Handling multiple forms or pages in Pas2JS

Michaël Van Canneyt

March 14, 2022

Abstract

In this article we show how to reduce coding when creating forms in a Pas2JS web application. Additionally we show how routing can be used to show multiple forms in an SPA (Single Page Application) and keep the browser experience of the user intact.

1 Introduction

The previous articles showed how to implement a Pas2JS dialog, and how to switch to another dialog when the user logged in. All the examples shared a common approach: whether they used the WebWidget components or plain HTML classes, they always had one field per HTML tag element in the web page: the field was either a TWebWidget component or one of the HTML classes found in the Web unit. This is identical to how Delphi code deals with forms.

For example, the login page resulted in this declaration:

```
TMyApplication = class(TBrowserApplication)
  edtEmail: TJSHTMLInputElement;
  edtPassword : TJSHTMLInputElement;
  btnLogin : TJSHTMLButtonElement;
  procedure doLoginClick(aEvent: TJSEvent);
```

This is of course similar to a form declaration in Delphi. In the previous articles, these "form declarations" were created manually. In the below, we show how to generate such a declaration directly from the HTML file.

In Delphi, it is very common to show a second form with code like this:

```
Procedure TMainform.mnShowUserClick(Sender : TObject);

var
  frm : TUserForm;
begin
  frm:=TUserForm.Create(Self);
  frm.Show;
end;
```

It is possible to mimic this behaviour in a web application. But this is in fact not really how a user will expect a web application to function: when the user form appears as shown in the code, the user expects to be able to use the browser's back button to return to the previous form, or to reload the page using the refresh button.

The solution for this problem is called *routing*. With each form of the application, a URL is associated. The URL must contain enough information to reconstruct the form. For

example, the following 3 URLs could be used to respectively show the overview of users, create a new user and edit user with ID 123:

```
/users/
/users/new
/users/123
```

If the user is currently viewing URL `/users`, and navigates to the details of user 123 then the URL becomes `/users/123`. When the user wants to go back to the overview of users, he'll press the back button. The application should catch this event and present the user again with the overview of the users. We'll explain how this can be achieved in a Pas2JS application.

2 Generating form declarations

To avoid having to manually create a form declaration for each HTML file in a web application, a tool called html2form has been created. Its sources are distributed with Pas2JS, in the directory tools/html2form. It is a command-line application. When executed with the `-h` command-line option, you get some help messages which explain the various options:

help show a help message

below-id=ID Only create fields for child elements of element ID in the HTML page.

formclass=NAME The name of the pascal "form" class to create.

form-file Generate also a form .frm file (see below).

getelementfunction=NAME Name of `getelementByID` function: this is the function that is used in a `BindElements` method to look up an HTML element based on their ID attribute.

events When specified, the tool will emit code to bind event handlers to methods.

input=file With this option, you specify the html file to read.

map=file Read a mapping file, which is used to map HTML tags to Pascal classes, based on tag and attributes. By default, the tool maps HTML tags to the native Javascript `HTMLElement` child classes.

no-bind By default, the `BindElements` call which maps variables to actual instances is called from the class constructor. When this option is specified, the call to `BindElements` is omitted from the constructor

output=file The pascal file to write a unit to.

parentclass=NAME Name of pascal "form" parent class. There is no fixed `TForm` class in Pas2JS, so the tool needs a class name. By default, this class is `TComponent`.

exclude=List You can specify a comma-separated list of IDs to exclude: for these IDs, no field will be created. If the value for this option starts with @, then the remainder of the option is assumed to be a filename, and the list is loaded from the file.

These options give you an idea of the possibilities.

So, how to use this tool? Let's take the index.html file from our previous examples – it contains a login dialog – and run it through the tool using the following command-line:

```
html2form --input=index.html  -o frmlogin.pas -f TLoginForm
```

The result is a file that looks like this (some comments have been removed):

```
unit frmlogin;
{$MODE ObjFPC}
{$H+}

interface

uses js, web,Classes;

Type
  TLoginForm = class(TComponent)
  Published
    edtEmail : TJSHTMLInputElement;
    error : TJSHTMLElement;
    edtPassword : TJSHTMLInputElement;
    btnContinue : TJSHTMLButtonElement;
  Public
    Constructor create(aOwner : TComponent); override;
    Procedure BindElements; virtual;
  end;

implementation

Constructor TLoginForm.create(aOwner : TComponent);

begin
  Inherited;
  BindElements;
end;

Procedure TLoginForm.BindElements;

begin
  edtEmail:=TJSHTMLInputElement(document.getelementByID('edtEmail'));
  error:=TJSHTMLElement(document.getelementByID('error'));
  edtPassword:=TJSHTMLInputElement(document.getelementByID('edtPassword'));
  btnContinue:=TJSHTMLButtonElement(document.getelementByID('btnContinue'));
end;

end.
```

This "form" declaration will compile as-is and can be added to the Pas2JS project.

Many controls on a page need some kind of event handler: a button without event handler is of little use. Luckily, the html2form tool can also generate event handlers for you. For this, a convention is used. when looking at a tag, all attributes that begin and end with an underscore character (_) are considered event names. The value of the attribute is the event handler method name.

To demonstrate this, we modify the index.html a little. The login button becomes:

```
<button id="btnContinue"
```

```
     class="button is-block is-info is-large is-fullwidth"
     _click_="DoLoginClick">
     Continue <i class="fa fa-sign-in aria-hidden="true"></i>
</button>
```

The idea is that the 'click' event for the `btnContinue` button is handled by a method called `DoLoginClick`.

We run again the html2form tool on this file, but we also pass the `-event` command-line option:

```
html2form --input=index.html  -o frmloginbase.pas --event -f TBaseLoginForm
```

As you see, we also specify another name for the class file and the unit name. The reason for this will become apparent soon.

The resulting class has more methods:

```
TBaseLoginForm = class(TComponent)
Published
  edtEmail : TJSHTMLInputElement;
  error : TJSHTMLElement;
  edtPassword : TJSHTMLInputElement;
  btnContinue : TJSHTMLButtonElement;
  Procedure DoLoginClick(Event : TJSEvent); virtual; abstract;
Public
  Constructor create(aOwner : TComponent); override;
  Procedure BindElements; virtual;
  Procedure BindElementEvents; virtual;
end;
```

The `BindElementEvents` is where the events are bound to the callbacks:

```
Procedure TBaseLoginForm.BindElementEvents;

begin
  btnContinue.AddEventListener('click',@DoLoginClick);
end;
```

Note that the callbacks are marked `virtual; abstract;`. This is configurable: If you prefer, you can also simply generate virtual methods with an empty body.

But there is a reason for making these methods abstract: The class above is not meant to be used directly: If you generate a class from the HTML file, it can happen that the HTML changes, and you must change the class definition. If you do this and regenerate the file, any changes you made to the file will be lost. This is of course not very convenient.

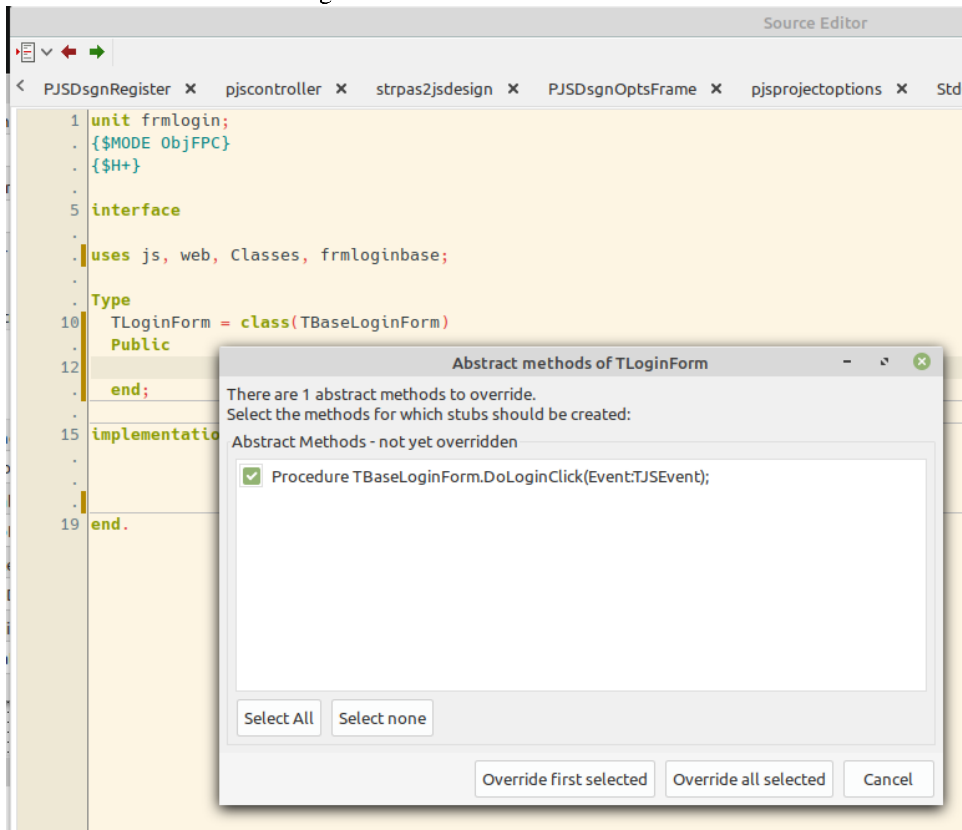
Instead, the above file is generated with abstract methods. To actually code the form's business logic, you create a new unit with a descendent of `TBaseLoginForm`:

```
unit frmlogin;
{$MODE ObjFPC}
{$H+}

interface

uses js, web, Classes, frmloginbase;
```

Figure 1: override abstract methods




```
Type
  TLoginForm = class(TBaseLoginForm)
  Public
  end;

implementation

end.
```

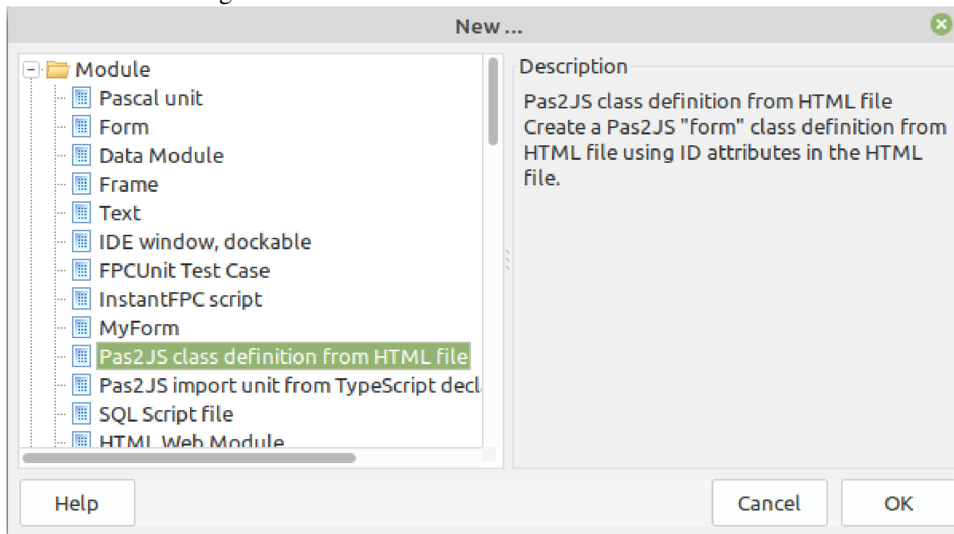
In this 'form' class, we override the abstract methods, and implement the GUI logic of the form. Now, when the HTML File changes, we can simply regenerate the `frmloginbase` unit, and continue to work in the `frmlogin` unit.

Overriding the abstract methods can be done trivially in the Lazarus IDE: The dialog under the *Source - Refactoring - Abstract methods* menu (see figure 1 on page 5) allows you to do this with a couple of mouseclicks. This dialog is also available from the source editor context menu popup, or you can attach a shortcut key to it.

The resulting code looks like this:

```
TLoginForm = class(TBaseLoginForm)
  procedure DoLoginClick(Event: TJSEvent); override;
Public
end;
```

Figure 2: Create a class definition from an HTML File




```
implementation

procedure TLoginForm.DoLoginClick(Event: TJSEvent);
begin

end;
```

All that is needed is to code the necessary UI or business logic. If you forget to implement some abstract methods, the compiler will warn you about this when you create an instance of a class which has abstract methods:

```
frmlogin.pas(29,14) Warning:
  Constructing a class "TLoginForm" with abstract method "DoLoginClick"
```

If you have the latest development version of Lazarus, this whole process has been automated in the IDE. In the `File-New` dialog, you can choose the `Pas2JS Class definition from HTML file` option (see figure 2 on page 6). When you choose this, you will be presented with a dialog that allows you to enter all possible options for the generating of the class definition, see figure 3 on page 7 and figure 4 on page 7. In this dialog, you can also opt to add the HTML file to the Lazarus project.

Once all the options have been set, the IDE will create the unit with the class declaration, and adds the new file to the project. In figure 4 on page 7 you can see that more options are available in the dialog than on the command-line.

In these screenshots, you see 2 toolbuttons: With these buttons you can load and save the options set in this dialog: this allows you to quickly re-use the same options for all forms in your application, and also allows you to use the saved options in an automated build procedure: the command-line appplication can read this file as well.

To ensure that you can recreate the class definitions at any given moment, the IDE automatically stores the options used to generate the unit in the Lazarus project file (the .lpi file). In the project inspector, you can use the context menu to regenerate one or more (the selected units) or all html form class files (see figure 5 on page 8).

Figure 3: Options for creating a class definition from an HTML File

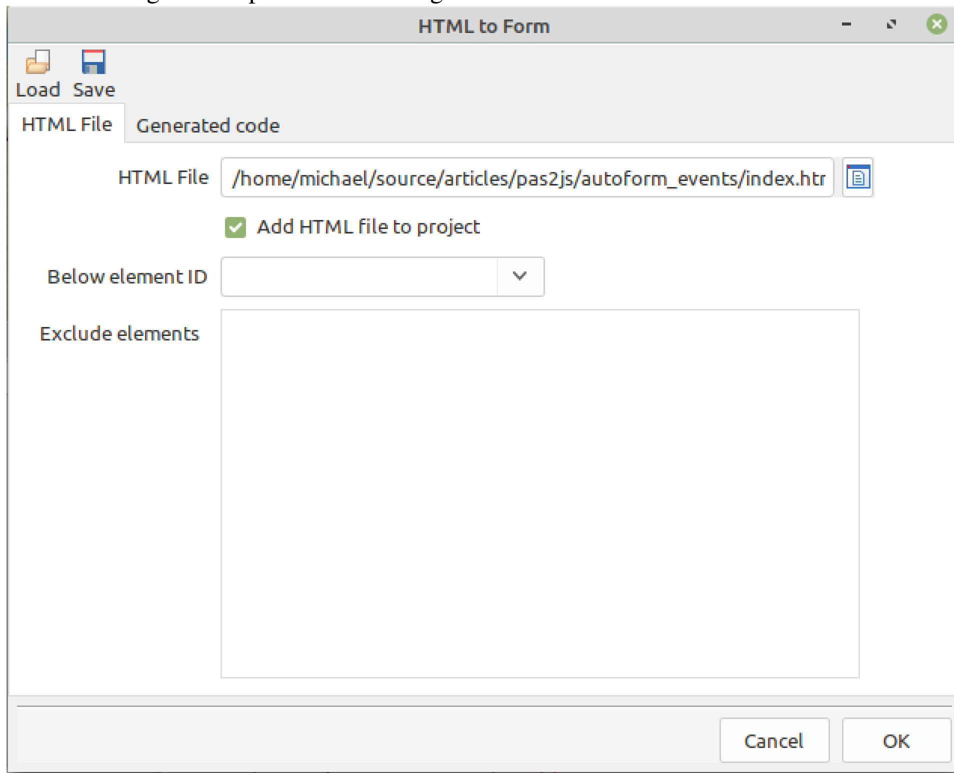


Figure 4: More options for creating a class definition from an HTML File

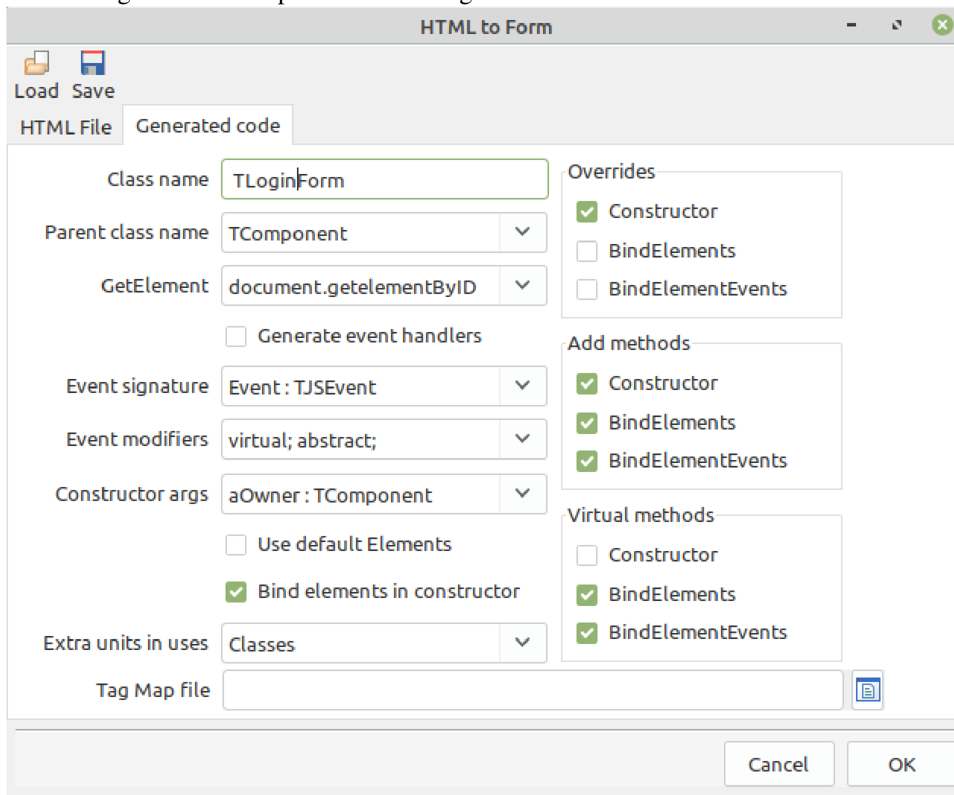
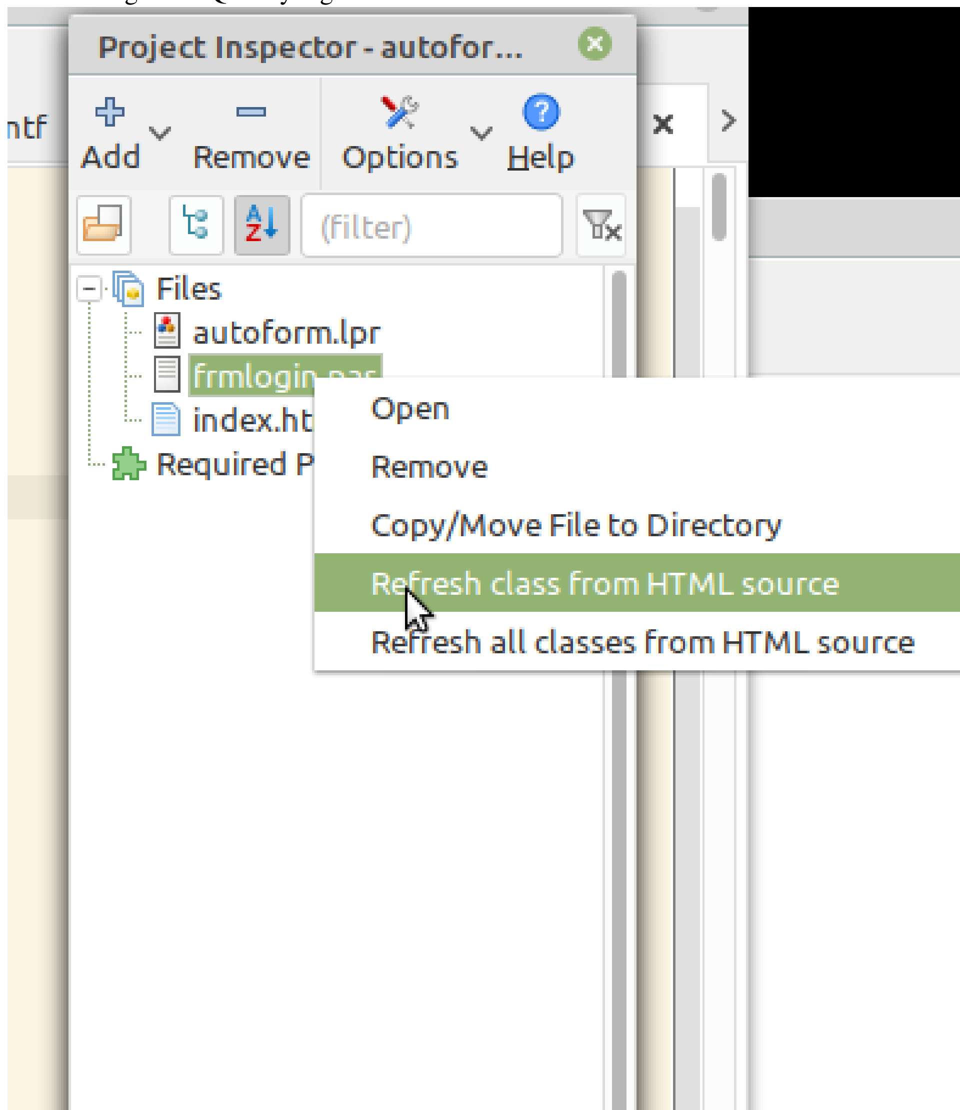


Figure 5: Quickly regenerate the class definitions from their HTML files



3 Navigating from one form to the next

A web application usually shows one form at a time: for instance, an overview of projects is shown, and when the user clicks a project, the overview disappears, and the details for the selected project is shown.

In a SPA (Single Page Application) this usually happens by showing all 'forms' below a designated HTML tag (let's give it an id: `form-parent`). This operation resembles docking a form in a main form in Delphi.

There are several ways to do this: all forms can be made part of the html - you just insert their HTML below the designated tag `form-parent`, give each form's top level HTML an ID. Then we can just show or hide parts of the HTML by adding or removing the following style element to the top level tag of the forms:

```
style="display: none;"
```

You could make the routine that does this part of the form constructor, and just create the form you need.

This is easy and convenient if there are only a few forms in your application. But in an application with many forms, the page's HTML will become unwieldy. Far better and easier is to have the HTML for each form in a separate file; By loading the HTML file at runtime, we can replace the HTML below the `form-parent` tag, and the browser will happily refresh the screen with your new form.

A difficulty with this approach is that loading a file from the server is an asynchronous operation; it takes some time. But this is not a big issue: we can start loading the forms as soon as the page is loaded. A second issue is of course that we should not reload a form each time it is opened: once it was loaded, we better keep the HTML somewhere in the browser, so we don't need to download it again next time the form is shown.

To help with all this, Pas2JS comes with a unit called `Rtl.TemplateLoader`. This unit will load a bunch of files (called templates) and keep them in some memory structure. When it is time to load a form, the needed template is requested from the template loader, and the form can be shown. If the template loader does not have it yet, you will need to tell it to load it and wait till it is loaded: the component will notify you when it was loaded so you can display the form.

The `TTemplateLoader` class is defined as follows:

```
TTemplateLoader = Class(TComponent)
  Procedure RemoveRemplate(aName : String);
  Function FetchTemplate(Const aName,aURL : String) : TJSPromise;
  Procedure LoadTemplate(Const aName,aURL : String;
                         aOnSuccess : TTemplateNotifyEvent = Nil;
                         AOnFail : TTemplateErrorNotifyEvent= Nil);
  Procedure LoadTemplates(Const Templates : Array of String;
                          aOnSuccess : TTemplateNotifyEvent = Nil;
                          AOnFail : TTemplateErrorNotifyEvent= nil);
  Property BaseURL : String ;
  Property Templates[aName : String] : String ;
  Property OnLoad : TTemplateNotifyEvent;
  Property OnLoadFail : TTemplateErrorNotifyEvent;
end;
```

The method names speak for themselves:

RemoveTemplate clears the template with name `aName`.

FetchTemplate Loads the template from URL `aURL` and stores the template with name `aName`. Returns a promise you can use to wait for the result.

LoadTemplate Loads the template from URL `aURL` and stores the template with name `aName`. You can optionally specify 2 event handlers, which will be called when the template is loaded or when the load fails.

LoadTemplates Passes a list of strings, strings at even indexes are the names of templates, strings at odd indexes are the URLS to load. You can optionally specify 2 event handlers, which will be called when a template is loaded.

The property names are equally clear:

BaseURL All urls in `FetchTemplate`, `LoadTemplate(s)` are relative to this URL.

Templates Here you can access a loaded template by name. If the template does not exist, an empty string is returned.

OnLoad Allows you to set a global template load notification event. This is called in addition to the ones specified in the load call.

OnFail Allows you to set a global template load failure notification event.

To demonstrate the use of this component, we'll make a web page with 3 "forms" – actually an HTML template file, and a button to show each form. The HTML template files will have an accompanying form declaration (we now know how to generate one quickly), which we will instantiate once the HTML has been loaded. For this, we need 3 html files:

1. The global HTML file. We'll name it `index.html`, and it will contain the buttons to display the 2 forms. This file would normally contain a menu, nav bar etc: the things which are always the same in every form.
2. The HTML file for the first form, a login page: we'll name it `login.html`.
3. The HTML file for the second form, a projects list page: we'll name it `projects.html`.
4. The HTML file for the third form, a users list page: we'll name it `users.html`.

Each HTML file will be accompagnied by a class form file, and we'll add some events to it, to demonstrate the capability of the html-to-form converter.

The `index.html` file is quite simple (we show just the HTML body):

```
<div class="container">
  <div class="box">
    <button class="button is-primary" id="btnLogin"
            _click_="DoLoginClick">Login</button>
    <button class="button is-info" id="btnProjects"
            _click_="DoProjectsClick">Projects</button>
    <button class="button is-info" id="btnUsers"
            _click_="DoUsersClick">Users</button>
  </div>
  <div class="box form-container" >
    <div id="form-parent" >
      <div class="notification is-info is-light">
        Click one of the buttons above.
      </div>
```

```
    </div>
  </div>
</div>
```

As you can see, there are 3 buttons, plus some tags that use Bulma CSS to create a visually more pleasing HTML page.

From this we use the File-New wizard to create a `frmIndex.pp` unit with the following class:

```
TIndexForm = class(TComponent)
Published
  btnLogin : TJSHTMLButtonElement;
  btnProjects : TJSHTMLButtonElement;
  btnUsers : TJSHTMLButtonElement;
  form_parent : TJSHTMLElement;
  procedure DoLoginClick(Event : TJSEvent);
  procedure DoProjectsClick(Event : TJSEvent);
  procedure DoUsersClick(Event : TJSEvent);
Public
  constructor create(aOwner : TComponent); override;
  procedure BindElements; virtual;
  procedure BindElementEvents; virtual;
end;
```

We do the same for the login, projects and users HTML files: For these files, the IDE will generate a class definition that looks much like the above. After doing this, we end up with 4 units in our project: `frmIndex`, `frmLogin`, `frmProjects` and `frmUsers`.

For simplicity, we will deviate from the 'proper' way to do things and simply implement the needed functionality in the units themselves.

The `TIndexForm` class is the 'main' form of our application. In this form, we must implement the logic for navigation between the login, projects and users form. Here is the logic to show the login page:

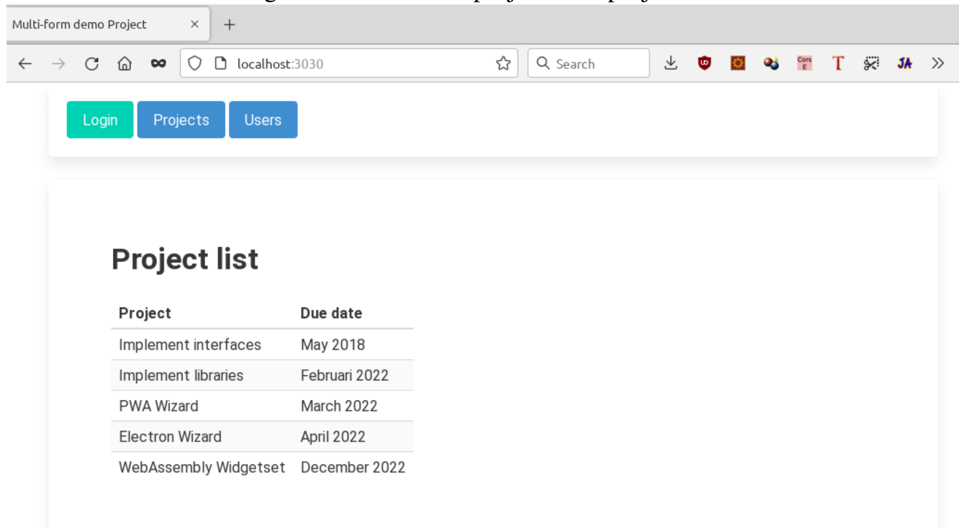
```
procedure TIndexForm.DoLoginClick(Event: TJSEvent);

  Procedure ShowLogin;
  begin
    form_parent.innerHTML:=GlobalTemplates.Templates['login'];
    FreeAndNil(FCurrentForm);
    FCurrentForm:=TLoginForm.Create(Self);
  end;

  procedure DoShowLogin(Sender: TObject; const aTemplate: String);
  begin
    ShowLogin;
  end;

begin
  if GlobalTemplates.Templates['login']<>'' then
    ShowLogin
  else
    GlobalTemplates.LoadTemplate('login','login.html',@DoShowLogin);
end;
```

Figure 6: Multi-form project with projects tab



The code is quite straightforward. `GlobalTemplates` is a global instance of the `TTemplateLoader` class, defined in the `Rtl.TemplateLoader` unit. If the template is known, then the `ShowLogin` is called. If the template is not yet known, it is loaded, and in the success handler, `ShowLogin` is called. For simplicity, we didn't use a failure event handler.

The `ShowLogin` routine enters the template HTML below the HTML tag with id `form-parent`. It then destroys any previous form instance in `FCurrentForm` - a variable that keeps the current form. Finally it creates the new form class and saves it.

That's all there is to it. For the `Projects` and `Users` pages, a similar routine is made, only the names differ. The result after pressing the `Projects` button is shown in figure 6 on page 12.

4 Using a factory pattern

The routines to show the login, projects, and users pages are the same. All that differs is the class name, and the name of the template and html file. If there are a lot of forms, then repeating the above code is of course not very efficient.

So, an obvious improvement to reduce code is to create a routine (or better, a class) which does all this in one call. It would also be nice if we could just pass a form name which says which form must be shown, without having to specify a class or a HTML file name.

To achieve this, we create a `TFormManager` class in a `frmBase` unit, which looks like this:

```
TFormManager = Class(TComponent)
Public
  Procedure RegisterForm(aClass : TBaseFormClass;
                         const aName : String = '';
                         aHTMLFile : String = '');
  Procedure UnregisterForm(aName : string);
  Procedure ShowForm(aName : string;
                     OnShow : TFormProcedure = nil);
  Property CurrentForm : TBaseForm;
```

```
  Property FormParent : TJSHTMLElement;
  Class property Instance : TFormManager;
end;
```

The `Instance` class property returns a global instance, which can be used to manage all forms.

With the `RegisterClass` routine, we can register a form class, using a name with which it can be shown, and a HTML file with which to load the HTML for the form. You can choose these last 2 parameters at will, but if you don't specify them, some defaults will be taken.

The `ShowForm` method can then be used to show a form using just the name used to register the form; A callback handler can be specified: it will be called when the form is shown.

The `ShowForm` routine looks much like the `OnClick` handler which we presented before, with as an addition a call to the `OnShow` handler that can be passed to the method:

```
procedure TFormManager.ShowForm(aName: string; OnShow: TFormProcedure);

Var
  Idx : Integer;
  Reg : TFormRegistration;

  Procedure ShowForm;
  var
    html : string;

  begin
    If Assigned(FCurrentForm) then
      FreeAndNil(FCurrentForm);
    html:=GlobalTemplates.Templates['form:'+Reg.Name];
    FFormParent.innerHTML:=html;
    FCurrentForm:=Reg.FFormClass.Create(Self);
    If Assigned(OnShow) then
      OnShow(Self,FCurrentForm);
  end;

  procedure FormFailed(Sender: TObject;
                       const aTemplate, aError: String;
                       aErrorcode: Integer);
  begin
    Writeln('Error loading form template',aTemplate,' : ',
            aError,' (Code:',aErrorCode,')');
  end;

  procedure HaveForm(Sender: TObject; const aTemplate: String);
  begin
    ShowForm;
  end;

begin
  Idx:=FForms.IndexOf(aName);
  if Idx=-1 then
    Raise EForms.CreateFmt(SErrUnknownForm,[aName]);
```

```
  Reg:=TFormRegistration(FForms.Objects[Idx]);
  if GlobalTemplates.Templates['form:'+Reg.Name]='' then
    GlobalTemplates.LoadTemplate('form:'+Reg.Name,Reg.HTML,
                                 @HaveForm,@FormFailed)
  else
    ShowForm;
end;
```

The `OnClick` handlers of our menu buttons in the index form can now be reduced to the following:

```
procedure TIndexForm.DoLoginClick(Event: TJSEvent);

begin
  FormManager.ShowForm('login');
end;

procedure TIndexForm.DoProjectsClick(Event: TJSEvent);

begin
  FormManager.ShowForm('projects');
end;

procedure TIndexForm.DoUsersClick(Event: TJSEvent);

begin
  FormManager.ShowForm('users');
end;
```

Obviously, before this can work, the login, projects and users forms need to be registered.

In the `RegisterForm` method of the `TFormManager` class, the `aClass` parameter is of type `TBaseFormClass`. This class reference type is also defined in the `frmBase` unit:

```
TBaseForm = class(TComponent)
Public
  Class Function FormName : String; virtual;
  Class Function FormHTMLFileName : String; virtual;
  Class Procedure Register;
end;
TBaseFormClass = class of TBaseForm;
```

The `Register` class method looks like this:

```
class procedure TBaseForm.Register;
begin
  With TFormManager.Instance do
    RegisterForm(Self,FormName,FormHTMLFileName);
end;
```

The `FormName` and `FormHTMLFileName` look like this:

```
class function TBaseForm.FormName: String;
```

```
Var
  P : integer;

begin
  Result:=LowerCase(ClassName);
  if Result.StartsWith('tfrm') then
    Result:=Copy(Result,5,Length(Result)-4)
  else if Result.StartsWith('t') then
    Result:=Copy(Result,2,Length(Result)-1);
  if Result.EndsWith('form') then
    begin
    P:=Pos('form',Result);
    Result:=Copy(Result,1,P-1);
    end;
end;

class function TBaseForm.FormHTMLFileName: String;
begin
  Result:=FormName+'.html';
end;
```

The result of all this code is that the line

```
TFrmLogin.Register;
```

will register the form class `TFrmLogin` with name `login` and html file `login.html`. The mechanism presented here is of course just a convention which makes life easier; you can perfectly invent other algorithms.

The start of our program becomes therefore:

```
TUsersForm.Register;
TProjectsForm.Register;
TLoginForm.Register;
FIndex:=TIndexForm.Create(Self);
FormManager.FormParent:=FIndex.form_parent;
```

Note that the `TIndexForm` is not registered: It has no associated HTML which must be loaded: the index.html file is already loaded.

5 Routing

We have now reduced the code it takes to show a form to a one-liner in an `onclick` handler. However, this does not solve our principal problem: the use of the *back* and *forward* buttons in the browser: if the user first opens the projects list and then goes to the users list, he will naturally assume he can go back to the projects list by hitting the back button.

With the application as it is coded now, if you press the back button while the users list is shown, either

- Nothing will happen if the demo is the first page loaded in your browser.
- Or you will be taken back to the website you were looking at before you opened the demo.

The solution to this problem is called *routing*: with each form we associate an URL. As the user navigates between the forms, the URL changes. This is easy with a website where each form is an actual and separate HTML page. But how to do this in a Single Page Application (SPA)?

Luckily, in HTML 5, this is possible: the browser offers access to the history mechanism of your browser page. You can be notified if the URL changes, and you can also change the URL. Since we are creating a SPA (Single Page Application) we must of course try to avoid a page reload, and remain in the current page.

But how to stay on the same page when we require that the URL must change when navigating from one form to another? This also is possible: the hash part of the URL can be used.

The following 3 URLs are the same page:

```
http://localhost:3000/index.html#/login
http://localhost:3000/index.html#/users
http://localhost:3000/index.html#/projects
```

These are 3 different URLs, but they all refer to the same HTML page. When you are on the last URL in the list, and press the back button, the browser will see that the previous URL is actually the same page, and will not reload the page from the webserver.

This mechanism can be further expanded, you can pass more information in the URL.

The following can refer to 1 page (a fictitious project detail page), which will – in turn – show the details for project 1, a new project and project 2.

```
http://localhost:3000/index.html#/project/1
http://localhost:3000/index.html#/project/new
http://localhost:3000/index.html#/project/2
```

What is more, the user can copy the URL, send it to someone else, and the receiver can open the application and be presented with the same page.

So, how to achieve this? The Pas2JS RTL comes with a webrouter unit, which implements a `TRouter` class. This class allows you to associate a callback with a route. A route is simply an URL fragment: when the URL changes, the router will catch the browse event for it, and match the new URL with the list of known routes. If it finds a route definition that matches the URL, it will call the registered callback for that route.

For example, these are possible routes for our application:

```
/login
/project
/project/new
/project/:ID
/user
/user/:ID/Tab/:TAB
/user/:ID/
/*
```

Notice the `:ID` and `:TAB` in these routes. They present parameters: any string that does not contain a / character. When the router matches the URL, it will replace `ID` with what was actually in the URL. This means that the following URL fragments:

```
/project/123
/project/789
```

will result in a match for the route /project/:ID, but with ID set to 123 and 789, respectively.

You can also use the wildcard character * to match any URL fragment. This can be used for example to register an error page if no matching URL was found, or to handle all URLS that start with a certain fragment in a single route definition.

The following is the declaration of the TRouter class, with only the most important methods:

```
TRouter = Class(TComponent)
  Procedure DeleteRoute(aIndex : Integer);
  Function RegisterRoute(Const aPattern : String;
                         aEvent: TRouteEvent;
                         IsDefault : Boolean = False) : TRoute;
  function FindHTTPRoute(const Path: String;
                         Params: TStrings): TRoute;
  function GetRoute(const Path: String;
                    Params: TStrings): TRoute;
  Function RouteRequest(Const aRouteURL : String;
                        DoPush : Boolean = False) : TRoute;
  Property Routes [AIndex : Integer] : TRoute ;
  Property RouteCount : Integer;
  Property BeforeRequest : TBeforeRouteEvent;
  Property AfterRequest : TAfterRouteEvent;
end;
```

The purpose of these methods should be clear:

DeleteRoute Delete given route by index.

RegisterRoute Register a callback for a route: the aPattern is a pattern to match with the URL. If the URL matches the route, then aEvent is called. If isDefault is True then this route is used if no matching route can be found for a given URL fragment.

FindHTTPRoute Find a route definition for Path, and return parameter values in Params. Returns the route definition. If no route is found, Nil is returned.

GetRoute calls FindHTTPRoute, and raises an exception if no route was found.

RouteRequest Perform the routing for a request with URL frament aRouteURL. If DoPush is true, the new route is pushed onto the browser's URL history.

Routes Array access to the registered routes.

RouteCount The number of known routes.

BeforeRequest An event that is fired before handling a routing request.

AfterRequest An event that is fired after handling a routing request.

How can we use this object to show our forms automatically in the application ? A simple mechanism suggests itself: each form registers a route starting with the form name used to create the form. This means that our three forms must register 3 routes:

```
/login
/projects
/users
```

Now we can pluck additional fruits of the factory pattern that we introduced earlier. We can use the `RegisterForm` call to register a route for the form.

To allow a form to register multiple routes for itself, we create a `FormRoutes` method in `TBaseForm`:

```
class function TBaseForm.FormRoutes: TStringDynArray;
begin
  Result:=[FormName];
end;
```

This method (which can return multiple routes) is then used to register the routes for the form in the form manager's `RegisterForm` method. This method starts with some sanity checks, before adding a form registration object to a list. The `FormRoutes` method is then used to register the various routes for the form:

```
function TFormManager.RegisterForm(aClass: TBaseFormClass;
                                   const aName: String;
                                   aHTMLFile: String):
                                   TRouteDynArray;

Var
  aRoute,N,H : String;
  aRoutes : TStringDynArray;
  aReg : TFormRegistration;
  Idx : Integer;

begin
  // Some cleanup
  N:=aName;
  if N='' then N:=aClass.FormName;
  H:=aHTMLFile;
  if H='' then H:=aClass.FormHTMLFileName;
  // Create and save form registration.
  aReg:=TFormRegistration.Create(aClass,N,H);
  FForms.AddObject(N,aReg);
  // Register routes
  aRoutes:=aClass.FormRoutes;
  SetLength(Result,Length(aRoutes));
  Idx:=0;
  for aRoute in aRoutes do
    begin
    Result[Idx]:=Router.RegisterRoute(aRoute,@DoFormRoute,False);
    Inc(Idx);
    end;
  // Save routes in registration.
  aReg.FRoutes:=Result;
end;
```

As a last step, the created routes are saved in the form registration. This is needed in the `DoFormRoute` method, which will be called when the route is matched.

In the `DoFormRoute` method, we start with looking up the form registration associated with the route. The `HasRoute` helper function checks if the given route is in the array of routes for that form registration.

```
procedure TFormManager.DoFormRoute(URl: String;
                                   aRoute: TRoute;
                                   Params: TStrings);

Var
  Idx : Integer;
  Reg : TFormRegistration;

begin
  // Find the form registration for this route:
  Reg:=Nil;
  Idx:=FForms.Count-1;
  While (Reg=Nil) and (Idx>=0) do
    begin
    Reg:=TFormRegistration(FForms.Objects[Idx]);
    if Not Reg.HasRoute(aRoute) then
      Reg:=Nil;
    Dec(Idx);
    end;
  // If we found a registration, show the form
  if Assigned(Reg) then
    ShowForm(Reg.Name,
      procedure (sender: TObject; aForm : TBaseForm)
      begin
      aForm.ShowRoute(URL,aRoute,Params);
      end);
end;
```

Finally, if a valid form registration is found, then we show the form using the existing `ShowForm` method. In the `OnShow` callback we call a new method of our base form class, `ShowRoute`:

```
procedure TBaseForm.ShowRoute(const aURL: String; aRoute: TRoute;
  aParams: TStrings);
begin
  Writeln('Showing route for URL ',aURL,' with pattern: ',
          aRoute.FullPath,' and params : ',aParams.CommaText);
end;
```

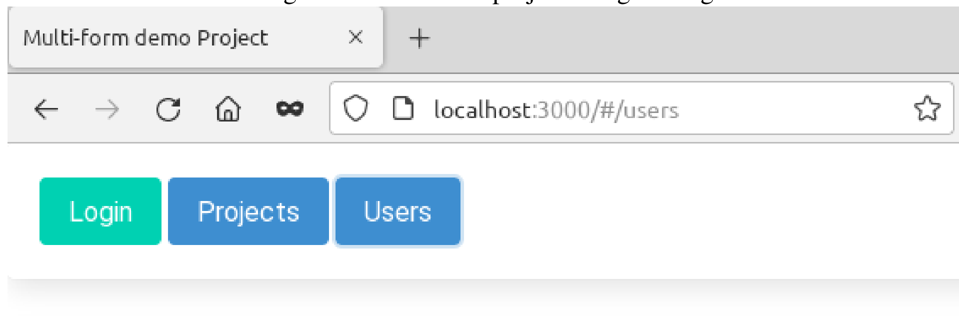
This virtual method can be overridden to let the form act on the particular route that was used to show the form. For instance, to react on parameters in the route.

So, now that we have our routing in place, how to use it? This is simple, and we actually end up with less code. The 3 buttons in the index.html page to show our 3 forms can now be replaced with 3 anchor elements:

```
<div class="box">
  <a class="button is-primary" id="btnLogin" href="#/login">Login</a>
  <a class="button is-info" id="btnProjects" href="#/projects">Projects</a>
  <a class="button is-info" id="btnUsers" href="#/users">Users</a>
</div>
```

As you can see, the `button` HTML tag has been replaced with an anchor HTML tag (`a`). In the anchor tag's `href` attribute, we enter the route for the form that must be shown: `#/`, followed by the form name. The `click` handler has also been removed: it is no longer

Figure 7: Multi-form project using routing



User list

Name	Country
Detlef overbeek	The netherlands
Mattias Gaertner	Germany
Sven Barth	Germany
Florian Klaempfl	Germany
Michael Van Canneyt	Belgium
Jonas Maebe	Belgium

needed. If we now regenerate the class file associated with our index.html file, we notice that the `click` handlers are gone. The navigation is now handled by the router.

The result can be seen in figure 7 on page 20. Notice how in in the address bar of the browser, the route is now displayed within the URL's hash. As you navigate between forms, the URL will change as you switch forms. Additionally, if you now use the back and forward buttons of the browser, you will actually switch forms ! With this mechanism, you are giving the user a real browser experience.

Incidentally, note that the hyperlink elements look exactly like button elements used before: this is one of the perks of using a CSS framework.

6 Route parameters

To demonstrate the use of parameters in the URL, we change the projects overview page to show links to a 'project details' page for a project:

```
<tr>
  <td>
  <a href="#/project/1">Implement interfaces </a>
```

```
  </td>
  <td>
  May 2018
  </td>
</tr>
```

The HTML of the project detail page (project.html) looks like this:

```
<h1 id="pagetitle"
    class="title is-3">Project:
      <span id="hdrProjectName">?</span>
</h1>
<div id="lblNotFound"
     class="notification is-danger is-light is-hidden">
     Project %d not found !</div>
<div class="field">
  <label class="label">Project Name</label>
  <div class="control">
  <input class="input"
         id="edtProjectName"
         type="text"
         placeholder="Project name">
  </div>
</div>

<div class="field">
  <label class="label">date due</label>
  <div class="control has-icons-left">
    <input class="input is-success"
           type="text" id="edtDueDate"
           placeholder="project due date">
    <span class="icon is-small is-left">
      <i class="las la-calendar-check"></i>
    </span>
  </div>
</div>

<div class="field is-grouped">
  <div class="control">
    <button id="btnSave"
            class="button is-link">
        Save
    </button>
  </div>
  <div class="control">
    <button id="btnCancel"
            class="button is-link is-light">
       Cancel
    </button>
  </div>
</div>
```

When we generate the form for this HTML, we call the form class `TProjectDetailForm`, and we override the following methods:

```
Procedure ShowRoute(Const aURL : String;
                    aRoute : TRoute;
                    aParams : TStrings); override;
Class function FormHTMLFileName: String; override;
Class function FormRoutes: TStringDynArray; override;
```

Since the form class name differs from the html file name (the convention that was presented earlier), we need to give the form factory the correct HTML file name:

```
class function TProjectDetailForm.FormHTMLFileName: String;
begin
  Result:='project.html';
end;
```

Since we wish to obtain the value of the form ID as a parameter in the URL, we must register a fitting route for this:

```
class function TProjectDetailForm.FormRoutes: TStringDynArray;
begin
  Result:=['/project/:ID']
end;
```

The result is that project ID will be passed to the `ShowRoute` in the `ID` parameter.

We can now use this parameter to load the correct project data. If a wrong ID or a false ID is loaded an error message is displayed: The user can type an arbitrary or outdated URl in the browser address bar, and we must be prepared to deal with errors. With a simple Bulma CSS class (`is-hidden`), a HTML element can be shown or hidden. Showing a warning is thus simply a matter of removing the `is-hidden` CSS class from the HTML element that shows the warning.

The data is loaded from 2 arrays of values (`ProjectNames` and `ProjectDates`).

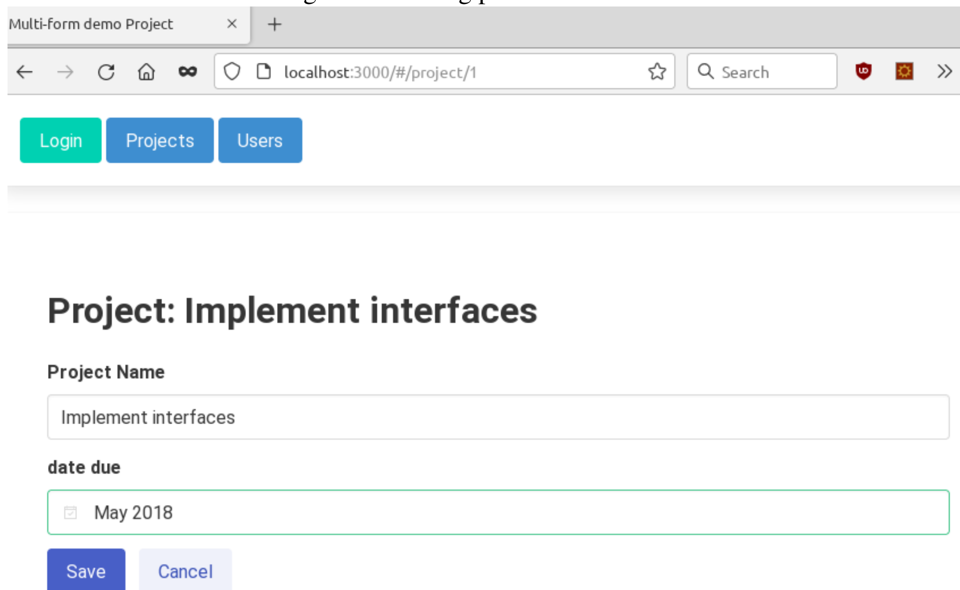
```
procedure TProjectDetailForm.ShowRoute(const aURL: String;
                                       aRoute: TRoute;
                                       aParams: TStrings);

Const
  NotFound = 'Project "%s" not found!';

Var
  aID : NativeInt;
  aError,aName,aDue : String;

begin
  aID:=StrToInt64Def(aParams.Values['ID'],-1);
  // Show an error if the ID is unknown.
  if (aID<1) or (aID>ProjectCount) then
    begin
    aError:=Format(NotFound,[aParams.Values['ID']]);
    lblNotFound.innerText:=aError;
    lblNotFound.classList.remove('is-hidden');
    Exit;
    end;
  // Show project data
```

Figure 8: Routing parameters in action




```
  aName:=ProjectNames[aID];
  aDue:=ProjectDates[aID];
  hdrProjectName.InnerText:=aName;
  edtProjectName.value:=aName;
  edtDueDate.value:=aDue;
end;
```

The last lines are not very different from what you would do in a regular VCL Class: only the property names are different.

The result of this code can be seen in figure 8 on page 23. Note the URL which contains the project ID. As you navigate between the various projects, you can always go back to a previously visited project with the browser's back button.

7 Conclusion

In this article, we've shown how to present the user with an actual browser experience: back and forward buttons for navigation now work. In doing so, the work needed to show forms was significantly reduced: Using a router and changing buttons to anchor elements in the html reduces code.

There are still small glitches: when reloading the page, you will return to the initial page, even though the URL contains the route for the last visited page. It would also be nice if data for the projects could be loaded from an actual database. We will deal with these issues in a next contribution.