Using the browser APIs from WebAssembly

Michaël Van Canneyt

May 4, 2024

Abstract

WebAssembly modules have no access to the world outside the webassembly virtual machine, except through the APIs that are made available from the host environment. The Browser has lots of APIs, and in this article we show how to make use of *all* possible Browser APIs in webassembly. Moreover we will show that you can use these APIs as if you were programming Javascript directly.

1 Introduction

The WebAssembly support of Free Pascal has been introduced in some previous articles: Free Pascal can compile your pascal code to WebAssembly, and the resulting webassembly file can be run in any hosting environment.

The most used hosting environment is still the browser. Still, many efforts are underway to make webasssembly usable in dedicated containers: this offers the possibility to create safe sandboxed environments for your programs.

Your programs will be safe and sandboxed, because a webassembly can only communicate with the world outside the webassembly through the APIs that are made available by the hosting environment.

The webassembly standard does not specify what APIs a hosting environment needs to expose, it only describes how these APIs can be exposed.

In order for a Free Pascal program to run, it requires the host environment to expose the WASI API to the webassembly. This API is managed separately by the WebAssembly committee, and offers some limited services: file access, getting the time and so on. It provides just the calls that allow the FPC team to implement the SysUtils unit, which provides these basic services to your pascal program.

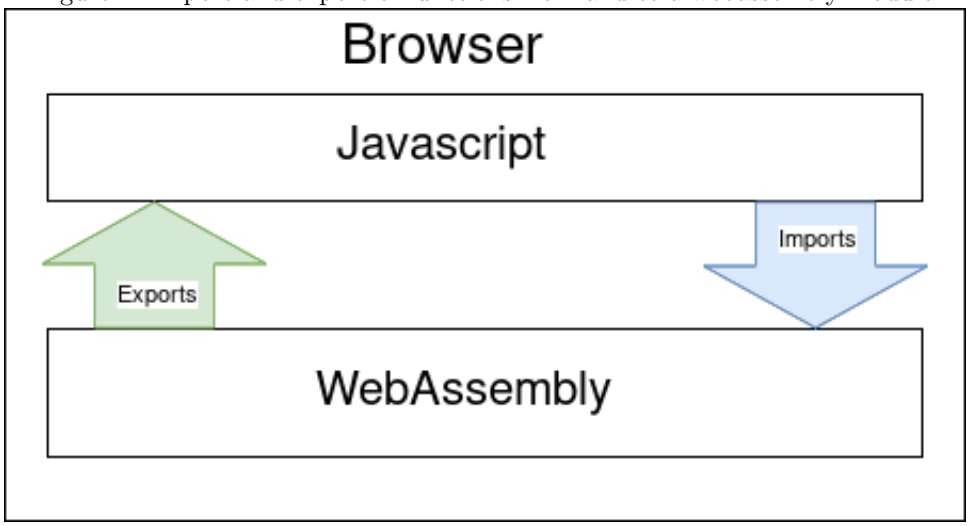
Inversely, a webassembly module can export some functions, which can be called from the hosting environment.

This situation is shown for the browser in figure 1 on page 2: the Javascript in a web page can load a webassembly module. The webassembly module imports some routines made available by the Javascript (the blue arrow), and exports some functions which can be called from Javascript (the green arrow).

In the browser, calling a webassembly function suspends the javascript execution flow: the Javascript waits for the called webassembly function to finish, before it resumes execution. It also means no event handlers will be executed while the Webassembly is executing.

Running a complete program in webassembly simply means calling the `main` function of the application, which must of course be exported from the webassembly; In pascal this means the program begin..end block will be executed.

Figure 1: Import and export of functions from and to a webassembly module



2 The JOB framework

The browser has hundreds of APIs available in Javascript, these APIs are standardized and described in the form of interfaces. For a webassembly program running in the browser, it would be interesting to have access to the full browser API: This would allow the Webassembly program to do everything that can be done in Javascript, with the additional advantages that no-one can read the code, and that for computationally intensive tasks, the webassembly executes faster than Javascript.

Free Pascal now offers a way to access the APIs of the browser: The *Javascript Object Bridge* or `JOB` for short. This development was sponsored by Tixeo, a company interested in porting their software to the browser.

The JOB mechanism (or API) offers a way to create a proxy interface or class in WebAssembly, for any Javascript API. This means that for every class available in Javascript, you can create a class in WebAssembly that will have the same declaration as its counterpart in Javascript.

Whenever you create an instance of a proxy class in WebAssembly, this will automatically create its counterpart in Javascript. When you call a method or set a property on the proxy class, this will call the method or set the property on the Javascript counterpart of the proxy class. All this is transparent for the webassembly programmer: to the webassembly, it is as if he is creating and using classes in WebAssembly.

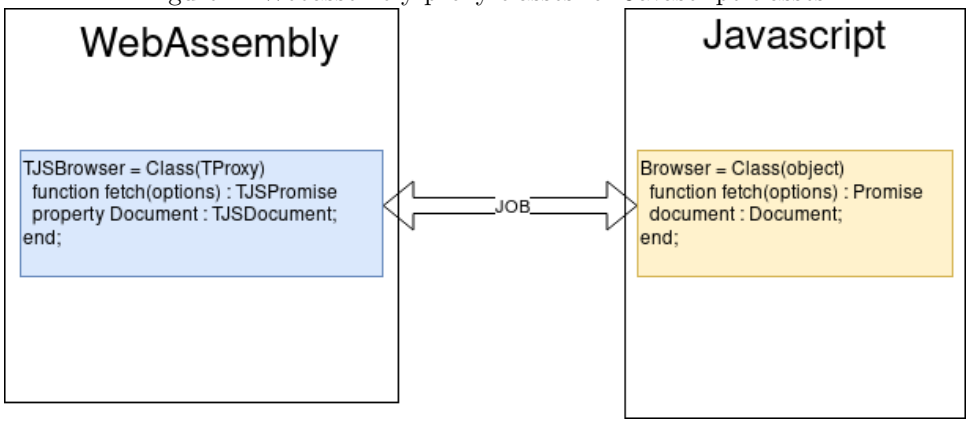
Schematically, this looks like figure 2 on page 3.

JOB does the following things to make this possible:

- You can create a Javascript object, and get a reference to this new object.
- You can get a reference to an existing Javascript object.
- Using this reference, you can call the methods of the object or set its properties as if you were manipulating a native Pascal object.

To illustrate this, in Javascript you can set the caption of a button as follows:

Figure 2: Webassembly proxy classes for Javascript classes




```
document.getElementById("mybutton").innerText="Press me";
```

when using JOB, in your webassembly Pascal program you can write

```
document.getElementById('mybutton').innerText:='Press me';
```

Which is of course a one-to-one translation of the Javascript code.

To understand what happens, let us analyse this code. First of all, the 'document' variable is used. The `document` is exposed in the browser. using JOB, we can define an interface and instantiate a variable:

```
Type
  // The API we want to use.
  IJSDocument = interface(IJSNode)
    function getElementById(const aElementId: UnicodeString): IJSElement;
  end;

  // A class that implements this API.
  TJSDocument = class(TJSNode)
    function getElementById(const aElementId: UnicodeString): IJSElement;
  end;

var
  JSDocument : IJSDocument;

initialization
  JSDocument:=TJSDocument.JOBCreateGlobal('document');
end.
```

The `JOBCreateGlobal` call will retrieve a reference to the `document` instance in Javascript, and uses it to create an instance of the `TJSDocument` proxy for the Document class.

The `getElementById` method is implemented as follows:

```
function TJSDocument.getElementById(const aElementId: UnicodeString): IJSElement;
begin
  Result:=InvokeJSObjectResult('getElementById',
```

```
                                [aElementId],
                                TJSElement) as IJSElement;
end;
```

The `InvokeJSObjectResult` method call is part of the JOB API, and it executes a method in Javascript: the name of the method to call must be specified, as well as any arguments that the method needs.

Since the result will be an object, the Javascript side of JOB will return simply a reference to the resulting object in Javascript (internally, this is an integer). To convert this reference to an actual class instance, the class of the object is specified (`TJSElement`): An instance of this class will be created, passing it the reference returned by the Javascript side of JOB.

When the `getElementById` call returns, the result is a IJSElement interface. On this result, the `innerHTML` property can be set. This is also handled by JOB:

All properties of a Javascript object can be represented by JOB as native pascal properties:

```
Type
  IJSElement = interface(IJSNode)
    function _GetinnerHTML: UnicodeString;
    procedure _SetinnerHTML(const aValue: UnicodeString);
    property innerHTML: UnicodeString read _GetinnerHTML
                                       write _SetinnerHTML;
  end;

  TJSElement = class(TJSNode)
    function _GetinnerHTML: UnicodeString;
    procedure _SetinnerHTML(const aValue: UnicodeString);
    property innerHTML: UnicodeString read _GetinnerHTML
                                       write _SetinnerHTML;
  end;
```

The implementation of the Read/Write accessors is quite simple:

```
function TJSElement._GetinnerHTML: UnicodeString;
begin
  Result:=ReadJSPropertyUnicodeString('innerHTML');
end;

procedure TJSElement._SetinnerHTML(const aValue : UnicodeString);
begin
  WriteJSPropertyUnicodeString('innerHTML',aValue);
end;
```

The use of interfaces make sure that when an (intermediate) object is no longer needed, the object also released on the Javascript side.

To make all this possible, on the Javascript side, the JOB API consists of (currently) 11 API methods. When these 11 methods are implemented, the webassembly can use proxy classes to execute any method on any object in the browser. A default Javascript implementation for JOB has been developed using Pas2JS (naturally), but one could write this API in plain Javascript as well.

The JOB technology is implemented in 2 units:

Job.js for the webassembly program: it implements the various JOB calls that handle encoding a call to the Javascript side of things, sends the call description to the Javascript environment and when the call returns, it retrieves the result and converts it, if needed, to an object instance.

Job_Browser for the pas2 program. This implements the decoding of a call, executes the call on the Javascript object, and when the call returns, it encodes the result and sends it back to the WebAssembly.

There is a third (shared) unit which contains some common constants and types that make up the JOB API.

3 WebIDL2pas revisited

In the above code examples, we showed how to access arbitrary methods and properties of some Javascript objects. The examples made use of an interface and a class that implements this interface. It makes clear that for every method you wish to call and for every property you wish to get or set, a small piece of 'glue' code needs to be created: a proxy object for every Javascript object.

If all classes and APIs of the browser must be encoded like this, this is a lot of work.

You could call the JOB methods directly, in that case no classes and no glue code needs to be produced. The disadvantage of that approach is that there is no type safety, and no code completion if you want to code in the IDE. You also will need to manage the lifetime of the objects explicitly.

Luckily, there is no need to code all these proxy classes. This task can be automated.

All browser APIs are standardized by the W3C committees using a IDL (Interface Definition Language) called WebIDL. All browser creators use these IDL files to implement their Javascript APIs. The Mozilla foundation maintains these files, they are available at:

`https://hg.mozilla.org/mozilla-central/file/tip/dom/webidl`

or on:

`https://github.com/mozilla/gecko-dev.git`

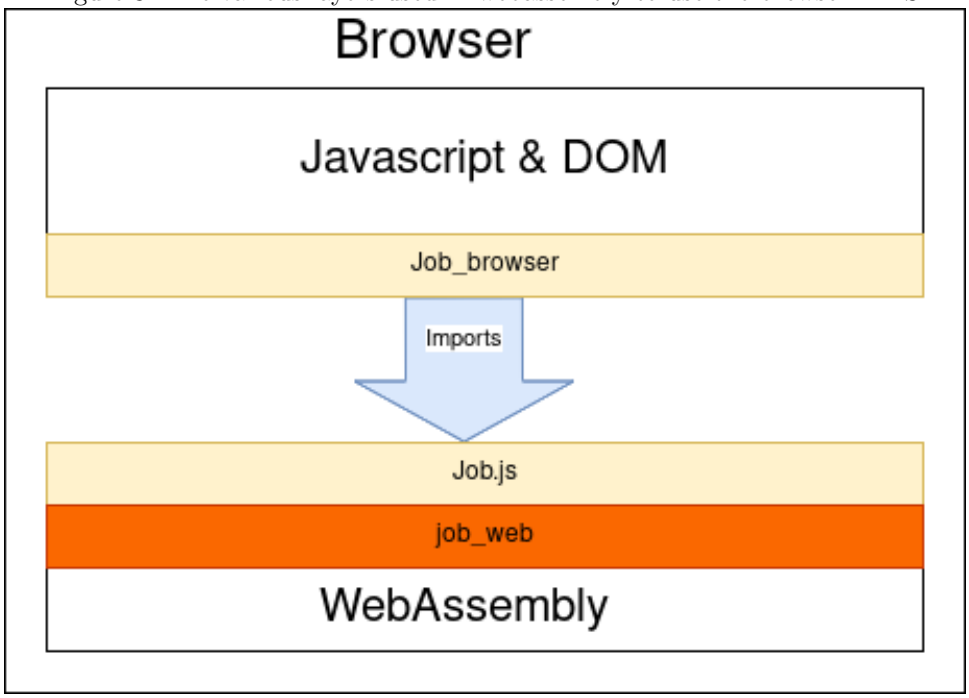
There are some really minor differences between these archives, most likely due to the time it takes to synchronize. As you can see in these archives, there are more than 700 files, representing all the APIs made available by the browser.

In a previous article on Pas2js, the webidl2pas tool that comes with Free Pascal and pas2js was discussed. This tool can transform a .webidl file to a Pascal external class definition that can be used by Pas2JS. The tool has been adapted so it can now also create the proxy classes to access all the browser APIs from webassembly.

By downloading and concatenating all .webidl files from the above sources and applying some small patches (the files are not perfect, and one or two constructs are not possible in Pascal), a pascal unit can be produced that describes all these APIs.

Such a file has been committed to the FPC git repository: `job_web` (the file is located in packages/wasm-job/examples). The file is huge. The interface section contains is about 80.000 lines long and contains roughly 1600 interface declarations, and a similar amount of classes. This represents all the available browser APIs: by using

Figure 3: The various layers used in webassembly to use the browser APIS



this file in your webassembly program, you have direct access to all possible browser APIs.

Diagrammatic, the architecture of a web application wishing to use the Javascript and DOM APIs using JOB looks like figure 3 on page 6.

4 A javascript camera application

To make all this a little more understandable, we'll create an example: a web page where we have a video element, connected to the camera, and a canvas where we can create a picture (a still) of what the camera is showing. Basically, a camera application as you would have it on your smartphone.

We will make this application first in Javascript, then in Pas2JS and lastly we'll make it using a webassembly program. We'll show how the code is similar at each stage.

The HTML for this webpage will be the same in all 3 cases. The actual program will be in the `camera.js` javascript file:

```
<!doctype html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Video capture still - Javascript</title>
    <link rel="stylesheet" href="css/bulma.min.css">
    <link rel="stylesheet" href="css/camera.css">
    <script src="camera.js" type="application/javascript"></script>
  </head>
```

```
  <body>
      <div class="container">
          <h1 class="title is-1">Capture still from video</h1>
          <div class="columns">
            <div class="camera column">
              <video id="video">Video stream not available.</video>
            </div>
            <div class=" column">
                <canvas id="canvas" ></canvas>
            </div>a
          </div>
          <div class="box columns is-centered">
            <div class="column is-3">
              <button id="start" class="button is-info"></button>
              <button id="still" class="button is-link"></button>
            </div>
          </div>
      </div>
  </body>
</html>
```

As usual we use some Bulma CSS classes to format the page. There are 4 elements which are important, so they have an `id` attribute:

video a video element, which will show the camera feed.

canvas a canvas element, which will show the still.

start a button to start the camera feed. When pressed, this will ask for permission to use the camera. Note that this button does not show a caption, it will be set in code.

still a button to create a still (photo) from the camera feed. Similarly, the caption for the button will be set in code.

The `id` attribute is used to get a reference to the elements when the page is loaded, in the `camera.js` javascript program:

```
var video = null;
var canvas = null;
var context = null;
var photo = null;
var startbutton = null;
var stillbutton = null;

function startup() {
   video = document.getElementById('video');
   canvas = document.getElementById('canvas');
   context = canvas.getContext('2d');

   startbutton = document.getElementById('start');
   startbutton.innerText = 'Start video';
   startbutton.addEventListener('click', startvideo);

   stillbutton = document.getElementById('still');
```

```
   stillbutton.innerText = 'Create still';
   stillbutton.addEventListener('click', createstill);
}

window.addEventListener('load', startup);
```

Note how a reference to each of the 4 elements is stored in a variable. We also store a context for the canvas, this context is used later to draw on the canvas.

The `startvideo` event handler is called when the user clicks the 'start' button:

```
function startvideo(ev) {
  navigator.mediaDevices.getUserMedia({
          video: true,
          audio: false
        })
        .then(function(stream) {
            video.srcObject = stream;
            video.play();
        })
        .catch(function(err) {
            console.log("An error occurred: " + err);
        });
}
```

The `getUserMedia` call will ask for permission to use the camera. This function returns a promise, and when the promise resolves, the stream is coupled to the video element.

Lastly, the `'click'` handler for the 'still' button draws the current video frame on the canvas:

```
function createstill(ev) {
  canvas.width = video.clientWidth;
  canvas.height = video.clientHeight;
  context.drawImage(video, 0, 0, video.clientWidth, video.clientHeight);
}
```

And that's all there is to creating a camera program using the browser. You can load this page from a webserver using the browser, or you can open it by double-clicking the file in the file explorer: your default browser will open and show the application. In both cases, the program will function.

5 The camera application in Pas2js

In a first step, we will code the camera application in pas2js. This will allow us to transform the Javascript to pascal, without concerning ourselves with the details of using webassembly.

The first thing to do is to add the mandatory script tag for running a pas2js application to the HTML:

```
<script>
window.addEventListener('load', rtl.run);
</script>
```

Then we translate our program piece by piece. We will put all code in a class, it will become apparent in the next example why this is necessary.

```
TCameraApp = class
  video : TJSHTMLVideoElement;
  canvas : TJSHTMLCanvasElement;
  context : TJSCanvasRenderingContext2D;
  startbutton : TJSHTMLElement;
  stillbutton : TJSHTMLElement;
  function StartStream(JS : JSValue) : JSValue;
  function DoError(JS : JSValue) : JSValue;
  Procedure StartVideo(Event: TJSEvent);
  Procedure CreateStill(Event: TJSEvent);
  procedure Run;
end;
```

This class declares the same variables and functions as our Javascript code. The main difference is of course that Pascal is a strongly typed language, and we must specify the types of all variables, method arguments and function results.

The main program simply creates an instance of this class and calls the `Run` method:

```
With TCameraApp.Create do
  Run;
```

The run method looks suspiciously familiar:

```
Procedure TCameraApp.Run;

begin
  video:=TJSHTMLVideoElement(document.getElementById('video'));
  canvas:=TJSHTMLCanvasElement(document.getElementById('canvas'));
  context:=TJSCanvasRenderingContext2D(canvas.getContext('2d'));

  startbutton:=TJSHTMLElement(document.getElementById('start'));
  startbutton.innerText:='Start video';
  startbutton.addEventListener('click', @startvideo);


  stillbutton:=TJSHTMLElement(document.getElementById('still'));
  stillbutton.innerText:='Create still';
  stillbutton.addEventListener('click', @createstill);
end;
```

As you can see, this method is an almost copy-and-paste of the main javascript method. The biggest difference is the typecasts, which are of course needed to keep the Pascal compiler happy.

The `StartVideo` callback is slightly different. Pas2js' `Web` unit contains a typed defintion of the constraints argument to the `getUserMedia` call. Using an instance of this class allows us to make sure that the correct elements are specified. We also don't use anonymous methods (although this would be possible), but use named functions to handle the various possible outcomes of the promise:

```
Procedure TCameraApp.StartVideo(Event: TJSEvent);
```

```
var
  constraints : TJSMediaConstraints;

begin
  constraints:=TJSMediaConstraints.new;
  constraints.video:=True;
  constraints.audio:=False;
  Window.navigator.mediaDevices.getUserMedia(constraints)
      ._then(@StartStream)
      .catch(@DoError)
end;
```

The `StartStream` method is executed when the promise resolves correctly. The promise resolved result (`JS`) must be typecast to the correct class before we can assign it to the `srcObject` property of the video element:

```
function TCameraApp.StartStream(JS : JSValue) : JSValue;
begin
  Result:=Undefined;
  video.srcObject:=TJSHTMLMediaStream(JS);
  video.play();
end;
```

Other than that, the code is identical to the Javascript implementation. The same is true for the `DoError` method:

```
function TCameraApp.DoError(JS : JSValue) : JSValue;

begin
  Result:=Undefined;
  console.log('An error occurred: ' + String(JS));
end;
```

Lastly, the `'click'` event handler of the still button is again almost a copy and paste of the corresponding Javascript code.

```
Procedure TCameraApp.CreateStill(Event: TJSEvent);

begin
  canvas.width:=video.clientWidth;
  canvas.height:=video.clientHeight;
  context.drawImage(video, 0, 0, video.clientWidth, video.clientHeight);
end;
```

And with this the demo application is translated to pascal.

The workings of this application are no different from the pure Javascript version, and the Pascal code is - disregarding its Pascal nature - the same as the Javascript code.

6 The camera application in WebAssembly

Lastly, we come to the part that is the focus of this article: the webassembly program.

To make this application using webassembly, we need to create actually 2 applications: the webassembly loader program, and the webassembly program itself.

The former is a small boilerplate application, created with pas2js. It is a generic program that can be used to load any webassembly program that uses JOB to communicate with the browser APIs.

The webassembly program is actually a library: in the initialization, the necessary callbacks are set up and then it needs to return control to the browser in order for the Javascript event loop to be run.

The program logic is implemented in `TMyApplication`. This class is a descendant of `TBrowserWASIHostApplication`.

The `TBrowserWASIHostApplication` class, in turn, is a `TCustomApplication` descendant which allows you to start a WebAssembly module written in Free Pascal: it has been introduced in an earlier article on FPC Webassembly support.

The class needs very little methods: a constructor, the `DoRun` method, and an `OnBeforeStart` method.

Note the `JOB_Browser` unit in the uses clause: this unit contains the `TJSObjectBridge` class, which is the implementation of the JOB mechanism:

```
program camera;

{$mode objfpc}

uses
  JS, Classes, SysUtils, Web, WasiEnv, WasiHostApp, JOB_Browser, JOB_Shared;

Type

  TMyApplication = class(TBrowserWASIHostApplication)
  Private
    FJOB: TJSObjectBridge;
    function OnBeforeStart(Sender: TObject;
      aDescriptor: TWebAssemblyStartDescriptor): Boolean;
  Public
    constructor Create(aOwner : TComponent); override;
    procedure DoRun; override;
  end;
```

The `TJSObjectBridge` is the class that registers the needed JOB functions in the webassembly. Under normal circumstances, only 1 property of this class needs to be set in order for it to do its work: the `WasiExports` property. Other than that it performs its work completely in the background.

So, we create an instance of `TJSObjectBridge`, pass it the `WasiEnvironment` so it can register itself with the Webassembly modules that are loaded later on, and store a reference to it in `FJOB`:

```
constructor TMyApplication.Create(aOwner: TComponent);
begin
  inherited Create(aOwner);
  FJOB:=TJSObjectBridge.Create(WasiEnvironment);
  RunEntryFunction:='_initialize';
end;
```

The last line in this function sets `RunEntryFunction` to `_initialize`. This must be done because our webassembly module is a library:

The default run entry point (used for programs) is `_start`. For a library, only the initialization of the library must be performed, and the exported function that handles this initialization is called `_initialize`.

In the `DoRun` method, we simply call `StartWebAssembly`, passing it the name of the created webassembly function

```
procedure TMyApplication.DoRun;

var
  wasm : String;

begin
  Terminate;
  // Allow to load file specified in hash: index.html#mywasmfile.wasm
  Wasm:=ParamStr(1);
  if Wasm='' then
    Wasm:='wasmcamera.wasm';
  StartWebAssembly(Wasm,true,@OnBeforeStart);
end;
```

The `ParamStr(1)` retrieves the first name after the hash sign in the URL. If set, then it is interpreted as the name of the webassembly file to load. If not set, we use 'wasmcamera.wasm' as the name.

The `StartWebAssembly` function will load the requested webassembly and executes the run entry function. (in our case, `_initialize`). The last parameter is an event which is executed right before calling the run entry function: this allows the caller to do extra initialization after the webassembly module was loaded, but before the start function is called.

The event handler sets the `WasiExports` property of the `TJSObjectBridge` instance to the list of exported functions from the webassembly:

```
function TMyApplication.OnBeforeStart(Sender: TObject;
  aDescriptor: TWebAssemblyStartDescriptor): Boolean;
begin
  FJOB.WasiExports:=aDescriptor.Exported;
  Result:=true;
end;
```

The JOB framework needs a single exported function which it uses to call callback functions (event handlers) in webassembly. It searches this function in the list in `WasiExports`.

All that is left to do is to create and initialize an instance of our application class and call the `Run` method, the usual code needed when using the application class:

```
var
  Application : TMyApplication;

begin
  Application:=TMyApplication.Create(nil);
  Application.Initialize;
```

```
  Application.Run;
end.
```

With this, the loader for our webassembly module is finished. Note that there is no code specific to our camera application: this is completely generic code that can be used to load any webassembly module which needs the JOB framework.

So now we turn to the code for our webassembly module, which is implemented as a library. It starts out in the usual way:

```
library wasmcamera;

{$mode objfpc}
{$h+}
{$codepage UTF8}

uses
  SysUtils, Variants, Job.Js, JOB_Web;
```

Note that it uses the `Job.Js` unit with the implementation of the webassembly side of the JOB mechanism, and the `JOB_WEB` unit, which was generated by the `webidl2pas` tool. Then it defines the `TCameraApp` class:

```
type
  TCameraApp = class
    Video: IJSHTMLVideoElement;
    Canvas: IJSHTMLCanvasElement;
    StartButton: IJSHTMLButtonElement;
    StillButton: IJSHTMLButtonElement;
    Context: IJSCanvasRenderingContext2D;
    function StartStream(const Res : Variant) : Variant;
    function DoError(const Res : Variant) : Variant;
    procedure StartVideo(Event: IJSEvent);
    procedure CreateStill(Event: IJSEvent);
    procedure Run;
  end;
```

As you can see, this class is virtually identical to the class for the Pas2js program. The only thing that changes are some types: Instead of classes (using prefix `TJS`) we use interfaces (using prefix `IJS`). The pas2js `JSValue` is replaced with `Variant`: both correspond to the `any` type in the IDL descriptions of the APIs.

The `Run` method, which actually will initialize our application, is almost a copy of the pas2js code:

```
procedure TCameraApp.Run;
begin
  Video:=TJSHTMLVideoElement.Cast(JSDocument.getElementById('video'));
  Canvas:=TJSHTMLCanvasElement.Cast(JSDocument.getElementById('canvas'));
  Context:=TJSCanvasRenderingContext2D.Cast(Canvas.getContext('2d'));

  StartButton:=TJSHTMLButtonElement.Cast(JSDocument.getElementById('start'));
  StartButton.InnerHTML:='Start video';
  StartButton.addEventListener('click', @StartVideo);
```

```
  StillButton:=TJSHTMLButtonElement.Cast(JSDocument.getElementById('still'));
  StillButton.InnerHTML:='Create still';
  StillButton.addEventListener('click', @CreateStill);
end;
```

Note the calls to the `Cast` class method in order to do a typecast from one interface type (in this case `IJSElement`) to another interface type.

This is needed in order to be able to do some reference count housekeeping. A regular typecast would result in wrong reference counts and could lead to objects being destroyed in Javascript when they're still used in the webassembly.

Other than that, the code is identical to the pas2js code or the Javascript code: no trickery is needed to set the callbacks, a real pascal event handler can be used.

It should be noted that all event handlers are declared with `'of object'`, meaning that only methods of classes can be used as callback handlers, plain routines cannot be used.

The `StartVideo` callback handler also looks surpisingly familiar:

```
Procedure TCameraApp.StartVideo(Event: IJSEvent);

var
  constraints : TJSMediaStreamConstraints;

begin
  constraints:=TJSMediaStreamConstraints.Create;
  constraints.video:=True;
  constraints.audio:=False;
  JSWindow.navigator.mediaDevices.getUserMedia(Constraints)
    ._then(@StartStream)
    .catch(@DoError)
end;
```

Except for a constructor that is named `Create`, as opposed to the customary `New` in pas2js, the code is identical.

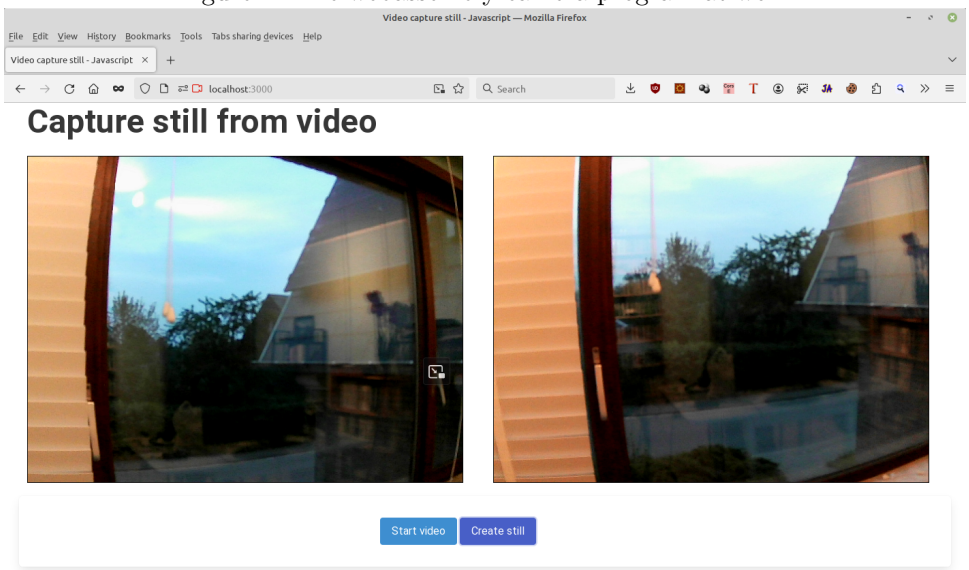
The `getUserMedia` returns a promise, and when this is resolved `StartStream` is called, which is again a copy of the pas2js method:

```
function TCameraApp.StartStream(const Res : Variant) : Variant;

var
  Stream : IJSMediaStream;

begin
  Stream:=IJSMediaStream(Res);
  Video.srcObject := Stream;
  Video.play();
end;
```

In case of an error, `doError` is called. Again, no change in code:

```
function TCameraApp.DoError(const Res : Variant) : Variant;

begin
```

Figure 4: The webassembly camera program at work




```
  writeln('Error accessing the webcam: '+string(Res));
end;
```

The code for the `CreateStill` method, is also unchanged:

```
procedure TCameraApp.CreateStill(Event: IJSEvent);

begin
  Canvas.width:=Video.clientWidth;
  Canvas.height:=Video.clientHeight;
  Context.drawImage(Video,0,0,Video.ClientWidth,Video.ClientHeight);
end;
```

All that is left to do is to export a callback function which the JOB framework needs (`JOBCallBack`, implemented in the `Job.Js` unit), and to create an instance of our camera application class:

```
exports
  JOBCallback;

begin
  With TCameraApp.Create do
    Run;
end.
```

For all practical purposes, the webassembly program can be coded as the javascript version or Pas2js version would be coded.

The result of all this work is shown in figure 4 on page 15.

7 Using custom objects

JOB is used to give access to all the browser APIs. However, is it also possible to use custom objects created in Pas2js or any other Javascript API from any Javascript framework? The answer is 'Yes, of course'. You can perfectly code a webassembly proxy for a Pas2js pascal class or a Javascript class. If a .webidl exists for the Javascript class, the proxy code could be generated by the webidl2pas tool.

Javascript classes have a function that serves as the constructor. If this is a globally registered function, the JOB framework will find the function: it looks for the constructor function in the global (`window`) scope. All that is needed is to declare the name of this function in the webassembly proxy class.

For Pas2js classes, you can specify a constructing function in the host environment. Given the following class implemented in Pas2js:

```
TMyObject = Class(TObject)
private
  fa: String; external name 'a';
public
  Constructor Create(aValue : string);
  Property a : String Read fa write fa;
end;

constructor TMyObject.Create(aValue: string);
begin
  fa:=aValue;
end;
```

You can create a constructor function to create a Javascript instance of this function. The constructor function accepts the name of the requested object, and the parameters for the constructor which are provided in an array of JSValue (variants, for all practical purposes).

In the host application presented earlier, this would mean adding a method as follows:

```
function TMyApplication.CreateMyObject(const aName: String;
                                       aArgs: TJSValueDynArray): TObject;
begin
  Result:=TMyObject.Create(String(aArgs[0]));
end;
```

Note that because the `aName` parameter contains the requested class name, you can use a single constructor function to construct many classes.

Registering the constructor function with the JOB framework is done using the `RegisterObjectFactory` call of the `TJSObjectBridge` class:

```
FJOB.RegisterObjectFactory('MyObject',@CreateMyObject);
```

You can do this call right after creating the `TJSObjectBridge` class.

If some Javascript class does not register itself in the global scope, then the JOB implementation will not find it without help. You can register a function that creates a regular Javascript object in a similar manner as for a Pascal class:

```
function TMyApplication.CreateBrowserObject(const aName: String;
```

```
                                                aArgs: TJSValueDynArray): TJSObject;
begin
  Result:=TJSObject.New;
  Result['Aloha']:=String(aArgs[0]);
end;
```

In the above example, a plain Javascript '`Object`' instance is created, but in fact any Javascript object can be returned. This constructor function must also be registered with the `RegisterJSObjectFactory` call:

```
FJOB.RegisterJSObjectFactory('MyBrowserObject',@CreateBrowserObject);
```

The reason that 2 different calls are needed is that from an Object Pascal point of view, the Javascript `TJSObject` inheritance tree is distinct from the Object Pascal `TObject` inheritance tree.

After these calls, when the webassembly part of JOB needs to create an instance of `MyObject` or a `MyBrowserObject`, the correct registered function will be called to create an instance. The necessary housekeeping will be done as it is done for Browser-provided objects: associate an ID with the object, and return that ID to the webassembly.

The webassembly proxy interface and class for the `TMyObject` class look as follows:

```
IJSTestObj = Interface (IJSObject)
   ['{DE03E9A4-3960-4090-A3FA-387B61E8AEA9}']
  function GetStringAttr : UnicodeString;
  procedure SetStringAttr(const aValue : UnicodeString);
  property StringAttr : Unicodestring Read GetStringAttr
                                      Write SetStringAttr;
end;

TMyTestObj = Class(TJSObject,IJSTestObj)
  constructor Create(a: String);
  class function JSClassName: UnicodeString; override;
  function GetStringAttr : UnicodeString;
  procedure SetStringAttr(const aValue : UnicodeString);
  property StringAttr : Unicodestring Read GetStringAttr
                                      Write SetStringAttr;
end;
```

The implementation of the proxy class is simple. The `JOBCreate` method of `TJSObject` can be used to construct a new object. In order to do its work, it needs to know the class name of the Javascript class. It expects the `JSClassName` class function to return the correct class name, so we override that function and let it return the name we used to register our constructor function:

```
class function TMyTestObj.JSClassName: UnicodeString;
begin
  Result:='MyObject';
end;

constructor TMyTestObj.Create(a: String);
begin
  Inherited JobCreate([a]);
end;
```

The `JOBCreate` method accepts parameters as an array of const, which are encoded and sent to the browser side.

The implementation of the property getters and setters are simple:

```
function TMyTestObj.GetStringAttr: UnicodeString;
begin
  Result:=ReadJSPropertyUnicodeString('a');
end;

procedure TMyTestObj.SetStringAttr(const aValue: UnicodeString);
begin
  WriteJSPropertyUnicodeString('a',aValue);
end;
```

Similarly named `ReadJSProperty*` and `WriteJSProperty*` calls exist for all simple Pascal types, you must choose the function that corresponds to the type of the property in your Javascript class.

Note that the name of the field is given as `'a'`: this is the Javascript name of the field in the Pascal class: it was forced to 'a' using the `external name 'a'` modifier in the class declaration. Without this modifier, `'fa'` would need to be used.

If a property must be set using a setter/getter, then you must adapt the proxy code accordingly, of course: you must then code a call to the getter and setter.

The class is now ready for use in your webassembly program:

```
var
  T : IJSTestObj;

begin
  Writeln('Creating TMyTestObj object');
  T:=TMyTestObj.Create('solo');
  Writeln('Property : ',T.StringAttr);
end;
```

The expected output is of course 'solo' for the property value.

The source code that demonstrates this is included in the pas2js suite of demos, under the `demo/wasienv/job/simple` directory.

8 Conclusion

With the JOB technology, it is now possible to use all browser APIs in a webassembly program without having to resort to lots of import/export routines: a list of 11 functions is sufficient to create and use every possible browser object. To the best of the author's knowledge, currently the only other compiled language – compilable to WebAssembly – that offers this possibility is Rust.

As indicated above, the `job_web` unit is large. This is somewhat of a disadvantage: the compiler takes a lot of time compiling this unit, well over 1 minute. The reason is the use of interfaces, which result in a lot of hidden code to call methods on an interface, and the resulting unit is well over 65Mb. While the linker removes all unused code and your program will contain only the needed code, the unit must be compiled (luckily only once) and this takes time.

For this reason, an extension of the webidl2pas program is envisioned: by passing it a list of classes that are actually needed, it can determine which types are needed (including of course all implicit types), and can then proceed to create code for only these needed types. By passing it the `HTMLVideoElement` and `HTMLCanvasElement` as needed classes, it would then create only the classes that are actually needed to create our program.

But in the meantime, the `job_web` unit can be used to create webassembly programs that have all the advantages of native code, but which can additionally create stunning user interfaces using the browser APIs.