

# FPReport - continued

Michaël Van Canneyt

December 27, 2021

## Abstract

In this second article we delve deeper in the possibilities of FPReport: we show how to save and load a design, make groups in our report, and how to display totals in footers or headers of these groups. We end with the visual report designer

## 1 Introduction

In a previous article, the design of the FPReport reporting engine was described. A simple report was created, and the use of an exporter to create for example a PDF file of a text file was demonstrated.

In this article, we'll demonstrate how to save and load a report design from file, and how this can be combined with the visual designer that comes with FPReport.

The previous article showed that a simple list - even a multi-column one - is easy to do. Grouping is not more difficult, and in this article, we'll show how this can be accomplished. We'll also show how to work with variables to define and print totals.

Although building of reports in code is enlightening and provides understanding of how the engine works, it can be quite tedious. So, a visual report designer is an absolute must, and fpreport comes with one. The highlights of the designer will also be shown - it is in fact pretty standard for a designer.

## 2 Saving and loading a report from stream

Creating a report design in code is cumbersome. A report design can be created visually, and the design can be saved to file. FPReport relies on a helper class to save/load a report from file: A descendent of `TFPReportStreamer`. This is done so various formats can be supported. The default format is JSON, but any format (XML, YAML) could be implemented.

The following will save the report to file, once it was designed:

```
procedure TPrintApplication.SaveReportDesign;
```

```
Var  
  J : TFPReportJSONStreamer;  
  F : TFileStream;  
  S : TJSONStringType;
```

```
begin  
  F:=Nil;
```

```

J:=TFPReportJSONStreamer.Create(Self);
try
  FReport.WriteElement(J);
  F:=TFileStream.Create('txt2pdf.fpr',fmCreate);
  S:=J.JSON.FormatJSON();
  F.WriteBuffer(S[1],Length(S));
finally
  F.Free;
  J.Free;
end;
end;

```

This is, in fact, quite simple; a streamer instance is created, the report is saved to the streamer, and the resulting JSON is written to a file. Nothing could be simpler.

The following code does the reverse operation: it loads the report from a file:

```

procedure TPrintApplication.LoadReportDesign;

Var
  J : TFPReportJSONStreamer;
  F : TFileStream;
  O : TJSONObject;

begin
  J:=Nil;
  F:=TFileStream.Create('txt2pdf.fpr',fmOpenRead);
  try
    O:=GetJSON(F) as TJSONObject;
    J:=TFPReportJSONStreamer.Create(Self);
    J.JSON:=O;
    J.OwnsJSON:=True;
    FReport.ReadElement(J);
  finally
    F.Free;
    J.Free;
  end;
end;

```

Again, nothing spectacular. Note the `OwnsJSON:=True`, this tells the streamer it should free the JSON object when it is freed itself.

Note that for this to work, some small changes are needed to the sample program of the previous article. The first change is that the data loop object must have a name, and must be registered with the report. This is done in the constructor of our application object:

```

constructor TPrintApplication.Create(AOwner: TComponent);

begin
  Inherited;
  FReport:=TFPReport.Create(Self);
  FLines:=TStringList.Create;
  FData:=TFPReportUserData.Create(Self);
  FData.Name:='Data';
  FReport.ReportData.AddReportData(FData);

```

The reason for this is that when writing the report design, the report elements will write the name of the data loop to the stream. When reading the report design from stream, they will use this name to look up the data loop in the list of registered report data loops: for this reason, the data loop needs a name and must be registered in the reportdata collection.

Since the name of the data loop is set, variable name in the memo that prints the actual line, must be changed:

```
M:=TFPReportMemo.Create(DB);
M.Text:=' [Data.Line]';
```

The name consists of 2 parts, separated by a dot: the data loop name, and the variable name. A report can have multiple data loops, so the name of the data loop becomes important. A version of the example program that uses a design in a separately stored file is also available.

The `fpjsonreport` unit contains a `TFPJSONReport` descendent which does all the above:

```
TFPJSONReport = class(TFPReport)
  procedure LoadFromStream(const aStream: TStream);
  procedure SaveToStream(const aStream: TStream);
  Procedure LoadFromJSON(aJSON : TJSONObject); virtual;
  Procedure SavetoJSON(aJSON : TJSONObject); virtual;
  Procedure LoadFromFile(const aFileName : String);
  Procedure SaveToFile(const aFileName : String);
end;
```

Why this elaborate design, why not simply have 2 methods in `TFPReport`: `LoadFromFile` and `SaveToFile`? Several reasons, in fact:

- One is an architectural pattern, separation of concerns: the report class should not be concerned with the file format. This allows to save to any desired format. JSON is the default choice but an (untested) XML streamer also exists. Each can use the format he/she chooses.
- The second is that the report stream only contains the design of the report. A standalone reporting tool that fetches data needs to add information to the stream about the data for the data loop: server connection data, the SQL to execute, parameters to ask etc. The standalone FPRReport designer tool uses this mechanism. By separating the streaming from the core reporting tool, it becomes possible to hook into the streaming mechanism and save whatever additional data is needed.

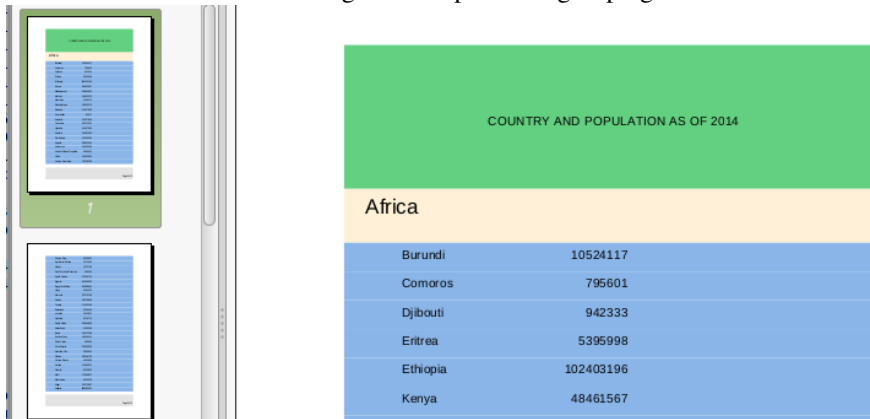
### 3 Grouping

Many reports will do some kind of grouping. For sales results or forecasts, the monthly, quarterly or even yearly totals are a must. For an itemized invoice, the items may be grouped in categories, or according to VAT tariff. A list of countries may be broken down in continents or even parts of a continent, or simply by grouping them according to the first letter of their name.

To create a group in a report requires 3 steps:

1. Determine a grouping condition. This is an expression that, when it changes, signals the start of a new group. In the example of the list of countries, this expression can be simply the name a field in the data that contains the continent name, or an expression:

Figure 1: Report with grouping



COUNTRY AND POPULATION AS OF 2014	
Africa	
Burundi	10524117
Comoros	795601
Djibouti	942333
Eritrea	5395998
Ethiopia	102403196
Kenya	48461567

`Copy (Country, 1, 1)`

This returns the first letter of the name of the country.

2. Sort the data so the items in the data list are grouped together. fpReport will not do this for you, you must take care of this yourself. In the example of the list of countries, the list of countries must be sorted first on the continent, or alphabetically - depending on what grouping mechanism you want to use.
3. A group header (a band of class `TFPReportGroupHeaderBand`) must be inserted in the report, and its `GroupCondition` must be set to the expression that determines the group. This must be done, even if you only want to display group totals at the end of a group. (you can set the header's `visible` property to `False`, or set the height to zero).

So, in the case of a list of countries, the following code will add a group header, which displays the continent of the countries that follow:

```
GroupHeader := TFPReportGroupHeaderBand.Create(p);
GroupHeader.Layout.Height := 20;
GroupHeader.GroupCondition := 'Continent';
Memo := TFPReportMemo.Create(GroupHeader);
Memo.Layout.Top := 5;
Memo.Layout.Width := 10;
Memo.Layout.Height := 8;
Memo.UseParentFont := False;
Memo.Text := '[Continent]';
```

Figure figure 1 on page 4 shows a screenshot of a report with grouping enabled.

## 4 Displaying totals and other aggregates

Grouping the items makes a list not only more easily to read, it can also be used to display additional information: a group total, for example, the total population of the countries per continent.

There are 2 ways to display such a total: A simple one, which is only usable in the group footer, the other - involving an extra variable - can be used both in the group header and the group footer.

We'll start with the simple one, which can only be used on a group footer. In the below code we'll display the total population of the continent at the end of the continent. This is done in a group footer (a band of type `TFPReportGroupFooterBand`).

```
// Create the group header
GroupHeader := TFPReportGroupHeaderBand.Create(p);
GroupHeader.Layout.Height := 20;
GroupHeader.GroupCondition := 'Continent';
//Create the group footer
GroupFooter := TFPReportGroupFooterBand.Create(p);
GroupFooter.Layout.Height := 20;
// Attach to the correct group
GroupFooter.GroupHeader:=GroupHeader;
Memo := TFPReportMemo.Create(GroupHeader);
Memo.Layout.Left := 15;
Memo.Layout.Top := 5;
Memo.Layout.Width := 10;
Memo.Layout.Height := 8;
Memo.UseParentFont := False;
// Display the total.
Memo.Text := 'Total for [data.continent]: '+
            '[FormatFloat(''#0.00'', '+
            ' sum(data.population/1000000))] million.';
```

The 2 things to take note of in the above code is first of all the fact that the group footer must be attached to the group header with its `GroupHeader` property. This is necessary to establish the logical structure of the report: if multiple groups and group footers and group headers are present, the reporting engine has no way of knowing what footer belongs to what header - bands have no 'position' in the report which could aid in determining the logical structure. The second thing to note is the use of the `Sum` aggregate function in an expression. The reporting engine expression parser knows that this function is an aggregate expression: whenever the loop changes record, it will update all aggregate functions. There are multiple aggregate functions:

**sum** calculates a simple sum of its argument.

**avg** calculates a simple average of its argument.

**max** Displays the maximum value of its argument.

**min** Displays the minimum value of its argument.

**count** keeps a simple count of the number of times it was updated and returns that count.

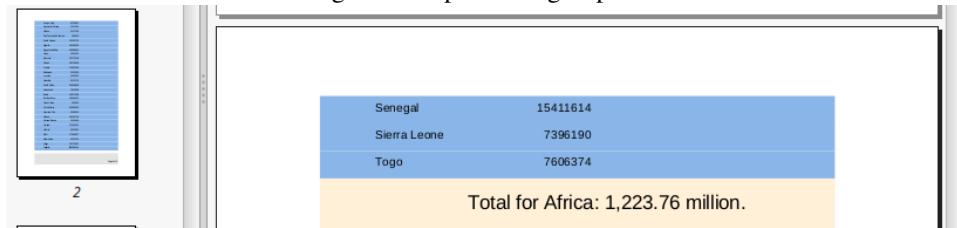
Whenever a memo containing an aggregate value was printed, the aggregate value is reset. This means that when the memo is displayed in a group footer, the values of the expression will take into account only the records of the group.

Should you wish to display e.g. a running total, then the reset of the aggregate values can be disabled through the `moNoResetAggregateOnPrint` option of the memo.

Figure figure 2 on page 6 shows a screenshot of a report with the total of a continent.

The above is quite simple to do, but also limited in possibilities. It does not allow us to put a total of the group header, since the total isn't known yet when the group is started. Also, when reprinting a group header on a page start, or displaying already a footer at the end of a page, this method will give wrong results. Some memos will need to display expressions that are reset on print with expressions that are not reset on print.

Figure 2: Report with group total



Senegal	15411614
Sierra Leone	7396190
Togo	7606374
Total for Africa: 1,223.76 million.	

Therefore, an alternative way to calculate and print group totals is introduced, and this is through report variables. The value of Report variables can be used in expressions by simply referencing the name of the value (you must take care not to use a name of a value in a dataloop, though).

Report variables exist in 2 kinds:

- Static variables. Their value is set once, and remains the same until a different value is set.
- Expression variables. Their value is calculated through an expression and is continuously updated as the reporting engine goes through the data loops.

The expression for this kind of variable can contain an aggregate function, but, more to the point: it can contain a reset expression.

The reset expression determines when an aggregate function in the variable expression is reset. There are several possibilities:

- At the start of a group.
- At the start of a new column.
- At the start of a new page.

To use an expression variable to display the total footer in our previous example, the following can be done.

First, the report variable must be defined:

```
Var
  V : TFPReportVariable;
begin
  V:=Report.Variables.Add('PopSum');
  V.Expression:='Sum(Population)';
  V.ResetValueExpression:='continent';
end;
```

After this, the expression for a memo displaying the total simply becomes:

```
// Display the total.
Memo.Text := '[PopSum]';
```

Lastly, the `TwoPass` option of the report must be set to `True`: the report will calculate the values for all groups in the first run of the report, and these values are then used in the second run of the report.

The amount of work to display a total in this manner is not so much harder than in the 'simple' way.

To display a total on a footer band at the end of a page, the reset expression simply becomes:

```
V.ResetValueExpression:=' PageNo' ;
```

Needless to say, the same variable cannot be used to display the page total and a group total.

To make life easier, there are even some auxiliary functions to make it easier to create these variables; they compute the reset expression for you.

```
Function AddExprVariable(aName : String; aExpr: String;
                        aType: TResultType = rtString;
                        aResetType: TFPReportResetType = rtNone;
                        aResetGroup: TFPReportCustomGroupHeaderBand = nil)
                        : TFPReportVariable;
Function AddExprVariable(aName : String;
                        aExpr: String;
                        aType: TResultType;
                        aResetType: TFPReportResetType;
                        aResetValueExpression: String) : TFPReportVariable;
```

The first form of this function can be used like this:

```
rpt.Variables.AddExprVariable('PopSum',
                              'sum(StrToFloat(population) / 1000000)',
                              rtFloat,
                              rtGroup,
                              GroupHeader);
```

This function will simply copy the value for the reset expression from the group band groupcondition, plus all parent groups. (in case of nested groups, all groups must be taken into account).

With expression variables, new things become possible, such as displaying a running total next to a group total. First, create the variables:

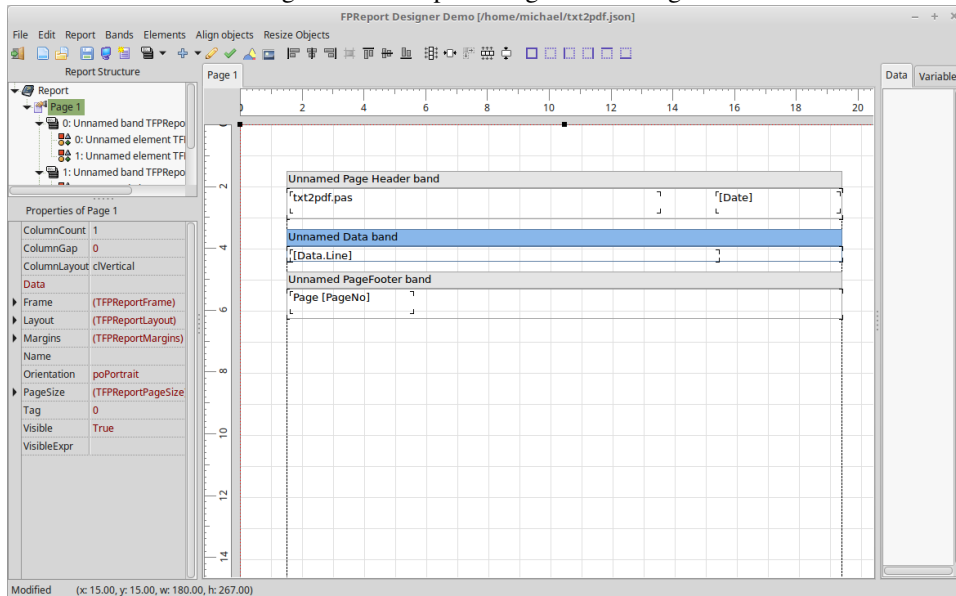
```
Var
  V : TFPReportVariable;
begin
  V:=Report.Variables.Add('PopSumGroup');
  V.Expression:=' Sum(Population)';
  V.ResetValueExpression:=' Continent';
  V:=Report.Variables.Add('PopSumGroupRunningTotal');
  V.Expression:=' Sum(Population)';
  // Reset expression empty !
end;
```

And then show them in a memo

```
// Display the total.
Memo.Text := '[PopSumGroup] (Running total: [PopSumGroupRunningTotal])';
```

The expression variables are a powerful tool, but care should be taken: do not attempt to use expression variables in an expression for another expression variable; The reporting engine will detect this and give an error.

Figure 3: The report design in the designer



## 5 The visual designer

Creating reports in code should not present a problem for even a junior programmer. But it will be tedious and hard work. For an end-user to make a report in this way is of course impossible - disregarding even the fact that the code needs to be compiled.

Luckily, a visual designer with a simple point-and-click interface is also available. It can be used in the Lazarus IDE, but can also be integrated in an end-user application to allow the user to design his own reports. There is also a stand-alone version of the designer.

For example, when loaded in the designer, the file printing report looks as in figure 3 on page 8. It's clear that for a novice, this is much easier to understand and manipulate than the same code needed to create the report.

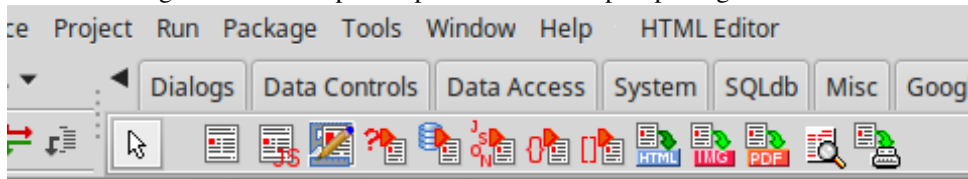
The reporting engine is configurable regarding the features it exposes to the user:

- Manage data.
- Manage variables.
- Manage bands.
- Manage pages.
- Load a report.
- Save a report .
- Add pages.
- Start a new report.
- Preview a report.

This means a very restricted version can be given to the user where he is able to set things like font color, position and size, but cannot do anything else: That means he can simply customize a pre-made design.



Figure 4: The component palette with FPReport package installed



But at the same time a version can be shipped where the user can do everything: in a large application with extended user management, the amount of options available could be based on the role of the user.

When called from the IDE, the designer does not allow to manage data, since the data must be provided by the programmer using various data loops. The report design can be stored in the Lazarus form file if the `TFPJSONReport` component is used. If it is used to edit a plain `TFPReport` component, the report has to be stored on disk, and loaded at runtime with some code.

The standalone designer has all options enabled (data choices are stored in the report file it generates).

The main form of the report designer as shown in figure 3 on page 8 is divided into 5 main areas:

1. The menu and toolbar at the top.
2. The object inspector and report structure on the right.
3. The pages of the report in the middle.
4. The data of the report to the right, it is possible to use drag&drop to drop a variable or field as a new memo to a report.
5. The status bar at the bottom, showing some extra information.

The toolbar can, in addition to the universal new, load and save buttons, be used to quickly set some properties or add some common elements to a report.

Less common elements can be added using the 'Element' menu or toolbar, where entries for all available elements are shown.

Adding a band is done using the band menu, or the band button; A menu item is available for all types of supported bands.

The 'Report' menu harbors the dialogs for adding a page, managing variables or data.

## 6 Using the designer in the IDE

To create a report design in the Lazarus IDE, the `lazidefpreport` package must be installed. the component palette will then be extended with a `FPReport` tab, as shown in figure 4 on page 9 The various components are, in the order that they appear on the component palette:

**TFPReport** a report component. The design must be loaded from and saved to file.

**TFPJSONReport** a report component. The design is stored in the (.lfm) form file.

**TFPJSONReport** a report component. The design is stored in the (.lfm) form file.

**TFPReportDesigner** The report designer component. This component is only needed if you wish to enable the end user to change the report design.

**TFPReportUserData** a report data loop component. The data is obtained through events.

**TFPReportDatasetData** a report data loop component. The data is obtained from a dataset.

**TFPReportJSONData** a report data loop component. The data is obtained from a JSON structure.

**TFPReportCollectionData** a report data loop component. The data is obtained from a `TCollection` instance. This collection is only available at runtime.

**TFPReportObjectListData** a report data loop component. The data is obtained from a `TObjectList` instance.

**TFPReportExportHTML** a report renderer that creates HTML pages.

**TFPReportExportfpImage** a report renderer that creates image files.

**TFPReportExportPDF** a report renderer that creates PDF files.

**TFPReportPreviewExport** a report renderer that previews the report on screen.

**TFPReportPrinterExport** a report renderer that sends the report to the printer.

How to use these components ? You need at least 3 components:

1. A report component (`TFPReport` or `TFPJSONReport`).
2. A data loop component.
3. A renderer (export) to generate output.

For the report component, it is important to decide in advance whether or not the end user should be able to modify (to a lesser or larger degree) the report design. The choice of report component depends on it:

When the report design is stored solely in the `.lfm` file, using a `TFPJSONReport` component, the user cannot modify it. When the report design is stored in a file (to be shipped with the application), a simple `TFPReport` component can be used. Both components can of course load a design from file, so when in doubt, use a `TFPJSONReport` component.

A data loop component is needed when designing the report in the IDE: the component editor will not allow you to fetch arbitrary data. The data component must be present on the form or data module that contains the report.

A report would not be useful without some output, so a renderer is needed.

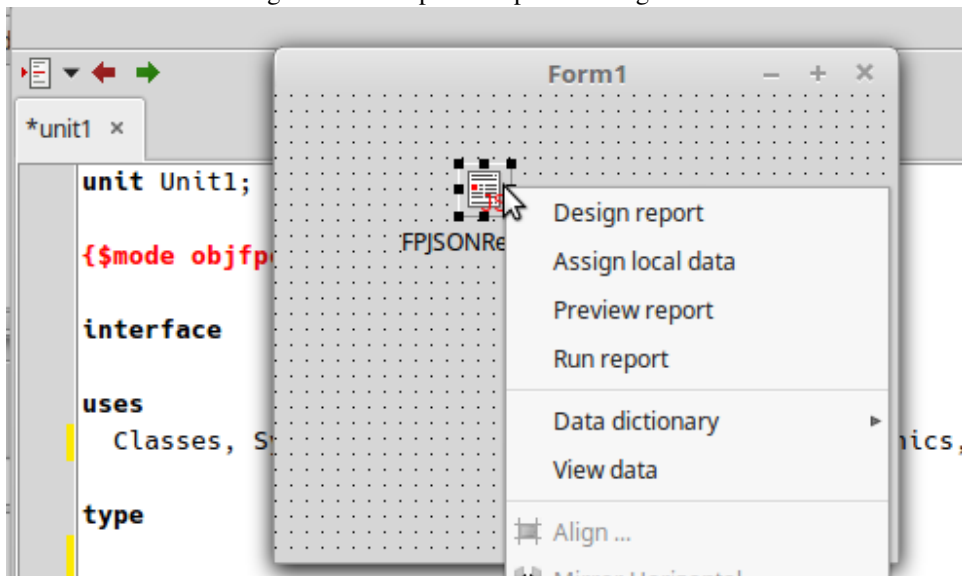
To design the report, right-click the report, a component design menu will pop up as show in figure 5 on page 11 The names of most of these menu items speak for themselves. The only one that may not be immediately obvious is the 'Assign local data' menu item:

When you have 3 report components and a bunch of data loops on a form or datamodule, you don't necessarily want all data loops to be available to all reports.

The available data loops for a report can be controlled in the Object Inspector: the `ReportData` property is a collection which enumerates the available data loops for a report instance.

In the simple case where all data loops on the form should be available to the report, the 'Assign local data' item can be used: Clicking this menu item will check the form for data loop components, and will make them available to the report.

Figure 5: The report component design menu



Once the report is designed, a small amount of code is needed to show the report. Assuming we've dropped a PDF report export component on the form and configured it properly, the following code will export the report:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    FPJSONReport1.RunReport;
    FPJSONReport1.RenderReport (FPReportExportPDF1);
end;
```

It's clear that writing only these 2 lines of code is a lot easier than designing and running the report completely in code.

## 7 Conclusion

In this article we've shown how to load and save a report design from and to file. We've also shown how grouping and aggregates work. Finally, we've shown that the coding involved in creating a report design can be dispensed with, the visual report designer makes this all a lot easier. In the next article, we'll show how to allow the end-user to design the report, and we'll have a look at some of the more exotic reporting elements in the report engine.