Blaise Pascal Magazine Library

Michaël Van Canneyt

June 7, 2023

Abstract

Blaise Pascal Magazine offers subscribers a library : a collection of all issues available till now. In this article we show how the PDF indexer application presented in the previous articles about indexing PDF files will be used to rewrite and enhance the Blaise Pascal Magazine library.

1 Introduction

In several earlier contributions, we showed how to show a PDF file in an offline Pas2JS application, and how to index PDF files and use that index to search and download PDF files. In this article, we'll show how to combine all these techniques to rewrite the Blaise Pascal Magazine library as a Pas2js application that works both offline and online.

The new edition of the Blaise Pascal Magazine library will need to have the following features:

- It must work as a local application, distributed on a USB stick.
- It must work as a web application, deployed on the Blaise Pascal Magazine website.
- When working locally, issues must be loaded from the local disk: usually a USB.
- The issues must be searchable.
- When working locally, and no internet connection is available, then a (limited) search must be performed locally in the list of articles. if an internet connection is available, the application must be able to search globally and download a PDF if needed.
- PDF Downloads are limited to the downloads purchased by the magazine subscriber.
- The user can enter an issue number, and the first page of the issue will be displayed.

All of the techniques needed to satisfy these requirements have been presented in previous articles. In this article we bring everything together: this will require some refactoring of the code presented in the previous articles.

Since we'll be needing a server part and a client part, we'll start by discussing the code changes needed on the server.

2 Adding security to the server

In previous articles about indexing PDF files, we implemented a simple mechanism to download issues from the server in addition to the search mechanism and word list mechanism. Any issues could be downloaded from the server. This needs to be modified so the user can only download what he has subscribed for.

This means we need to add a login mechanism: the server needs to know who is attempting to download an issue. We'll also need a mechanism to determine what issues a user is allowed to download. In order to implement this, we need to extend the database with this information.

First, we'll need a list of users: a table with at least a username and a password. The SQL to create such a table (let's name it `users`)can for example look like this in Postgres:

```
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 unique index udx_users on users (u_login);
```

The fieldnames speak for themselves, and the index is there to make sure every login is unique.

Similarly, we'll need a table with all available issues. We could take the articles table presented in the previous articles, but every issue occurs more than once in that table, so it is difficult to create a foreign key on this table for referential integrity.

Instead, we create a new table, aptly named `issues`:

```
create sequence seq_issues;

create table issues (
  i_id bigint not null default nextval('seq_issues'),
  i_issue varchar(10) NOT NULL,
  i_filename varchar(127) NOT NULL,
  constraint pk_issues primary key (i_id)
);
```

The `i_issue` field is not a number to accomodate for double issues, so we can support a notation as 81_82 for the combined issues 81 and 82.

To know which issues a given user can access, we need a third table (named `userissues`):

```
create sequence seq_userissues;
create table userissues (
  ui_id bigint not null default nextval('seq_userissues'),
  ui_user_fk bigint NOT NULL,
```

```
  ui_issue_fk bigint NOT NULL,
  constraint pk_userissues primary key (ui_id)
);
```

For each user and each issue that user can access, a record is inserted in this table, with a link to the `users` table and `issues` table.

For referential integrity, we enforce a foreign key to these tables;

```
alter table userissues add constraint fk_userissues_users
    foreign key (ui_user_fk) references users(u_id) on delete cascade;
alter table userissues add constraint fk_userissues_issues
    foreign key (ui_issue_fk) references issues(i_id) on delete cascade;
```

Additionally, we make sure there is only one record for each user - issue combination.

```
create unique index udx_userissue on userissues (ui_user_fk,ui_issue_fk);
```

Armed with these tables, we can now implement some security mechanisms. We will not describe how data gets into these tables: we'll assume the tables have been filled through some external mechanism, through some link with the subscription system.

The first thing is to implement a kind of login routine: we implement a simple HTTP endpoint which receives a JSON with a username and password (this is not a JSON-RPC mechanism):

```
{
   "username": "michael",
   "password": "verysecret"

}
```

The server will respond with a token (a simple GUID) :

```
{
  "token" : "{F5A07A5B-5184-4256-8EE6-20E2DE987AF5}",
  "expires" : "2023-06-03T17:12:09.489Z"
}
```

This token can be passed to the server when a PDF is downloaded: the server will then verify this token and allow the download if it is valid.

The tokens are also stored in the database, in a table called `tokens`:

```
create sequence seqTokens;

CREATE TABLE tokens
(
  tk_id bigint NOT NULL DEFAULT nextval('seqTokens'::regclass),
  tk_token character varying(38) NOT NULL
    DEFAULT (upper((('{'::text || uuid_generate_v4()) || '}'::text)))::character varying(38),
  tk_user_fk bigint NOT NULL,
  tk_expires_on timestamp without time zone NOT NULL
    DEFAULT (now() + '00:30:00'::interval),
  CONSTRAINT pktokens PRIMARY KEY (tk_id)
```

```
);

create index idx_token_expires on tokens(tk_expires_on);
create unique index udx_tokens on tokens(tk_token);
```

The `tk_expires_on` is by default filled with a timestamp 30 minutes from the time of insert of the record. In effect, the token will be valid for 30 minutes.

The `uuid_generate_v4` function is part of a Postgres extension and generates a GUID, which will serve as our unique session token. The extension needs to be activated with the following SQL statement:

```
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
```

To implement this scheme, we create a new class `TSessionManager`:

```
TSessionManager = Class(TComponent)
Public
  constructor Create(aOwner : TComponent); override;
  destructor destroy; override;
  // Public API
  Procedure GetToken(aRequest : TRequest; aResponse : TResponse);
  Function CheckToken(const aToken : string) : int64;
  function CheckFileAllowed(aUserID : int64; const aFileName : string) : Boolean;
  Property DB : TSQLConnection Read FDB Write FDB;
end;
```

The `GetToken` call is registered as the handler for the `/token` login during application startup:

```
aSession:=TSessionmanager.Create(Nil);
aSession.DB:=aSearch.Connection;
HTTPRouter.RegisterRoute('/token',@aSession.GetToken,False);
```

The `DB` property is the SQL connection used by the search class: it was set up in code of the previous articles and is simply reused here.

The `GetToken` call is pretty straightforward. It starts with checking for a `CORS` request, in a similar manner to the way it was done in the PDF file download in the previous article. This is needed, because when the client application is started from a local disk, the origin differs from the server.

When the `CORS` request indicates that all is well, the code starts by decoding the request payload as a JSON structure. If something goes wrong then the `ReportInvalidParam` routine is used to report a HTTP 400 return code. If the JSON is decoded correctly, the username and password are extracted. Again a check is done to see if values have been passed for both username and password. If not a HTTP 400 error is again reports:

```
procedure TSessionManager.GetToken(aRequest: TRequest; aResponse: TResponse);

Var
  Req,Resp : TJSONData;
  Obj : TJSONObject absolute Req;
  token,UserName,UserPW : String;
  aExpiresOn : TDatetime;
```

```
begin
  if FCors.HandleRequest(aRequest,aResponse,[hcDetect,hcSend]) then exit;
  Req:=Nil;
  Resp:=Nil;
  try
    Req:=GetJSON(aRequest.Content);
    if not (Req is TJSONObject) then
      ReportInvalidParam(aResponse)
    else
      begin
      userName:=Obj.Get('username','');
      userPW:=Obj.Get('password','');
      if (UserName='') or (userPW='') then
        ReportInvalidParam(aResponse)
      else
        begin
        token:=ValidateUser(UserName,UserPW,aExpiresOn);
        if Token='' then
          ReportForbidden(aResponse)
        else
          begin
          Resp:=TJSONObject.Create([
              'token',token,
              'expires',DateToISO8601(aExpiresOn)
          ]);
          SendJSON(aResponse,200,'OK',Resp);
          end;
        end;
      end;
  except
    on E : Exception do
      ReportException(aResponse,E);
  end;
  Resp.Free;
  Req.Free;
end;
```

The username and password are validated using the `ValidateUser` call: it will return a token if the username/password pair was valid. If the token is empty, it means the combination was not valid and an error is reported. Lastly, if we received a token and expiration date, we send both back to the client in a JSON structure using SendJSON:

```
procedure TSessionManager.SendJSON(aResponse : TResponse;aCode : Integer; aText : String; aJS

begin
  if aResponse.ContentSent then
    exit;
  aResponse.Code:=aCode;
  aResponse.CodeText:=aText;
  aResponse.ContentType:='application/json';
  aResponse.Content:=aJSON.FormatJSON();
  aResponse.SendContent;
```

```
end;
```

The same SendJSON method is used in e.g. the ReportInvalidParam method:

```
  procedure TSessionManager.ReportInvalidparam(aResponse: TResponse);
Var
  J : TJSONObject;

begin
  J:=TJSONObject.Create(['message','need username/password']);
  try
    SendJSON(aResponse,400,'INVALID PARAM',J);
  finally
    J.Free;
  end;
end;
```

The ValidateUser method is again very straightforward: the only noteworthy thing is that the password is stored encrypted in the database.

We use the native Postgres database cryptographic mechanisms for this: the crypt function encrypts a value using a salt, and the same function can be used to insert the data.

The crypto functionality must be enabled with the following SQL statement:

```
CREATE EXTENSION pgcrypto;
```

Using this function, the SQL statement to verify a user password is as follows:

```
SELECT
  U_password=crypt(:password,U_password) as PasswordOK, *
from
  Users
where
  (U_login=:login);
```

If the user login is not found, the query will return no records. If the user is found, then there will be a single record (because the login is unique), and the PasswordOK field will be True if the password passed in parameter:password matches the one stored in the database, and it will be False if not.

Using this SQL statement we can easily create the ValidateUser call. It starts out by creating a database transaction: every operation is done in its own transaction. After the transaction is created, it is used to create a TSQLQuery dataset and run the SQL statement (The CreateTransaction and CreateQuery) functions are trivial and will not be presented here):

```
function TSessionManager.ValidateUser(
    const aUser, aPassword: String;
    out aExpires: TDateTime): String;

Const
  SQLSelectUser =
    'SELECT  U_password=crypt(:password,U_password) as PasswordOK,*' +
    'from Users '+
```

```
    ' where (U_login=:login)';

Var
  Tr : TSQLTransaction;
  Qry : TSQLQuery;
  Res,OK : Boolean;
  aID : Int64;

begin
  OK:=False;
  Result:='';
  qry:=Nil;
  Tr:=CreateTransaction;
  try
    Qry:=CreateQuery(SQLSelectUser,['LOGIN',aUser,'PASSWORD',aPassword],Tr);
    Qry.Open;
    // If we have a user and the password matches
    Res:=(Not Qry.IsEmpty) and (Qry.FieldByName('PasswordOK').AsBoolean);
    aID:=Qry.FieldByName('u_id').AsLargeInt;
    if Res then
      // We get a token
      Result:=CreateToken(aID,aExpires,Qry.SQLTransaction);
    Tr.Commit;
    OK:=True;
  finally
    if not OK then
      Tr.Rollback;
    ReleaseQuery(Qry,Tr);
  end;
end;
```

If the user is verified, then the user ID and transaction are passed to `CreateToken`, which will create a new token. Note that the token is created in the same transaction:

The `CreateToken` function is again quite simple, it makes use of the fact that the 'default' values in the table column definitions create usable values, and simply returns the values created by Postgres.

```
function TSessionManager.CreateToken(aUser: Int64;
                                     out Expires: TDateTime;
                                     aTransaction : TSQLTransaction): String;

Const
  SQLInsert =
    'insert into tokens (tk_user_fk) values (:USER) ' +
    'returning tk_token, tk_expires_on;';

Var
  Qry : TSQLQuery;
  OK : Boolean;

begin
  OK:=False;
  Qry:=CreateQuery(SQLInsert,['USER',aUser],aTransaction);
```

```
  try
    Qry.Open;
    if Qry.IsEmpty then
      DatabaseError(SErrFailedToCreateToken, self);
    Result:=Qry.FieldByName('tk_token').AsString;
    Expires:=Qry.FieldByName('tk_expires_on').AsDateTime;
    OK:=True;
  finally
    if Not OK then
      Qry.SQLTransaction.RollBack;
    ReleaseQuery(Qry);
  end;
end;
```

With these routines we have created a HTTP endpoint that can be used in the application to ask for a token.

3 Securing the download

When the user wants to downloads a PDF file, the token must be supplied so the server can verify who is making the download, and whether the user is allowed to download the requested PDF file. The token can be specified in one of 2 ways:

- As a URL query parameter, called 'token':

  ```
  http://localhost:3010/pdf/BlaisePascalMagazine_61_UK.pdf?token=%7BF5A07A5B-5184-4256-8EE
  ```

- As a HTTP header, called 'X-Access-Token':

  ```
  X-Access-Token: {F5A07A5B-5184-4256-8EE6-20E2DE987AF5}
  ```

This means we must adopt the download module so it first checks the token, and then checks if the user that owns the token can download the file. The change is trivial:

```
  procedure TCorsFileModule.HandleRequest(ARequest: TRequest; AResponse: TResponse
  );
begin
  Cors.Enabled:=true;
  if Cors.HandleRequest(aRequest,aResponse) then exit;
  if not CheckToken(aRequest,aResponse) then exit;
  inherited HandleRequest(ARequest, AResponse);
end;
```

The `CheckToken` function does the actual work. It uses the `CheckToken` function from the `TSessionManager` class to verify the token. If the token is OK, the user ID is returned, if the token is not OK, -1 is returned.

The returned user ID is then used to check if the user is allowed to download the requested PDF (the `GetRequestFileName` function is a method of the file download datamodule that comes with FPC):

```
function TCorsFileModule.CheckToken(
```

```
     ARequest: TRequest;
     AResponse: TResponse): Boolean;

var
  aToken,aFileName : String;
  aID : int64;

begin
  // Check URL parameter and HTTP header for token.
  aToken:=aRequest.QueryFields.Values['token'];
  if aToken='' then
    aToken:=aRequest.CustomHeaders.Values['x-access-token'];
  Result:=(aToken<>'');
  if Result then
    begin
    // Check token in database
    aID:=aSession.CheckToken(aToken);
    Result:=aID<>-1;
    if Result then
       begin
       // Get requested filename.
       aFileName:=ExtractFileName(GetRequestFileName(aRequest));
       // Check if user is allowed to download this file.
       Result:=aSession.CheckFileAllowed(aID,aFileName)
       end;
    end;
  if not Result then
     begin
     aResponse.Code:=403;
     aResponse.CodeText:='FORBIDDEN';
     aResponse.SendContent;
     end;
end;
```

Note that if the token is invalid, or the user is not allowed to download the PDF, a `403 FORBIDDEN` HTTP return code is sent to the browser.

The `CheckToken` function of the session manager does the actual check of the token. It is again quite simple:

```
function TSessionManager.CheckToken(const aToken: string): int64;

const
  SQLSelect = 'select tk_user_fk,tk_expires_on from tokens where (tk_token=:token)';

var
  Tr : TSQLTransaction;
  Qry : TSQLQuery;
  OK : Boolean;

begin
  OK:=False;
  Qry:=nil;
  Result:=-1;
  TR:=CreateTransaction;
```

```
  try
    Qry:=CreateQuery(SQLSelect,['token',aToken],Tr);
    Qry.Open;
    if Not Qry.IsEmpty
       and (Qry.FieldByName('tk_expires_on').asDateTime>Now) then
      Result:=Qry.FieldByName('tk_user_fk').asLargeInt;
    if Result<>-1 then
      UpdateToken(aToken,Tr);
  finally
    if not OK then TR.Rollback;
    ReleaseQuery(Qry,Tr);
  end;
end;
```

If the token has not expired, it is extended with 30 minutes using the `UpdateToken`:

```
function TSessionManager.UpdateToken(
      const aToken: String;
      aTrans: TSQLTransaction): TDateTime;

Const
  SQLUpdate =
    'update tokens set ' +
    ' tk_expires_on = clock_timestamp() + interval ''30 minutes'' '+
    ' where (tk_token=:TOKEN) returning tk_vervalt_op;';

Var
  Qry : TSQLQuery;

begin
  Qry:=CreateQuery(SQLUpdate,['TOKEN',aToken],aTrans);
  try
    Qry.Open;
    if not Qry.IsEmpty then
      Result:=Qry.Fields[0].AsDateTime
    else
      Result:=0;
  finally
    ReleaseQuery(Qry);
  end;
end;
```

This is to avoid that the user needs to login every 30 minutes, but after more than 30 minutes of inactivity, the token does expire. The above mechanism is a simple one, in practice, more advanced strategies can be used.

Lastly, the `CheckFileAllowed` call is used to check whether the user is entitled to download the requested PDF file. This is done using the `userissues` table: if a record is present for the requested issue and user, the user is allowed to download the pdf. The check is then very simple:

```
function TSessionManager.CheckFileAllowed(aUserID: int64;
  const aFileName: string): Boolean;

Const
```

```
  SQLSelect =
    'select '+
    '  ui_id '+
    'from '+
    '  userissues '+
    '  inner join issues on (i_id=ui_issue_fk) '+
    'where '+
    '  (i_filename=:filename) '+
    '  and (ui_user_fk=:uid)';

var
  Tr : TSQLTransaction;
  Qry : TSQLQuery;

begin
  Tr:=CreateTransaction;
  try
    Qry:=CreateQuery(SQLSelect,[
        'uid',aUserID,
        'filename',lowercase(aFileName)],Tr);
    Qry.Open;
    Result:=Not Qry.IsEmpty;
  finally
    ReleaseQuery(Qry,Tr);
  end;
end;
```

4 Allowing to download an issue by number

One of the requirements was that the user can download and view an issue by entering the number of the issue. To map this to a PDF filename, a routine is needed that checks the `issues` table and returns the corresponding filename. To implement this, when the application needs to show an issue, we'll let it download a PDF with a special URL:

```
http://localhost:3010/issue/45
```

In the above URL, the number `45` must be replaced with the actual issue.

To code this on the server, we must implement a handler for the above URL. We register it with the HTTP router as follows:

```
HTTPRouter.RegisterRoute('/issue/:Issue',@IssueToPDF);
```

The `IssueToPDF` is a simple routine. The search mechanism has the list of articles in an issue in an array in memory. This list can be accessed to retrieve the filename of the issue.

When a filename is found, instead of sending the file, a redirect response is sent to the browser with the 'normal' pdf download location: The redirect response means a 307 HTTP return code is sent, and the location of the PDF is returned in the `Location` HTTP header.

The server response will be something like this:

```
HTTP/1.1  307 Temporary Redirect
Location: /pdf/BlaisePascalMagazine_45_46_UK.pdf
```

Upon receiving the 307 return code, the browser will immediatly do a second request to the new location. To the user, this is transparant.

To code this is quite simple:

```
Procedure IssueToPDF(ARequest: TRequest; AResponse: TResponse);

var
  Cors : TCORSSupport;
  PDF : String;

begin
  Cors:=TCORSSupport.Create;
  try
    Cors.Enabled:=true;
    if Cors.HandleRequest(aRequest,aResponse) then exit;
  finally
    Cors.Free;
  end;
  PDF:=aSearch.IssueToPDF(aRequest.RouteParams['Issue']);
  if (PDF<>'') then
    aResponse.SendRedirect('/pdf/'+PDF)
  else
    begin
    aResponse.Code:=404;
    aResponse.CodeText:='Not Found';
    end;
  aResponse.SendContent;
end;
```

With this, we have completed the extension of the server. We can now turn to the changes in the client (pas2js) application.

5 Searching in the client

In the previous articles where searching through a PDF in a browser application was handled, 3 mechanisms have been treated, where a search was performed in 3 different locations:

1. In the displayed PDF, using the mechanisms provided by the `pdf.js` package in the browser.
2. In a list of articles, using an in-memory copy of the list of articles.
3. In a database built with a PDF indexer.

If our application should be able to work online and offline, we must consider if each of the mechanisms is usable: Searching in the displayed PDF is of course always possible. When offline, the search in the database is unavailable, and the best we can do is replace it transparently by a search in the list of articles.

In order to do so, we need to adapt the search mechanism: in our last iteration of the PDF application, the search algorithms (PDF and database) were handled by the `TPDFSearchControl`. We now need to add the search in the list of articles (as demonstrated in the first article about showing a PDF) to this class.

To keep the code simple, we'll split out the search mechanisms in separate classes. The `TPDFSearchControl` will then, depending on the user setting and the online/offline status of the browser, select a search mechanism.

The 3 search classes are responsible for searching and displaying the results below a given HTML tag. When the user selects a result, a special event is triggered with enough information to show the selected result. The `TPDFSearchControl` will then do what is necessary to display the PDF and jump to the page containing the result.

The search class is also responsible for returning a list of words for auto-completion in the search box: the mechanism that was built to search the indexed database has an implementation. For the list of articles, a list of words can be built on the fly, and a list of words in the current PDF can also be constructed.

Since the three mechanisms need to perform the same functions, we define the following interface to encapsulate the requirements:

```
TPageInfo = record
  Issue, Title, FileName : String;
  Page: Integer;
  useIssue : Boolean;
end;

TShowPDFPageEvent = procedure(aPage : TPageInfo) of object;
TWordListCallBack = reference to procedure(List : TStrings);

{ ISearchEngine }

ISearchEngine = Interface
  // Property getters & setters
  function GetOnShowResultPanel: TNotifyEvent;
  procedure SetOnShowResultPanel(AValue: TNotifyEvent);
  Procedure SetResultsElement(aValue : TJSHTMLElement);
  Function GetResultsElement : TJSHTMLElement;
  Procedure SetShowPageEvent(aValue : TShowPDFPageEvent);
  function GetShowPageEvent : TShowPDFPageEvent;
  // Actual interface
  Procedure Search(const aTerm : string; const aIssue : String);
  procedure GetWordList(aTerm : string; aOnResults : TWordListCallBack);
  // Easy access using properties
  Property ResultsElement : TJSHTMLElement Read GetResultsElement Write SetResultsElement;
  Property ShowPDFPageEvent : TShowPDFPageEvent Read GetShowPageEvent Write SetShowPageEvent
  Property OnShowResultPanel : TNotifyEvent Read GetOnShowResultPanel Write SetOnShowResultPa
end;
```

The `Search` call will display the list of found occurrences of the search term below `ResultsElement`. The `OnShowResultPanel` event can be used to notify the caller that there were results, and that the result element needs to be shown (the result panel is by default closed, it needs to be opened when results are available).

When the user clicks a result, the `ShowPDFPageEvent` event is triggered with a `TPageInfo` record: this record contains enough information to download a PDF if

needed, and jump to the correct page.

The `GetWordList` method is also clear: it needs to show a list of words. When a word list is available the `aOnResults` callback is called, passing it the list of words. A callback must be used, since the search can be asynchronous: consulting the database on the server is an asynchonous call.

In the previous iteration of the PDF viewing and indexing application, the `TPDFSearchControl` contained 2 search mechanisms. We'll factor these out into their own classes, so we'll have 4 classes that work together to implement the search functionality. The first 3 classes are just a refactoring of the existing classes.

TPDFSearchControl This will just handle the search mechanism's UI: it managed the edit and search buttons, shows the word list for completion and shows or hides the results panel. The actual search is handled by the other 3 components. When a PDF must be shown, an event handler is called.

TServerSearch Implements the above interface using the server search mechanism discussed in the previous article.

TPDFSearch Implements the above interface for searching in the displayed PDF. It uses the PDF search mechanism discussed in the first article in this series.

TArticleSearcher Implements the above interface using a search mechanism in a list of articles which is included in the application when it is loaded from disk.

When the `TPDFSearchControl` class is created, it creates instances of the 3 search mechanisms:

```
constructor TPDFSearchControl.Create(aOwner: TComponent);
begin
  Inherited;
  FLocalsearch:=TArticleSearcher.Create(Self);
  FServerSearch:=TServerSearch.Create(Self);
  FSearch:=TPDFSearch.Create(Self);
end;
```

And it initializes them in its `BindElements` method:

```
procedure TPDFSearchControl.BindElements;

begin
  // ... other code...
  PrepareEngines(True);
end;

procedure TPDFSearchControl.PrepareEngines(Full : boolean);

begin
  PrepareEngine(FLocalsearch as ISearchEngine,Full);
  PrepareEngine(FServerSearch as ISearchEngine,Full);
  PrepareEngine(FSearch as ISearchEngine,Full);
end;

procedure TPDFSearchControl.PrepareEngine(aEngine : ISearchEngine; Full : Boolean);
```

```
begin
  aEngine.ShowPDFPageEvent:=FOnShowPDFPage;
  if Full then
    begin
    aEngine.OnShowResultPanel:=@HandleShowResultPanel;
    aEngine.ResultsElement:=pnlResults;
    end;
end;
```

The `PrepareEngine` is called for all 3 searchengines: it will initialize the relevant properties so the classes can do their work. the `FOnShowPDFPage` is an event handler that is set by the main application class: the main application class is responsible for loading a PDF file.

The 'click' event handler of the search button in `TPDFSearchControl` now becomes quite simple:

```
procedure TPDFSearchControl.onSearch(aEvent: TJSEvent);
var
  aterm : string;

begin
  aterm:=SearchTerm;
  if Length(aTerm)<=1 then
    exit;
  CurrentSearchEngine.Search(aTerm,FIssueFilter);
end;
```

The `CurrentSearchEngine` property returns an `ISearchEngine` interface. The getter of this property decides which search engine to return based on the `PDFSearch` property (basically the value of the 'Search PDF' checkbox) and a property `OffLine` :

```
function TPDFSearchControl.GetSearchEngine: ISearchEngine;

begin
  if PDFSearch then
    Result:=FSearch as ISearchEngine
  else if OffLine then
    Result:=FLocalsearch as ISearchEngine
  else
    Result:=FServerSearch as ISearchEngine;
end;
```

The `OffLine` property is determined by the online or offline status of the navigator. It is determined during startup of the application, and is maintained during the lifetime of the application. We'll show how to do this later on.

The application has a feature where the edit box shows a list of words for completion, based on the returns from the server. This mechanism needs to be reworked so the list of words is fetched from the current search engine. This mechanism was implemented in a timer event, which now becomes quite short:

```
function TPDFSearchControl.DoCompleteWord(Event: TEventListenerEvent): boolean;
```

```
  procedure DoServerSearchWord;

  begin
    if Length(edtSearch.Value)>1 then
      CurrentSearchEngine.GetWordList(edtSearch.Value,@DoShowWordList);
  end;

begin
  Result:=False;
  if FSearchTimerID<>0 then
    window.clearTimeout(FSearchTimerID);
  FSearchTimerID:=window.SetTimeout(@DoServerSearchWord,200);
end;
```

The `GetWordList` will call `DoShowWordList` as soon as the list of words has been retrieved. The `DoShowWordList` routine which will actually show the list of words now simply needs to iterate over all words in the list:

```
procedure TPDFSearchControl.DoShowWordList(List : TStrings);

Var
  S : String;
  P : TJSHTMLELement;
  A : TJSHTMLAnchorElement;

begin
  mnuAutoComplete.style.setProperty('display','none');
  mnuAutoComplete.InnerHTML:='<div class="dropdown-content"></div>';
  P:=TJSHTMLELement(mnuAutoComplete.firstElementChild);
  For S in List do
    begin
    a:=TJSHTMLAnchorElement(Document.createElement('a'));
    a.href:='#';
    a.classlist.Add('dropdown-item');
    a.innerText:=s;
    a.dataset['value']:=s;
    a.addEventListener('click',@DoWordSelected);
    P.appendChild(a);
    end;
  mnuAutoComplete.style.setProperty('display','block');
end;
```

The implementation of the algorithm to retrieve a word list was moved to the `TServerSearch` class. It remains almost the same as it was, with exception that it fills a `TStringlist`.

The `ISearchEngine` interface contains some boilerplate code to define 3 properties (they must be defined through getters and setters).

In order to reduce the code needed in the 3 search classes to implement this interface, we'll descende all 3 classes from a common ancestor: `TSearchBase`. Here is the definition:

```
TSearchBase = class(TComponent)
Private
```

```
  FShowPDFPageEvent : TShowPDFPageEvent;
  FOnShowResultPanel : TNotifyEvent;
  FResultsElement: TJSHTMLElement;
Protected
  // Property getters & setters
  function GetOnShowResultPanel: TNotifyEvent;
  procedure SetOnShowResultPanel(AValue: TNotifyEvent);
  Procedure SetResultsElement(aValue : TJSHTMLElement);
  Function GetResultsElement : TJSHTMLElement;
  Procedure SetShowPageEvent(aValue : TShowPDFPageEvent);
  function GetShowPageEvent : TShowPDFPageEvent;
  // Easy access for descendents
  procedure ShowPDFPage(aInfo : TPageInfo);
  // Show results panel.
  procedure ShowResultsPanel;
  // Clear results panel.
  procedure ClearResultPanel;
  // Append a HTML node to the results panel.
  procedure AppendToResults(aElement : TJSHTMLElement);
Public
  // Easy access using properties
  Property ResultsElement : TJSHTMLElement Read GetResultsElement Write SetResultsElement;
  Property ShowPDFPageEvent : TShowPDFPageEvent Read GetShowPageEvent Write SetShowPageEvent;
  Property OnShowResultPanel : TNotifyEvent Read GetOnShowResultPanel Write SetOnShowResultPa
end;
```

The various `Get*` and `Set*` methods do nothing but setting and setting the values of the private fields for the properties. The easy access methods are there to call the event handlers, if they have been set. This avoids that descendents must all implement a check.

The `TServerSearch` class is reworked as a descendent of this class, but contains no new code compared to the previous iteration of our application, so we won't repeat it here. The same is true for the `TPDFSearch` class: nothing changes for this class, except that the signature of the method changes somewhat.

6 Working offline

When working offline, we cannot contact the server and perform a search on a database. We also cannot distribute and access the database from within the browser.

What we can do is offer limited search: the list of articles and issues is distributed with the offline version of the application. Basically, we create a javascript file in containing a 'database' of articles. The database is then simply a Javascript array of records, which looks like this (formatting added for display purposes):

```
var BPMArticles = [
    {
      "i" : "1",
      "p" : 6,
      "a" : "Representing graphics for math functions",
      "u" : "Peter Bijlsma",
      "c" : ""
```

```
},
{
  "i" : "1",
  "p" : 8,
  "a" : "Client Dataset Toolkit",
  "u" : "Detlef Overbeek",
  "c" : ""
},
// ....
]
```

This javascript file is included in the html page using a script element:

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

Accessing this array from a Pascal program is easy: One record in this array can be declared in Pascal as an external class:

```
Type
  TArticle = Class external name 'Object' (TJSObject)
    Issue : String; external name 'i';
    Page : Integer; external name 'p';
    Title : String; external name 'a';
    Author : String; external name 'u';
    Code : string; external name 'c';
  end;
  TArticleArray = Array of TArticle;
```

Note the use of 'external name' to map the human-readable fields (`Issue`, `Page` etc.) to the actual member names used in Javascript.

The array itself is then defined as follows:

```
var
  BPMArticles : TArticleArray; external name 'BPMArticles';
```

As you can see, the variable is declared external: this means it is actually defined outside the pascal code.

Armed with his, we can now set about creating a class that implements a local search mechanism and a mechanism to get a word list: the `TArticleSearcher` class in the `articlesearch` unit.

This class is defined as a descendent of `TSearchBase`, and has 2 main methods. The first is to get a list of words:

```
procedure TArticleSearcher.GetWordList(aTerm: string;
      aOnResults: TWordListCallBack);

var
  L : TStringList;
  aArticle : TArticle;
  S : String;
  R : TJSRegexp;
begin
  if not assigned(aOnResults) then
```

```
    exit;
  aTerm:=UpperCase(aTerm);
  L:=TStringList.Create;
  try
    L.Sorted:=True;
    L.Duplicates:=dupIgnore;
    R:=TJSRegexp.New('\b(?:\w|-)+\b','g');
    For aArticle in BPMArticles do
      for S in TJSString(aArticle.Title).match(R) do
        if pos(aTerm,Uppercase(S))>0 then
          L.Add(s);
     aOnResults(L);
  finally
    L.Free;
  end;
end;
```

This is a very simple loop over the array of article records: For every article, the `Title` field is split into words using the Javascript 'match' method of the String type: the compli If the word contains the search term (we check this case-insentitively), we add it to the list: the list ignores duplicates, so we get each word only once. At the end we call the callback.

The search mechanism works in a completely similar way. It clears any previous results, loops over the article list, and if an article matches the search term, it is included in the result.

```
procedure TArticleSearcher.Search(const aTerm: string; const aIssue: String);

Var
  aIdx : integer;
  aArticle : TArticle;
  V,IssueFilter : string;
  I : integer;

begin
  if Not Assigned(ResultsElement) then exit;
  IssueFilter:='';
  For I:=1 to Length(aIssue) do
    if Pos(aIssue[I],'0123456789_')>0 then
      IssueFilter:=IssueFilter+V[i];
  ClearResultPanel;
  For aIdx:=0 to Length(BPMArticles)-1 do
    begin
    aArticle:=BPMArticles[aIdx];
    if aArticle.IsMatch(aTerm,IssueFilter) then
      ResultsElement.AppendChild(CreateArticleRow(aIdx,aArticle));
    end;
  ShowResultsPanel;
end;
```

At the end, the results panel is shown.

The `IsMatch` procedure which is used to determine if an article is matched, is a helper method for `TArticle`:

```
TArticleHelper = class helper for TArticle
  Function IsMatch (aTerm : String; aIssue : string) : Boolean;
end;

function TArticleHelper.IsMatch(aTerm: String; aIssue: string): Boolean;
begin
  aTerm:=UpperCase(aTerm);
  Result:=(aTerm='') or ((Pos(aTerm,UpperCase(Author))>0)
                     or (Pos(aTerm,UpperCase(Title))>0));
  if Result and (aIssue<>'') then
    Result:=(aIssue=Issue);
end;
```

This must be implemented as a helper method, since the `TArticle` class is defined as an external class, and therefore its definition cannot contain pascal methods.

The `CreateArticleRow` method uses a string constant `DefaultPanel` with a HTML template to construct the actual HTML using a simple search and replace mechanism:

```
  function TArticleSearcher.CreateArticleRow(aIdx: Integer; aArticle: TArticle
  ): TJSHTMLElement;


Var
  Panel : String;

begin
  Result:=TJSHTMLElement(Document.createElement('div'));

  Panel:=StringReplace(DeeaultPanel,'{{issue}}',aArticle.Issue,[rfReplaceAll]);
  Panel:=StringReplace(Panel,'{{page}}',IntToStr(aArticle.Page),[rfReplaceAll]);
  Panel:=StringReplace(Panel,'{{title}}',aArticle.Title,[rfReplaceAll]);
  Panel:=StringReplace(Panel,'{{author}}',aArticle.Author,[rfReplaceAll]);

  Result.dataset['issue']:=aArticle.Issue;
  Result.dataset['page']:=IntToStr(aArticle.Page);
  Result.dataset['articleid']:=intToStr(aIdx);
  Result.dataset['title']:=aArticle.Title;
  Result.AddEventListener('click',@OnArticleClick);
  Result.innerHTML:=Panel;
end;
```

Finally, the `OnClick` handler for the result element collects some data to set up a `TPageInfo` record, which is then used to display the correct PDF page:

```
procedure TArticleSearcher.OnArticleClick(aEvent : TJSEvent);

var
  aPage : TPageInfo;

begin
  aPage:=Default(TPageInfo);
  With TJSHTMLElement(aEvent.currentTargetElement) do
    begin
```

```
    aPage.Issue:=dataset['issue'];
    aPage.Page:=StrToIntDef(dataset['page'],-1);
    aPage.Title:=dataset['title'];
    aPage.useIssue:=True;
    end;
  ShowPDFPage(aPage);
end;
```

The `ShowPDFPage` uses the `ShowPDFPageEvent` event handler to actually show the PDF on the correct page. And with this, our offline search mechanism is ready.

7 Detecting offline status and showing a PDF

The offline search mechanism has to be activated when the navigator has no access to internet: We implemented the `OffLine` property for this in the `TPDFSearchControl`. But this property has not yet been set to a correct value.

Luckily, the browser has a property that indicates whether it is currently online or offline: The `window.Navigator.onLine` property indicates whether the browser is currently online or offline. What is more, the `Window` class implements 2 events 'online' and 'offline' that are triggered when the browser goes online or offline, respectively. So we can use `AddEventListener` to install an event handler and react to changes in online or offline status.

This is done in the `DetectOffLine` routine in the application class. It does 2 things: it detects whether the application was started by double clicking the index.html file in the file explorer or whether it was started from a website. The result is stored in the `IsLocal` property, and the online status is stored in the `IsOffLine` property:

```
procedure TBPMLibraryApplication.DetectOffline;

  Procedure updateOnlineStatus(event : TJSEvent);

  begin
    IsOffLine:=not window.Navigator.onLine;
  end;

begin
  IsLocal:=Copy(window.location.protocol,1,4)='file';
  IsOffLine:=not window.Navigator.onLine;
  window.addEventListener('online',@updateOnlineStatus);
  window.addEventListener('offline',@updateOnlineStatus);
end;
```

Note that the `online` and `offline` eventhandler is used to update the `IsOffLine` property.

The `IsOffLine` property of the application object has a setter, and is used to propagate the value to the searchcontrol:

```
procedure TBPMLibraryApplication.SetIsOffLine(AValue: Boolean);
begin
  if FIsOffLine=AValue then Exit;
  FIsOffLine:=AValue;
  FSearchPane.OffLine:=FisOffline;
```

```
end;
```

Thus, the search mechanism will know whether to search locally or remote.

As we've seen, the search mechanism only has an event which it must use to open a PDF and display a certain page. The event is set in the main application to the following event handler:

```
procedure TBPMLibraryApplication.HandleShowPDFPage(aPage: TPageInfo);

  Function IsSameAsLastPDF : Boolean;
  begin
    if aPage.useIssue then
      Result:=(FLastIssue<>'') and (FLastIssue=aPage.Issue)
    else
      Result:=(FLastPDF<>'') and (FLastPDF=aPage.FileName)
  end;

begin
  // local search or PDF already loaded
  if IsSameAsLastPDF then
    FViewer.ShowPage(aPage.Page)
  else if Not HaveLocalFile(aPage) then
    LoadRemotePDF(aPage)
  else
    LoadLocalPDF(aPage);
end;
```

If the PDF is already loaded (checked in `IsSameAsLastPDF`), then the viewer panel is simply instructed to show the new page. If the PDF is not yet loaded, then it must be loaded before the correct page can be shown. If the page was loaded from local disk (as is the case on the USB-stick version of the Blaise Pascal magazine), then it is loaded from disk using `LoadLocalPDF`, else it is loaded using `LoadRemotePDF`.

The `HaveLocalFile` function uses the `IsLocal` property that was initialized by the application to decide whether a file can be loaded from disk or not:

If `IsLocal` is false, we know the page is loaded from a website, and the PDF files will not be available locally. But if `IsLocal` is true, it still can be that the PDF is not available locally: When an online search was performed, the search result could have returned a PDF that is not available locally. To cater for that case we must check the list of available articles to see if the requested issue is present, and this is done by checking the issue number in the list of articles:

```
function TBPMLibraryApplication.HaveLocalFile(aPage : TPageInfo) : Boolean;

Var
  aArticle : TArticle;

begin
  Result:=IsLocal;
  if Result then
    begin
    // Determine if we have the PDF locally
    Result:=False;
    for aArticle in BPMArticles do
```

```
      if (aPage.Issue=aArticle.Issue) then
        Exit(True);
    end
end;
```

When the PDF is available locally, we load it using the trick shown in the first article in this series: a script tag is inserted which defines contents of the PDF as a Javascript variable (pdfData):

```
var
  pdfData : String; external name 'pdfData';

procedure TBPMLibraryApplication.LoadLocalPDF(aPage: TPageInfo);

  Procedure DoShowPage;
  begin
    FViewer.ShowPage(aPage.Page);
  end;

  function DoLoaded(Event : TEventListenerEvent) : Boolean;

  var
    Src : TPDFSource;

  begin
    Src:=TPDFSource.new;
    Src.Data:=pdfData;
    Fviewer.StartPDFRender(Src,@DoShowPage);
  end;

Var
  Script : TJSHTMLScriptElement;
  FN : String;

begin
  Script:=TJSHTMLScriptElement(document.CreateElement('script'));
  FN:=aPage.FileName;
  if Pos('file://',FN)=1 then
    Delete(FN,1,7);
  else if FN='' then
    FN:='issues/issue'+aPage.Issue+'.js';
  Script.Src:=FN;
  Script.Onload:=@DoLoaded;

end;
```

Finally, the LoadRemotePDF routine loads a PDF from the server. It needs to distinguish between a call where only the issue number is given (the user requested simply to see an issue) or when the PDF filename is known. In the former case it uses the /issue/ URL which we showed in the beginning of the article, in the latter case the /pdf/ URL is used:

```
procedure TBPMLibraryApplication.LoadRemotePDF(aPage: TPageInfo);
```

Figure 1: Showing issue 1 of the Blaise Pascal magazine library

```
  Procedure DoShowPage;
  begin
    FViewer.ShowPage(aPage.Page);
  end;

var
  Src : TPDFSource;

begin
  Src:=TPDFSource.new;
  if aPage.useIssue then
    Src.url:=ServerURL+'issue/'+aPage.Issue
  else
    Src.url:=ServerURL+'pdf/'+aPage.FileName+'?token='+ encodeURIComponent(FLogin.Token);
  FLastPDF:=aPage.FileName;
  FViewer.StartPDFRender(Src,@DoShowPage);
end;
```

And with this routine, the application is ready to go. The application with an issue loaded locally is shown in figure 1 on page 24.

8 Conclusion

In this article, we've shown how to make a real-world application using Pas2JS that can work both online and offline and adapts itself: we refactored techniques introduced in the previous articles in this series. The application can still be improved: for instance the online/offline status can be made visual - e.g. changing a background color. The application times out after 30 minutes, but no warning is given: this can also be improved. As with all software, the work is never finished...