# FPReport - a new reporting engine

Michaël Van Canneyt

December 27, 2021

**Abstract**

In this article we discuss the new FPReport reporting engine, the design goals that were at the basis of the engine, what can be done with it, and we show how it can be used.

## 1 Introduction

Many applications need to print data from time to time. Delphi and Lazarus offer a printer canvas, on which you can paint the page to be printed, as if you would paint it on the screen. This works well for a small and quick print, but when printing needs become more complicated and elaborate, this approach is slow and cumbersome.

For this reason, reporting tools exist. There are external reporting tools such as Crystal Reports, JasperReport. Integrated solutions for Delphi are FastReport, Quickreport, Rave reports and many others. Lazarus ships with lazReport (based on a free version of Fastreport 2), and FastReport has a version that compiles for lazarus. They are banded reporting tools: the output is divided in 'bands' which are repeated once or more on a page. This can be a list of students attending a class, or an invoice for a customer purchase, or an quartely overview of incoming funds...

All these integrated engines share a common design fault: by design they require a GUI subsystem on the system where the report is generated. Today, when more and more development is shifted to the web, this becomes an increasingly difficult restriction: many webservers are simple Linux containers without an X-Windows system installed.

While this requirement of having a GUI system present seems logical, it is not: strictly speaking, a generating a report is just layouting text and pictures on a page, determined by the data that drives the report. This is just a matter of calculating the sizes and positions of a series of rectangles on a virtual 'page'. The GUI system can come in handy to design the report layout, and it is necessary to view the resulting output. But the actual layouting does not need a GUI.

A typical scenario is a webshop: the developers designs the invoice to be sent to the customer as a report, and integrates it in the web application: on the server the report design is stored (it can be crated in code in the binary, or as a file on disk).

When the customer has finalized his purchase, a PDF is generated on the server, and sent to the client by mail. Or the client can opt to show the invoice in the browser, in which case the report can be rendered to HTML directly.

To generate this PDF, or the HTML based on the report design created by the programmer, no GUI system is needed. HTML and PDF are just text files with layouting instructions - one for a browser, the other for a PDF reader.

In this scenario, PDF and HTML are possible outputs of the reporting engine.

The manager in the office who wishes to see and print an overview the monthly purchases, may well be using a desktop program to access the web shop data. He can ask for a printed version of the monthly report. Here, no PDF is needed, the report can be sent directly to the printer or viewed on screen: again 2 forms of output for the reporting engine.

Considering all these use cases we arrive at a set of requirements:

- The core layouting engine may not rely on a GUI system to do its work. It results in a description of a set of output pages.
- Various output formats (called renderers) must exist: PDF, HTML, Image. These renderers again may not rely on a GUI system
- Screen and printer are also output formats.
- The report designer to create the reports may depend on a GUI system.
- Multi-column layout must be possible.

These requirements are the basis for FPReport, with the `TFPReport` class as the main class.

This set of requirements has an interesting consequence: To calculate the layout of a text, the reporting engine needs to know the extent of a text in the chosen font - a service that is commonly provided by the GUI subsystem, but which by our requirements, is unavailable. Luckily, the freetype library is a free library that can also provide this service for TrueType fonts.

All classes in FPReport start with `TFPReport`, and each class `TFPReportNNN` is a simple descendant of a `TFPReportCustomNNN` class, the former publishes the protected/published properties of the latter.

# 2 Printable elements

What should a reporting engine be able to print ? There are some obvious candidates, we call them report elements:

- Text. Preferably with some limited formatting inside the text: bold, colors, and for PDF or HTML output: hyperlinks. The text should be customizable: this means that we must be able to get it from a data source, and we should have some formatting options available. (the class `TFPReportMemo`)
- Images. This can be a company logo, but can also be an image of an item you purchased, or the picture of a student in a list of students.. (the class `TFPReportImage`). It can load any FPC supported image type.
- Checkboxes: these are just a special case of an image: an image to represent 'true', and an image to represent false. (`TFPReportCheckbox`)
- Shapes: squares, circles, triangles or simple lines. (`TFPReportShape`)

But preferably the list of 'printable' things should be larger:

- Barcodes (available in `TFPReportBarcode`).
- QR codes (available in `TFPReportQRCode`).
- Graphs. (not yet available, but planned)

- Pivot tables. (not yet available, but planned)
- ...

The system must be extensible: it must be possible to register additional printing elements, and a renderer for this element must exist. The most common renderer simply draws whatever is needed on a bitmap, and then the report renderer draws the bitmap on screen, in HTML or whatever output is desired. it is possible to create and register renderers for a specific format (for improved quality of output), but this is entirely optional.

FPReport comes with barcode and QRCode renderers. It allows to use simple HTML tags inside text elements (bold, italic, anchor, font), and allows you to embed formulas in the text.

# 3 Data and calculations

Formula in the text will be replaced by their calculated result in the output. The reporting engine uses the Free Pascal expression parser engine to provide formula support. This engine allows the use of variables (identifiers), meaning that you can do something like

```
The amount to be paid is [formatfloat('##0.##',total)] EUR.
```

The text between square brackets is a formula, which will format the variable 'total' using the formatfloat function. All fields in the data of the report is available as variables in the formula. It is also possible to add named report variables to a report: the value of these variables will be made available in the formula by their names. The engine also supports aggregate data such as Min(SomeVariable)

```
The total amount is [formatfloat('##0.##',sum(itemprice))] EUR.
```

If `itemprice` represents the price of an item in the invoice, the engine will update the formula with each iteration over the items in the invoice.

A report is driven by data. Traditionally this data comes from a database, and is fetched through a dataset: The report loops over the records in the dataset, and prints a detail band for each record in the dataset.

The idea of looping over data can be generalized, and FPReport supports several 'data loops' (all descendents of `TFPReportDataLoop`) out of the box:

- A dataset-backed loop. (`TFPReportDatasetData`). Just hook up the dataset to the report. This can be done visually in an IDE, there is no need to create code.
- A JSON-array backed loop. (`TFPReportJSONData`) The array contains objects, and each property of the object is available as a field.
- A Collection backed loop (`TFPReportCollectionData`). The published properties of the collection items are the data available in the report.
- A list backed loop. The published properties of the objects in the list are the data available in the report. (`TFPReportListData`).
- A user event driven loop (`TFPReportUserData`): if none of the above suits your needs, a simple solution is to use the event driven data loop: here the names and values of variables are fetched through events, and when the loop needs to go to the next iteration of the loop, it calls an event as well.

These loops are implemented in a separate units, so only code that you actually use is included in your application. This means is possible to create reports without including any database code in your application.

# 4 Creating a report in code

To get a feel for what is involved in designing and rendering a report, we'll create a report in code. It's a simple report, it just prints the contents of a text file, nicely formatted. It adds a page header with date and filename, and a page footer with the page number.

Instead of loading the stringlist contents from file, this could be the contents of a memo: the code can be used to print the contents of a memo instead of a file.

The program is extremely simple, the main code is in the `DoRun` method.

```
procedure TPrintApplication.DoRun;

Var
  PG : TFPReportPage;
  PH : TFPReportPageHeaderBand;
  PF : TFPReportPageFooterBand;
  DB : TFPReportDataBand;
  M : TFPReportMemo;
  PDF : TFPReportExportPDF;
  Fnt : String;

begin
  Fnt:='DejaVuSans';
  FLines.LoadFromFile(ParamStr(1));
  gTTFontCache.ReadStandardFonts;
  gTTFontCache.BuildFontCache;
  PaperManager.RegisterStandardSizes;
```

The first two lines speak for themselves. The `ReadStandardFonts` and `BuildFontCache` lines tell the font engine to load standard fonts from standard locations. This is a catch-all method, which registers all available fonts. More fine-grained control is possible. The important thing is that the engine loads in memory the needed font information before the reporting engine starts layouting the report.

After that the `RegisterStandardSizes` call is used to register a set of commonly-used page sizes. Again, this is necessary once, to be able to set the paper size of a report page. The next step is adding a design page to the report:

```
  // Page
  PG:=TFPReportPage.Create(FReport);
  PG.Data:=FData;
  PG.Orientation := poPortrait;
  PG.PageSize.PaperName := 'A4';
  PG.Margins.Left := 15;
  PG.Margins.Top := 15;
  PG.Margins.Right := 15;
  PG.Margins.Bottom := 15;
```

If no paper size is set, then unexpected things can and will happen. Setting the margins is natural, the whole page cannot be filled by a printer. The used units are millimeters. Note

that the page owner is the report. This is not a requirement, but doing so adds the page to the report: a report can have multiple designer pages, which will be rendered one after the other.

Note that the page data is set to `FData` - this is an event data loop, which will be set up later. The reporting engine needs to know for each design page which data loop must be run.

Once the page is set up, we set up a page header with 2 memos: one to contain the filename of the printed file, the other contains the date:

```
// Page header
PH:=TFPReportPageHeaderBand.Create(PG);
PH.Layout.Height:=10; // 1 cm.
// Filename
M:=TFPReportMemo.Create(PH);
M.Layout.Top:=1;
M.Layout.Left:=1;
M.Layout.Width:=120;
M.Layout.Height:=7;
M.Text:=ParamStr(1);
M.Font.Name:=Fnt;
M.Font.Size:=10;
// date
M:=TFPReportMemo.Create(PH);
M.Layout.Top:=1;
M.Layout.Left:=PG.Layout.Width-41;
M.Layout.Width:=40;
M.Layout.Height:=7;
M.Text:='[Date]';
M.Font.Name:=Fnt;
M.Font.Size:=10;
```

The filename is just entered as the contents of the report memo. The date is entered using a formula: the `Date` function is available in formulas used in the report, and will be formatted using standard date notation (obviously, there are functions to change the formatting).

Similarly, we can set up the page footer:

```
// Page footer
PF:=TFPReportPageFooterBand.Create(PG);
PF.Layout.Height:=10; // 1 cm.
M:=TFPReportMemo.Create(PF);
M.Layout.Top:=1;
M.Layout.Left:=1;
M.Layout.Width:=40;
M.Layout.Height:=7;
M.Text:='Page [PageNo]';
M.Font.Name:=Fnt;
M.Font.Size:=10;
```

The `PageNo` variable contains the current page. The `PageCount` variable is also available, and contains the total number of rendered pages. The page count can be substituted at the end of the rendering, or the report can be rendered twice (this happens automatically) and will be set to the number of pages that were rendered at the end of the first run.

All that must be done is create a band in which the contents of the string list will be displayed. The data loop will return 1 line of the string list on each iteration. That means that the data band will be printed once for each line in the string list. So we set up a data band with 1 memo that stretches over the width of the band:

```
  // Actual line
  DB:=TFPReportDataBand.Create(PG);
  DB.Data:=FData;
  DB.Layout.Height:=5; // 0.5 cm.
  DB.StretchMode:=smActualHeight;
  M:=TFPReportMemo.Create(DB);
  M.Layout.Top:=1;
  M.Layout.Left:=1;
  M.Layout.Width:=PG.Layout.Width-41;
  M.Layout.Height:=4;
  M.Text:='[Line]';
  M.StretchMode:=smActualHeight;
  M.Font.Name:=Fnt;
  M.Font.Size:=10;
```

The memo contains a formula with a simple variable name: `Line`. This is the name that our data loop will need to report. The memo by default will perform wordwrap, and we set the stretch mode of the memo and the band to `smActualHeight`. This means that the memo will increase its height to fit the length of the text, so long lines are accomodated. In turn, the band will increase its height as the memo grows in height. By default, the heights of bands and printable elements is fixed (`smDontStretch`).

Then we set up our data loop, which is event based:

```
  // Set up data
  FData.OnGetNames:=@DoGetNames;
  FData.OnNext:=@DoGetNext;
  FData.OnGetValue:=@DoGetValue;
  FData.OnGetEOF:=@DoGetEOF;
  FData.OnFirst:=@DoFirst;
```

The events are extremely simple. The first one reports the 'variables' that are managed by the loop, as we've seen the memo expects a 'Line' variable:

```
procedure TPrintApplication.DoGetNames(Sender: TObject; List: TStrings);
begin
  List.Add('Line');
end;
```

As the report loops over the data loop, the next line in the stringlist must be returned. In essence this loop is coded as

```
  Data.First;
  While not Data.EOF do
    begin
    // Get data and Print bands
    Data.Next;
    end;
```

So we need a current line index, we'll keep it in a `FLineIndex` variable which is initialized, updated and checked in the following routines:

```
procedure TPrintApplication.DoFirst(Sender: TObject);
begin
FLineIndex:=0;
  end;

procedure TPrintApplication.DoGetNext(Sender: TObject);
begin
  Inc(FLineIndex);
end;

procedure TPrintApplication.DoGetEOF(Sender: TObject; var IsEOF: boolean);
begin
  isEOF:=FLineIndex>=FLines.Count;
end;
```

Finally, when converting the memo formula to text, the `OnGetValue` event is called, and it needs to return the correct variable. There is only one variable, so this is easy:

```
procedure TPrintApplication.DoGetValue(Sender: TObject;
                                       const AValueName: string;
                                       var AValue: variant);
begin
  Avalue:=FLines[FLineIndex];
end;
```

In an actual report with more variables, a check on `AValueName` would have to be performed, and the correct value corresponding to the name would have to be returned.

Now everything is set up and the report can be rendered:

```
  // Go !
  FReport.RunReport;
  PDF:=TFPReportExportPDF.Create(Self);
  try
    PDF.FileName:=ChangeFileExt(Paramstr(1),'.pdf');
    FReport.RenderReport(PDF);
  finally
    PDF.Free;
  end;
```

The `RunReport` method does the actual layouting of the report.

The result of this layouting is a structure in memory which can then be rendered. The rendering happens using the appropriate rendering class, and in the example above, we render the report to a PDF File using the `TFPReportExportPDF` class.

The result of this is a PDF file, which can look as in figure 1 on page 8.

Figure 1: The program run on its own source code