Real-world applications with Pas2JS

Michaël Van Canneyt

October 28, 2021

Abstract

Pas2js is more than just a toy project: as the underlying compiler of TMS Web Core, it is used to create web applications in a RAD manner, in both Delphi and Lazarus. But it can also be used by itself to create real-world applications. We'll show how in a series of articles.

1 Introduction

People using Delphi and Lazarus are very much used to a RAD approach to programming. TMS Web core brings this approach to web development. But a lot (in fact, most) web development these days is done using technologies such as Angular, VueJS or React: they are very successful development models. Indeed, so much so that React Native has brought web development techniques to the desktop.

These technologies have in common that they do not use RAD techniques: no point-and-click, no 2-way integrated IDE. They only use HTML and Javascript (or transpiled typescript) code.

Such an approach can of course also be achieved with pas2js. The superior Object Pascal editing skills of the Lazarus IDE make creating pascal code a breeze, editing HTML can be done in and outside of the Lazarus IDE.

At this moment, HTML and plain pascal code which interact is the most reliable way to create an Object Pascal application to run in the web when disregarding TMS Web core.

The Lazarus and Free Pascal teams are working on bringing RAD to native HTML, a concept as can be seen in figure 1 on page 2. The purpose of that development is not to create a VCL or LCL for the web, but to create and use a series of components that are as close to the underlying HTML as possible: there will be no position and size properties, instead such things are delegated to CSS. But this technology is in a conceptual stage, and not yet ripe for production development.

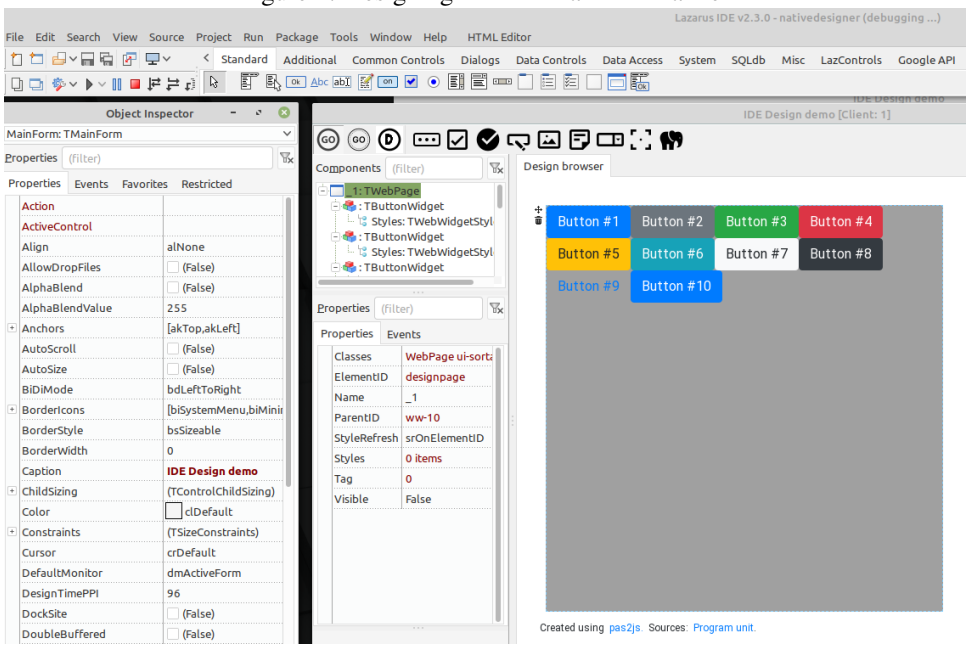
2 A server-side backend

Most Web applications or websites need a back-end: Something to authenticate a user, to fetch data from a database or add data to it. Communication with this backend can be programmed using plain HTTP or WebSockets as a transport mechanism, using several messaging protocols:

SOAP: an older, XML based, protocol.

REST: currently the most used protocol, usually using JSON as the data exchange format. Very suitable for data exchange.

Figure 1: Designing HTML in a RAD manner



JSON RPC: a Remote Procedure call using a more or less standardized JSON format.

A server that exchanges data in one of these ways can be programmed in Lazarus or in Delphi. For example Remobjects SDK can be used for RPC and REST-like programming both with Lazarus and Delphi. It supports many transport frameworks: Windows-specific transport, Indy, Synapse, a fast TCP/IP stack.

Lazarus and Free Pascal have their own Web programming interfaces for RPC and REST mechanisms. There are several:

Brook framework a general web framework built on top of FCL-Web.

mORMot is a complete client server ORM/SOA framework.

Fano Framework is a web application framework.

WST The Web Services Toolkit is aimed at Webservices: SOAP and JSON-RPC messaging protocols are both possible.

In this set of articles we'll concentrate on the frameworks that are delivered with FPC and Lazarus itself:

FCL-Web this is a low-level technology that allows you to write a HTTP-based application server. This can be a FastCGI server, a CGI, an Apache module or a stand-alone HTTP(s) server. On windows the Windows-specific HTTP service can also be used.

fpjsonrpc Implements a JSON RPC mechanism, which can be used to implement a RPC server in a point-and-click fashion.

SQLDBRestBridge Implements a REST layer which can be used to exchange information with a database suppored by FPC.

All three can be accessed easily from within Pas2JS.

Which mechanism or framework is used is largely a matter of preference. What they all have in common is that communication in the browser is asynchronous:

3 GUI considerations

When it comes to programming a GUI in the browser, there are many choices. There are some general-purpose frameworks, which favour a Web Component based approach. Basically, a web component is small HTML template with (possibly custom) HTML tags, and a little piece of Javascript (or typescript) code to control the HTML template based on available data and user interactions. Here are some of the most popular frameworks:

Angular Created by and maintained by google.

React Created by and maintained by Facebook.

Vue.JS An alternative to Angular and Bootstrap, but similar in design philosophy.

Svelte A relatively new kid on the block, which actually compiles HTML and Javascript to some very compact applications.

JQuery one of the first comprehensive Javascript frameworks to manipulate HTML and CSS, it is still used widely.

But many more exist. Although it is theoretically possible to use these frameworks, with the exception of JQuery, these frameworks are not very suitable for work with PasJS: they require extensive tooling and a separate application compile step: the javascript and HTML is combined into the final web app.

The above frameworks deal with creating and altering HTML dynamically. They do not define a style for your components: what HTML and CSS you use is entirely up to you. But there are a huge amount of HTML+CSS frameworks out there which provide visual markup:

Material The Material Design UX specifications of Google have been implemented for Angular (for example Angular-Material) and React (Material Kit React).

Quasar Is a rich set of visual components built on top of Vue.JS.

Bootstrap Originally created by twitter it can be used with JQuery, React and JQuery, or just like that.

Bulma Similar in design to Bootstrap, but without any Javascript.

Semantic-UI Like Bootstrap, it can be used by itself, but it can also be used

JQueryUI a set of UI components for JQuery.

There are many other frameworks out there, and a lot of plain Javascript components that integrate to a bigger or lesser degree with the above frameworks. Any of these can be used with Pas2js.

4 Application Routing

Basically, pas2js allows you to program the browser in Object Pascal. That also means that you do not need a server to generate the pages and to rewrite part of a page when the user clicks something or enters data. Pas2js also does not force you to choose between a multi-page site or a single-page site: you are free to choose this. The advantage of a multi-page site is that there is little or simple logic in each of the pages, so they load fast. The disadvantage is that you need a way to maintain application state between the pages:

For instance, if you want to display the name of the currently active user, it must be stored somewhere and retrieved whenever a new page loads.

A single-page application is more in line with a desktop application: a single html page is loaded, and the logic in the page is responsible for showing different 'forms': these forms can be built dynamically in code, or it can be just parts of the HTML which are shown and hidden as needed. The advantage of this approach is that the application state is maintained while the user navigates from form to form: there is no page reload.

Since this application runs in the browser, the user will expect to be able to use the back and forward buttons to navigate between the various "forms" in your application. To enable this behaviour, a router can be used. Navigation in your application happens with `#hash` links in the page URL: The browser provides a mechanism to be notified of hash changes, and you can use this to show the appropriate form in your application. It is also possible to set the `#hash` part of the URL in code and the browser will add it to the browser history. Thus, when you move programmatically to a new form and the user presses 'back' in the browser history, he will go to the previous form.

Pas2JS has a router component that allows you to work with such a routing scheme, we'll explain how to work with it in one of the following articles.

5 Loading HTML

As written before, when showing forms or pages in the browser, this means changing the HTML in code. Changing the HTML can be done in 2 ways:

- Build the HTML in code using the Javascript DOM classes. Pas2js has a unit called `web` that defines most available classes.
- Load a HTML snippet and insert that in the DOM tree somewhere.

The second way has the advantage that you can design and preview the HTML snippets in the browser. Additionally, ready-to-use HTML snippets can be found all over the web, many of them beautifully styled, so they can be used as-is without the need to tranform them to code that reproduces them.

Because loading HTML in the browser happens asynchronously, it is a good idea to load the HTML snippets in advance. There are several ways to do this; pas2js contains a component `TTemplateLoader` that can load html snippets (in fact, any text file). This component can be used to load HTML snippets in the background:

```
Function FetchTemplate(Const aName,aURL : String) : TJSPromise;
Procedure LoadTemplate(Const aName,aURL : String;
                       aOnSuccess : TTemplateNotifyEvent = Nil;
                       AOnFail : TTemplateErrorNotifyEvent = Nil);
```

The first of these calls returns a `TJSPromise` class, which can be used to wait for the result. The second call uses 2 callbacks: one which is called when the template was loaded, and the second which is called when an error occurs during the loading of the template?

Loading HTML in the background means you must sometimes wait for the snippet to be loaded before you can continue the logic of the application. This means possibly a callback for every snippet you wish to use.

As an alternative mechanism, pas2js also allows you to "link" in the html, much like a form file in Delphi is linked into your application:

```
{$R mydialog.html}
```

Depending on the options given to the pas2js compiler, this will create a HTML file which contains the HTML as a series of `<link />` tags, or a javascript file that contains and registers all the resources as strings with the pas2js runtime.

We'll demonstrate both options in future articles.

Does this mean you should always use HTML snippets when changing the HTML ? No, of course not: sometimes you want to fill a table with data, or a `<select >` tag, and this can just as well be done in code.

6 Getting started

In a series of articles on pas2js we'll develop an application which allows us to demonstrate some of the techniques explained in the above. The application will be developed using `Bulma` as a CSS framework: this is a CSS framework without need for any javascript framework; the programmer is responsible for setting the necessary CSS classes on the html tags. It is well-documented and can be downloaded from the following website:

```
http://bulma.io/
```

So, how to get started with pas2js in Lazarus ? First of all, download the pas2js compiler. The currently latest version can be downloaded from:

```
https://downloads.freepascal.org/fpc/contrib/pas2js/2.0.6/
```

A version exists for Windows, Linux and MacOS. It can be unzipped anywhere on your harddisk.

When pas2js is installed, the first thing to do is install the `pas2jsdsgn` package in the Lazarus IDE. It adds some dialogs and wizards to the Lazarus IDE that help with pas2js applications: in the `Packages - Install/Uninstall packages` menu, the `pas2jsdsgn` package can be selected in the right-hand side list of available packages, as shown in figure 2 on page 6. After selecting the correct package and clicking the `Save and rebuild IDE` button, the IDE will prompt you for confirmation (figure 3 on page 6) When this is done, the IDE will recompile itself and install the new package. The new package will install some items in the `Project - New project` dialog, and creates a page in the `Tools - Options` menu (figure 4 on page 7). This page must be used to tell lazarus where it can find several external tools:

- The pas2js compiler.
- Simpleserver: A small HTTP server to launch when you want to serve the application files. pas2js distributes a modified version of this server, called `compileserver`.
- The TCP/IP port where the HTTP server is supposed to listen.
- Where to find the browser executable you wish to use to run the application.
- Where to find the node.js executable you wish to use to run node.js applications.

The default entries are macros that append the correct extension for your platform, and which assume that the binary is in the `PATH` somewhere.

You can write pas2js source code if you don't set these options correctly, but you won't be able to compile or run the code.

Figure 2: Installing the pas2js support package

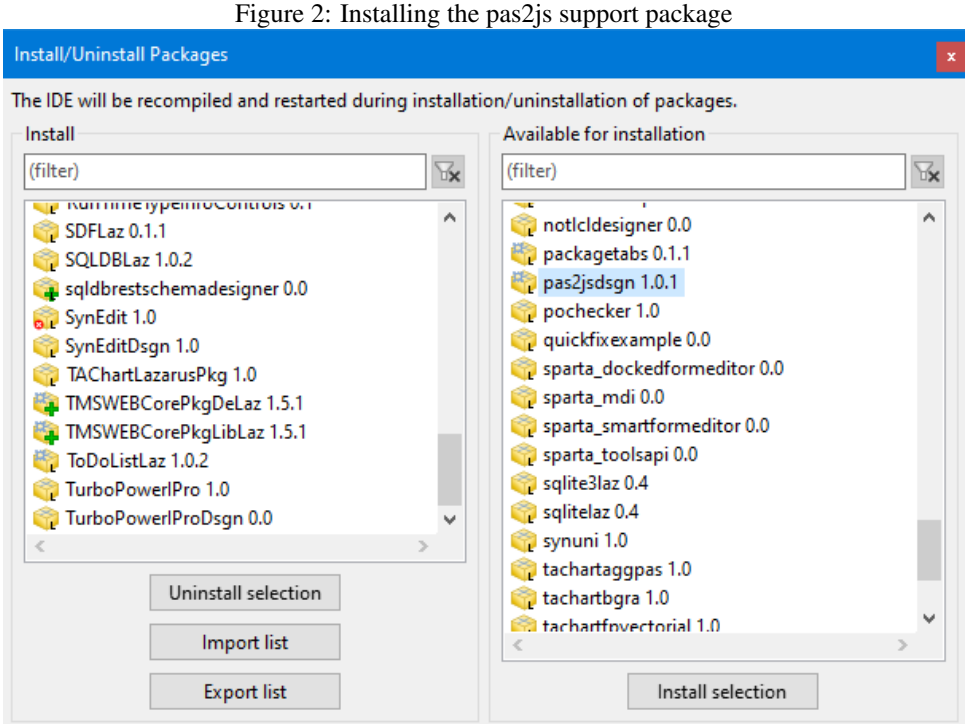


Figure 3: Confirm installation of the pas2js package

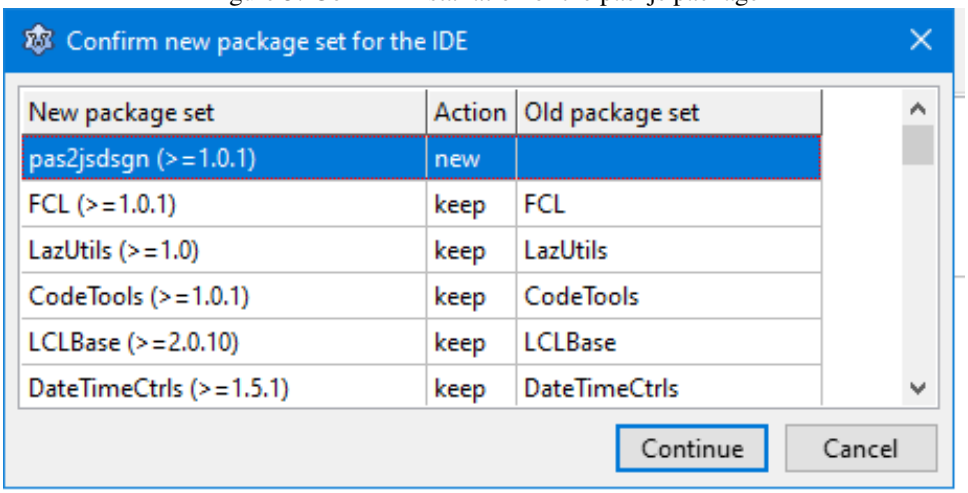
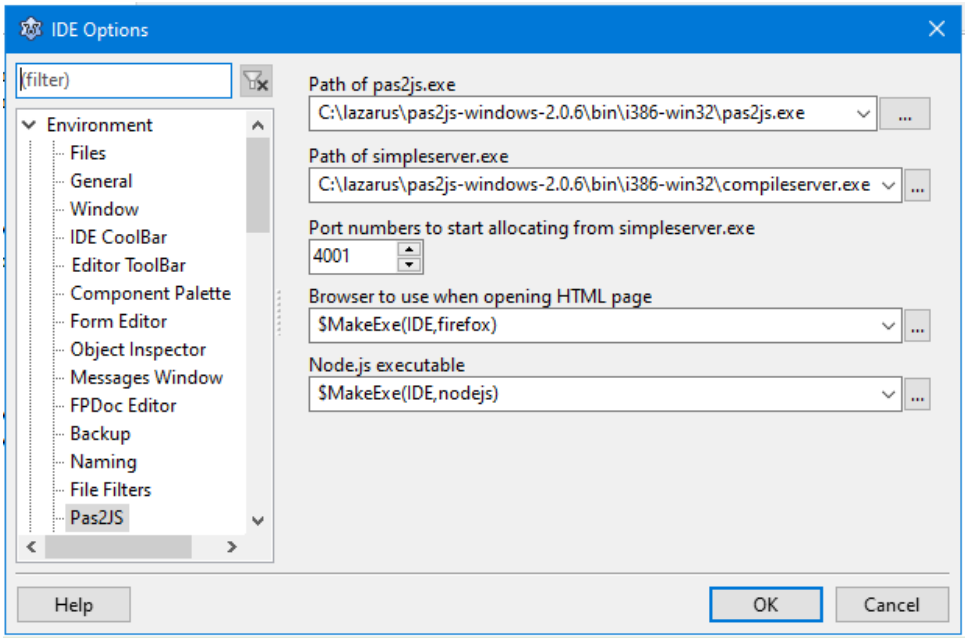


Figure 4: Options for the pas2js package



When the package is installed and configured, the `Project - New project` dialog gets 2 new options - see figure 5 on page 8:

- *Web Browser Application* this starts an application designed to run in the browser.
- *Node.js Application* this starts an application designed to run in Node.js.

From the Lazarus point of view, there is not much difference between these applications. They both generate some boilerplate project code, and for the Web Browser Application an initial HTML page is generated in which the Web Application will be running.

When you choose for the *Web Browser Application* project type, an additional dialog appears which allows you to set some options, see figure 6 on page 9. The options have the following meaning:

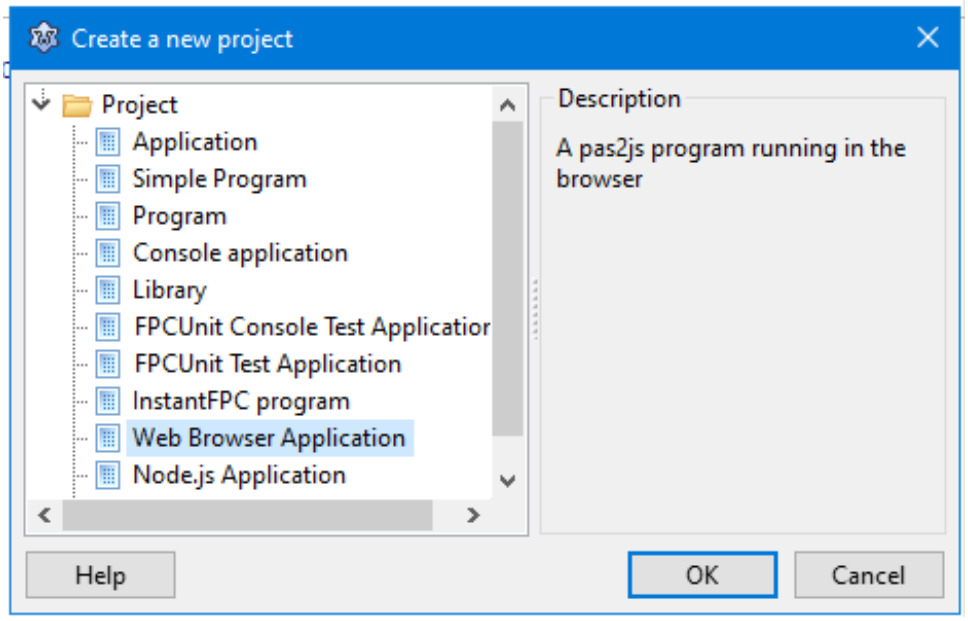
- *Create initial HTML page* A HTML page is created as part of the project. This is the HTML page in which your code will be run.
- *Maintain HTML page* this is an experimental option: the IDE will try to change the project filename when you rename the project.
- *Run rtl when all page resources are fully loaded* This option changes the code in the script tag in the generated HTML. Instead of running the rtl (and hence your program) directly:

  ```
  rtl.run();
  ```

  The RTL is only run when all page resources are loaded by the browser:

  ```
  window.onload=function(){
    rtl.run();
  }
  ```

Figure 5: New project types for developing with pas2js



- *Use browser application object*. When this is unchecked, the project code generated by the IDE is absolutely minimal: an empty program. When checked, the IDE will define an application object for which you must fill the `DoRun` method. This application object resembles the `TApplication` or `TCustomApplication` objects found in regular desktop applications. The class has some methods for finding elements in the HTML and handling the URL query string (which is treated as the environment variables in a regular application).

- *Use browser console unit to display writeln output* By default, the output of `writeln` statements is displayed in the browser debug console. When this option is checked, the `browserconsole` unit is included in the program uses clause. This will cause the output of writeln to be displayed in the HTML page, in a specially styled DIV element, in addition to being displayed in the browser debug console.

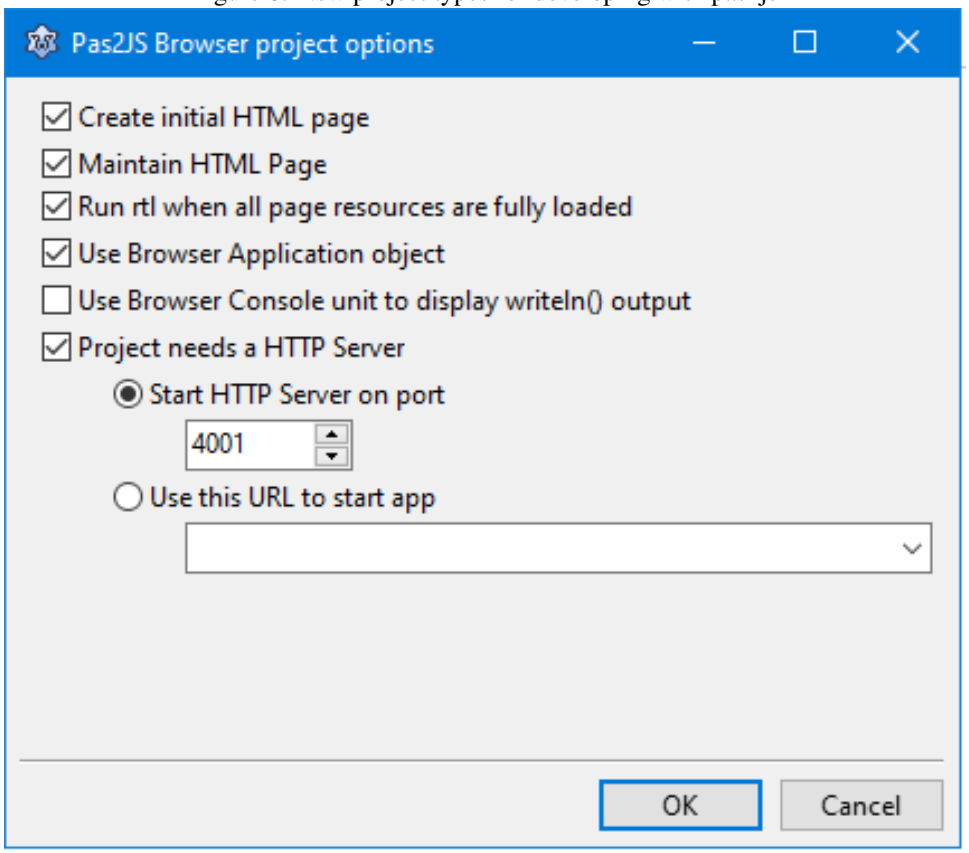
- *Project needs a HTTP server* Some pages can be displayed and run correctly when you open the HTML file on disk by double-clicking it in the file explorer. Pages that load resources in Javascript using `Fetch` or `XMLHTTPRequest` will fail, they need a HTTP server to load the page from. Checking this option offers you 2 choices:

  1. To start a HTTP server on the specified port, and the page can be loaded by pointing the browser to `http://localhost:NNN/yourhtmlfile.html` with `NNN` the indicated port: the IDE will do this when you run the application.
  2. You can enter an URL which will be opened using the system browser when you run the application. You can use this when you have a separate HTTP server (apache, NGinx or somesuch) configured to run your application.

The following project source is generated with the default options and *Use browser application object* set:

```
program Project1;

{$mode objfpc}
```

Figure 6: New project types for developing with pas2js



```
uses
  browserapp, JS, Classes, SysUtils, Web;

type
  TMyApplication = class(TBrowserApplication)
    procedure doRun; override;
  end;

procedure TMyApplication.doRun;

begin
  // Your code here
  Terminate;
end;

var
  Application : TMyApplication;

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

Older versions of Lazarus add a

```
  Application.Free;
```

Statement to this code. This statement must be deleted.

The following HTML page is generated:

```
<!doctype html>
<html lang="en">
<head>
  <meta http-equiv="Content-type" content="text/html; charset=utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Project1</title>
  <script src="project1.js"></script>
</head>
<body>
  <script>
  rtl.run();
  </script>
</body>
</html>
\end{document}
```

The whole HTML can be changed, but 2 elements should remain. The first element includes the generated Javascript program:

```
  <script src="project1.js"></script>
```

The second snippet runs your program:

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

As long as these 2 tags are present, your compiled program will run as soon as the HTML is loaded in the browser.

7 Interacting with the HTML

To show how to interact with the HTML from a pas2js program, we'll program a small Login dialog. The html of the page for a login dialog can be designed as follows:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Login demo</title>
  <link rel="stylesheet"
   href="https://maxcdn.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.min.
  <link rel="stylesheet"
   href="https://fonts.googleapis.com/css?family=Questrial&display=swap">
  <link rel="stylesheet"
   href="https://unpkg.com/bulma@0.9.3/css/bulma.min.css">
  <link rel="stylesheet"
   type="text/css" href="login.css">
  <script src="login.js"></script>
</head>

<body>
  <section class="hero is-success is-fullheight">
    <div class="hero-body">
      <div class="container has-text-centered">
        <div class="column is-4 is-offset-4">
          <h3 class="title has-text-black">Login</h3>
          <div class="box">
            <form action="javascript:void(0)">
              <div class="field">
                <div class="control">
                  <input id="edtEmail"
                         class="input is-large"
                         type="email"
                         placeholder="Your Email"
                         autofocus="">
                </div>
              </div>
              <div class="field">
                <div class="control">
                  <input id="edtPassword"
                         class="input is-large"
                         type="password"
                         placeholder="Your Password">
```

```
              </div>
            </div>
            <button id="btnLogin"
                    class="button is-block is-info is-large is-fullwidth">
              Login <i class="fa fa-sign-in"
                       aria-hidden="true"></i>
            </button>
          </form>
        </div>
      </div>
    </div>
  </div>
</section>
<script>
  rtl.run();
</script>
</body>
</html>
```

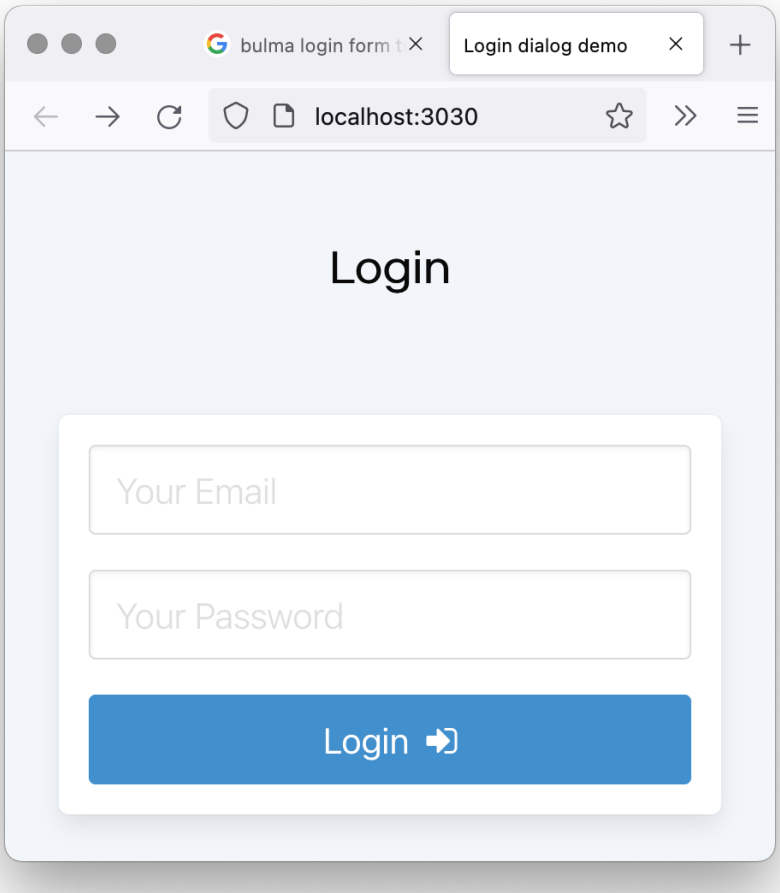
When you open the file in the browser, it will look like figure 7 on page 13:

The important elements in this HTML are the 2 input fields and the button: they have an "id" attribute which can be used to access them from within the pascal code.

There are currently 2 ways to do this with Pas2jS. The first is with plain and simple DOM methods.

```
program Project1;

{$mode objfpc}

uses
  browserapp, JS, Classes, SysUtils, Web;

type

  { TMyApplication }

  TMyApplication = class(TBrowserApplication)
    edtEmail: TJSHTMLInputElement;
    edtPassword : TJSHTMLInputElement;
    btnLogin : TJSHTMLButtonElement;
    procedure doRun; override;
  private
    procedure BindElements;
    procedure doLoginClick(aEvent: TJSEvent);
    procedure doServerLogin(const aUser, aPassword: String);
  end;

procedure TMyApplication.BindElements;
begin
  edtEmail:=TJSHTMLInputElement(GetHTMLElement('edtEmail'));
  edtPassword:=TJSHTMLInputElement(GetHTMLElement('edtPassword'));
  btnLogin:=TJSHTMLButtonElement(GetHTMLElement('btnLogin'));
end;
```

Figure 7: The login page as displayed in the browser



```
procedure TMyApplication.doRun;

begin
  BindElements;
  btnLogin.AddEventListener('click',@DoLoginCLick);
  Terminate;
end;
```

The application declaration contains a variable for every HTML tag that has an ID attribute: 2 edits and one button; the types of the elements are the plain javascript HTML classes that correspond to the used tags (they are defined in the `web` unit that comes with pas2js). The `BindElements` method called from vareuse the `GetHTMLELement` method of the `TBrowserApplication` class to find the element in the HTML page.

To the `btnLogin` element we attach an event handler that will be called when the 'click' event occurs. The `AddEventListener` method is the standard recommended method to attach an event handler to a HTML element. The `DoLoginClick` method declaration can be automatically constructed by the Lazarus IDE if you press CTRL-SHIFT-C when the cursor is on the `DoLoginClick` identifier in the `AddEventListener` method parameter list.

The method uses the values of the email and password edits to call a hypothetical `doServerLogin` method. At this point, that method just shows a message to the user:

```
procedure TMyApplication.doLoginClick(aEvent : TJSEvent);

begin
  DoServerLogin(edtEmail.Value,edtPassword.Value)
end;

procedure TMyApplication.doServerLogin(const aUser,aPassword: String);
begin
  window.alert('Logging in with user: '+aUser);
  // to be implemented.
end;
```

The rest of the code is the boilerplate code the IDE has generated for you when you created the project.

When run, and you click the 'Login' button, the browser will respond with a message, as shown in figure 8 on page 15

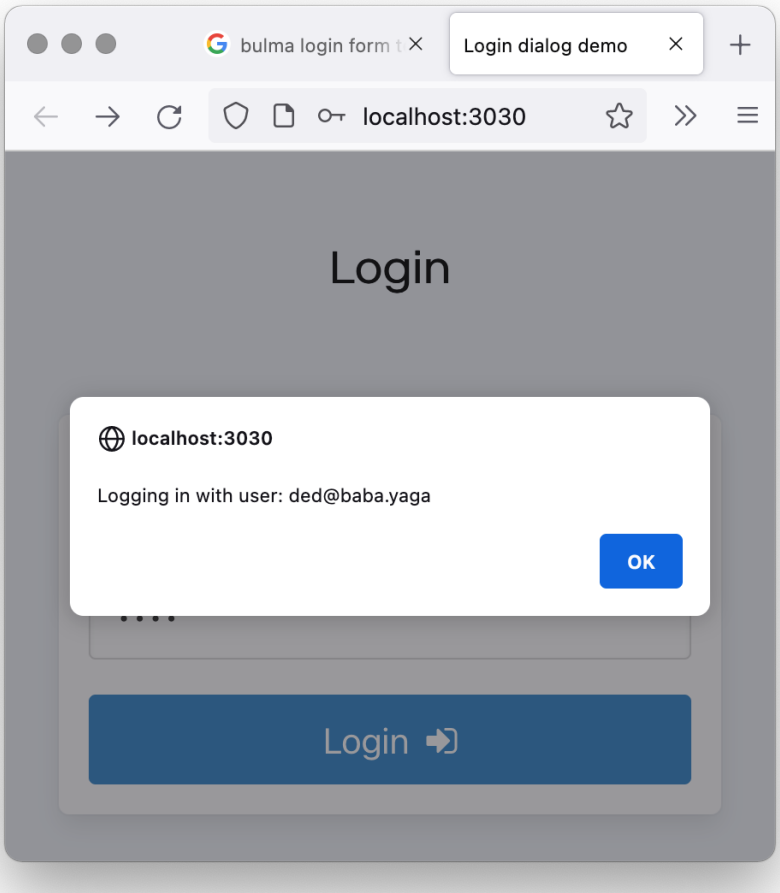
A second way to program the Login dialog is to use the `webwidget` unit. This unit contains a set of components that map to the HTML tags; but they are components, not plain Javascript classes. So the fields in our application class become:

```
edtEmail: TTextInputWidget;
edtPassword : TTextInputWidget;
btnLogin : TButtonWidget;
```

Since we cannot yet design these components visually, they must be created in code. This happens in the `BindElements` call. This code is not different from similar code in a LCL or VCL application:

```
procedure TMyApplication.BindElements;
begin
```

Figure 8: Clicking the login button in the browser



```
  edtEmail:=TTextInputWidget.Create(Self);
  edtEmail.elementID:='edtEmail';
  edtEmail.Refresh;
  edtPassword:=TTextInputWidget.Create(Self);
  edtPassword.ElementID:='edtPassword';
  edtPassword.Refresh;
  btnLogin:=TButtonWidget.Create(Self);
  btnLogin.ElementID:='btnLogin';
  btnLogin.Text:='Login <i class="fa fa-sign-in" aria-hidden="true"></i>';
  btnLogin.TextMode:=tmHTML;
  btnLogin.Refresh;
end;
```

The `ElementID` property tells the component that the HTML tag for this component is already present in the HTML page and has the ID which you assign to the property. The `Refresh` method will either generate the necessary HTML for the component, or links the component to the HTML tag in the page.

For the `Button` widget, the button Text is set, and `TextMode` must be set to `tmHTML`, to allow the button to be correctly rendered: the text must be rendered as HTML, not as-is.

To react on the click, the `OnClick` event handler must be set. It is similar to the one for the plain Javascript classes version of the page:

```
procedure TMyApplication.doLoginClick(sender : TObject; event : TJSEvent);

begin
  DoServerLogin(edtEmail.Value,edtPassword.Value)
end;

procedure TMyApplication.doRun;

begin
  BindElements;
  btnLogin.OnClick:=@DoLoginClick;
end;
```

The page will of course look identical to the page when using plain Javascript classes.

8 Conclusion

Programming a basic GUI in a browser application is not difficult and not very different from programming a LCL application. In the next article, we'll show how to actually implement a login call using a server written in Lazarus.