Viewing PDF files in the browser using Pas2JS

Michaël Van Canneyt

August 25, 2022

Abstract

PDF is probably the most used document format on the web. You don't need to install any special software to view it: The browser can perfectly display a PDF. The technology to view a PDF is available as a Javascript library and so we can use it in pas2js. In this article we show how.

1 Introduction

Chances are that if you order some product or service online, you will get an invoice or order confirmation as a PDF document. When properly configured, the browser will display this document for you, without the need to save it and open an application such as Acrobat Reader or Foxit reader.

The browser can do this thanks to a Javascript library (pdf.js) developed by the Mozilla team: This javascript library takes care of everything, and can even create a complete GUI to navigate the PDF, search, select text etc. The implementation is very complete, and will result in a very qualitative rendering of a PDF.

Since it is Javascript and open source, we can use this library to handle display of a PDF ourselves in a Pas2JS program.

2 The javascript

The Pdf.js implementation is available on github:

```
https://github.com/mozilla/pdf.js
```

The library comes in 2 versions: one for older browsers, one for more modern browsers. In this text, we'll assume a modern browser.

There are 2 parts to the library: the first part is the actual PDF.js API, which can parse and draw a PDF file. The second part is a viewer, which provides a standard UI for manipulating the PDF: browsing through the pages, searching, copying text etc. In this article, we'll limit ourselves to the first part and use that to build a small PDF viewing application.

Pre-built versions of this library can be found on:

```
https://github.com/mozilla/pdfjs-dist
```

To include it in your pas2js program, the following line can be added to your HTML page:

```
<script src="https://mozilla.github.io/pdf.js/build/pdf.js"></script>
```

That is sufficient to get started.

The API of PDF.js exists as a typescript declaration module, and this has been translated to pascal using the `dts2pas` tool that comes with pas2js. The resulting file has been included in the pas2js distribution as a unit called `pdfjs`. This unit is not to be confused with the `jspdf` unit in the jsPDF package which also exists, but serves to generate a PDF file using Javascript, instead of displaying it.

The Javascript library makes heavy use of promises: the heavy lifting is done using web workers (a kind of threads), and communication with the web workers is done through asynchronous messages.

There is a global object in the library which has some configuration variables that can be set, and which also contains the starting point for displaying PDF files.

The global object is declared as follows:

```
TPDFJSStatic = class external name 'Object' (TJSObject)
  maxImageSize : Double;
  cMapUrl : string;
  cMapPacked : boolean;
  disableFontFace : boolean;
  imageResourcesPath : string;
  disableWorker : boolean;
  workerSrc : string;
  disableRange : boolean;
  disableStream : boolean;
  disableAutoFetch : boolean;
  pdfBug : boolean;
  postMessageTransfers : boolean;
  disableCreateObjectURL : boolean;
  disableWebGL : boolean;
  disableFullscreen : boolean;
  disableTextLayer : boolean;
  useOnlyCssZoom : boolean;
  verbosity : Double;
  maxCanvasPixels : Double;
  openExternalLinksInNewWindow : boolean;
  isEvalSupported : boolean;
  GlobalWorkerOptions :  TGlobalWorkerOptions;
  Procedure PDFSinglePageViewer(params : TPDFViewerParams);
  Procedure PDFViewer(params : TPDFViewerParams);
  Function getDocument(url : string;
        pdfDataRangeTransport : TPDFDataRangeTransport;
        passwordCallback : TgetDocument_passwordCallback;
        progressCallback : TgetDocument_progressCallback):
        TPDFLoadingTask;
  Function getDocument(url : string): TPDFLoadingTask;
  Function getDocument(url : string;
        pdfDataRangeTransport : TPDFDataRangeTransport): TPDFLoadingTask;
  Function getDocument(url : string;
        pdfDataRangeTransport : TPDFDataRangeTransport;
        passwordCallback : TgetDocument_passwordCallback): TPDFLoadingTask;
  Function getDocument(data : jsvalue;
        pdfDataRangeTransport : TPDFDataRangeTransport;
        passwordCallback : TgetDocument_passwordCallback;
```

```
        progressCallback : TgetDocument_progressCallback): TPDFLoadingTask;

  Function getDocument(data : jsvalue): TPDFLoadingTask;
  Function getDocument(data : jsvalue;
        pdfDataRangeTransport : TPDFDataRangeTransport): TPDFLoadingTask;
  Function getDocument(data : jsvalue;
        pdfDataRangeTransport : TPDFDataRangeTransport;
        passwordCallback : TgetDocument_passwordCallback): TPDFLoadingTask;
  Function getDocument(source : TPDFSource;
        pdfDataRangeTransport : TPDFDataRangeTransport;
        passwordCallback : TgetDocument_passwordCallback;
        progressCallback : TgetDocument_progressCallback): TPDFLoadingTask;
  Function getDocument(source : TPDFSource): TPDFLoadingTask;
  Function getDocument(source : TPDFSource;
        pdfDataRangeTransport : TPDFDataRangeTransport): TPDFLoadingTask;
  Function getDocument(source : TPDFSource;
        pdfDataRangeTransport : TPDFDataRangeTransport;
        passwordCallback : TgetDocument_passwordCallback): TPDFLoadingTask;
end;
```

The meaning of most of these properties are pretty obvious from their names, we will not discuss them here except the ones we need.

This global object is not declared in the pdfjs library, but can be inserted in your program as follows:

```
var
  pdfjsLib : TPDFJSStatic; external name 'pdfjsLib';
```

As you can see, there are only 3 calls in this global object:

PDFSinglePageViewer Creates a single-page PDF viewer UI.

PDFViewer Creates a multi-page PDF viewer UI.

getDocument download a document (or obtain one from in-memory data) and parse it. a Task object is returned.

Before using these calls, however, the `workerSrc` property must be set: this is the location of the worker script that contains the background processing logic.

It can be set as follows:

```
pdfjsLib.GlobalWorkerOptions.workerSrc:= 'https://mozilla.github.io/pdf.js/build/
```

3 A small demo

To demonstrate this API, we'll create a simple pas2js application that allows to enter the URL of a PDF. It will download and show the PDF. To navigate, we'll add the usual previous/next/first/last buttons and similar buttons to set the zoom. We will use again bulma as a CSS framework.

For this, in the Project - New project menu of the Lazarus IDE, we select the 'Web Browser Application" type, and we select the option to use the browser application class. All code will be put in the browser application class.

The HTML for the navigation buttons (2 buttons, an edit for a page number and again 2 buttons) looks like this:

```
<div class="field has-addons">
  <p class="control">
    <button id="btnFirst" class="button  is-info">
      <span class="icon is-small">
        <i class="las la-angle-double-left"></i>
      </span>
    </button>
  </p>
  <p class="control">
    <button id="btnPrevious" class="button is-info">
      <span class="icon is-small">
        <i class="las la-angle-left"></i>
      </span>
    </button>
  </p>
  <p class="control is-small">
    <input id="edtPageNo" class="input" style="max-width: 3em;">
  </p>
  <p class="control">
    <button id="btnNext" class="button is-info">
      <span class="icon is-small">
        <i class="las la-angle-right"></i>
      </span>S
    </button>
  </p>
  <p class="control">
    <button id="btnLast" class="button is-info">
      <span class="icon is-small">
        <i class="las la-angle-double-right"></i>
      </span>
    </button>
  </p>
</div>
```

For the zoom buttons and edit, the HTML looks similar, so we will not repeat it here.

As usual, we start the application by binding all elements to an identifier in the application class, and hook up events to the elements:

```
procedure TMyApplication.BindElements;
begin
  btnPrevious:=TJSHTMLButtonElement(GetHTMLElement('btnPrevious'));
  btnPrevious.addEventListener('click', @onPrevPage);
  btnFirst:=TJSHTMLButtonElement(GetHTMLElement('btnFirst'));
  btnFirst.addEventListener('click', @onFirstPage);
  btnNext:=TJSHTMLButtonElement(GetHTMLElement('btnNext'));
  btnNext.addEventListener('click', @onNextPage);
  btnLast:=TJSHTMLButtonElement(GetHTMLElement('btnLast'));
  btnLast.addEventListener('click', @onLastPage);
  btnLoad:=TJSHTMLButtonElement(GetHTMLElement('btnLoad'));
  btnLoad.addEventListener('click', @onLoad);
```

```
  btnZoomIn:=TJSHTMLButtonElement(GetHTMLElement('btnZoomIn'));
  btnZoomIn.addEventListener('click', @onZoomIn);
  btnZoomOut:=TJSHTMLButtonElement(GetHTMLElement('btnZoomOut'));
  btnZoomOut.addEventListener('click', @onZoomOut);
  lblZoom:=GetHTMLElement('lblZoom');

  edtPageNo:=TJSHTMLInputElement(GetHTMLElement('edtPageNo'));
  edtPageNo.onkeyup:=@DoPageKeyUp;
  FCanvas:=TJSHTMLCanvasElement(GetHTMLElement('PDFCanvas'));
  FCtx:=TJSCanvasRenderingContext2D(FCanvas.getContext('2d'));
end;
```

The purpose of the buttons and edits are clear: they will be the UI elements for our application. The last 2 lines need some explanation: The PDF.js API needs a canvas element to draw the PDF. The application that makes use of the PDF.js API must provide the 2D rendering context this element.

In our HTML page we have included such an element below the buttons:

```
<!-- pdf -->
<div class="is-flex is-justify-content-center">
  <canvas id="PDFCanvas" height="737" width="538"></canvas>
</div>
```

and the code in the last 2 lines of the `BindElements` routine saves a reference to the 2D rendering context of the canvas element.

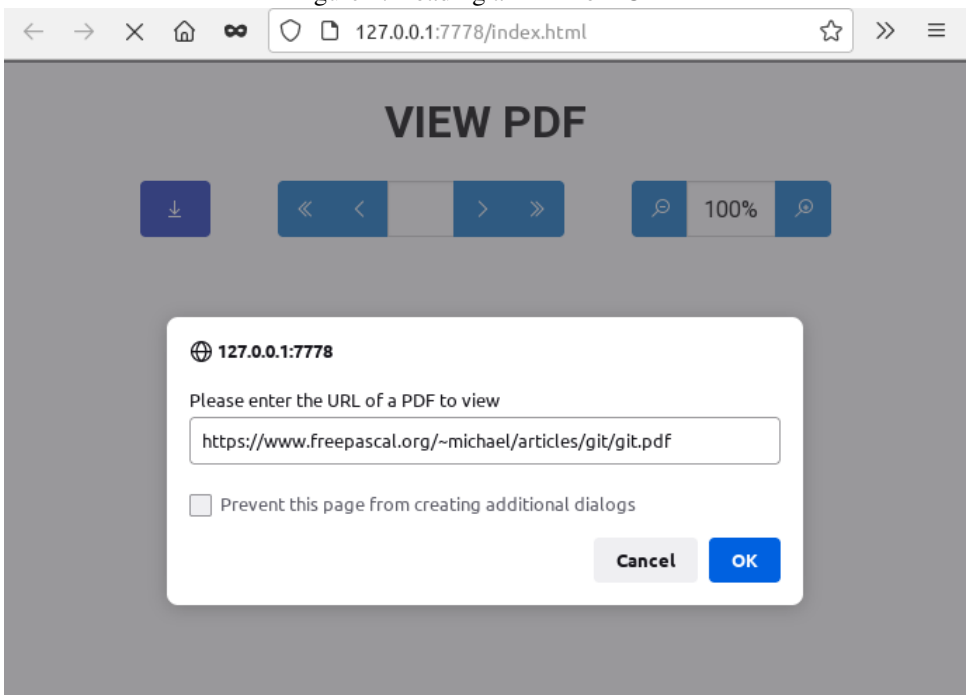
The `btnLoad` element is the button to press when you wish to select a URL to download. The `onClick` handler is not very exciting:

```
procedure TMyApplication.onLoad(aEvent: TJSEvent);
begin
  StartPDFRender;
end;

procedure TMyApplication.StartPDFRender;

var
  aSource : TPDFSource;
  aURL : String;

begin
  aUrl:=window.prompt('Please enter the URL of a PDF to view');
  if IsNull(aURL) then
    exit;
  aSource:=TPDFSource.New;
  aSource.URL:=aURL;
  ShowPdf(aSource);
end;
```

The `StartPDFRender` routine is also called when the application is started. The result of this routine looks like figure 1 on page 6. The last 3 lines create a `TPDFSource` class, which is the input expected by the `Pdf.js` API. It passes the created class to the `ShowPDF` routine.

This routine does the actual work, using the `getDocument` call mentioned earlier:

Figure 1: Loading a PDF from URL




```
procedure TMyApplication.ShowPDF(aSource: TPDFSource);

  function pdfLoaded(aValue : JSValue) : JSValue;
  begin
    pdfDoc:=TPDFDocumentProxy(aValue);
    Writeln('PDF loaded');
    FpageNum:=1;
    QueueRenderPage(FpageNum);
    Result:=True;
  end;

  Procedure pdfLoadError(aReason : String);

  begin
    Writeln('PDF loading failed: '+aReason);
  end;

var
  loadingTask : TPDFLoadingTask;

begin
  loadingTask:=pdfjsLib.getDocument(aSource);
  loadingTask.promise.&then(@pdfLoaded,@pdfLoadError);
end;
```

The `loadingTask` has a promise member which we can use to wait for the PDF to load: The loading of a PDF is an asynchronous operation, and we can only display the PDF when the PDF has been downloaded and parsed. The promise resolves into a `TPDFDocumentProxy` class which can be used to execute commands on the downloaded and parsed PDF. The

`PdfLoaded` routine does exactly that: it saves the PDF proxy object, and schedules a render of the first page. Again, rendering is asynchronous, so we must be careful not to interrupt a previous rendering task by starting new one. If a page is currently being rendered, we simply save the number of the page to render. If no page is being rendered, we schedule a rendering task for the requested page:

```
procedure TMyApplication.queueRenderPage(aNum: Integer);

begin
  if FPageRendering then
    FpageNumPending:=aNum
  else
    renderPage(aNum)
end;
```

The `RenderPage` routine is where the actual drawing procedure starts. It starts with a setting a variable, and calling the `pdfDoc.GetPage` routine: this will fetch all the data necessary to draw the page: it returns a Promise.

```
procedure TMyApplication.renderPage(aNum: Integer);

begin
  FpageRendering:=True;
  pdfDoc.getPage(aNum).&then(@HavePage);
  edtPageNo.Value:=IntToStr(anum);
end;
```

Note that the page number is set in the edit control. When the page data is fetched, the promise is resolved and the local function `HavePage` is called with a `TPDFPageProxy` value. The `HavePage` function will do the actual painting:

```
function havePage (aValue : JSValue) : JSValue;

var
  page : TPDFPageProxy absolute aValue;
  viewport : TPDFPageViewport;
  renderContext: TPDFRenderParams;
  renderTask : TPDFRenderTask;
  viewportParams : TViewportParameters;

begin
  viewportParams:=TViewportParameters.new;
  viewportParams.scale:=FScale;
  viewport:=page.getViewport(viewportParams);
  Fcanvas.height := viewport.height;
  Fcanvas.width := viewport.width;
  renderContext:=TPDFRenderParams.New;
  renderContext.canvasContext:=Fctx;
  renderContext.viewport:=viewport;
  renderTask:=page.render(renderContext);
  renderTask.promise.&then(@renderOK);
  Result:=True;
end;
```

The drawing is done by preparing some parameters for a `RenderTask`. As the name implies, this task renders a page. This task needs several parameters, one of which is the canvas rendering context which we saved at the beginning of the program.

The canvas is first prepared: the `Width` and `height` are set to match the needed width and height of the PDF page using the current zoom level: these dimensions can be obtained from the `getViewPort` call of the `TPDFPageProxy` object.

All this is then passed to the `Render` call of the Page. This call returns a `TPDFRenderTask` object which is again a promise: when this promise resolves, the page will have actually been drawn on the canvas, and `RenderOK` is called.

Because drawing is asynchronous, we need to check whether the user navigated to another page while the rendering took place, and call `RenderPage` again with the new page number - the page to render was saved in `FPageNumPending` in the `QueueRenderPage` presented earlier.

```
function renderOK(aValue : JSValue) : JSValue;

Var
  N : Integer;

begin
  FPageRendering:=false;
  if (FPageNumPending <> -1) then
    begin
    N:=FPageNumPending;
    FpageNumPending:=-1;
    renderPage(N);
    end;
  Result:=True;
end;
```

The result of all this work can be seen in figure 2 on page 9.

With these routines, the heavy lifting is actually done. Navigation is now easy. We just determine the page that the user wishes to see, and queue the rendering of the desired page. Here are the 'OnClick' events of the 'previous' and 'first' buttons.

```
procedure TMyApplication.onPrevPage(aEvent : TJSEvent);

begin
  if (FpageNum <= 1) then exit;
  Dec(FpageNum);
  queueRenderPage(FpageNum);
end;

procedure TMyApplication.onFirstPage(aEvent : TJSEvent);

begin
  if not assigned(pdfDoc) then exit;
  FPageNum:=1;
  queueRenderPage(FpageNum);
end;
```

The 'Next' and 'Last' buttons have of course similar procedures. The user can enter a page number in the page edit box, and hit the Enter key to jump to the desired page. This is

Figure 2: Showing the loaded PDF

handled in the key up event of the page number edit:

```
function TMyApplication.DoPageKeyUp(aEvent: TJSKeyBoardEvent): boolean;

Var
  aPage : Integer;

begin
  if not assigned(pdfDoc) then exit;
  Result:=False;
  if (aEvent.Key<>TJSKeyNames.Enter) then
    exit;
  aPage:=StrToIntDef(edtPageNo.Value,0);
  if (aPage>0) and (aPage<=PdfDoc.numPages) then
    queueRenderPage(aPage);
end;
```

We've seen in the `HavePage` routine that the rendering of the current page is parametrized with a zoom level. This means that to change the zoom, it suffices to change the zoom level variable (`FScale`), and request a re-rendering of the current page:

```
procedure TMyApplication.onZoomIn(aEvent : TJSEvent);

begin
  if not assigned(pdfDoc) then exit;
  FScale:=FScale+ScaleStep;
  DisplayZoom;
  queueRenderPage(FpageNum);
end;

procedure TMyApplication.DisplayZoom;
begin
  lblZoom.innerText:=IntToStr(Round(FScale*100))+'%';
end;
```

The `OnZoomOut` routine is of course similar to the routine to zoom in.

4 Selecting a file from disk

The previous paragraphs show that it is really not difficult to load a PDF from a URL. In the demo we showed how to load a PDF from a URL. In most programs, the user will of course not need to enter a URL manually: most of the time, the application will know the URL where the PDF can be found.

But what if you want to allow the user to select a PDF from disk for upload to a server, and wish to present the user with a preview of the PDF before actually uploading the file? It would be silly to first upload the file to a temporary area and then download it again to preview it...

Fortunately, showing a file selected from disc can easily be implemented. To do so, we add an 'upload' button to the demo application.

In HTML, an input tag with type "file" can be used to select a file. The Bulma CSS framework has some special CSS classes that can be used to mask this file input edit so it is

invisible, and just shows the button that opens the file picker dialog. The necessary HTML looks like this:

```
<div class="field has-addons mr-6">
  <div class="file">
    <label class="file-label">
      <input id="edtPDFFile" class="file-input" type="file" name="pdfFile">
      <span class="file-cta is-link">
        <span class="file-icon  is-link">
          <i class="las la-upload"></i>
        </span>
        <span class="file-label is-link  ">
          Load PDF...
        </span>
      </span>
    </label>
  </div>
</div>
```

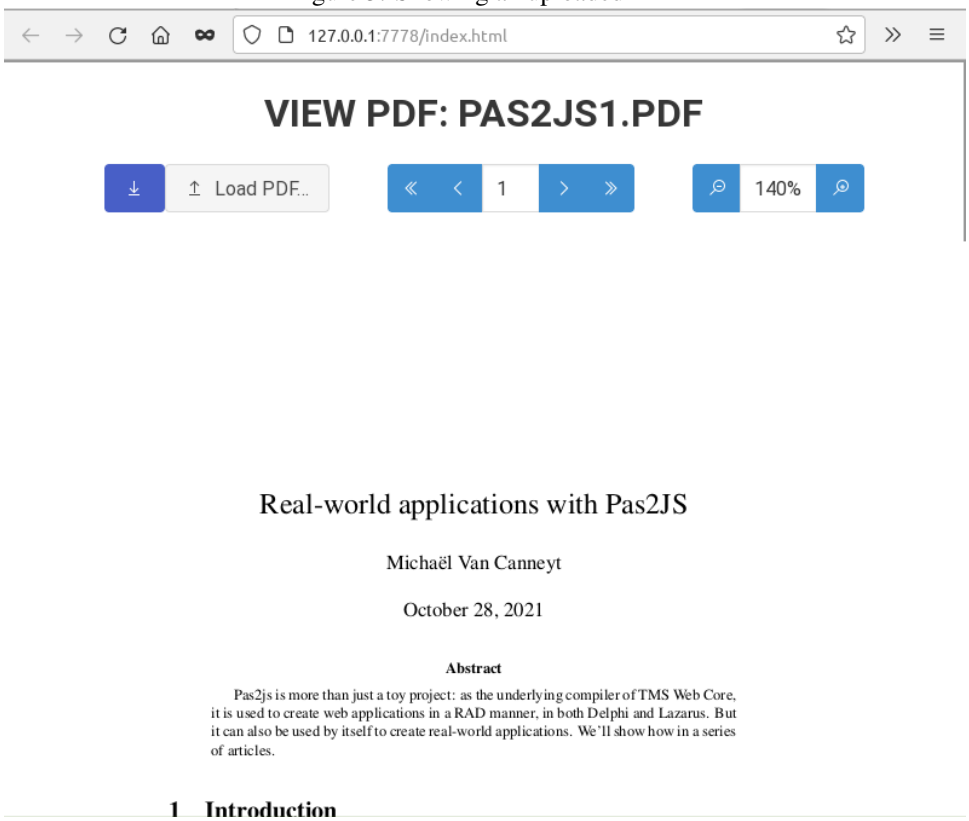
We bind the input element and handle the `OnChange` event:

```
edtPDFFile:=TJSHTMLInputElement(GetHTMLElement('edtPDFFile'));
edtPDFFile.onchange:=@DoLoadFile;
```

The `DoLoadFile` is a short routine.

```
function TMyApplication.DoLoadFile(Event: TEventListenerEvent): boolean;

  Var
    aFileName : string;

    Function LoadFromFile(aData : JSValue) : JSValue;

    var
      aSource : TPDFSource;

    begin
      Result:=False;
      pdfDoc:=Nil;
      aSource:=TPDFSource.New;
      aSource.data:=aData;
      ShowPDF(aSource);
      DisplayFileLocation(aFileName);
    end;

begin
  Result:=False;
  if (edtPDFFile.files.length=1) then
    begin
    aFileName:=ExtractFileName(edtPDFFile.files[0].Name);
    edtPDFFile.files[0].arraybuffer._then(@loadFromFile);
    end;
end;
```

Figure 3: Showing an uploaded PDF



The start of the routine uses the input file API to get the data into an array buffer using the `arraybuffer` function: calling this will load the file into memory, and of course works with a promise. The promise resolves in an array buffer with the data of the file. The real work then happens in the local `LoadFromFile` routine: it creates a `TPDFSource` object, but instead of setting the `url` element of that object (as we did before), it now sets the `data` element. The data element is assumed to contain the data of the PDF file to display.

We add a little routine to display the file location in the browser title bar and the title at the top of the page.

The result of this little piece of code can be seen in figure 3 on page 12.

5 Distributing PDF files on disk

What if you wanted to make and distribute a PWA (Progressive Web Application) with multiple PDF files and allow the user to select one from a list, which is then viewed in the PWA without needing to download it ?

You could include the data from all the PDF files as array variables in the HTML page. But if you have a lot of files, this would cause the page load to be very slow, and the memory consumption could become very high when the files are large.

It would be better to load only the needed PDF into memory. But how to do so without asking the user to select a file ?

This can be done using a small trick:

It is possible at runtime to add a script tag to the HTML page, simply inject

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

into the DOM of the HTML.

This can be used to inject a variable definition in the browser namespace (let's call it `myPDF`), if the `mypdf.js` file contains simply a variable definition, like this:

```
var
  myPDF = atob("JVBERi0xLjcNCiXi48/TDQo5ODIgMCBvYmoNC...");
```

The string literal (truncated in the above sample) is the PDF file contents, encoded as base64, and `atob` is the browser function that allows to decode the base64 string.

By injecting different script tags, the contents of the variable can be changed: the browser will happily redefine the `myPDF` variable every time a new script tag is included.

The contents of the PDF can then be shown using the following code, similar to what was shown earlier. The first line defines the global `myPDF` variable in the Pascal program:

```
var
  myPDF : JSValue; external name 'myPDF';

Procedure LoadFromFile;

var
  aSource : TPDFSource;

begin
  pdfDoc:=Nil;
  aSource:=TPDFSource.New;
  aSource.data:=myPDF;
  ShowPDF(aSource);
end;
```

As you can see, the code is no different from what was presented earlier, the source of the PDF data is simply a global variable.

The following code is part of a class called `TFileToJSBase64Var`, and can be used to create the script file:

```
procedure TFileToJSBase64Var.Convert(aInputStream, aOutputStream: TStream);

Var
  BES : TBase64EncodingStream;
  VN,SPrefix,SSuffix : UTF8String;

begin
  // Set the variable name from the VarName property.
  VN:=VarName;
  if VN='' then
    VN:='file';
  SPrefix:='var '+VN+' = ';
  SSuffix:='"';
  // if UseAToB is true,
  // we include the call to atob in the variable definition:
```

```
  if UseAToB then
    begin
    SPrefix:=SPrefix+'atob(';
    SSuffix:=SSuffix+')';
    end;
  SPrefix:=SPrefix+'"';
  SSuffix:=SSuffix+';'#10;
  aOutputStream.WriteBuffer(SPrefix[1],Length(SPrefix));
  BES:=TBase64EncodingStream.Create(aOutputStream);
  try
    BES.SourceOwner:=False;
    Bes.CopyFrom(aInputStream,0);
    BES.Flush;
    aOutputStream.WriteBuffer(SSuffix[1],Length(SSuffix));
  finally
    BES.Free;
  end;
end;
```

The full class is available with the source code of this article in the `Filetojsvar` unit.

By creating a Javascript file for each PDF you wish to include, you can inject the appropriate file into the DOM after the user selected a PDF, and display it with the above code.

This can for example be used to distribute a USB stick with PDFs and include a web page that can be used to view the PDF files. No actual executables will be needed, so this is a completely cross-platform solution to distribute PDFs with a PDF viewer included.

6 Adding a search option

The PDF.js library allows you to retrieve the text fragments in the PDF. This functionality can be used to add an option to search the text in the PDF.

For demonstration purposes, we'll add a double mechanism to the PDF viewer. The user can enter a search term in an edit box. When matches are found, he or she can browse through the pages that contain a match. For this, we add the following HTML, which uses a popup mechanism to show the matching pages browser;

```
<div class="field has-addons ml-6">
  <p class="control is-small">
    <div class="popover is-popover-bottom is-not-popover-hover"
        id="searchPopover">
      <input id="edtSearch" class="input" style="max-width: 15em;">
      <div class="popover-content">
        Page <span id="lblCurrentSearchResult">1</span>/
             <span id="lblSearchResultCount">10</span>
        <span class="icon is-small ml-2" id="btnNextSearchResult">
          <i class="las la-angle-down"></i>
        </span>
        <span class="icon is-small" id="btnPreviousSearchResult">
          <i class="las la-angle-up"></i>
        </span>
        <button id="btnCloseSearchResult"
                class="delete  ml-4 is-small"></button>
      </div>
```

```
    </div> <!-- .popover -->
  </p>
  <p class="control">
    <button id="btnSearch" class="button is-info">
      <span class="icon is-small">
        <i class="las la-search"></i>
      </span>
    </button>
  </p>
</div>
```

The elements with IDs `searchPopover`, `edtSearch`, `lblCurrentSearchResult`, `lblSearchResultCount` `btnPreviousSearchResult` `btnNextSearchResult`, `btnCloseSearchResult` and `btnSearch` are added as fields to the application class and bound to the HTML tags in the usual manner.

Clicking the `btnSearch` button starts the search, as well as hitting the Enter key in the search edit;

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

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


procedure TMyApplication.onSearch(aEvent: TJSEvent);
begin
  DoSearch;
end;
```

The DoSearch starts the search mechanism, which is implemented in a separate class `TPDFSearch`. An instance of this class is created if it did not yet exist. The search class has a method `searchDocument`, which results in a promise.

```
procedure TMyApplication.DoSearch;

var
  aterm : string;

begin
  if not assigned(pdfDoc) then exit;
  aTerm:=edtSearch.Value;
  if Length(aTerm)<=1 then exit;
  if FSearch=Nil then;
    FSearch:=TPDFSearch.Create(Self);
  FSearch.PDF:=pdfDoc;
  FSearch.searchDocument(aTerm)._then(@OnHaveResults);
end;
```

The promise resolves in a `TPDFSearchResult`, which is defined as follows:

```
TPDFSearchMatch = Record
  Page : Integer;
  Text,
  Context1,
  Context2 : String;
end;
TPDFSearchMatchArray = Array of TPDFSearchMatch;

TPDFSearchResult = Record
  Matches : TPDFSearchMatchArray;
end;
```

For every match in the text, a `TPDFSearchMatch` exists, which has the text, page and 2 pieces of context (surrounding text).

The local method `OnHaveResults` will handle the display of results using the TPDFSearchMatch records. It starts by creating an array of unique page numbers containing a match and saving that in the form variable `FSearchResultPages`:

```
Function OnHaveResults(aValue : JSValue) : JSValue;

var
  Results : TPDFSearchResult absolute aValue;
  Pages : Array of Integer;
  I : Integer;

begin
  Result:=Undefined;
  If Length(Results.Matches)>0 then
    begin
    Pages:=[Results.Matches[0].Page];
    // Create page number list
    For I:=1 to Length(Results.Matches)-1 do
      if TJSArray(Pages).indexOf(Results.Matches[i].Page)=-1 then
        TJSArray(Pages).Push(Results.Matches[i].Page);
    FSearchResultPages:=Pages;
    ShowSearchResultMatchCount;
    ShowSearchResultPage(0);
    FSearchResultsPopover.classList.add('is-popover-active');
    ClearResultPane;
    For I:=1 to Length(Results.Matches)-1 do
      ShowResultMatch(I,Results.Matches[i]);
    ShowResultPane;
    end;
end;
```

After constructing the page array, the count is shown in the popover, and the first page with a result is displayed:

```
procedure TMyApplication.ShowSearchResultMatchCount;

begin
  lblSearchResultCount.InnerText:=IntToStr(Length(FSearchResultPages));
end;
```

```
procedure TMyApplication.ShowSearchResultPage(aIdx: Integer);

begin
  FCurrentSearchResultPage:=aIdx;
  lblCurrentSearchResult.InnerText:=IntToStr(aIdx+1);
  queueRenderPage(FSearchResultPages[aIdx]);
end;
```

The arrow buttons `btnPreviousSearchResult` `btnNextSearchResult` can be used to navigate the page numbers in the `FSearchResultPages` array:

```
procedure TMyApplication.onNextSearchResult(aEvent: TJSEvent);
begin
  if FCurrentSearchResultPage>=(Length(FSearchResultPages)-1) then
    exit;
  ShowSearchResultPage(FCurrentSearchResultPage+1);
end;

procedure TMyApplication.onPreviousSearchResult(aEvent: TJSEvent);
begin
  if FCurrentSearchResultPage<1 then
    exit;
  ShowSearchResultPage(FCurrentSearchResultPage-1);
end;
```

If the user found the page he was looking for, the `btnCloseSearchResult` button can be used to close the popover:

```
procedure TMyApplication.onCloseSearch(aEvent: TJSEvent);
begin
  FSearchResultsPopover.classList.Remove('is-popover-active');
end;
```

Scrolling through the list of pages with a match for the search term is nice, but not altogether satisfying. It is of course easier if the user can see a list of the matches together with some context for each match. For this, we create a side panel (initially hidden) that will show the full result list with more details. The user can scroll through the list, click on a match and jump to the particular page. To make things more interesting, we'll allow the user to filter the search result list as well.

The HTML for our side pane looks like this:

```
<div id="pnlSidebar" class="sidebar sidebar_closed">
  <!-- close/open button -->
  <button id="btnClosePane"
          class="close-button has-background-link has-text-white">
    <i class="las la-arrow-right la-lg" ></i>
  </button>
  <!-- side bar content -->
  <div id="pnlResults" class="panel is-link">

    <!-- 1. title -->
    <p class="panel-heading">
      Search results
    </p>
```

```
    <!-- 2. search bar -->
    <div id="pnlSearch" class="panel-block panel-search">
      <p class="control has-icons-left">
        <input id="edtSearchInResults" class="input is-link"
               type="text" placeholder="Search in results">
        <span class="icon is-left">
          <i class="las la-search" aria-hidden="true"></i>
        </span>
      </p>
    </div>

    <!-- here follow items -->

  </div> <!-- panel -->
</div> <!-- bar -->
```

The button `btnClosePane` can be used to close the search panel, and the `edtSearchInResults` edit can be used to filter the shown results even further.

Each item will be displayed using a HTML snipped like this

```
<a class="panel-block result-item">
  <span class="panel-icon"
    <i class="las la-book" aria-hidden="true"></i>
  </span>
  <div class="subtitles">
    <p class="has-text-weight-semibold">Page X:</p>
    <p>
      <span>some leading text </span>
      <span class="has-text-weight-bold">matched text in bold</span>
      <span>some trailing text </span>
    </p>
  </div>
</a>
```

the following 3 lines shown earlier take care of displaying the search results in the search pane:

```
ClearResultPane;
For I:=1 to Length(Results.Matches)-1 do
  ShowResultMatch(I,Results.Matches[i]);
ShowResultPane;
```

The `ClearResultPane` and `ShowResultPane` are quite simple:

```
procedure TMyApplication.ShowResultPane;

begin
  pnlSidebar.classList.Remove('sidebar_closed');
  edtSearchInResults.Value:='';
end;

procedure TMyApplication.ClearResultPane;
```

```
Var
  Rem,Child : TJSHTMLElement;

begin
  Child:=TJSHTMLElement(pnlResults.firstElementChild);
  While Assigned(Child) do
    begin
    Rem:=TJSHTMLElement(Child);
    Child:=TJSHTMLElement(Rem.nextElementSibling);
    if Rem.classlist.contains('result-item')  then
      pnlResults.removeChild(Rem);
    end;
end;
```

The loop simply removes all elements that have `result-item` in their class list.

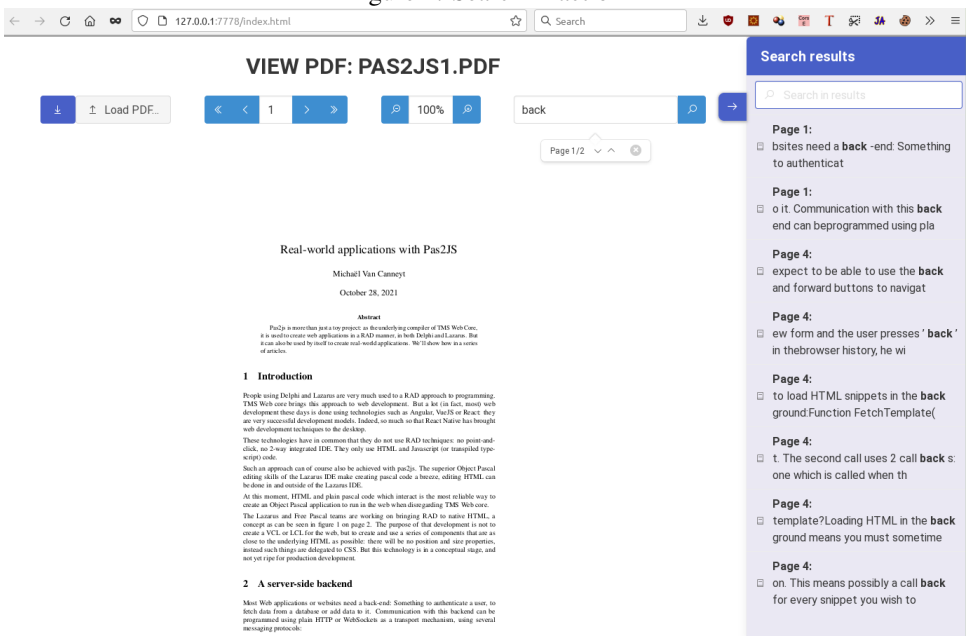
The `ShowResultMatch` does the actual work of constructing the HTML with the results. It does this by creating an anchor element, and filling the anchor element with a HTML snippet. The `ResultContent` constant contains the HTML snippet shown above (minus the anchor element itself), with some formatting placefolders for the call to `Format`:

```
Function TMyApplication.ShowResultMatch(Idx : Integer;
   aResult : TPDFSearchMatch) : TJSHTMLElement;

begin
  Result:=TJSHTMLElement(Document.CreateElement('a'));
  Result.Dataset['page']:=IntToStr(aResult.Page);
  Result.Dataset['idx']:=IntToStr(Idx);
  Result.ClassName:='panel-block result-item';
  Result.InnerHTML:=Format(ResultContent,[aResult.Page,
              aResult.Context1,aResult.Text,aResult.Context2]);
  Result.AddEventListener('click',@DoShowMatch);
  pnlResults.appendChild(Result);
end;
```

As you can see in the above code, the index of the match, and the page on which the match occurs are saved in the `Dataset` property of the result element; The resulting anchorelement also gets a `onclick` handler which will use the 'page' variable to determine the page to jump to:

```
Procedure TMyApplication.DoShowMatch(aEvent : TJSEvent);

var
  aPage : Integer;
  aAnchor : TJSHTMLElement;

begin
  pnlResults.querySelectorAll('result-item').foreach(@MakeInactive);
  aEvent.currentTargetElement.classList.add('is-active');
  aAnchor:=TJSHTMLElement(aEvent.currentTargetElement);
  aPage:=StrToIntDef(String(aAnchor.Dataset['page']),-1);
  if aPage<>-1 then
    queueRenderPage(aPage);
end;
```

Figure 4: Search in action



Note that the clicked result is shown visually as active, while the other results are made visually inactive first:

```
procedure MakeInactive(currentValue : TJSNode; currentIndex: NativeInt; list : TJ

begin
  TJSHTMLElement(CurrentValue).classList.remove('is-active');
end;
```

The result of all this can be seen in figure 4 on page 20.

As mentioned earlier, to close the search result panel, the user can click the button `btnClosePane`:

```
procedure TMyApplication.onClosePane(aEvent : TJSEvent);

begin
  HideResultPane;
end;

procedure TMyApplication.HideResultPane;
begin
  pnlSidebar.classList.Add('sidebar_closed');
end;
```

The search result list can be further filtered by typing in the filter box at the top of the result pane. The results are filtered by simply hiding the results that do not contain the filter term: this can be done by adding or removing the `is-hidden` Bulma CSS class from the anchor element:

```
Function TMyApplication.DoSearchResultsKeyUp(aEvent: TJSKeyBoardEvent): boolean;

Var
```

Figure 5: Filtered search results


```
  I,J,Len : Integer;
  List1,List2 : TJSNodeList;
  Pnl,Span : TJSHTMLElement;
  Vis : Boolean;
  aTerm : String;

begin
  Result:=True;
  aTerm:=UpperCase(edtSearchInResults.value);
  List1:=pnlResults.querySelectorAll('.result-item');
  for I:=0 to List1.Length-1 do
    begin
    Pnl:=TJSHTMLElement(List1.item(I));
    List2:=pnl.querySelectorAll('div p span');
    Vis:=(aTerm='');
    Len:=List2.Length;
    J:=0;
    While (Not Vis) and (J<Len) do
      begin
      Span:=TJSHTMLElement(List2.item(J));
      if (ATerm='') or (Pos(aTerm,UpperCase(Span.InnerText))>0) then
        Vis:=True;
      Inc(J);
       end;
    if vis then
      pnl.ClassList.Remove('is-hidden')
    else
      pnl.ClassList.add('is-hidden');
    end;
end;
```

To determine whether an element matches the filter term, all tags containing text ('div' 'p' and 'span') are examined: if the `innertext` of one of them contains the filter text, the anchor is set to visible - or invisible otherwise. The result can be seen in figure 5 on page 21.

Note that the double search/filter mechanism is probably not something one would actually

implement like this, but the above is meant to show how one could go about it in a real application.

The above code is just the display of the results. The actual searching happens in the `TPDFSearch` class:

```
TPDFSearchPageResultsEvent = Reference to Procedure(aPage : Integer; Matches : TP

TPDFSearch = Class(TComponent)
Public
  function searchPage(aPageNo : Integer; aSearchTerm : String) : TPDFPromise;
  function searchDocument(aSearchTerm : String) : TJSPromise;
  Property PDF : TPDFDocumentProxy Read FPDF Write FPDF;
  Property ContextChars : Integer;
  Property OnPageResult : TPDFSearchPageResultsEvent;
end;
```

The `PDF` property is of course the document to search. The `ContextChars` is the number of characters to use in the search context. The `OnPageResult` event will be called when all results for a single page are in: the PDF.js implementation delivers the text per page, so the search mechanism searches one page at a time, and using this event, you can notify the user of the progress of the search algorithm.

One reason for putting this in a separate class (except reusability) is that the PDF.js implementation uses promises when returning the text of a page in the PDF, and some state must be kept.

The `SearchDocument` call is the call that starts the actual search. It actually starts a search on every page using the `SearchPage` method. This method returns a promise. All promises are collected in an array, and the `TJSPromise.All` class method is then to wait till all results of all promises are in: this promise is the result of the `SearchDocument` call.

The `TJSPromise.All` class method in itself returns a promise that resolves to an array with all the results of all individual promises, in the same order as the original array of promises. When the results are in, the `GotAllResults` method concatenates all the results of all promises, and uses this to resolve the `All` promise.

```
function TPDFSearch.searchDocument(aSearchTerm: String): TJSPromise;

  Function GotAllResults(aValue : JSValue) : JSValue;

  Var
    AllRes : TPDFSearchResultArray absolute aValue;
    aRes : TPDFSearchResult;
    DocumentResults : TPDFSearchResult;

  begin
    DocumentResults:=Default(TPDFSearchResult);
    for aRes in AllRes do
      DocumentResults.Matches:=Concat(DocumentResults.Matches,aRes.Matches);
    Result:=DocumentResults;
  end;

var
  results : array of TPDFPromise;
  I : Integer;
```

```
begin
  Results:=[];
  For I:=1 to PDF.numPages do
    Results:=Concat(Results,[SearchPage(I, aSearchterm)]);
  Result:=TJSPromise.all(TJSPromiseArray(Results))._then(@GotAllResults);
end;
```

The `SearchPage` method is where the actual search for the search term in a page happens. Because PDF.js returns the page (using `getPage`) in a promise, and you must get the page text with `getTextContent` which again returns a promise, 2 callbacks are needed:

```
function TPDFSearch.searchPage(aPageNo: Integer;
                               aSearchTerm: String): TPDFPromise;

  Function GotPage(aValue: JSValue) : JSValue;

  Var
    aPage : TPDFPageProxy absolute aValue;

  begin
    Result:=aPage.getTextContent();
  end;

begin
  Result:= PDF.getPage(aPageNo).&then(@GotPage).&then(@GotContent);
end;
```

The `GotContent` method does the actual work. It gets as incoming parameter the text content of the page, which is the result of the `getTextContent` call, a record of type `TTextContent`. This record contains an element `items`, which is an array of `TTextContentItem` records. Each `TTextContentItem` has a member `str` which is a chunk of text found in the PDF.

The routine starts by concatenating all text chunks using `TJSArray.map` and `join`, and uses a regular expression to search for the search term:

```
Function GotContent(aValue: JSValue) : JSValue;

  Function DoMap(Itm : JSValue; index: NativeInt; anArray : TJSArray) : JSValue;
  var
    aItem : TTextContentItem absolute itm;

  begin
    Result:=aItem.Str;
  end;

Var
  aContent : TTextContent absolute aValue;
  aTerm,aReg, aText : String;
  aRegEx : TJSRegexp;
  aRegexMatches : TStringDynArray;
  Res : TPDFSearchResult;
  aMatch : TPDFSearchMatch;
```

```
begin
  Res:=Default(TPDFSearchResult);
  aText:=TJSArray(TJSArray(aContent.items).map(@DoMap)).join('');
  aTerm:=aSearchTerm; // Todo: escape specials
  if FContextChars=0 then
    FContextChars:=30;
  aReg:=Format('(.{0,%d})%s(.{0,%d})',[FContextChars,aTerm,FContextChars]);
  aRegEx:=TJSRegExp.New(aReg,'gi');
  aRegexMatches:= aRegEx.exec(aText);
  While Length(aRegexMatches)>0 do
    begin
    aMatch:=Default(TPDFSearchMatch);
    aMatch.Page:=aPageNo;
    aMatch.Text:=aSearchTerm;
    if isString(aRegexMatches[1]) then
      aMatch.Context1:=aRegexMatches[1];
    if isString(aRegexMatches[2]) then
      aMatch.Context2:=aRegexMatches[2];
    Res.Matches:=Concat(Res.Matches,[aMatch]);
    aRegexMatches:=aRegEx.exec(aText);
    end;
  Result:=Res;
  If Assigned(FOnPageResult) then
    FOnPageResult(aPageNo,Res.Matches);
end;
```

In this code, the regular expression is constructed with the following line:

```
aReg:=Format('(.{0,%d})%s(.{0,%d})',[FContextChars,aTerm,FContextChars]);
```

If the search term is 'back' and the number of context characters is 20, this will result in a regular expression:

```
(.{0,20})back(.{0,20})
```

which is in essence saying search "all occurrences of 'back' with at most 20 characters before and after" The result of executing this regular expression is then an array of 3 elements:

1. Element 0: the full search text
2. Element 1: the content of the first bracket expression.
3. Element 2: the content of the second bracket expression.

A loop is used to execute the regular expression till no more results are found. In each step of the loop, the regular expression result is converted to a `TPDFSearchMatch` record, which is appended to the total result. If the `OnPageResult` event is assigned, it is called with the results for the page.

7 Conclusion

The PDF.js library is a powerful library which allows to show a PDF in the browser without the need for an external program. In this article we show how to use this to display a PDF

and search in it. With this, we have not yet exhausted the possibilities of the PDF.js: the mechanism for searching can also be used to allow highlighting search results and allowing the user to select, copy and paste the PDF text. These possibilities will be explored in a future contribution.