

Table of contents

Introduction, process overview	2
How to create a simple Report, testing your reports	3
How to configure the Engine	5
A Slightly Harder Report, With Repeating Rows	10
Repeating Cells, for variable Column number	13
Adding Images to your report	15
Adding Expressions	17
Control break reports.....	21

Introduction, process overview

ReportPlugin is symphony 1.4 plugin for generating reports with PHP, even though it provides a PHP API it integrates several technologies to achieve it (XSLT, XSL-FO, WordML, ApacheFOP which is written in java).

The reports are generated in three phases:

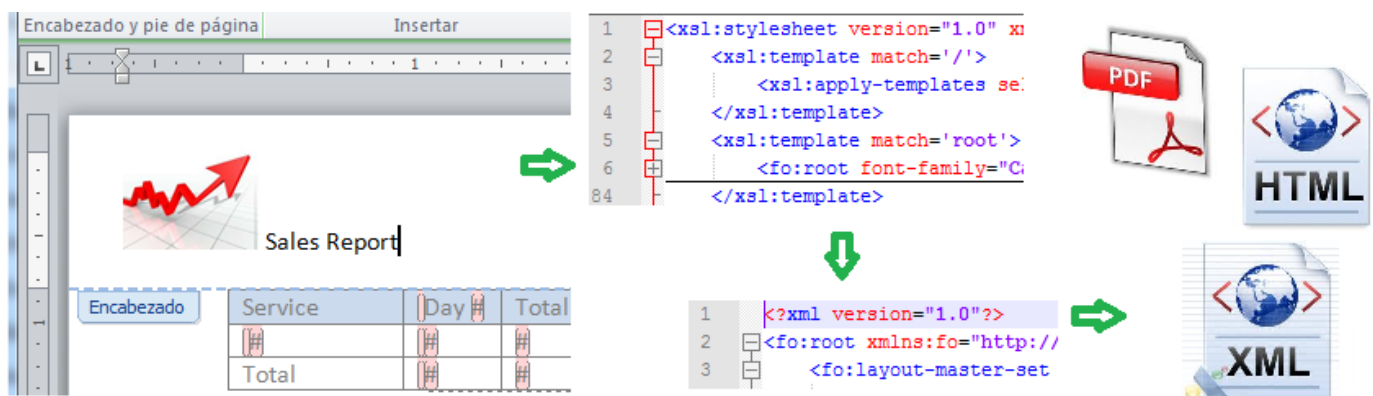
- Data Generation, create the data XML structure (logical screen)
- Design your template with MS Word & export as XSLT stylesheet
- Data and template merging, produces the XSL-FO representation of the report.
- Rendering. XSL-FO => {PDF, XHTML, etc...}

The Data generation consists of anything that produces a RAW XML with the report data. Of course, this XML structure must be consistent among several rendering of the same report, since you will have it in mind when creating the template, as the template must consume this data XML. The reportEngine provides a placeholder class for putting this kind of logic, but you can generate it in methods elsewhere, so you either provide the callback to generate it or the XML itself. To make things easier, a Doctrine XML Hydrator is bundled so you can get XML out of your Doctrine Queries, this is really handy if you use Doctrine, if not you have an arrayToXML method in utils (reportPlugin/lib/RenderStep1/Utils/Array2XML.php), that will also help if you represent your report data as an array().

The template must be an XSLT transformation stylesheet that will be processed by PHP XSLT Processor, don't get scared yet, the engine provides an automatic way of generating it from a MS WordML 2003 document (@see Word2XSLTRenderingFO.xml). This stylesheet consumes your data XML and Renders a XSL-FO document, that represents the rendered report, but in FO. By default the engine uses XSL-FO as an intermediate language for ultimately rendering any output as the result of this XSL-FO processing, but you could provide any XSLT, rendering any XML with your data, XHTML for instance, and use this, with no further transformation as the final report. Normally the XSL-FO report must still go through the Rendering phase, if you want something different from XSL-FO.

In the Rendering phase the engine transforms the XSL-FO representation into PDF, XHTML, or simply the same FO *that could also be any XML generated by your phase two XSLT*. More transformations can be easily added.

The look and feel of the generated PDFs are great, thanks to XSL-FO and ApacheFOP. They look almost identical to the Word template you make (Thanks to Word2FO). By the time you feel the need to tweak the automatically generated template, if it ever happens, you'll have already made some nice reports and will feel more at home with XSLT and XSL-FO, which by the way is infinitely easier to tweak than to write from scratch.



How to create a simple Report, testing your reports

1st of all, you need to know the structure of the data XML your report will consume. Take a look at the example in the image. The values that the report will try to consume are denoted by a hash character (#) commented with the node name, in the example [SubNode, day, month, year]

The screenshot shows an XML editor interface. On the left, an XML document is displayed with the following structure:

```
<root>
  <Node>
    <SubNode>Some Text</SubNode>
  </Node>
  <date>
    <day>15</day>
    <month>6</month>
    <year>2012</year>
  </date>
</root>
```

Below the XML, there is a table with three columns: Day, Month, and Year. Each cell in the first row contains a hash character (#) followed by a red dashed line pointing to a label on the right:

Day	Month	Year
#	#	#

On the right side, there are four red rounded rectangular labels with the following text:

- Comentario [j1]: SubNode
- Comentario [j2]: day
- Comentario [j3]: month
- Comentario [j4]: year

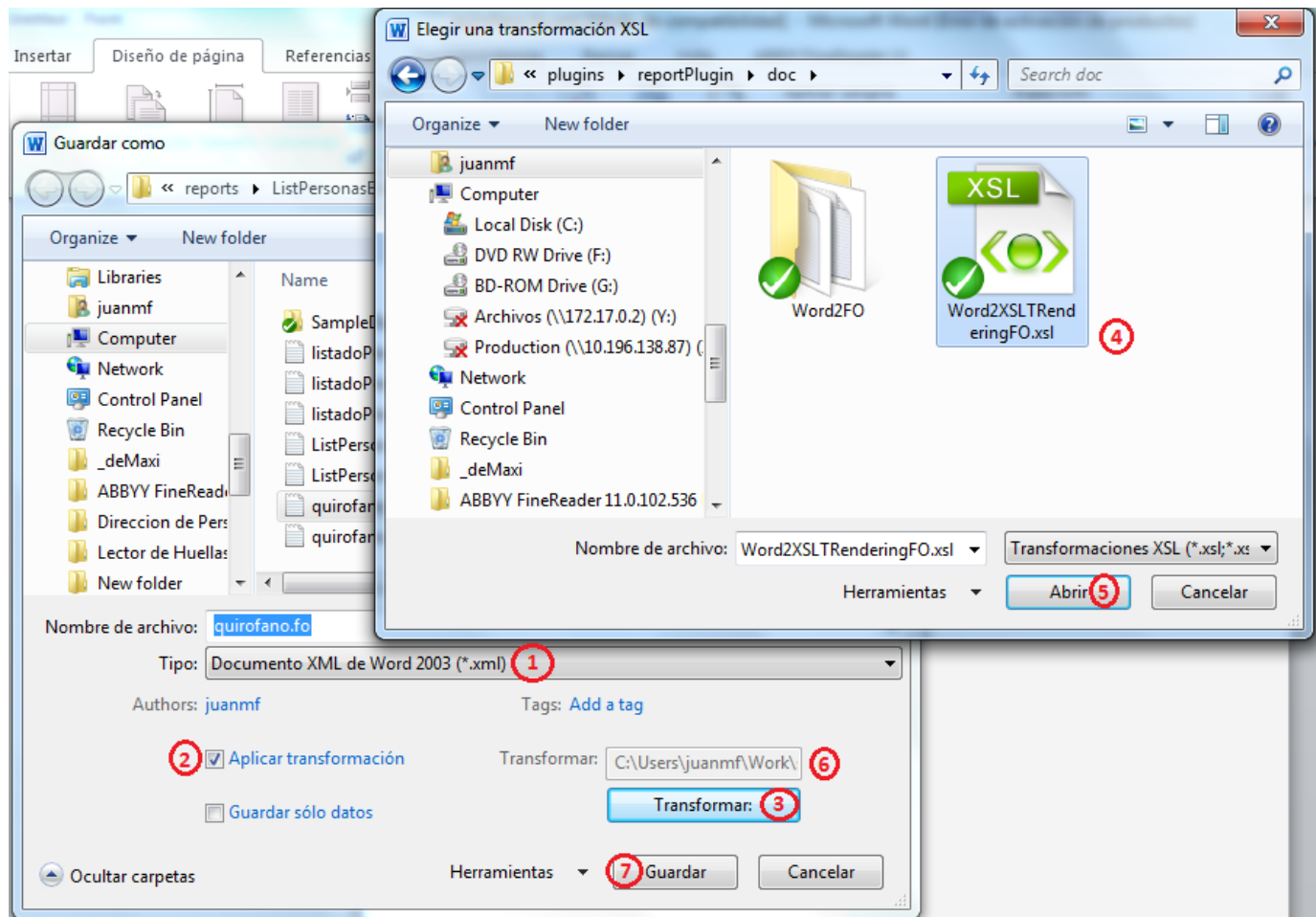
Red dashed lines connect the hash characters in the table to these labels: the first hash character connects to 'Comentario [j1]: SubNode', and the subsequent three hash characters connect to 'Comentario [j2]: day', 'Comentario [j3]: month', and 'Comentario [j4]: year' respectively.

The previous template should render something like:

The screenshot shows the rendered report. It consists of a header section with the text "Some Text" and a table below it. The table has three columns: Day, Month, and Year. The values in the first row are 15, 6, and 2012.

Day	Month	Year
15	6	2012

To achieve that, we need to save the word document in Word 2003 XML format (WordML) (ReportStylesheet.xml). Once you saved the report template in WordML you can proceed to save it again but this time applying the transformation to generate the actual XSLT report template (ReportStylesheet.xsl) *follow the steps in the image*.



Note that Word2FO directory **is not** packaged with this tool, you need to download it from RenderX in the Free tools Section at <http://www.renderx.com/download/shop.html> and decompress it next to Word2XSLTRenderingFO.xsl otherwise it will complain about unsatisfied dependencies.

How to configure the Engine

Once you have ReportStylesheet.xsl saved, you need to add it to the new report configuration in conjunction with a provider method that must generate the XML data that will be handed over to the Report stylesheet.



The provider method is optional, if you use the engine as a library instead of as a controller with its own route, you can generate the XML data as you wish. More on that later...

Provider method example follows:

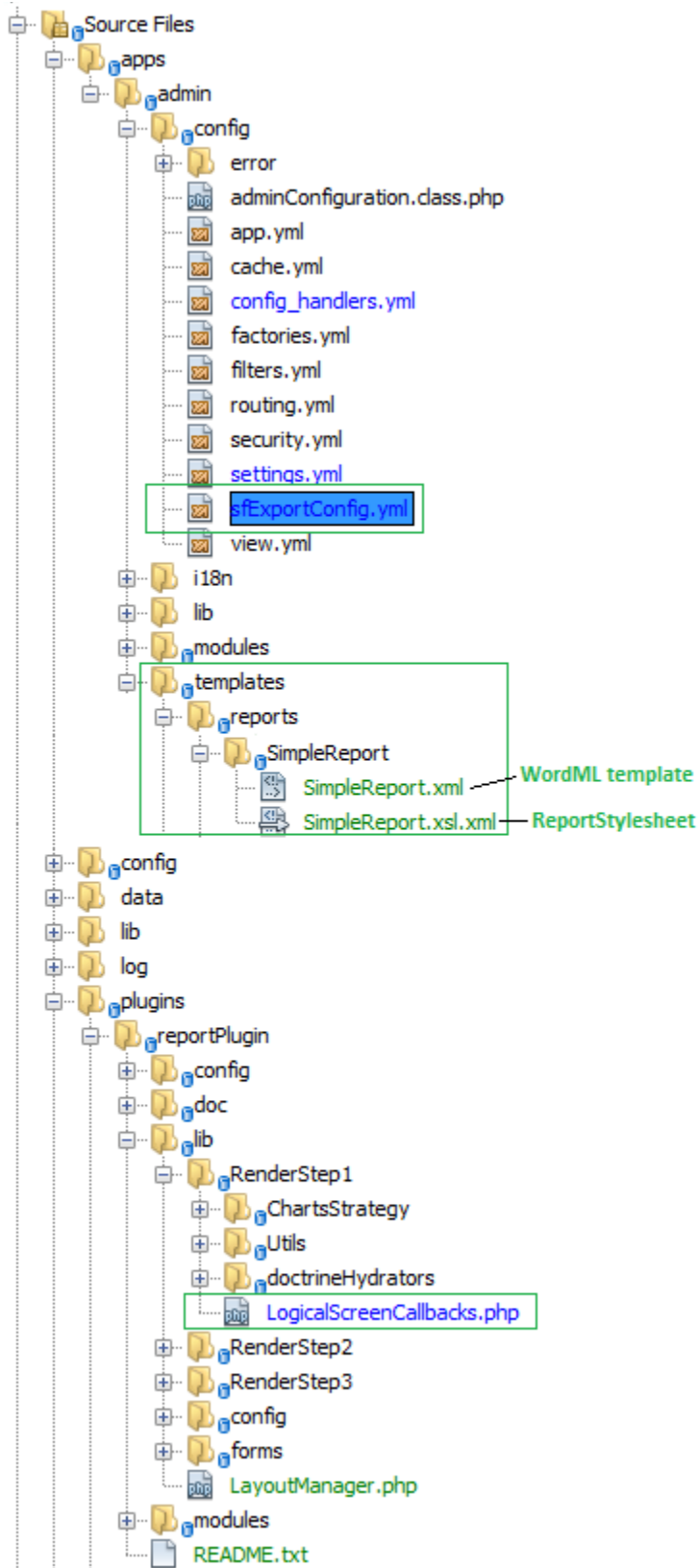
```
// projects\src\plugins\reportPlugin\lib\RenderStep1\LogicalScreenCallbacks.php
/**
 * This class helps in the Step1 of a three steps Process.
 *
 * The process consists of the following steps:
 *
 * Step 1 consists in generating the XML based raw data for the report. RawXMLData
 * Step 2 consists in merging this raw data with a XSL-FO template to give it
 * the presentation information. XSL-FO acts as an intermediate language used
 * to render the final report in any format with a last transformation.XSL-FO
 * Step 3 consists on rendering the XSL-FO representation (the intermediate
 * language) of the report to the desired output format. FinalReport
 *
 * This class should be used to add the methods that creates or access the
 * RawXMLData and returns it to be processed in Step2.
 *
 * To help extract data from database in XML format we have a Doctrine XML Hydrator
 * you can see a usage example in LogicalScreenCallbacks::helloworld().
 *
 * @author Juan Manuel Fernandez <juanmf@gmail.com>
 * @see %sf_plugins_dir%/reportPlugin/config/sfExportConfig.yml
 */
class LogicalScreenCallbacks
{
    /**
     * This method generates the xml data for the 'How to create a simple Report'
     * topic of the Report Engine tutorial
     */
    public static function howToSimple()
    {
        $xml = <<<XML
<root>
  <Node>
    <SubNode>Some Text</SubNode>
  </Node>
  <date>
    <day>15</day>
    <month>6</month>
    <year>2012</year>
  </date>
</root>
XML;
        $q = new DOMDocument();
        $q->loadXML($xml);
        return $q;
    }
}
```

Now that we have both, **data** and **report Stylesheet** we need to tell the report engine that they go together. For that we use a YAML configuration file that associates the Stylesheet with the data provider method, *note that the callback is a [ClassName, methodName] notation*:

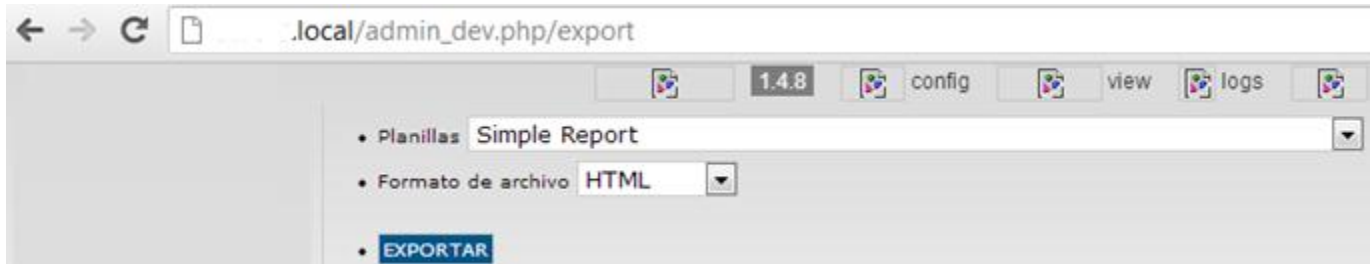
```
# project\src\apps\admin\config\sFExportConfig.yml
reports:
  SimpleReport:
    name: Simple Report
    step1: # Cooking Data, optional Step.
      logical_screen_callback: [LogicalScreenCallbacks, howToSimple]
      # e.g. [p1, p2, ..] using this key, the action can pass parameters too.
      callback_params: ~
    step2: # Merging data and Report XSLT Stylesheet
      structure:
        default:
          layout:
            # No need to know this for now.
            class: ~
            # An XPath expression selecting data nodes for this view.
            nodes: '/root'
          xslt_structure:
            # A XML to FO transformation adding structure. Path from %sf_app_dir%
            sheet: '/templates/reports/SimpleReport/SimpleReport.xsl.xml'
            # Associative array with relevant parameters for this xslt.
            params: {}
          xslt_style:
            # A FO to FO transformation adding style. Path from %sf_app_dir%
            sheet: ~
            # Associative array with relevant parameters for this xslt.
            params: {}
```

The files should be placed accordingly, as described in the configuration. Note that the paths in config are relative to the *apps* directory, in the following image you can see the relevant files.

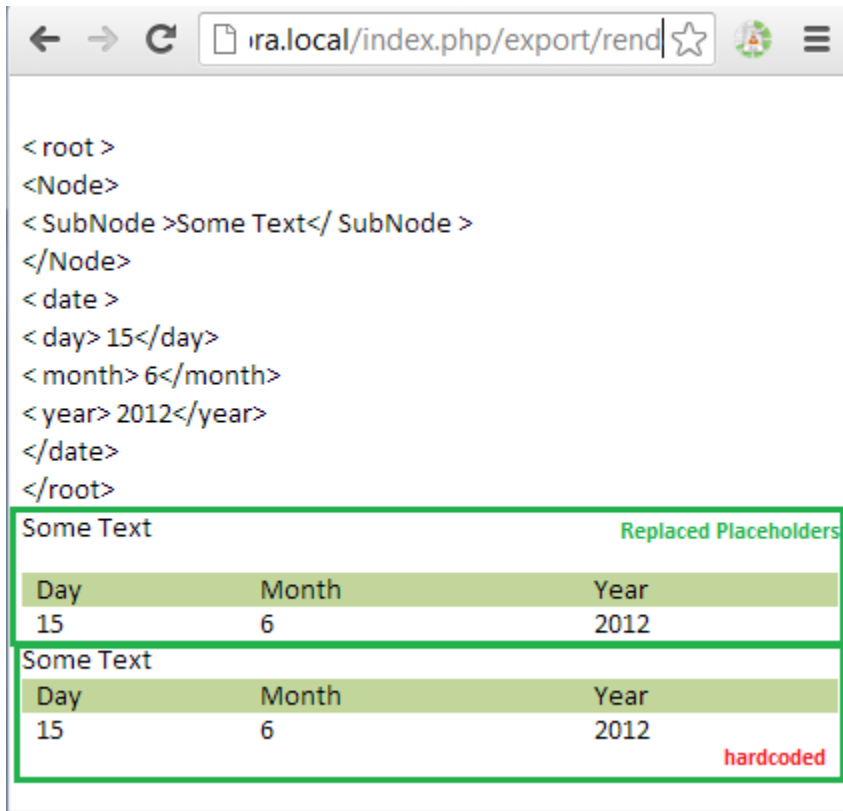
File hierarchy:



Then you go to the “<http://virtualHost.local/index.php/export>” route to see the report selection form, which now contains the new report descriptor, remember that *Simple Report* is the **name** configuration value:

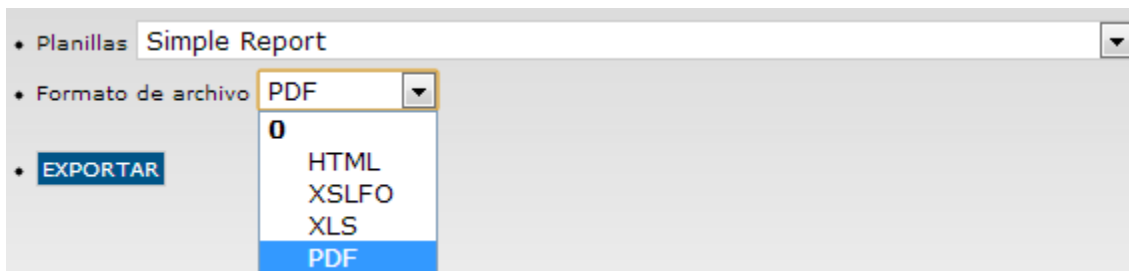


If you click “EXPORTAR” you get this:

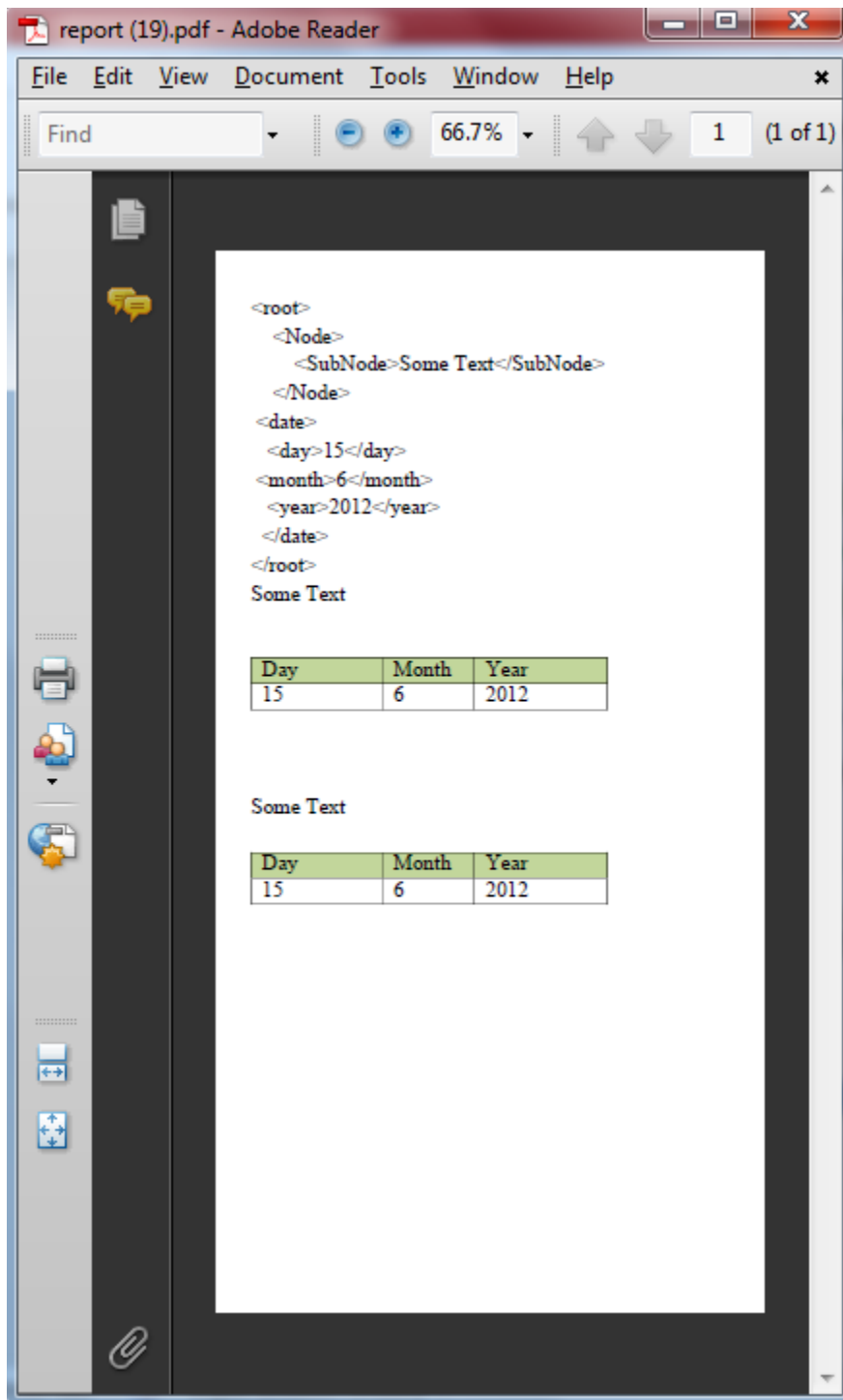


Note: Chrome doesn't show borders if they are thinner than “1pt;”, but Firefox'd show them and they are in the HTML markup.

If you select PDF and you have ApacheFOP webservice running (or the local executable, which still needs a calling rederer, @see Step3);

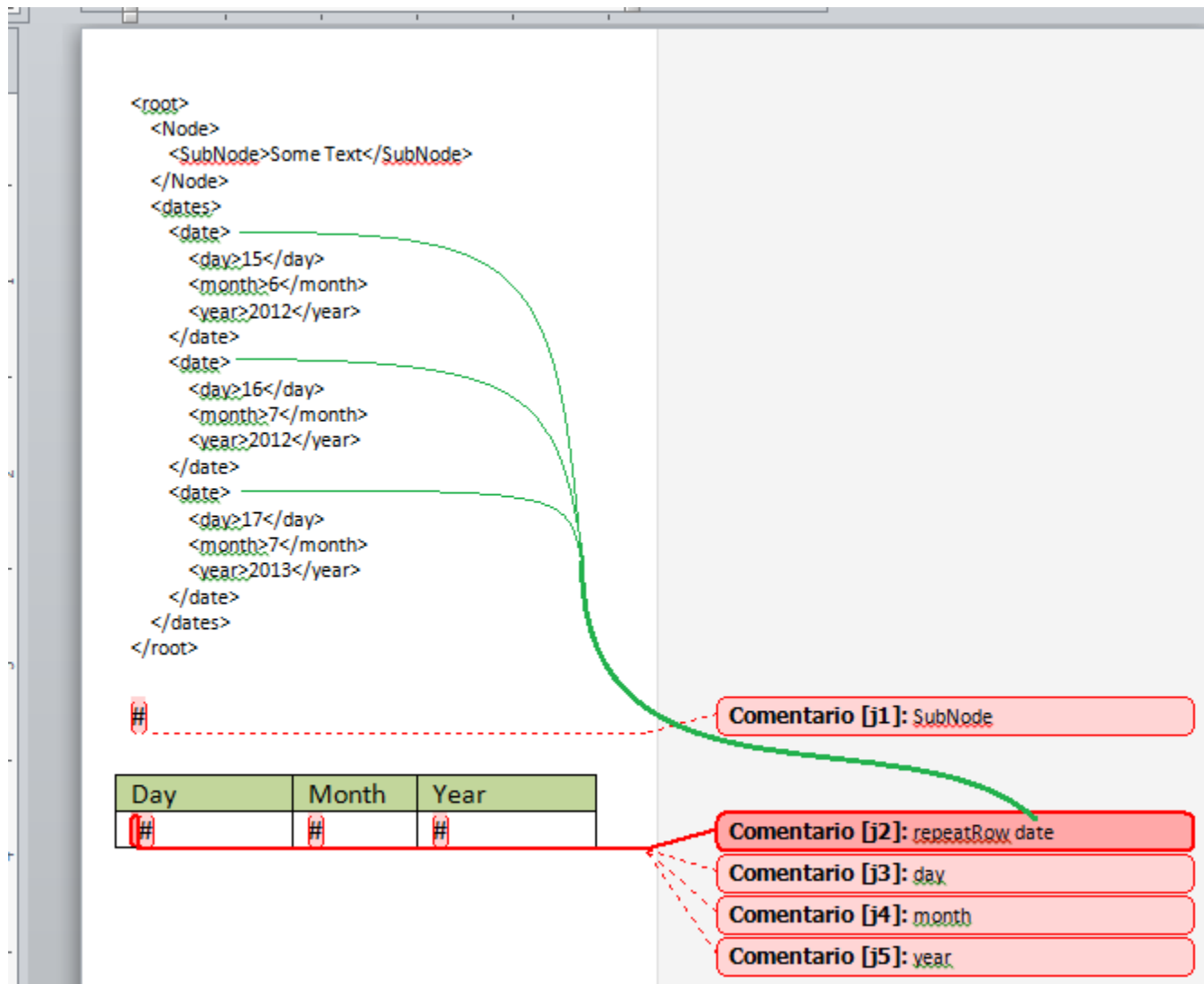


You'd get this (note that the 2nd table is hardcoded in the report template, the 1st has the placeholders):



A Slightly Harder Report, With Repeating Rows

The only difference (*besides the irrelevant fact that I removed the hardcoded “expected” table*) with the *Simple Report* example is that this time I added a new Comment surrounding a **leading white space in the 1st cell** with a command that gets interpreted by the modified Word2FO.xsl stylesheet and generates a new XSLT template that will repeat for each **date** and will be called from inside the table body. It’s important to know that whenever you use a repeat command, either **repeatTable** , **repeatRow** or **repeatCell**, the context for the repeating block (table, Row or Cell) is the current repeating node being iterated. In the following example that’d be **1st date** first, **2nd date** after and finally **3rd date** node. Therefor the **day**, **month** and **year** node selectors in the following Word Comments are relative to their **date ancestor** (that also happens to be parent, but that doesn’t matter).



When you select the HTML output you get this:

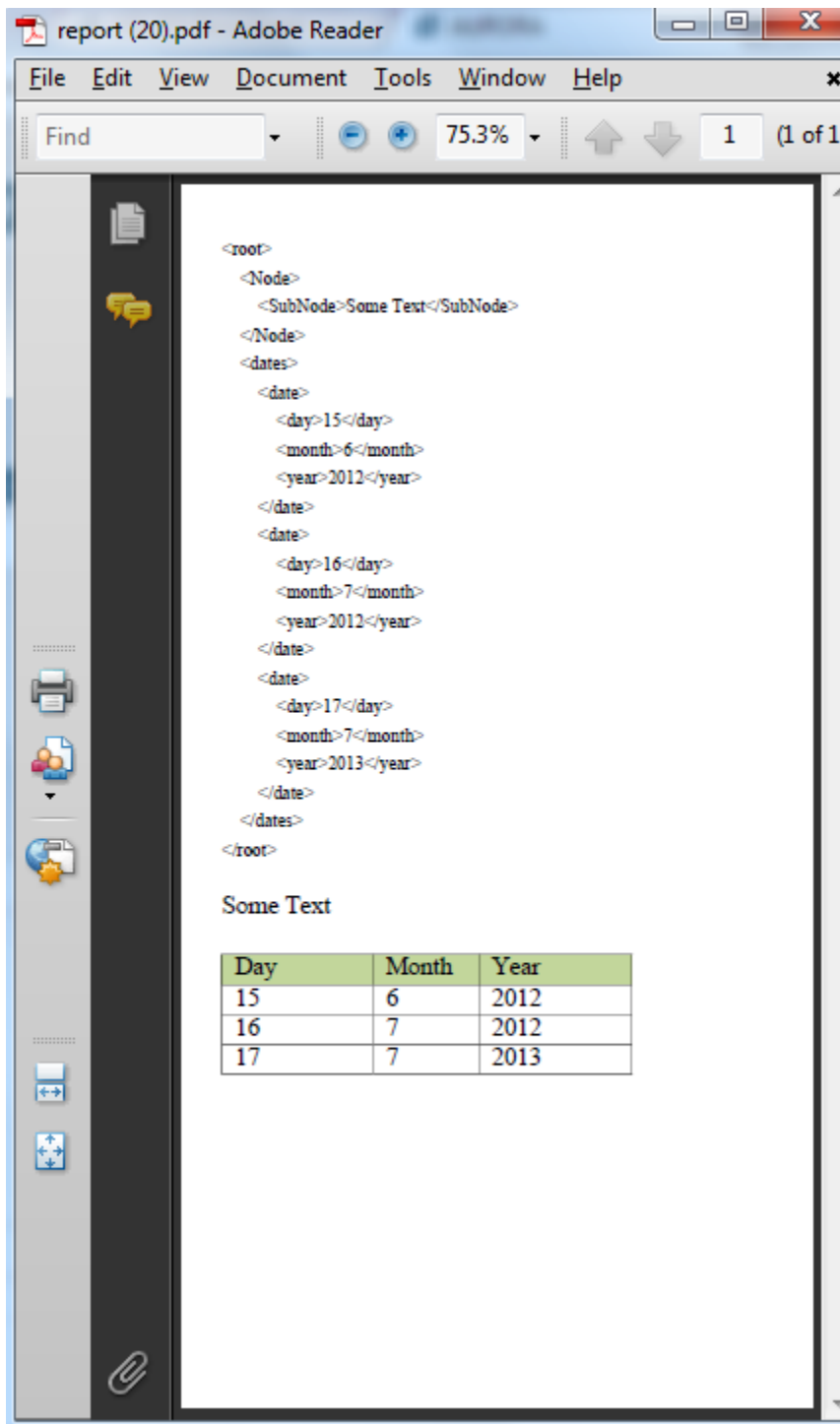
The screenshot shows a web browser window with the address bar displaying `i.local/index.php/export/render`. The main content area displays XML data and its rendered HTML output. The XML data is as follows:

```
<root>
<Node>
<SubNode>Some Text</SubNode>
</Node>
<dates>
<date>
<day>15</day>
<month>6</month>
<year>2012</year>
</date>
<date>
<day>16</day>
<month>7</month>
<year>2012</year>
</date>
<date>
<day>17</day>
<month>7</month>
<year>2013</year>
</date>
</dates>
</root>
```

Below the XML data, the text "Some Text" is displayed. Underneath, there is a table with three columns: Day, Month, and Year. The table contains three rows of data, each corresponding to a date entry in the XML. The first row (Day 15, Month 6, Year 2012) is highlighted with a green border. The second row (Day 16, Month 7, Year 2012) is highlighted with an orange border. The third row (Day 17, Month 7, Year 2013) is highlighted with a black border.

Day	Month	Year
15	6	2012
16	7	2012
17	7	2013

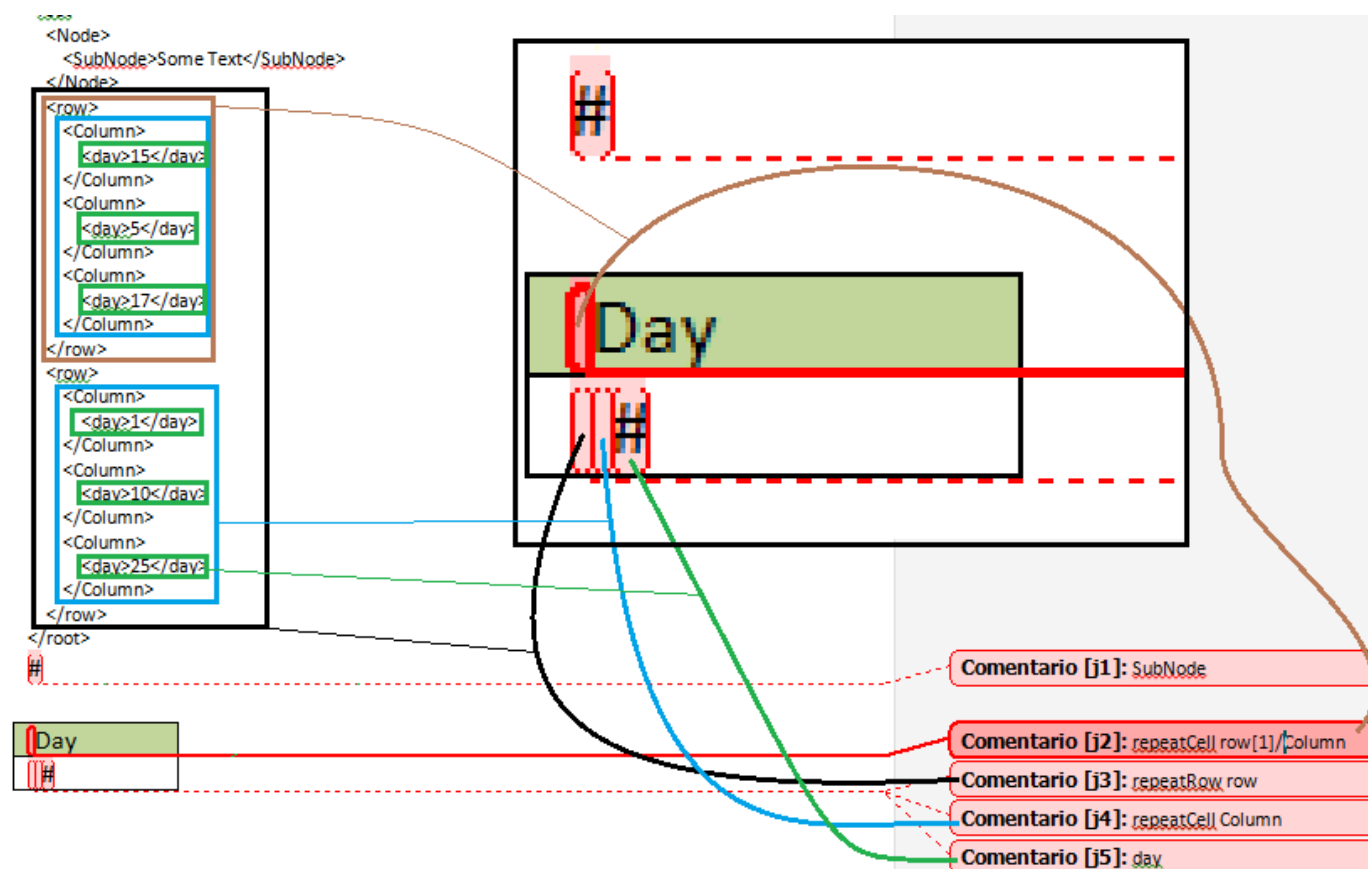
For the PDF output of the same report you get:



As you can see, page metrics are preserved in the PDF output, margins, page size and orientation, etc.

Repeating Cells, for variable Column number

Now we'll make cells repetitive, inside repetitive columns, for that I changed the data so we have a matrix of days, with an XPath view like this "//row/Column/day"



The PDF output of this Report StyleSheet looks like this.

The screenshot shows a PDF viewer interface with a search bar at the top containing 'Find', zoom controls at 76.7%, and page navigation showing '1 (1 of 1)'. The main content area displays an XSL-FO document. The XML structure is as follows:

```
<root>
  <Node>
    <SubNode>Some Text</SubNode>
  </Node>
  <row>
    <Column>
      <day>15</day>
    </Column>
    <Column>
      <day>5</day>
    </Column>
    <Column>
      <day>17</day>
    </Column>
  </row>
  <row>
    <Column>
      <day>1</day>
    </Column>
    <Column>
      <day>10</day>
    </Column>
    <Column>
      <day>25</day>
    </Column>
  </row>
</root>
```

Below the XML, the rendered output is shown:

Some Text

Day	Day	Day
15	5	17
1	10	25

As you can see, Header cells repeated three times, once for each **Column** node inside 1st **row** (that's "repeatCell row[1]/Column"). The two body rows obey to the two **row** nodes inside **root** (that's "repeatRow row"). On each row in turn, we get three cells (that's "repeatCell Column"), this works because in a current **row** context, XPath finds only 3 **Column** nodes.

Obviously, these repetitive cells would also work without repetitive rows. This example shows both together.

Adding Images to your report

As with commented “#” placeholder you can add commented images to your report template and they will be converted into an `<fo:external-graphic>` with the contents taken from a XML node containing a base64 encoded image, denoted by the comment, see the illustrative pic. The size and location of the image will be preserved, but the content will be replaced by your data as selected by the XPath expression “`./chartNode`” in this case, with `<root>` as context, for repetitive elements (tables, rows and cells) the context would be each matching node.

The screenshot shows a report template with a table at the top, a placeholder image below it, and a list of comments on the right. The table has four columns: DNI, Nombre, Fecha Nacimiento, and Localidad. The placeholder image is a red rectangle. The comments list on the right includes:

- Comentario [j1]: nombre
- Comentario [j2]: apellido
- Comentario [j3]: repeatTable Persona
- Comentario [j4]: nro_documento
- Comentario [j5]: nombre
- Comentario [j6]: fecha_nacimiento
- Comentario [j7]: Localidad_Collection/nombre
- Comentario [j8]: chartNode

At the bottom right, there is a button labeled "Área de revisiones".

This XPath expressions are written with the following XML in mind, which gets generated with the method `LogicalScreenCallbacks::listadoPersonasXedad()`:

```

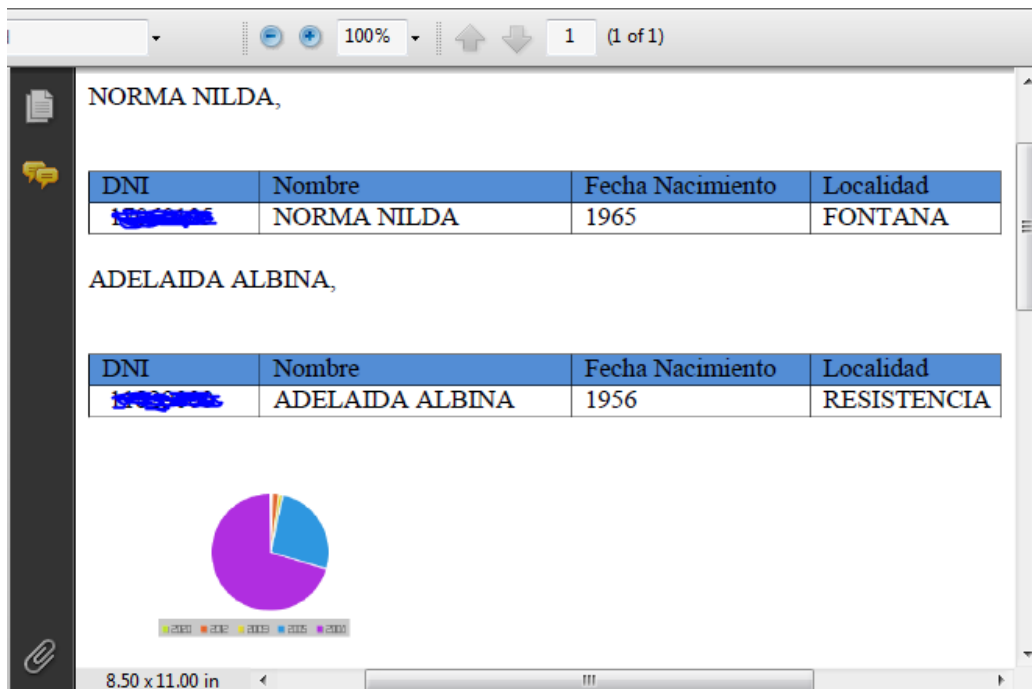
1  <?xml version="1.0"?>
2  <result>
3    <Persona_Collection>
4      <Persona>
5        <id>78db0f3c-c002-102f-bf64-001a64ab842a</id>
6        <nro_documento>17****</nro_documento>
7        <nombre>NORMA NILDA</nombre>
8        <YEAR>1965</YEAR>
9        <PersonaDomicilio_Collection>
10       <PersonaDomicilio>
11         <id>10494318-c00b-102f-bf64-001a64ab842a</id>
12         <Domicilio_Collection>
13           <id>1048cd66-c00b-102f-bf64-001a64ab842a</id>
14           <Localidad_Collection>
15             <id>1194</id>
16             <nombre>FONTANA</nombre>
17           </Localidad_Collection>
18         </Domicilio_Collection>
19       </PersonaDomicilio>
20     </PersonaDomicilio_Collection>
21   </Persona>
22   <Persona>
40 </Persona_Collection>
41 <chartNode>iVBORwOKGgoAAAANSUth...=</chartNode>
42 </result>

```

For creating some relevant image in `<chartNode>` you can use [pChart](#) for instance, as follows:

```
public static function listadoPersonasXedad()
{
    $q = Doctrine_Query::create()
        ->from('Persona p')
        ->innerJoin('p.PersonaDomicilio pd')
        ->innerJoin('pd.Domicilio d')
        ->innerJoin('d.Localidad l')
        ->select(
            'p.nro_documento, p.nombre, YEAR(p.fecha_nacimiento), '
            . 'pd.id, d.id, l.id, l.nombre'
        )->limit(2)
        ->execute(array(), 'xml');
    $birthYearCount = Doctrine_Query::create()
        ->from('Persona p')
        ->select(
            'YEAR(p.fecha_nacimiento) as years, '
            . 'count(YEAR(p.fecha_nacimiento)) as counted'
        )->groupBy('years')
        ->orderBy('years DESC')
        ->limit(5)
        ->execute(array(), 'xml');
    /** Here I use pchart to make a pie chart counting YEAR frequency */
    $chart =
        ListPersonasEdadCharts::listadoPersonasXedadCreateChart($birthYearCount);
    $encodedChart = base64_encode($chart);
    /* @var $q DomDocument */
    $chartNode = $q->createElement('chartNode', $encodedChart);
    $q->documentElement->appendChild($chartNode);
    return $q;
}
```

This Template, merged with the shown data XML, generated by the above method, generates the following PDF:

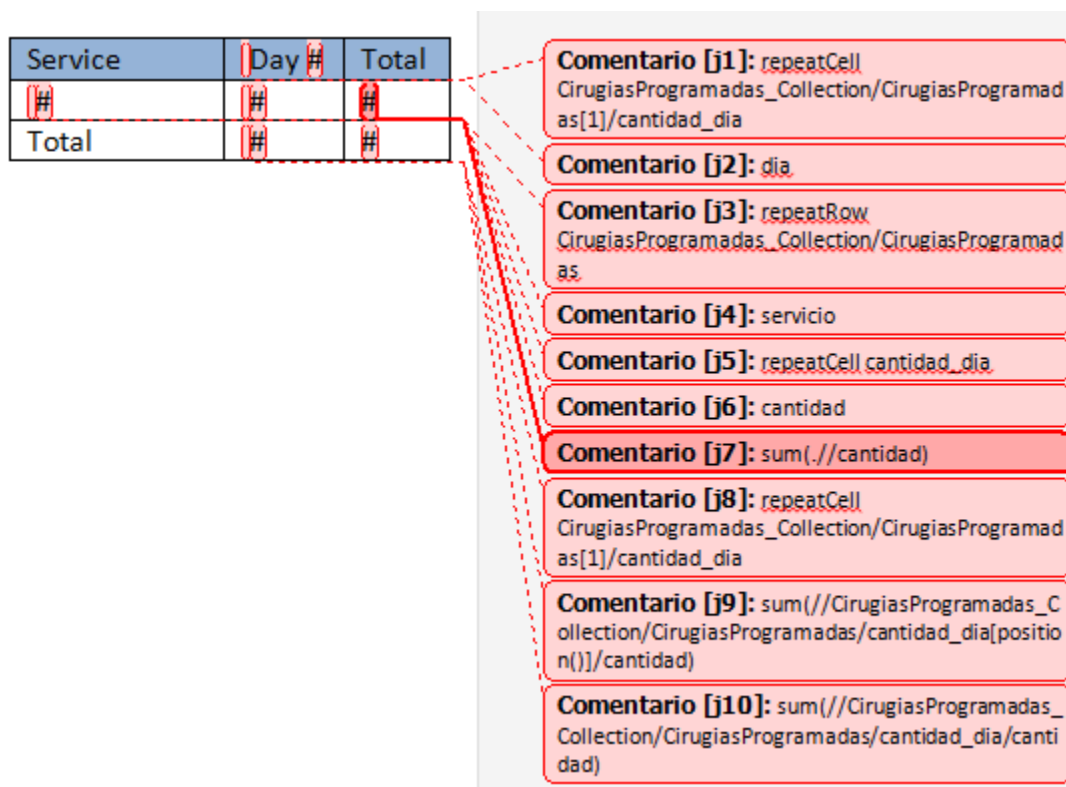


Adding Expressions

Sometimes you might need to summarize or count some data nodes, for this trivial operations, it might be easier to process those nodes in the report template (with XSLT and XPaht) rather than adding calculated data to your raw XML data structure. By default, commented “#” are considered simple Xpath expressions, like node names or relative routes, relative to **<root>** or any repetitive table row or cell context.

So if you put **someNode** in the comment it will match any **someNode** element from the current context down. Originally current context is **<root>** but when you use **repeatTable**, **repeatRow** or **repeatCell** the node who’s name you use as the repeat parameter will be the context for the expressions inside the repetitive structure e.g. When we used “**repeatRow date**” the context for each **day**, **month** and **year** inside a row is the respective date node.

Let’s consider the following template:



As you can see, it has a lot of directives for such a little table. This is because this little table will expand in rows (for each **service**) and columns (for each **day** in each **service** context) and also will summarize each row totals and a global total value, without getting them from the raw XML data, but calculating it with XPath expressions, directly input in the word comments.

These expressions are designed to query the following raw data XML:

```
<root>
  <CirugiasProgramadas_Collection>
    <CirugiasProgramadas> <!--Scheduled surgery [1]-->
      <servicio>Cardiología</servicio> <!--1st Service name-->
      <cantidad_dia>
        <cantidad>0</cantidad> <!--surgery count-->
        <dia>01</dia> <!--day-->
      </cantidad_dia>
    </CirugiasProgramadas>
  </CirugiasProgramadas_Collection>
```

```

    <cantidad_dia>
      <cantidad>0</cantidad> <!--surgery count-->
      <dia>02</dia> <!--day-->
    </cantidad_dia>
  :::
  <cantidad_dia>
    <cantidad>1</cantidad> <!--surgery count-->
    <dia>30</dia> <!--30th day of the month-->
  </cantidad_dia>
</CirugiasProgramadas>
:::
<CirugiasProgramadas> <!--Scheduled surgery [1]-->
  <servicio>Urología</servicio> <!--5th Service name-->
  <cantidad_dia>
    <cantidad>0</cantidad> <!--surgery count-->
    <dia>01</dia> <!--day-->
  </cantidad_dia>
  <cantidad_dia>
    <cantidad>0</cantidad> <!--surgery count-->
    <dia>02</dia> <!--day-->
  </cantidad_dia>
  :::
  <cantidad_dia>
    <cantidad>1</cantidad> <!--surgery count-->
    <dia>30</dia> <!--30th day of the month-->
  </cantidad_dia>
</CirugiasProgramadas>
</CirugiasProgramadas_Collection>
</root>

```

I'll explain each word comment, so you get the idea:

J1: repeatCell CirugiasProgramadas_Collection/CirugiasProgramadas[1]/cantidad_dia

Will match the 31 `cantidad_dia` nodes inside the 1st `CirugiasProgramadas`, thus repeating the header cell "Day #" 31 times, each time with a different `cantidad_dia` as context

J2: dia

With `cantidad_dia` as context, matches the only `dia` node it has, for each "Day #" repetition.

J3: repeatRow CirugiasProgramadas_Collection/CirugiasProgramadas

Will match all `CirugiasProgramadas` (we have 5) inside `CirugiasProgramadas_Collection`, thus repeating body rows 5 times, with a different `CirugiasProgramadas` node as context each time.

J4: servicio

With `CirugiasProgramadas` as context it will match the only `servicio` node inside it and fill the service cell for each row.

J5: repeatCell cantidad_dia

With `CirugiasProgramadas` (since we are in the repeating row) as context it will match the 31 `cantidad_dia` nodes inside it and repeat the cell 31 times, adding 31 columns.

J6: cantidad

With `cantidad_dia` (since we are in the repeating cell) as context it will match the only `cantidad` node inside it and fill the cell, replacing "#" with `cantidad` value.

J7: `sum(../cantidad)`

In the final row cell, with `CirugiasProgramadas` as context it will sum all `cantidad` nodes it finds inside `CirugiasProgramadas` calculating the total amount of surgeries for the current context service. (*1 generated template hand editing needed)

J8: `repeatCell CirugiasProgramadas_Collection/CirugiasProgramadas[1]/cantidad_dia`

Will match the 31 `cantidad_dia` nodes inside the 1st `CirugiasProgramadas`, thus repeating the last row cell 31 times. Note that this time, we just use it for repeating, not for filling with any data.

J9: `sum(//CirugiasProgramadas_Collection/CirugiasProgramadas/cantidad_dia[position()]/cantidad)`

Inside each repeating cell in the last row (Totals row) we don't use the context node, as we need to summarize by column (i.e. one `cantidad` node on each row, and our context is only one row). We use an absolute XPath expression to summarize, for the current column, all `cantidad`, that's all the Nth `cantidad` in all rows, N being the current column. (*2 generated template hand editing needed)

J10: `sum(//CirugiasProgramadas_Collection/CirugiasProgramadas/cantidad_dia/cantidad)`

This is a global absolute expression, summarizing all `cantidad` nodes in the data XML, the Total Total. (*1 generated template hand editing needed)

Note: these two pitfalls below are already fixed, no hand editing needed. Still I leave this documentation, in case you need to tweak the generate markup.

(*1) For XPath Expressions that are not compatible (or produce unexpected behavior) with a `../` prefix (so far hardcoded in the template generation) like `sum()`, `count()`, `../`, etc. you need to tweak the generated XSLT template by hand as follows:

Locate the broken expression, for **J7** it will look as follows:

```
<xsl:value-of select="../sum(../cantidad)">
```

Edit it (remove `../` prefix) so it looks like:

```
<xsl:value-of select="sum(../cantidad)">
```

(*2) For XPath Expressions that need to account for the position of the node currently being processed, like in **J9** where we need to get all `cantidad` nodes that are the Nth child of their parent row (see the useless `cantidad_dia[position()]` part). It doesn't produce the expected result, as the `[position()]` sub expression is not relative to the cell context, but to the `cantidad_dia` being traversed by the XPath expression itself, i.e. every `cantidad_dia` in turn.

The resulting expression looks like follows:

```
<xsl:value-of select="sum(//CirugiasProgramadas_Collection/CirugiasProgramadas/cantidad_dia[position()]/cantidad)">
```

Edit it so it looks like:

```
<xsl:variable name="pos" select="position()" />
```

```
<xsl:value-of select="
```

```
sum(//CirugiasProgramadas_Collection/CirugiasProgramadas/cantidad_dia[$pos = position()]/cantidad)"/>
```

(note the `cantidad_dia[position()]` sub expression turned into `cantidad_dia[$pos = position()]` and we added a new variable `$pos` that holds the current context cell's position before entering into the XPath expression)

Of course, you might as well add this value as `rad` data inside the input XML at runtime, but you have more options, consider also that, obviously, these hand editing changes are overridden if you re-generate the template.

And the resulting report looks as follows (this is not the exact same template but it has identical logic and same data XML input):

Reporte de Cantidad de cirugías del mes

Fecha y Hora de Impresión: 29/10/2012 02:10:51

Cirugías Programadas del mes: 2012-10

	01 (L)	02 (Ma)	03 (Mi)	04 (J)	05 (V)	06 (S)	07 (D)	24 (Mi)	25 (J)	26 (V)	27 (S)	28 (D)	29 (L)	30 (Ma)	31 (Mi)	Total
Cardiología	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
Centro Quirúrgico	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
Cirugía	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bucomaxilofacia	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Neurocirugía	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Urología	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
Total:	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	3

Control break reports

By nesting any number of tables and abusing of the repeatTable command, on borderless tables, in the word Comments you can get Control Break reports of any depth.

The diagram illustrates the structure of a report with nested tables and their corresponding comments. On the left, four nested table levels are shown, each with a red '#' icon in its top-left corner:

- Level 1 (outer most table)
- Level 2 (inner table)
- Level 3 (deeper table)
- Level 4 (inner most table)

Below the table levels, the corresponding XML-like code is displayed:

```
<root>
  <lv11>
    <lv11Name>Cars from World</lv11Name>
    <lv12>
      <lv12Name>Cars from America</lv12Name>
      <lv13>
        <lv13Name>Cars from Argentina</lv13Name>
        <lv14>
          <lv14Name>Cars from Corrientes</lv14Name>
        </lv14>
        <lv14>
          <lv14Name>Cars from Chaco</lv14Name>
        </lv14>
      </lv13>
      <lv13>
        <lv13Name>Cars from Brasil</lv13Name>
        <lv14>
          <lv14Name>Cars from Rio de
Janeiro</lv14Name>
        </lv14>
      </lv13>
    </lv12>
  </lv11>
</root>
```

On the right, a list of comments is shown, each with a red dashed line pointing to its corresponding table level:

- Comentario [j1]: repeatTable.lv11
- Comentario [j2]: lv11Name
- Comentario [j3]: repeatTable.lv12
- Comentario [j4]: lv12Name
- Comentario [j5]: repeatTable.lv13
- Comentario [j6]: lv13Name
- Comentario [j7]: repeatTable.lv14
- Comentario [j8]: lv14Name

Below the comments, a button labeled "Área de revisiones" is visible.

Two minutes later, after:

- Saving the new template, optional, and re-saving as XSLT using the Word2FO transformation, in the templates directory.
- Creating dummy method in LogicalsreenCallback:: howToControlBreak() that returns the shown XML.
- Editing reports Engine configuration to call howToControlBreak() and then render the new XSLT template.
- Clearing symfony cache and loading the export module.

I could select the new Control Break report from the menu and render it as HTML or PDF.

