Indexing PDFs - searching the index

Michaël Van Canneyt

March 28, 2023

Abstract

In a previous contribution, we've shown how to create an index of words in a PDF file. In this article, we'll show how to use this index in a search prgram and use that to implement a search in a website.

1 Introduction

In the first article on searching PDF files, we showed how to create a PDF indexer: a program to analyse a bunch of PDF files, and store occurrences of words in the PDF files in a database. All this was accomplished with classes that are part of Free Pascal: the end result was that we have a database with all the words in the PDF files and the locations on which these words occur.

In this article, we'll show how to query this database: we'll make a small webpage in which the user can enter a search term. The search term is sent to the server, which will reply with a list of pages and the title of the article of which the page is a part. The user can then choose which article he wants to read. The correct PDF will be downloaded and opened on the correct page. Once the PDF is downloaded, the user can search locally in the PDF. For simplicity, the indexed PDF files are assumed to be in a directory on the server, but they can of course be located in a database or distributed over many directories.

To help the user, we'll also create an auto-complete mechanism for the search box: as soon as the user typed 2 letters, the program will present a list of words that contain the letters he or she typed. The user can then select the complete words and let the backend search for it. This will not only save typing, but will also increase the accuracy of the search.

2 The server search part: overview

The server is a simple HTTP application which has 3 functions:

1. Return a list of words from the database, based on some letters in the word.
2. Return a list of matches for a word, returning the pdf name, the pages with a match and the article name to which the page belongs.
3. Return a PDF file.

To make this HTTP application, we start a new HTTP application as usual in the Lazarus IDE.

The main program source code is changed somewhat, so it reads as follows:

```
begin
  ConfigureApp;
  {$ifndef usecgi}
  Application.Port:=3010;
  {$endif}
  Application.Initialize;
  Application.Run;
  ASearch.Free;
end.
```

The configureApp is a call to a routine that creates the search mechanism (the 2 first statements in the routine) and configures the HTTP router, which is responsible for routing the 3 HTTP requests that the application supports:

```
Var
  aSearch : TSearcher;

Procedure ConfigureApp;

begin
  // Create search mechanism
  aSearch:=TSearcher.Create(Application);
  aSearch.Init;
  // Register routes for the search mechanism
  HTTPRouter.RegisterRoute('/search',@aSearch.DocSearch,true);
  HTTPRouter.RegisterRoute('/list',@aSearch.WordList,False);
  // Set default file downloader
  DefaultFileModuleClass:=TCorsFileModule;
  // Where are the PDF files located.
  RegisterFileLocation('pdf',aSearch.PDFLocation);
end;
```

The last 2 lines use a descendent of a standard Free Pascal file downloader module (`TFPCustomFileModule`) named `TCorsFileModule` to set up a download functionality for the PDF files. Note that the path must of course be matched.

The `TCorsFileModule` is there to enable CORS:

```
TCorsFileModule = Class(TFPCustomFileModule)
  Procedure HandleRequest(ARequest: TRequest;
                          AResponse: TResponse); override;
end;

procedure TCorsFileModule.HandleRequest(ARequest: TRequest;
                                        AResponse: TResponse);
begin
  Cors.Enabled:=true;
  if Cors.HandleRequest(aRequest,aResponse) then exit;
  inherited HandleRequest(ARequest, AResponse);
end;
```

The standard `TFPCustomFileModule` does not allow CORS requests, so a descendent is made that allows this. It enables the CORS support (standard part of every FCL-Web module) and if a CORS preflight request is detected, it is handled. If it is a regular request, the PDF download request is handled: We need not do more than this to serve the PDF files from an url formed as:

```
http://localhost:3010/pdf/somepdf.pdf
```

(the host & port may differ depending on your setup) all the rest will be handled by the file module itself.

The actual searching happens in the `TSearcher` class, which is defined as follows:

```
TSearcher = Class(TComponent)
private
  procedure ConfigSearch(aRequest: TRequest;
                         aResponse: TResponse);
  procedure ConfigWordList(aRequest: TRequest;
                           out aContaining : UTF8string;
                           Out Partial : TAvailableMatch;
                           Out aSimple : Boolean);
  procedure ConnectToDatabase;
  procedure DisConnectFromDatabase;
  function FindArticle(Position: Integer;
                       const aFile: String): TArticleData;
  function SearchDataToJSON(aID: Integer;
                            const aRes: TSearchWordData): TJSONObject;
  procedure SendJSON(J: TJSONObject;
                     aResponse: TResponse);
  procedure SetupMetadata;
  procedure LoadArticles;
Protected
  function InitSearch(aResponse: TResponse): Boolean;
  function SetupDB(aIni: TCustomIniFile): TCustomIndexDB;
  Property DB : TCustomIndexDB;
  Property Search : TFPSearch;
  Property MinRank : Integer;
  Property FormattedJSON : Boolean;
  Property Articles : TArticleDataArray;
Public
  Procedure Init;
  procedure DoDBLog(Sender: TSQLConnection;
                    EventType: TDBEventType;
                    const Msg: String);
  Function CheckParams(aRequest : TRequest;
                       aResponse : TResponse) : Boolean;
  Function CheckSearchParams(aRequest : TRequest;
                             aResponse : TResponse) : Boolean;
  Procedure DocSearch(aRequest : TRequest;
                      aResponse : TResponse);
  Procedure WordList(aRequest : TRequest;
                     aResponse : TResponse);
  Property AllowCors : Boolean;
end;
```

The `DocSearch` and `WordList` routines are the entry points for handling the browser requests, we'll treat them in a moment.

The `Init` which is called during program procedure initializes the search engine. It reads a configuration file with the settings for the database connection and some further settings that configure the search mechanism. The configuration file is

searched in several directories, in the `GetConfigFileName` routine, which we will not present here.

```
procedure TSearcher.Init;

Const
  // Adapt this default to your setup....
  DefaultLocation = '/home/michael/Documents/pdf/blaise/';

Var
  CFN : String;
  aIni: TMemIniFile;
begin
  CFN:=GetConfigFileName;
  aIni:=TMemIniFile.Create(CFN);
  try
    FFormattedJSON:=aIni.ReadBool('search','formatjson',False);
    FDefaultMinRank:=aIni.ReadInteger('search','minrank',1);
    FDefaultMetadata:=aIni.ReadBool('search','metadata',true);
    FAllowCors:=aIni.ReadBool('search','allowcors',true);
    FPDFLocation:=aIni.ReadString('Files','PDF',DefaultLocation);
    FDB:=SetupDB(aIni);
    FSearch:=TFPSearch.Create(Self);
    FSearch.Database:=FDB;
  finally
    aIni.Free;
  end;
  SetupMetadata;
end;
```

4 variables are initialized from the configuration file, they influence the returned results to the client requests:

FFormattedJSON Should the JSON returned be formatted or not.

FDefaultMinRank minimum rank for a word occurrence to be included in the result.

FDefaultMetadata Should metadata be returned in absence of a parameter that specified it.

FAllowCors Should CORS HTTP headers be set on the result ?

FPDFLocation The location of the PDF files on disk.

The `FDB` variable is an instance of `TCustomIndexDB` as presented in the previous article: it handles the database connection for the `TFPSearch` instance which does the actual search in the database.

The `TFPSearch` class is part of the fpIndexer unit presented in the previous article. It handles searching in the index database. The class is declared as follows:

```
TFPSearch = class (TComponent)
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
```

```
  function Execute: int64;
  procedure AddResult(index: integer; AValue: TSearchWordData);
  procedure SetSearchWord(AValue: UTF8String);
  Function GetAvailableWords(out aList : TUTF8StringArray;
                             aContaining : UTF8String;
                             Partial : TAvailableMatch) : Integer;
  property Count: integer;
  property RankedCount: integer;
  property Results[index: integer]: TSearchWordData ;
  property RankedResults[index: integer]: TSearchWordData;
published
  property Database: TCustomIndexDB;
  property Options: TSearchOptions;
  property SearchWord: TWordParser;
  Property UsePositionInRank : Boolean;
end;
```

The published properties can be used to configure the search mechanism: The `Database` property represents the connection to the database. Options can be set to

$$soContains$$

in which case the search will be for words containing the search term. `UsePositionInRank` will determine whether the position should be taken into account when ranking words (normally, only the file/name is taken into account)

The methods of the class have the following purpose:

SetSearchWord Set the word to search for.

Execute Start the search. Returns the number of found results.

AddResult add a match to the results index.

GetAvailableWords Get the list of words that contain the word `aContaining`. The matching mechanism is determined by the value of the `Partial` parameter: one of `amAll` (all words),`amExact` (exact match on the search term),`amContains` (words that contain the search term),`amStartsWith` (words that start with the search term)). The results are returned in the `aList` array, and the function returns the number of results.

The public properties of the class allow to examine the results of the search. The `Results` and `Count` properties describe the raw results of the search operation. The `RankedResults` and `RankedCount` properties describe the ranked results of the search operation.

The last line of the `Init` routine is a call to `SetupMetadata`. This routine sets up metadata definitions for the 2 REST result sets that can be handled by the server program: the program is set up in such a way that the result it returns is a JSON structure that can be consumed by the `TJSONDataset` implementation in pas2js.

The JSON structure that is consumed by a `TJSONDataset` dataset contains a "metadata" element, that describes the data. Rather than creating this structure every time we create a result set, we create it once and reuse it when a result is being returned.

```
procedure TSearcher.SetupMetadata;

begin
  FMetadata:=TJSONObject.Create([
    'root', 'data',
    'idField','id',
    'fields',TJSONArray.Create([
      TJSONObject.Create(['name','id','type','int']),
      TJSONObject.Create(['name','rank','type','int']),
      TJSONObject.Create(['name','articlePage','type','int']),
      TJSONObject.Create(['name','articleIssue',
                          'type','string','maxlen',10]),
      TJSONObject.Create(['name','articleAuthor','
                          type','string','maxlen',127]),
      TJSONObject.Create(['name','articleTitle',
                          'type','string','maxlen',255]),
      TJSONObject.Create(['name','url',
                          'type','string','maxlen',100]),
      TJSONObject.Create(['name','context',
                          'type','string',
                          'maxlen',MaxContextLen]),
      TJSONObject.Create(['name','date','type','date'])
     ])
  ]);
  FWordsMetadata:=TJSONObject.Create([
    'root', 'data',
    'idField','id',
    'fields',TJSONArray.Create([
      TJSONObject.Create(['name','id','type','int']),
      TJSONObject.Create(['name','word',
                          'type','string','maxlen',100])
     ])
  ]);
end;
```

3 The server search part: The word list for auto-completion

The `WordList` routine is called whenever the user types a search term, and the browser wants to show a completion list.

The `WordList` resource accepts several query parameters:

q (for query), this mandatory parameter contains the letters typed by the user.

t (for type) determines which words will be returned: The value is one of

all will return all words, the value of `q` must be empty.

contains will return all words that contain the value of `q`.

exact will return only an exact match: this allows to determine whether the word is present in the database.

startswith will return all words that start with the value of `q`.

s a boolean (0 or 1), if True, then the return only contains an array of words. If false, then an `id` field is returned as well.

m a boolean (0 or 1): include metadata in the response or not. If omitted, the default as in the config file is taken.

When invoked, it starts by initializing the response that will be sent to the browser, including the CORS headers for optional CORS support, and then proceeds by checking the query parameters: The `CheckSearchParams` routine checks whether the supplied query parameters are valid. If not, it puts an error code in the HTTP request and returns `False`. It will not be presented here. The `InitSearch` routine simply checks whether the database is available.

```
procedure TSearcher.WordList(aRequest: TRequest; aResponse: TResponse);

Var
  I : Integer;
  J : TJSONObject;
  A : TJSONArray;
  w,aContaining : UTF8String;
  aPartial : TAvailableMatch;
  aSimple : Boolean;
  aList : TUTF8StringArray;

begin
  aResponse.ContentType:='application/json';
  if AllowCORS then
    AResponse.SetCustomHeader('Access-Control-Allow-Origin','*');
  if not CheckSearchParams(aRequest,aResponse) then
    exit;
  if not InitSearch(aResponse) then
    exit;
  ConfigWordList(aRequest,aContaining,aPartial,aSimple);
  FSearch.GetAvailableWords(aList,aContaining,aPartial);
  J:=TJSONObject.Create;
  try
    if FIncludeMetadata then
      J.Add('metaData',FWordsMetadata.Clone);
    A:=TJSONArray.Create;
    if aSimple then
      For W in aList do
        A.Add(W)
      else
        begin
        For I:=0 to Length(aList)-1 do
          A.Add(TJSONObject.Create(['id',I+1,'word',aList[i]]));
        end;
    J.Add('data',A);
    SendJSON(J,aResponse);
  finally
    J.Free;
  end;
end;
```

The `ConfigWordList` routine extracts the query parameters from the request, and

makes sure the returned values (`aContaining`,`aPartial`,`aSimple`) are consistent. One could integrate this functionality with the `CheckSearchParams` routine. The `FSearch.GetAvailableWords` function is used to get the actual results.

Once the results are in, the rest of the routine is just constructing the JSON to be returned to the browser: here the 'simple' parameter is used to determine the structure of the elements in the result array.

The `SendJSON` simply returns the constructed JSON in the HTTP response:

```
procedure TSearcher.SendJSON(J : TJSONObject; aResponse: TResponse);

begin
  if FormattedJSON then
    aResponse.Content:=J.FormatJSON()
  else
    aResponse.Content:=J.AsJSON;
  aResponse.ContentLength:=Length(aResponse.Content);
  aResponse.SendContent;
end;
```

With this, the functionality so get a word list is complete. Being a simple HTTP request, it can be tested simply in the browser using for example the following URL:

```
http://localhost:3010/list?q=class&t=contains&s=1
```

A sample result for the word `class` can be seen in figure 1 on page 9. Note that the browser recognizes the result is JSON based on the content-type, and formats the result accordingly.

4 The server search part: The list of pages with word matches

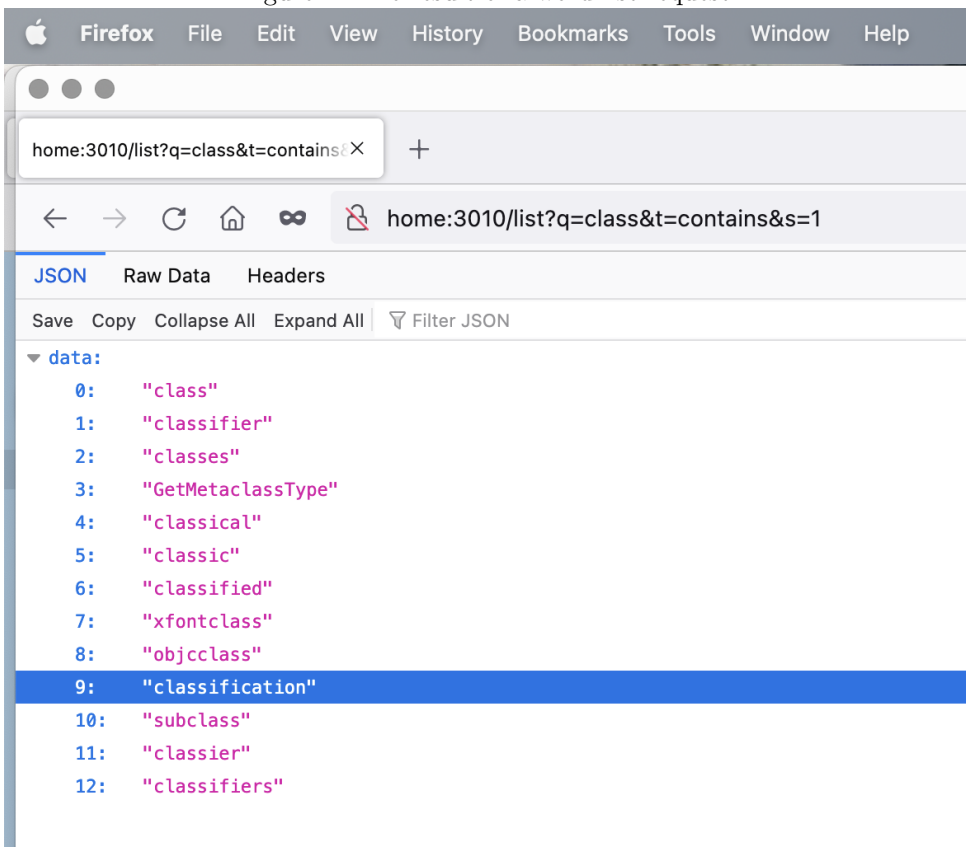
The second call needed in the server will find all occurrences for a word in the indexed PDFs. The PDF indexer stores for each word the name of the file in which the word occurs, and the page on which the word occurs.

For the application that we wish to make, the page number needs to be matched with an article: To match a page with an article, we have a table with the PDF name (and issue), starting and ending pages, author and title.

Since there is a limited amount of articles and the list changes not so often, it makes sense to cache the list in the server. We'll do so in the following array of records:

```
TArticleData = record
    StartPage,
    EndPage : Integer;
    Author,
    Title,
    Issue,
    PDF : String;
    function Match(aPDF : String; aPage : Integer) : boolean;
  end;
  TArticleDataArray = Array of TArticleData;
```

Figure 1: The result of a word list request



This information will matched with the result of the PDF index data. The HTTP entry point for the search mechanism is the 'search' path, linked to the `PDFSearch` method.

Similar to the `WordList` entry point, the `PDFSearch` method accepts some options through the query variables:

q (for query), this mandatory parameter contains the word to search for.

r (for rank) determines the minimal rank a match must have for the match to be included in the result. The default is zero.

c a boolean (0 or 1), if True, words containing the query will be returned. If False (the default) then only an exact word match is returned.

m a boolean (0 or 1): include metadata in the response or not. If omitted, the default as in the config file is taken.

The `PDFSearch` method is surprisingly short, and has the same overall structure as the `WordList` method. It starts by preparing the response, setting CORS headers and checking the parameters. The next step is to initialize and configuring the search. If the initializing goes wrong, the routine exits.

```
procedure TSearcher.PDFSearch(aRequest: TRequest; aResponse: TResponse);

Var
  I : Integer;
  J : TJSONObject;
  A : TJSONArray;

begin
  aResponse.ContentType:='application/json';
  if AllowCORS then
    AResponse.SetCustomHeader('Access-Control-Allow-Origin','*');
  if not CheckParams(aRequest,aResponse) then
    exit;
  if not InitSearch(aResponse) then
    begin
    aResponse.Code:=500;
    aResponse.CodeText:='Internal error';
    aResponse.SendResponse;
    exit;
    end;
  ConfigSearch(aRequest,aResponse);
```

The `ConfigSearch` will configure the FPC index search object from the query variables:

```
procedure TSearcher.ConfigSearch(aRequest: TRequest; aResponse: TResponse);

Var
  S : string;
  O : TSearchOptions;
  B : Boolean;
```

```
begin
  FMinRank:=StrToIntDef(aRequest.QueryFields.Values['r'],0);
  if FMinRank=0 then
    FMinRank:=FDefaultMinRank;
  S:=aRequest.QueryFields.Values['m'];
  if (S='') or Not TryStrToBool(S,FIncludeMetaData)  then
    FIncludeMetaData:=FDefaultMetaData;
  FSearch.SetSearchWord(aRequest.QueryFields.Values['q']);
  O:=[];
  S:=aRequest.QueryFields.Values['c'];
  if (S<>'') and TryStrToBool(S,B) and B then
    Include(O,soContains);
  FSearch.Options:=O;
end;
```

When the call to the `CongfigSearch` routine has configured the search mechanism, the `PDFSearch` routine starts the actual search:

The call to `FSearch.Execute` will execute the necessary SQL statements on the index database. The results will be available in the `Results` and `RankedResults` array properties. The latter will be combined with the articles array in the `SearchDataToJSON` routine to form the actual result set:

```
FSearch.Execute;
A:=nil;
J:=TJSONObject.Create;
try
  if FIncludeMetadata then
    J.Add('metaData',FMetadata.Clone);
  A:=TJSONArray.Create;
  For I:=0 to Search.RankedCount-1 do
    begin
    if Search.RankedResults[I].Rank>=MinRank then
      A.Add(SearchDataToJSON(I+1,Search.RankedResults[I]));
    end;
  J.Add('data',A);
  SendJSON(J,aResponse);
finally
  J.Free;
end;
```

The result is combined with the metadata if needed, and is sent to the browser with the `SendJSON` command, just as in the `WordList` routine.

The search data is combined with the article list in the `SearchDataToJSON` routine, which starts by finding the article which matches the page of the word match (stored in `Position`). The found article and the search result are then unified in a single JSON structure:

```
function TSearcher.SearchDataToJSON(aID: Integer; const aRes: TSearchWordData
  ): TJSONObject;

Var
  Article : TArticleData;
```

```
begin
  Article:=FindArticle(aRes.Position,aRes.URL);
  Result:=TJSONObject.Create([
    'id',aID,
    'rank',aRes.Rank,
    'articlePage', aRes.Position,
    'articleTitle', Article.Title,
    'articleAuthor', Article.Author,
    'articleIssue', Article.Issue,
    'url',aRes.URL,
    'context',ares.Context,
    'date',FormatDateTime('yyyy"-"mm"-"dd"T"hh":"nn":"ss',aRes.FileDate)
  ]);
end;
```

The routine to find the articles is simplicity itself:

```
function TSearcher.FindArticle(Position: Integer; const aFile: String
  ): TArticleData;

Var
  A : TArticleData;

begin
  Result:=Default(TArticleData);
  if Length(Articles)=0 then
    LoadArticles;
  for A in Articles do
    if A.Match(aFile,Position) then
      Exit(A);
end;
```

The `LoadArticles` routine executes a simple select SQL query and stores the results in the `Articles` array, part of the `TSearcher` class. The interested reader can consult the code supplied with the articles for the details. The `Match` method of the `TArticleData` record returns `True` if the article is located in the PDF and contains the page number:

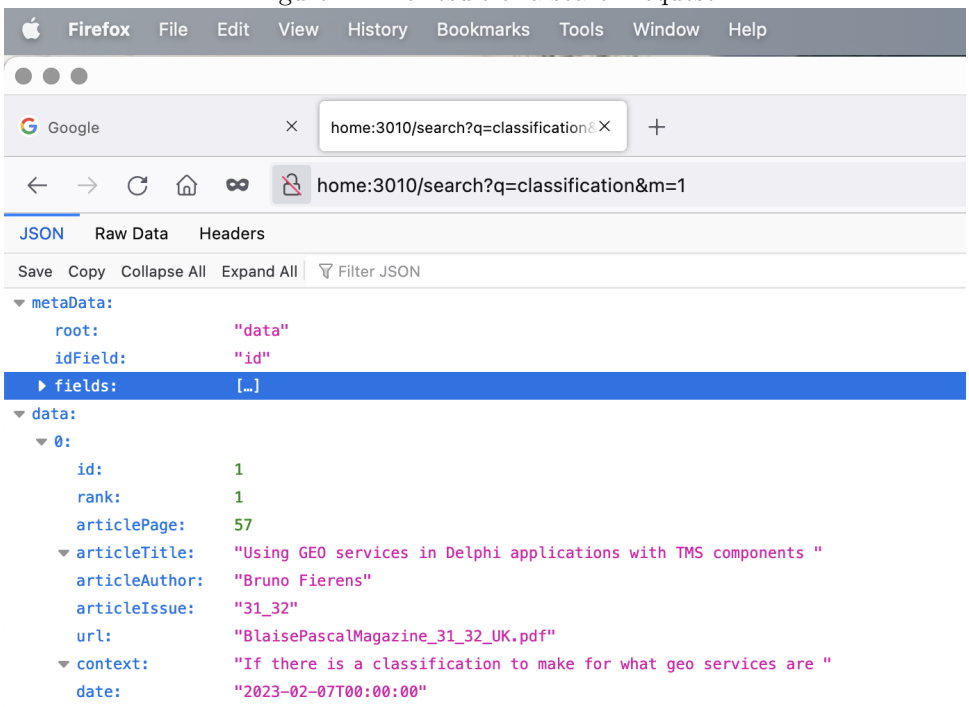
```
function TArticleData.Match(aPDF : String;
                            aPage : Integer) : boolean;
begin
  Result:=SameText(Pdf,aPdf)
          and (StartPage<=aPage)
          and (aPage<=EndPage)
end;
```

With this routine we've covered the search mechanism in the index database. Again, the result can easily be tested in the browser, as shown in figure 2 on page 13.

5 The browser application

Now that we have an API in place for getting word lists and matches of a word in our collection of PDF documents, we can create a browser application that uses these APIs to query and show a collection of PDF documents.

Figure 2: The result of a search request



To do so, we will refactor the PDF viewing application presented in an earlier contribution, and change it so we can either search locally in the shown PDF or look in the collection of PDFs on the server.

To do so, we start by refactoring the application: initially, the application consisted of a single application class. This will now be separated in 4 different classes:

TPDFPanel this class abstracts away the PDF viewer `PDF.js`. It has a method to load a PDF, a method to set the current page, and 2 events that are called when the PDF is loaded and the page is shown, respectively.

```
TPDFPanel = Class(TComponent)
  Procedure QueueRenderPage(aNum : Integer);
  Procedure ShowPDF(aSource : TPDFSource; AtPage : Integer = 1);
  Property CanvasID : String;
  Property PageRendering : Boolean;
  Property Scale : Double;
  Property DisplayedPage : Integer;
  Property PageCount : Integer;
  Property OnPageLoaded : TNotifyEvent;
  Property OnLoaded : TNotifyEvent;
  Property PDFDoc : TPDFDocumentProxy;
end;
```

The properties are pretty straightforward: The `CanvasID` can be set to the ID of the canvas element in which the PDF element is drawn. The `Scale` property is there to set the zoom factor for the PDF. The read-only properties `PageRendering`, `DisplayedPage`, `PageCount` and `PDFDoc` have obvious meanings.

TPDFControlPanel This class contains the controls for the PDF viewer: The page navigation buttons and the zoom button. The component has in fact only some public properties to set the PDF panel and the IDs of various HTML elements.

```
TPDFControlPanel = class(TComponent)
  Procedure BindElements;
  property PDFPanel : TPDFPanel Read FPDFPanel Write SetPDFPanel;
  Property EditPageID : string;
  Property PreviousButtonID : string;
  Property ZoomLabelID : string;
  // Properties for all other buttons
end;
```

The `BindElements` call will fetch the references to all needed elements in the HTML page.

TPDFSearchControl This class takes care of the actual search mechanism: it has some properties for the various IDs of the HTML tags that it needs to do its job, and some methods to show/hide/clear the search results panel .

```
TPDFSearchControl = class(TComponent)
  procedure BindElements;
  Procedure ShowResultPanel;
  Procedure HideResultPanel;
  Procedure ClearResultPanel;
  Property pdfPanel : TPDFPanel;
  Property LocalCheckboxID : String;
  Property SearchButtonID : String;
  Property SearchEditID : String;
  Property DivAutoCompleteID : String;
  Property ResultPanelID : String;
  Property SidebarPanelID : String;
  Property ShowResultsPanelButtonID;
end;
```

Again, the `BindElements` call will fetch the references to all needed elements in the HTML page.

The code of `TPDFControlPanel` and `TPDFPanel` classes can be found in the previous article on showing a PDF, so it will not be repeated here. The only changed code is that the `TPDFControlPanel` class does not access the low-level `TPdfDoc` class, but instead uses the `TPDFPanel` methods (which in turn of course call the methods of the `TPdfDoc` class).

The `TPDFIndexApp` application class ties everything together: it has instances of the above three classes, and some references to the HTML elements to load a PDF file from disk:

```
TPDFIndexApp = class(TBrowserApplication)
  pnlSidebar,
  lblFileLocation : TJSHTMLElement;
  btnLoad,
  btnClosePane : TJSHTMLButtonElement;
  edtPDFFile: TJSHTMLInputElement;
```

```
  FPDFPanel : TPDFPanel;
  FPDFControls : TPDFControlPanel;
  FPDFSearch: TPDFSearchControl;
  procedure doRun; override;
  procedure BindElements;
  function DoLoadFile(Event: TEventListenerEvent): boolean;
  procedure onLoad(aEvent: TJSEvent);
  procedure ShowPDF(aSource: TPDFSource);
  Procedure DisplayFileLocation(const aLocation : String);
  procedure onClosePane(aEvent: TJSEvent);
end;
```

The various event handlers in the application class will not be treated here, the code has not changed from the code presented in the article on showing a PDF file.

The `DoRun` code has changed, so we will present it here:

```
procedure TPDFIndexApp.doRun;

const
  TheURL = 'https://mozilla.github.io/pdf.js/build/pdf.worker.js';

begin
  pdfjsLib.GlobalWorkerOptions.workerSrc:=TheURL;
  FPDFPanel:=TPDFPanel.Create(Self);
  FPDFControls:=TPDFControlPanel.Create(Self);
  FPDFControls.PDFPanel:=FPDFPanel;
  FPDFSearch:=TPDFSearchControl.Create(Self);
  FPDFSearch.PDFPanel:=FPDFPanel;
  BindElements;
  Terminate;
end;
```

As can be seen in the code, the code is reduced to creating the various panels and passing a reference to the `TPDFPanel` instance to the `TPDFControls` panel.

The `BindElements` code is much reduced, since most of the elements have now been moved to the other classes. Their `BindElements` method is called from the application class `BindElements` method.

```
procedure TPDFIndexApp.BindElements;
begin
  btnLoad:=TJSHTMLButtonElement(GetHTMLElement('btnLoad'));
  btnLoad.addEventListener('click', @onLoad);
  lblFileLocation:=GetHTMLElement('lblFileLocation');
  edtPDFFile:=TJSHTMLInputElement(GetHTMLElement('edtPDFFile'));
  edtPDFFile.onchange:=@DoLoadFile;
  btnCloseSidebar:=TJSHTMLButtonElement(GetHTMLElement('btnClosePane'));
  btnCloseSidebar.addEventListener('click', @onClosePane);
  pnlSidebar:=TJSHTMLInputElement(GetHTMLElement('pnlSidebar'));
  FPDFPanel.CanvasID:='PDFCanvas'; // Will call bindelements
  FPDFControls.BindElements;
  FPDFSearch.BindElements
end;
```

The other elements and events are related to loading a user-selected PDF file: the

original functionality of being able to search any file is preserved.

The `TPDFPanel` and `TPDFControlPanel` classes do not contain code that differs from the article on showing a PDF file, so their code will not be repeated here. The `TPDFSearchControl` does change, since now 2 search mechanisms must be supported, as well as a mechanism to show an auto-complete list.

We'll start with the auto-complete list. It is activated in the `oninput` event of the edit element. The event handler is set in the `BindElements` method=

```
edtSearch:=TJSHTMLInputElement(GetHTMLElement(SearchEditID));
edtSearch.onkeyup:=@DoSearchKeyUp;
edtSearch.oninput:=@DoCompleteWord;
```

The `DoCompleteWord` method is responsible for starting (or restarting) a timer (200 ms), and when the timer expires, the word completion list is fetched. Fetching the word completion list is done by opening a dataset:

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

  procedure DoServerSearchWord;

  begin
    if Length(edtSearch.Value)>1 then
      begin
      FSearchTerm:=edtSearch.Value;
      FWords.Close;
      FWords.Load([],Nil);
      end;
  end;

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

As mentioned in the first part of this article, the result of the server calls can be in a format that is suitable for a rest dataset. The `FWords` variable in the above code is a `TRestDataset`, and it is initialized in the `SetupDatasets` method of the `TPDFSearchPanel` class:

```
procedure TPDFSearchControl.SetupDatasets;

Const
  ServerURL = 'http://localhost:3010/';

begin
  FConn:=TRESTConnection.Create(Self);
  FConn.BaseURL:=ServerURL;
  FConn.OnGetURL:=@DogetURL;
  FResult:=TRestDataset.Create(Self);
  FResult.Connection:=FConn;
  Fresult.AfterOpen:=@DoOpenResults;
```

```
  FWords:=TRestDataset.Create(Self);
  FWords.Connection:=FConn;
  FWords.AfterOpen:=@DoWordsOpen;
end;
```

The method also sets up the `FResult` dataset, which will handle the result of the search for word matches, later on.

The `FConn.OnGetURL` is an event called by the `TRestConnection` component, which can be used to construct the URL for a dataset that gets its data from a REST server. In our implementation, we need to choose the correct endpoint depending on what dataset is opened:

```
procedure TPDFSearchControl.DogetURL(Sender: TComponent;
                                     aRequest: TDataRequest;
                                     var aURL: String);

var
  Q : String;

begin
  Q:=encodeURIComponent(FSearchTerm);
  if aRequest.Dataset=FResult then
   aURL:=FConn.BaseURL+'search?m=1&q='+q
 else
   aURL:=FConn.BaseURL+'list?t=contains&m=1&q='+q;
end;
```

You can see from the above that the 2 endpoints we defined in the server are used for both datasets. In both cases the query variables are set up to request metadata, and the `q` is set to the `FSearchTerm` variable, which is the value of the `edtSearch` HTML input tag.

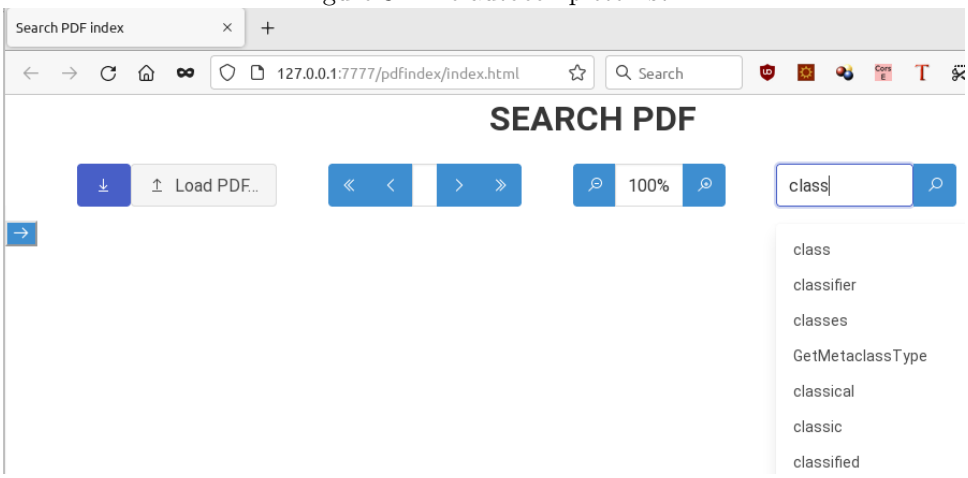
The `AfterOpen` event of the `TDataset` class is used to generate the HTML for the word completion list. The HTML for this completion list is actually quite simple. The edit control is wrapped in 2 classes:

```
<p class="control is-small">
  <div class="dropdown">
    <div class="dropdown-trigger">
      <input id="edtSearch" class="input" style="max-width: 15em;">
      <div class="dropdown-menu" id="mnuAutoComplete" role="menu" />

    </div> <!-- .dropdown-trigger -->
  </div> <!-- .dropdown -->
</p>
```

And the html for the completion list is inserted below the tag with id `mnuAutoComplete`. It is generated by looping over the records in the dataset and generating the necessary HTML elements. The routine starts by hiding the element, clearing the content and then adding the elements:

```
procedure TPDFSearchControl.DoWordsOpen(DataSet: TDataSet);

Var
```

Figure 3: The autocomplete list




```
  S : String;
  F : TField;
  P : TJSHTMLELement;
  A : TJSHTMLAnchorElement;

begin
  mnuAutoComplete.style.setProperty('display','none');
  mnuAutoComplete.InnerHTML:='<div class="dropdown-content"></div>';
  P:=TJSHTMLELement(mnuAutoComplete.firstElementChild);
  if Dataset.RecordCount<=0 then
    exit;
  F:=Dataset.FieldByName('Word');
  While Not Dataset.EOF do
   begin
   S:=F.AsString;
   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);
   Dataset.Next;
   end;
  mnuAutoComplete.style.setProperty('display','block');
end;
```

Note that the 'OnClick' event is set for every word. At the end, the menu is made visible again. figure 3 on page 18 shows what the effect is of this code. Needless to say, using some CSS the list can be made to look much nicer. The `OnClick` event handler on the words sets the word in the search edit, which is a really simple operation:

```
procedure TPDFSearchControl.DoWordSelected(Event: TJSEvent);

begin
```

```
  event.PreventDefault;
  edtSearch.value:=event.targetelement.innerText;
  mnuAutoComplete.style.setProperty('display','none');
end;
```

When the user presses `Enter` in the search edit, or presses the search button, the resulting action depends on the value of the `cbLocal` checkbox: when checked, a local search is performed. If unchecked, then the server is queried:

```
function TPDFSearchControl.DoSearchKeyUp(aEvent: TJSKeyBoardEvent): boolean;

begin
  Result:=False;
  if (aEvent.Key<>TJSKeyNames.Enter) then
    exit;
  onSearch(aEvent);
end;

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

begin
  aTerm:=edtSearch.Value;
  if Length(aTerm)<=2 then
    exit;
  if cbLocal.Checked then
    begin
    if not assigned(pdfPanel) then
      exit;
    DoLocalSearch(aTerm);
    end
  else
    DoIndexSearch(aTerm);
end;
```

As you can see, only words of length 3 or longer will be searched. The `DoLocalSearch` is the search mechanism as implemented in the article on showing a PDF, and will not be repeated here.

The `DoIndexSearch` method is the method we are interested in, and it is very simple. It opens the `FResult` dataset:

```
procedure TPDFSearchControl.DoIndexSearch(aTerm : String);

begin
  FResult.Close;
  FSearchTerm:=aTerm;
  FResult.Load();
end;
```

Again, the `AfterOpen` event of the `FResult` dataset is where the real work is done. It is again a simple loop over the dataset. The `TServerResultsMap` is a little helper class, which contains a field definition for every field in the result set (comparable to persistent fields, only created at runtime). Every record from the dataset is stored

in a record of type `TServerMatch` and passed on to a routine `ShowServerMatch`, which generates the HTML for the record.

```
procedure TPDFSearchControl.DoOpenResults(DataSet: TDataSet);

var
  i : Integer;
  aResult: TServerMatch;
  aMap : TServerResultsMap;
  NoFilter : Boolean;

begin
  I:=0;
  ShowResultPanel;
  aMap:=TServerResultsMap.Create(Dataset);
  While not aMap.Dataset.EOF do
    begin
    Inc(I);
    aResult.FromMap(aMap);
    ShowServerMatch(aResult);
    Dataset.next;
    end;
end;
```

For completeness, here is the `TServerMatch` record:

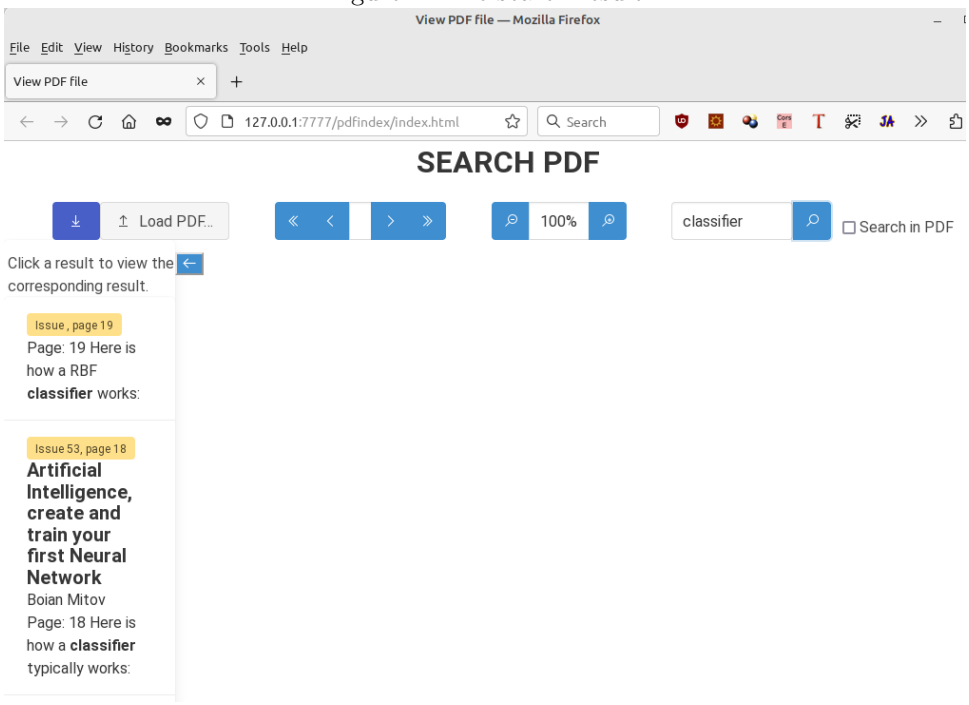
```
TServerMatch = record
  ID : Integer;
  Rank : Integer;
  Page : Integer;
  Issue : String;
  Author : String;
  Title : string;
  URL : String;
  Context : String;
  Date : TDateTime;
  Procedure FromMap(aMap : TServerResultsMap);
end;
```

The `FromMap` copies the values of all dataset fields into the typed record fields. The `ShowResultMatch` routine uses the `TServerMatch` record to generate the HTML for a match. Basically, this routine is a set of search-and-replace operations on a HTML template, and the resulting HTML is inserted in a HTML element for which the `OnCLick` is then set.

The `ResultContent` constant contains the HTML template, it is not shown here, the interested reader can find it in the source code. It has variable placeholders in {{ }} brackets: the name in the brackets is a field name and is replaced by the value of the named field.

```
function TPDFSearchControl.ShowServerMatch(aResult: TServerMatch
  ): TJSHTMLElement;

Var
  aReplace, Content, Res : String;
```

Figure 4: The search result

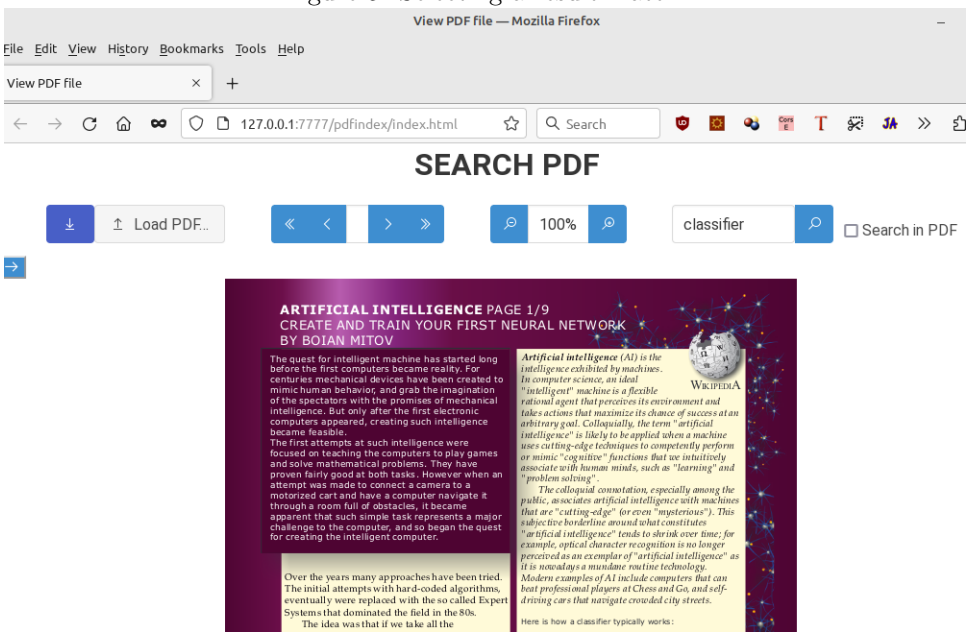



```
begin
  Result:=TJSHTMLElement(Document.CreateElement('a'));
  Result.Dataset['page']:=IntToStr(aResult.Page);
  Result.Dataset['idx']:=IntToStr(aResult.ID);
  Result.Dataset['title']:=aResult.Title;
  Result.Dataset['url']:=aResult.url;
  Result.ClassName:='panel-block result-item';
  Res:=ResultContent;
  Res:=StringReplace(Res,'{{page}}',IntToStr(aResult.Page),[rfReplaceAll]);
  Res:=StringReplace(Res,'{{author}}',aResult.Author,[rfReplaceAll]);
  Res:=StringReplace(Res,'{{issue}}',aResult.Issue,[rfReplaceAll]);
  Res:=StringReplace(Res,'{{title}}',aResult.Title,[rfReplaceAll]);
  aReplace:=StringReplace(Highlight,'{{match}}',edtSearch.Value,[rfReplaceAll]);
  Content:=StringReplace(aResult.Context,edtSearch.Value,aReplace,[rfReplaceAll]);
  Res:=StringReplace(Res,'{{content}}',Content,[]);
  Writeln('Res : ',Res);
  Result.InnerHTML:=Res;ku
  Result.AddEventListener('click',@DoServerMatchSelected);
  pnlResults.appendChild(Result);
end;
```

Note that the HTML element in which the generated HTML is inserted (the `Result` HTML element) has several data attributes with all information needed to locate the article from which the match forms a part. The result of this code is shown in figure 4 on page 21. The last piece of the puzzle is the `DoServerMatchSelected` event handler, which is invoked when the user selects a match in the result list.

It is actually a quite simple method. It uses the data-attributes of the generated

Figure 5: Selecting a result match.



HTML to select the PDF and page to show. If there is a valid PDF name and page number, the `ShowPDF` method of the `TPDFPanel` instance is used to show the correct page.

```
procedure TPDFSearchControl.DoServerMatchSelected(aEvent: TJSEvent);
var
  aPage : Integer;
  aTitle,aURL : string;
  Src : TPDFSource;
  El : TJSHTMLElement;

begin
  aEvent.currentTargetElement.classList.add('is-active');
  El:=TJSHTMLElement(aEvent.currentTargetElement);
  aPage:=StrToIntDef(String(El.Dataset['page']),-1);
  aTitle:=String(El).Dataset['title']);
  aUrl:=El.Dataset['url'];
  if (aPage<>-1) and (aUrl<>'') then
    begin
    Src:=TPDFSource.new;
    Src.url:=ServerURL+'pdf/'+aUrl;
    pdfPanel.ShowPDF(Src,aPage);
    HideResultPanel;
    end;
end;
```

As a last step, the results panel is closed to maximize the space for the PDF viewer. The user can open the results panel with a button, and select another match, if so desired. The result can be seen in figure 5 on page 22

6 Conclusion

In this second article on PDF indexing, we showed how to use the PDF index database constructed in the previous article on PDF indexing. All this was accomplished with standard components of Free Pascal and Pas2JS. Although the resulting mechanism works satisfactorily and does what is needed, there is room for improvement. For example the search mechanism can be switched to manticore search (which has more options than the FPC mechanism). Additionally the client code can be reduced by using some of the data-aware widgets distributed by pas2js instead of the custom code written here. These improvements will be the subject of a future contribution.