How to load and save documents in TWebBrowser in a Delphi-like way

Contents

Why do it

I use the *TWebBrowser* control quite a lot. But, since the control is simply a wrapper round the Microsoft® control, it doesn't really do things the "Delphi way".

If, like me, you've found yourself saving dynamically generated HTML code to disk just so you can access it using *TWebBrowser.Navigate()* you probably know what I mean. You wouldn't have to do such a thing with a native Delphi control such as a *TMemo* – you would simply access a relevant property like *TMemo.Lines[]* or use a method like *TMemo.LoadFromStream*. You can also do this with *TWebBrowser*, but its not straightforward – not very "Delphi" – you have to query and manipulate interfaces and all sorts of stuff. It's all so very COM!

So, I decided to create a wrapper class for *TWebBrowser* that makes navigating, loading and saving a whole lot easier and more intuitive. This article walks through the development of that class and examines some of the key techniques for working with *TWebBrowser* along the way.

A word of caution before we get started. The code I'll present here is for illustration purposes only. Don't expect it to be perfect for production code, although you should be able to use it as a basis. Please feel free to take the core unit from the demo and modify, specialise or generalise it for your own purposes. It's open source and licensed under the MPL / GLP / LGPL tri-license which should suit most open-source hackers.

Requirements

The approach we will take is to develop a wrapper class for *TWebBrowser* rather than derive a new class from it. Let's first decide of the main functions of the web browser we want to be able to access easily – these will be our requirements for the wrapper class. Here's the list I drew up:

- Load HTML code into an existing document from a local file, a stream or a string.
- Navigate to a URL (as now) but without worrying about remembering the `file://` protocol when loading an local file and with easy access to HTML resources using the `res://` protocol.
- Programmatically save the browser's content as HTML source code. We should also be able to store the content in a string or write it to a stream or file.
- Provide access to the underlying *TWebBrowser* object so that we can use it directly to access any functionality not provided by the new code.

Now we're living in a Unicode world the following extra requirements will be added:

- Be able to read and write HTML code in ANSI or Unicode encodings. And be able to specify the encoding used when saving a document or when loading from a string.
- Get the encoding (character set) used by the browser object for the current document.

From this list you can see we are focussing on navigating, loading and saving documents.

Implementation

Now we have our specification we will approach implementation in two stages:

- **Stage 1:** Basic implementation of the first set of requirements. This code will be suitable for most Delphi compilers.
- **Stage 2:** Add on the required Unicode support. This will require Delphi 2009 or later.

Stage 1: Basic Implementation

This first stage won't worry about Unicode support other than that needed to make the code compile and work with Delphi 2009 and later.

Here is an outline of a class that, once implemented, meets our basic requirements:

```
type
  TWebBrowserWrapper = class(TObject)
  private
    fWebBrowser: TWebBrowser; // wrapped control
  protected
    procedure InternalLoadDocumentFromStream(const Stream: TStream);
    procedure InternalSaveDocumentToStream(const Stream: TStream);
  public
    constructor Create(const WebBrowser: TWebBrowser);
    procedure LoadFromFile(const FileName: string);
    procedure LoadFromStream(const Stream: TStream);
    procedure LoadFromString(const HTML: string); overload;
    function NavigateToLocalFile(const FileName: string): Boolean;
    procedure NavigateToResource(const Module: HMODULE;
      const ResName: PChar; const ResType: PChar = nil); overload;
    procedure NavigateToResource(const ModuleName: string;
      const ResName: PChar; const ResType: PChar = nil); overload;
    procedure NavigateToURL(const URL: string);
    function SaveToString: string;
    procedure SaveToStream(const Stm: TStream); overload;
    procedure SaveToFile(const FileName: string); overload;
    property WebBrowser: TWebBrowser read fWebBrowser;
  end;
```

Listing 1

The first thing to notice is the *WebBrowser* property that enables access to the wrapped control. The public methods fall naturally into several groups and, rather than explaining the purpose of each method now, we will look at them in groups. The protected "helper" methods will be discussed along with the public methods they service.

Constructor

The constructor is very simple – it just stores a reference to the *TWebBrowser* control that the object is wrapping. This control reference is passed as a parameter to the constructor:

```
constructor TWebBrowserWrapper.Create(const WebBrowser: TWebBrowser);
begin
  inherited Create;
  fWebBrowser := WebBrowser;
end;
```

Listing 2

Navigation Methods

We have declared three different methods for navigation:

- *NavigateToURL* – A thin wrapper around the existing *TWebBrowser.Navigate* method that only deals with standard URLs and intelligently sets the cache and history flags. The main difference between this method and the underlying control's method is that our method blocks until the required document has completely downloaded.
- *NavigateToLocalFile* – This convenience method simply adds the required `file://` protocol to the URL before calling *NavigateToURL*.
- *NavigateToResource* – Loads HTML code from the program's (or other module's) resources into the browser. Overloaded versions of this method allow resources to be accessed by instance handle (like the *TResourceStream* constructors) or by providing the name of the module containing the resources.

NavigateToLocalFile and both versions of *NavigateToResource* work by creating the required URL from the parameters passed to them and then calling *NavigateToURL* to do the actual navigation. Let's first look at how *NavigateToURL* handles the navigation then come back to look at how the other routines put the URLs together.

```
procedure TWebBrowserWrapper.NavigateToURL(const URL: string);
  // ------------------------------------------------------------------------
  procedure Pause(const ADelay: Cardinal);
  var
    StartTC: Cardinal;  // tick count when routine called
  begin
    StartTC := Windows.GetTickCount;
    repeat
      Application.ProcessMessages;
    until Int64(Windows.GetTickCount) - Int64(StartTC) >= ADelay;
  end;
  // ------------------------------------------------------------------------
var
  Flags: OleVariant;  // flags that determine action
begin
  // Don't record in history
  Flags := navNoHistory;
  if AnsiStartsText('res://', URL) or AnsiStartsText('file://', URL)
    or AnsiStartsText('about:', URL) or AnsiStartsText('javascript:', URL)
    or AnsiStartsText('mailto:', URL) then
    // don't use cache for local files
    Flags := Flags or navNoReadFromCache or navNoWriteToCache;
  // Do the navigation and wait for it to complete
  WebBrowser.Navigate(URL, Flags);
  while WebBrowser.ReadyState <> READYSTATE_COMPLETE do
    Pause(5);
end;
```

Listing 3

This method essentially falls into two parts. Firstly we decide whether to use the browser's cache to access the document. The decision is based on whether the document is stored locally or is on the internet. We simply check the start of the URL string for some known local protocols etc. and don't use the cache if the URL conforms to one of these types. It would be a simple matter to adapt the method by adding a default parameter to let the user of the code specify what if any caching should take place (this is left as an exercise). The final part of the routine simply uses the browser object's *Navigate()* method to load the resource into the

document. We then go into a loop and wait for the document to load completely. The local *Pause* procedure does a busy wait, polling the message queue for about 5ms at a time.

Now let us review the specialised navigation methods. The simplest of these is the *NagivateToLocalFile* method. This method simply checks if the file exists and, if so, prefixes the given file name with the `file://` protocol then calls *NavigateToURL*. If the file doesn't exist no action is taken. A boolean value indicating whether the file exists is returned. You may prefer to modify the method to raise an exception when the file does not exist.

```
function TWebBrowserWrapper.NavigateToLocalFile(
  const FileName: string): Boolean;
begin
  Result := FileExists(FileName);
  if Result then
    NavigateToURL('file://' + FileName)
end;
```

Listing 4

NavigateToResource is slightly more complicated in that we need to create the required URL using the IE specific `res://` protocol URL. We discussed this protocol in *article #10* where we also developed some functions to return `res://` formatted URLs. (We will re-use these functions later).

Two overloaded methods are provided. Both methods create the required URL for a given module, resource name and an optional resource type. They then call *NavigateToURL* to do the actual navigation. The overloaded methods vary in the way the module is described. The first method accepts the handle of a loaded module (pass *HInstance* to access the current program). The second method is simply passed a module name as a string. If the resource type parameter is omitted then it is left out of the URL – the `res://` protocol simply defaults to expecting an *RT_HTML* (*=MakeIntResource(23)*) resource in such cases. Here are the methods:

```
procedure TWebBrowserWrapper.NavigateToResource(const Module: HMODULE;
  const ResName, ResType: PChar);
begin
  NavigateToURL(MakeResourceURL(Module, ResName, ResType));
end;

procedure TWebBrowserWrapper.NavigateToResource(const ModuleName: string;
  const ResName, ResType: PChar);
begin
  NavigateToURL(MakeResourceURL(ModuleName, ResName, ResType));
end;
```

Listing 5

As can be seen, the methods simply rely on overloaded versions of the *MakeResourceURL* function. The implementation of these functions is *described in article #10*. In both cases we could improve the methods by checking that the required resources exist and raising an exception or returning false if not. This is left as an exercise (*hint:* use the Windows *FindResource* function to check for the resource's existence).

The *MakeResourceURL* functions are included in the demo source code that accompanies this article.

Document Loading Methods

There are three methods that load new content into an existing document:

- *LoadFromString* – Replaces the current document with the HTML code stored in a string.
- *LoadFromFile* – Replaces the current document with the HTML code read from a file.
- *LoadFromStream* – Replaces the current document with the HTML code read from a stream.

At first sight *LoadFromFile* is similar to *NavigateToLocalFile*, but *NavigateToLocalFile* reads a file into the browser, creating a new document whereas *LoadFromFile* requires that a document already exists and replaces it's HTML code – i.e. the HTML is changed dynamically.

Both *LoadFromString* and *LoadFromFile* simply create a suitable stream and call *LoadFromStream*, which in turn uses the protected *InternalLoadDocumentFromStream* method to perform the actual loading of the

code. Let's first look at the stream and string methods, which are very similar:

```
procedure TWebBrowserWrapper.LoadFromFile(const FileName: string);
var
  FileStream: TFileStream;
begin
  FileStream := TFileStream.Create(
    FileName, fmOpenRead or fmShareDenyNone
  );
  try
    LoadFromStream(FileStream);
  finally
    FileStream.Free;
  end;
end;

procedure TWebBrowserWrapper.LoadFromString(const HTML: string);
var
  StringStream: TStringStream;
begin
  StringStream := TStringStream.Create(HTML);
  try
    LoadFromStream(StringStream);
  finally
    StringStream.Free;
  end;
end;
```

Listing 6

As can be seen this is all quite straightforward if you're used to using *TStream*s. *LoadFromFile* opens a read-only *TFileStream* onto the file and passes the stream to *LoadFromStream*. Similarly *LoadFromString* uses the very useful *TStringStream* class to open a stream that can read the HTML code string.

LoadFromStream itself is quite straightforward because it hands most of its work off to *InternalLoadFromStream*:

```
procedure TWebBrowserWrapper.LoadFromStream(const Stream: TStream);
begin
  NavigateToURL('about:blank');
  InternalLoadDocumentFromStream(Stream);
end;
```

Listing 7

The main thing to note here is that we must ensure there is a document present in *TWebBrowser* since, as we will see in a moment, we need it in order to load the code from the stream. We do this by navigating to the special `about:blank` document, which simply creates a blank document in the web browser control. Once we have our blank document we load the stream into it using *InternalLoadDocumentFromStream*, which is defined below:

```
procedure TWebBrowserWrapper.InternalLoadDocumentFromStream(
  const Stream: TStream);
var
  PersistStreamInit: IPersistStreamInit;
  StreamAdapter: IStream;
begin
  if not Assigned(WebBrowser.Document) then
    Exit;
  // Get IPersistStreamInit interface on document object
  if WebBrowser.Document.QueryInterface(
    IPersistStreamInit, PersistStreamInit
  ) = S_OK then
  begin
    // Clear document
    if PersistStreamInit.InitNew = S_OK then
    begin
      // Get IStream interface on stream
      StreamAdapter:= TStreamAdapter.Create(Stream);
      // Load data from Stream into WebBrowser
      PersistStreamInit.Load(StreamAdapter);
    end;
```

```
  end;
end;
```

Listing 8

And this is where it gets more complicated – for the first time we have to mess around with the COM stuff.

We check that the web browser control's document object is available and bail out if not. We then check to see if the document supports the *IPersistStreamInit* interface, getting a reference to the supporting object. *IPersistStreamInit* is used to effectively "clear" the document object (using the interface's *InitNew* method). If this succeeds we finally load the stream's content into the document by calling the *IPersistStreamInit.Load* method.

IPersistStreamInit.Load accepts a stream object, but the stream it expects is a COM one that must support the *IStream* interface. Since *TStream* does not natively support this interface, we have to find some way to provide it. *TStreamAdapter* from Delphi's `Classes` unit comes to the rescue here – this object implements *IStream* and translates *IStream*'s method calls into equivalent calls onto the *TStream* object that it wraps. We create the needed *TStreamAdapter* object by passing a reference to our stream in its constructor. Finally, we pass the adpated stream to *IPersistStreamInit.Load,* and we're done.

If you're wandering why we don't free *StreamAdapter* and *PersistStreamInit* it's because they are both interfaced objects and will be automatically destroyed at the end of the method by Delphi's built in interface reference counting. Note that even though the stream adapter class is freed, the underlying *TStream* object continues to exist, which is what we want.

Note that the browser control interprets the encoding of the stream sets the document's character set accordingly. However, the character set can also be specified in HTML code.

Document Saving Methods

There are three methods that are used to save a document's code. They complement the three *LoadXXX* methods as follows:

- *SaveToString* – Returns the document contents in a string.
- *SaveToFile* – Saves the document contents to a specified file.
- *SaveToStream* – Saves the document contents to a given stream.

Like their *LoadXXX* counterparts, the *SaveToFile* and *SaveToString* methods simply map down onto *SaveToStream*, after creating suitable output streams. Their operation is quite simple and needs little explanation:

```
procedure TWebBrowserWrapper.SaveToFile(const FileName: string);
var
  FileStream: TFileStream;
begin
  FileStream := TFileStream.Create(FileName, fmCreate);
  try
    SaveToStream(FileStream);
  finally
    FileStream.Free;
  end;
end;

function TWebBrowserWrapper.SaveToString: string;
var
  StringStream: TStringStream;
begin
  StringStream := TStringStream.Create('');
  try
    SaveToStream(StringStream);
    Result := StringStream.DataString;
  finally
    StringStream.Free;
  end;
end;
```

Listing 9

The only thing of note in the above methods is the use of *TStringStream*'s *DataString* property to read out the completed string after writing to the stream.

The *SaveToStream* method follows. It's as simple as it could be, it simply calls *InternalSaveDocumentToStream*.

```
procedure TWebBrowserWrapper.SaveToStream(const Stm: TStream);
begin
  InternalSaveDocumentToStream(Stm);
end;
```

Listing 10

Finally, we get to see how the protected *InternalSaveDocumentToStream* method is implemented. It interacts with the browser control to save the whole document to a stream.

```
procedure TWebBrowserWrapper.InternalSaveDocumentToStream(
  const Stream: TStream);
var
  StreamAdapter: IStream;
  PersistStreamInit: IPersistStreamInit;
begin
  if not Assigned(WebBrowser.Document) then
    Exit;
  if WebBrowser.Document.QueryInterface(
    IPersistStreamInit, PersistStreamInit
  ) = S_OK then
  begin
    StreamAdapter := TStreamAdapter.Create(Stream);
    PersistStreamInit.Save(StreamAdapter, True);
  end;
end;
```

Listing 11

Note that, like in *InternalLoadDocumentFromStream*, we again try to get the web browser document's *IPersistStreamInit* interface and then use its *Save* method to write the document to the stream. We also use *TStreamAdapter* once again to provide the required *IStream* interface for the *TStream*.

Note that the browser control writes the stream in the correct character set for the document.

Stage 2: Adding Unicode Support

Since the browser control supports different character encodings we need to add support for this to our code. For much of this we're going to rely on the encoding support built into Delphi 2009 and later, so get ready for some conditionally defined code.

When we discussed requirements we decided we needed to be able to specify an encoding when writing to files and streams and when reading from a string. To handle this we define new overloads of the *SaveToStream*, *SaveToFile* and *LoadFromString* methods.

Thanks to Mauricio Julio for suggesting some of the ideas and code used in this section. Particular thanks are due for the *GetStreamEncoding* function (see *Listing 14* from his *TcyComponents pack*).

The second requirement was to provide access to the encoding used for the browser control's current document.

Taking these into account the definition of our *TWebBrowserWrapper* class becomes:

```
type
  TWebBrowserWrapper = class(TObject)
  private
    fWebBrowser: TWebBrowser; // wrapped control
  protected
    procedure InternalLoadDocumentFromStream(const Stream: TStream);
    procedure InternalSaveDocumentToStream(const Stream: TStream);
    {$IFDEF UNICODE}
    function GetDocumentEncoding: TEncoding;
    {$ENDIF}
  public
    constructor Create(const WebBrowser: TWebBrowser);
```

```
    procedure LoadFromFile(const FileName: string);
    procedure LoadFromStream(const Stream: TStream);
    procedure LoadFromString(const HTML: string); overload;
    {$IFDEF UNICODE}
    procedure LoadFromString(const HTML: string;
      const Encoding: TEncoding); overload;
    {$ENDIF}
    function NavigateToLocalFile(const FileName: string): Boolean;
    procedure NavigateToResource(const Module: HMODULE;
      const ResName: PChar; const ResType: PChar = nil); overload;
    procedure NavigateToResource(const ModuleName: string;
      const ResName: PChar; const ResType: PChar = nil); overload;
    procedure NavigateToURL(const URL: string);
    function SaveToString: string;
    procedure SaveToStream(const Stm: TStream); overload;
    {$IFDEF UNICODE}
    procedure SaveToStream(const Stm: TStream;
      const Encoding: TEncoding); overload;
    {$ENDIF}
    procedure SaveToFile(const FileName: string); overload;
    {$IFDEF UNICODE}
    procedure SaveToFile(const FileName: string;
      const Encoding: TEncoding); overload;
    {$ENDIF}
    property WebBrowser: TWebBrowser read fWebBrowser;
    {$IFDEF UNICODE}
    property Encoding: TEncoding read GetDocumentEncoding;
    {$ENDIF}
  end;
```

Listing 12

Encoding Property

We provide access to the browser's current document encoding via the read only *Encoding* property which has a read accessor method named *GetDocumentEncoding*, defined in the following listing.

```
{$IFDEF UNICODE}
function TWebBrowserWrapper.GetDocumentEncoding: TEncoding;
var
  Doc: IHTMLDocument2;
  DocStm: TStream;
begin
  Assert(Assigned(WebBrowser.Document));
  Result := TEncoding.Default;
  if WebBrowser.Document.QueryInterface(IHTMLDocument2, Doc) = S_OK then
  begin
    DocStm := TMemoryStream.Create;
    try
      InternalSaveDocumentToStream(DocStm);
      Result := GetStreamEncoding(DocStm);
    finally
      DocStm.Free;
    end;
  end;
end;
{$ENDIF}
```

Listing 13

To get the document encoding we need to examine the structure of the stream that is generated when then the document is saved. We first record the default encoding to return if we can't examine the document for any reason. Once we have a reference to the current document in *Doc* we create a memory stream object and save the browser content into it by calling *InternalSaveDocumentToStream*. The resulting stream is then examined by Mauricio Julio's *GetStreamEncoding* function to get the encoding. *Listing 14* shows the implementation *GetStreamEncoding*.

```
{$IFDEF UNICODE}
function GetStreamEncoding(const Stream: TStream): TEncoding;
var
  Bytes: TBytes;
  Size: Int64;
```

```
begin
  Stream.Seek(0, soFromBeginning);
  Size := Stream.Size;
  SetLength(Bytes, Size);
  Stream.ReadBuffer(Pointer(Bytes)^, Size);
  Result := nil;  // must initialise Result to pass as var param below
  TEncoding.GetBufferEncoding(Bytes, Result);
end;
{$ENDIF}
```

Listing 14

This routine simply copies the provided stream into a *TBytes* array then uses the *GetBufferEncoding* class method of *TEncoding* to determine the encoding.

Revised Document Loading Methods

As noted already we will provide a new overloaded version of *LoadFromString* that takes a *TEncoding* parameter that determines the encoding that will be used to load the string containing the HTML. We will also need to re-implement the original *LoadFromString* method. Here's the new code:

```
{$IFDEF UNICODE}
procedure TWebBrowserWrapper.LoadFromString(const HTML: string;
  const Encoding: TEncoding);
var
  HTMLStm: TMemoryStream;
begin
  Assert(Assigned(Encoding));
  HTMLStm := TMemoryStream.Create;
  try
    StringToStreamBOM(HTML, HTMLStm, Encoding);
    HTMLStm.Position := 0;
    LoadFromStream(HTMLStm);
  finally
    HTMLStm.Free;
  end;
end;
{$ENDIF}

procedure TWebBrowserWrapper.LoadFromString(const HTML: string);
{$IFDEF UNICODE}
begin
  LoadFromString(HTML, TEncoding.Default);
end;
{$ELSE}
var
  StringStream: TStringStream;
begin
  StringStream := TStringStream.Create(HTML);
  try
    LoadFromStream(StringStream);
  finally
    StringStream.Free;
  end;
end;
{$ENDIF}
```

Listing 15

The first thing to note is that, on non-Unicode compilers, the original version of *LoadFromString* is unchanged. However the Unicode version now calls the new overloaded version of the method, passing the default encoding in the *Encoding* parameter.

The new, Unicode only, overloaded method first writes the the string to a temporary memory stream, encoded according to the *Encoding* parameter. The stream is prefixed by any byte order mark required by the encoding. All this work is done by the *StringToStreamBOM* helper routine that is shown in *Listing 16*. Once we have the stream we simply call the existing *LoadFromStream* method to load the stream into the document.

```
{$IFDEF UNICODE}
procedure StringToStreamBOM(const S: string; const Stm: TStream;
```

```
  const Encoding: TEncoding);
var
  Bytes: TBytes;
  Preamble: TBytes;
begin
  Assert(Assigned(Encoding));
  Bytes := Encoding.GetBytes(S);
  Preamble := Encoding.GetPreamble;
  if Length(Preamble) > 0 then
    Stm.WriteBuffer(Preamble[0], Length(Preamble));
  Stm.WriteBuffer(Bytes[0], Length(Bytes));
end;
{$ENDIF}
```

Listing 16

StringToStreamBOM, described above, first converts the string into a byte array according the required encoding. It then writes any required byte order mark to the stream (stored in the *Preamble* variable) followed by the byte array.

Revised Document Saving Methods

In addition to providing new overloaded versions of *SaveToStream* and *SaveToFile* we must re-implement *SaveToString* when compiling with Unicode compilers to take account of the browser document's encoding.

We will first look at the revised *SaveToString* method before discussing the new overloaded methods. Here is the new implementation of *SaveToString*.

```
function TWebBrowserWrapper.SaveToString: string;
{$IFDEF UNICODE}
var
  MS: TMemoryStream;
  Encoding: TEncoding;
  Bytes: TBytes;
begin
  MS := TMemoryStream.Create;
  try
    SaveToStream(MS);
    // This stream may have a pre-amble indicating encoding
    Encoding := GetStreamEncoding(MS);
    MS.Position := Length(Encoding.GetPreamble);
    SetLength(Bytes, MS.Size - MS.Position);
    MS.ReadBuffer(Bytes[0], Length(Bytes));
    Result := Encoding.GetString(Bytes);
  finally
    MS.Free;
  end;
{$ELSE}
var
  StringStream: TStringStream;
begin
  StringStream := TStringStream.Create('');
  try
    SaveToStream(StringStream);
    Result := StringStream.DataString;
  finally
    StringStream.Free;
  end;
{$ENDIF}
end;
```

Listing 17

Just like with *LoadFromString* the non-Unicode version of *SaveToString* remains unchanged.

The Unicode version of the method writes the browser's document into a temporary memory stream. This is done because we need to interpret the output stream according to its encoding. So we call *GetStreamEncoding* to find the encoding used to generate the stream. Next we set the memory stream's position to skip over any byte order mark (preamble). The remainder of the stream is then copied into a byte array that we can pass to the *GetString* method of *TEncoding* which, in turn, returns the required string.

Having disposed of the most complex method we now turn to the new overloaded versions of *SaveToFile* and *SaveToStream*:

```
{$IFDEF UNICODE}
procedure TWebBrowserWrapper.SaveToFile(const FileName: string;
  const Encoding: TEncoding);
var
  FileStream: TFileStream;
begin
  FileStream := TFileStream.Create(FileName, fmCreate);
  try
    SaveToStream(FileStream, Encoding);
  finally
    FileStream.Free;
  end;
end;
{$ENDIF}

{$IFDEF UNICODE}
procedure TWebBrowserWrapper.SaveToStream(const Stm: TStream;
  const Encoding: TEncoding);
var
  HTML: string;
begin
  HTML := SaveToString;
  StringToStreamBOM(HTML, Stm, Encoding);
end;
{$ENDIF}
```

Listing 18

SaveToFile is very similar to the version that does not take an *Encoding*. It creates a stream onto the file then passes the stream and the encoding to the overloaded version of *SaveToStream*.

In contrast the overload of *SaveToStream* is very different. First it calls *SaveToString* to retrieve a string containing the document's content as HTML. It does this because *SaveToString* handles the document encoding correctly. We then save the string to the stream using the encoding provided in the *Encoding* parameter, prefixed by any required byte order mark.

If *Encoding* is the same as the document's encoding then calling this method is wasteful. You should call the *SaveToStream* overload that does not take an *Encoding* parameter instead.

Conclusion

So there we have it, a class that makes *TWebBrowser* a lot more friendly to use when loading and saving documents.

There is obviously a lot more we could do. A couple of things immediately spring to mind.

1. We retro-fit some of the Unicode functionality and support for non-ANSI encodings to the pre-Unicode compiler code. The present code when compiled with anything earlier than Delphi 2009 will not save document content to strings correctly if the document character set is not ANSI.
2. Although this code checks input and output streams for preambles that identify HTML file and document encodings, it does not check any encoding (character sets) included explicitly in the HTML. *Hint:* check the *IHTMLDocument2.charset* property or the `<meta>` tag with the `http-equiv="Content-Type"` attribute.

I hope the code will be useful and has shared some insight into the way this large and often obscure control works.

If you think I've missed anything major, please feel free to add it to the code. If you add anything really useful, please *let me know*.

Demo Program

A demo program that can be used to test and exercise the code presented here is available for download. The demo contains:

- `UWebBrowserWrapper.pas` – contains the *TWebBrowserWrapper* class and helper routines presented in this article along with other required routines.
- A test application that exercises the code. Instructions for use are displayed in the application.
- Some test HTML files using various different encodings.

The code was developed using Delphi 7 Professional (non-Unicode) and Delphi 2010 (Unicode) and tested with both those compilers and Delphi 2006. The code requires the *TWebBrowser* component. The demo may compile with earlier versions of Delphi but this has not been tested.

As noted above, the code presented in this article does not work correctly when loading and saving in Unicode (or UTF-8) when built with a non-Unicode version of Delphi. These limitations are also present in the demo. Choosing Unicode or UTF-8 examples when loading strings into the demo will fail when it is built with compilers earlier than Delphi 2009.

UTF-8 examples can work in non-Unicode builds if the strings contain only ASCII characters.

This source code is merely a proof of concept and is intended only to illustrate this article. It is not designed for use in its current form in finished applications. The code is provided on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied.

The code is open source. `UWebBrowserWrapper.pas` is licensed under the MPL / GPL / LGPL tri-license and other code can be re-used as required. See the source files for full details. If you agree to all this then please download the code using the following link.

Download the demo code

Acknowledgements

Several people's ideas and code samples have been used in developing this code. In particular I'd like to thank Christian Schwarz, Nick Hodges and Babur Saylan for writing some very useful tips on working with *TWebBrowser* at the now defunct "Delphi Pool" website. Various useful articles can also be found on the *Microsoft Developer Network*

Thanks are also due to Mauricio Julio for his ideas and code.
