Design-time components for pas2js

Michaël Van Canneyt

September 1, 2022

Abstract

In this article we present a visual way to use some components that have been introduced in previous articles: While WYSIWYG is not yet possible for HTML applications, it is perfectly possible to use the object inspector in Lazarus for pas2js projects, thus speeding up development of pas2js applications considerably.

1 Introduction

In the previous articles about programming with pas2js, everything was always done in code: with the exception of some wizards to generate or update code, everything had to be done in the source editor. However, Delphi and Lazarus are RAD environments. There is no intrinsic reason why RAD development should not be possible for pas2js applications.

The RAD mechanism in Delphi and Lazarus is based on a streaming mechanism: a form file contains definitions for all components placed on a form or datamodule. At runtime, the components are created by the streaming mechanism, and the published properties which you manipulate using the object inspector are set from the values found in the .dfm or .lfm file .

The streaming mechanism exists in pas2js as well. That makes it possible to create a form file in the IDE and use it to recreate at runtime - in the Javascript environment - the form definition that you created in the IDE. Indeed, all classes presented in the previous articles - in particular, the widgets - have been created as descendents of `TComponent`, with the intent that one day they would usable in the Object Inspector of the IDE.

There is only one problem: The components in the pas2js world assume a browser and do their work with the browser classes. So how can we compile these components for installation in the IDE, where the browser environment is not available ?

At this moment, this is not directly possible. But, a trick can be used that will allow us to manipulate components meant for pas2js in the IDE.

2 Stub components

In order to use the object inspector for a component, the IDE only needs the published properties of a component. It does not need (all) the actual functionality of the components to be able to display them in the object inspector.

Keeping this in mind, it is possible to create a 'stub' component: this is a component (a native class) that has the same published properties as the pas2js class, and enough code to set the various properties and make them behave as they would at runtime: if changing the value of a property changes another property, then the stub code should reflect this.

The stub component can be created in at least 2 ways:

1. By taking the actual class and put all code that somehow refers to the browser in a conditional define:

   ```
   Procedure TMyLabel.SetCaption(aCaption : string);
   begin
     FCaption:=aCaption;
   {$IFDEF PAS2JS}
     // FElement is a TJSHTMLElement instance
     if Assigned(FElement) then
       FElement.InnerText:=FCaption;
   {$ENDIF}
   end;
   ```

   This approach has the advantage that only 1 set of source files is needed: one for in the browser, one for in the IDE. The disadvantage is that it is a lot of work to keep the code compilable on both platforms.

2. By taking a copy of the class and removing all code that is not directly involved in the handling of the published property: all public, private and protected methods can be removed. Obviously, properties that descend from `TPersistent` must still be created. The advantage is that the code is simple, the disadvantage is that there are 2 sets of files to be maintained.

Pas2js comes with a tool (makestub) which helps creating such a stub class using the second method: it scans the original file for classes and outputs a new class with only the published properties. It is not perfect, but can be used to make a first version of a stub.

Once the stub component is made, the component can be installed in the IDE, property and component editors can be registered and it can be dropped on a datamodule or a form.

Currently, the trunk version of the lazarus IDE comes with 15 components that were made this way. They will be presented in the below.

3 Data Modules and Forms

A data module is a container for components that is not visible: it can be created like a form, components can be dropped on it, but at runtime it exists only in memory, it has no visible counterpart. It should not come as a surprise then that using a `TDatamodule` is possible in pas2js: you can create a datamodule in the IDE, drop some components on it and compile it. If the components are supported by Pas2JS (or have a stub), the data module can be created at runtime, the form file will be read, and all will be as in a native application.

But how to do forms ?
Forms are deeply embedded in the VCL or LCL, and there is no HTML counterpart for this concept. Indeed, what would constitute a form in the browser ? The

complete browser HTML window area or just a part of it ? Various answers are possible.

It would be possible to mimic `TForm` and most of its properties in the browser, but this would be enforcing an alien concept on the browser:

Visually, the browser is simply a powerful engine to display HTML in a window. Trying to hide the use of HTML in LCL-like components is maybe not the best idea: rather, the power of HTML and CSS should be embraced. The internet is full of beautiful frameworks, many components that render HTML directly. It makes sense to allow us to be able to use of this rich ecosystem.

So, how can we visually represent a part of the browser window in the IDE as if it were a form and allow the user to use the object inspector and component palette?

The answer is the `THTMLFragment` "form" component. In the IDE, this is currently simply a `TDataModule` descendant, so you can create it with the 'File-New' menu; and drop components on it and manipulate them in the IDE: it has a form file as a datamodule or form, so it can be streamed.

You will notice the familiar resource directive in the source code:

```
{$R *.lfm}
```

Pas2js will recognize this directive and will link in the resource (more about that later in this article)

With the `THTMLFragment` component, you can associate a fragment of HTML. At runtime, this HTML fragment will be inserted in the browser DOM tree, at a location that you specify: much like the 'form' classes presented in the previous articles.

At the same time a component has been developed that allows to associate HTML elements to a pascal identifier, and attach event handlers for most common HTML events to this tag in a visual manner using the object inspector. This component is called the `THTMLElementActionList` and can be found on the component palette. Its workings are analogous to the `TActionList` class found in the LCL and VCL.

4 The `pas2jscomponents` package.

The concepts presented in the previous sections have been implemented in a package `pas2jscomponents`. The package is available in the trunk version of Lazarus in the directory `Components/Pas2js/Components`. When installed in the IDE, (currently) 15 components are installed on the component palette on 4 different tabs:

Pas2js Some general purpose components, plus some Bootstrap CSS based UI-omponents.

The components are (in order of appearance on the component palette tab):

THTMLElementActionList a component to access the HTML tags and associate events handlers to various events in a visual manner.

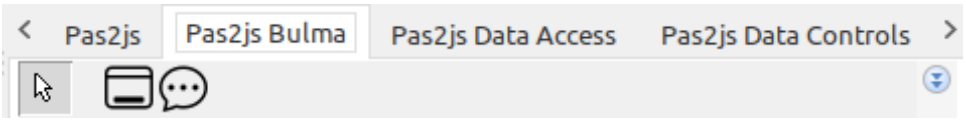
TBootstrapModal a bootstrap modal dialog component.

TBootstrapToastWidget a bootstrap toast message component.

TPas2JSRPCClient The RPC Client presented in the previous pas2js article. The name is changed to prevent a clash with the native `TRPCClient` class.

THTMLEditor a component for a simple HTML editor.

Pas2js Bulma A dialog and toast component based on Bulma CSS

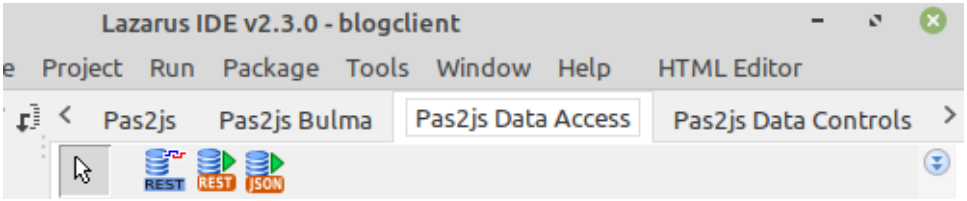


2 components based on Bulma CSS:

TBulmaModal a Bulma-based modal dialog component.

TBulmaToastWidget a Bulma-based toast message component.

Pas2js Data access Components for data access.



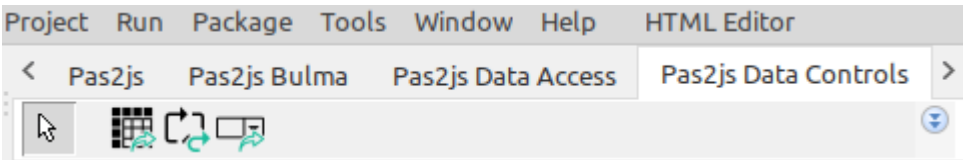
This tab contains 3 components for data access:

TSQLDBRESTConnection The connection component for a SQLDBRest server, presented in the previous article.

TSQLDBRESTDataset The dataset component for a SQLDBRest server, presented in the previous article.

TLocalJSONDataset A local dataset component: you can use this for storing local data, by defining a set of fields, like in `TClientDataset` in Delphi, or `TBufDataset` in Lazarus. The component can load and save the (JSON) data from and to the browser's local storage.

Pas2js Data Controls These controls are data-aware: they generate HTML based on the contents of a dataset.



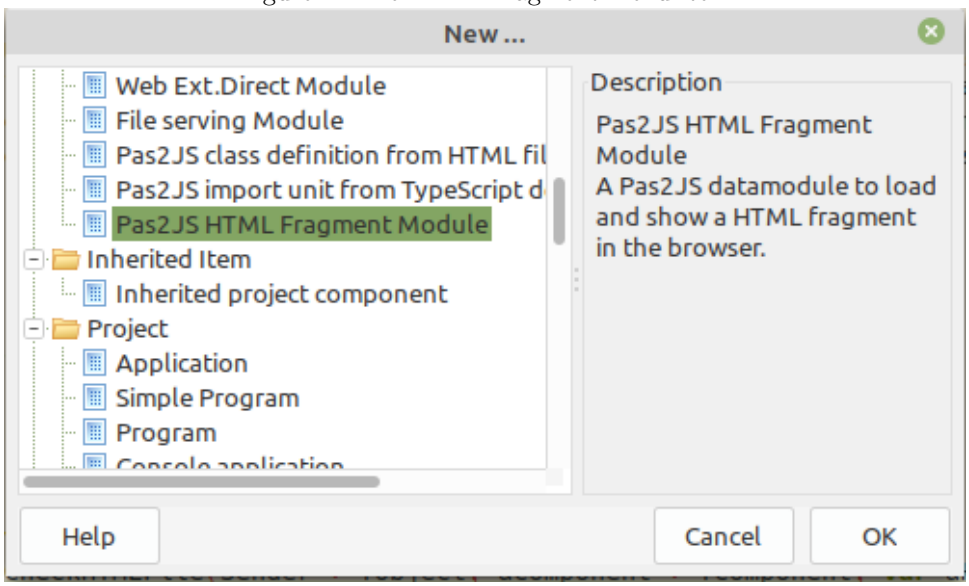
The components are (in order of appearance on the component palette tab):

TDBBootstrapTableWidget Fill a grid based on a dataset. It uses bootstrap-table for the styling and handling.

TDBLoopTemplateWidget Repeat a HTML snippet for every record in a dataset, replacing template variables based on the field values in the dataset.

TDBSelectWidget Fill a HTML SELECT tag with OPTION tags, based on records in a dataset.

Figure 1: The HTMLFragment menu item



In addition to the components, the `pas2jscomponents` package registers the `THTMLFragment` as an item in the 'File' - 'New' dialog, as shown in figure 1 on page 5.

5 A demo application

To demonstrate the use of these components and the visual HTML, we'll rewrite the application we created in the previous article using the HTMLFragment and the visual components. For some components, nothing changes: instead of creating them in code and setting the properties, this can now simply be done visually. For some others, some changes are needed.

As a quick reminder, the application was a small blogging application. The server part has an RPC mechanism to log in, the client has several forms:

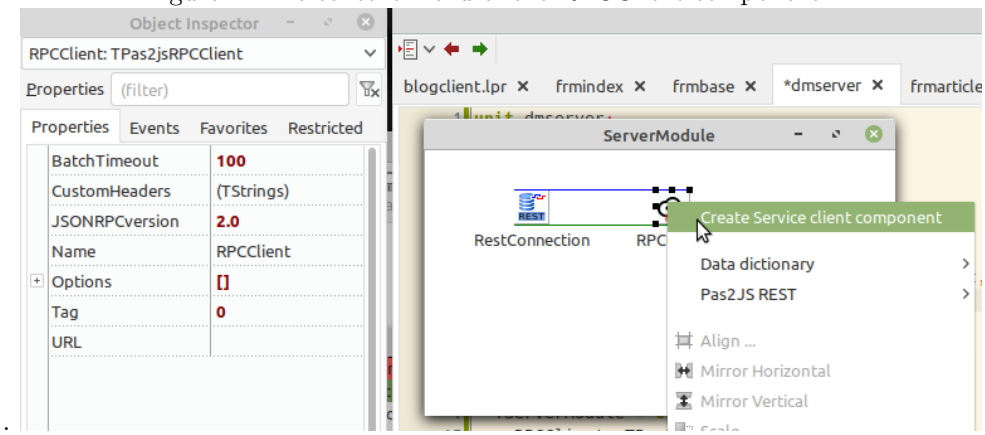
1. A data module which has a RPC client and the SQLDB Rest connection.
2. A main form to handle the main HTML page (index.html)
3. A form to log in.
4. A form to show an overview of available posts
5. A form to view or edit a post.

The server part does need one change: the `sqldbrestcds` unit must be added to the uses clause. The reason for this is that the design-time support for various components queries the server and expects data to be returned in the format read by `TBufDataset` format. Adding the `sqldbrestcds` unit to the server enables the support of this format.

So the server program can be compiled as it was (with the added `sqldbrestcds` unit), and started - this is needed to enable some of the data handling functionalities described below.

The client part needs more changes, obviously.

Figure 2: The context menu of the RPCClient component



To get started, you need of course to have the latest trunk version of lazarus (see the article that explains how to do this) and the package *pas2jscomponents* needs to be installed.

Just as in the original application, we again start with creating a new pas2js browser application, and we set the option to use the `TBrowserApplication` class. After this, we use the 'File - new' dialog to create a data module: we will name it `ServerModule` and save the unit with the name `dmServer`.

On this data module, we drop 2 components from the `Pas2js` and `Pas2js Data access` tab:

1. a `TPas2JSRPCClient` component. We call it `RPCClient` set the `URL` property to

   ```
   http://localhost:8080/RPC
   ```

   if you used another port than 8080 for the server application, obviously you must set the `URL` property accordingly.

2. a `TSQLDBRestConnection` component. We call it `RestConnection` and set the `BaseURL` property to

   ```
   http://localhost:8080/REST
   ```

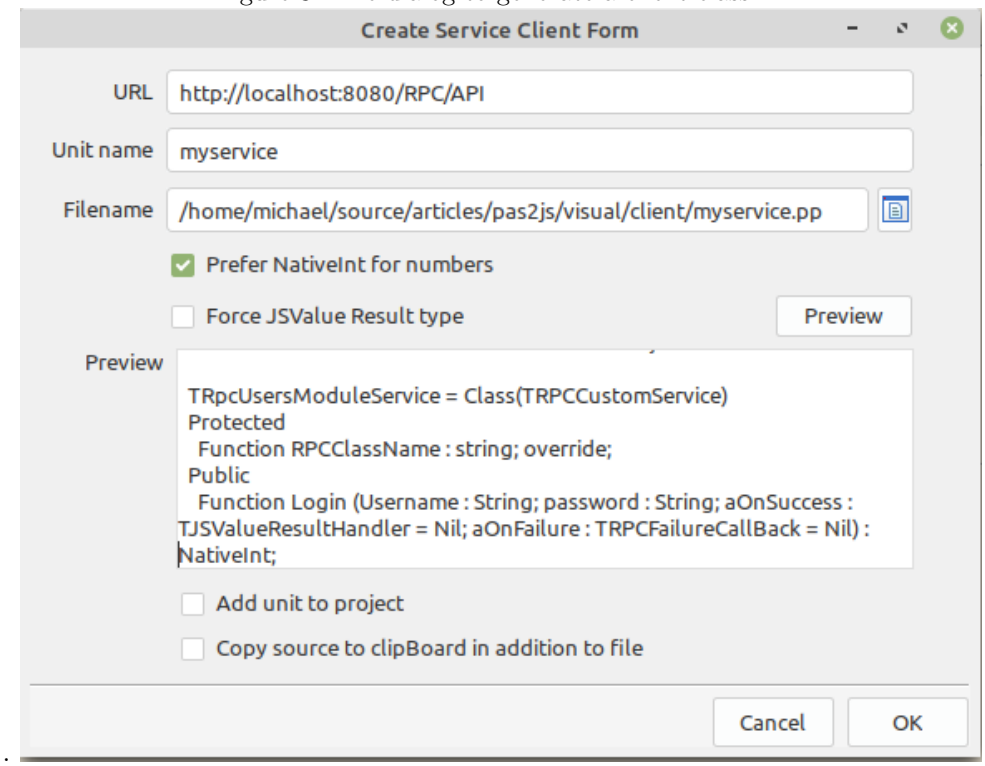
   Here as well, the port number must be changed if necessary.

The IDE support for `TPas2JSRPCClient` contains a wizard to generate the service definition: in the context menu of the `RPCClient` (see figure 2 on page 6), select 'Create Service Client component'. This will bring up the 'Service Client generation' dialog.

This dialog does the same as the `apiclient` website page presented in the second article in this series: based on the RPC service URL, it creates a unit with a client-side proxy class for the service that has all the calls presented by the service. It has some options, most of which are obvious in what they do:

1. The URL of the service, this is taken from the `RPCClient` component.
2. The name of the unit to generate.

Figure 3: The dialog to generate a client class



3. The file name of the unit to generate.
4. Options which influence the code: use `NativeInt` for number-typed arguments and an option to force the use `JSValue` for result values in callbacks.
5. A button to show a preview of the code in a memo.
6. Options to add the unit to the project, and copy the code to the clipboard. The latter is useful if you want to have multiple services in 1 unit: you can simply paste the declaration into an already existing unit.

All these options can be seen in figure 3 on page 7.

The class can be generated and renamed (`TUserService`) to match the class name used in the previous version of the project. In the data module, we define again a variable for the service, and add the login method with 2 events to signal success and failure, just as in the previous version of this program. The `OnCreate` event of the data module can be used to configure the connection at runtime:

```
Const
  ServerURL = 'http://localhost:8080/';

procedure TServerModule.DataModuleCreate(Sender: TObject);

begin
  FUserService:=TUserService.Create(Self);
  FUserService.RPCClient:=RPCClient;
  RPCClient.URL:=ServerURL+'RPC/';
  RestConnection.BaseURL:=ServerURL+'REST/';
```

```
end;
```

The login method remains exactly as it was in the first version of our application: we can simply copy and paste it: no code changes are necessary.

```
  procedure TServerModule.DoLogin(const aUserName, aPassword: String);

  procedure DoOK(aResult: NativeInt);
  begin
    FUserInfo.Login:=aUserName;
    FUserInfo.ID:=aResult;
    if Assigned(FOnLogin) then
      FonLogin(Self);
  end;

  procedure DoFail(Sender: TObject; const aError: TRPCError);
  begin
    FUserInfo.Login:='';
    FUserInfo.ID:=-1;
    If Assigned(FOnInvalidLogin) then
      FOnInvalidLogin(Self);
  end;

begin
  UserService.Login(aUserName,aPassword,@DoOK,@DoFail);
end;
```

6 Accessing HTML elements

In the previous articles we used the lazarus IDE wizard to generate a "form" class definition from a HTML page, where the form would have a field for every HTML element in the HTML page. We could do the same here and tell the wizard to use the `THTMLFragment` class as a parent.

Doing so leads to a small problem: the wizard does not generate a form file, a problem which still needs to be solved.

But there is another way: we can do this differently and in a manner which will allow us to work in visual (point-and-click) manner for creating the HTML element events. We will show this by starting with the main html file (`index.html`).

As a reminder: The index HTML file serves as the shell for the various forms in the application, and contains the application menu as well as the login menu and hamburger menu for responsive design. It also contains the parent HTML tag where the various pages of the Single Page Application are shown. These HTML tags are accessed from the login page to enable/diasble the menu and login menu items. So we make a 'form' for this page.

The 'File' - 'New' menu item "Pas2JS HTML fragment module" can be used to create a form-like enviromnent. The `THTMLFragment` class corresponds to a 'form': a fragment of HTML that will be downloaded and inserted in the main html page.

This form (we'll call it "form" for ease of use) has several properties.

HTMLFileName if set, this is the HTML file that will be downloaded by the template loader. If not set, the name of the file is the `TemplateName`, in

lowercase and with extension `.html`

ParentID The ID of the parent HTML element for this form. This is the element below which the HTML for this form will be inserted.

TemplateName if set, this is the name of a global template that will be used as the HTML form.

UseProjectHTMLFile If set, no extra HTML will be loaded, and the main HTML file of the project will be used to find IDs for tags etc.

There are some events:

OnAllowUnrender This will be called before unrendering (removing) the form's HTML, and can be used to disallow unrendering: in case there is data to be saved.

OnRendered This event will be called when the form HTML is inserted in the parent HTML tag. At this moment, the HTML is inserted in the DOM.

OnHTMLoaded This event will be called when the form HTML has been downloaded but is not yet inserted in the parent HTML tag.

OnUnRendered This event will be called when the form HTML has been removed from the browser's DOM tree.

It has few methods:

Render Render the HTML by inserting it in the DOM tree. This method is synchronous and can only be called if the HTML for the form has already been downloaded.

UnRender Remove the form's HTML by removing it from the DOM tree. This method is synchronous.

Show Download and subsequently render the HTML by inserting it in the DOM tree. This method can be asynchronous and can be called even if the HTML for the form is not yet available: it will use the `HTMLFileName` and `TemplateName` to use the global templateloader (presented in an earlier article in this series) and download the necessary HTML.

Hide Will check if the form is rendered and if so, unrenders it.

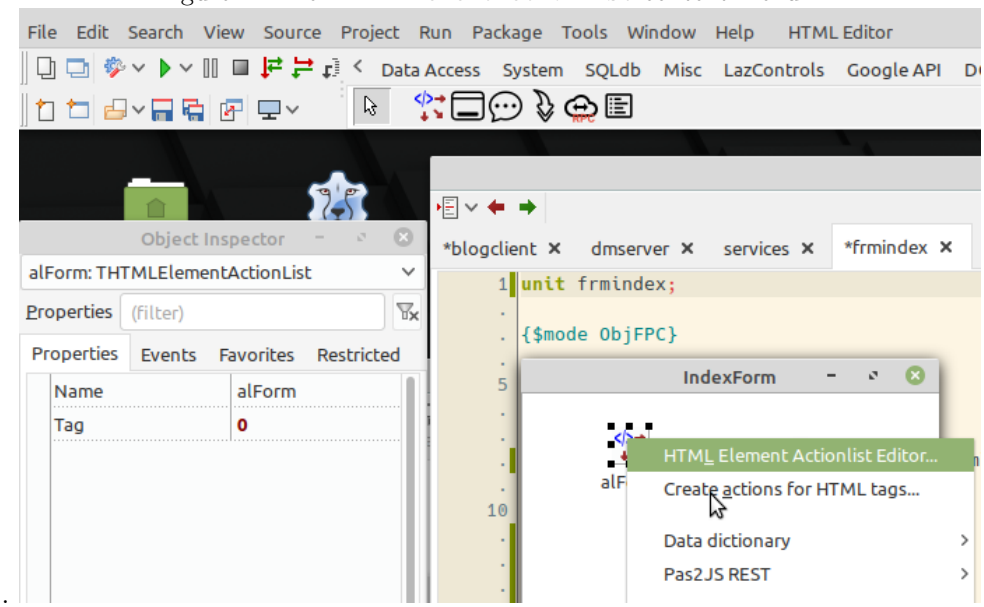
The reason that there are so little properties and methods is simple: There is not much else to do in the concept of a HTML Fragment. Designing the form is expected to happen in a HTML editor (you can use the Lazarus IDE for this). For capturing events, we can use another component described below.

Once the THTMLFragment is created, we will name it `TIndexForm` and call the unit `frmIndex`. Then we can drop a `THTMLElementActionList` component on it from the 'Pas2js' tab of the component palette.

This component is modeled after the `TActionList` component: it has a component editor that allows to create items for the various HTML elements in the form's HTML. When you display the context menu of this component, there will be 2 menu items, as shown in figure 4 on page 10:

HTML Element Actionlist Editor this will show the element actionlist editor, a component editor that closely resembles the `TActionList` editor found in the LCL.

Figure 4: The `THTMLElementActionList` context menu.



Create actions for HTML tags this will scan the HTML file for HTML element tags that have their `id` attribute set, and will create an action for each of them.

There are various action classes possible: a `THTMLElementAction` is a `TComponent`, just like `TAction` in the LCL. As such, it will be inserted in the form and hence is available in code, you can do code completion on it etc.

When you select the `Create actions for HTML tags` menu item, a dialog will pop up that shows the ids of HTML tags for which no `THTMLElementAction` definition exists yet. The dialog is shown in figure 5 on page 11.

As can be seen in the editor, you can set an option to use Data-aware actions: setting this option this will instruct the wizard to use data aware actions for all input elements and buttons. You can select the action class to use for each element. Once done, the necessary component definitions will be inserted in the form. The wizard will choose default names for the various actions: `'act'` appended with the id attribute of the tag (suitably modified so it is a valid pascal identifier).

The `THTMLElementAction` class has the following published properties:

Events A set of events for which you want to create an event handler. One event handler is created that handles all events.

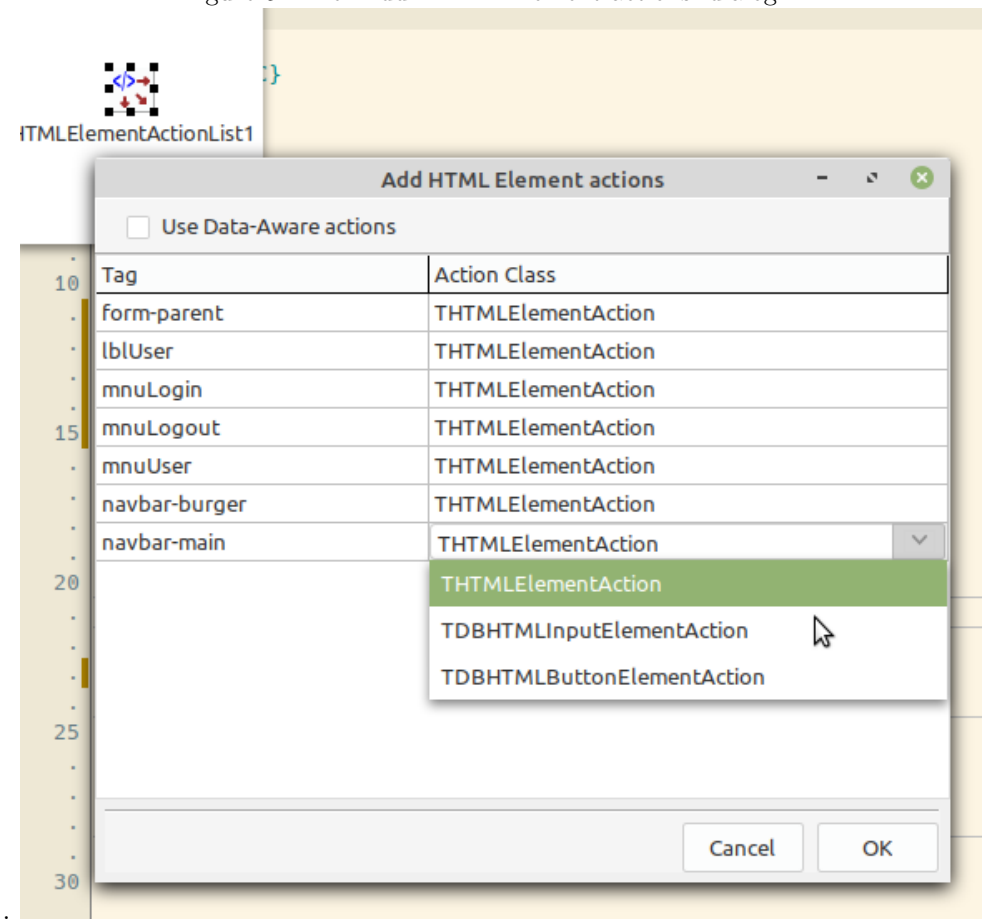
CustomEvents The names of custom events, not present in the standard events list. These event names must be separated by commas or spaces.

ElementID The 'id' attribute of the HTML tag to which to bind this action. This will result in a single HTML tag being bound.

CSSSelector a CSS selector: this will be used using `querySelectorAll` to bind one or more HTML tags to this action.

PreventDefault when set to `True`, the `preventDefault` method of the event object will be called after the `OnExecute` event was executed.

Figure 5: The 'Add HTML Element actions' dialog.



StopPropagation when set to `True`, the `stopPropagation` method of the event object will be called after the `OnExecute` event was executed.

OnExecute This event handler is executed for all selected events. You can see which event was executed by examining the `event` parameter.

BeforeBind This event is executed just before the action is bound to the HTML tags.

AfterBind This event is executed just after the action is bound to the HTML tags.

Note that only a single event handler is created for all possible HTML events. It is possible to create 2 actions for the same HTML tag, and handle different events in each action. Besides the published properties, there are also some public properties which can be used in code:

ActionList The action list of which the action is part.

Element In case an was specified, the HTML element (of type `TJSHTMLElement`) to which the action is bound.

Elements An array of HTML Elements corresponding to the CSS Selector. In case an element ID was specified, the array will contain 1 element, corresponding to the `Element` property;

Index The index of this action in the `ActionList`

Value When the property is read, this returns the value of the first element in the `Elements` array. In case of writing to the property, the value will be set for *all* elements in the array.

`Value` can be set on any HTML element: For `Input` and `Select` HTML tags, `Value` is the value property of the tag. For all other tags, this is the value of the `InnerHtml` property.

It is possible to manipulate the HTML tag using the `Element` property. For ease of use, there are some methods in the action:

Bind Call this to bind the action again to the HTML tags: This is called automatically when the form is rendered, but you can call it again after the HTML inside the form has dynamically changed. For instance, it can be used to bind the onclick event to all rows of a table using the selector '`#mytable tr`'. If the table was filled with rows dynamically, you would call `Bind` to get the array of row elements.

ClearValue Clear the value of the element.

FocusControl Focus the element.

BindEvents Bind all events for a given element.

HandleEvent Handle an event and call the event handlers

ForEach execute a callback for each element in the array of elements for this action. Optionally a Data object can be specified which is passed to the callback.

AddClass add a class to the list of CSS classnames for the HTML tag(s). The argument must be a single CSS class name.

RemoveClass remove a class from the list of CSS classnames for the HTML tag(s). The argument must be a single CSS class name.

ToggleClass toggle a class in the list of CSS classnames for the HTML tag(s). The argument must be a single CSS class name.

Descendents of `THTMLElementAction` may have more properties and methods.

These action classes can be used to interact with the HTML; just like for classical controls, the events can be created in a point-and-click manner. If the form HTML has a toplevel tag with an ID attribute, events can be handled for the whole form as well: In that case you may need to set the `StopPropagation` to `True` on individual elements to prevent the event from being handled twice.

For the index form, the actions can be created, but no events are needed: the events will be handled in the login form. The menu is initially disabled, so we do add a method to show the menu when the user logs in. This method (`StartLogin`) demonstrates some of the methods and properties of the action class:

```
procedure TIndexForm.StartLogin(const aUserName: String);
begin
  mnuUser.RemoveClass('is-hidden');
  mnuLogout.removeClass('is-hidden');
  mnuLogin.AddClass('is-hidden');
  lblUser.Value:=aUserName;
end;
```

For the `login` form, we do the exact same as for the index form. But in this case, we fill in the `TemplateName` and `HTMLFileName`.

The context menu for the `THTMLElementActionList` and in particular the 'Create actions for HTML tags' menu item will know in what file to look for the HTML tags and their ID attributes: the wizard will create actions for the login button and the input tags for username and password.

Only the login button needs an 'onclick' event handler:

```
procedure TLoginForm.actbtnDoLoginExecute(Sender: TObject;
                                          Event: TJSEvent);
begin
  server.DoLogin(edtEmail.Value,edtPassword.value);
end;
```

The server module still has the `OnLogin` and `OnInvalidLogin` events, which we assign in the `OnCreate` event of the login form:

```
procedure TLoginForm.DataModuleCreate(Sender: TObject);
begin
  Server.OnInvalidLogin:=@DoLoginFailed;
  Server.OnLogin:=@DoLoginOK;
end;
```

7 Toast components

In the `OnLogin` or `OnInvalidLogin` event handlers, a Bulma toast was shown. The `Pas2js` tab of the component palette contains a `TBootstrapToastWidget`, and the

`Pas2js Bulma` tab contains a `TBulmaToastWidget`. These components can be used to show a toast message. The bulma toast has the following properties:

Header A string with the HTML for the toast's header.

Body A string with the HTML for the toast's body.

HeaderImage An URL used to add an image to the header.

CloseButton A boolean indicating whether to show a close button or not.

Contextual Set the color scheme for the toast: The `TContextual` type is an enumerated with the primarys colors used in Bulma: link, warning, danger etc.

HideDelay The delay - in milliseconds - to wait before hiding the toast. Only effective when `AutoHide` is `True`.

AutoHide A boolean to indicate whether the toast must hide automatically after some delay.

Animate A boolean to indicate whether to use an animation to show or hide the toast.

MinWidth An integer indicating the minimal width of the toast.

Single single toast or multiple toasts ?

Position The position of the toast on the browser form: An enumerated that can be set to top-left, bottom-right etc.

ParentID The ID of the HTML Element where the toasts will be rendered. If none is set, the ParentID of the toast manager (toasts) will be used, if that is not set, the body element is used.

The Bootstrap toast component has the same properties.

So, to show a toast on succesful login, we drop 2 bulma toasts on the form, enter some text in the `Body` and `Header` properties and set the parameters for `Contextual`, autohide. The resulting form in the designer looks as in figure 6 on page 15.

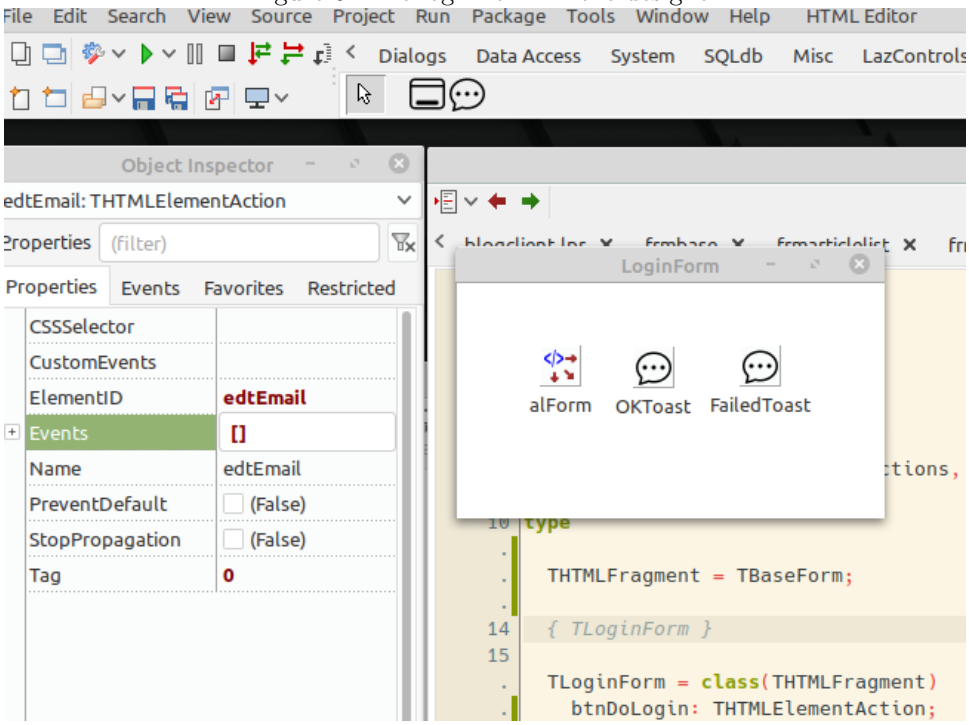
All that's left to do is to call `Refresh` in the appropriate places, which will actually render the toast:

```
procedure TLoginForm.DoLoginFailed(Sender: TObject);
begin
  FailedToast.Refresh;
end;

procedure TLoginForm.DoLoginOK(Sender: TObject);
begin
  OKToast.Body:='Hello, '+Server.UserInfo.Login;
  OKToast.Refresh;
  IndexForm.StartLogin(Server.UserInfo.Login);
  Router.RouteRequest('/articles',True);
end;
```

As you can see, at the end of the `DoLoginOK` we have the same code as in the previous version of the application: a call to route the brower to the articles form.

Figure 6: The login form in the designer



The form manager class we introduced in the article on routing remains as it was, but there is a minor change: the `TBaseForm` form that underpins all forms is now a descendent of the `THTMLFragment` class.

We just need to make sure that our forms are descendents of `TBaseForm`. Ideally, we could create a Lazarus IDE wizard to create a descendent of `TBaseForm`, but a simpler method can also be used.

We simply create an alias for the `THTMLFragment` class:

```
uses
  SysUtils, Rtl.HTMLActions, htmlfragment, Web,
  bulmawidgets, frmBase;

type
  THTMLFragment = TBaseForm;

  TLoginForm = class(THTMLFragment)
  // etc.
```

Using this method, all the code for displaying forms as it was presented in the article on routing remains functional, but behind the scenes the `THTMLFragment` class is used, which allows us to remove the code to actually download the html from the `TFormManager` class, as this part is now handled by the `THTMLFragment` class.

8 Using Data-aware components

The article list form used a dataset to fetch a list of articles, which were subsequently shown with the `TDBLoopTemplateWidget` component class. Since the tem-

Figure 7: The `TSQLDBRestDataset` component context menu

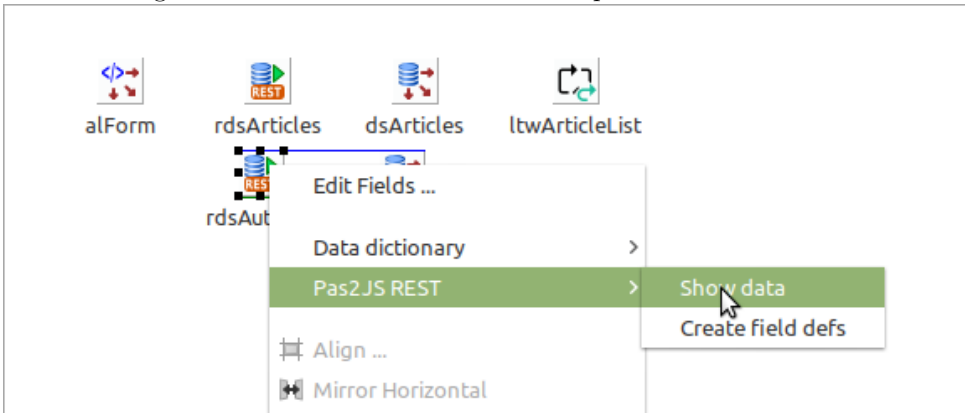


plate widget is a component, it is available on the component palette. The same can be said for the `TDBSelectWidget` that is used to display a select element to filter the authors. Recreating the article list form with these components is easy to do:

- We create a pas2js HTMLFragment form and set the `HTMLFilename` property to articles.html.
- We create a `THTMLElementActionList` on the form and let it create actions for all elements.
- We drop 2 datasets (for articles and authors) on the form and connect them to the connection component on the server datamodule.
- We drop 2 datasources (`dsArticles` and `dsAuthors`) and connect them to the datasets.
- We drop a `TDBLoopTemplateWidget` component (`ltwArticlesList`) and connect it to the `dsArticles` datasource.
- We drop a `TDBSelectWidget` component (`dswAuthors`) and connect it to the `dsAuthors` datasource.

The `TSQLDBRestDataset` component has a context menu with a `Pas2JS REST` submenu in it (figure 7 on page 16). This submenu has 2 entries:

Create field defs Using this menu you can create fielddefs for the REST resource indicated in `ResourceName`. After this, the fields editor can then also be used to create persistent fields for the dataset.

Show data Using this menu you can actually see the data of the REST resource in a grid, as shown in figure 8 on page 17.

Note that the SQLDBRest server must be running for these menu items to work.

The various template properties of the `TDBSelectWidget` and `TDBLoopTemplateWidget` component can now be edited using the Object Inspector: a property editor that shows a HTML editor with syntax highlighting is available for editing the HTML snippets, as shown in figure 9 on page 17.

The author selection dropdown (`dswAuthors`) contains a `ItemField` property which determines the field to show in the dropdown. It is set to `u_author`.

Figure 8: Showing the REST resource data

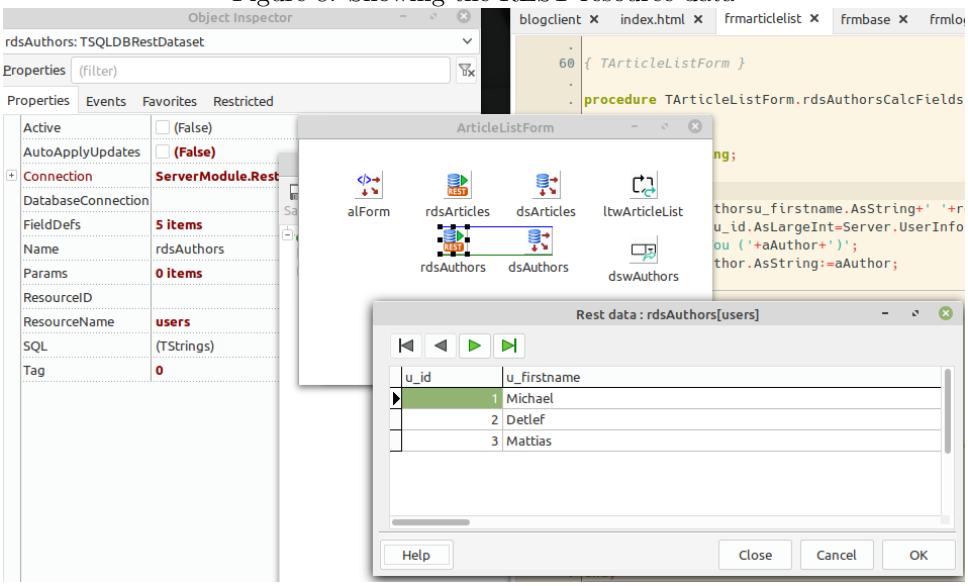
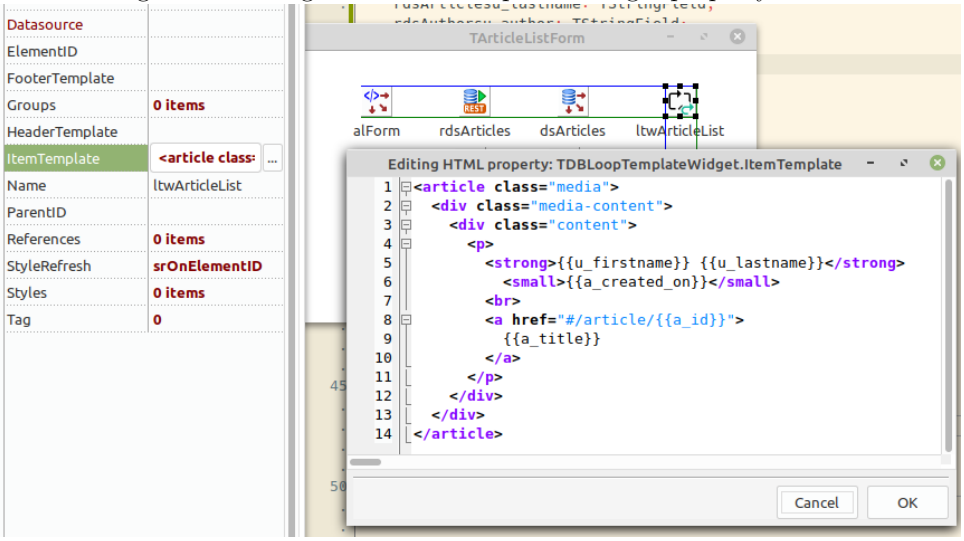


Figure 9: Editing the HTML templates using a property editor



This is a calculated field which we calculate in the `OnCalcFields` event of the `rdsAuthors` dataset:

```
procedure TArticleListForm.rdsAuthorsCalcFields(DataSet: TDataSet);

var
  aAuthor : String;

begin
  aAuthor:=rdsAuthorsu_firstname.AsString;
  aAuthor:=aAuthor+' '+rdsAuthorsu_lastname.AsString;
  if (rdsAuthorsu_id.AsLargeInt=Server.UserInfo.ID) then
     aAuthor:='You ('+aAuthor+')';
  rdsAuthorsu_author.AsString:=aAuthor;
end;
```

Here we take advantage of the persistent fields: the compiler will check all code, and code completion helps us not to make any mistakes against the fieldnames.

The `OnChange` event handler of the `dswAuthors` element can now be implemented much as it was in the previous version of our program:

```
procedure TArticleListForm.dswAuthorsChange(Sender: TObject;
                                            Event: TJSEvent);

Var
  aAuthorID : Integer;

begin
  aAuthorID:=StrToIntDef(selAuthor.value,-1);
  With rdsArticles.ParamByName('a_author_fk') do
    begin
    AsInteger:=aAuthorID;
    Enabled:=(aAuthorID<>-1);
    end;
  rdsArticles.Close;
  rdsArticles.Load;
end;
```

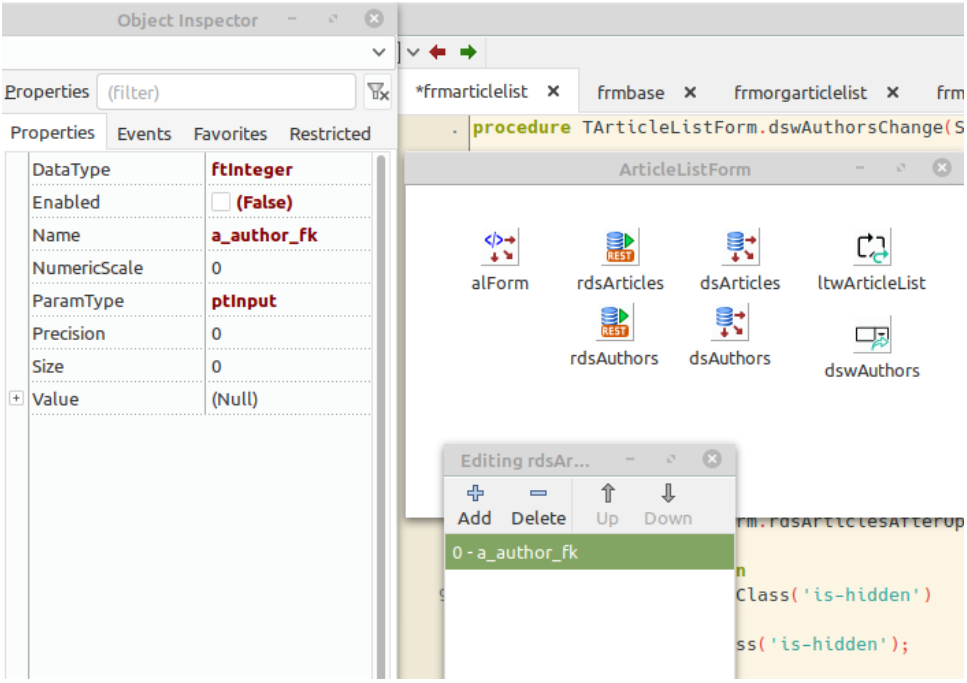
Note that we do not create the parameter `a_author_fk` in code, because we can use the object inspector to create it, as seen in figure 10 on page 19.

The `AfterOpen` event handler of the dataset can now also be created using the object inspector:

```
procedure TArticleListForm.rdsArticlesAfterOpen(DataSet: TDataSet);
begin
  if Dataset.IsEmpty then
    divNoArticles.RemoveClass('is-hidden')
  else
    divNoArticles.AddClass('is-hidden');
end;
```

As you can see the `divNoArticles` component (of type `THTMLELementAction`) has a convenience method to add or remove CSS classes.

Figure 10: Creating filter parameters using the Object Inspector



Last but not least, the `OnRendered` method of the form can be used to load the data:

```
  procedure TArticleListForm.DataModuleRendered(Sender: TObject);
begin
  if Server.UserInfo.ID>0 then
    divNewArticle.removeClass('is-hidden');
  rdsArticles.Load;
  rdsAuthors.Load;
end;
```

A word of caution: do not load the data in the `OnCreate` method. After the create, the form HTML is not yet rendered, and the data-aware components will not be able to render the data if the data arrives before the HTML fragment. Similarly, the `divNewArticle` element action is not yet bound to the HTML tag, and the `removeClass` method will not work yet.

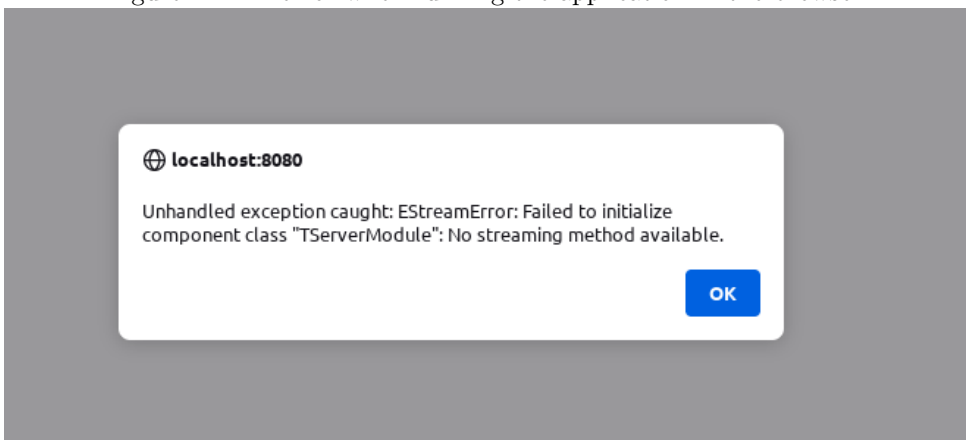
9 Ready to run

The last form to convert is the article form. In fact, it is completely similar to converting the article list, no new mechanisms are used, so we will not present it here.

The application class of our application can remain exactly the same as it was before, and with this the application is ready. Compile and running will however result in an error in the browser, as shown in figure 11 on page 20. The reason for this is that the pas2js compiler allows 2 kinds of resources:

- Resources added as Javascript in the application javascript files.

Figure 11: An error when running the application in the browser



- Resources added as link elements using a data url.

Since we didn't tell the compiler which resource mechanism to use, the compiler did not include our form files in the application. The solution is to add the -JRjs option in the custom compiler options, as can be deen in figure 12 on page 21. once that is done, compiling and running the program will work as before, see figure 13 on page 21

10 Conclusion

The support for visually designing HTML applications in the Lazarus IDE is a work in progress: although the current state is not yet WYSIWYG, the use of 2-way code editing tools is already possible. It makes working on an HTML application easier, and reduces the need for writing code considerably. The same components will be usable once the actual HTML designer works, so the way of working presented here will not change once the HTML editing is actually available: at that point you will simply be able to see the HTML in the designer, instead of a white background.

Figure 12: Adding the resource type to the compiler options

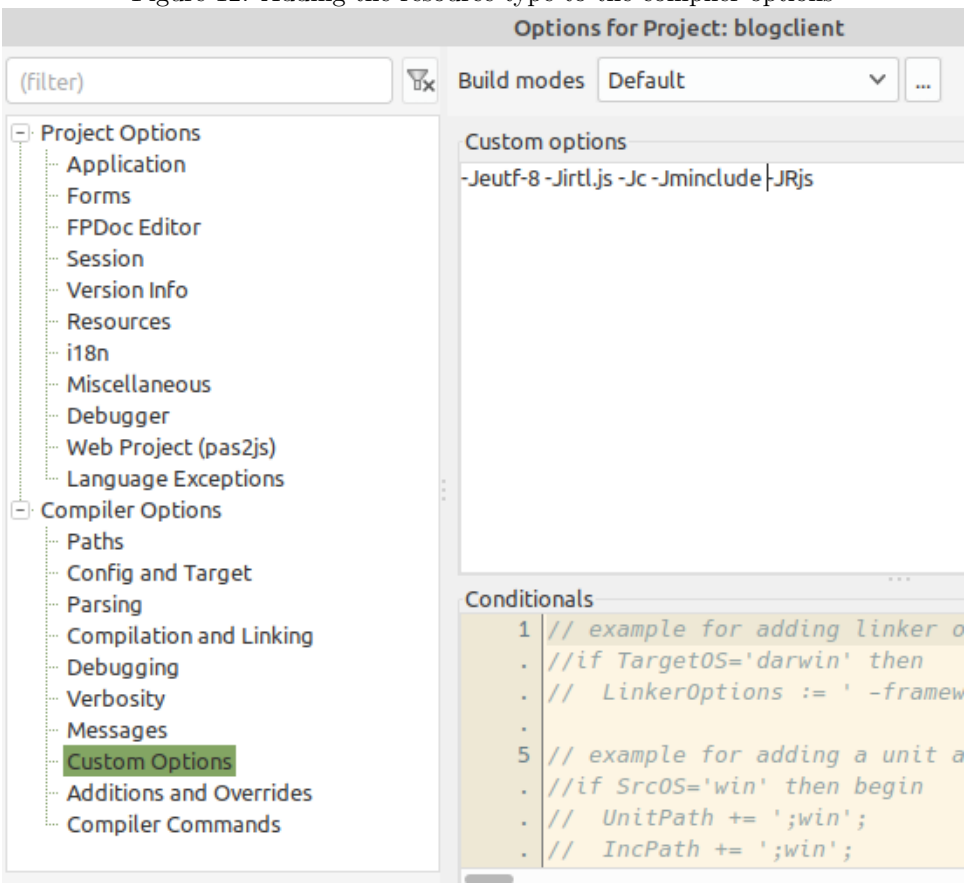


Figure 13: The reworked program in action