Embedding the browser in Lazarus

Michaël Van Canneyt

April 9, 2021

Abstract

The Chrome browser is by far the most popular browser of the moment. The technology underlying this browser is freely available: the Chromium project distributes the rendering and Javascript engine underlying this browser. In this article we show how to embed the Chromium browser in your Lazarus Application.

1 Introduction

Survey after survey shows that the Chrome browser is by far the most popular browser of the moment. Google has made the technology underlying this browser freely available: The Chromium Embedded Framework, managed by the Chromium project, makes available the rendering and Javascript engine to any programmer.

This can be used to do something simple such as show webpages in your application: this could be used to embed help or even a complete helpdesk in your application. If you're adventurous, you can try to build a complete browser.

But one can go further: Projects such as Node.js, Electron, Atom, Spotify and many others since a long time use this technology to allow you to run Web Applications on the desktop or on a tablet: The Chromium embedded framework runs on all major platforms, so when using it, your program will run on all major programs, with just a single codebase.

Now why would one want to do this ? Isn't it better to run a native application ? Everything depends on what your application is for.

For applications with an extensive and rich UI, the choice to run the UI in a browser has considerable advantages: There is a multitude of UI frameworks for the browser, and if you want to roll your own, the standards of HTML and CSS allow you to fine-tune your UI and add visually appealing effects with little or no effort.

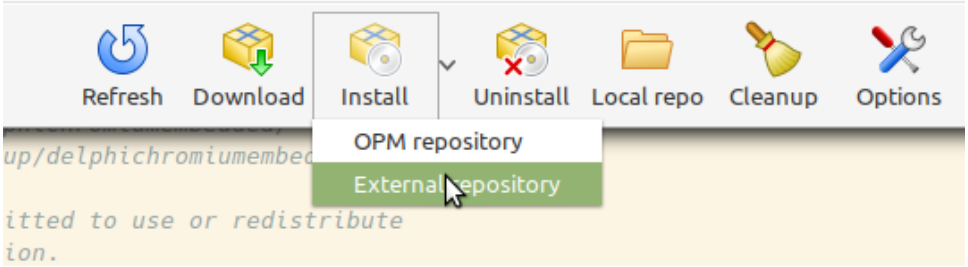
Delphi's VCL or Lazarus' LCL and derived components are hopelessly outdated where it concerns visual effects and gadgets, simply because the Javascript world is meanwhile much bigger and more people contribute to its development.

You may ask: Do these visual effects really matter, and is it worth the effort ? Each must decide this for himself, but the technology exists, and the Free Pascal & Lazarus foundation wants to make it available for Object Pascal programmers.

With the development of pas2js and TMS Web core, it is possible to program the browser itself.

This means that we can now create a program that has a native part, programmed in Pascal, and a GUI part running in the browser - also programmed in pascal. This opens new perspectives.

Figure 1: Install from external repository



2 Getting started

So, you want to embed the browser in your application. How to start ?

The first thing to do is to download the Chromium Embedded Framework. You can find binaries for your platform on

`https://cef-builds.spotifycdn.com/index.html`

Second is to install CEF4delphi in the Lazarus IDE. CEF4Delphi was initially developed as DCEF3 by Henri Gourvest, and is continued by Salvador Diaz Fau.

The name suggests that the project was initially developed for Delphi and as such the support was best on Windows. There was preliminary support for Lazarus, and the Free Pascal and Lazarus foundation has sponsored some development by Martin Friebe to make it universally usable on all major Lazarus-supported platforms: Windows, Linux and MacOS.

The CEF4Delphi project has a package file for installation in Lazarus (CEF4Delphi_Lazarus). The Lazarus OnLine Package manager can be used to install this package.

You will need to install the `DCPCrypt` package as well, as the CEF4Delphi_Lazarus package depends on this package.

Note that when installing through the Online Package Manager, you must install from the external repository, as shown in figure 1 on page 2. The reason is that the changes needed to run in Lazarus are not yet packaged in the zip distributed by the people managing the Online Package manager.

Alternatively, you can also install directly from the CEF4Delphi sources. Simply clone the official repository using your favourite git client:

`https://github.com/salvadordf/CEF4Delphi`

The lazarus package is located in the packages directory.

The package is a Lazarus package as any other and can be installed in the IDE with the usual 'Use - Install' menu from the package dialog. The Lazarus IDE will offer to rebuild itself, and if all goes well, 2 extra tabs will appear on the component palette, shown in figure 2 on page 3 and figure 3 on page 3

The following components are registered on the 'Chromium' tab:

TChromium The general interface to the Chromium Embedded Framework: It basically gives you access to the CEF API, and can be seen as the API for the browser.

Figure 2: Chromium components - Tab 1

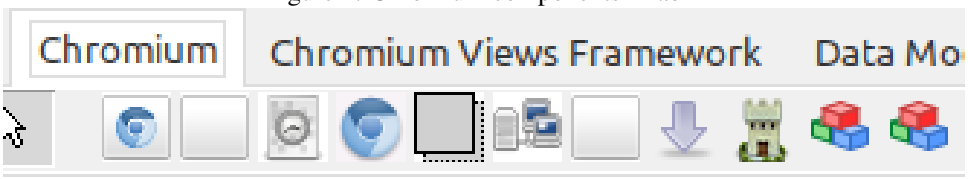
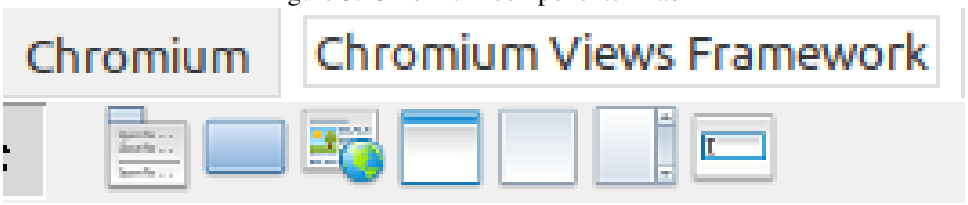


Figure 3: Chromium components - Tab 2



TBufferPanel A panel which can be used by the CEF browser to draw on, it provides double buffering (See the simple OSR (Off Screen Rendering) Browser demo)

TCefServerComponent A small non-visual component that implements a small web-server and websocket server.

TChromiumWindow A `TWinControl` descendent which can be used by the CEF browser to host its windows. Normally you will not use this, you should use `TBrowserWindow`

TCEFWindowParent A `TWinControl` descendent which can be used by the CEF browser to host its windows. Normally you will not use this, you should use `TBrowserWindow`

TCEFLinkedWindowParent A `TWinControl` descendent which can be used by the CEF browser to host its windows. Normally you will not use this, you should use `TBrowserWindow`

TCEFURLRequestClientComponent a `TComponent` wrapper around a `TCustomCefUrlrequestClient`, used to handle request progress.

TCEFWorkScheduler a component that can be used to control when the browser does work and when it is idle.

TBrowserWindow A control that is essentially a combination combination of a `TCEFLinkedWindowParent` control and `TChromium` component. Usually this will be sufficient for most applications.

TOSRBrowserWindow a control similar to `TBrowserWindow` that shows the browser window using OSR (Off Screen Rendering).

TCEFSentinel a control that can be used to check the status of the CEF browser: You must be careful when to close the window, when the window is ready for you to send commands etc.

The components on the 'Chromium Views Framework' tab are normally not necessary for embedding the browser. They are wrappers around CEF classes which - under normal circumstances - you will not need, but they are provided for completeness.

This might look like a lot of components just to implement a browser, but for simple cases, only a single component is needed: `TBrowserWindow`. This visual control has a `TChromium` component built-in and it uses this component to control the browser.

3 Creating a simple browser

To create a simple browser, not much code is needed. To show this, we create a small demo program. We drop an edit control along the top border of the window (edtAddress), and a `TBrowserWindow` control that occupies the rest of the window: `bwBrowser`. Next to the address bar, we put a small button `btnGo`. In the `OnClick` handler of the button, we place the following code:

```
bwBrowser.LoadURL(UTF8Decode(edtAddress.Text));
```

The UTF8Decode is needed because the LCL uses UTF8 for all texts in controls, but the CEF API expects a WideString (or Unicode String) in it's APIs.

If this were all there was to it, it would of course be very simple.

Alas, some more work is needed.

The Chromium browser is a complex piece of software. To let it work correctly, some initialization is needed, and when closing down, also some extra code is needed to make sure the CEF browser closes down properly.

Specifically, you can't just close the form on which the browser window is located. In the `OnCloseQuery` of the form, you must check whether the browser window was actually closed before allowing the form to be closed.

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
  With bwBrowser do
    begin
    CloseBrowser(True);
    CanClose:=IsClosed;
    end;
end;
```

The first line tells the browser to close itself – the parameter `True` means close forcedly. The second line checks if the browser window was actually closed, and if so, allows the form to be closed.

But what if the browser form was not yet closed ? How to close the form ? Well, the `TBrowserWindow` window as an `OnBrowserClosed` event that is triggered when the browser form is actually closed. We can use this event to close the main form again:

```
procedure TForm1.bwBrowserBrowserClosed(Sender: TObject);
begin
  Close;
end;
```

The effect of these 2 event handlers is the following sequence of events:

1. The user closes the main form.
2. The `OnCloseQuery` event is triggered. It tells the browser window to close. If the window is not yet closed, the main form is prohibited from closing.
3. When the browser window is closed, the `OnBrowserClosed` event is triggered. It closes the main form.
4. Again the `OnCloseQuery` event is triggered. But this time, the `IsClosed` will evaluate to `True`, and the main form is allowed to close.

Additionally, on Windows, some extra code is needed to handle the use of the menu. The menu uses a separate modal loop, and this must be communicated to the CEF browser. The following code can be inserted in the form declaration:

```
{$IFDEF WINDOWS}
procedure WMEnterMenuLoop(var aMessage: TMessage); message WM_ENTERMENULOOP;
procedure WMExitMenuLoop(var aMessage: TMessage); message WM_EXITMENULOOP;
{$ENDIF}
```

And the methods must be implemented as follows:

```
uses
   uCEFApplication;

{$IFDEF WINDOWS}
procedure TForm1.WMEnterMenuLoop(var aMessage: TMessage);
begin
  inherited;
  if (aMessage.wParam = 0) and (GlobalCEFApp <> nil) then
    GlobalCEFApp.OsmodalLoop := True;
end;

procedure TForm1.WMExitMenuLoop(var aMessage: TMessage);
begin
  inherited;
  if (aMessage.wParam = 0) and (GlobalCEFApp <> nil) then
    GlobalCEFApp.OsmodalLoop := False;
end;
{$ENDIF}
```

The `uCEFApplication` unit contains the `GlobalCEFApp` object instance which takes care of communication with the CEF library. Setting its `OsmodalLoop` property notifies the CEF library that a menu global event loop is active (or not).

The above code is not so difficult, and is the same for every application that uses CEF. Alas, this is not yet everything you must do.

4 Initializing the CEF library

The CEF library itself must also be initialized and stopped. Because the CEF browser runs in a separate process (this is a security measure), this takes time to set up and a lot of initialization needs to be done. Luckily, the CEF components take care of a lot of the details involved in setting up the library. In a Lazarus program, you don't need to code a lot to start the initialization process.

Unfortunately, this must be done at different points in time, depending on the OS you are using: on Linux, the initialization must happen before the LCL itself is initialized. On Windows and Mac, the LCL must be initialized before starting CEF.

The best way to do this is to put all code in a separate unit. We'll name it `CEFControl`:

```
unit CEFControl;

{$mode objfpc}
```

```
{$h+}

interface

uses
  uCEFApplication, uCEFWorkScheduler;

Procedure InitCEF;
Procedure StopCEF;

implementation

Procedure InitCEF;

var
  Dir : String;

begin
  if Assigned(GlobalCEFApp) then exit;
  {$IFDEF DARWIN}
  AddCrDelegate;
  {$ENDIF}
  CreateGlobalCEFApp;
  // Set the location of the CEF libs.
  {$IFDEF WINDOWS}
  Dir:='c:\CEF\Current\Dist':
  {$ELSE}
  Dir:='/opt/CEF/current/dist';
  {$ENDIF}
  GlobalCEFApp.FrameworkDirPath:=Dir;
  GlobalCEFApp.ResourcesDirPath:=Dir;
  GlobalCEFApp.LocalesDirPath:=Dir+PathDelim+'locales';
  if not GlobalCEFApp.StartMainProcess then
    begin
    StopCEF;
    halt(0); // exit the subprocess
    end;
end;

Procedure StopCEF;
begin
  if GlobalCEFWorkScheduler <> nil then
     GlobalCEFWorkScheduler.StopScheduler;
   DestroyGlobalCEFApp;
   DestroyGlobalCEFWorkScheduler;
end;

{$IFDEF LINUX}
initialization
  InitCEF;
{$ENDIF}
end.
```

On Linux, the `InitCEF` routine is called in the initialization section of the unit. If the

`CEFControl` unit is inserted before the `Interfaces` unit in the program's uses clause, the result is that CEF will be initialized before the LCL is initialized.

To initialize CEF on MacOS and Windows, the `InitCEF` is called in the `Initialization` of the main form unit:

```
initialization
  InitCEF;

finalization
  StopCEF;
end.
```

Since the initialization section of the main form is only executed after the LCL was initialized, the CEF library will be initialized only after the LCL is initialized, and CEF will be stopped before the LCL is finalized.

Let's see what happens in more detail during initialization:

The first line of the `InitCEF` routine checks if the global CEF application exists. If it does, it exits: this ensures that even on Linux, the CEF library is initialized only once.

On MacOS, after this, an application delegate is created, this is a MacOS-specific routine, needed to handle communications with CEF.

The call to `CreateGlobalCEFApp` creates the actual `GlobalCEFApp` instance, and does some configuration.

```
procedure PumpWork(const aDelayMS : int64);
begin
  if (GlobalCEFWorkScheduler <> nil) then
    GlobalCEFWorkScheduler.ScheduleMessagePumpWork(aDelayMS);
end;

procedure CreateGlobalCEFApp;
begin
  if GlobalCEFApp <> nil then
    exit;
  GlobalCEFWorkScheduler := TCEFWorkScheduler.Create(nil);
  GlobalCEFApp:= TCefApplication.Create;
  GlobalCEFApp.ExternalMessagePump:= True;
  GlobalCEFApp.MultiThreadedMessageLoop:= False;
  GlobalCEFApp.OnScheduleMessagePumpWork:= @PumpWork;
  {$IFDEF LINUX}
  GlobalCEFApp.DisableZygote := True;
  {$ENDIF}
end;
```

The configuration is concerned with how messages are handled; in the above case the message pump is handled in `PumpWork`. There are other ways to handle this, but it is outside the scope of this article to treat all possibilities: The above code can simply be used in most case.

After calling `CreateGlobalCEFApp`, a couple of lines set some path properties of the global `GlobalCEFApp` instance. This needs some explanation.

By default, CEF4Delphi looks for the CEF libraries and auxiliary files in the same directory as your application. This means that you must copy the complete contents of the

`Resource` and `Release` (about 1.2 Gb) to your application directory, and this for every time you create an application that uses CEF.

That's of course not very convenient when you're making various applications. A better approach may be to copy the contents of these 2 directories to a single directory (in the above example code this is `CEF/current/dist`) and tell CEF4Delphi to use the libraries and support files in that directory. That is what these lines in `InitCEF` do:

```
{$IFDEF WINDOWS}
Dir:='c:\CEF\Current\Dist':
{$ELSE}
Dir:='/opt/CEF/current/dist';
{$ENDIF}
GlobalCEFApp.FrameworkDirPath:=Dir;
GlobalCEFApp.ResourcesDirPath:=Dir;
GlobalCEFApp.LocalesDirPath:=Dir+PathDelim+'locales';
```

The 3 paths must be set. Note that if you plan to distribute the application, you will have to distribute the contents of that directory as well, and the directory on the end-user's system will probably be something else.

After this configuration, it is time to actually load and initialize the CEF library. This is done with

```
if not GlobalCEFApp.StartMainProcess then
  begin
  StopCEF;
  halt(0); // exit the subprocess
  end;
```

Depending on the configuration this will start a second (sub) process. In the subprocess the return value of this function is `False`, in the main process the function returns `True`. The subprocess should stop at once, as actually a new program is started in the sub process.

That's it. Now the program is ready to go. Running the program will result in something like figure 4 on page 9.

5 Controlling the browser

Now that we've created a simple browser, we can extend the browser a little. The control of the browser is done through a `TChromium` instance. If you used a `TBrowserWindow` control as in the example, then it has an internal instance of TChromium, which you can use to control the browser. You can also assign another `TChromium` instance if you so desire.

Some common operations in a browser is to go forward and backward in the browser history. the `TChromium` class has some methods for this:

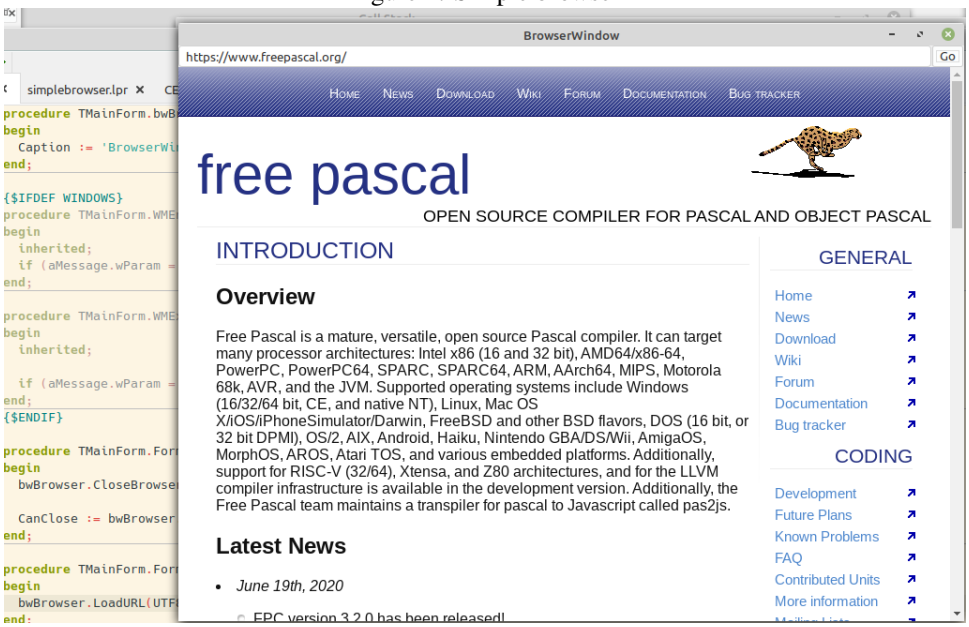
CanGoBack A function that returns `True` if the browser go back in the browsing history.

CanGoForward A function that returns `True` if the browser go forward in browsing history.

GoBack Go 1 item back in the browser history.

GoForward Go 1 item forward in the browser history.

Figure 4: Simple browser



Using these functions, it is easy to add 2 buttons to go back and forth in the browser history. To demonstrate this, we add 2 buttons to the top panel, `btnForward` and `btnBack`. Their `OnClick` handlers are quite simple:

```
procedure TMainForm.btnBackClick(Sender: TObject);
begin
  if bwbrowser.Chromium.CanGoBack then
    bwbrowser.Chromium.GoBack;
end;

procedure TMainForm.btnBackClick(Sender: TObject);
begin
  if bwbrowser.Chromium.CanGoBack then
    bwbrowser.Chromium.GoBack;
end;
```

The `CanGoBack` and `CanGoForward` functions can be used to create actions in an actionlist and let the buttons be enabled or disabled, depending on whether there is history to go forward or backward to. This is left as an exercise for the reader.

As developers, it is useful to be able to show the browser developer tools: this way we can inspect the HTML, CSS or debug the Javascript in the webpage. The chromium component also has a method to help with this: `ShowDevtools` and its counterpart `HideDevTools`. So, we create a menu button that shows a popup menu, and in the popup menu, we add menu items to show and hide the developer console.

The `ShowDevtools` call has 2 arguments: a point and a `TWinControl` instance. The point is a coordinate in the browser window that will be inspected in the developer tools HTML inspector. The `TWinControl` instance is a `TWinControl` instance that will be replaced with the developer tools. The latter can be `Nil`, in which case the developer tools are shown in a floating window.

We'll create a bottom-aligned panel, controller by a splitter, which will contain a second panel: the second panel will be replaced with the developer tools. Initially, the panel is

invisible, and when the user asks to see the developer tools, we'll make the panel visible. When the user asks to close the developer tools, we ask the browser to close them and hide the panel.

```
procedure TMainForm.miHideDevToolsClick(Sender: TObject);
begin
  bwBrowser.Chromium.CloseDevTools(pnlDevToolsInner);
  splDevtools.Visible:=False;
  pnlDevtools.Visible:=False;
end;

procedure TMainForm.miSHowDevToolsClick(Sender: TObject);
begin
  splDevtools.Visible:=True;
  pnlDevtools.Visible:=True;
  pnlDevtools.Height:=ClientHeight div 3;
  bwBrowser.Chromium.ShowDevTools(Point(0,0),pnlDevToolsInner);
  bwBrowser.Chromium.OnDevToolsAgentDetached:=@DoDevtoolsDetached;
end;
```

The visibility of the panel can be used to enable/disable the appropriate menu items in our popup menu:

```
procedure TMainForm.pmMenuPopup(Sender: TObject);
begin
  miSHowDevTools.Enabled:=not pnlDevtools.Visible;
  miHideDevTools.Enabled:=pnlDevtools.Visible;
end;
```

6 Events from the browser

The `TChromium` component has a lot of event handlers. These are used by the CEF framework to notify your program of many things.

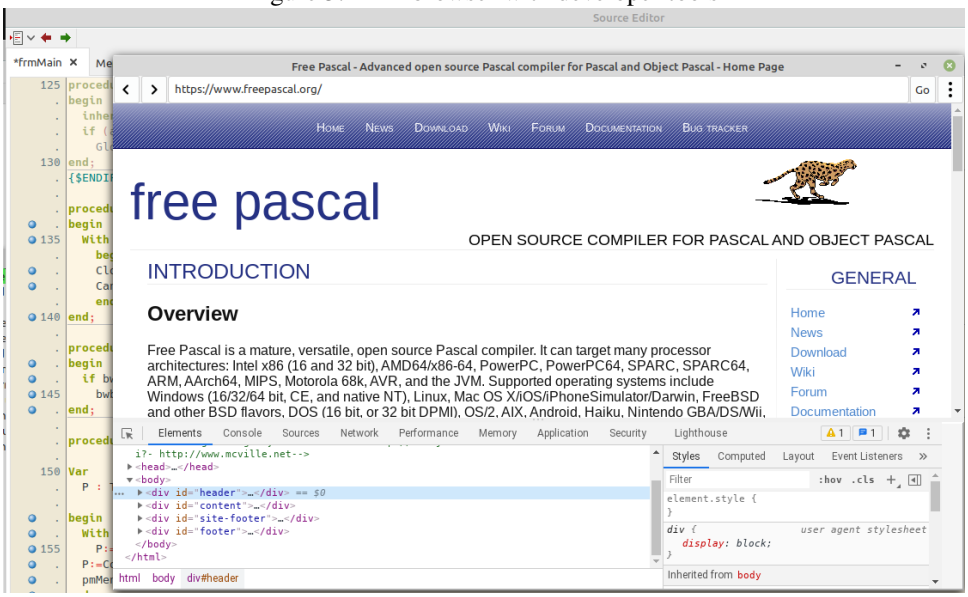
However, care must be taken when using these event handlers: The browser is running in a separate process, so all communication happens through the sending of messages, which are translated to events by CEF4Delphi. These events arrive in a thread which does not necessarily have to be the main UI thread. Since the widgetsets used by the LCL are not thread-safe this means that you cannot just update the GUI in the event handler. Luckily, Lazarus has a mechanism that can be used for this: `Appliction.QueueAsyncCall`.

`QueueAsyncCall` schedules a callback, which will be called when the main application loop as finished processing pending messages. This callback is therefor called in the main application thread, and so it can be used to update the UI.

To demonstrate this, we'll use a simple event. The `OnTitleChange` event of the `TChromium` component notifies you when a HTML page has a `Title` tag in it, and it sends you the title. We can use this event to update the title bar of the form:

```
procedure TMainForm.ChromiumTitleChange(Sender: TObject;
  const browser: ICefBrowser; const title: ustring);
begin
  if (length(title) > 0) then
```

Figure 5: Mini browser with developer tools




```
    FNewTitle := UTF8enCode(title)
   else
    FNewTitle:=UTF8enCode(bwBrowser.Chromium.DocumentURL);
  Application.QueueAsyncCall(@DoSetHTMLTitle,0);
end;
```

The new title is saved to a temporary variable, and an async call is scheduled: `DoSetHTMLTitle`:

```
procedure TMainForm.DoSetHTMLTitle(Data: PtrInt);
begin
  Caption:=FNewTitle;
end;
```

Since this call is executed in the main thread, it is safe to update the form caption.

The result of this can be seen in figure 5 on page 11.

7 Using your own scheme to serve local files

The browser normally uses the 'http(s)://' URL scheme to access files on a webserver. If you double-click on a HTML file in the file explorer, the 'file://' URL scheme.

If you want to create a GUI using a browser, and you want to distribute an application that can be only be used to load the files you want, you can register a custom URL scheme.

You can use this to serve a restricted list of files, only files from a ZIP file, or even not to distribute files, but have all files in resources embedded in the executable.

We'll demonstrate this by creating an application that plays the Pacman demo from the pas2js demos.

To do this is not very difficult. We'll use the first demo and extend it a little. When the `GlobalCEFApp` application is created in the `CEFControl` unit, we add a callback in which custom URL schemes can be registered:

```
GlobalCEFApp.OnRegCustomSchemes:=@CEFRegisterCustomSchemes;
```

This routine will be called when the browser is started, and is quite simple. When called, it gets a reference to a scheme registrar object , which can be used to register a scheme:

```
procedure CEFRegisterCustomSchemes(const registrar: TCefSchemeRegistrarRef);

Var
  aOptions : TCefSchemeOptions;

begin
  aOptions:=CEF_SCHEME_OPTION_STANDARD or CEF_SCHEME_OPTION_LOCAL;
  registrar.AddCustomScheme('local', aOptions);
end;
```

We register here the URL Scheme name 'local'. The options mean that the scheme is a local one, which ensures that the browser will use the same safety mechanisms as when using the `file://` scheme.

The `TCefSchemeRegistrarRef` class is defined in unit `uCEFSchemeRegistrar`, so we must add that to the uses class of our `CEFControl` unit. Likewise we need the `uCEFTypes` unit for the `TCefSchemeOptions` and the `uCEFConstants` unit for the various constants.

The above just tells the browser it can expect to see URLS starting with the `local://` scheme. It does not yet handle this scheme.

To actually handle the 'local' scheme, we need to implement a descendent of the `TCefResourceHandlerOwn` class and register it. At least 3 methods of this class must be overridden:

```
TLocalResourceHandler = Class(TCefResourceHandlerOwn)
private
  FFileName : String;
  FStream : TStream;
Public
  function  ProcessRequest(CEFRequest: ICefRequest;
                           callback: ICefCallback): Boolean; override;
  procedure GetResponseHeaders(CEFResponse: ICefResponse;
                               out responseLength: Int64;
                               out redirectUrl: ustring); override;
  function  ReadResponse(const dataOut: Pointer;
                         bytesToRead: Integer;
                         var bytesRead: Integer;
                         callback: ICefCallback): Boolean; override;
  Destructor destroy; override;
end;
```

To be able to add this definition, you need to add the `uCEFInterfaces`, and `uCEFResourceHandler` units to the uses clause.

This class must be registered with the browser, the `CefRegisterSchemeHandlerFactory` function from the `uCEFMiscFunctions` unit can be used for this:

```
CefRegisterSchemeHandlerFactory('local','',TLocalResourceHandler);
```

Now, when the browser needs to handle a URL scheme of 'local', it will create an instance of `TLocalResourceHandler`, and call `ProcessRequest`, `GetResponseHeaders` and `ReadResponse`, in that order.

The first method to be called is `ProcessRequest`:

```
function TLocalResourceHandler.ProcessRequest(const CEFRequest: ICefRequest;
  const callback: ICefCallback): Boolean;

Var
  S : String;
  P : Integer;

begin
  S:=CEFRequest.GetUrl;
  P:=Pos('://',S);
  if P>0 then
    Delete(S,1,P+2);
  FFileName:=SysUtils.SetDirSeparators(ExtractFilePath(ParamStr(0))+S);
  Result:=FileExists(FFileName);
  if Result then
    FStream:=TFileStream.Create(FFileName,fmOpenRead or fmShareDenyWrite);
  if assigned(CallBack) then
    Callback.Cont;
end;
```

For this example we just check if the file exists, and if it does, we return `True`, otherwise we return false. For a real application, you would probably check that the URL does not contain any strange or forbidden locations, or is restricted to a list of files. When the file exists and can be served, we create a file stream (it is destroyed in the destructor).

The browser passes a callback, which you must call in order to indicate that the process can continue.

After this call, the `GetResponseHeaders` is called. This method should simulate the headers of a HTTP Response, and return the size of the data:

```
procedure TLocalResourceHandler.GetResponseHeaders(
  const CEFResponse: ICefResponse; out responseLength: Int64; out
  redirectUrl: ustring);

Var
  Ext : String;

begin
  Ext:=ExtractFileExt(FFileName);
  Ext:=Copy(Ext,2,Length(Ext)-1);
  if Assigned(FStream) then
    begin
    CefResponse.Status:=200;
    CefResponse.StatusText:='OK';
    responseLength:=FStream.Size;
    end
  else
    begin
    CefResponse.Status:=400;
    CefResponse.StatusText:='NOT FOUND';
    responseLength:=0;
    end;
```

```
  CefResponse.MimeType:=CefGetMimeType(Ext);
  // No other headers.
end;
```

Normally, you would also set other HTTP headers of the request, but for simplicity we've omitted that part in our implementation.

After these 2 calls, if there is data to be read, the browser will call `ReadResponse` to actually read the data. This is quite simple, we just read the data from the stream we created in `ProcessRequest`, taking care we do not read too much data:

```
function TLocalResourceHandler.ReadResponse(const dataOut: Pointer;
  bytesToRead: Integer; var bytesRead: Integer; const callback: ICefCallback
  ): Boolean;


Var
  aAvail : Integer;

begin
  Result:=Assigned(FStream) and Assigned(DataOut);
  if Result then
    begin
    aAvail:=FStream.Size-FStream.Position;
    If aAvail>BytesToRead then
      aAvail:=BytesToRead;
    BytesRead:=FStream.Read(DataOut^,aAvail);
    Result:=aAvail>0;
    end;
end;
```

Note that the function must return `True` if data was read, and `False` if there is no more data, or the browser will keep trying to read data. (If the size of the data is not known in advance, this can happen).

That's it. Our application is now ready to use the 'local://' scheme.

We remove the address bar, the go button, and simply load the HTML file we wish to show in the browser with the `local` scheme:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  CefRegisterSchemeHandlerFactory('local','',TLocalResourceHandler);
  bwBrowser.LoadURL('local://files/index.html');
end;
```

When using the pacman example from pas2js, this will look as in figure 6 on page 15

8 Conclusion

In this article, we showed how to embed a browser in your Lazarus application: due to recent improvements in the CEF4Delphi framework, this has been made really easy. But the article has scratched only the surface of what is possible. In a next article, we'll show how to interact with the host application from within the browser application.

Figure 6: PacMan example