Embedding JavaScript in an Object Pascal application

Michaël Van Canneyt

January 31, 2010

Abstract

Javascript is a scripting language targeted mainly at the web-browser. Libsee is a library that provides a Javascript engine which can be embedded in any application. In this article we show how this can be done for applications written in Object Pascal.

1 Introduction

Javascript is abundant in web-applications. Indeed, it is the standard language for scripting web pages. It is standardized in the ECMAScript language description: ECMA-262. However, it's use is not limited to the browser: any application that needs scripting capabilities can use Javascript. Several javascript engines exist that can be embedded in an existing application. Libsee (Simple ECMAScript Engine) is one of these engines, written in C, so it can be accessed from an Object Pascal program.

In this article exactly that will be done: the library will be embedded in an Object Pascal program. The C header files of libsee have been translated to Pascal, and the unit containing the translation will be used to access the Libsee library. Since Javascript by default has no features to interact with the host program or the operating system, it will be shown how to construct such an interface.

2 Library installation

Libsee is not included in all Linux distributions or installed on Windows. So it must be downloaded and installed before it can be used. Libsee is written by David Leonard, and can be downloaded from

```
http://www.adaptive-enterprises.com.au/~d/software/see/
```

The current version – at the time of writing – is 3.1, the sources are in a see-3.1.1424.tar.gz archive. The compilation and installation of the library are very simple. Assuming a source archive was downloaded, the library can be compiled and installed on unix systems with the following commands:

```
gzip -d see-3.1.1424.tar.gz | tar xf -
cd see-3.1.1424
./configure
./make all
./make install
```

Prior to compiling, make sure that a Boehm Garbage Collection library is installed. This can be installed on most systems using the system package manager.

For Windows, a cygwin (or mingw) installation can be used to compile the library; No pre-compiled library is available, although a pre-compiled binary of the JS command shell (see-shell) is available.

3 Using the library

The libsee unit contains the interface to the library. It contains all calls available in the library, and defines all types available in the library. All type names have been prepended with T (as customary in Object pascal), and a pointer type has been defined for each type as well.

The libsee library must be loaded dynamically by the program that wishes to use it: the unit initialization code loads the default library, but there are 3 calls available to control which library to load:

```
procedure LoadLibsee(Const Alib : string);
procedure FreeLibsee;
Function  LibseeLoaded : Boolean;
```

The default library name is available in the `LibSeeLibraryName` constant.

After the library has been succesfully loaded, the `see_init` call must be used to initialize it. After this, libsee is ready for use.

4 Running Javascript code

Embedded Javascript is interpreted. To run javascript in a program the following steps must be taken:

- A Javascript interpreter must be created. The interpreter is a record type: `TSEE_interpreter`, which must be initialized prior to using it. This is done using the `SEE_interpreter_init` call:

  ```
  procedure SEE_interpreter_init(i:PTSEE_interpreter);
  ```

  Multiple interpreters can be used simultaneously, independently of one another, so multiple scripts can be run at once.

- The interpreter can be used to evaluate a JavaScript program: this is done using the `See_global_eval` call:

  ```
  procedure SEE_Global_eval(i:PTSEE_interpreter;
                            input:PTSEE_input;
                            res:PTSEE_value);
  ```

  This evaluates the program in the global JavaScript context. The `input` parameter is a record containing some callbacks, used to read the input to the interpreter.

- The result of the interpreter is a `TSEE_Value` value. The `TSEE_Value` is a record that represents a Javascript value.

A Javascript value is defined in the following record, resembling the variant type used in Object Pascal:

```
TSEE_type = (SEE_UNDEFINED,SEE_NULL,SEE_BOOLEAN,
             SEE_NUMBER, SEE_STRING,SEE_OBJECT);

TSEE_value = record
  _type : TSEE_type;
  u : record
    case longint of
      0 : ( number : TSEE_number_t );
      1 : ( boolean : TSEE_boolean_t );
      2 : ( _object : PTSEE_object );
      3 : ( _string : PTSEE_string );
    end;
  end;
```

The `_type` field determines the type of value held by the record, and can be one of the following enumerated values:

SEE_UNDEFINED an undefined value.

SEE_NULL a null value.

SEE_BOOLEAN a boolean value.

SEE_NUMBER a numerical value: all numerical values are doubles, no integer values exist.

SEE_STRING a string value, which in itself is contained in a record.

SEE_OBJECT an object value, also described by a record structure.

There are, in fact, 2 more types available, but they are for internal use by the LibSee engine only, so they will not be treated here.

This is all that one should know in order to evaluate a javascript program. The following puts it all together:

```
program tlibsee;

uses libsee;

Var
  Interp : TSEE_interpreter;
  ainput : PSEE_INPUT;
  res : TSEE_Value;

Const
  Program_text = 'Math.sqrt(3 + 4 * 7)+9;';

begin
  see_init;
  SEE_interpreter_init(@interp);
  ainput :=SEE_input_utf8(@interp, pchar(program_text));
  See_global_eval(@interp,ainput,@res);
```

```
  if (res._type=SEE_NUMBER) then
    Writeln('Result is : ',res.u.number)
  else
    Writeln('Result is not a number');
  see_input_close(ainput);
end.
```

The `SEE_input_utf8` call creates an input object which reads its input from a null-terminated string, which is subsequently used as the input for the `SEE_global_eval` call. After the call is finished, the `res` variable contains the result of the program (if any): it is examined and the result written to screen. After this, the input for the interpreter is closed.

The output of this simple program is the following:

```
Result is :  1.45677643628300E+001
```

Which is the expected result.

5 Creating simple functions

Javascript does not define any native functions that allow the running programm to interact with the environment. In the browser, some objects are defined that make interaction with the browser possible: the `window`, `navigator` and `document` (the DOM tree for the currently shown HTML document) objects give some control over the browser: these objects are standardized, and make it possible to write web applications in a - pretty much - browser independent way.

In a user program, no such objects exist, and they must be provided somehow. To demonstrate how this is done, a set of 'write' and 'writeln' calls that act as their Pascal counterparts will be added to the Libsee JavaScript environment.

Libsee provides an API to add native objects or functions to the interpreter, in a so-called module API. The running program can provide libsee with a number of modules, each module providing a set of functionalities. Each module is described in the `TSEE_module` record:

```
TSEE_module = record
  magic : TSEE_uint32_t;
  name : PTcchar;
  version : PTcchar;
  index : Tcuint;
  mod_init : function :Tcint;cdecl;
  alloc : procedure (para1:PTSEE_interpreter);cdecl;
  init : procedure (para1:PTSEE_interpreter);cdecl;
end;
```

The first field must contain a magic number and is used for version control: Libsee provides a constant with the magic number. The second and third fields contain null-terminated strings containing the name and version of the module. The `index` field is filled with a sequential number by the libsee module API: the number can be used to identify the module in the list of registered modules.

The last 3 fields contain callbacks which will be called by the libsee module API at various stages:

mod_init this is called exactly once for each module, before an interpreter was initialized. If the function returns 0, the initialization is considered succesful. A nonzero value means failure.

alloc This is called for each module after all modules were initialized. This is useful when modules need to refer to one another in their initialization code.

init this is called each time an interpreter is initialized. This must be used to set up the actions of the module for that interpreter: Here the functions, objects and other variables must be registered in the interpreter instance.

An instance of the `TSEE_Module` record with proper field values must be registered with the libsee module system. This is done with the `SEE_module_add` call. For the writeln support, the following can be used:

```
Var
  WriteModule : TSEE_module;

Procedure RegisterWriteModule;

begin
  With WriteModule do
    begin
    magic:=SEE_MODULE_MAGIC;
    name:='Write';
    version:='1.0';
    Index:=0;
    Mod_init:=@WriteInitModule;
    alloc:=Nil;
    init:=@WriteInit
    end;
  AllocateWriteStrings;
  SEE_module_add(@WriteModule);
end;
```

The `allocateWriteStrings` call allocates some global Javascript strings: these strings are put in global tables, common to all interpreters, using the `SEE_intern_global` call. This is used to register the names of the 'write' and 'writeln' functions, and store them in a couple of global variables:

```
Procedure AllocateWriteStrings;

begin
  GWriteWrite:=SEE_intern_global('write');
  GWriteWriteln:=SEE_intern_global('writeln');
end;
```

The `SEE_intern_global` call can be used until the first interpreter is created. Attempting to call it afterwards will result in an error.

The `WriteInitModule` function initializes the module. For the 'Write' module, it does nothing:

```
Function WriteInitModule : Integer; cdecl;
begin
  Result:=0;
end;
```

More important is the `WriteInit` call, which is called each time an interpreter is initialized. It must register the 'write' and 'writeln' functions with the interpreter:

```
Procedure WriteInit(Interp : PSEE_Interpreter); cdecl;

begin
  createJSFunction(Interp,Interp^.Global,
                   @WriteWrite,GWriteWrite,0);
  createJSFunction(Interp,Interp^.Global,
                   @WriteWriteln,GWriteWriteln,0);
end;
```

As one can see, the `WriteInit` routine calls the `CreateJSFunction`, passing it the first time the `WriteWrite` call with the `GWriteWrite` name. It then does the same for the writeln function. Both times, the interpreter and its 'Global' object are passed along to the function.

The `CreateJSFunction` does the actual work of creating a function that the interpreter understands:

```
Procedure CreateJSFunction(Interp : PSEE_Interpreter;
                           Obj : PSee_Object;
                           Func : TSEE_call_fn_t;
                           AName : PSEE_String;
                           Len : Integer);

var
  V : PSEE_Value;

begin
  v:=new_SEE_value;
  see_set_object(V,see_cfunction_make(interp,Func,AName,len));
  see_object_put(Interp,Obj,AName,v,SEE_ATTR_DEFAULT);
end;
```

The `see_cfunction_make` call creates a javascript function object, which is stored in the `V` value. The `see_object_put` is a libsee function which attaches a property (attribute) to a Javascript object: in this case, it attaches the function object created in the first statement to the object `Obj`. Note that in both cases the same name is used. The `SEE_ATTR_DEFAULT` argument is a pre-defined constant, which is suitable for default attributes.

The function passed to the `see_cfunction_make` must have the following signature:

```
TSEE_call_fn_t =
    procedure (i:PTSEE_interpreter; obj:PTSEE_object;
               thisobj:PTSEE_object;
               argc:Tcint; argv:PPTSEE_value;
               res:PTSEE_value);cdecl;
```

The form of this function follows from the Javascript language. The first parameter is the interpreter instance that has called the function. The second parameter (`obj`) is the actual function object, while the `thisobj` parameter contains the Javascript `this` instance. Since in Javascript, the actual number of arguments to a call need not match the declared number of arguments, the 2 next parameters (`argc` and `argv`) give the actual number of

arguments and a pointer to an array of arguments. The last parameter `res` must be filled with the result of the function, or `SEE_UNDEFINED` if there is no return value (as in a procedure).

Note that since the global object of the interpreter function is used to attach the 'write' and 'writeln' functions to, the effect is that these functions are known as global functions in the interpreter.

The write function support uses the following handler:

```
procedure WriteWrite (i:PTSEE_interpreter; obj:PTSEE_object;
                      thisobj:PTSEE_object;
                      argc:Tcint; argv:PPTSEE_value;
                      res:PTSEE_value);cdecl;
Var
  A,C : Integer;
  t : string;
  v : TSEE_Value;

begin
  if (ArgC=0) then
    SEE_error__throw0(i,I^.RangeError,'Missing argument');
  C:=0;
  For A:=0 to Argc-1 do
    begin
    SEE_ToString(i,argv[a], @v);
    T:=ValueToString(V);
    If Length(T)>0 then
      begin
      Write(T);
      C:=C+Length(T);
      end;
    end;
  SEE_SET_NUMBER(Res,C);
end;
```

The first thing that the function does is checking if an argument was passed. If not, an error exception is thrown. This is done using the `SEE_error_throw0` function, which needs an interpreter, exception object and message text as parameters. The error object used in this case is the standard `RangeError` object from the interpreter: it is available in the `RangeError` field of the interpreter record.

Then a loop is started: each argument is considered, and converted to a `SEE_string` value with the standard `SEE_ToString` function. The resulting value is stored in the variable `V`, which is then passed to the `ValueToString` function, which creates a native Pascal string from a `SEE_string` value. The resulting string is then written to standard output with the regular pascal 'write' function.

During the loop, a counter (`C`) is kept with the number of bytes written. The last statement in the function is to store the number of bytes in the result value: this is done with the function `SEE_SET_NUMBER` (a C macro translated to pascal).

Everything is now set up to use the 'write' or 'writeln' functions in the javascript interpreter. This can be easily demonstrated with the implementation of the 'Hello world' application in Javascript:

```
program testnative;
```

```
{$mode objfpc}
{$H+}

uses
  Classes, libsee, mod_stream;

Var
  interp : TSEE_interpreter;
  ainput : PSEE_INPUT;
  res    : TSEE_Value;

const
   Program_text = 'writeln("Hello, world!");';

begin
  see_init;
  RegisterWriteModule;
  SEE_interpreter_init(@interp);
  ainput :=SEE_input_utf8(@interp, pchar(program_text));
  See_global_eval(@interp,ainput,@res);
  see_input_close(ainput);
end.
```

This program does not look very different from the first Javascript program. The only difference is the `RegisterWriteModule` call at the start, and the different Javascript source pased to the evaluator. Running the program should result in a well-known greeting being printed on the screen.