# *Los rincones del API Win32*
## *Más sobre WinInet y HTTP*

*Ya es hora de terminar lo que dejamos pendiente: en esta ocasión tratamos aspectos más avanzados sobre el uso del protocolo HTTP desde el API WinInet.*

# *Índice*

# *Decíamos ayer...*

Me permito el lujo de tomar prestada la frase de Fray Luís de León para comenzar este artículo, que no es más que la continuación del aparecido en nuestro número anterior.
Si por un casual te incorporas ahora a esta serie, te recomiendo que leas el artículo “WinInet y HTTP” (www.lawebdejm.com/?id=21220), ya que este que vamos a empezar se basa en lo explicado anteriormente.

A modo de repaso, recordaremos que en nuestro anterior artículo explicamos las bases del protocolo HTTP, basado en peticiones de texto entre el cliente (un navegador web) y el servidor (el programa que está escuchando al otro lado de la línea). Ya entrando en temas específicos de Wininet, dimos una descripción detallada de cómo acceder a un recurso remoto a través del protocolo HTTP, pero utilizando un sistema simplificado que nos ofrece Wininet: el método directo.

Ahora es cuando vamos a adentrarnos más en este API de Microsoft, y vamos a explicar cómo realizar las mismas tareas (y muchas más) desde el método detallado (por llamar de algún modo al método de acceso más completo).

Si recordamos el esquema que apareció en el artículo anterior, vimos que existían una serie de pasos comunes a ambos métodos, entre ellos:

Operaciones previas: una serie de llamadas para asegurarnos que la conexión está establecida, y una llamada a InternetOpen para abrir una instancia dentro del API Wininet.

Acceso a recursos: una vez que hemos enviado la petición HTTP por la red, podemos acceder a los recursos remotos (normalmente leyendo de un fichero).

Tanto las operaciones previas, como el acceso a recursos son operaciones comunes, y debemos realizarlas tanto con el método directo (explicado en el anterior número) como con el método detallado (que vamos a explicar a continuación).

Pero entre las operaciones previas como el acceso a los recursos, había una serie de operaciones que podíamos realizar de dos maneras muy diferentes: con el método directo o con el detallado. En caso de utilizar el método detallado, debíamos llamar a distintas funciones dependiendo del protocolo a utilizar: HTTP, FTP o Gopher.

En este caso vamos a explicar las funciones del método detallado para HTTP, aunque las propias para FTP las explicaremos próximamente.

# *Método detallado*

Como ya vimos, la opción más fácil para acceder a un recurso remoto era a través de la función InternetOpenUrl. Con una simple llamada a esta función, conseguíamos acceder a una URL y recuperar el contenido del archivo, independientemente del protocolo que quisieramos utilizar.
Internamente, la función InternetOpenUrl hace una serie de llamadas a las funciones propias del método detallado, así que cuando leamos este artículo seremos capaces de crear nuestra propia función InternetOpenUrl.
Sin embargo, esta función está bastante limitada, ya que no permite las mismas posibilidades que contempla el protocolo. Para ello contamos con el método detallado, que nos ofrece un mayor control sobre los principales pasos, entre ellos:

- Conexión
- Creación de la petición
- Gestión de cabeceras de HTTP
- Envío de la petición
- Vamos allá con todos estos pasos

## *Conexión*

Este paso debe ser el siguiente a la llamada a InternetOpen, es decir, el primer paso después de realizar las operaciones previas, y es en este momento cuando tenemos que decidir, tanto el servidor al que nos vamos a conectar, como el número de puerto y el protocolo a utilizar.

Esta operación se hace a través de la función InternetConnect, que se utiliza tanto en protolos con conexión (FTP) como sin ella (HTTP y Gopher). En el primer caso, debemos establecer la conexión en el momento en que el usuario lo decida (indicando usuario y clave), sin embargo, en el segundo caso (para el protocolo HTTP), debemos minimizar el uso de esta función, y evitar establecer una conexión por cada petición que hagamos contra el mismo servidor. Una buen método de hacer esto es a través de un *pool de conexiones*, almacenando un descriptor por cada servidor distinto al que nos conectamos y reutilizando ese descriptor en las distintas peticiones.

La sintaxis de la función InternetOpen es la siguiente:

```
function InternetConnect(
            hInet:        HINTERNET;
            lpszServidor: PChar;
            nPuerto:      INTERNET_PORT;
            lpszUsuario:  PChar;
            lpszClave:    PChar;
            dwProtocolo:  LongWord;
            dwOpciones:   LongWord;
            dwContexto:   LongWord
        ): HINTERNET;
```

Vamos a ver qué significan estos parámetros:

- **hInet**: el descriptor que hemos obtenido a través de la llamada anterior a InternetOpen.

- **lpszServidor**: una cadena con el nombre o dirección IP del servidor a conectar. Hay que recordar que en este parámetro sólo hay que indicar el nombre de host (p.e. www.lawebdejm.com) y no el protocolo y el host (http://www.lawebdejm.com)

- **nPuerto**: podemos utilizar un valor numérico cualquiera, o bien una de las siguientes constantes predefinidas:

  - INTERNET_DEFAULT_FTP_PORT: puerto por defecto para servidores FTP (21)
  - INTERNET_DEFAULT_GOPHER_PORT: puerto por defecto para servidores Gopher (70)
  - INTERNET_DEFAULT_HTTP_PORT: puerto por defecto para servidores HTTP (80)
  - INTERNET_DEFAULT_HTTPS_PORT: puerto por defecto para servidores HTTPS (443)
  - INTERNET_DEFAULT_SOCKS_PORT: puerto por defecto para corta-fuegos (1080)
  - INTERNET_INVALID_PORT_NUMBER: utiliza el puerto por defecto según el protocolo pasado en el parámetro dwProtocolo.

- **lpszUsuario**: un puntero a cadena con el nombre de usuario a utilizar. Para el protocolo HTTP, debemos utilizar una cadena vacía para conectarnos de forma anónima.

- **lpszClave**: la clave del nombre de usuario. Para el protocolo HTTP, debemos utilizar una cadena vacía para conectarnos de forma anónima.

- **dwProtocolo**: indica el tipo de protocolo que vamos a utilizar en la conexión. Este parámetro puede contener uno de los siguiente valores:

  - INTERNET_SERVICE_HTTP: conectarnos a un servidor HTTP
  - INTERNET_SERVICE_FTP: conectarnos a un servidor FTP
  - INTERNET_SERVICE_GOPHER: conectarnos a un servidor Gopher.

- **dwOpciones**: opciones para la conexión. Si utilizamos el protocolo HTTP, no tenemos ninguna opción disponible.

- **dwConexto**: tiene el mismo significado que para la función InternetOpenUrl, es decir, se utilizará para pasarlo a la función de "callback" en las conexiones asíncronas (pasando INTERNET_FLAG_ASYNC en el parámetro dwOpciones de la función InternetOpen). Para conexiones síncronas (por defecto), este parámetro se ignorará, por lo que el valor pasado no tendrá ningún significado.

Esta función retorna un descriptor de conexión en caso de que todo haya ido bien, o el valor **nil** si ha ocurrido un fallo.

En caso de error, se puede utilizar la función GetLastError para retornar el código de error. Recordemos que si GetLastError retorna ERROR_INTERNET_EXTENDED_ERROR, debemos llamar a la función InternetGetLastResponseInfo para que nos retorne el código y la descripción del último mensaje de error recibido desde el servidor.

## *Creación de la petición*

Una vez que contamos con un descriptor de conexión a un servidor, obtenido con la función InternetConnect, podemos empezar a enviar distintas peticiones. Para ello, lo primero que debemos hacer es crear la petición, a través de la función HttpOpenRequest.

Esta función nos permite crear una petición HTTP, sin tener que indicar todos los datos requeridos por el protocolo, aunque sí que debemos tener una noción del protocolo.

La función tiene la siguiente sintaxis:

```
function HttpOpenRequest(
            hConexión:      HINTERNET;
            lpszOperación:  PChar;
            lpszObjecto:    PChar;
            lpszVersión:    PChar;
            lpszReferencia: PChar;
            lpszTiposAceptados: ^PChar;
            dwOpciones:     LongWord;
            dwContexto:     LongWord
        ): HINTERNET;
```

- **hConexión**: el descriptor de conexión que hemos obtenido a través de la función InternetConnect.

- **lpszOperación**: un puntero a cadena que nos indica el tipo de petición HTTP que estamos creando. La operación es el valor que se pasa en primer lugar en el protocolo, e informa al servidor de la acción que estamos requiriendo. En la sección “Introducción al protocolo HTTP” del anterior artículo, ya explicamos los tipos de petición disponibles en cada una de las versiones del protocolo. Si se pasa el valor **nil**, se utilizará la operación GET.

- **lpszObjeto**: es el segundo valor que se pasa en la primera línea de la petición HTTP. Representa el objeto sobre el que actúa la operación indicada. Para operaciones GET indica el archivo a descargar, para operaciones HEAD indica el archivo del cual se obtendrá la cabecera, etc.

- **lpszVersión**: un puntero a cadena que indica la versión del protocolo que estamos utilizando. Este valor es el que se pasa en tercer lugar en la primera línea de la petición. La cadena debe contener un valor "HTTP/x.y" donde "x.y" es la versión a utilizar. Si se pasa el valor **nil**, se utilizará la versión 1.1.

- **lpszReferencia**: un puntero a una cadena que contiene una dirección URL. Esta dirección debe ser la del recurso que hace referencia al cual estamos accediendo, es decir, el recurso que nos ha permitido llegar a este punto. Por ejemplo, si navegamos, a través de un hiperenlace, desde la página "index.htm" a la página "otra.htm", la referencia de la segunda petición será la URL de "index.htm". Este dato sirve para que el servidor optimice sus operaciones, creando listas de navegación, direcciones de caché, etc. Este valor es opcional, y sólo tendrá sentido si estamos navegando de una URL a otra. En caso de querer ignorar este parámetro, nos bastará con pasar el valor **nil**.

- **lpszTiposAceptados**: debe pasarse un array de punteros a cadena en el que se enumeran los tipos MIME que acepta el cliente. Se puede pasar el valor **nil** para indicar que no se acepta ningún tipo, aunque, normalmente, esto se suele interpretar como que sólo se aceptan documentos de texto (el tipo MIME "text/*"). En la URL http://www.freesoft.org/CIE/RFC/1700/19.htm aparece una lista con todos los tipos de datos MIME existentes.

- **dwOpciones**: se puede combinar cualquiera de los siguientes valores. Todos ellos (excepto uno), tienen el mismo significado que explicamos para la función InternetOpenUrl.

    - INTERNET_FLAG_CACHE_IF_NET_FAIL: en caso de que la conexión remota falle, se retorna el recurso del caché en vez de su ubicación original.
    - INTERNET_FLAG_HYPERLINK: ver la explicación de la función InternetOpenUrl en la URL www.lawebdejm.com/?id=21220
    - INTERNET_FLAG_NO_AUTO_REDIRECT: ver InternetOpenUrl.
    - INTERNET_FLAG_IGNORE_REDIRECT_TO_HTTP: ver InternetOpenUrl.
    - INTERNET_FLAG_IGNORE_REDIRECT_TO_HTTPS: ver InternetOpenUrl.
    - INTERNET_FLAG_NEED_FILE: ver InternetOpenUrl.
    - INTERNET_FLAG_NO_CACHE_WRITE: ver InternetOpenUrl. Si hemos utilizado una operación distinta a GET o POST, este valor será utilizado automaticamente.
    - INTERNET_FLAG_NO_COOKIES: ver InternetOpenUrl.
    - INTERNET_FLAG_NO_UI: ver InternetOpenUrl.
    - INTERNET_FLAG_PRAGMA_NOCACHE: ver InternetOpenUrl.
    - INTERNET_FLAG_RELOAD: ver InternetOpenUrl. Si hemos utilizado una operación distinta a GET o POST, este valor será utilizado automaticamente.
    - INTERNET_FLAG_RESYNCHRONIZE: ver InternetOpenUrl.
    - INTERNET_FLAG_SECURE: ver InternetOpenUrl.

- **dwContexto**: tienen el mismo significado que el explicado en InternetOpenUrl.

La función retornará un descriptor de petición o bien **nil** si ha ocurrido algún error (llamar a GetLastError para más detalles).

## *Gestión de cabeceras adicionales*

Algunos de los parámetros que pasamos mientras creamos la petición, se traducen en cabeceras del protocolo HTTP. Sin embargo, quizá en algún momento tengamos la necesidad de enviar o eliminar una cabecera. Un caso típico de esto suele ser el envío de formularios (con la instrucción POST) a través de Wininet, o bien la recuperación de una página si está modificada a partir de una fecha (con la cabecera HTTP “If-Modified-Since”).

Para solucionar estas situaciones tenemos disponible la función HttpAddRequestHeaders, que precisamente se encarga de gestionar las cabeceras internas de la petición que acabamos de crear. La sintaxis es la siguiente:

```
function HttpAddRequestHeaders(
              hPetición:           HINTERNET;
              lpszCabeceras:       PChar;
              dwLongitudCabeceras: LongWord;
              dwOpciones:          LongWord
         ): LongBool;
```

- **hPetición**: un descriptor obtenido con una llamada previa a HttpOpenRequest. Las cabeceras se añadirán a la petición identificada por este descriptor.

- **lpsCabeceras**: Un puntero a un cadena que contiene el conjunto de cabeceras a añadir. Si pasamos más de una cabecera, debemos separar cada una de ellas a través de los caracteres retorno de carro (ASCII 10 y 13).

- **dwLongitudCabeceras**: indica la longitud de la cadena que hemos pasado en el parámetro lpszCabeceras. Si pasamos el valor -1 en este parámetro se calculará la longitud de la cadena buscando el carácter nulo final.

- **dwOpciones**: indica distintas opciones para añadir las cabeceras. Puedes ser cualquier combinación de los siguientes valores:

  - HTTP_ADDREQ_FLAG_ADD: añade la cabecera si no existe
  - HTTP_ADDREQ_FLAG_ADD_IF_NEW: añade la cabecera sólo si no existe, en caso contrario, retorna un error.
  - HTTP_ADDREQ_FLAG_COALESCE: combina distintas cabeceras con el mismo nombre en una sola, siempre y cuando el tipo de cabecera que estemos combinando lo permita.
  - HTTP_ADDREQ_FLAG_COALESCE_WITH_COMMA: Combina distintas cabeceras del mismo modo que el valor anterior, pero utilizando una "coma" como separador. Por ejemplo, Si existe la cabecera "Accept: text/*" y añadimos una nueva cabecera de tipo "Accept: image/*", el resultado combinado sería "Accept: text/*, image/*".
  - HTTP_ADDREQ_FLAG_COALESCE_WITH_SEMICOLON: Combina distintas cabeceras del mismo modo que el valor anterior, pero utilizando un "punto y coma" como separador.
  - HTTP_ADDREQ_FLAG_REPLACE: Reemplaza o elimina una cabecera. Si el valor de la cabecera (dentro de lpszCabeceras) es nulo, entonces la cabecera se eliminará. Si el valor no es nulo, se reemplazará con el indicado en lpszCabeceras. En este caso sólo se puede pasar una cabecera dentro del parámetro lpszCabeceras.

Esta función retorna un valor booleano indicando su éxito o fracaso.

## *Envío de la petición*

Una vez que hemos creado una petición, y conocemos su descriptor, debemos enviarla por la red, para que llegue al servidor HTTP y este la procese.

Para ello contamos con la función HttpSendRequest, que dada una petición creada con HttpOpenRequest, la envía a través de la conexión activa y recibe la respuesta.

Esta petición puede ser tanto de recepción de datos (con la operación GET, HEAD, etc.) como de envío (con la operación POST o PUT).

Hay que avisar que este es el paso más lento del método detallado, ya que es donde se establece la conexión y envío físico a través de la red, y es en este momento cuando podemos caer en un timeout más o menos largo.

```
function HttpSendRequest(
               hPetición:           HINTERNET;
               lpszCabeceras:       PChar;
               dwLongitudCabeceras: LongWord;
               lpOpcional:          Pointer;
               dwLongitudOpcional:  LongWord
          ): LongBool;
```

- **hPetición**: un descriptor obtenido a través de una llamada a HttpOpenRequest.

- **lpszCabeceras**: un puntero a cadena donde se almacenan las cabeceras adicionales a enviar en la petición. Lo más normal es que no sea necesario enviar cabeceras adicionales, así que pasaremos el valor **nil**. Más abajo, hablaremos sobre cómo utilizar este parámetro para simular un envío de formulario a través de Wininet.

- **dwLongitudCabeceras**: número de caracteres que hemos pasado en el parámetro lpszCabeceras. Si pasamos -1, y en lpszCabeceras hay un valor distinto de **nil**, la función calculará la longitud buscando el carácter nulo final.

- **lpOpcional**: es un puntero a un buffer que contiene información adicional. Esta información será enviada inmediatamente después de las cabeceras estándar, y normalmente se utiliza en peticiones de tipo POST o PUT. Para peticiones de tipo GET, o si no necesitamos enviar información adicional, pasaremos el valor **nil**.

- **dwLongitudOpcional**: la longitud, en bytes, del buffer pasado en lpOpcional, o 0 si se ha pasado el valor **nil**.

La función retornará TRUE o FALSE dependiendo del éxito.

Para un mayor control sobre en envío de la petición, podemos hacer uso de la función extendida HttpSendRequestEx. La principal razón para utilizar esta otra función es el envío de peticiones de gran tamaño (como hacer un "upload" de un archivo a través de la operación POST), ya que con el uso de la función HttpSendRequest, se requiere que la petición se envíe en una sola intrucción. Usando la función extendida HttpSendRequestEx, podemos comenzar una petición, enviar la petición en distintas llamadas, y finalizar el envío de la petición una vez que todo está completado.

No voy a explicar con detalle el uso de esta función, aunque aquellos que queráis profundizar en su uso, podéis ver un programa de ejemplo en la URL
http://support.microsoft.com/support/kb/articles/Q177/1/88.asp

Con todos estos pasos ya hemos sabemos cómo utilizar el método detallado. Como habéis visto, se tiene un control mucho mayor sobre la petición HTTP que se envía al servidor.

# *Acceder a una URL protegida*

Una de las situaciones más comunes dentro de internet es encontrarnos con una URL protegida con usuario y contraseña. Este mecanismo permite al webmaster un control sobre qué usuarios acceden a los datos considerados más importantes.

Como ya hemos visto, el API WinInet gestiona bien estas situaciones, simplemente indicando un usuario y contraseña como parámetros de la función InternetConnect. Sin embargo, si nos paramos a pensar nos daremos cuenta de que tenemos un problema... ¿cómo vamos a saber si una cierta URL está protegida y debemos pasar los datos de identificación en la llamada a InternetConnect? es más... antes he dicho que debemos llamar a InternetConnect una sola vez para cada host, pero ¿qué pasa si distintos recursos utilizan distintas claves? ¿debemos llamar a InternetConnect varias veces? La respuesta es no, ya que el API WinInet nos ofrece un mecanismo para establecer el usuario y contraseña para una sola petición (y no para una conexión completa).

El modo de actuar es el siguiente, partiendo del método detallado:

1. Apertura del API con InternetOpen
2. Conexión al host con InternetConnect (sin usuario ni contraseña)
3. Creación de la petición con HttpOpenRequest
4. Envío de la petición con HttpSendRequest
5. Recuperación del código de estado retornado por el servidor con HttpQueryInfo

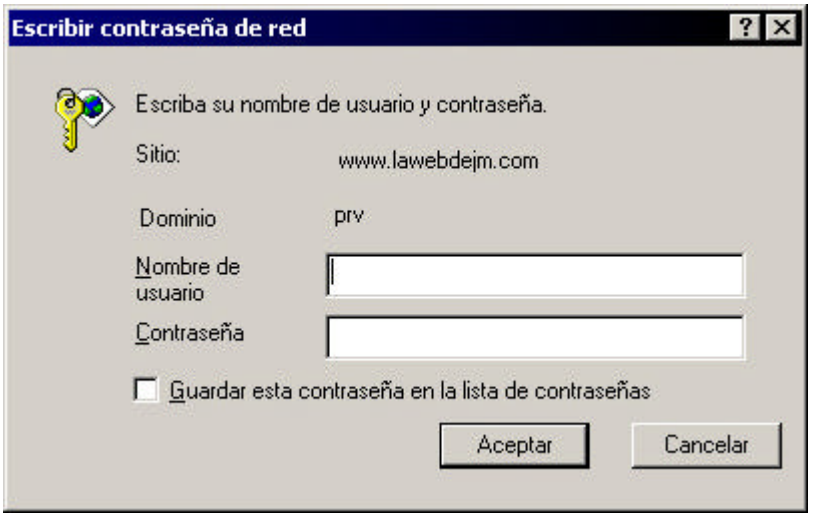
Una vez realizados estos pasos, debemos comprobar si el código retornado en el punto 5 es uno de los siguiente:

- HTTP_STATUS_DENIED (valor 401): se retorna cuando el recurso remoto al que estamos accediendo está protegido con usuario y contraseña.

- HTTP_STATUS_PROXY_AUTH_REQ (valor 407): se retorna cuando accedemos al recurso a través de un servidor proxy y éste requiere de validación del usuario. Algunos entornos empresariales utilizan la validación de usuarios a través del proxy para limitar el acceso a internet a los usuarios que conozcan la contraseña correcta.

Si recibimos alguno de los siguientes códigos, debemos actuar del mismo modo: mostrando una ventana de login en la que mostramos los campos para que se introduzca el nombre de usuario y contraseña.

Para ello podemos valernos de nuestro entorno de desarrollo o bien utilizar la ventana de diálogo propia de Internet Explorer, tal y como aparecen en la siguiente **Figura 1**:



## *Uso de la ventana de diálogo estándar*

Una vez que sabemos que la URL (o proxy) está protegida con contraseña, debemos pedir al usuario los datos de identificación. La opción más común es utilizar el interfaz de usuario que ha diseñado el equipo de WinInet para ello, tal y como hemos visto en la **Figura 1**.

Para ello, debemos hacer uso de la función InternetErrorDlg, que se encargará de mostrar distintas ventanas de diálogo, dependiendo del evento que haya sucedido (petición de clave, envíos de datos no seguros, etc.).

Lo más habitual es que la propia ventana almacene los datos introducidos como atributos del descriptor de la petición. Vamos a ver que significa esto: cada descriptor de tipo HINTERNET, además de identificar un objeto concreto, tiene asociados una serie de atributos o datos. Cada tipo de descriptor tendrá unos atributos distintos, así, un descriptor creado con InternetConnect (una conexión), tendrá como atributos el usuario y la clave; un descriptor creado con HttpOpenRequest (una petición), tendrá como atributos el tiempo de espera (timeout), la URL, etc. Sabiendo esto, podemos hacer que la función InternetErrorDlg almacene los datos correspondientes en los atributos del descriptor.

Una vez que el descriptor ha sido correctamente configurado por InternetErrorDlg, solamente debemos reenviar la petición con HttpSendRequest, para que el servidor reciba la nueva petición modificada (con usuario y clave) y podamos acceder libremente al recurso remoto.

La sintaxis de la función es la siguiente:

```
function InternetErrorDlg(
             hVentanaPadre: HWND;
             hPetición:     HINTERNET;
             dwTipoError:   LongWord;
             dwOpciones:    LongWord;
             var lppvDatos: Pointer
         ): LongWord;
```

Los parámetros tienen los siguientes usos:

- **hVentanaPadre**: un descriptor de la ventana que será padre del diálogo a mostrar. Como todas las ventanas de diálogo que muestra InternetErrorDlg son de tipo modal, es necesario conocer la ventana padre, la cual quedará bloqueada mientras no se cierre la ventana modal.

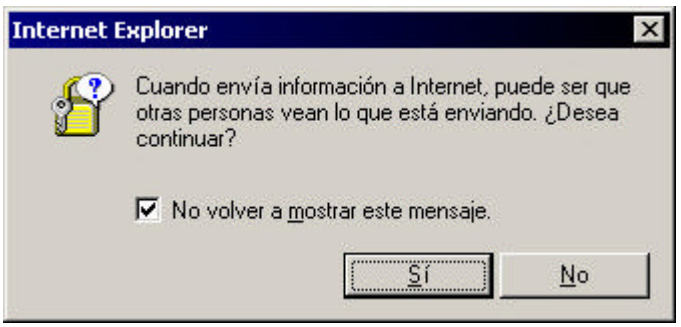
- **hPetición**: un descriptor obtenido a través de una llamada a HttpSendRequest. La ventana de diálogo mostrará y almacenará datos conforme al tipo de petición que le pasemos en este parámetro.

- **dwTipoError**: un valor que indica el tipo de error que ha sucedido y la ventana de diálogo que se mostrará. Lo más habitual es que se utilice el valor ERROR_SUCCESS si HttpSendRequest se ha ejecutado correctamente, o en caso contrario, el valor devuelto por GetLastError, ya que esta función retornará un valor indicando la causa del error, para que InternetErrorDlg muestre la ventana correspondiente (si procede). Los valores permitidos que se pueden utilizar son:

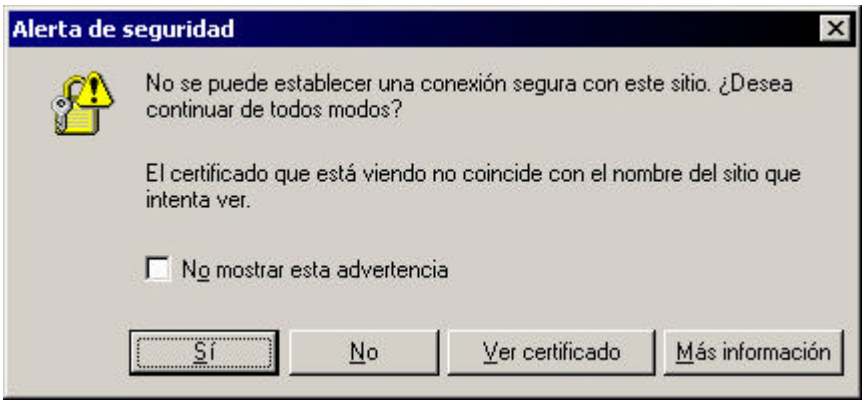
  - ERROR_INTERNET_INCORRECT_PASSWORD: se muestra la ventana de diálogo para que el usuario introduzca el nombre de usuario y contraseña (la de la **Figura 1**).
  - ERROR_INTERNET_HTTP_TO_HTTPS_ON_REDIR: se muestra una ventana informando al usuario que se va a salir o entrar a una zona segura (a través de SSL). Se debe mostrar este diálogo siempre que se navegue de una dirección que comience

"http://" a otra que comience por "https://" o viceversa. Es muy conveniente avisar al usuario de este suceso, para que tenga constancia de la entrada o salida de zonas seguras y pueda decidir si quiere enviar datos comprometedores por la red.

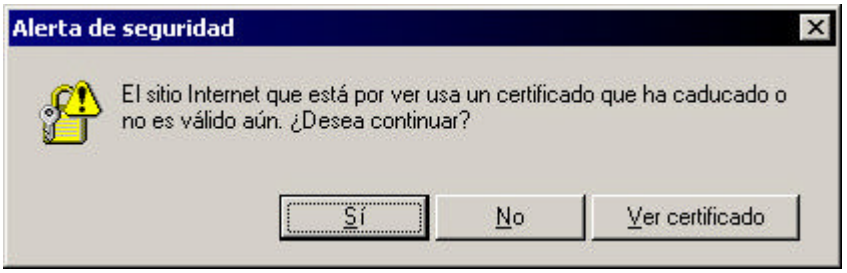
- ERROR_INTERNET_POST_IS_NON_SECURE: se muestra una ventana de aviso cuando se envía información desde el cliente al servidor (operación POST) y ésta información no está cifrada. Se puede ver una imagen de esta ventana en la siguiente imagen:



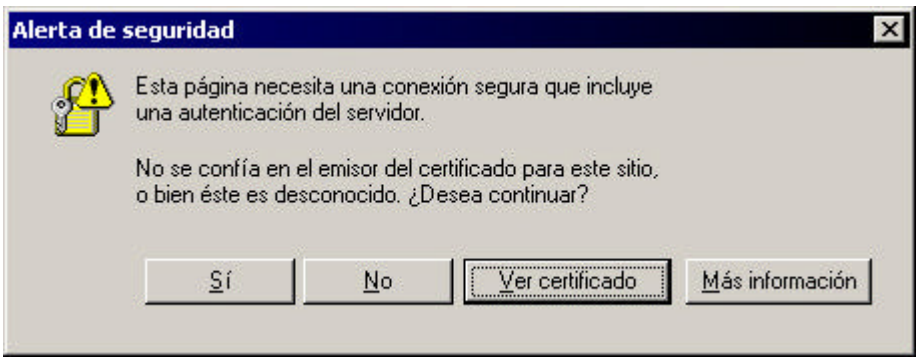

- ERROR_INTERNET_SEC_CERT_CN_INVALID: muestra una ventana informando al usuario de que el certificado que se va a aplicar en la conexión segura no se ha reconocido (aunque es válido). Además permite ver los datos del certificado. Puede verse la imagen de esta ventana en esta otra imagen:




- ERROR_INTERNET_SEC_CERT_DATE_INVALID: se informa al usuario de que el certificado digital que estaba utilizando ha caducado, tal y como aparece en la imagen siguiente:



- ERROR_INTERNET_INVALID_CA: se informa al usuario de que el certificado SSL ha sido generado por una autoridad no reconocida. La imagen es la siguiente:




- **dwOpciones**: indica que tipo de opciones se tendrán en cuenta a la hora de mostrar la ventana de diálogo. Se puede utilizar uno de los siguientes valores:
  - FLAGS_ERROR_UI_FILTER_FOR_ERRORS: buscará errores en las cabeceras retornadas por el servidor (un caso típico de esto puede ser, por ejemplo, errores de validación de usuarios). Sólo se puede utilizar esta bandera después de enviar la petición a través de HttpSendRequest.
  - FLAGS_ERROR_UI_FLAGS_CHANGE_OPTIONS: se almacenarán los datos introducidos por el usuario en los atributos del propio descriptor (por ejemplo, el usuario y contraseña).
  - FLAGS_ERROR_UI_FLAGS_GENERATE_DATA: recupera de los atributos del descriptor los datos para mostrarlos (si es necesario) en la ventana de diálogo. No todas las ventanas de diálogo necesitan utilizar este parámetro.
  - FLAGS_ERROR_UI_SERIALIZE_DIALOGS: este valor debe utilizarse para sincronizar los accesos simultáneos.
- **lppvDatos**: este parámetro sólo se debe utilizar cuando se utiliza el valor FLAGS_ERROR_UI_SERIALIZE_DIALOGS, para pasar un puntero a una estructura de tipo INTERNET_AUTH_NOTIFY_DATA. No voy a explicarlo ya que resultaría demasiado complejo para este artículo.

Esta función puede retornar uno de los siguientes valores:

- ERROR_SUCCESS: la función se ha completado con éxito. En caso de haber pasado el dwTipoError a ERROR_INTERNET_INCORRECT_PASSWORD, este retorno significa que el usuario ha pulsado el botón "Cancelar".
- ERROR_CANCELLED: la función ha sido cancelada por el usuario.
- ERROR_INTERNET_FORCE_RETRY: Este valor indica que es necesario volver a enviar la petición. En la ventana de introducción de usuario y clave (indicando el tipo de error ERROR_INTERNET_INCORRECT_PASSWORD), este retorno significará que ha pulsado el botón "Aceptar".
- ERROR_INVALID_HANDLE: el descriptor que se ha pasado en el parámetro hVentanaPadre no es válido.

## *Ventana personalizada*

El otro modo de preguntar por el nombre de usuario y contraseña es a través de una ventana personalizada. No voy a explicar cómo crear la ventana de diálogo y mostrarla, ya que eso es una tarea que depende del entorno de desarrollo que utilicemos (Visual C++, C++Builder, etc.), y este artículo no trata sobre ello.

Sin embargo, después de pedir los datos al usuario, debemos realizar lo que la función InternetErrorDlg hacía por nosotros: establecer los valores correspondientes a los atributos del descriptor de la petición. En nuestro caso, debemos establecer los atributos "usuario" y "contraseña" con los valores introducidos en la ventana personalizada, y volver a reenviar la petición para que el servidor reciba los nuevos atributos.

Algunos de los atributos aplicados a un descriptor son heredables, es decir: un atributo heradable se aplica al tipo concreto (un descriptor de conexión, de petición, etc.) y a todos los descriptores creados a partir de ese descriptor.

Vamos a poner un ejemplo para aclarar un poco el asunto: supongamos que hemos creado un descriptor de tipo conexión (a través de la función InternetConnect). Se establecemos los atributos "usuario" y "contraseña" en este descriptor, estos valores se utilizarán para todos los descriptores descendientes de éste, es decir: para todas las peticiones creadas sobre esta conexión. Del mismo modo, podemos establecer valores a los atributos del descriptor de instancia (creado con InternetOpen) y todos los descriptores descendientes (conexión, petición, etc.) aplicarán estos valores.

La operación de establecer los atributos de un descriptor (cualquiera) se hace a través de la función InternetSetOption:

```
function InternetSetOption(
               hDescriptor:      HINTERNET;
               dwAtributo:       LongWord;
               lpBuffer:         Pointer;
               dwLongitudBuffer: LongWord
           ): LongBool;
```

- **hDescriptor**: indica el descriptor en que se establecerá el atributo, o bien el valor **nil** para establecer atributos globales por defecto. En este caso, estaremos modificando las opciones por del API WinInet, que el usuario ha configurado a través del Panel de control - Opciones de internet.
- **dwAtributo**: el atributo (u opción) que se establecerá. A continuación se enumeran algunos de los atributos más importantes que se pueden utilizar en este parámetro. Para una lista completa, se debe consultar en la documentación oficial de Microsoft.

  - INTERNET_OPTION_USERNAME: el nombre de usuario asociado al descriptor, que se utilizará en las peticiones de este descriptor y de los descendientes.
  - INTERNET_OPTION_PASSWORD: la contraseña asociada al descriptor, que se utilizará junto con el nombre de usuario.
  - INTERNET_OPTION_PROXY_USERNAME: el nombre de usuario para la conexión al servidor proxy.
  - INTERNET_OPTION_PROXY_PASSWORD: la contraseña para la conexión al servidor proxy.
  - INTERNET_OPTION_CONNECT_RETRIES: número de intentos de conexión a un host que se hará. Si el host en cuestión tiene una única dirección IP, sólo se hará una conexión, pero si tiene varias, se harán tantos intentos de conexión como indique este parámetro, hasta un máximo del número de direcciones IP que tenga.
  - INTERNET_OPTION_CONNECT_TIMEOUT: número de milisegundos que se tardará en dar por fallido un intento de conexión.

  - INTERNET_OPTION_DIAGNOSTIC_SOCKET_INFO: recupera una estructura de tipo INTERNET_DIAGNOSTIC_SOCKET_INFO en la que se almacena información relativa a los sockets utilizados durante la coneción.
  - INTERNET_OPTION_HTTP_VERSION: versión del protocolo HTTP que se utilizará en las peticiones.
  - INTERNET_OPTION_VERSION: versión de la librería "wininet.dll" que se está utilizando.
  - INTERNET_OPTION_REQUEST_PRIORITY: indica la prioridad con la que se ejecutará la petición.

- lpBuffer: un puntero a un buffer donde se almacena el valor a establecer en el atributo.

- dwLongitudBuffer: longitud de los datos almacenados en lpBuffer. Si se trata de una cadena de caracteres, la longitud vienen expresada en número de caracteres, en caso contrario, la longitud debe ser en bytes.

La función retornará un valor booleano indicando el éxito o fracaso de la operación.

Nuestro ejemplo, como habréis supuesto, utilizará los valores INTERNET_OPTION_USERNAME e INTERNET_OPTION_PASSWORD para establecer los atributos de identificación en el descriptor.

La función complementaria a InternetSetOption es InternetQueryOption, que nos permite recuperar el valor de los atributos de un descriptor dado. Esta función se comporta de un modo similar a InternetSetOption, así que no voy a entrar en más detalles. De todas formas, podéis leer una descripción completa en el MSDN de Microsoft (http://msdn.microsoft.com/library/en-us/wininet/wininet/internetqueryoption.asp)

# *Enviar información al servidor*

Hasta ahora nos hemos centrado principalmente en como acceder desde el cliente a información almacenada en el servidor. Se puede decir que esta tarea es la más usual, ya que la mayoría de las ocasiones lo que queremos hacer es consultar información de la red. Sin embargo, en algunas ocasiones necesitamos realizar lo contrario: enviar información al servidor para que haga algo con esta información.

Pongámonos en el siguiente caso: tenemos una página web en la que vamos publicando noticias diariamente (los conocidos weblogs). Lo más habitual es que consultemos esta página para leer las distintas noticias que van apareciendo, imprimirlas, o incluso descargar algún archivo que esté disponible.

Pero estar páginas tienen algo más: permiten a los visitante incluir sus propios comentarios sobre las distintas noticias, de este modo se crea un debate en la web que puede dar mucho de sí.

Con lo que sabemos hasta ahora, nos resultaría relativamente fácil descargar con el API WinInet los contenidos de una de esas webs para tratarlos desde nuestros programas. Incluso, con la especificación RDF, (Resource Description Format) podríamos acceder a los datos y metadatos de una forma estructurada, con un dialecto de basado en XML (si quieres saber de qué va eso del RDF pásate por la URL http://www.malditainternet.com/index.php?section=article&sid=107).

Pero... ¿qué ocurre si queremos añadir un comentario desde un programa hecho con WinInet? Desde la propia página sabemos que se suelen utilizar los formularios propios de HTML para estar tareas, llamando a un script en el servidor (escrito en PHP, ASP, etc) que almacena el comentario en una base de datos para luego mostrarlo a otros usuarios. Sin embargo, desde WinInet no sabemos ni cómo empezar.

Pero lo primero es entender cómo se envía un formulario a través de HTTP y después veremos cómo se hace esto desde WinInet.

## *Formularios en HTTP*

Los formularios son el elemento para el envío de información contemplado dentro del protocolo HTTP. El lenguaje de marcas HTML introduce una serie de etiquetas para manejarlos facilmente, como <FORM>, <INPUT>, etc.

No voy a explicar cómo se crearía un formulario en HTML, aunque sí vamos a ver qué ocurre cuando se envía un formulario a través del protocolo HTTP.

Normalmente, un formulario se envía desde una página HTML a un script que reside en el servidor, como puede ser una página PHP, ASP, JPS o un programa CGI, ISAPI, etc. Este script será el encargado de procesarlo y hacer lo que sea necesario con los datos recibidos (almacenarlos en una base de datos, enviar un correo electrónico, etc.).

Existen dos métodos de enviar un formulario en HTTP, atendiendo al modo en que se envía la información:

1. **Método GET**: se envía una petición típica con el verbo GET. Los datos a enviar se incluyen dentro de la propia URL de la petición, precedidos del carácter "?", y se componen de un conjunto de pares "variable"-"valor". Estas "variables" y "valores" deben codificarse de un modo especial, llamado "application/x-form-urlencoded", según la cual se define un conjunto de normas, entre ellas:

    - Las "variables" deben separarse de los "valores" a través del carácter "=".

- Los distintos pares "variable"-"valor" deben separarse por el carácter "&".
- Tanto las "variables" como los "valores" deben codificarse en secuencias caracteres "seguros", según se define en el documento RFC 1738. Esto ya lo explicamos en el primer artículo sobre WinInet, cuando hablamos de la función InternetCanonicalizeUrl, aunque lo más importante que debemos recordar es que los caracteres "no seguros" se sustituyen por su código ASCII en hexadecimal, precedidos del carácter "%". Por ejemplo, el carácter espacio tiene el código ASCII 32, por lo que lo traduciremos a hexadecimal para generar el carácter seguro: "%20". Lo mismo ocurre con los caracteres "fin de línea" y "retorno de carro": "%0D%0A".

Aplicando estas normas, podemos enviar un texto a través de la cadena codificada, poniendo especial cuidado a los caracteres no seguros (espacios, retornos de carro, etc.), por ejemplo:

```
http://www.host.com/script.php?var1=valor1&var2=valor%20compuesto
```

Este es el método utilizado para enviar datos de texto, como un comentario, los parámetros de una búsqueda, etc. Una de las ventajas de este método es que no es obligatorio que el script al que accedamos sea de tipo server-side (es decir, un programa de ejecución en el servidor como PHP, ASP, CGI, etc.), sino que podríamos enviar un formulario a una sencilla página web, y desde esta procesar los datos incluidos dentro de la URL (con Javascript, por ejemplo).

2. Método POST: los datos se envían dentro del cuerpo de la petición, y no como parte de la URL. Como ya sabemos, cada petición HTTP se formaba de un grupo de cabeceras (Accept, Content-type, Content-Length, etc.) y un retorno de carro. Las peticiones de tipo POST tienen un elemento adicional después del retorno de carro: los datos a enviar. Sabiendo esto, una petición POST tiene el siguiente aspecto:

```
(1) POST /script.php HTTP/1.0
    Accept: text/html
    Accept: video/mpg
    Accept: image/*
    Accept: */*
    User-Agent: Mozilla/3.0
(2)
(3) cuerpo: datos a enviar
```

Según vemos, en el número (1) aparecen las cabeceras, en el número (2) el retorno de carro y en el número (3) aparece el cuerpo de la petición, es decir: los datos a enviar.

Según esto, ya sabemos que los datos a enviar no serán visibles en la propia URL, por lo que este método es más seguro que el anterior, ya que permite enviar información "privilegiada" como contraseñas, nombres de usuario, etc. De todas formas, es fácil averiguar qué datos se han enviado, con un simple proxy situado entre el cliente y el servidor.

Pero la gran ventaja de este método no es la ocultación (relativa) de la información a enviar, sino que podemos decidir el formato en que se enviarán los datos dentro del cuerpo, pudiendo utilizar el "application/x-form-urlencoded" o bien el formato "multipart/form-data".

Una petición POST de tipo "application/x-form-urlencoded" será como cualquier otra petición POST, incluyendo os datos codificados (como ya hemos explicado) en el cuerpo de la petición:

```
     POST /script.php HTTP/1.0
     Accept: text/html
     Accept: video/mpg
     Accept: image/*
     Accept: */*
     User-Agent: Mozilla/3.0
     
(1)  variable1=valor1&variable2=valor%20compuesto
```

En la zona marcada con (1) se puede ver el cuerpo con los datos a enviar, codificados según las reglas que ya hemos explicado.

Las peticiones POST del tipo "multipart/form-data" con muy distintas, ya que permiten el envío de información "no-textual", como ficheros, datos binarios, etc.
Básicamente, las peticiones de este tipo siguen las reglas marcadas por la recomendación MIME para codificar archivos binarios (es la misma recomendación que se utiliza para enviar correos electrónicos con ficheros adjuntos). Una vez codificada la información, se presenta en distintas secciones dentro de la petición HTTP.

No voy a explicar en profundidad como utilizar la recomendación MIME ya que es un tema demasiado extenso. De todas formas, para todos aquellos que queráis aprender a utilizarlo, podéis echar un vistazo en la URL
http://www.w3.org/TR/REC-html40/interact/forms.html#didx-multipartform-data
donde se explica en profundidad cómo enviar datos binarios a través de "multipart/form-data".

Este es el método recomendado para enviar peticiones más "delicadas", como formularios de registro, ficheros, contraseñas, etc. Se debe utilizar este método cuando la recepción de la peticiones tienen efectos colaterales, como por ejemplo: modificación de una base de datos, envío de un correo electrónico, etc.

## *Formularios a través de WinInet*

Una vez que sabemos lo que ocurre cuando se envía un formulario con el protocolo HTTP, vamos a ver qué pasos tenemos que dar para hacer esta tarea desde el API WinInet.

Voy a explicar el envío a través de los dos métodos conocidos (GET y POST) aunque no voy a entrar en la codificación "multipart/form-data".

La base de todo es construir una cadena en la que incluiremos los valores y variables que queramos enviar. Por ejemplo: supongamos que estamos implementando un sistema para enviar comentarios a una web (desde un programa escrito con WinInet). Los datos típicos pueden ser "nombre", "país" y "comentarios". Lo primero que tenemos que hacer es construír una cadena con estas variables y sus valores. Es importante que la cadena cumpla con las normas que explicamos para la codificación "application/x-form-urlencoded".

Vamos a utilizar los siguientes datos:

```
nombre:
   Juancito Pérez Pí

pais:
   España

comentarios:
   Hola!

   Este es mi comentario.

   Un saludo

   Juancito Pérez Pí.
```

Para construír esta cadena:

```
nombre=Juancito%020P%E9rez%020P%ED
&pais=Espa%F1a
&comentario=Hola%21%0D%0A%0D%0AEste
%020es%020mi%020comentario.
%0D%0A%0D%0AUn%020saludo%0D%0A%
0D%0AJuancito%020P%E9rez%020P%ED.
```

Fijáos los caracteres que se han sustituido:

```
é      -->   %E9
í      -->   %ED
ñ      -->   %F1
!      -->   %21
nueva
línea  -->   %0D%0A
```

Una vez que tenemos la cadena codificada, es fácil crear la petición de tipo POST (o GET) incluyendo la cadena codificada.

Los únicos puntos en los que cambia algo respecto a una petición de descarga normal, son los siguientes:

- El parámetro lpszMétodo de la función HttpSendRequest debe ser el correspondiente: POST o GET.

- En la llamada a HttpSendRequest es necesario añadir una nueva cabecera en la petición, para definir el tipo de codificación a utilizar. La cabecera a añadir puede ser:

  - Content-Type: application/x-www-form-urlencoded
  - Content-Type: multipart/form-data

- El formato de los datos a enviar en la llamada a HttpSendRequest deben corresponderse al tipo de codificación utilizada (el valor pasado en la cabecera "Content-Type"). Si utilizamos el tipo "application/x-www-form-urlencoded" deben utilizarse las normas que ya hemos explicado. Si quisieramos enviar datos binarios, deberíamos codificar los datos según las normas de "multipart/form-data".

# *Descargas segmentadas*

En estos tiempos que corren, lo más habitual es que descarguemos archivos relativamente grandes: un video, un zip con el código fuente de esa librería que vamos a utilizar, un documento de ayuda, etc. Esta situación llevó hace unos años a generalizar el uso de gestores de descarga (o download managers en inglés) que nos permitían optimizar nuestra lenta conexión a internet (GetRight, Download Accelerator o FlashGet son algunos de los más conocidos).

La técnica para lograr esto es tan sencilla como efectiva: consiste en dividir nuestro archivo remoto en trozos (llamados formalmente segmentos), y descargar cada uno de esos trozos por separado, normalmente a través de un hilo de ejecución. Cada uno de estos trozos se puede ir guardando en un archivo, y al finalizar se fusionarán todos los segmentos en un único archivo, que contendrá los datos del recurso remoto.
Esto lo puede mejorar y refinar todo lo que queramos: utilizando un archivo único de destino donde vamos guardando los distintos segmentos, descargando cada segmento desde un proceso distinto y todo lo que se nos ocurra etc.

Para conseguir esto, podemos hacerlo de dos maneras: una fácil (y mala) y otra más elaborada (pero buena). Veamos pues:

## *Jugando con el puntero del recurso*

El método más sencillo, y menos eficiente, es a través de la función InternetSetFilePointer, que ya explicamos en nuestro anterior artículo (www.lawebdejm.com/?id=21220). Con esta función, conseguimos situar el puntero del archivo en una posición concreta, con lo que conseguiremos que las llamadas a InternetReadFile hagan la lectura a partir de la posición en que hemos situado el puntero.

Con esto podemos descargar un rango determinado, por ejemplo:

```
var
   hInet, hReq: HINTERNET;
   byteIni: DWORD;
begin
    // se abre la instancia de internet
    hInet := InternetOpen('Descarga segmentada',
                          INTERNET_OPEN_TYPE_PRECONFIG,
                          nil, nil, 0);

    // se abre la URL, enviando también las cabeceras adicionales
    hReq := InternetOpenUrl(hInet,
                            'http://www.servidor.com/recurso.zip',
                            nil, 0, INTERNET_FLAG_RELOAD, 0);

    // almaceno en una variable el límite inferior del rango
    byteIni := 512;

    // situamos el puntero en el byte 512
    InternetSetFilePointer(hReq, byteIni, nil, FILE_BEGIN, 0);

    // ... lectura (a partir del byte 512) del recurso
    // con InternetReadFile

    InternetCloseHandle(hReq);
    InternetCloseHandle(hInet);
end;
```

Este sistema, aunque es muy sencillo de utilizar, tiene dos serios problemas:

1. Para utilizar esta función es necesario que para la descarga se esté utilizando el caché de WinInet, ya que, en realidad, el desplazamiento del puntero se hace sobre el fichero local almacenado en el caché, y no de forma remota como podría parecer. Si estamos descargando archivos grandes (y precisamente para eso queremos la descarga segmentada), el caché contendrá una copia del archivo completo, por lo que puede ser un gran desperdicio de espacio.

2. Como ya hemos dicho, la función InternetSetFilePointer en realidad lo que mueve es el puntero del archivo local almacenado en el caché de WinInet. Esto implica que, en realidad, existirá un único puntero por cada proceso, por lo que, si distintos hilos de ejecución utilizan este puntero, unos hilos estarán modificando el puntero de los otros. En resumen: no es posible utilizar esta función desde aplicaciones multi-hilo. Como dijimos anteriormente, un escenario habitual en las descargas segmentadas es utilizar varios hilos de ejecución para descargar cada uno de los segmentos, y como vemos, esto no es posible con InternetSetFilePointer.

Para solucionar estos problemas, que son bastante graves, nos tenemos que apoyar en una nueva característica del protocolo HTTP, introducida en la versión 1.1: los rangos.

## *Rangos en HTTP*

Con lo que sabemos hasta ahora, es fácil hacer un programa que descargue un recurso de internet, sin embargo, no sabemos como trocear nuestro recurso para ir descargando por separado cada una de las partes. Para ello tenemos que utilizar una nueva característica de HTTP, introducida en la versión 1.1 del protocolo.

Esta característica consiste en informar al servidor de que no queremos descargar el recurso completo, sino un rango de byte de él. De este modo, conseguimos ir descargando el recurso por rangos. Por ejemplo, imaginemos un archivo que ocupa 2 KB. (2048 bytes). Podemos dividirlo en los siguientes 4 segmentos:

1. Segmento 1: desde el byte 0 al 511 (ambos inclusive).
2. Segmento 2: desde el byte 512 al 1023 (ambos inclusive)
3. Segmento 3: desde el byte 1024 al 1535 (ambos inclusive)
4. Segmento 4: desde el byte 1536 al 2047 (ambos inclusive)

Como vemos, hemos dividido el tamaño total en cuatro partes iguales, de 512 bytes cada una de ellas, marcando las fronteras de bytes entre unas y otras partes. El segmento final no llega hasta el byte 2048, sino uno menos, ya que como empezamos a contar en el byte 0, el último byte será el tamaño menos 1 (n-1).

El modo de informar al servidor del rango que queremos descargar, es a través de una cabecera adicional al comando GET, que como sabemos, podemos enviar a través de la función HttpAddRequestHeaders.
Esta petición, con la cabecera, tiene el siguiente aspecto:

```
GET /recurso.zip HTTP/1.1
Accept: text/html
Range: bytes=512-1023
```

Enviando esta cabecera, indicamos al servidor que queremos descargar los bytes comprendidos entre el 512 y el 1023.

Adicionalmente, podemos indicar que queremos descargar los bytes desde una posición determinada, hasta el final del recurso, utilizando una cabecera como la siguiente:

```
GET /recurso.zip HTTP/1.1
Accept: text/html
Range: bytes=1536-
```

De este modo, indicamos que descargaremos desde el byte 1536, hasta el final del recurso. Esta misma sintaxis, nos permite definir un único rango, para descargar el recurso por completo, como hemos hecho hasta ahora:

```
GET /recurso.zip HTTP/1.1
Accept: text/html
Range: bytes=0-
```

Esto indicará que se descargue desde el byte 0 (el primero) hasta el final del recurso, o dicho de otro modo: se descargará el recurso completo, en un único segmento.

Si nos fijamos en la línea de la petición GET de todos estos ejemplos, veremos que la versión que se utiliza del protocolo es la 1.1. Esto es debido a que, como ya he dicho, los rangos de descarga sólo se soportan a partir de esta versión. En el mundo WinInet, si utilizamos el método directo (con la función InternetOpenUrl), no será necesario indicar la versión del protocolo. Sin embargo, si utilizamos el método detallado, os recuerdo que indicamos la versión a utilizar del protocolo a través del parámetro lpszVersion de la función HttpOpenRequest. Si olvidamos indicar la versión 1.1, e intentamos utilizar este tipo de cabecera, recibiremos un error de tipo ERROR_HTTP_HEADER_NOT_FOUND.

Y ahora que sabemos cómo se forma la cabecera que definen nuestros rangos, lo único que tenemos que hacer es añadir esta cabecera a nuestra petición, utilizando los mecanismos que nos ofrece WinInet para ello:

Con el parámetro "lpszCabeceras" de la función InternetOpenUrl: consiste en enviar las cabeceras adicionales a través de la función utilizada en el método directo: InternetOpenUrl. No es necesario indicar la versión del protoco, ya que se seleccionará automaticamente.

```
var
   hInet, hReq: HINTERNET;
   byteIni, byteFin: DWORD;
   cabecera: string;
begin
    // se abre la instancia de internet
    hInet := InternetOpen(...);

    // almacenamos en variables los límites del rango
    byteIni := 512;
    byteFin := 1023;

    // se compone la cabecera con su sintaxis
    cabecera := Format('Range: bytes=%d-%d', [byteIni, byteFin]);

    // se abre la URL, enviando también las cabeceras adicionales
    hReq := InternetOpenUrl(hInet,
                            'http://www.servidor.com/recurso.zip',
                            cabecera, Length(cabecera),
                            INTERNET_FLAG_RELOAD, 0);

    // ... lectura del recurso con InternetReadFile

    // y cierre de descriptores
end;
```

Con el parámetro "lpszCabeceras" de la función HttpSendRequest: utilizando el método detallado, podemos enviar cabeceras adicionales de la petición a través del segundo parámetro en la función HttpSendRequest. En este caso, debemos indicar la versión del protocolo (1.1) en la llamada a la función HttpOpenRequest:

```
var
   hInet, hConn, hReq: HINTERNET;
   byteIni, byteFin, len: DWORD;
   cabecera: string;
begin
    // se abre la instancia de internet
    hInet := InternetOpen(...);

    // se conecta al servidor
    hConn := InternetConnect(hInet, 'servidor.com',
                             INTERNET_DEFAULT_HTTP_PORT, nil, nil,
                             INTERNET_SERVICE_HTTP, 0, 0);

    // se crea la petición GET. Ojo a la versión
    hReq := HttpOpenRequest(hConn, 'GET', '/recurso.zip',
                            'HTTP/1.1', nil, nil, 0, 0);

    // almacenamos en variables los límites del rango
    byteIni := 512;
    byteFin := 1023;

    // se compone la cabecera con su sintaxis
    cabecera := Format('Range: bytes=%d-%d', [byteIni, byteFin]);

    // se manda la URL, enviando también las cabeceras adicionales
    len := Length(cabecera);
    if not HttpSendRequest(hReq, cabecera, len, nil, 0) then
    begin
        // Se ha producido un error.
        // Si no se admite la cabecera, al llamar al GetLastError
        // obtendremos el error ERROR_HTTP_HEADER_NOT_FOUND.
    end;

    // ... lectura del recurso con InternetReadFile

    // y cierre de descriptores
end;
```

Con la función HttpAddRequestHeaders: otra opción si utilizamos el método detallado es añadir la cabecera a la petición, utilizando HttpAddRequestHeaders. Esto nos permite, no solo añadir, sino eliminar, cambiar o reemplazar las cabeceras a enviar junto con la petición.

```
var
   hInet, hConn, hReq: HINTERNET;
   byteIni, byteFin, len: DWORD;
   cabecera: string;
begin
    // se abre la instancia de internet
    hInet := InternetOpen(...);

    // se conecta al servidor
    hConn := InternetConnect(...);

    // se crea la petición GET. Ojo a la versión
    hReq := HttpOpenRequest(...);

    // almacenamos en variables los límites del rango
    byteIni := 512;
    byteFin := 1023;

    // se compone la cabecera con su sintaxis
    cabecera := Format('Range: bytes=%d-%d', [byteIni, byteFin]);

    // se añade la nueva cabecera a la petición
    if not HttpAddRequestHeaders(hReq, cabecera, Length(cabecera),
                                 HTTP_ADDREQ_FLAG_ADD) ) then
    begin
        // Se ha producido un error.
    end;

    // se manda la petción
    if not HttpSendRequest(hReq, nil, 0, nil, 0) then
    begin
        // Se ha producido un error.
    end;

    // ... lectura del recurso con InternetReadFile

    // y cierre de descriptores
end;
```

# *Reanudar la descarga*

Como ya hemos visto, una de las características propias de un gestor de descarga, es el dividir el archivo remoto en segmentos, y descargarlos de forma simultanea, utilizando múltiples hilos o cualquier otro sistema de multi-proceso. Sin embargo, hay otra característica que puede resultar igual o más interesante: la reanudación de la descarga.

Cuando estamos descargando un recurso grande, lo más habitual es detener la descarga en un momento dado, para continuarla más tarde en el punto en que fue detenida. Para ello, podemos utilizar la técnica de descargas segmentadas, del siguiente modo: cuando se está descargando el archivo (es decir: dentro del bucle de InternetReadFile) debemos mantener en todo momento el números bytes que hemos descargado. En el momento en que el usuario cancele el proceso de descarga, guardaremos ese dato (el número de bytes descargados) en un archivo (ya sea un archivo INI, el registro de Windows o cualquier otro tipo de archivo). Cuando intentemos reanudar la descarga en el punto en que lo dejamos, debemos leer el número de bytes que habíamos almacenado, y descargar desde esa posición hasta el final, a través de un rango.

Pongamos un ejemplo: supongamos que estamos descargando un recurso de 500 KB (512.000 bytes). Nuestro programa comienza a descargar, y el usuario pulsa el botón "Cancelar" cuando se vamos por el byte 150.272. En ese momento, guardamos este valor en un archivo INI y terminamos el proceso de descarga, informando al usuario de que la descarga se ha cancelado y se reanudará en el mismo punto. Cuando el usuario vuelva a iniciar la descarga, definiremos un rango a partir del byte 150.272 hasta el final, con la siguiente petición GET:

```
GET /recurso.zip HTTP/1.1
Accept: text/html
Range: bytes=150272-
```

El proceso de descarga continuará a partir del punto en que lo hemos dejado, aunque tenemos que tener cuidado de añadir (y no sobrescribir) al archivo destino los nuevos datos. Si el usuario cancela de nuevo el proceso, digamos en el byte 252.672, realizaremos el mismo proceso.

Para combinar los dos técnicas (descarga segmentada y reanudación), debemos mantener en el archivo toda la información de los rangos, y la posición de cada uno de ellos en que hemos detenido la descarga.

**Atención**

Microsoft no garantiza el funcionamiento correcto de la cabecera “Range” utilizada desde WinInet. De hecho, si leemos varios mensajes de las news de WinInet (en la dirección news://microsoft.public.inetsdk.programming.wininet), veremos que los propios empleados de soporte de Microsoft avisan de que la cabecera “Range” no funciona correctamente desde WinInet.
En las pruebas que he podido hacer para una descarga segmentada, la función *HttpSendRequest* se queda colgada cuando es llamada para enviar la petición del tercer segmento.
Sin embargo, lo que si funciona es el uso de los rangos para reanudar las descargas, ya que en este caso no se envia más de una petición a la vez.
Es una verdadera lástima que WinInet tenga todavía este tipo de errores, ya que en esta ocasión, si queremos desarrollar un gestor de descargas que soporte descargas segmentadas, debemos hacer uso de otras tecnologías, como WinSock en el mundo de Microsoft.

# *Conclusión*

Y creo que esto es todo lo que sabría deciros sobre el API WinInet y el protocolo HTTP. Espero que después de estos dos artículos hayáis aprendido las bases de cómo realizar un programa que utilice el protocolo HTTP para el envío y recepción de datos on-line.

En el próximo artículo cambiaremos de protocolo, y empezaremos a utilizar el API WinInet para acceder a un servidor FTP.

# *Los ejemplos*

**Delphi 5**
Un pequeño programa que demuestra cómo acceder a páginas protegidas mostrando la ventana de login, o cómo realizar el envío de formularios a través de WinInet.
Fuentes en ZIP (http://www.lawebdejm.com/?id=22231)

**Delphi 5**
Otro programa que implementa un sistema para realizar descargas multihilo, pero permitiendo cancelar y reanudar la operación en el momento en que lo hemos dejado. Para ello hace uso de la cabecera HTTP "Range", junto con una implementación de los hilos basada en el API Win32, sin utilizar la clase *TThread* de la VCL.
Fuentes en ZIP (http://www.lawebdejm.com/?id=22232)

Autor: JM - http://www.lawebdejm.com