# How to handle drag and drop in a TWebBrowser control (part 2 of 4)

*General solution*

## Overview of Solution

In *article #24* we investigated how to handle OLE drag and drop by implementing the *IDropTarget* interface. *TWebBrowser* uses this method of accepting dragged objects. This means that we need to implement *IDropTarget* and find a way to get the browser control to use our implementation instead of its own.

So how do we supply our own drag drop handler to *TWebBrowser*? The answer will come as no surprise if you have read some of my earlier *TWebBrowser* articles. We must implement the *IDocHostUIHandler* interface and use its *GetDropTarget* method to hook up our custom *IDropTarget* implementation with *TWebBrowser*.

**Further reading**

For detailed information about creating a container for *TWebBrowser* and implementing *IOleClientSite* and *IDocHostUIHandler* please see "*How to customise the TWebBrowser user interface*".

Once we have implemented *IDocHostUIHandler* we must tell the browser control about it. Whenever *TWebBrowser* is instantiated it tries to find a *IDocHostUIHandler* implementation in its host (or container). This is done by querying the host's *IOleClientSite* interface. This also means we need to implement *IOleClientSite* in the same object as *IDocHostUIHandler*.

So, to summarise we will:

1. Create a container for the web browser control that implements *IDocHostUIHandler* and *IOleClientSite* and associate this container with the web browser control.
2. Create an implementation of *IDropTarget* that handles drag and drop as we would like.
3. Use the *IDocHostUIHandler.GetDropTarget* method to notify the browser control about our *IDropTarget* implementation.

## Generic Implementation of Solution

### Creating the Web Browser Container

Luckily, in "*How to customise the TWebBrowser user interface*" we developed a reusable, generic browser container class – *TNulWBContainer* – that provided minimal implementations of *IDocHostUIHandler* and *IOleClientSite*. A do-nothing implementation of *IDocHostUIHandler* was provided that leaves the browser control's behaviour unchanged. This base class also takes care of associating our object with the browser control as its container. We will reuse that code here and define a sub class named *TWBDragDropContainer* that overrides the default implementation of *IDocHostUIHandler.GetDropTarget* to ensure the browser control hooks into our *IDropTarget* implementation.

**Where's the Source?**

Source for *TNulWBContainer* and *IDocHostUIHandler* is not listed in this article. However the code is provided in this article's *demo code*, in the `UNulContainer` and `IntfDocHostUIHandler` units.

We will make the new class as general as possible by providing a *DropTarget* property that can be set to any object that implements *IDropTarget*. This way the class' user can change the drag drop behaviour of the browser control simply by assigning a different object to the *DropTarget* property.

*Listing 2* shows the whole of the `UWBDragDropContainer` unit that contains our implementation of *TWBDragDropContainer*.

```
unit UWBDragDropContainer;
```

```
interface

uses
  ActiveX,
  IntfDocHostUIHandler, UNulContainer;  // from article #18

type

  TWBDragDropContainer = class(TNulWBContainer,
    IUnknown, IOleClientSite, IDocHostUIHandler
  )
  private
    fDropTarget: IDropTarget;
  protected
    function GetDropTarget(const pDropTarget: IDropTarget;
      out ppDropTarget: IDropTarget): HResult; stdcall;
  public
    property DropTarget: IDropTarget
    read fDropTarget write fDropTarget;
  end;

implementation

{ TWBDragDropContainer }

function TWBDragDropContainer.GetDropTarget(
  const pDropTarget: IDropTarget;
  out ppDropTarget: IDropTarget): HResult;
begin
  if Assigned(fDropTarget) then
  begin
    // We are handling drag-drop: notify browser of drop target object
    ppDropTarget := fDropTarget;
    Result := S_OK;
  end
  else
    // We are not handling drag-drop: use inherited default behaviour
    Result := inherited GetDropTarget(pDropTarget, ppDropTarget);
end;

end.
```

*Listing 2*

Our new class includes a reimplementation of the inherited *GetDropTarget* method and defines the new *DropTarget* property.

*GetDropTarget* checks to see if a drop target handler is assigned to the *DropTarget* property. If so we return a reference to our drop target object via the method's *ppDropTarget* parameter and return *S_OK*. Conversely if *DropTarget* is unassigned then we just call the inherited *GetDropTarget* method to get the default behaviour. Doing all the hard work in the base class has made this class very easy to implement.

# Implementing IDropTarget

*Article #24*, "How to receive data dragged from other applications", discussed how to implement *IDropTarget*. Implementation depends on what type of data the application will accept, so we won't discuss that further in this section. We will come back to look a couple of specific examples later.

# Some Boilerplate Code for the Main Form

To use our web browser container object we need to create it in the main form and set its *DropTarget* property. *Listing 3* shows the code that needs to be added to the main form.

```
unit FmDemo3;

interface

uses
  ..., UWBDragDropContainer, ...
```

```
type
  TForm1 = class(TForm)
    ...
    WebBrowser1: TWebBrowser;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    ...
  private
    ...
    fWBContainer: TWBDragDropContainer;
    ...
  end;

var
  Form1: TForm1;

implementation

uses
  ActiveX;

{$R *.dfm}

...

procedure TForm1.FormCreate(Sender: TObject);
begin
  OleInitialize(nil);
  fWBContainer := TWBDragDropContainer.Create(WebBrowser1);
  fWBContainer.DropTarget := TMyDropTarget.Create;
  WebBrowser1.Navigate('about:blank');
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  fWBContainer.DropTarget := nil;
  OleUninitialize;
  FreeAndNil(fWBContainer);
end;

...

end.
```

*Listing 3*

In the form's *OnCreate* event handler We create an instance of *TWBDragDropContainer* and assign its *DropTarget* property with our drop target object (see below for examples of such an object). Next we navigate to `about:blank` just to make sure the browser control has created a document object. The drop target object won't be noticed by the browser until this has been done. Of course you could load any document here – and it would be better practice to wait for the document to finish loading before continuing. We also intialise OLE as before.

In *FormDestroy* we simply clear the container object's *DropTarget* property before freeing the container itself. It is probably not necessary to set *DropTarget* to **nil**, but it is neater, and may be safer, to explicitly release it. Again we unintialise OLE.

That takes care of the implementation of *IDocHostUIHandler* and how to notify the browser control of it's implementation. We can now *move on* to look at exactly how we implement the drop target. This is done by means of a couple of case studies.

This article is copyright © Peter Johnson 2007-2013

SOME RIGHTS RESERVED

Licensed under a *Creative Commons License*.

*Copyright* © Peter Johnson (*DelphiDabbler*) 2002-2020