. HTTP `Request` data structure.

```
#include "frame/req.h"

typedef struct {
    /* Server's working variables */
    pblock *vars;

    /* Method, URI, and Protocol specified */
    pblock *reqpb;

    /* Protocol Specific Headers */
    int loadhdrs;
    pblock *headers;

    /* Server's Response headers */
    pblock *srvhdrs;

    /* Object Set constructed to handle this request */
    httpd_objset *os;

    /* Last stat returned by request_stat_path */
    char *statpath;
    struct stat *finfo;

} Request;
```

Contained within the `Request` structure are several sources of data that your function can take advantage of, depending on what function class your function belongs to and what you're looking for.

The `vars` parameter block contains function-specific variables, which are different for every function class. In the section "Functions, Variables, and Their Responses," you will look in more detail at what the possible values can be.

One of the first things your function comes in contact with is the `reqpb` parameter block because it contains the data you first need to evaluate, as outlined in Table 26.4.

Table 26.4. Request parameters contained in `reqpb`.

Parameter	*Purpose*
`method`	HTTP method used to initialize the request; equivalent to the `REQUEST_METHOD` variable
`uri`	The URI that the client requests
`protocol`	The HTTP protocol version that the client supports
`clf-request`	The first line of the client's request to be used for logging or other similar purposes

The `headers` block is just what you would expect-the headers sent by the client. If more than one value is sent for the same header, they are joined together with a comma, as follows:

```
Header: value1, value2
```

Tip

Netscape recommends that you do not access the pblock for headers directly but instead use the following function:

```
int request_headers(char *name, char *value,
Session *sn, Request *rq)
```

Even though you can access the pblock directly, that capability may change in the future. Because Netscape doesn't want you to create code that won't work later, they have made this recommendation.

The data contained in `srvhdrs` is just the reverse of the headers block: this is the place where you can specify headers to be sent back to the client for the request result.

The last three parts of Listing 26.4 (`os`, `statpath`, and `finfo`) are used by the base server itself and are basically transparent to your application. They essentially verify the status of a given path, returning a `stat` structure if it's successful or an error code if it isn't.

Functions, Variables, and Their Responses

With each of the application function classes, specific data is evaluated, and certain response codes are valid. In the following sections, I help you review each one of the main classes in the order that the server handles them, and then I cover what's

available with each one for both variables and response codes. Following that, I provide further information on some other common functions that are available to your applications.

AuthTrans

`AuthTrans` decodes any authorization type data sent by the client so that it can compare the data to internal tables and determine the validity of the user. If the user is verified, `AuthTrans` makes the following data available in a `vars pblock`:

- `auth-type`: Defines the authorization scheme to which it complies. Currently, the only possible value is `basic`. (The CGI variable equivalent is `AUTH_TYPE`.)
- `auth-user`: Provides the username that has been verified. (The CGI variable equivalent is `AUTH_USER`.)

`AuthTrans` returns one of the following three response codes, with the following meanings:

- `REQ_ABORTED`: Abort the process (error).
- `REQ_NOACTION`: No authorization took place.
- `REQ_PROCEED`: Authorization occurred; continue.

NameTrans

`NameTrans` converts a virtual path, such as `/stuff/docs` into the absolute directory path, such as `/usr/bin/netscape/docs/stuff/docs`.

`NameTrans` functions expect to receive two variables in the `vars pblock` on execution: `ppath` and `name`. `ppath` is the partial path, that is, the virtual path that was supplied such as /stuff/docs. It may have already been partially translated, and your function can modify it, no matter what it decides to return. The `name` variable specifies additional server objects (besides `default`) that should add their list of things to do to this process.

Return codes froma `NameTrans` function can be any of the following:

- `REQ_PROCEED`: Translation was performed; no more translation should occur.
- `REQ_ABORTED`: An error should be sent to the client; no other functions should execute.
- `REQ_NOACTION`: The function doesn't admit translating `ppath`, though it may have changed it anyway. The rest of the functions should be carried out.
- `REQ_EXIT`: An I/O error occurred, and the process should be aborted.

PathCheck

`PathCheck` functions verify that a given path can be safely returned to the client, based

on authorization, URL screening, and other similar checks. The only information supplied to the `PathCheck` function is the `path` variable, which specifies the location to be checked.

For return codes, anything other than `REQ_ABORTED` is considered to be a success.

ObjectType

`ObjectType` functions are supposed to locate a `filesystem` object for the path supplied. MIME type checks and similar functionality are handled here. If no objects can be matched to the path in question, this function returns an error. The `path` variable is the only one passed to `ObjectType` functions, like `PathCheck`, and returns other than `REQ_ABORTED` are considered a success.

Service

Service functions are the methods that the server uses to reply to the client's initial request. Usually, the response is automatically generated based on the type of file being sent back, its location, or the authorization of the client for the request in question. Server-Side Includes are parsed before being sent in this stage of execution, whereas other files just go on their merry way.

Because Service functions have to take the initiative for sending the information back to the client, they need to initialize the response process with the following function call:

```
int protocol_start_response(Session *sn, Request *rq);
```

Return codes for the Service class functions can be any of the following:

- `REQ_PROCEED`: A response was sent to the client with no problems.
- `REQ_NOACTION`: A Service function did nothing; the server should execute one of the subsequent Service directives.
- `REQ_ABORTED`: Something bad has happened, and the server should direct an error to the client.
- `REQ_EXIT`: The Session should be closed.

General Functions

You can use many different functions. The current printed list is 61 pages, including details. They range from the mundane `MALLOC` (a cross-platform substitute for the `malloc()` function in C) to the following fun function (which checks for URIs containing `../` or `//` and returns `1` if they do or `0` if they don't):

```
int util_uri_is_evil(char *t);
```

Whatever task you want to perform, you can choose from literally dozens of functions, because you are at the base level of the server itself-anything that it does, you can do as well. Most of the functions that you will need for your server will likely involve reading data from the client, such as `pblock_findval()` for grabbing the request method, or sending back messages, such as the `protocol_status()` for sending server error codes to the client.

The complete list is available as part of the NSAPI Development White Papers and Technical Notes, all of which you can find at http://help.netscape.com. In addition, developers who are part of the Netscape Developer's program can obtain further information through the Developer's Technical Library. For further details, visit http://developer.netscape.com to see what's available to the general public and what's available to registered developers.

Error Reporting

Reporting when problems occur is important. As I just mentioned, the `protocol_status()` function enables you to send back information to the client in the form of a traditional server error code. The syntax for the function is as follows:

```
void protocol_status(Session *sn, Request *rq, int n, char *r);
```

The `Session` and `Request` structures establish which user session and for which request the error is being generated, to be sure it goes to the right place. To specify which error you want to send back, you can use the `r` string to enter your own specific reason that the process encountered an error, and the `n` int to specify one of the available error codes listed in Table 26.5. If you do not specify a string in `r`, the string sent to the client defaults to `Reason Unknown`.

Table 26.5. Acceptable server error codes.

Value	*Error Code*
200	PROTOCOL_OK
204	PROTOCOL_NO_RESPONSE
302	PROTOCOL_REDIRECT
304	PROTOCOL_NOT_MODIFED
400	PROTOCOL_BAD_REQUEST
401	PROTOCOL_UNAUTHORIZED
403	PROTOCOL_FORBIDDEN
404	PROTOCOL_NOT_FOUND
407	PROTOCOL_PROXY_UNAUTHORIZED

500	PROTOCOL_SERVER_ERROR
501	PROTOCOL_NOT_IMPLEMENTED

Implementation Considerations

When you're ready to take the plunge into developing NSAPI functions, or you're at least looking at themseriously, you should be aware of a couple of points that will affect both how easy it will be to developyour solution and where you can use that solution. I describe these points in the following sections.

Cross-Platform Capabilities

Who said, "You can't take it with you"? One of the great things about the NSAPI is that it's cross-platform Currently, it is supported with Netscape products on several UNIX platforms as well as Windows NT, and that list is bound to grow with time. Because the internal function calls remain exactly the same fromplatformto platform, all you need to worry about is the C code that does the processing for your particular functions.

Informational Resources

A growing amount of information is available on the NSAPI, but it is still in the early stages of development. You can obtain the best resources available through the Netscape Developer's program, called DevEdge, or through taking one of the Netscape training courses on the NSAPI, which were started in the second quarter of 1996. In public newsgroups, you also can find a number of people making use of the NSAPI, as it encompasses a wide range of programming topics. Secured newsgroups on Netscape's site, for registered developers, have a high turnout rate for questions and answers, and they provide the added benefit of serving as a link to Netscape support personnel and advanced developers.

A growing number of developers are also beginning to offer commercial services for functions such as helping you determine how you might move a CGI function into NSAPI functionality or designing server extensions fromthe ground up.

Programming Knowledge

API programming is much easier to approach if you're already a C programmer. If you are, you're well on your way to taking advantage of all the power that API has to offer. If you're not a C programmer, however, or you work with a groupwith limited time for developing constantly changing functions, you may want to consider how you would go about making NSAPI functions and whether they're worth the time and effort. CGI programming still maintains one significant advantage over API programming-it's easy to

do, in any language. If all your development experience is in Perl, you may not be willing to invest the considerable time and frustration it will take to learn C and become fluent enough in it to code advanced and robust functions. For many people, this decision will be the biggest obstacle to developing their functions and one of the factors that will limit API development impact on the general server marketplace.

With all the resources available and with training under your belt, you might begin to wonder if API programming is a good direction to be going in. After all, programming these functions is very specific to servers that support the NSAPI standard, and this programming is somewhat involved. Is it worth all the effort? The answer to this question depends on whomyou ask, but the majority opinion is that this type of programming is definitely an area in which you should become well versed. CGI will probably never go away. Its quick and dirty, easily implemented data processing functions are always needed, and it's just way too much fun. Corporate-level solutions, however, are increasingly demanding and increasingly standards oriented.

Debugging

To debug a built-in server function, you must rely heavily on your faith in your code because some possible symptoms can occur due to an errant return fromyour code or a slight error in placing data within a `pblock`. One of the best considerations is to include a logging function within your function so that a look at the system's error logs can at least helpyou pinpoint if your function is doing anything and what data it has.

Here is one relatively common pitfall to avoid when calling NSAPI functions through references in an HTML page. When they're being referenced in a formelement like this

```
<FORM METHOD="POST" ACTION="/cgi-bin/nsapi/blah.useaction">
```

you must have a file called blah.useaction in `/cgi-bin/nsapi`. The file doesn't need to have anything in it (it can be a shopping list or a doghouse blueprint), but the file itself needs to exist because the server first checks the validity of the file by using the `PathCheck` class. If the server doesn't see a file there, it fails and doesn't do what you want it to do.

The Future of the NSAPI

Standards are a big issue when it comes to the Internet. All companies want their technology to be the one that everyone else adopts because money rides on being the leader. To this point, Netscape has dominated the commercial sector in developing Internet standards, or at least close enough to it that they might as well be the leader. By proposing the NSAPI as a standard, the folks at Netscape have positioned themselves to try to take the lead in server extensions, just as they have in creating demand for the HTML tags that they've adopted ahead of scheduled standards implementation.

The NSAPI is not alone in the race for standardization, however. The Internet Server API (ISAPI), created by Microsoft and Process Software (see Chapter 25 for more details on ISAPI), is generating its own band of followers. In addition, other entries into the fray exist as well, either existing as something similar to NSAPI or ISAPI or as something completely different. In the face of competition from two industry giants, though, how can standardization of other entries succeed when they aren't part of either larger camp's proposal?

Would it be possible to meet somewhere in the middle? Not easily, and not likely. The NSAPI is heavily tied to the inner workings of the server itself, relying on the obj.conf and magnus.conf files, whereas the ISAPI standard goes more along the line of Microsoft's traditional DLL additions with message hooks.

Who has the advantage? Again, the answer to this question depends on whom you ask. Independent research agencies have reported conflicting information about who's in the lead and which implementation of the API functions is better suited for standardization. The real advantage, however, currently lies in the Netscape camp because of cross-platform capabilities. Microsoft's push is currently for NT as a substitute server platform for UNIX, but with the number of UNIX boxes already in use, a large and stubborn group of the more advanced developers doesn't want to switch platforms and server software just to take advantage of the API functions.

I'm not saying that the battle for who's in the lead is over, by any means. Microsoft has never been shy about pursuing its goals for market share, and if they feel that they're at a disadvantage, they'll fight even harder to become top dog. For the moment, however, Netscape maintains the edge that may well determine what direction the more serious developers go in.

Is all this competition good news for you, as an individual? Sure. It means that more powerful functions will surely be developed for a whole range of Web servers. If you're a developer, you've got a new marketplace to play in. If you're a user, many of the tasks that you perform on the Web could become faster and more powerful. No matter from which angle you look at this competition, it's a win-win situation for everyone.

Summary

The Netscape Server API is a powerful tool for customizing your Web server and making it jump through the hoops you need it to. NSAPI is not a replacement for CGI because people will always have a need for the role CGI plays. But NSAPI can perform the same functions, be called in the same manner, and do it all with less work for your server. These benefits are balanced by its being harder for developers to get up to speed and create stable functions. After you're familiar with the way NSAPI works and have had the chance to experiment, you'll understand why it's a great tool to have around.