

Connecting to a database with pas2js

Michaël Van Canneyt

July 2, 2022

Abstract

In this article we explore how to read and write data from a database using Pas2JS and Free Pascal.

1 Introduction

Many websites use data from a database to serve content to users. It can be any kind of data, but most often it will be some SQL-based database: MySQL, PostgreSQL, MS-SQL server or any other. However, the browser does not allow direct connections to such a database: It only supports the HTTP protocol. For this reason, database access must be handled by a server process that sits between the browser and the database.

HTTP does not provide a protocol to access data. So, a protocol that makes use of HTTP must be devised to access data. In the previous articles, the RPC protocol was introduced to execute methods on the server. This could be used to access data: introduce some methods to fetch and update data on the server. The most common protocol for data access these days is not RPC based, it is a mechanism called REST. REST is not a real protocol. It is a series of practices that translate to 2 basic rules:

- All data is addressable using a HTTP URL.
- the HTTP verbs can be used to get, create, update or delete data.

Most REST implementations use JSON to encode the data for transport over HTTP. This is because JSON is lightweight and, as a subset of Javascript, it can be handled natively by the browser.

So we have 2 elements in our story so far: an SQL database and a REST protocol. Free Pascal uses SQLDB to access SQL databases. To allow access to SQL databases a series of components exist which, taken together, form the SQLDBRestBridge framework.

Basically, SQLDBRestBridge forms a bridge between the SQL database and the browser, using a REST approach and the SQLDB components. It is a set of components that is integrated in the FCL-Web technology that exists on the server: whatever technology you use to embed your server logic in, your REST server will function the same.

Once the SQLDBRestbridge is set up, the data can be accessed in the browser using a regular `TDataSet` descendant. Updates are handled using the well-known `ApplyUpdates` call.

2 SQLDBRestbridge features

The SQLDBRestbridge system exposes a REST interface for one or more databases in a HTTP server.

It has the following main features:

1. Connect to multiple databases. Any database type supported by SQLDB can be accessed.
2. Have more or one resource schemas.
3. Various in/output formats are supported: JSON, XML (in multiple flavours: TClient-Dataset, Access format (ADO), and custom XML), CSV. It is easy to add your own format.
4. A descriptive Metadata resource is available : all resources are described.
5. Simple URL scheme:

/REST/ResourceName the list of all resources named `ResourceName`

/REST/ResourceName/ID A single resource named `ResourceName`

/REST/ResourceName?CustomerID=123 the list of all resources named `ResourceName` with property `CustomerID` equal to 123.

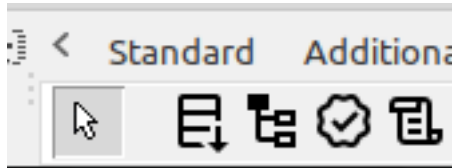
6. Complete configuration of filtering of resource lists. Any field can be configured to be filtered on, custom filters can be added too.
7. You can control the list of fields in the output using optional query strings. Either specify fields to include or fields to exclude from the output.
8. Paging of results is supported, when possible using the native mechanisms of the SQL database, simulated when needed.
9. CORS support.
10. Authentication can be handled out-of-the box or completely customized. Simple authentication (the username and password are in one of the database tables) can be handled with no code at all, only an SQL statement is needed.
11. Optional: allow the client to specify an SQL statement.
12. Optional: Connections can be managed through a resource as well. This allows you to set up a single general-purpose binary which can be configured to connect to any database at runtime.
13. Optional: reverse engineer a database to expose all tables as REST resources.
14. Optional: Save and load a configuration from an .ini file.

3 Server-side SQLDBRestbridge components

SQLDBRestbridge can be installed in the lazarus IDE using the `lazsqldbrest` package. Once installed, 4 components appear on the component palette, see figure 1 on page 3.

The main component is implemented in the `SQLDBRestbridge` unit:

Figure 1: The SQLDBRestBridge components on the *fpWeb* tab



TSQLDBRestDispatcher This component is responsible for handling the HTTP request. It takes care of error handling, running queries and transforming data into a transportable format: JSON, XML, CSV etc. It holds the Database connection definitions (more than one connection is possible) and has lots of properties that describe what the REST interface looks like to the clients.

TSQLDBRestSchema This component defines available resources for the dispatcher: a resource is essentially a view (an SQL select statement) on a database, together with SQL statements that can be used to update the data. The schema can be edited visually in the IDE or with a stand-alone program. There are also some methods to import a schema directly from a database.

The dispatcher can have one or more schemas attached to it (They can be specified in the `Schemas` property), and when a client requests a resource, it will look in each of the schema for the definition of the resource.

TSQLDBRestBusinessProcessor Although both the `TSQLDBRestDispatcher` and the `TSQLDBRestSchema` components have lots of options to customize and fine-tune the behaviour of the REST bridge, sometimes you will need to handle some cases specially.

The `TSQLDBRestBusinessProcessor` component lets you implement a set of event handlers that allow you to control the behaviour of the REST bridge at the level of a single resource: for instance fine-grained access control, check input etc. It follows that you can have many business processors attached to the REST dispatcher: one for each defined resource.

TRestBasicAuthenticator Is a component that allows you to implement BASIC http authentication. You can do this with an event handler, or you can set up an SQL statement that will return a user ID based on a username/password.

All these components have lots of properties. They will not be discussed in detail here, it would lead too far. The properties are discussed in the FPC Wiki pages:

https://wiki.freepascal.org/SQLDBRestBridge#Available_Components

4 Client-side SQLDBRestbridge components

In the browser application, there are 2 components that must be used, they are implemented in the `sqldbrestdataset`.

TSQLDBRestConnection Basically, this component specifies the location where the SQLDBRestBridge server process is listening. It has some properties that describe the server configuration: the name of the metadata resource, the connections resource name. Optionally a username and password can be specified.

SQLDBRestDataset This is an actual dataset which can be used to get the data of a resource. You must specify the connection, the name of the resource and optionally an ID if you want to get a single resource, or add some filters if you want to get a selection of all resources.

This works mostly as you would expect it to work in a native client/server database application using `TClientDataset`. There are some caveats, as the browser works asynchronously, we will describe this later on.

5 A sample application

To demonstrate the use of `SQLDBRestBridge` components, we will create a small blogging application. It has 3 tables: one for users, one for articles and one for comments. The SQL for these tables looks like this:

```
create sequence seq_users;
create table users (
  u_id bigint not null default nextval('seq_users'),
  u_firstname varchar(50) NOT NULL,
  u_lastname varchar(50) NOT NULL,
  u_login varchar(127) not null,
  u_password varchar(127) not null,
  constraint pk_users primary key (u_id)
);

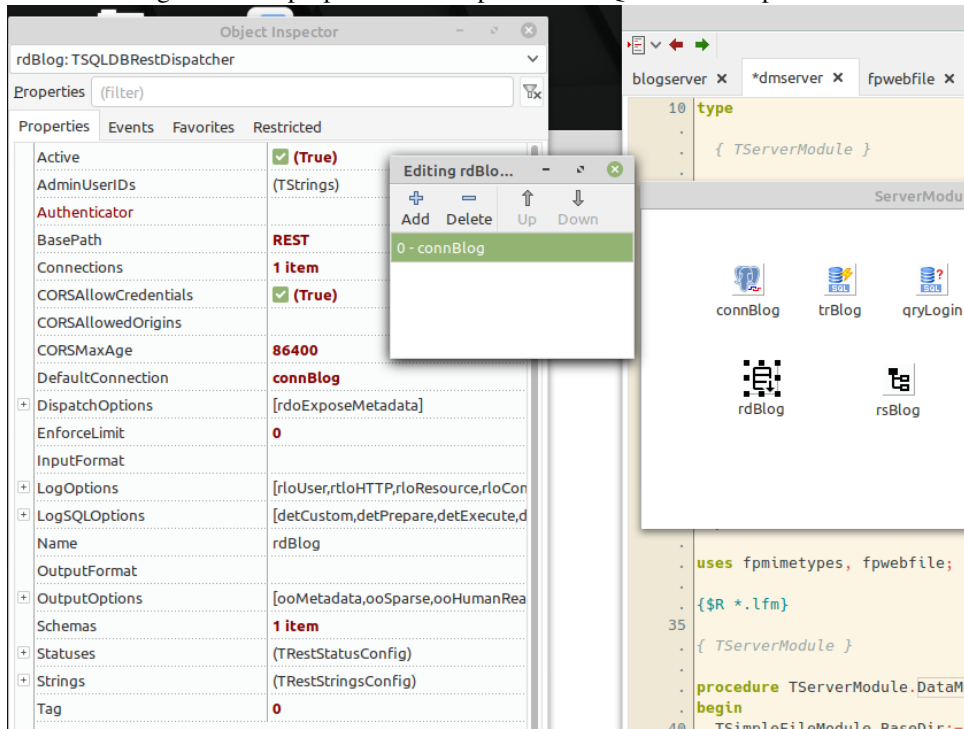
create sequence seq_articles;
create table articles (
  a_id bigint not null default nextval('seq_articles'),
  a_created_on timestamp not null default CURRENT_TIMESTAMP,
  a_author_fk bigint not null,
  a_title varchar(255) not null,
  a_text text not null,
  constraint pk_articles primary key (a_id)
);

create sequence seq_comments;
create table comments (
  c_id bigint not null default nextval('seq_comments'),
  c_created_on timestamp not null default CURRENT_TIMESTAMP,
  c_article_fk bigint not null,
  c_author_fk bigint not null,
  c_text text not null,
  constraint pk_comments primary key (c_id)
)
```

The SQL is for a PostgreSQL database, but for other databases the SQL will look similar, the use of a sequence may vary somewhat.

The server will expose a RPC service to allow a user to log in and out (we will reuse the code presented in the previous articles for this), and a REST service using `SQLDBRestBridge`.

Figure 2: The properties to set up for the TSQLDBRestDispatcher



6 The server application

The server application is a simple HTTP application. It will serve the HTML files to the browser, and host the RPC and REST services. There are 2 modules in the application:

1. ServerModule a simple data module with an SQLDB Database connection component, and the SQLDBRestBridge components.
2. dmRPC a simple FPWeb RPC module which offers the RPC API to authenticate a user.

The RPC service will not be discussed here, it is essentially the same as presented in the previous articles in this series, but the 2-factor authentication code has been stripped.

The REST functionality is achieved by dropping 2 components on the ServerModule:

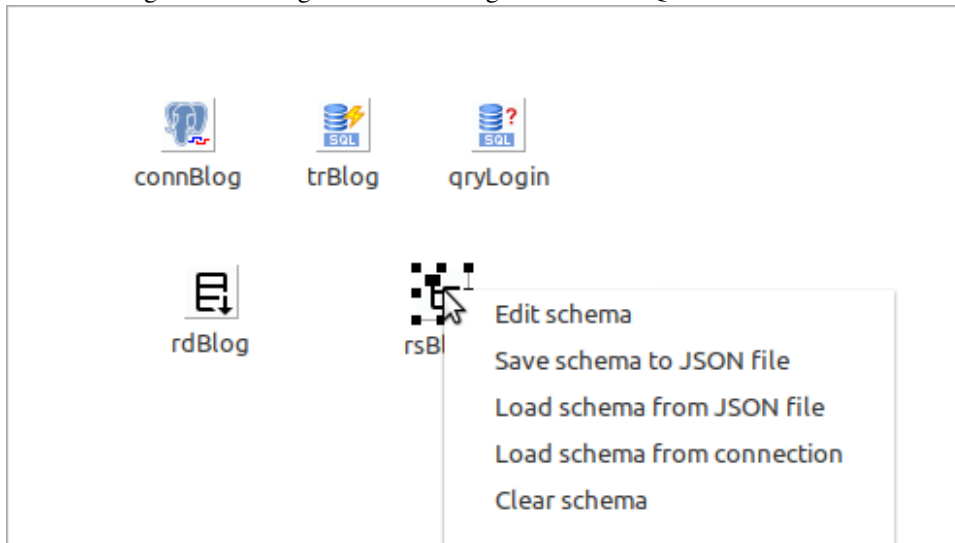
1. a TSQLDBRestDispatcher component (we'll name it rdBlog)
2. a TSQLDBRestSchema component (we'll name it rsBlog)

In the rdBlog component, we need to set up several properties (see figure 2 on page 5)

Connections this is a property which defines the SQLDB connections to the databases you want to expose. Each collection item has a unique name and contains the standard DatabaseName, HostName, UserName and Password parameters needed to set up a connection. You can also specify an existing connection component in the SingleConnection property of the collection item: in that case this connection component will be used to connect to the database.

DefaultConnection must be set to the default connection.

Figure 3: Starting the schema designer for the TSQLDBRestSchema



Schemas A collection of schemas to serve to clients. Here you specify the schema component `rsBlog` which has been dropped on the datamodule.

BasePath This is the first component of the URL which will be checked to determine if a HTTP request must be handled by the `SQLDBRESTBridge`. The `RPC` path is used to identify RPC requests, so `REST` is a logical choice for the REST requests.

To set up the schema with the resource definitions, there are 2 possibilities:

1. Use the stand-alone schema editor: This is a tool that can be used to edit schema files. The tool is distributed with `lazarus`, but you will need to build it yourself; the sources are in the

```
components/sqldbrest/editor
```

directory. The schema file (an ordinary JSON file) can be loaded into the schema component in the designer or at runtime.

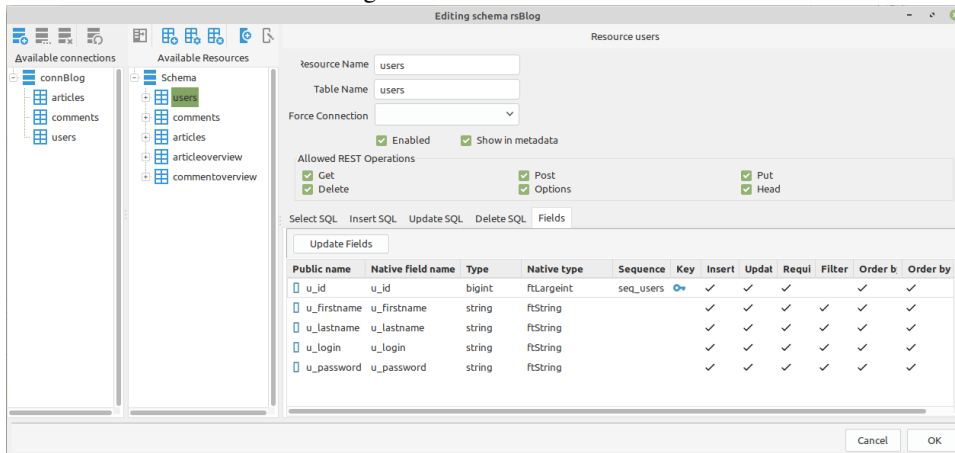
2. Use the component editor menu to edit the schema right in the IDE.

We'll use the latter approach, see figure 3 on page 6. When started, the main form of the designer is divided into three parts.

1. At the left is a tree view with the available connections and the tables of each connection.
2. In the middle, a list of defined resources with their fields. Each resource will be available through the REST interface.
3. At the right, there is a pane which will change depending on the selection in the resources tree. It can be the resource definition (SQL statements, allowed operations etc.) or the definition of a field or an overview.

At the top is a menu bar which allows you to add/edit/delete connections, resources and fields. To start a schema, the simplest is to drag and drop a table from the left pane to

Figure 4: A default schema



the middle pane: When you do this, a resource corresponding to the table contents will be defined with all the fields of the table and some default SQL statements (empty, as a matter of fact) and default properties. You can see the result when we drag the `users` table from the `connBlog` connection to the middle pane in figure 4 on page 7. This is in fact what would happen if you allow the `SQLDBRestBridge` to reverse-engineer a database at runtime.

If you check the SQL statements for the resource, you will find they are empty: this is because the `SQLDBRestbridge` will generate the statements at runtime from the table name and the field definitions. It needs of course to know the primary key field, but normally it can detect this from the table definition.

While `SQLDBRestBridge` can cope with primary keys that span multiple fields, it complicates things somewhat, and it is recommended to have a primary key consisting of a single field.

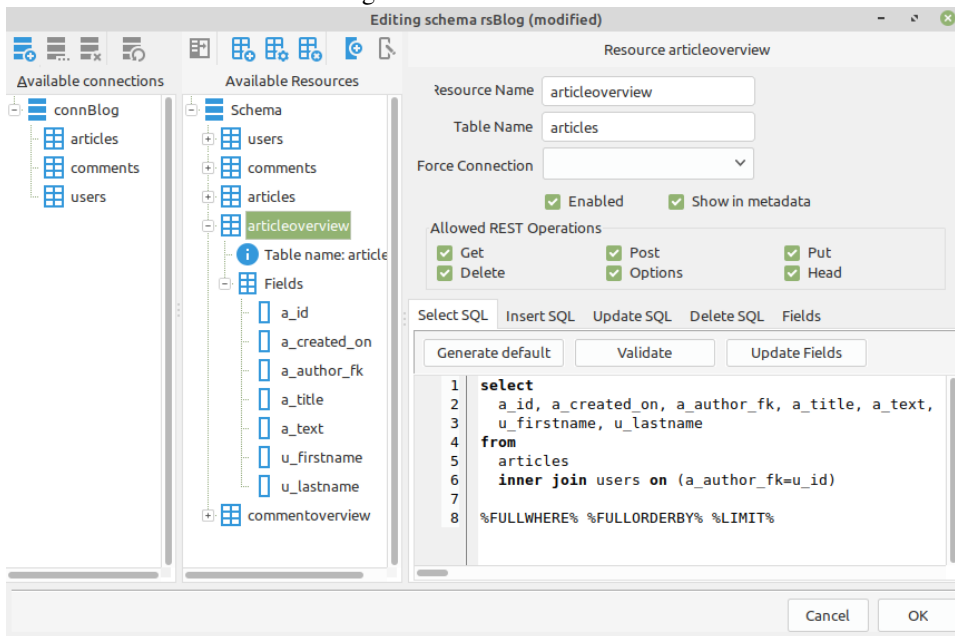
If the defaults are not to your liking, you can adapt the SQL, and you can ask the schema editor to generate a default SQL statement for you, which you can then modify at will. Alternatively, you can do everything manually: Press the 'New resource' button in the menu bar, specify a resource name and enter a table name and some SQL statements. You can set for each resource what REST operations are allowed. It is not recommended to disable the `OPTIONS` operation: this is used by the `CORS` support of the browser to discover (using a preflight request) whether a `POST` request can be sent or not.

Again, the schema editor can generate SQL update statements for you if you specified the table name (it is not mandatory to specify a table name).

Once the `Select SQL` statement is created, you can use the `Update Fields` button on the `Fields` tab to generate the field definitions. For each field in your resource, you can specify the following things:

1. The type of field that the client will receive - since the data is transferred over JSON or some other text format, logically the list of available client types is more limited than the available field types in `SQLDB`.
2. Whether the field must be specified when inserting/updating and if it is required or not.
3. Whether the field can be used in sorting or filtering operations. The client can request filters and sort order for its result, but for some fields this could result in lengthy operations for the DB server, so it makes sense to disallow it for some fields.

Figure 5: A custom schema



4. whether the field is part of the primary key.
5. The name of a database sequence that must be used to generate a field value.

Once the schema is defined, the server is almost ready to go.

If you set the 'Active' property of the `TSQLDBRestDispatcher` component to `True` then the component will register the HTTP routes in `fpweb`'s request router system. You can do this also manually with the `RegisterRoutes` method.

In some cases it may be necessary to register the routes manually: the order of route definitions matters and an automatic system such as the form loading mechanism does not necessarily respect the order you expect or need.

As an example, the `OnCreate` event of the `ServerModule` can have the following code:

```
procedure TServerModule.DataModuleCreate(Sender: TObject);
Var
  ClientDir : String;
begin
  ClientDir:=ExtractFilePath(ParamStr(0))+ '../client/';
  TSimpleFileModule.BaseDir:=ExpandFileName(ClientDir);
  TSimpleFileModule.RegisterDefaultRoute;
  TSimpleFileModule.IndexPageName:='index.html';
  mimetypes.LoadKnownTypes;
  rdBlog.RegisterRoutes;
end;
```

With all this, our REST server is ready to go. It can be tested easily with the browser or the `curl` or `wget` command-line utilities, by querying the following URL:

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

For example with `wget` you can do:

```
wget -q -O - http://localhost:8080/REST/metadata
```

If all went well, then you can expect output similar to the one shown in figure 6 on page 10. You can see that there are 2 parts in the JSON response: `metadata`, which contains the field definitions for the requested resource. `data` contains the actual data for the resource. The presence of the `metadata` in a REST response can be controlled by a property of the `TSQLDBRestDispatcher` component. You can also test actual data by testing one of the resources:

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

You can also try to post data with tools like Postman in the browser or the `curl` and `wget` command-line tools.

When the server tests work to your satisfaction, it is time to turn to the client application in the browser.

7 The client application

The `pas2js` client application to demonstrate the `TSQLDBRestBridge` is a simple site with 3 pages:

1. A login page. This is essentially the login page presented in the previous articles. On successful login, the list of articles is shown.
2. An overview page with the list of Blog articles. A logged in user can create a new article.
3. A detail page with the details of an article. When the user is logged in, (s)he can comment if (s)he is not the author of the article. The author of an article can change the text.

The login page will not be discussed as it is the same as in the previous articles, except to note that:

- In the case of a successful login, the display will switch to the list of articles.
- The 2-Factor authentication has been removed.

In the servermodule (which already holds the RPC Client from the Login example) we now add a `TSQLDBRestConnection`:

```
Property RestConnection : TSQLDBRestConnection Read FRestConnection;
```

And we set it up in the constructor:

```
FRestConnection:=TSQLDBRestConnection.create(Self);  
FRestConnection.BaseURL:=ServerURL+'REST/';
```

First we'll start with the article list page. We start obviously by drafting the HTML which will show the articles.

We need 4 things:

Figure 6: Metadata returned by the server

```
localhost:8080/REST/metadata
JSON Raw Data Headers
Save Copy Collapse All Expand All Filter JSON
▼ metaData:
  ▼ fields:
    ▼ 0:
      name: "name"
      type: "string"
      maxLen: 255
    ▶ 1: {...}
    ▶ 2: {...}
    ▶ 3: {...}
    ▶ 4: {...}
    ▶ 5: {...}
    ▶ 6: {...}
    ▶ 7: {...}
  ▼ data:
    ▼ 0:
      name: "users"
      schemaName: "rsBlog"
      Get: true
      Post: true
      Put: true
      Delete: true
      Options: true
      Head: true
    ▶ 1: {...}
    ▶ 2: {...}
    ▶ 3: {...}
    ▶ 4: {...}
```

1. A dropdown to filter on author.
2. A tag with a message to show if no articles are found.
3. A div in which to show the articles.
4. A button to allow a logged-in user to start a new article.

All these elements can be found in the following HTML:

```
<div class="section">
  <div id="divSearch" class="block">
    <div class="field is-horizontal">
      <div class="field-label is-normal">
        <label class="label" for="selAuthor">Select author</label>
      </div>
      <div class="field-body">
        <div class="field">
          <div class="control">
            <div class="select is-link">
              <select id="selAuthor">
            </select>
            </div> <!-- .select -->
          </div> <!-- .control -->
        </div> <!-- .field -->
      </div> <!-- .field-body -->
    </div> <!-- field -->
  </div> <!-- .block -->
  <div id="divNoArticles" class="block">
    <div class="notification is-warning is-light">
      Alas, no articles were found...
    </div>
  </div>
  <div id="divArticleList" class="block">
  </div>
  <div id="divNewArticle" class="block is-hidden">
    <a href="#/article/new" class="button is-primary">New article</a>
  </div>
</div>
```

The above is not a lot of HTML, but it is sufficient to do what is needed.

With the IDE wizard to create a 'form' class from this HTML we can now create a form definition, which we'll name `TArticleListForm`, and which will be a descendent of `TBaseForm` so we can register a route.

We need to override the `FormRoutes` and `FormHTMLName` to match the URL we want and the `articles.html` filename from which we generated the class:

```
class function TArticleListForm.FormRoutes: TStringDynArray;

begin
  Result:=['/articles/'];
end;

class function TArticleListForm.FormHTMLFileName: String;
```

```
begin
  Result:='articles.html';
end;
```

In the constructor we call `SetupDatasets` which - not surprisingly - will set up the datasets we need in this form.

We need 2 datasets: one for the authors (`rdsAuthors`) to fill the dropdown with authors, and one for displaying the blog articles (`rdsArticles`).

Setup of these datasets is straightforward:

```
procedure TArticleListForm.SetupDatasets;

begin
  rdsAuthors:=TSQLDBRestDataset.Create(Self);
  With rdsAuthors do
    begin
      Connection:=Server.RestConnection;
      ResourceName:='users';
      AfterOpen:=@DoOpenAuthors;
      Load;
    end;
```

The `Connection` and `ResourceName` are set up, and an `AfterOpen` event is set up.

The last step is calling `Load`. This will fetch data from the server, and when the data is returned, it will set it up and then call `Open`.

Why not do this using the `Open` call and instead introduce a new call `Load`? Loading data from the server is an asynchronous process, and by putting this in a separate call, this is made explicit.

The `Open` method of `TDataset` is expected to work synchronously: when it returns, the data is expected to be there, enabling code flow like this:

```
With aDataset do
  begin
    Open;
    While not EOF do
      begin
        // Do stuff
      Next;
    end;
  end;
```

Because loading data works asynchronously, this flow would not be possible if the implementation fetched data in the `Open`.

Note that the `Open` method is still functional in the Pas2JS `TDataset`: For a JSON-Dataset you can fetch data using another mechanism, set the data in the `Rows` property and call `Open`.

The second dataset fetches the articles:

```
rdsArticles:=TSQLDBRestDataset.Create(Self);
With rdsArticles do
  begin
    Connection:=Server.RestConnection;
```

```

ResourceName:='articleoverview';
AfterOpen:=@DoOpenArticles;
Params.AddParam('sort',true).AsString:='a_created_on desc';
Params.AddParam('a_author_fk',False);
end;
dsArticles:=TDataSource.Create(Self);
dsArticles.Dataset:=rdsArticles;
rdsArticles.Load;
end;

```

Here a little more is done: a datasource is attached to the dataset - it will become clear why shortly - and 2 parameters are set up. The parameters of a TSQLDBRestDataset are attached to the URL when the data is fetched: The above will result in an URL like this:

```
http://localhost:8080/REST/articleoverview?sort=a_createdon%20desc
```

Note that the parameter a_author_fk is disabled and is not visible in the URL.

When the parameter a_author_fk is enabled and set to value 3, the URL will look like this:

```
http://localhost:8080/REST/articleoverview?sort=a_createdon%20desc&a_author_fk=3
```

The DoOpenAuthors event handler is called when the authors have been fetched and the dataset is opened. In it, we will fill the dropdown with a list of authors the user can choose from.

This routine simply loops over the available records and constructs the HTML needed to fill the SELECT html element. It uses a template which is filled with a Format statement.

```

procedure TArticleListForm.DoOpenAuthors(DataSet: TDataSet);
Const
  ItemTemplate = '<option value="%d">%s</option>'+sLineBreak;
var
  aHTML,aAuthor : String;
  FID,FFirstName,fLastname : TField;
begin
  aHTML:=Format(ItemTemplate,[-1,'All authors']);
  FID:=Dataset.FieldByName('u_id');
  FFirstName:=Dataset.FieldByName('u_firstname');
  FLastName:=Dataset.FieldByName('u_lastname');
  While not Dataset.EOF do
    begin
      aAuthor:=FFirstName.AsString+' '+fLastname.AsString;
      if (FID.AsInteger=Server.UserInfo.ID) then
        aAuthor:='You ('+aAuthor+')';
      aHTML:=aHTML+Format(ItemTemplate,[FID.AsInteger,aAuthor]);
      Dataset.Next;
    end;
  selAuthor.InnerHTML:=aHTML;
end;

```

There is little magical about this routine. If you prefer, you can of course also create the HTML elements in code:

```

procedure TArticleListForm.DoOpenAuthors(DataSet: TDataSet);

var
  aAuthor : String;
  FID,FFirstName,fLastname : TField;

  Function MakeOption(aID : NativeInt;
    const aName : String) : TJSHTMLOptionElement;

begin
  Result:=TJSHTMLOptionElement(Document.createElement('option'));
  Result.innerText:=aName;
  Result.value:=IntToStr(aID);
end;

begin
  selAuthor.InnerHTML:='';
  SelAuthor.append(MakeOption(-1,'All authors'));
  FID:=Dataset.FieldByName('u_id');
  FFirstName:=Dataset.FieldByName('u_firstname');
  FLastName:=Dataset.FieldByName('u_lastname');
  While not Dataset.EOF do
    begin
      aAuthor:=FFirstName.AsString+' '+fLastname.AsString;
      if (FID.AsInteger=Server.UserInfo.ID) then
        aAuthor:=' You ('+aAuthor+')';
      selAuthor.append(MakeOption(FID.AsInteger,aAuthor));
      Dataset.Next;
    end;
  end;
end;

```

The routines do not differ much, but the second routine is a little safer, since the compiler will check the statements: all is done using the declared Javascript HTML classes.

The list of articles can be constructed in a similar way, but we will do things differently. The dbbwebwidgets unit contains a component (TDBLoopTemplateWidget) that, given a datasource, will construct a HTML fragment by looping over the records in a dataset and for every record fill a template with the values of the fields, and append it to the html. Basically, it does the loop above. No coding is involved except setting up the component. This is of course why the articles dataset has a datasource attached to it in the code above.

The TDBLoopTemplateWidget is set up in the SetupTemplates method:

```

procedure TArticleListForm.SetupTemplates;

Const
  articleTemplate =
    '<article class="media">' +
    '  <div class="media-content">' +
    '    <div class="content">' +
    '      <p>' +
    '        <strong>{{u_firstname}} {{u_lastname}}</strong>' +
    '        <small>{{a_created_on}}</small>' +
    '      <br>' +
    '      <a href="#/article/{{a_id}}">' +

```

```

    '          {{a_title}}' +
    '          </a>' +
    '        </p>' +
    '      </div>' +
    ' </div>' +
    '</article>';

```

```

begin
  ltwArticleList:=TDBLoopTemplateWidget.Create(Self);
  ltwArticleList.Datasource:=dsArticles;
  ltwArticleList.ItemTemplate:=articleTemplate;
  ltwArticleList.ParentID:=divArticleList.ID;
end;

```

The method is called from the constructor of the form. Since it is data-aware, it will do its work as soon as the dataset is opened. The template is quite simple, it uses the bulma media classes, which are exactly meant for this kind of content.:

```

<article class="media">
  <div class="media-content">
    <div class="content">
      <p>
        <strong>{{u_firstname}} {{u_lastname}}</strong>
        <small>{{a_created_on}}</small>
        <br>
        <a href="#/article/{{a_id}}">
          {{a_title}}
        </a>
      </p>
    </div>
  </div>
</article>

```

The placeholders in double braces {{ }} will be replaced with the contents of the fields from the dataset. There is also an event that can be used to get values for placeholders which are not the name of a field, so basically any content can be inserted in the template.

The result of this can be seen in figure 7 on page 16

When the user selects an author to filter the list of articles, the 'change' event is triggered by the HTML. The following code is inserted in the event handler that was automatically generated from the html.

The code makes clear why we added a disabled parameter to the dataset when creating the rdsArticles dataset, because at this point we enable the parameter depending on whether an author was selected, or the 'all authors' option was selected (this option was given the value -1)

```

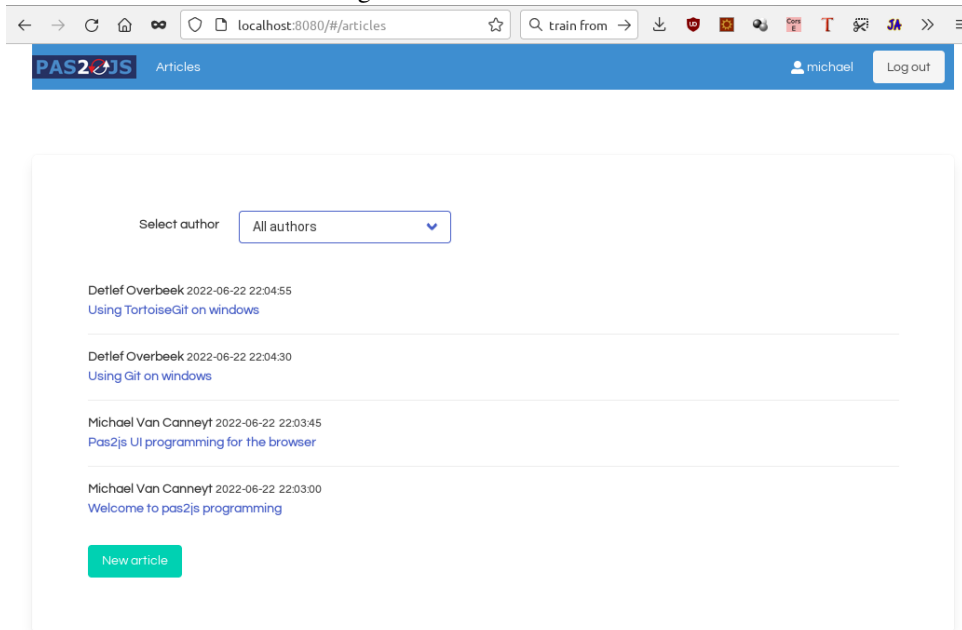
procedure TArticleListForm.DoSelectAuthor(Event: TJSEvent);

Var
  aAuthorID : Integer;

begin
  aAuthorID:=StrToIntDef(selAuthor.value,-1);
  With rdsArticles.ParamByName('a_author_fk') do

```

Figure 7: The full article list



```
begin
  AsInteger:=aAuthorID;
  Enabled:=(aAuthorID<>-1);
end;
rdsArticles.Close;
rdsArticles.Load;
end;
```

After setting the parameters, the dataset is closed and reopened. Since the `TDBLoopTemplateWidget` component is data-aware, it will automatically re-render the list of articles.

Finally, in the `AfterOpen` event of the articles, we show or hide the 'no articles' message on our page depending on whether the dataset is empty or not:

```
procedure TArticleListForm.DoOpenArticles(DataSet: TDataSet);
begin
  if DataSet.IsEmpty then
    divNoArticles.classList.Remove('is-hidden')
  else
    divNoArticles.classList.Add('is-hidden');
end;
```

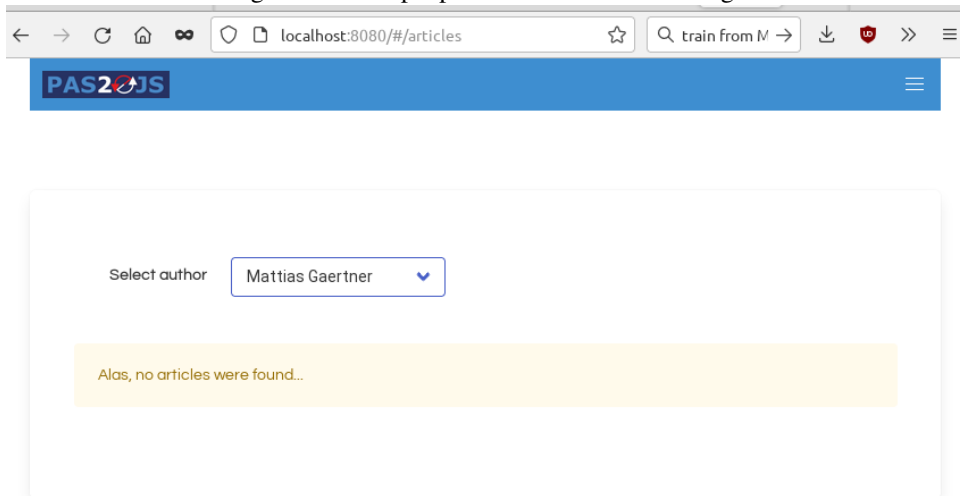
The result of this little routine can be seen in figure 8 on page 17.

At the bottom of the page there is a button to create a new article. This button should only be visible when the user is logged in. This is done in the constructor of the page:

```
constructor TArticleListForm.create(aOwner : TComponent);

begin
  inherited;
  BindElements;
```


Figure 8: Some people do not like to write blogs



```
BindElementEvents;  
SetupDatasets;  
SetupTemplates;  
if Server.UserInfo.ID>0 then  
  divNewArticle.classList.remove('is-hidden');  
end;
```

Looking at the HTML template for an article, we can see that the article title functions as a hyperlink: when the user clicks on it, the url

```
#/article/{{a_id}}
```

is opened.

8 Editing data

This is the second page that we add to our application: the article detail page. The page is a little special in the sense that it must handle 2 cases: The logged-in user is the author of the article (in which case he can edit it) or the user can only see the article. There is also the possibility that the asked-for article no longer exists. The page must also handle that situation and deal with it gracefully.

The top of the page contains a block that is shown if the article does not exist. It is initially hidden. Below it is the section that either displays the title and author, or allows to edit the title of the author: which part will be shown is determined in code.

```
<div class="block is-hidden" id="divNoArticles">  
  <h1 class="title is-3">Article not found</h1>  
  <p>You can return to <a href="#/articles">  
    the list of articles</a>  
    and select another article.</p>  
</div>
```

```

<!-- title and date etc. -->
<div class="block">
  <div id="divTitle" >
    <h1 class="title is-3" id="lblTitle"></h1>
    <h5 class="title is-5">
      <strong id="lblAuthor">Author</strong>
      <small id="lblDate">Date</small>
    </h5>
  </div>
  <div id="divEditTitle"
    class="field is-horizontal is-hidden">
    <div class="field-label is-normal">
      <label class="label" for="edtTitle">Title</label>
    </div>
    <div class="field-body">
      <div class="field">
        <div class="control">
          <input id="edtTitle"
            type="text"
            class="input"
            placeholder="Type your title here">
        </div> <!-- .control -->
      </div> <!-- .field -->
    </div> <!-- .field-body -->
  </div> <!-- field -->
</div> <!-- .block -->

```

After the title, there is a block where the article will be shown, together with a 'Save' button, which is initially hidden. Note that for the article content, there are no separate edit and display blocks. The reason for this will become clear soon.

```

  <div id="divArticleBlock"
    class="block"
    style="min-height: 40vh">
    <div id="divArticle" class="content">
      <h5>This will be a long rant</h5>
      But it will end well...
    </div>
  </div> <!-- .block -->

<div class="block">
  <button id="btnSave"
    class="button is-primary is-hidden"
    _click_="HandleSavePost">Save post</button>
</div>

```

The last part of the page is where comments can be shown, and where a logged-in user can add a comment:

```

<div id="divComments">
  <div class="block">
    <h1 class="is-title is-5">Comments</h1>
  </div>
  <div id="divExistingComments" class="block">

```

```

</div>
<div class="divAddComment">
  <div>
    <div id="editcomment" style="min-height: 10em;"></div>
  </div>
  <div>
    <button id="btnAddComment"
      _click_="HandleSaveComment"
      class="button is-primary">Add comment</button>
  </div>
</div>
</div>

```

The list of comments will again be rendered by a `TDBLoopTemplateWidget` component.

Again, we use the 'File-New' wizard to create a class definition for this HTML file. We start by correcting the `formroutes` method so it returns a route that has a parameter for the article ID.

```

class function TArticleForm.FormRoutes: TStringDynArray;
begin
  Result:=['/article/:ArticleID'];
end;

```

This route matches the URL that was specified in the article overview.

In the constructor, we again add code to set up the datasets and the `TDBLoopTemplateWidget` component:

```

procedure TArticleForm.SetupDatasets;

begin
  rdsArticle:=TSQLDBRestDataset.Create(Self);
  rdsArticle.Connection:=Server.RestConnection;
  rdsArticle.ResourceName:='articles';
  rdsArticle.AfterOpen:=@DoAfterArticleOpen;
  rdsComments:=TSQLDBRestDataset.Create(Self);
  rdsComments.Connection:=Server.RestConnection;
  rdsComments.ResourceName:='commentoverview';
  With rdsComments.Params.AddParam('c_article_fk') do
    begin
      DataType:=ftLargeInt;
      Enabled:=true;
    end;
  dsComments:=TDatasource.Create(Self);
  dsComments.Dataset:=rdsComments;
end;

```

Note that the data is not yet loaded in this routine. Note also the `datasource`, needed to render the comments.

The code to set up the comment renderer is very similar to the one for the article list:

```

procedure TArticleForm.SetupTemplates;

```

```

Const
  CommentTemplate =
    '<article class="media">' +
    '  <div class="media-content">' +
    '    <div class="content">' +
    '      <p>' +
    '        <strong>{{u_firstname}} {{u_lastname}}</strong>' +
    '        <small>{{c_created_on}}</small>' +
    '        <br>' +
    '        {{c_text}}' +
    '      </p>' +
    '    </div>' +
    '  </div>' +
    '</article>';

begin
  ltwComments:=TDBLoopTemplateWidget.Create(Self);
  ltwComments.Datasource:=dsComments;
  ltwComments.ItemTemplate:=CommentTemplate;
  ltwComments.ParentID:=divExistingComments.id;
end;

```

When the web router creates the form and shows it, it will call the 'ShowRoute' method of the form. We must override this method and add code to handle the value of the ArticleID parameter: After recording the value of the parameter, we must distinguish between the value New and a numerical value. In the former case, we must append to the dataset. In the latter case, we must fetch the data for the article with the given ID.

```

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

Var
  aID : String;

begin
  Inherited;
  aID:=aParams.Values['ArticleID'];
  if SameText(aID,'New') then
    begin
      FArticleID:=-1;
      rdsArticle.Params.AddParam('a_author_fk',true).asInteger:=-1;
    end
  else
    begin
      FArticleID:=StrToIntDef(aID,-1);
      rdsArticle.ResourceID:=IntToStr(FArticleID);
    end;
  rdsArticle.Load();
  if (FArticleID>0) then
    begin
      rdsComments.Params[0].asLargeInt:=FArticleID;
      rdsComments.Load();
    end
  end

```

```
end;
```

When loading an existing article, we can simply set the `ResourceID` to the ID of the requested article. The `ResourceID` value will be appended to the URL.

When creating a new article, we must also load the dataset, because during the load operation, the structure of the dataset (which fields to create) is fetched from the server in the metadata property of the returned data. But we cannot set `resourceID` to -1 or any other non-existing value, because doing so will result in a 404 error code: asking for a non-existent resource results in a 404, and the dataset will not be opened, instead a load error is reported. So instead we load using a filter that is guaranteed to give an empty result. (there is another approach possible, which will be discussed in a future contribution)

The last piece of code simply opens the comments dataset, so the comments will be rendered.

The logic of this page continues in the `AfterOpen` event of the article dataset. Here there are 4 possibilities

1. An article was requested but not found.
2. A new article was requested (and no data is incoming) so we must edit it.
3. There is data for an article, and we must allow to edit it.
4. There is data for an article, and we must simply display it.

We distinguish these cases by showing and hiding some parts of the page. We start with the case that the dataset is empty. What to do depends on whether an article is expected (the `FArticleID` is positive) or not:

```
procedure TArticleForm.DoAfterArticleOpen(DataSet: TDataSet);

Var
  IsEdit : Boolean;
  S : String;

begin
  if DataSet.IsEmpty then
  begin
    if FarticleID>0 then
    begin
      HideEl(divArticle);
      HideEl(divTitle);
      HideEl(divEditTitle);
      HideEl(divComments);
      ShowEl(divNoArticles);
    end
  else
  begin
    // Appending, show known data:
    lblAuthor.InnerText:=Server.UserInfo.Login;
    S:=FormatDateTime('yyyy-mm-dd hh:nn',Now);
    lblDate.InnerText:=S;
    edtTitle.value:='';
    divArticle.InnerHTML:='';
    SetupForEdit(True);
  end;
end;
```

```

        Dataset.Append;
    end;
end

```

The first part of the `if` statement treats the case when an article was asked, but the server does not have it. The second part handles the case when we need to append. The `SetupForEdit` takes care of showing/hiding the edit controls.

In case there is data, we must decide whether we allow the user to edit:

```

else
begin
    isEdit:=Dataset.FieldName('a_author_fk').AsInteger=
        Server.UserInfo.ID;
    SetupForEdit(isEdit);
    ShowData(Dataset);
    if IsEdit then
        Dataset.Edit;
    end;
    With Dataset.FieldName('a_id') do
        ProviderFlags:=ProviderFlags+[pfInKey];
    end;
end;

```

Note that when editing is allowed, the dataset is put in edit or insert mode.

The last statement is needed to set up the key field of the dataset: without this, the client will not know which field is the key field. (a change is planned in `SQLDBRESTBridge` which removes the need for this statement)

The data is actually shown in the `showdata` method. It simply copies the data from the dataset to the various html tags:

```

procedure TArticleForm.ShowData(Dataset : TDataset);

var
    S : string;

begin
    With Dataset do
        begin
            S:=FieldName('u_firstname').AsString + ' '
                + FieldName('u_lastname').AsString;
            lblAuthor.InnerText:=S;
            S:=FormatDateTime('yyyy-mm-dd hh:nn',
                FieldName('a_created_on').AsDateTime);
            lblDate.InnerText:=S;
            S:=FieldName('a_title').AsString;
            edtTitle.value:=S;
            lblTitle.InnerText:=S;
            S:=FieldName('a_text').AsString;
            divArticle.InnerHTML:=S;
        end;
    end;
end;

```

The last piece in the puzzle is the `SetupForEdit` routine, which simply shows or hides/disables some elements.

```

procedure TArticleForm.SetupForEdit(IsEditing : Boolean);
begin
  if IsEditing then
    begin
      HideEl (divComments);
      HideEl (divTitle);
      ShowEl (divEditTitle);
      tinyEditor.transformToEditor (divArticle);
      ShowEl (btnSave);
    end
  else
    begin
      HideEl (divEditTitle);
      ShowEl (divComments);
      btnAddComment.disabled:= (Server.UserInfo.ID<=0);
      if not btnAddComment.disabled then
        begin
          editcomment.style.setProperty('min-height', '10em;');
          tinyEditor.transformToEditor (editcomment);
        end;
      end;
    end;
end;

```

Note the following statement:

```
tinyEditor.transformToEditor (divArticle);
```

This activates the TinyEditor Javascript component. This is a small javascript function that transforms a html element into an editor with basic editing capabilities such as bulleted lists, headings, indent etc. It can be downloaded from github:

<https://github.com/fvilers/tiny-editor>

With this, we're all set to display an article, as can be seen in figure 9 on page 24.

When editing, the save button must save the data, this is handled in the 'click' event handler of the button. Note that the dataset was already set in edit or insert mode:

```

procedure TArticleForm.HandleSavePost (Event : TJSEvent);
begin
  SaveData (rdsArticle);
  rdsArticle.Post;
  rdsArticle.ApplyUpdates;
end;

```

There is nothing special about this: SaveData will transfer the contents of the HTML tags to the dataset. The Post and ApplyUpdates have the same purpose as in any other dataset. For completeness, we present here SaveData

```

procedure TArticleForm.SaveData (Dataset : TDataset);

begin
  With Dataset do
    begin

```

Figure 9: A displayed article and comments

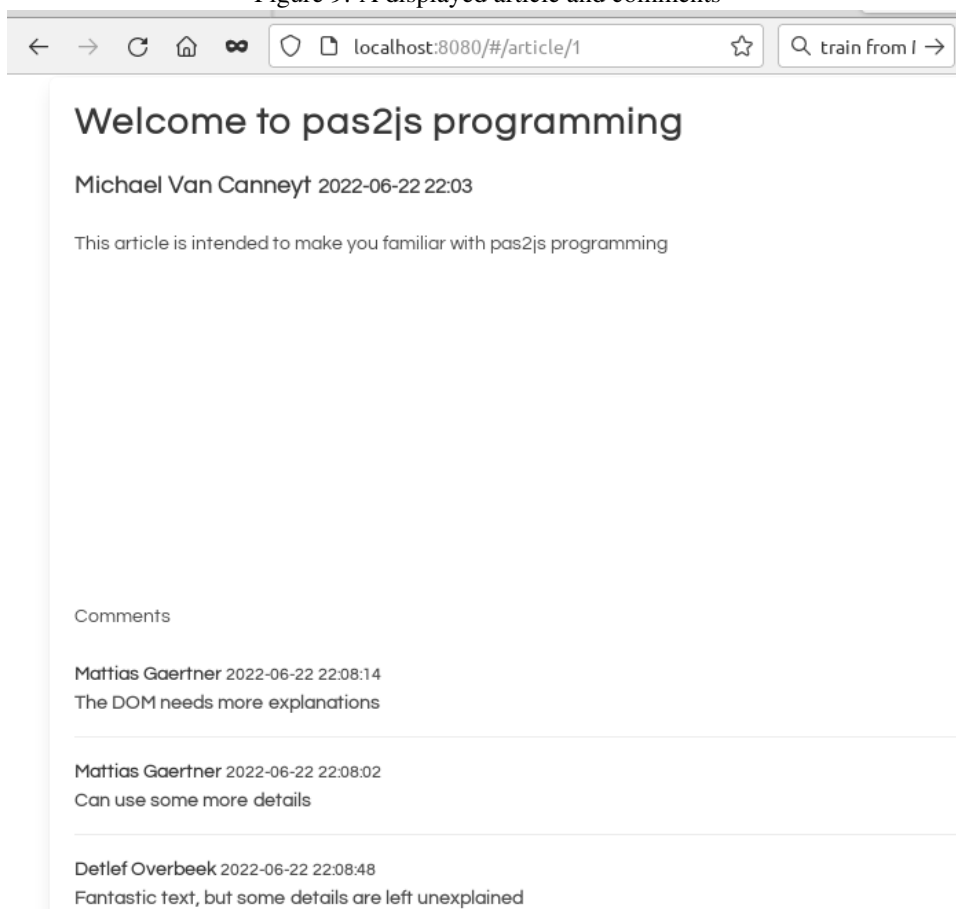
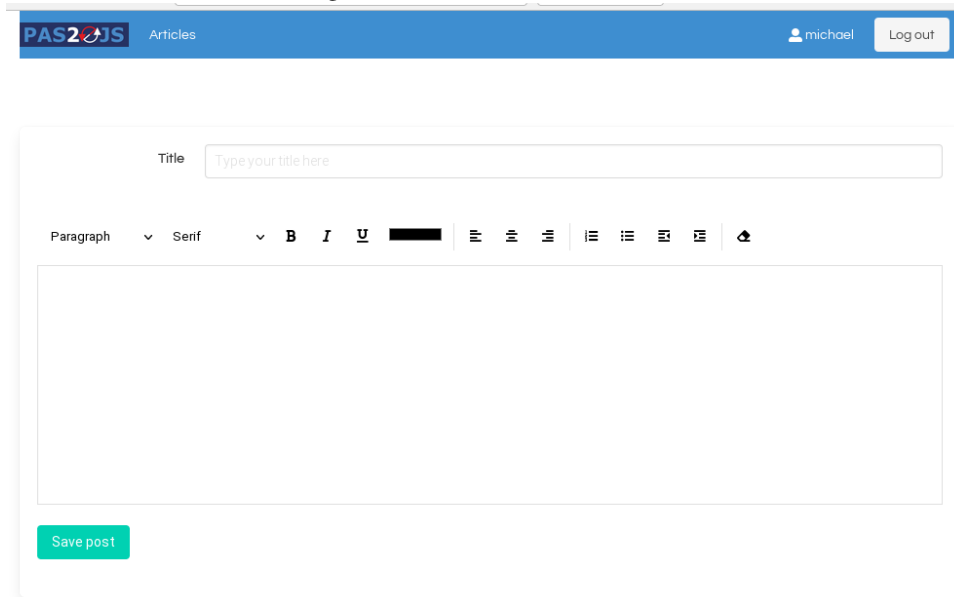


Figure 10: An article in edit mode



```
FieldByName('a_created_on').AsDateTime:=Now;  
FieldByName('a_text').AsString:=divArticle.InnerHTML;  
FieldByName('a_title').AsString:=edtTitle.Value;  
FieldByName('a_author_fk').AsInteger:=Server.UserInfo.ID;  
end;  
end;
```

The `tinyeditor` component directly edits the inner html of the tag it was assigned to, so the actual HTML content is directly available for us.

Similarly, the button to save a comment does the same:

```
procedure TArticleForm.HandleSaveComment(Event: TJSEvent);  
begin  
  With rdsComments do  
    begin  
      Append;  
      FieldByName('c_text').AsString:=editcomment.innerHTML;  
      FieldByName('c_createdon').AsDatetime:=Now;  
      FieldByName('c_author_fk').AsDateTime:=Server.UserInfo.ID;  
      Post;  
      ApplyUpdates;  
    end;  
end;
```

And that's it. An article in edit mode is shown in figure 10 on page 25

9 Conclusion

In this article we've shown that displaying and editing data is not different in a `pas2js` application than it is in a classic 3-tier desktop application. We didn't pay a lot of attention

to error handling and did not handle security at all. Also, in this article, we've done everything manually: Create the dataset and other components, transfer data between HTML and dataset. However, Pas2js is progressing and we can do better: drag and drop of components, data-aware components that automatically transfer data between data and HTML elements. This is the subject of a new contribution.