# Webifying Outlook

*by Jani Järvinen*

If you rely on Microsoft Outlook as your personal information manager, you might run into a situation where you'd like to access your information but can't. In these cases a simple web interface that allows you to peek into your data would be more than bliss. Read on to learn how to create your personal web interface to Outlook.

This July I had the privilege to attend the 11th Annual Borland Developer Conference in San Diego. The conference was great in itself, but when I went to write postcards to my friends back home, I noticed that I had forgotten the addresses. I knew they were stored in my Outlook, but there was no way of getting the information, even though I had left my computer on and connected to the internet.

Fortunately, I could call a friend of mine who had most of the addresses I needed, so everything was well. Nonetheless, this incident gave me the idea to write a web interface to Outlook. With such an interface I could use any web browser to read information in my Outlook, including those precious contact cards, from anywhere in the world.

Of course, such a system would be beneficial in the business world, too. For example, how many times have you said to your colleague or customer 'I'll have to get back to you' before making an appointment? Well, now that web access is so common, there really is no excuse for saying that any more.

If you know what I'm talking about, you will definitely appreciate the example application presented in this article. So, let's begin.

## The Architecture

Briefly put, the architecture of the example application, Weblook, is a three-tier architecture. As a web-based application, the first tier is devoted to the web server app, which is an ISAPI (Internet Server API) extension DLL to be precise. The second tier is actually the tier of a helper application that handles all the communications with Outlook. The final tier is, without doubt, Outlook itself.

This kind of architecture was required because ISAPI applications do not generally run under the context of the interactive user, but instead under a system account, depending on the web server configuration. Trying to access Outlook using COM interfaces is difficult under such situations, so I chose to create a simple helper application with which the ISAPI DLL could communicate (curious readers might want to read Microsoft KnowledgeBase article Q156223 for further illumination).

The helper application uses Delphi 5's Office components to access Outlook, although I've written a wrapper class around the component to the keep the code clean. I've tested the helper application (creatively named OutlookHelper) with Outlook 2000, but the application should be compatible with Outlook 98 with only minor modifications.

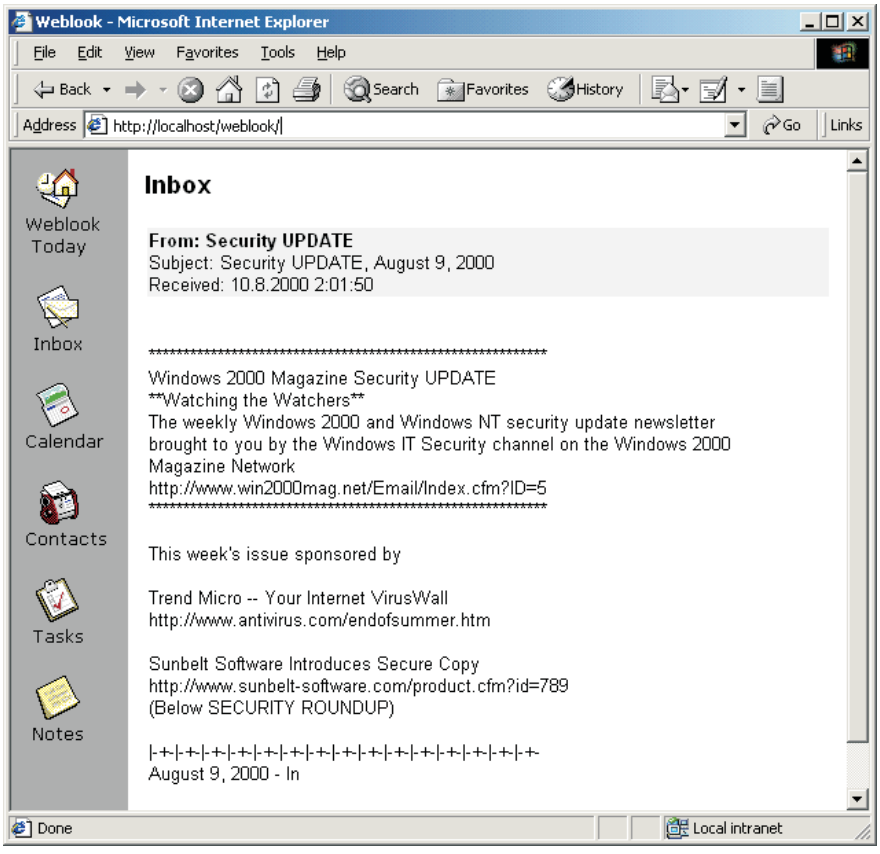
The ISAPI DLL uses Delphi's WebBroker technology. This technique allows the developer to build advanced web applications fast, and I've personally found it to be very reliable. Of course, using an ISAPI DLL requires that you have an ISAPI compatible web server at your disposal.

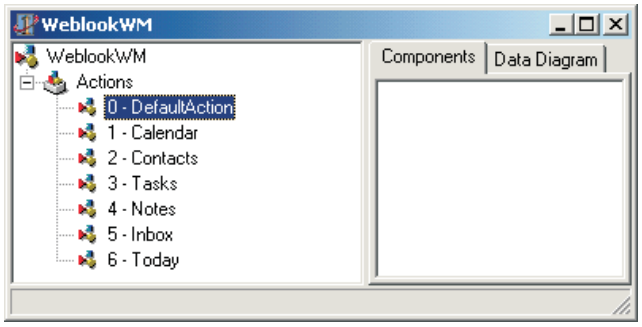
My favorite web server is Microsoft Internet Information Server (IIS) running on Windows NT or Windows 2000. Although you can run ISAPI applications in Windows 95/98, I suggest sticking with NT technology operating systems because of their greater stability.

## Playing Pipes

As you might know, when a process in the 32-bit Windows world loads a DLL into memory, the DLL gets loaded into the address space of the process. In particular, this is

➤ *Figure 1: The Weblook example application running.*






➤ *Figure 2: The ISAPI DLL web module at design-time.*

exactly what happens when IIS (or a similar web server) loads the Weblook DLL into memory in default configuration.

As OutlookHelper is an entirely different application, the ISAPI DLL and the helper application need some way to communicate. The Win32 API provides many different methods for inter-process communication, or IPC. For the example applications presented here, I've chosen to use a method called *named pipes*.

Describing the inner workings of pipes is beyond the scope of this article, but I will still provide you with the information about how bits are put into motion through pipes. If you wish to learn more about pipes, or IPC in general, I suggest reading Brian Long's article *Win32 Inter-Process Communication* in the October 1999 issue.

In the example applications, pipes are used to communicate information about items in Outlook. For example, when the user wants to see the contents of his/her Inbox, the ISAPI DLL sends a textual command to the helper application (I call this a request), for example `INBOX COUNT`.

The helper application will note the request, process it, and finally respond to the request. In the case of the aforementioned request, the reply would simply be the number of mail messages in the Inbox, like this: `4`.

## The ISAPI DLL

When the user wants to see, for instance, his or her calendar, he/she clicks on the `Calendar` link on the Weblook main menu on the leftmost frame of the Weblook interface (see Figure 1). The ISAPI DLL supports six basic commands, namely for displaying the calendar, contacts, tasks, notes and Inbox. Additionally, I've written a 'Weblook Today' command that mimics the Outlook Today command in Outlook, but is somewhat simpler.

Basically, the ISAPI DLL's work is to process a request from the user, contact the helper application for details about an item in Outlook, get the results back and finally process them as appropriate. The DLL must then format an HTML response that will eventually be sent to the user's browser.

The different commands the web DLL supports are divided into a similar number of actions on the web module, as shown in Figure 2. The first action, `Calendar`, has a `PathInfo` of `/calendar`. In effect, this means that when the relative URL `/scripts/weblook.dll/calendar` is used, the ISAPI DLL knows to execute the `OnAction` event handler of the `Calendar` action.

Although the calendar is probably the most interesting action in Weblook, it is also the most complex one to implement. For example, the `Calendar` actions must deal with the additional overhead of formatting a monthly calendar in HTML. For this reason I feel it is better to start with less advanced actions so that you learn how those basic actions work.

## An Action Roundtrip

When the execution of the ISAPI DLL ends in an `OnAction` event handler, most of the actions first send a request to the helper application asking for the number of items in the given folder. For example, the action for displaying the notes in Outlook first sends a `NOTE COUNT` request to the helper application.

When the DLL receives the response from the helper application, it starts to iterate through the items in the folder in question. The event handler for the `Notes` action would, for instance, iterate through all the notes one by one, and construct appropriate HTML code along the way.

Once all items have been iterated, the partially constructed HTML code will be merged with an HTML template. I've chosen to store those templates as string constants in the `HTMLTemplates` unit of the Weblook application. In a commercial application you might want to store the templates in a database.

After the merging has been completed, with a bit of help from the `StringReplace` function, the final HTML code will be sent to the browser using the `TWebResponse` class. The `Response` parameter of the action's `OnAction` event handler will provide an instance of this class.

Usually, the HTML template is written such that all the items found are stored in one large HTML table (`<TABLE>`). This way the code iterating through the items needs only to construct HTML table rows. That is, the code must be aware of how the `<TR>` and `<TD>` HTML tags work. Listing 1 shows you how a typical action works.

## Requests And Responses

As I noted earlier, when the ISAPI DLL wants to send a request to the helper application, it uses pipes.

➤ *Listing 1*

```
procedure TWeblookWM.WeblookWMNotesAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
Var
  iIndex,iCount : Integer;
  strRows       : String;
begin
  iCount := StrToInt(ReadPipeRequestResponse('NOTE COUNT'));
  For iIndex := 0 to iCount-1 do Begin
    strRows := strRows + '  <TR VALIGN="TOP">' + CRLF +
       ReadPipeRequestResponse('NOTE ' + IntToStr(iIndex)) + ' </TR>' + CRLF;
  End;
  Response.Content := StringReplace(cstrNotesHTML,'%rows%',strRows,[]);
end;
```

The Win32 API makes it easy to work with byte-oriented named pipes, as they act almost like files. For example, the API functions `ReadFile` and `WriteFile` work perfectly well with pipes as well as files.

To send a request to the helper application, the ISAPI DLL uses the method named `ReadPipeRequestResponse`, shown in Listing 2. Before any text commands can send through the pipe, it must be opened using the `CreateFile` API function. Because the `CreateFile` function can sometimes fail when trying to open the pipe, I've created a helper method named `CreateNamedPipe` to handle most error situations.

Pipe names are prefixed with a multitude of backslashes, so the name of the pipe used by the example applications is `\\.\pipe\weblook_tdm_version_1`. It goes without saying that both the ISAPI DLL and the helper application must use the exactly same pipe name to be able to communicate.

The `ReadPipeRequestResponse` method first uses a `WriteFile` API call to write the request to the helper application. Then, it calls the `ReadFile` function to read the response from the helper application. The `ReadFile` is a blocking function, meaning that it will not return until there is something to read. This is mandatory, as the helper application needs some time to process the request.

To give you a concrete example, let's assume that the user clicks on the `Notes` icon to see the notes in his/her Outlook. First, the `NOTE COUNT` command would be needed to get the count of notes. The `WriteFile` function would transmit the command to the helper application. While the ISAPI DLL is blocking in the `ReadFile` call, the helper application is querying Outlook to get the count.

Then, the helper application returns the appropriate number (as a text string), and sends it to the other side of the pipe, to the DLL. This causes the `ReadFile` function to return, and the ISAPI DLL now has the number of notes. This allows it to begin the note iteration in similar fashion.

## Constructing The Calendar

As I noted previously, the command displaying the calendar is the most complex to implement. You should now know how the simpler commands are implemented, so it is time to see what's inside the `OnAction` event handler of the `Calendar` action. It is shown in Listing 3.

The first step in creating the calendar HTML code is determining the month and year for which the calendar is to be displayed. I've written the Weblook application so that the user can use `Next` and `Previous` links to navigate the calendar one month at a time.

Because of this browsing capability, the application must read the given month and year numbers from the URL. The request URL is automatically parsed by Delphi's WebBroker architecture so that the example application can simply read the `Request.QueryFields` property. If there are errors in the month or year numbers (or they don't exist), the code uses the current month and year.

➤ *Listing 2*

```
Function TWeblookWM.ReadPipeRequestResponse(strRequest : String) : String;
Var
  iBW,iBR   : Cardinal;
  cResponse : Array[0..1023] of Char;
Begin
  CreateNamedPipe;
  Result := '';
  If (Not WriteFile(hPipe,Pointer(strRequest)^, Length(strRequest),iBW,nil)) Then
    Raise Exception.Create('Cannot write to pipe: ' +
      SysErrorMessage(GetLastError));
  If ReadFile(hPipe,cResponse,SizeOf(cResponse)-1,iBR,nil) Then Begin
    cResponse[iBR] := #0; { properly terminate the string }
    Result := String(cResponse);
  End Else Raise Exception.Create('Cannot read from pipe: ' +
    SysErrorMessage(GetLastError));
  CloseHandle(hPipe);
End;
```

➤ *Listing 3*

```
procedure TWeblookWM.WeblookWMCalendarAction(Sender:
  TObject; Request: TWebRequest; Response: TWebResponse;
  var Handled: Boolean);
Var
  iCol,iRow,iDay  : Integer;
  strHTML,strRows : String;
  wY,wM,wD        : Word;
begin
  DecodeDate(Date,wY,wM,wD);
  Try
    iYear := StrToInt(Request.QueryFields.Values['year']);
    If (iYear < ciFirstYear) Or (iYear > ciLastYear) Then
      iYear := wY;
    iMonth := StrToInt(Request.QueryFields.Values['month']);
    If (iMonth < 1) Or (iMonth > 12) Then
      iMonth := wM;
  Except
    iYear := wY;
    iMonth := wM;
  End;
  For iCol := 0 to 6 do Begin
    For iRow := 1 to 6 do
      cgCalendarGrid[iCol,iRow] := ' ';
  End;
  iCol := DayOfWeek(EncodeDate(iYear,iMonth,1))-1;
  iCol := (iCol-ciStartOfWeek) mod 7;
  If (iCol < 0) Then
    iCol := 7+iCol;
  iRow := 1;
  For iDay := 1 to
    MonthDays[IsLeapYear(iYear),iMonth] do Begin
    cgCalendarGrid[iCol,iRow] := IntToStr(iDay) + '<BR>' +
      ReadPipeRequestResponse('CALENDAR ' +
      IntToStr(Trunc(EncodeDate(iYear,iMonth,iDay))));
    Inc(iCol);
    If (iCol > 6) Then Begin
      iCol := 0;
      Inc(iRow);
    End;
  End;
  strHTML := StringReplace(cstrCalendarHTML,'%month%',
    LongMonthNames[iMonth]+' '+IntToStr(iYear),[]);
  strHTML := StringReplace(strHTML,'%prev%',
    GetMonthLinkHTML(iMonth,iYear,-1),[]);
  strHTML := StringReplace(strHTML,'%next%',
    GetMonthLinkHTML(iMonth,iYear,+1),[]);
  strHTML := StringReplace(strHTML,'%current%',
    GetMonthLinkHTML(wM,wY,0),[]);
  For iRow := 1 to 6 do Begin
    strRows := strRows+'  <TR>' + CRLF;
    For iCol := 0 to 6 do Begin
      strRows := strRows+
        '    <TD WIDTH="102"><FONT SIZE="2" ' +
        'FACE="Arial, Helvetica, sans-serif">' +
        cgCalendarGrid[iCol,iRow] + '</FONT></TD>' + CRLF;
    End;
    strRows := strRows+'  </TR>'+CRLF;
  End;
  strHTML := StringReplace(strHTML,'%rows%',strRows,[]);
  Response.Content := strHTML;
end;
```

```
Procedure TPipeServerThread.Execute;
Var
  hPipe       : THandle;
  bConnected  : Boolean;
  cRequest    : Array[0..1024-1] of Char;
  iBR,iBW     : Cardinal;
  strResponse : String;
Begin
  hPipe := CreateNamedPipe(strPipeName,Pipe_Access_Duplex,
    Pipe_Type_Byte Or Pipe_Wait,1,
    4096,4096,nmpWait_Use_Default_Wait,nil);
  If (hPipe = Invalid_Handle_Value) Then Begin
    strMessage := 'Cannot create pipe: ' +
      SysErrorMessage(GetLastError);
    Synchronize(LogMessage);
    Exit;
  End;
  CoInitialize(nil);
  While (Not Terminated) do Begin
    bConnected := ConnectNamedPipe(hPipe,nil);
    If ((Not bConnected) And
      (GetLastError = Error_Pipe_Connected)) Then
      bConnected := True;
    If bConnected Then Begin
      If (Not ReadFile(hPipe,cRequest,
        SizeOf(cRequest),iBR,nil)) Then Begin
        strMessage := 'Cannot read from pipe: ' +
          SysErrorMessage(GetLastError);
        Synchronize(LogMessage);
      End Else Begin
        cRequest[iBR] := #0;
        strMessage := 'Got request "'+cRequest+'"';
        Synchronize(LogMessage);
        If (cRequest = 'FREE') Then Begin
          strMessage := 'Freeing Outlook objects';
          Synchronize(LogMessage);
          ooOutlook.Free;
        End Else Begin
          strResponse := ProcessPipeRequest(cRequest);
          strMessage := 'Responding "'+strResponse+'"';
          Synchronize(LogMessage);
          WriteFile(hPipe,Pointer(strResponse)^,
            Length(strResponse),iBW,nil);
          FlushFileBuffers(hPipe);
        End;
        DisconnectNamedPipe(hPipe);
      End;
    End;
  End;
  CloseHandle(hPipe);
  CoUninitialize;
  strMessage := 'Pipe thread terminated';
  Synchronize(LogMessage);
End;
```

➤ *Listing 4*

After the month and year have been checked for validity, the code starts to fill an array variable named `cgCalendarGrid` with the calendar entries for the month for which the calendar is going to be displayed. The ` ` HTML entity (`nbsp` means non-blocking space) is needed because otherwise the user's web browser would render empty table cells too plain.

Anyway, filling the calendar requires some arithmetic to make the calendar appear correctly and take into account leap years, etc. The `SysUtils` unit helps a lot when filling the calendar. For example, the `MonthDays` array can be used to quickly get the number of days in a month with a bit of help from the `IsLeapYear` function:

```
iDays := MonthDays[IsLeapYear(
  iYear),iMonth];
```

## More Of The Calendar

Because parsing date formats can be difficult if the regional settings change, I've chosen to use a universal mechanism to represent a date when the ISAPI DLL and the helper application are communicating. When the DLL must know the calendar entries for a given date, my code doesn't send the date a string literal like `08/10/00`, but instead converts the date to an ordinal number like `32673`.

If you look at the code closely, you will see the following snippet:

```
IntToStr(Trunc(EncodeDate(
  iYear,iMonth,iDay)))
```

The `EncodeDate` function will return a value of type `TDateTime`, which is actually defined as a `Double` in SYSTEM.PAS. The `TDateTime` type holds a number of days that have elapsed since 30th of December 1899. By using the `Trunc` function the code can get the integral part of the date, thereby wiping off all digits on the right side of the decimal point.

Now let's get back to the calendar. When the ISAPI DLL sends a command like `CALENDAR 38765` to the helper application, the application will respond with the appointments on the given date. Then, the DLL can continue to fill the HTML table cell with the appropriate information.

After the calendar table has been generated, the event handler still needs to create URLs with which the user can request to see the previous, current or the next month in the calendar. After several calculations, the helper function `GetMonthLinkHTML` will produce an URL like the following:

```
/scripts/weblook.dll/
  calendar?year=2000&month=7
```

## The Helper Application

You should now have a good understanding how the ISAPI DLL works, so it's time to turn to the helper application, OutlookHelper. One of the key functions of this application is to read the requests sent by the ISAPI DLL through the named pipe. This is the function I'm going to describe next.

If you recall, the API functions `ReadFile` and `WriteFile` operate generally in synchronous (blocking) mode. So, if the helper application had called either of these functions in the main thread, the whole helper application would become unresponsive to the user.

To avoid such spiteful behavior, I've written the helper application so it uses another thread to handle the pipe communications. This way the user interface will still stay operational while the threaded code will continue to monitor the pipe normally. The overridden `Execute` method of the `TThread` class can be seen in Listing 4.

The first thing done by the pipe thread is to create the server end of the pipe with the `CreateFile` API call. The difference to the similar call in the ISAPI DLL is that the helper application doesn't pass in the `OPEN_EXISTING` flag to `CreateFile`.

After creating the pipe, the thread must initialize the COM architecture with a call to the `CoInitialize` API function. Although the main thread has had a chance to do this, the pipe thread must nonetheless initialize COM, as the initialization occurs on a per-thread basis. To balance the initialization call, a call to `CoUninitialize` is needed at the end.

## Processing A Pipe Request

If you take a look at the code in Listing 4, you will quickly see that

reading and writing to the server end of the pipe occurs using the same `ReadFile` and `WriteFile` API calls as in the ISAPI DLL. However, there are two API calls I'd like to point out.

After the pipe request has been processed and the response has been written to the `strResponse` variable, a call to `WriteFile` writes data to the pipe. However, because Windows does internal buffering on many I/O operations (including pipes), the code uses a call to the `FlushFileBuffers` function. This makes sure the data actually flows to the client (the ISAPI DLL). Finally, the pipe is disconnected using a call to `DisconnectNamedPipe`.

A request coming through a pipe is processed by the `ProcessPipeRequest` method (see Listing 5). This method simply divides the request into two parts, the 'verb' and the parameter. Then, it constructs an instance of my `TOutlookObjects` class, which handles all communications with Outlook itself.

Because some requests, like the `NOTE` request, can have two functions based on the parameter (either a `COUNT` or a note number like `2`), a simple wrapper function is used to get the response, as shown in Listing 6. Here, the variable `ooOutlook` is an instance of the `TOutlookObjects` class.

## Communicating With Outlook

As I noted previously, the `TOutlookObjects` class I've written handles all the communications between Outlook and the helper application. This class is actually a wrapper around Delphi's `TOutlookApplication` component, which can be found from the `Servers` Component Palette tab.

In the constructor of the class, the code simply creates an instance of the Outlook component, and then attaches itself to appropriate interfaces that Outlook supports. Going through the Outlook object hierarchy is beyond the scope of this article, but a good introduction to the subject can be found from Microsoft's website at http://msdn.microsft.com. For instance, the article *Automating Microsoft Outlook 97* by Mike Gilbert is a good one to start with.

Once the appropriate interfaces have been acquired, the helper class is ready for action. For example, if the user wants to see his/her notes in Outlook, the ISAPI DLL will send a request to the helper application. The application will eventually route the call to the helper class' `GetNoteDetails` method, which will look like the one in Listing 7.

As you can see, the code simply uses the Outlook interfaces to construct an HTML code fragment that will contain all the necessary details about the note (or any other item type in question). Again, constructing the calendar is more cumbersome than handling the other item types.

When processing the calendar, the helper class will first construct a calendar cache for the month that the user wants to see. I wanted to implement such a cache because otherwise querying the calendar would be too slow. Because Outlook doesn't sort appointments in the calendar by date, it is difficult to quickly find the appointments on a given date.

By using a cache, the code can scan through the appointments only once, and then quickly see the appointments on any day of the month in question. Anyway, Weblook will only display a month at a time, so this kind of simple caching is sufficient.

## Deploying Weblook

As a multi-tiered web application, there are many steps in deploying Weblook, although none of the steps are very complicated in themselves.

The first step in the deployment is to find a web server and create a virtual directory on it. I'd suggest you create a directory named Weblook, but you can choose any name you wish.

After copying the example application source code from this month's disk, copy the three HTML files and all the .GIF images to this virtual directory (make sure index.html can act as the default document for this directory). Next, you need to copy the file WEBLOOK.DLL (after compiling it first, if necessary) to the scripts directory of your web server.

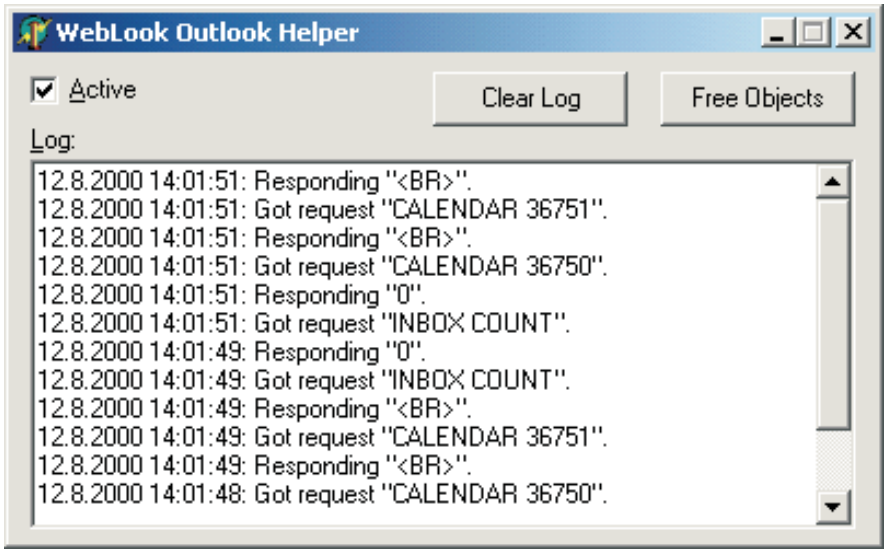
Please note that the DLL is written so that it assumes the virtual directory name for the scripts directory is actually `/scripts`, which is the default in IIS. If you are using any other web server, you might either want to create a new virtual directory with this name, or alternatively modify the DLL to use the appropriate directory.

```
Function TPipeServerThread.ProcessPipeRequest(strRequest : String) : String;
Var
  iPos     : Integer;
  strVerb  : String;
  strParam : String;
Begin
  iPos := Pos(' ',strRequest);
  strVerb := Copy(strRequest,1,iPos-1);
  strParam := Copy(strRequest,iPos+1,Length(strRequest)-iPos);
  Try
    If HelperMainForm.HelperActive.Checked Then Begin
      If (ooOutlook = nil) Then ooOutlook := TOutlookObjects.Create;
      If (strVerb = 'CALENDAR') Then Result := GetCalendarResponse(strParam)
      Else If (strVerb = 'CONTACT') Then Result := GetContactResponse(strParam)
      Else If (strVerb = 'TASK') Then Result := GetTaskResponse(strParam)
      Else If (strVerb = 'NOTE') Then Result := GetNoteResponse(strParam)
      Else If (strVerb = 'INBOX') Then Result := GetInboxResponse(strParam)
      Else Result := 'Unknown verb.';
    End Else
      Result := 'Helper application not active';
  Except
    On E : Exception do Result := E.Message;
  End;
```

➤ *Above: Listing 5*

➤ *Below: Listing 6*

```
Function TPipeServerThread.GetNoteResponse(strIndex : String) : String;
Begin
  If (strIndex = 'COUNT') Then
     Result := IntToStr(ooOutlook.GetNoteCount)
  Else
     Result := ooOutlook.GetNoteDetails(StrToInt(strIndex));
End;
```




➤ *Figure 3: The OutlookHelper application communicating with Outlook.*

Once the DLL has been put into the correct place, compile and run the helper application, OUTLOOKHELPER.EXE. Please note that this application must run on the same machine as the web server and on the same machine as your Outlook. That is, the web server, the helper application and Outlook must all be running on the same computer. It's worth pointing out that this almost certainly means you won't be able to use this approach on an ISP's web server: the programs need to be on a machine in your office, with proper firewall protection from the rest of your machines of course!

As a simple piece of software, the helper application needs just to be running for it to be active. Outlook, however, doesn't need to be running, as the helper application will automatically launch it behind the scenes if necessary.

## Testing Weblook

Once you have all the three pieces (the web server, the DLL and OutlookHelper) set, it is time to fire up your web browser. First, try to access your web server by typing `localhost` at the URL field (or the name of your web server, if it is a different computer). If you get a decent reply, try entering `localhost/weblook`. This should display the welcome page of Weblook.

Now, try to click any of the familiar Outlook icons on the left. If everything goes smoothly, you should see, for example, the contents of your Inbox. If you get an error message saying *Cannot open pipe*, you most probably have forgotten to activate the OutlookHelper application.

If the helper application is properly activated, you should see some text appearing in the log, as seen in Figure 3. If something goes wrong in the helper application, you should see the error messages appearing in the log. If you wish to temporarily stop the OutlookHelper application from responding (for security reasons, for instance), just make sure the `Active` checkbox is not selected on the OutlookHelper main window.

After you have done testing, be sure to click the `Free Objects` button on OutlookHelper to stop the pipe server thread and disconnect from Outlook. After this has been done you can freely shut down OutlookHelper and your web server.

## Conclusion

Having got this far, I hope you have a good understanding of how my example application works. I suggest you copy the example code from the disk and study it. When it comes to building distributed web applications, there's always something interesting to learn, at least for me.

If you wish to improve Weblook, you might want to allow the user to view even more folders that Outlook supports, for example the `Journal`, `Drafts`, and `Deleted Items`. And if you're really anxious, you might create a version of Weblook that would allow the user to add and modify existing items!

Email is always welcome, so if you want customize Weblook or discuss the architecture in more detail, don't hesitate to contact me.

Jani Järvinen works as a technical support person for Borland products. He specializes in internet and Windows API technologies. You can reach him at janij@dystopia.fi

➤ *Listing 7*

```
Function TOutlookObjects.GetNoteDetails(iNoteNumber : Integer) : String;
Var
  niItem : NoteItem;
Begin
  niItem := NoteItem(itNotes.Item(iNoteNumber+1));
  Result :=
    ' <TD WIDTH="30%"><FONT FACE="Arial, Helvetica, sans-serif" SIZE="2"><B>'+
    niItem.Subject+'</B></FONT></TD>'+CRLF+
    ' <TD WIDTH="70%"><PRE><FONT FACE="Arial, Helvetica, sans-serif" SIZE="2">'+
    niItem.Body+'</FONT></PRE></TD>'+CRLF;
End;
```