Implementing APIs for Chromium

Michaël Van Canneyt

May 18, 2021

Abstract

In the previous article about embedding Chromium, we showed how to embed Chromium in your application. In this article, we go a step further, and we show how to add APIs to the Chromium environment, and how these APIs can be used in your Javascript or Pas2JS code.

1 Introduction

In the previous article, we showed how you can embed Chromium in your Lazarus (or Delphi, for the procedure is the same) application. We also showed how you can load files using a private protocol. This can be used to show any website, or to limit the shown HTML files shown to the files you choose.

However, the same can more or less be done by starting the browser e.g. in kiosk mode as a separate process: The possibilities are limited to what the browser offers you.

It becomes really interesting when you start adding possibilities to the embedded browser, that allow it to interact with the user environment. It is possible to add file management, tray icon management, menu management, or virtually anything you want to the Browser environment: You can create Javascript objects with methods, and make these available to the HTML and Javascript running in the browser windows.

In this article, we'll show how to do this: We'll demonstrate how to add separate logging to the browser, or how to let the browser interact with a tray icon and popup menu that lives in your application. Obviously, much more could be done, but the examples serve just to show how to do these things.

2 Architecture

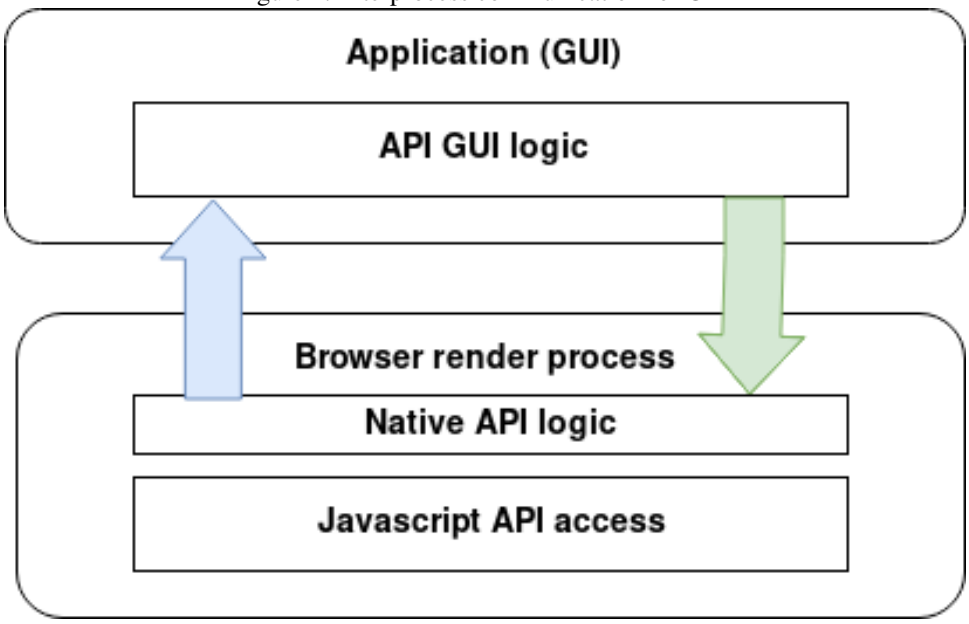
Adding Javascript classes to the Chromium browser is not so difficult. CEF has a rich API for this, and the complete API is available to you through CEF4Delphi.

Basically, you build a Javascript object using the appropriate APIs, attach some methods or properties to it with some callback methods, and that is it.

However, it becomes more complicated when you need to handle GUI in your Javascript objects. For instance, you could decide to show a file-open dialog, or allow access to a menu, or anything else that requires access to the GUI.

The reason it becomes more complicated is that the browser is running in a separate process: when you start the CEF browser, it creates a new process (called the RENDER process), and everything connected with the browser is running in that process. This includes the Javascript and any classes you make available to the browser.

Figure 1: Interprocess communication for CEF



The code for your Javascript class will be running in the RENDER process, but your GUI components will live in your original program: called the BROWSER process (somewhat of a misnomer).

So, if your Javascript classes wish to change the GUI (show or hide a tray icon), or if an event in the GUI (a menu click) must be communicated to the Javascript, this will involve communication between the RENDER process and the BROWSER process. This communication happens with messages, and is asynchronous.

This is schematically depicted in figure figure 1 on page 2.

Imagine you wish to make a special log call available to the Javascript code, which sends the log message to a memo on the main window. This involves creating a class which will live in the browser render process. When the Javascript log method is executed, the method in the class will receive the logged items. To actually display the message, it needs to send this to the GUI process using a message (depicted by the blue, upward arrow).

The reverse is also true: If there is an API that allows the browser Javascript code to respond to a menu click (for example a popup menu item of a tray icon), and the user clicks a message, there must be some way that the GUI process communicates this to the browser RENDER process (depicted by the green, downward arrow), and then the classes that implement the Javascript API will communicate this to the browser.

All this requires quite some code, but luckily the CEF API has a mechanism for sending messages between the two processes. In the below, we show how to do this.

We'll build on one of the previous examples: we create a small application with 2 APIs: one simply replaces console.log with a custom method: it will display the logged data in a memo. Since we implement a logging method, we'll also log unhandled Javascript exceptions, as a useful debugging tool.

The second API is an object that represents a Tray icon, managed by the program. The API is an object, with a property that controls the visibility of the tray icon – not surprisingly, the property is called 'visible'. It also has 2 methods: one to add a menu item to the popup menu of the tray icon, the second to remove the menu item.

In JavaScript, these 2 APIs would look like this:

```
window.trayIcon = {
  visible : boolean,
  addMenuItem : function(aCaption, ACallBack) {

  },
  removeMenuItem : function (aID) {

  }
};
window.appLog = function()  {

};
```

So, how to implement these in CEF? The CEF offers a rich API for representing Javascript values. The basis of this is the `ICefv8Value` interface, defined in the uCEFInterfaces unit, like all other interfaces offered by CEF. It allows to check the value type, get or set the value, or, if the interface represents an object, query the members of the object.

3 Creating a Javascript function

Functions in Javascript are values like any other, so if we want to create a logging function, it stands to reason that this will be a `Function` value.

In CEF4Delphi, the `TCefv8ValueRef` class implements the `ICefv8Value` interface. So our logging function `appLog` starts with a `TCefv8ValueRef` instance:

We create it like this, in the `CreateLoggerObject` function:

```
Function CreateLoggerObject : ICefv8Value;

begin
  Result:=TCefv8ValueRef.NewFunction(SFuncDoLog,
                                     TLogHandler.Create);
end;
```

The `NewFunction` creates a function object. The first parameter is the name, the second parameter is the function handler interface: This must be a `ICefv8Handler` interface. This interface will be called when the function is invoked in Javascript.

In our logging function, the `TLogHandler` class is created. This is a descendent of the `TCefv8HandlerOwn` class in CEF4Delphi, which implements `ICefv8Handler` for us.

This class has only 1 method that we must override, aptly named `Execute`:

```
TLogHandler = class(TCefv8HandlerOwn)
function Execute(const name: ustring;
                 const obj: ICefv8Value;
                 const arguments: TCefv8ValueArray;
                 var retval: ICefv8Value;
                 var exception: ustring): Boolean; override;
end;
```

The meaning of the parameters should be clear from their names.

In the `Execute` function, the logic of the function must be implemented.

The Javascript `console.log` function accepts any number of arguments, and writes a string representation of these arguments to the console. Here we mimic this behaviour. We start by creating a string representation of the arguments, separated by spaces.

As explained above, the code of `TLogHandler` is executed in the RENDER process. To actually display the string, we must send it to the BROWSER process.

This can be done with the `ICefProcessMessage` interface, which is implemented by the `TCefProcessMessageRef` class. So we must create an instance of this class, and attach the string to it: an arbitrary amount of data can be attached to a message, the `ArgumentList` property of the message must be used for this.

Every message is named, and the BROWSER process can handle the messages by checking the name. In our program, the names of the messages are defined as pascal constants. For the log message, the name is in the `SMsgDisplayLogMessage` constant:

```
function TLogHandler.Execute(const name: ustring;
  const obj: ICefv8Value;
  const arguments: TCefv8ValueArray;
  var retval: ICefv8Value;
  var exception: ustring): Boolean;

Var
  msg: ICefProcessMessage;
  i,j : Integer;
  S : uString;

begin
  Result:=True;
  msg := TCefProcessMessageRef.New(SMsgDisplayLogMessage);
  msg.ArgumentList.SetSize(1);
  S:='';
  For I:=0 to Length(Arguments)-1 do
    begin
    if I>0 then
      S:=S+' ';
    S:=S+CefValToString(Arguments[i]);
    end;
  msg.ArgumentList.SetString(0,S);
  SendProcessMessage(PID_BROWSER, msg);
end;
```

The last 2 statements attach the constructed string to the message, and send it to the BROWSER process using the `SendProcessMessage` method of the `iCEFFrame` interface: this interface represents a HTML frame of a webpage.

The `SendProcessMessage` is a small helper method that encapsulates

```
TCefv8ContextRef.Current.Browser.MainFrame.SendProcessMessage
```

This long expression is a rather complex way of getting a reference to the main frame of the HTML page from which the Javascript function was called.

How the BROWSER process must respond to this message, we will see later in this article.

The `CefValToString` function transforms a `iCEFv8Value` to a string, it uses the methods of `iCEFv8Value` to inspect the value type and convert it in an appropriate way

to a string:

```
Function CefValToString(A : ICefv8Value) : String;

var
  i,j : Integer;
  L : TStrings;

begin
  if A.IsString then
    Result:=A.GetStringValue
  else if A.IsBool then
    Result:=BoolToStr(A.GetBoolValue)
  else if A.IsInt then
    Result:=IntToStr(A.GetIntValue)
  else if A.IsDouble then
    Result:=FloatToStr(A.GetDoubleValue)
  else if A.IsNull then
    Result:='null'
  else if A.IsArray then
    begin
    Result:='[';
    for J:=0 to A.GetArrayLength-1 do
      begin
      if J>0 then
        Result:=Result+', ';
      Result:=Result+CefValToString(A.GetValueByIndex(J));
      end;
    Result:=Result+']';
    end
  else if A.IsObject then
    begin
    Result:='{';
    L:=TstringList.Create;
    try
    A.GetKeys(L);
    For J:=0 to L.Count-1 do
      begin
      if J>0 then
        Result:=Result+', ';
      Result:=Result+'"'+L[i]+'": '+CefValToString(A.GetValueByKey(L[i]));
      end;
    finally
      l.Free;
    end;
    Result:=Result+']';
    end
  else
    Result:=Result+'[Cannot display this value]';
end;
```

Note that this function recursively calls itself in the case of Javascript arrays and objects.

4 Creating a Javascript object

Creating a Javascript object is not much different from creating a function: we define it as a value of type object (using the `NewObject` class method), and attach the definition of the `visible` property and the methods to add or remove menu items to it.

To control access to the properties of an object, the `ICefV8Accessor` interface must be used. This interface must be passed to the `NewObject` method. We'll create a class `TTrayIconAccessor` to handle the access for our `trayIcon` object: The CEF4Delphi framework defines a `TCefV8AccessorOwn` class which has the necessary methods for the `ICefV8Accessor` interface, and if we make a descendent of this class, we must simply override these methods.

Adding a property to a Javascript object can be done using the `SetValueByKey` method of `iCEFV8value`. This method takes 3 parameters: the name, the initial value (a boolean in our case) and a set of attributes (some OR-ed integer values).

Object methods are functions, and functions are Javascript values like any other. So, in order to define a method for an object, we must simply add a property with the name of the method and supply a function value as initial value. As seen above, functions can be implemented with an instance of a handler class.

Because the `Execute` method of the handler class receives the name of the function that is called, we can use a single handler class to handle both the `addMenuItem` and `removeMenuItem` methods: The `TTrayIconHandler` class.

Armed with the 2 classes `TTrayIconHandler` and `TTrayIconAccessor`, we can now construct our `trayIcon` object in the `CreateTrayIconObject` class.

The main method to define a property is `SetValueByKey`:

```
Function CreateTrayIconObject : ICefv8Value;

Var
  TempAccessor : ICefV8Accessor;
  TempHandler : ICefv8Handler;
  Ref : ICefv8Value;
begin
  TempAccessor := TTrayIconAccessor.Create;
  TempHandler:= TTrayIconHandler.Create;
  // Create object
  Result:=TCefv8ValueRef.NewObject(TempAccessor,nil);
  // Add visible property
  if not Result.SetValueByKey(SPropVisible,
                              TCefv8ValueRef.NewBool(False),
                              V8_PROPERTY_ATTRIBUTE_NONE) then
    Raise Exception.Create('Could not set visible attribute');
  Result.SetValueByAccessor(SPropVisible,
                            V8_ACCESS_CONTROL_DEFAULT,
                            V8_PROPERTY_ATTRIBUTE_NONE);
  // Add addMenuItem method
  Ref:=TCefv8ValueRef.NewFunction(SFuncAddMenuItem,TempHandler);
  if not Result.SetValueByKey(SFuncAddMenuItem,
                              Ref,
                              V8_PROPERTY_ATTRIBUTE_NONE) then
    Raise Exception.Create('Could not add addMenuItem function');
  // Add removeMenuItem method
  Ref:=TCefv8ValueRef.NewFunction(SFuncRemoveMenuItem,TempHandler);
```

```
  if not Result.SetValueByKey(SFuncRemoveMenuItem,
                              Ref,
                              V8_PROPERTY_ATTRIBUTE_NONE) then
    Raise Exception.Create('Could not add addMenuItem function');
end;
```

Object properties have a lot of attributes in CEF, but we don't need those, so we pass `V8_PROPERTY_ATTRIBUTE_NONE` for all of them.

Access to the `visiblity` property is handled by the following class:

```
TTrayIconAccessor = class(TCefV8AccessorOwn)
private
  FVisible : Boolean;
  procedure SendBrowserShowHideTrayIcon(aValue: Boolean);
protected
  function Get(const name: ustring; const obj: ICefv8Value;
               var retval : ICefv8Value;
               var exception: ustring): Boolean; override;
  function Set_(const name: ustring; const obj,
               value: ICefv8Value;
               var exception: ustring): Boolean; override;
end;
```

It is quite a simple class: it has methods `Get` and `Set_`, which are called whenever a property is read or written, much like the getter and setter of properties in pascal.

The name of the property is passed to these methods. In our case, only the 'visible' property is handled.

The following is then pretty straightforward:

```
function TTrayIconAccessor.Get(const name  : ustring;
                           const obj : ICefv8Value;
                           var retval : ICefv8Value;
                           var exception : ustring): Boolean;
begin
  Result:=(name=SPropVisible);
  if Result then
    retval:=TCefv8ValueRef.NewBool(FVisible);
end;
```

The setter is slightly more complicated: we must not only check the name, but also the value type:

```
function TTrayIconAccessor.Set_(const name      : ustring;
                           const obj       : ICefv8Value;
                           const value     : ICefv8Value;
                           var   exception : ustring): Boolean;
begin
  Result:=(name=SPropVisible);
  if Result then
    begin
    if value.IsBool then
      begin
      FVisible:=value.GetBoolValue;
```

```
      SendBrowserShowHideTrayIcon(FVisible);
      end
    else
      exception := 'Invalid value type, expect boolean';
    end
end;
```

Note that we set the `exception` parameter if something is wrong with the passed value: This will cause a Javascript exception to be raised.

To actually show (or hide) the tray icon, the BROWSER process must be notified. For this, we send again a message, and this is done in the `SendBrowserShowHideTrayIcon` method. The process is similar to the way the browser is notified that a log message is sent.

```
procedure TTrayIconAccessor.SendBrowserShowHideTrayIcon(aValue : Boolean);

var
  msg: ICefProcessMessage;

begin
  msg := TCefProcessMessageRef.New(SMsgSetTrayIconVisibility);
  msg.ArgumentList.SetBool(0,aValue);
  SendProcessMessage(PID_BROWSER, msg);
end;
```

The handler class for the tray icon much resembles the function handler for the log function:

```
TTrayIconHandler = class(TCefv8HandlerOwn)
Private
  Class Var FMenu : TTrayMenu;
  function AddMenuItem(aCaption: uString; AHandler: ICefv8Value;
                       AContext : ICefV8Context): Integer;
  function RemoveMenuItem(aID: Integer;
                          AContext: ICefV8Context): Integer;
  class procedure CheckMenu; static;
protected
  class procedure CallTrayMenuCallBack(aID : integer); static;
  function Execute(const name: ustring; const obj: ICefv8Value;
                   const arguments: TCefv8ValueArray;
                   var retval: ICefv8Value;
                   var exception: ustring): Boolean; override;
end;
```

Again, the `Execute` method is the entry point when the Javascript environment needs to call the methods: It checks the name of the called function:

```
function TTrayIconHandler.Execute(const name: ustring;
  const obj: ICefv8Value;
  const arguments: TCefv8ValueArray;
  var retval: ICefv8Value;
  var exception: ustring): Boolean;

Var
  aLen,aID : Integer;
```

```
begin
  Result:=True;
  aLen:=length(arguments);
  Case name of
  SFuncAddMenuItem:
    begin
    if (aLen>1)
        and arguments[0].IsString
        and arguments[1].IsFunction then
      begin
      aID:=AddMenuItem(arguments[0].GetStringValue,
                       arguments[1],
                       TCefv8ContextRef.Current);
      retVal:=TCefv8ValueRef.NewInt(aID);
      end
    else
      exception:='Need 2 arguments: caption and callback';
    end;
  SFuncRemoveMenuItem:
    begin
    if (aLen>0) and arguments[0].IsInt then
      begin
      aID:=RemoveMenuItem(arguments[0].GetIntValue,
                          TCefv8ContextRef.Current);
      retVal:=TCefv8ValueRef.NewInt(aID);
      end
    else
      exception:='Need 1 arguments: menu item ID';
    end;
  else
    Result:=False;
  end;
```

The `AddMenuItem` and `RemoveMenuItem` functions do the actual work: they update the local copy of the tray icon menu, and then send a message to the BROWSER process.

The local copy of the menu is a simple global collection with ID, caption, handler and context. These fields are needed to keep the various event handlers that have been registered in the Javascript: this is the `aHandler` argument.

The `aContext` parameter (and property) is necessary to be able to call the correct handler when the BROWSER process sends a message notifying that the menu item is clicked:

```
function TTrayIconHandler.AddMenuItem(aCaption : uString;
    AHandler : ICefv8Value; AContext : ICefV8Context) : Integer;

Var
  M : TTrayMenuItem;
  msg: ICefProcessMessage;

begin
  CheckMenu;
  M:=FMenu.AddMenu(aCaption);
  M.Handler:=aHandler;
  M.Context:=aContext;
```

```
  Result:=M.ID;
  // Send message to browser process.
  msg := TCefProcessMessageRef.New(SMsgAddTrayMenuItem);
  msg.ArgumentList.SetSize(2);
  msg.ArgumentList.SetInt(0,M.ID);
  msg.ArgumentList.SetString(1,M.Caption);
  SendProcessMessage(PID_BROWSER, msg);
end;
```

As you can see, the last statement again sends the message to the BROWSER process.

The `RemoveMenuItem` is similar, but it needs less arguments:

```
function TTrayIconHandler.RemoveMenuItem(aID : Integer;
                   AContext : ICefV8Context) : Integer;

Var
  msg: ICefProcessMessage;

begin
  CheckMenu;
  FMenu.RemoveMenu(aID);
  Result:=0;
  // Send message to browser process.
  msg := TCefProcessMessageRef.New(SMsgRemoveTrayMenuItem);
  msg.ArgumentList.SetSize(1);
  msg.ArgumentList.SetInt(0,aID);
  SendProcessMessage(PID_BROWSER, msg);
end;
```

With this, we have implemented a javascript object that can be manipulated from within a HTML Page displayed in the embedded browser.

5 Adding the identifiers to the browser

The previous paragraphs showed how to construct Javascript functions and objects using the classes that CEF4Delphi makes available, but it did not yet make the browser aware of these classes. This must be handled explicitly. Also, we still need a mechanism to react on the actual `OnClick` of the tray icon's popup menu: from the architecture sketched above, it should be clear that the BROWSER process will have to notify the RENDER process with a message when a click happens.

We also want to do 2 extra things:

- Replace console.log with our own handler, so all log messages are sent to the Console.
- Log a message whenever a Javascript exception occurs. This is useful for debugging.

The place to set up the necessary callbacks for all this is in the `CreateGlobalCEFApp` routine we developed in the previous article. In this routine, we set some additional event handlers:

```
procedure CreateGlobalCEFApp;
```

```
begin
  if GlobalCEFApp <> nil then
    exit;
  GlobalCEFWorkScheduler := TCEFWorkScheduler.Create(nil);
  GlobalCEFApp:= TCefApplication.Create;
  GlobalCEFApp.ExternalMessagePump:= True;
  GlobalCEFApp.MultiThreadedMessageLoop:= False;
  GlobalCEFApp.OnScheduleMessagePumpWork:=
     @GlobalCEFApp_OnScheduleMessagePumpWork;
  GlobalCEFApp.OnRegCustomSchemes:=@CEFRegisterCustomSchemes;
  // New handlers
  GlobalCEFApp.OnContextCreated:=@HandleNewContext;
  GlobalCEFApp.OnWebKitInitialized:=@HandleWebkitInitialized;
  GlobalCEFApp.OnUncaughtException:=@HandleJSException;
  GlobalCEFApp.OnProcessMessageReceived := @HandleProcessMessage;
  // End of new handlers
  {$IFDEF LINUX}
  GlobalCEFApp.DisableZygote := True;
  {$ENDIF}
end;
```

From the comments, we can see that 4 extra events have been assigned. We'll discuss them one by one.

The `OnContextCreated` event is fired when a new browser context is created. This is where you can introduce new Javascript identifiers for that particular browser.

We use the 2 functions we created in the above to create an instance of the objects we wish to expose. Any global Javascript object or identifier is actually attached to the Window object. So, we must attach our new identifiers to the global Window object.

Since this is a Javascript object like any other, the code to do so is no different from the code to add methods and properties to an object, except that in this case the object is the Window object, which is made available to us through the `context.Global` value:

```
procedure HandleNewContext(const browser: ICefBrowser;
   const frame: ICefFrame;
   const context: ICefv8Context);

Var
  logger,
  trayIcon : ICefv8Value;

begin
  trayIcon:=CreateTrayIconObject;
  logger:=CreateLoggerObject;
  With Context.Global do
    begin
    if not SetValueByKey(STrayIcon,
                         trayIcon,
                         V8_PROPERTY_ATTRIBUTE_NONE) then
      Raise Exception.Create('Could not set global tray icon object');
    if not SetValueByKey(SFuncDoLog,
                         logger,
                         V8_PROPERTY_ATTRIBUTE_NONE) then
    Raise Exception.Create('Could not set global log function');
```

```
    end;
end;
```

That's all there is to it.

CEF offers the possibility to execute some custom Javascript when the browser window is created and ready, but before all other javascript is executed. This is called an 'extension'. Extensions can be registered using the `CefRegisterExtension` method of CEF.

We will use this to reroute the console.log function to our own `appLog` function. The place to register an extension is in the `OnWebKitInitialized` event:

```
procedure HandleWebkitInitialized;

var
  HookConsole : string;

begin
  HookConsole:='(function() {' +
               '  var oldlog;'+
               '  if (!console) console = {}; '+
               '  if (console.log) oldlog=console.log; '+
               '  console.log = function() {' +
               '    if (oldlog) oldlog.apply(window,arguments); '+
               '    window.'+SFuncDoLog+'.apply(window,arguments); '+
               '  };' +
               '})();';
  CefRegisterExtension('v8/hookconsole', HookConsole,Nil);
end;
```

People familiar with Javascript will see that this little piece of Javascript defines `console.log` if it does not yet exist, or reroutes it to our `appLog` function.

6 Getting notified of exceptions

We want to be notified whenever the Javascript code throws an exception that is not explicitly handled. This can be done in the `OnUncaughtException` event, which we set to the `HandleJSException` method.

This method is quite simple. It converts the exception message to a string and sends a log message to the BROWSER process:

```
procedure HandleJSException(const browser: ICefBrowser;
                            const frame: ICefFrame;
                            const context: ICefv8Context;
                            const exception: ICefV8Exception;
                            const stackTrace: ICefV8StackTrace);

Var
  msg: ICefProcessMessage;

begin
  msg := TCefProcessMessageRef.New(SMsgDisplayLogMessage);
  msg.ArgumentList.SetString(0, 'Exception : '+Exception.Message);
  Browser.MainFrame.SendProcessMessage(PID_BROWSER, msg);
```

```
end;
```

We could also add a stacktrace, but for simplicity we limited ourselves to simply logging the exception message. Note that this method received the actual browser instance: we use that to send the message to the BROWSER process.

7 Communication from BROWSER to RENDER process

We've shown how to create javascript objects and functions, and in the code handling these functions we've seen how messages are sent from the RENDER process to the BROWSER process.

When the user clicks a menu item in the tray icon's popup menu, the BROWSER process needs to communicate this back to the RENDER process, and specifically to the Javascript handler that was passed as part of the creation of the menu item.

This is where the Chromium app's event handler `OnProcessMessageReceived` comes in: this event is triggered when the RENDER process receives a message from the BROWSER process. We set the event to the `HandleProcessMessage` procedure:

```
procedure HandleProcessMessage(const browser: ICefBrowser;
                               const frame: ICefFrame;
                               sourceProcess: TCefProcessId;
                               const aMessage: ICefProcessMessage;
                               var   aHandled: boolean);
var
  aID : Integer;

begin
  if (aMessage.name=SMsgExecuteTrayMenuClick)
      and (aMessage.ArgumentList.GetSize>0) then
    begin
    aID:=aMessage.ArgumentList.GetInt(0);
    TTrayIconHandler.CallTrayMenuCallBack(aID);
    aHandled:=true;
    end;
end;
```

The `CallTrayMenuCallBack` method is responsible for calling the Javascript callback that was registered together with the menu item. It's again quite simple:

```
class procedure TTrayIconHandler.CallTrayMenuCallBack(aID: integer);

Var
  M : TTrayMenuItem;
  arguments: TCefv8ValueArray;

begin
  If not Assigned(FMenu)then exit;
  M:= TTrayMenuItem(FMenu.FindItemID(aID));
  if (M=Nil) then
  SetLength(arguments,1);
  Arguments[0]:=TCefv8ValueRef.NewInt(aID);
  M.Handler.ExecuteFunctionWithContext(M.Context, nil, arguments);
```

```
end;
```

Here we can see that the `M.Handler` property, a function value (which is of type `ICefv8Value`) has a `ExecuteFunctionWithContext` method: this will execute the function in the Javascript context which was captured when the handler was registered. We must pass to this function the ID of the menu item, and this is done by creating an array of `iCEFv8value` interfaces. This array will be available in the Javascript function as the `arguments` variable.

8 Implementing the GUI

All code presented till now will run in the `RENDER` process: the process where the HTML is displayed and Javascript is running.

We now turn to the `BROWSER` process. This is the GUI application that we implement in Lazarus, the LCL.

This is also where the tray icon and it's associated popup menu is handled, and where the memo with the log messages is handled: the main form of the application. The application is similar to the application developed in the previous article.

The `bwBrowser` component we used to display the browser window has a `Chromium` property, representing the `TChromium` instance responsible for the browser window. The `TChromium` class has a `OnProcessMessageReceived` event. This event is triggered whenever the RENDER process sends a message to the BROWSER process. Since we need to be informed when the javascript logs a message or wants to create or remove a menu item, we set this event in the form's `OnCreate` event:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  TLocalResourceHandler.BaseDir:=ExtractFilePath(Paramstr(0));
  TLocalResourceHandler.RegisterHandler('local');
  bwBrowser.Chromium.OnProcessMessageReceived:=@HandleProcessMessage;
  bwBrowser.LoadURL(STrayIconURL);
end;
```

The implementation of this message is again quite simple, and looks exactly the same as its counterpart in the RENDER process:

```
procedure TMainForm.HandleProcessMessage(Sender: TObject;
  const browser: ICefBrowser;
  const frame: ICefFrame;
  sourceProcess: TCefProcessId;
  const aMessage: ICefProcessMessage;
  out Result: Boolean);

Var
 aList : ICEFListValue ;

begin
  aList:=Nil;
  if Assigned(aMessage) then
     aList:=aMessage.ArgumentList;
  if not (Assigned(aList) and (aList.getSize>0)) then
    exit;
```

```
  Case aMessage.Name of
    SMsgSetTrayIconVisibility:
      Application.QueueAsyncCall(@SetTrayIconVisibility,
                                 Ord(aList.getBool(0)));
    SMsgAddTrayMenuItem:
      if (aList.getSize>1) then
        AddTrayMenuItem(aList.getInt(0),
                        utf8Encode(aList.getString(1)));
    SMsgRemoveTrayMenuItem:
      RemoveTrayMenuItem(aList.getInt(0));
    SMsgDisplayLogmessage:
      DoLog('Render process: '+UTF8Encode(aList.getString(0)));
  end;
end;
```

As you can see, we capture 4 messages: the 4 different messages that we implemented in the RENDER process. Note the use of UTF8Encode: strings in the CEF4Delphi APIs are UTF16-encoded unicode strings. Lazarus uses UTF8-encoded strings, so to avoid compiler warnings, we manually convert them.

The first method sets the visibility of the tray icon. It is wrapped in a `QueueAsyncCall` method, because as we've seen in the previous article, the messages can arrive in another thread than the main thread. The implementation is quite simple:

```
procedure TMainForm.SetTrayIconVisibility(aVisible : PtrInt);

begin
 TIApp.Visible:=aVisible<>0;
end;
```

The message signaling that a menu item must be added to the Tray Icon's popup menu is a little more complicated. It gets 2 parameters: the ID identifying the menu item, and the caption:

```
procedure TMainForm.AddTrayMenuItem(aAPIID : integer; ACaption : String);

Var
 Itm : TTrayMenuItem;

begin
  Itm:=TTrayMenuItem.Create(Self);
  Itm.APIID:=aAPIID;
  Itm.Caption:=aCaption;
  Itm.OnClick:=@HandleTrayIconClick;
  PMTray.Items.Add(Itm);
end;
```

As you can see, the code creates a `TTrayMenuItem` menu item, it has an extra property `APIID` which holds the ID that was allocated for the menu item in the RENDER proces.

The method to remove an item is similar, and uses the APIID to find the menu item that must be removed:.

```
procedure TMainForm.RemoveTrayMenuItem(aAPIID : integer);
```

```
Var
  Itm : TTrayMenuItem;
  I : Integer;

begin
  Itm:=Nil;
  I:=PMTray.Items.Count;
  While (I>=0) and (Itm=Nil) do
    begin
    Itm:=TTrayMenuItem(PMTray.Items[i]);
    if (Itm.APIID<>aAPIID) then
      Itm:=Nil;
    Dec(I);
    end;
  Itm.Free;
end;
```

The last message is the logging message. It uses a form variable `FLogMsg` to store the log message, and then uses the `QueueAsyncCall` mechanism to actually display the log message in `ShowLogMsg`:

```
procedure TMainForm.DoLog(const aFmt: String; aArgs: array of const);
begin
  DoLog(Format(aFmt,aArgs))
end;

procedure TMainForm.DoLog(const aMsg: String);
begin
  FLogMsg:=aMsg;
  Application.QueueAsyncCall(@ShowLogMsg,0);
end;

procedure TMainForm.ShowLogMsg(Data: PtrInt);

begin
  MLog.Lines.Add(FLogMsg);
end;
```

There is one more method that must be explained. The menu items created in the `AddTrayMenuItem` method had their `OnClick` handler set to the `HandleTrayIconClick` method. This method needs to send a message to the `RENDER` process to signal the click.

The same mechanism as in the RENDER process is used: create a message object, attach the ID to its list of arguments, and use the `SendProcessMessage` method of `bwBrowser.Chromium` to actually send the message:

```
procedure TMainForm.HandleTrayIconClick(Sender: TObject);

var
  TempMsg    : ICefProcessMessage;
  MID : Integer;
begin
  MID:=(Sender as TTRayMenuItem).APIID;
  DoLog('BROWSER process: click for tray menu item %d',[MID]);
  TempMsg:=TCefProcessMessageRef.New(SMsgExecuteTrayMenuClick);
```

```
  if TempMsg.ArgumentList.SetInt(0,MID ) then
    bwBrowser.Chromium.SendProcessMessage(PID_RENDERER, TempMsg);
end;
```

This message will then be handled by the `HandleProcessMessage` routine of the RENDER process.

9 Using the API in Javascript

After all this coding, it's time to show the fruits of all this labour: actually use the created classes in Javascript. The following very simple HTML is used:

```
<!DOCTYPE html>
<html>
<head>
  <title>Tray icon demo</title>
  <link rel="stylesheet" href="notyf.min.css">
  <script src="notyf.min.js"></script>
  <script src="app.js"></script>
</head>
<body>
<h1>Tray icon.</h1>
<p>The following buttons set the visibility of the tray icon: </p>
<button onclick="makeVisible()">Visible</button>
<button onclick="makeInVisible()">Invisible</button>

<p>Enter a caption and click the button to add an entry to the tray menu:</p>
<input type="text" id="myCaption" value="">
<button onclick="addMenuItem()">Add menu item</button><br>
<input type="text" id="myRemoveId" value="">
<button onclick="removeMenuItem()">Remove menu item</button>

<p>After adding a menu item, you can right-click on the menu item,
and the callback will be executed.</p>
<p>Click count: <span id="count">0</span>,
last ID: <span id="lastid">?</span></p>
</body>
```

the `notyf.min.js` script and associated CSS are a minimalistic message toast implementation. It can be found on

```
https://carlosroso.com/notyf/
```

and it is used in the actual Javascript for our little HTML page, in the `app.js` file:

```
var
  aCount = 0;
  notyf =  new Notyf({position: {x:"center",y:"top"}});

function doLog(msg) {
  console.log(msg);
  notyf.success(msg);
}
```

```
function makeVisible() {
  window.trayIcon.visible=true;
  doLog('TrayIcon: '+window.trayIcon.visible);
}

function trayIconClicked(aID) {
  aCount=aCount+1;
  doLog("Menu item "+aID+" Clicked!");
  document.getElementById("count").innerText=aCount;
  document.getElementById("lastid").innerText=aID;
}

function makeInVisible() {
  window.trayIcon.visible=false;
  doLog('TrayIcon : '+window.trayIcon.visible);
}

function addMenuItem() {
  var aCaption = document.getElementById("myCaption").value;
  var aID = window.trayIcon.addMenuItem(aCaption,trayIconClicked);
  doLog("Menu item with ID "+aID+" Created!");
}

function removeMenuItem() {
  var aID = document.getElementById("myRemoveId").value;
  window.trayIcon.removeMenuItem(parseInt(aID,10));
  doLog("Menu item with ID "+aID+" Created!");
}
```

The `console.log` messages will end up in our memo, and the results of calls will be shown with a `Notyf` sucesss toast as well.

The result of all this work can be admired in the screenshot in figure 2 on page 19.

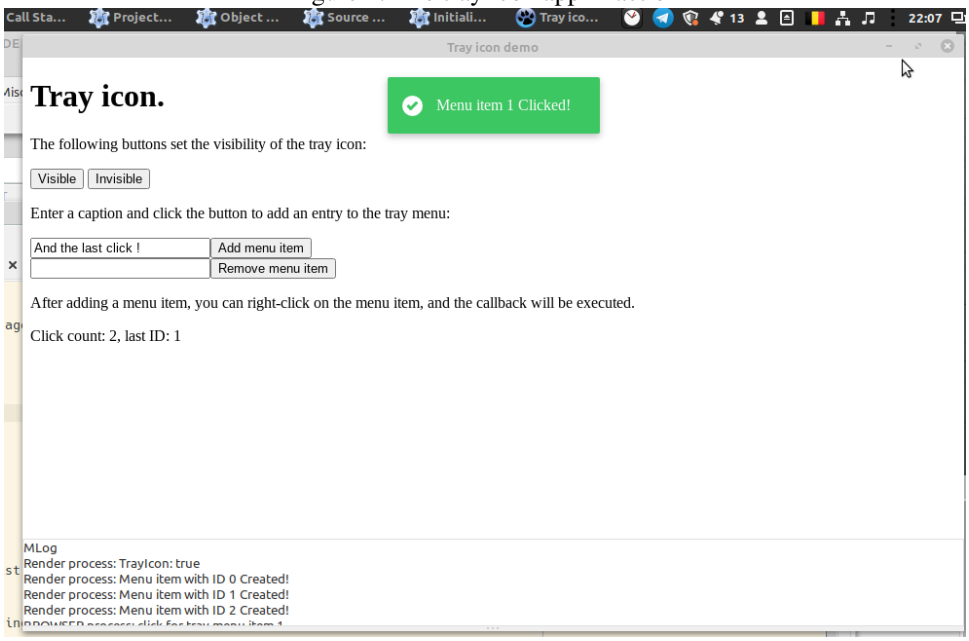
10 Using pas2js

Obviously, we would prefer to make the Javascript program as a pascal program. This is also possible. We need to change the HTML a little bit for this: we must remove the onclick attributes in the button tags and we must make sure all buttons have an ID, so we can attach an onclick handler in code. Like all pas2js programs, we must call `rtl.run()` in the HTML file.

The result of such changes is the following HTML:

```
<!DOCTYPE html>
<html>
<head>
  <title>Tray icon demo</title>
  <link rel="stylesheet" href="notyf.min.css">
  <script src="notyf.min.js"></script>
  <script src="trayicon.js"></script>
</head>
```

Figure 2: The tray icon app in action




```
<body>
<h1>Tray icon.</h1>

<p>The following buttons set the visibility of the tray icon: </p>
<button id="btnVisible">Visible</button>
<button id="btnInvisible">Invisible</button>

<p>Enter a caption and click the button to add an entry to the tray menu:</p>
<input type="text" id="myCaption" value="">
<button id="btnAddMenuItem">Add menu item</button><br>
<input type="text" id="myRemoveId" value="">
<button id="btnRemoveMenuItem">Remove menu item</button>
<p>After adding a menu item, you can right-click on the
menu item, and the callback will be executed.</p>
<p>Click count: <span id="count">0</span>,
last ID: <span id="lastid">?</span></p>

<script>
  rtl.run();
</script>
</body>
```

To be able to use the tray icon in pas2js, we must declare an external class: this allows the compiler to check the validity of the code. As a side effect, the Lazarus code completion will also work.

The definition is as follows:

```
Type
  TClickHandler = reference to procedure (aID : Integer);

  TTrayIcon = class external name 'Object' (TJSObject)
```

```
    visible : boolean;
    function addMenuItem(aCaption :string;
                         aHandler : TClickHandler) : integer;
    procedure removeMenuItem(aID: Integer);
  end;

Var
  trayIcon : TTrayIcon; external name 'window.trayIcon';
```

With this declaration, the compiler knows that there is a TTrayIcon class, and that there is an instance in `window.trayIcon`.

Something similar must be made for the Notyf class, this is done in the unit `libnotyf`. This unit is now part of pas2js.

The `DoLog` procedure can now be written as:

```
var
  notyf : TJSNotyf;

procedure DoLog(Msg : string);

begin
  console.log(msg);
  notyf.success(msg);
end;
```

The instance of `notyf` is create in the program main code block. We must also attach an `onclick` handler to the buttons in our HTML, because the HTML cannot refer to the pas2js methods. For this reason we had to give an id to each button in the HTML.

Attaching the `onClick` handler is also done in the program initialization code:

```
begin
  opts:=TJSNotyfOptions.new;
  opts.position:=TJSNotyfPosition.New;
  opts.position.x:='center';
  opts.position.y:='top';
  notyf:=TJSNotyf.new(opts);
  AddClick('btnVisible',@DoVisibleClick);
  AddClick('btnInvisible',@DoInVisibleClick);
  AddClick('btnAddMenuItem',@DoAddMenuItem);
  AddClick('btnRemoveMenuItem',@DoRemoveMenuItem);
end.
```

As you can see, the creation of the `notyf` instance is done using pascal classes only: the compiler will check your code and guarantees a correctly constructed `Notyf` instance.

The `AddClick` routine is a small helper routine which makes the adding of the onclick handler a little more readable. The code of this routine is quite simple:

```
Procedure AddClick(aName: string; aHandler : TJSRawEventHandler);

Var El : TJSHTMLElement;

begin
  El:=TJSHTMLElement(document.getElementById(aName));
```

```
  El.AddEventListener('click',aHandler);
end;
```

The event handlers to make the tray icon visible or invisible are very simple:

```
Procedure DoVisibleClick(aEvent: TJSEvent);
begin
  trayIcon.visible:=true;
  doLog('TrayIcon: '+BoolToStr(trayIcon.visible));
end;

Procedure DoInVisibleClick(aEvent: TJSEvent);
begin
  trayIcon.visible:=false;
  doLog('TrayIcon: '+BoolToStr(trayIcon.visible));
end;
```

Again, the compiler will check your code.

The code to add and remove a menu item are equally short. For readability, the code to get the value of an input element has been split out to a `getValue` function:

```
function getValue(aID : String) : string;
var
  EL : TJSElement;
begin
  El:=document.getElementById(aID);
  Result:=JSHTMLInputElement(el).Value;
end;

procedure DoAddMenuItem(aEvent : TJSEvent);

Var
  aCaption : String;
  aID : Integer;

begin
  aCaption:=getValue('myCaption');
  aID:=trayIcon.addMenuItem(aCaption,@trayIconClicked);
  DoLog('Menu item with ID '+IntToStr(aID)+' Created!');
end;

procedure DoRemoveMenuItem(aEvent : TJSEvent);

var
  aID : Integer;

begin
  aID:=StrToIntDef(GetValue('myRemoveId'),-1);
  if aID<>-1 then
    trayIcon.removeMenuItem(aID);
  DoLog('Menu item with ID '+IntToStr(aID)+' removed!');
end;
```

The main difference of this code with the corresponding Javascript code is that it is type-safe.

The last routine is the `onclick` handler of the menu item, which we called `trayIconClicked`:

```
procedure trayIconClicked(aID : integer);
begin
  aCount:=aCount+1;
  doLog('Menu item '+IntToStr(aID)+' Clicked!');
  TJSHTMLElement(Document.getElementById('count')).innerText:=IntToStr(aCount);
  TJSHTMLElement(document.getElementById('lastid')).innerText:=IntToStr(aID);
end;
```

And that's all there is to it. The code is a little more verbose than the Javascript code, but it is type-safe and the compiler has verified that all your code is syntactically correct, and that the functions have the correct signatures and get the correct amount of parameters.

The resulting program will of course behave exactly the same as the Javascript version.

11 Conclusion

To add objects to a Javascript environment in CEF is possible, and opens a lot of possibilities for interacting with the OS. It allows you to create an Electron-like environment - much as the 'Miletus' product from TMS Software does. The mechanisms are a little cumbersome, but it should be possible to reduce the amount of needed code by making clever use of RTTI: leveraging (extended) RTTI, it should be possible to directly expose a Pascal class in the Javascript environment, without having to write all this glue code. This we will investigate in a future contribution.