Spice up Those Old Reports with SVG and JavaScript

Presented by: **Keisuke Miyako**

INTRODUCTION - 1

`GRAPH` is a high-level 4D command that lets you create graphic chart representations of data by simply specifying the desired graph type and an array of categories. The produced image is SVG (Scalable Vector Graphics), a format recognized by all major imaging software and suitable for high-resolution displays as well as printing. It is by far the easiest way to draw data driven charts in 4D.

The `GRAPH` command does what its is designed to do, that is, create charts based on the minimum set of instructions, but the downside is that charts created with this command are hardly customizable, which is not going to go down well with the end user if they are expecting professional quality charts customized for their business application.

As mentioned earlier, `GRAPH` generated charts are actually SVG documents. Therefore it is possible by means of DOM/SVG commands and/or the *4D SVG component* to tweak the chart color, layout, format, etc. by code, but such would require significant amount of extra coding as well as an understanding of the document structure to begin with. In fact, you could end up doing so much customization to the point where it might be better to ditch the `GRAPH` command completely and construct an SVG chart from scratch instead.

The purpose of this session is to save you from the trouble of going down that route. The accompanying component has the simplicity similar to that of the `GRAPH` command, while offering a wide range of customizable options that are easy to use and extend. It can be integrated seamlessly to your existing v11/v12 projects and lets you add spice to your chart generating code.

WHERE THE GRAPH COMMAND MIGHT FAIL YOU

Before discussing details of the component, it should be worth spelling out what exactly are the limitations of the `GRAPH` command that we want to overcome.

Limitation on the number of data sets

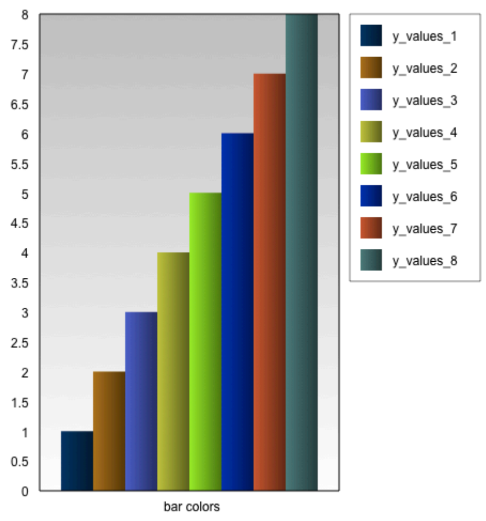
As specified in the documentation, only "up to eight data sets can be graphed."

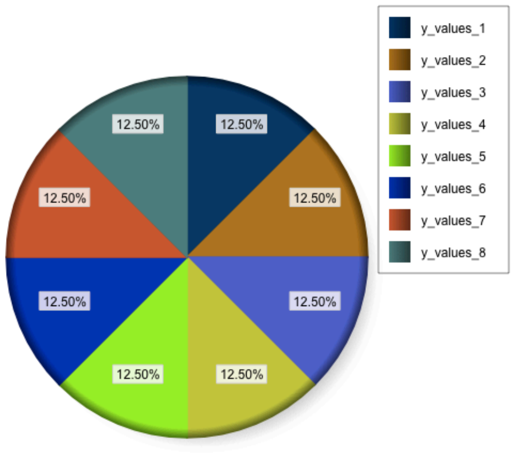
Note: The command will fail if more than 9 data sets are passed.

Fixed index color for chart pieces

The following gradients are always used for bars and pie pieces.

1	`rgb(0,51,102)/rgb(0,26,51)`
2	`rgb(179,112,0)/rgb(90,56,0)`
3	`rgb(85,82,204)/rgb(43,41,102)`
4	`rgb(188,204,51)/rgb(89,102,26)`
5	`rgb(115,251,29)/rgb(58,126,15)`
6	`rgb(19,15,180)/rgb(10,8,90)`
7	`rgb(215,72,29)/rgb(108,36,15)`
8	`rgb(64,127,126)/rgb(32,64,63)`



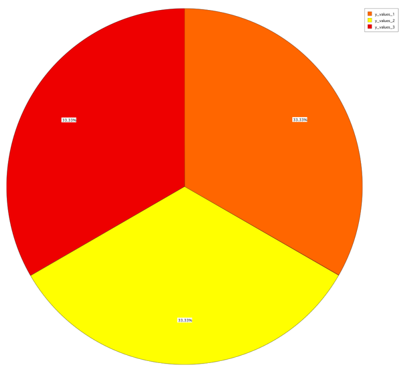


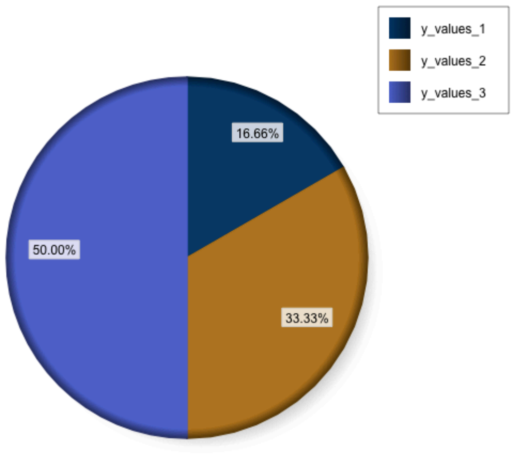
No graph titles

The `GRAPH SETTINGS` command only allows for setting the legend title.

No accumulative data set support for pie charts

As specified in the documentation, "pie charts graph only the first *yElements*." This means you get a different pie charts depending on whether you used the SVG engine or the *4D Chart* engine, since the latter takes into account all *yElements*.





```
ARRAY TEXT($x_values;3)
$x_values{1}:="y_values_1"
$x_values{2}:="y_values_2"
$x_values{3}:="y_values_3"

ARRAY REAL($y_values_1;3)
$y_values_1{1}:=1
$y_values_1{2}:=2
$y_values_1{3}:=3

ARRAY REAL($y_values_2;3)`ignored with svg
$y_values_2{1}:=3
$y_values_2{2}:=2
$y_values_2{3}:=1

GRAPH($graph;7;$x_values;$y_values_1;$y_values_2)
```

INTRODUCING THE GRAPH COMPONENT

Transition from the built in `GRAPH` command to using this component should be extremely easy, since the argument lists are practically identical. Listed below are the key features of this component.

1. Compatible with v11 and v12.

2. Supports SVG and HTML (JavaScript) charts.

3. SVG uses the 4D GRAPH template by default, HTML uses *Highcharts* template by default.

4. Both SVG and HTML charts support swapping of categories and x-values for all chart types.

5. SVG charts are fully customizable by property key-value API.

6. HTML charts are fully customizable by *jQuery* API.

7. Both SVG and HTML charts support adding custom templates.

8. Supports direct drawing of charts from array-based listboxes.

Note: For more information about Highcharts JS please visit:

http://www.highcharts.com/

Note: For more information about jQuery please visit:

http://jquery.com/

USING THE GRAPH COMPONENT

Suppose your original code uses the `GRAPH` command as illustrated below.

```
ARRAY TEXT($x_values;3)
$x_values{1}:="Region 1"
$x_values{2}:="Region 2"
$x_values{3}:="Region 3"

ARRAY REAL($y_values_1;3)
$y_values_1{1}:=1
$y_values_1{2}:=0
$y_values_1{3}:=3

ARRAY REAL($y_values_2;3)
$y_values_2{1}:=2
$y_values_2{2}:=3
$y_values_2{3}:=3

ARRAY REAL($y_values_3;3)
$y_values_3{1}:=1
$y_values_3{2}:=1
$y_values_3{3}:=2

ARRAY REAL($y_values_4;3)
$y_values_4{1}:=4
$y_values_4{2}:=5
$y_values_4{3}:=3

GRAPH($graph;1;$x_values;$y_values_1;$y_values_2; $y_values_3;$y_values_4)
```

After installing the component, modify the final line to call the component method like this:

```
C_TEXT($graphData)

$graphData:=GDATA_Create_from_arrays (->$x_values; ->$y_values_1;\
 ->$y_values_2; ->$y_values_3; ->$y_values_4)
```

As you can see, apart from the fact that we do not specify the graph type at this point and that we pass pointers to arrays instead of the arrays themselves, the argument lists are very similar.

`$graphData` will contain a proprietary XML representation of the data source arrays needed to produce a graph. You don't need to care too much about the exact content of this XML structure.

To create an SVG graph of type 1 (as was the intent of the original code) you add the following code.

```
C_PICTURE($graph)

$graph:=GRAPH_Create_SVG ($graphData;"Graph1")
```

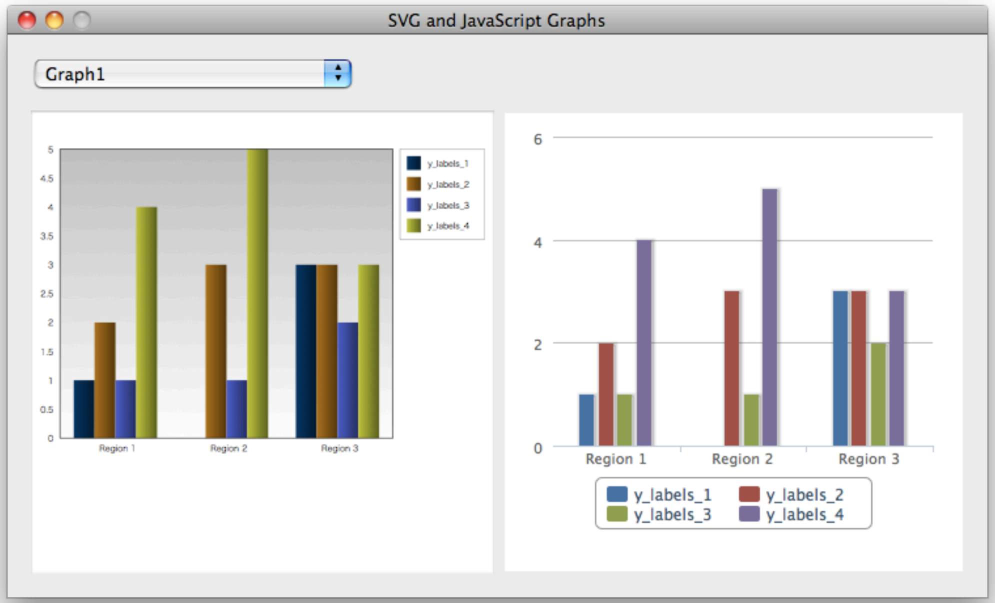
`$graph` will contain an SVG image of the created graph.

To create an HTML (JavaScript) graph of type 1, you add the following code.

```
C_TEXT($graph)

$graph:=GRAPH_Create_HTML ($graphData;"Graph1")
```

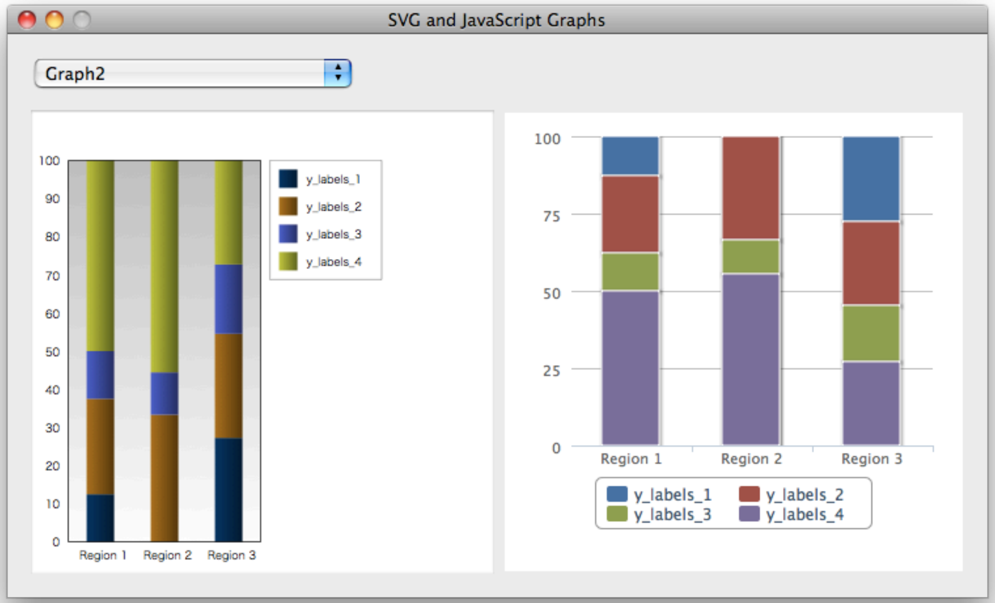
`$graph` will contain the full HTML source code of the created graph.

The advantage of this two-tiered system is that the data (managed the `GDATA` method) element is completely separated from the graphic (managed by the `GRAPH` method) element and therefore the same data structure can be used to produce different types of charts, SVG or HTML versions of the graph type 1 in this case.



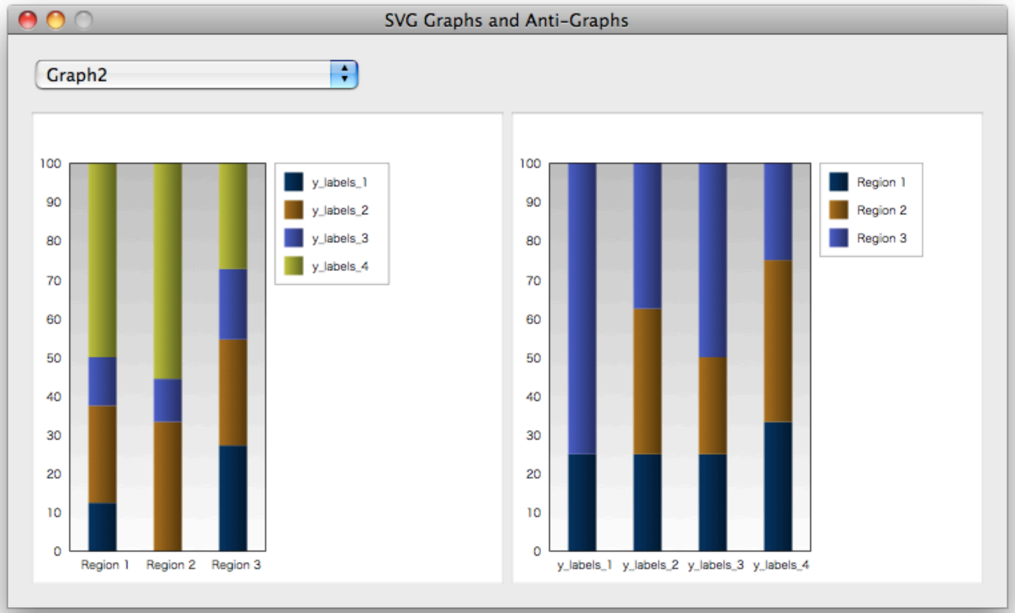
To change the graph type, simply use the same `$graphData` with a different graph name, as shown below.

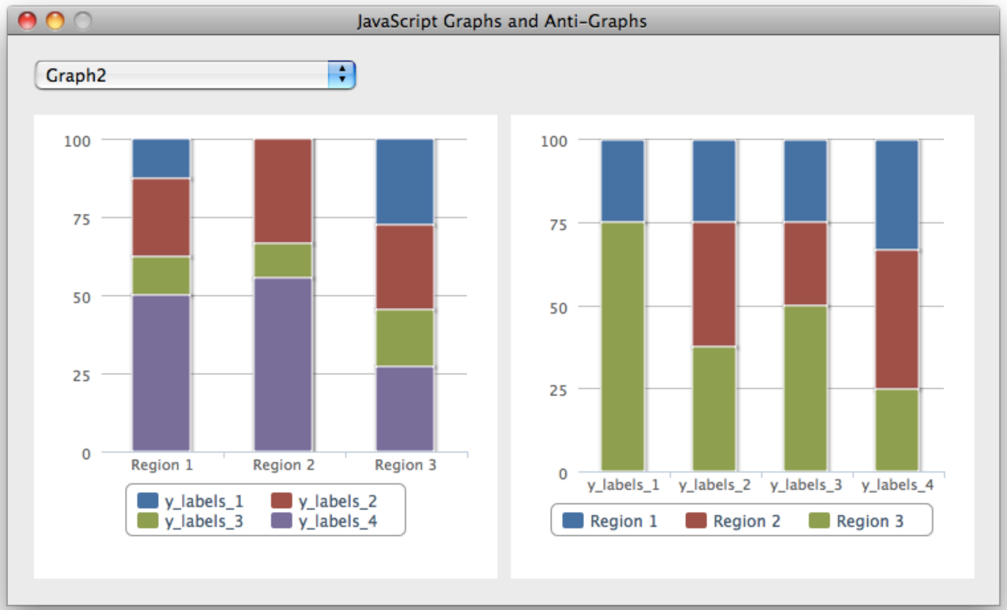
```
$graph:=GRAPH_Create_SVG ($graphData;"Graph2")

$graph:=GRAPH_Create_HTML ($graphData;"Graph2")
```



To swap the x-values and categories to produce atypical or anti-graphs, add an "A" to the graph name.

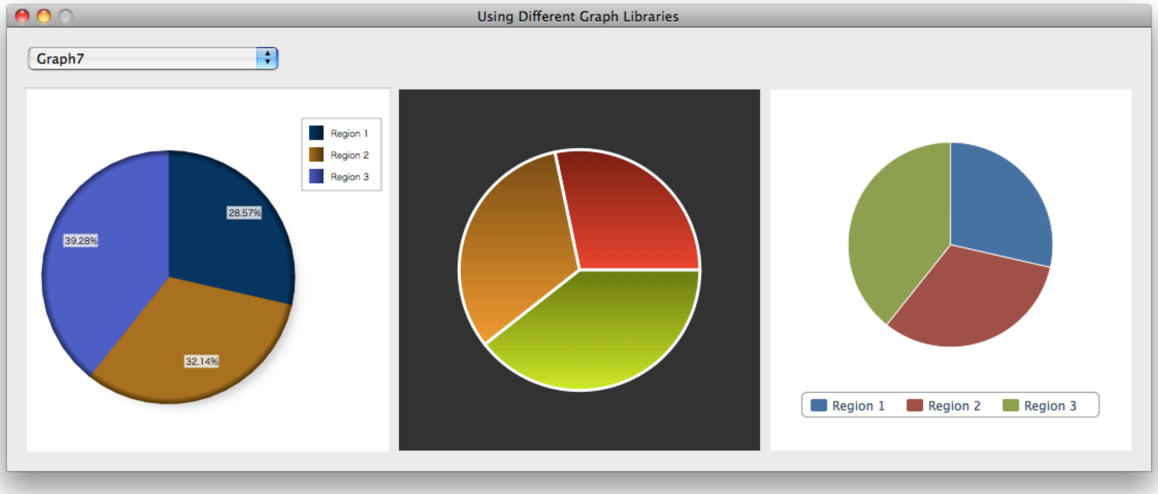
```
$graph:=GRAPH_Create_SVG ($graphData;"Graph2A")

$graph:=GRAPH_Create_HTML ($graphData;"Graph2A")
```





The component supports adding any number of extra graph drawing libraries. As an example, there is a *Raphaël* version of the pie chart (graph type 7) which can be called as illustrated below.

```
$graph:=GRAPH_Create_HTML ($graphData;" Raphael.Graph7")

$graph:=GRAPH_Create_HTML ($graphData;" Raphael.Graph7A")
```



Note: For more information about Raphaël please visit:

http://raphaeljs.com/

THE ALTERNATIVE SYNTAX FOR GDATA

In addition to the `GRAPH` style syntax described above, the `GDATA` method takes this alternative method.

```
ARRAY POINTER($y_values;4)
$y_values{1}:=->$y_values_1
$y_values{2}:=->$y_values_2
$y_values{3}:=->$y_values_3
$y_values{4}:=->$y_values_4

$graphData:=GDATA_Create_from_arrays (->$x_values;->$y_values)
```

Of course, the resulting XML is exactly the same.

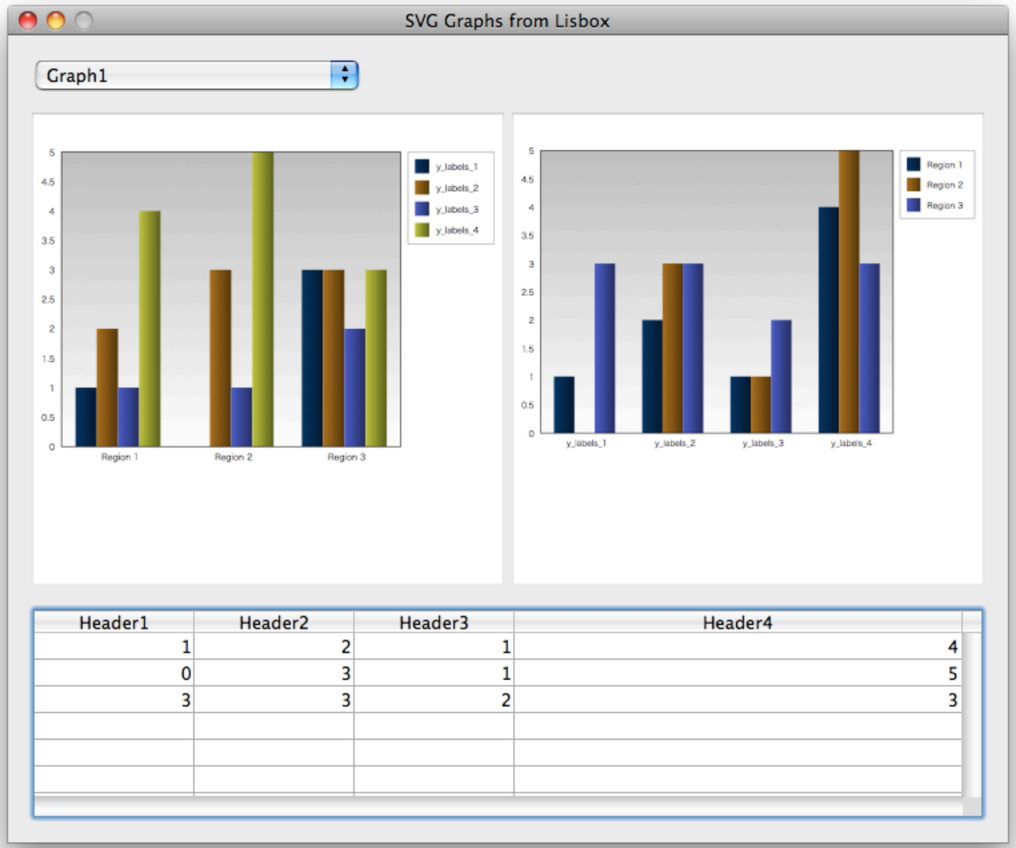
The advantage of this style is that it is more generic and prevents the line from getting annoyingly long.

Note: You can also pass more than 9 data sets using the primary syntax.

DRAWING A GRAPH BASE ON A LISTBOX

To create a GDATA structure from a listbox, pass a pointer to the listbox, which should be an `ARRAY BOOLEAN`.

```
$graphData:=GDATA_Create_from_arrays (->$x_values;->LISTBOX1)
```



Header1	Header2	Header3	Header4
1	2	1	4
0	3	1	5
3	3	2	3

Note: Invisible rows and columns are exempt from the data set.

UNDER THE HOOD

The component heavily uses XSLT, which is only natural, since XSLT is what the built-in `GRAPH` command basically works around.

REVERSE ENGINEERING THE GRAPH COMMAND

The XSL style sheets used internally by the `GRAPH` command can be located inside the 4D application package at `Resources/XSL/`. How data should be formatted prior to being fed to these style sheets is loosely defined in `graph_datas.xsd` found at `Resources/XSD/`.

Note: Ideally, the data format should be consistent across all graph data types. However, 4D actually uses 3 data formats; one format for graph types 1, 3, 4, 5, another for graph type 2 and another for graph type 7.

When the `GRAPH` command is called, 4D first produces an XML data format structure, applies the XSL style sheet corresponding to the specified graph type, which results in an SVG document. The document is then loaded into the `C_PICTURE` variable passed to the command.

The XML returned by the component's ***`GDATA_Create_from_arrays`*** method is actually a derivative of the internal XML data format used for graph types 1, 3, 4, 5, 6. You can apply the internal 4D XSL style sheets with `XSLT APPLY TRANSFORMATION` to this XML document and get the same result as with the `GRAPH` command.

Note: The command XSLT APPLY TRANSFORMATION has two "modes", BLOB and document. The XSL style sheets used by the GRAPH command, and also by this component, will only work in document mode, since they are designed to import codes from other XSL documents. BLOB mode expects a single self-contained XSL styles sheet.

Taking a closer look at the XSL style sheets reveal that many elements are actually parameterized, in the form of *XSL parameters*. For example, at line 68 of `graph1.xsl`, you will find the following code:

```
<xsl:param name="title" select="''" />
```

This implies that were we to call `XSLT SET PARAMTER` immediately before calling the `GRAPH` command, we can effectively set the graph title.

Note: XSL parameters are not Unicode compatible; their primary purpose is to pass xPath expressions, not literal text. One way to pass non-ASCII Unicode text from 4D is to pass the URI-encoded version of the text and decode it in the style sheet using the str:decode-uri function.

At this point, we have identified a few areas in which the XSL mechanism can be improved for greater control.

1. Use the same data format for all graph types, 1 through to 7.

2. Expose all customizable elements to 4D.

3. Avoid using XSL parameters to support Unicode text for graph titles, etc.

GDATA – EXTENDED GRAPH DATA FORMAT

The XML returned by the component's ***`GDATA_Create_from_arrays`*** method addresses all 3 points mentioned above. It is designed to work for all types of graph types and it takes into all the customizable elements that were previously XSL parameters, without having to depend on `XSLT SET PARAMETER.`

Note: For the extension to be fully functional, it needs to be used the modified version of the graph style sheets, included in the component's Resources folder.

CUSTOMIZING GRAPHS

There are basically two ways to customize the generated chart; either you set the parameters *before* generating the graph or modify its properties *after* the graph has been generated.

To set or get a single parameter value based on its property name, use the following method:

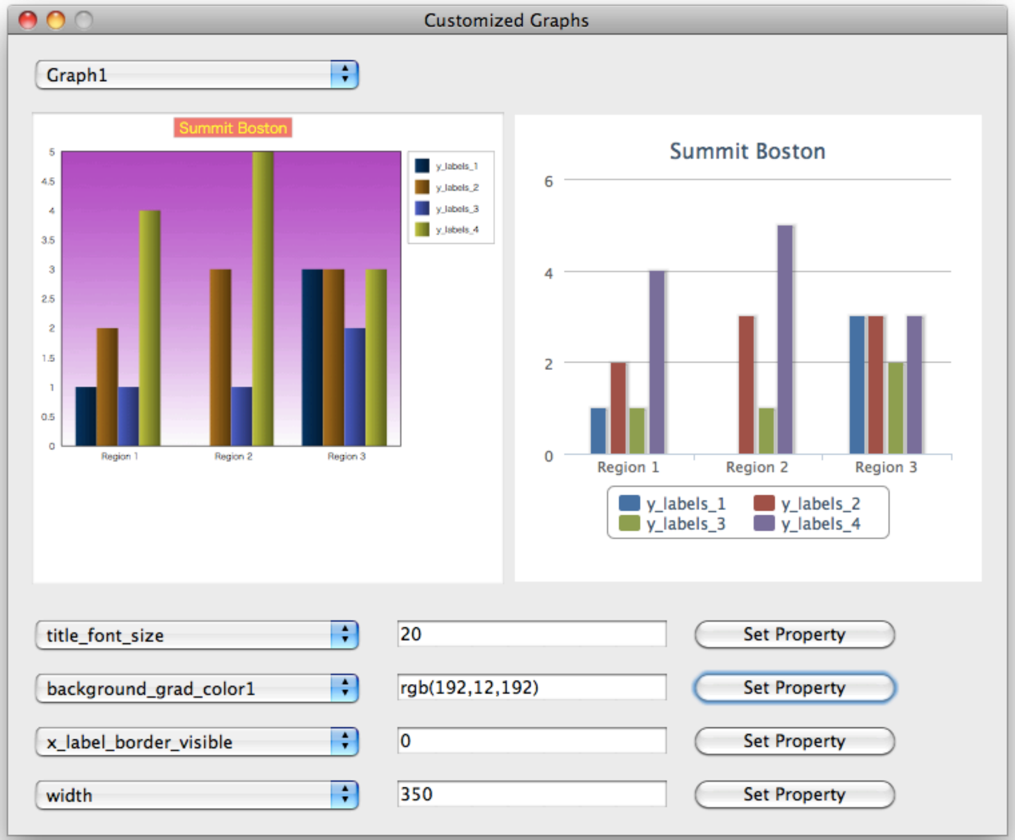
GDATA_SET_PROPERTY `(->$graphData;$propertyName;$propertyValue)`

`$propertyValue:=`***GDATA_Get_property*** `($graphData;$propertyName)`

To set or get multiple values in one call, use the following method and its counterpart:

GDATA_SET_PROPERTIES `(->$graphData;->$propertyNames;->$propertyValues)`

In all cases, the property value should be represented textually, even if their were Boolean or numeric values.



title_font_size	20	Set Property
background_grad_color1	rgb(192,12,192)	Set Property
x_label_border_visible	0	Set Property
width	350	Set Property

The following property names are recognized for SVG graphs:

```
title                          title_font_family
title_font_size                title_font_color
title_font_style               title_font_weight
title_font_decoration          title_margin
title_border_visible           title_border_margin
title_border_fill_color        title_border_fill_opacity
title_border_stroke_color      title_border_stroke_width
base_margin                    background_visible
```

bar_size	bar_size_min
bar_r	viewport_color
viewport_opacity	viewport_height
viewport_width	x_grid
x_grid_color	x_grid_width
y_grid	y_grid_color
y_grid_width	font_family
font_color	font_size
font_style	font_weight
font_decoration	legend_rect_stroke_width
legend_rect_fill_color	legend_rect_stroke_color
legend_rect_fill_opacity	legend_font_size
legend_bullet_size	background_grad_color1
background_grad_color2	background_grad_color3
background_grad_color4	grad1_color1
grad1_color2	grad2_color1
grad2_color2	grad3_color1
grad3_color2	grad4_color1
grad4_color2	legend1_color1
legend1_color2	legend2_color1
legend2_color2	legend3_color1
legend3_color2	legend4_color1
legend4_color2	legend5_color1
legend5_color2	legend6_color1
legend6_color2	legend7_color1
legend7_color2	legend8_color1
legend8_color2	x_label_border_visible
x_label_border_margin	x_label_border_fill_color
x_label_border_fill_opacity	x_label_border_stroke_color
x_label_border_stroke_width	y_label_border_visible
y_label_border_margin	y_label_border_fill_color
y_label_border_fill_opacity	y_label_border_stroke_color
y_label_border_stroke_width	label_format_number
label_format_date	label_format_time
axis_color	axis_y0_color
axis_width	line_bullet_r
line_stroke_width	line_bullet_visible
area_opacity	area_stroke_width
area_stroke_color	pie_size_min
pie_filter	pie_ellipse
pie_offset1	pie_offset2
pie_focal_x	pie_focal_y
pie_label_border_visible	pie_label_border_margin
pie_label_border_fill_color	pie_label_border_fill_opacity
pie_label_border_stroke_color	pie_label_border_stroke_width
pie_label_visible	pie_label_font_size
pie_label_format	shadow_visible
shadow_color	shadow_offset_x
shadow_offset_y	

Note: Certain properties apply only to specific types of graphs.

The following property names are recognized for HTML (*Highcharts*) graphs:

`title`	`width`
`height`	`map_zero_to_null`
`subtitle`	`y_label_title`
`fill_opacity`	`datalabels_enabled`
`legend_rect_stroke_width`	`legend_rect_fill_color`
`legend_rect_stroke_width`	`legend_visible`
`legend_shadow`	`legend_floating`
`mousetracking_enabled`	`plot_background_color`
`plot_border_width`	`plot_shadow`
`border_width`	`background_grad_color1`
`legend1_color1`	`background_grad_color2`
`legend2_color1`	`title_color`
`legend3_color1`	`title_font_weight`
`legend4_color1`	`title_vertical_align`
`legend5_color1`	`title_align`
`legend6_color1`	`title_floating`
`legend7_color1`	`title_margin`
`legend8_color1`	`title_x`
`title_y`	`title_font_size`
`title_font_style`	`title_font_decoration`
`subtitle_align`	`subtitle_vertical_align`
`subtitle_floating`	`subtitle_x`
`subtitle_y`	`subtitle_font_weight`
`subtitle_color`	`subtitle_font_size`
`subtitle_font_style`	`subtitle_font_decoration`
`subtitle_font_family`	

To set the background image of an SVG graph, use these dedicated methods:

GDATA_SET_BACKGROUND_IMAGE `(->$graphData;$image)`

Note: To clear the background image, simply pass an empty picture variable.

You can retrieve the image using its counterpart method:

`$image:=`***GDATA_Get_background_image*** `($graphData)`

Note: This may not be identical to the original image; the component converts the image to PNG format, unless it is already SVG.

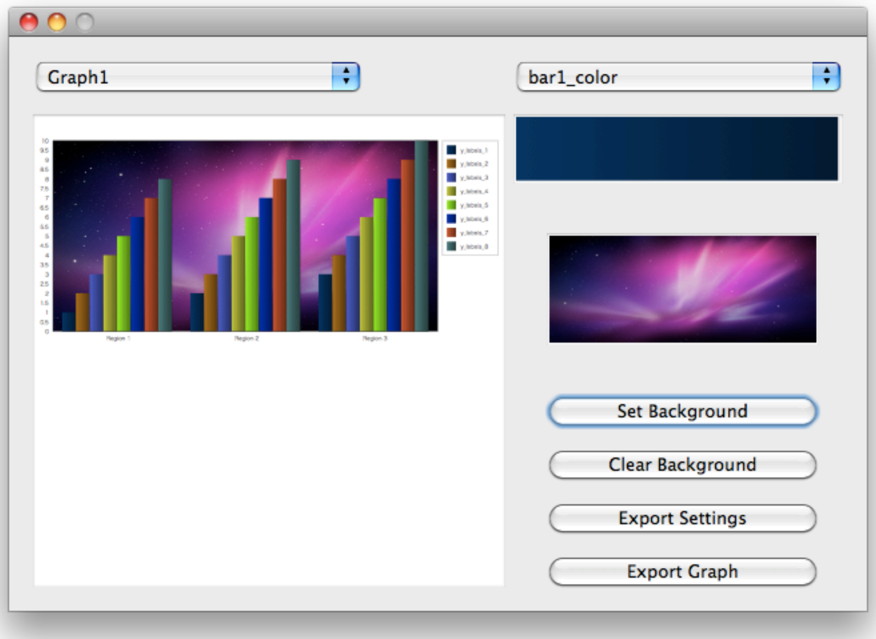
The customized properties can be exported, independent of the graph data. You can store and reuse them later, for example, to apply the same settings to a different data set.

`$settings:=`***GDATA_Get_settings*** `($graphData)`

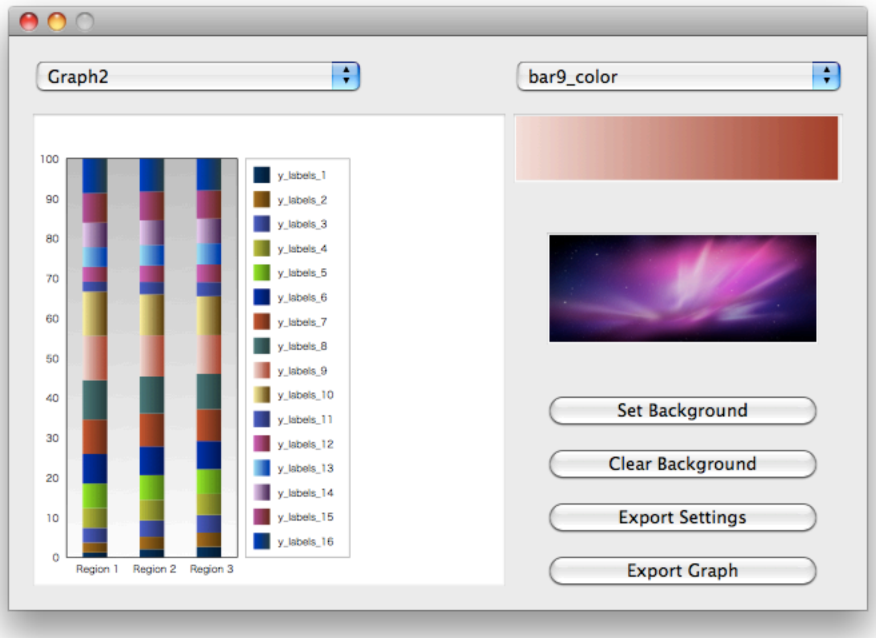
`$settings:=`***GDATA_SET_SETTINGS*** `(->$graphData;$settings)`

The content of the settings BLOB is actually an XML document.

Note: If you do not intend to use the background image, make sure you clear the image data before exporting the settings XML. The document can be substantially larger if it contains the BASE64 encoded image data.



Finally, the component doubles the number of data sets from 8 to 16.



SVG CODING TIPS

Drawing borders surrounding a piece of text can be quite tricky in SVG. This is because, in SVG, `text` and `rect` (or `polyline`) are basically independent objects. You cannot depend on the rendering engine to automatically draw the lines at the right size and position relative to the text. Rather, you need to calculate them yourself.

The component deals with this problem by drawing the text offline to determine the height, width and baseline for a given font name, style, size, decoration and weight. The exact rendering result on other SVG engines may be slightly different, but in general, the metrics should be of an acceptable level.

When you embed an image file (PNG, BMP, JPG, or SVG) into an SVG document, you might have to apply some calculations for the image to appear at the right location and size. More specifically, if you set the `width` and `height` attributes of the `image` element to that of the original image, you should also have a `transform` attribute with translates and scales the image so that it fits neatly inside the bounding rectangle. The formula for this is as follows:

```
sx=width/original_width
sy=height/original_height

tx=-((sx*x)-x)
ty=-((sy*y)-y)

transform="translate(tx,ty) scale(sx,sy)"
```

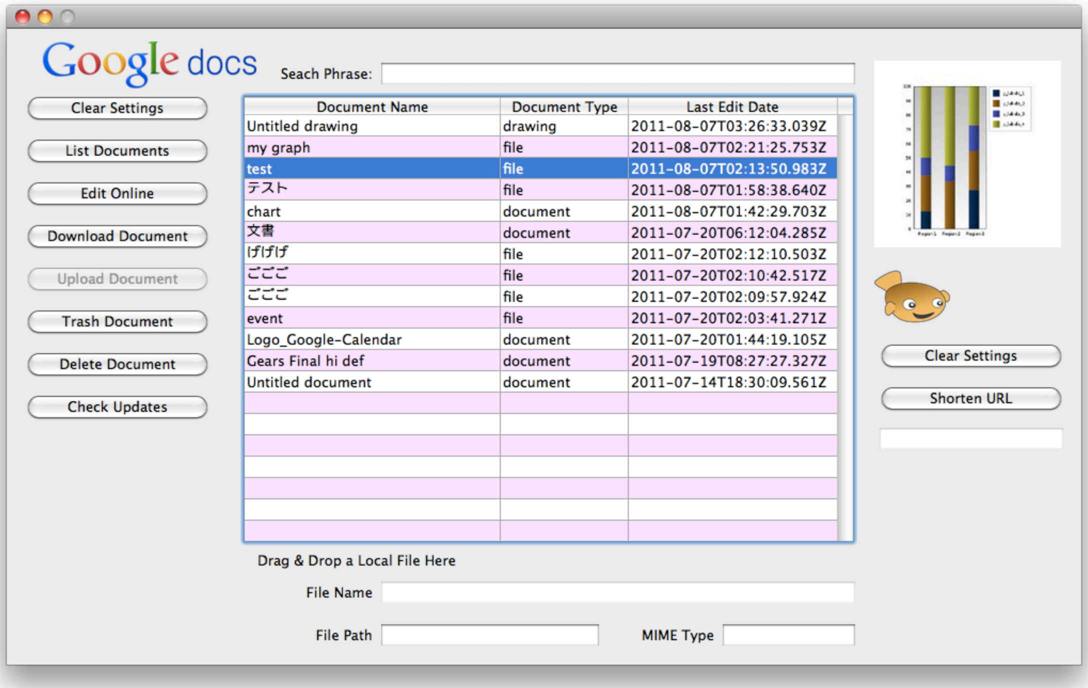
LISTBOX CODING TIPS

The component depends on `LISTBOX GET ARRAYS` to resolve the data source. As of v119/v12.2, this command apparently fails to resolve columns that were programmatically added. You might have to move the method from component to host in such cases.

Most form objects, including the listbox, can be bound to dynamic variables, or form variables, as of v12. However, if you intend to populate the listbox using the SQL INTO LISTBOX keyword, you must bind the object to a process variable. The code may work interpreted but fail compiled, if the object has no variable assigned in the Form Editor.

CHARTS ON THE CLOUD

SVG images can be uploaded to *Google documents*. Putting our charts online makes their content accessible from anywhere on any computer where you can access Google.



Seach Phrase:

Document Name	Document Type	Last Edit Date
Untitled drawing	drawing	2011-08-07T03:26:33.039Z
my graph	file	2011-08-07T02:21:25.753Z
test	file	2011-08-07T02:13:50.983Z
テスト	file	2011-08-07T01:58:38.640Z
chart	document	2011-08-07T01:42:29.703Z
文書	document	2011-07-20T06:12:04.285Z
げげげ	file	2011-07-20T02:12:10.503Z
ごごご	file	2011-07-20T02:10:42.517Z
ごごご	file	2011-07-20T02:09:57.924Z
event	file	2011-07-20T02:03:41.271Z
Logo_Google-Calendar	document	2011-07-20T01:44:19.105Z
Gears Final hi def	document	2011-07-19T08:27:27.327Z
Untitled document	document	2011-07-14T18:30:09.561Z

File Name

File Path

MIME Type

Note: Google doesn't support SVG to Drawings conversion; only the reverse is supported. As such, SVG images are simply uploaded as files without any conversion. Alternatively you can upload the images as GIF, PNG, JPG or PDF, in which case the image will be converted to a Google Document. For full list of supported conversions, go to:

http://code.google.com/intl/en-us/apis/documents/docs/3.0/developers_guide_protocol.html - MetaDataFeed

WEBAREA CODING TIPS

Note: The information presented in this section is based on anecdotal evidence observed using the latest version (Mac OS X Snow Leopard, 4D 11.9, 4D 12.2) at the point of editing this document. The behavior may be different on subsequent versions.

The component takes advantage of sophisticated JavaScript libraries such as *Highcharts*, *Raphaël* and *jQuery*. Apparently the Mac OS Web Area can lead to problems after displaying such content. Here are some tips to avoid running into trouble.

Don't run JavaScript code in the Main Process

Clicking the green button located at top left corner of the Form Editor lets you run the form in the Main Process. Avoid doing this if possible, *if the Web Area runs any timer-driven JavaScript code* (ease-in/ease-out, etc.). Once you run such code in the Main process, the code may cease to work on all other process, or even the Main Process after reopening the database. Moreover, once you close the form, holding a menu open for a couple of seconds will crash the application. You would have to restart the application to fix this.

Don't reopen an application after running JavaScript code

One can restart an application (database) without having to re-launch the 4D application itself, by selecting *Recent Databases* from the *File* menu, or *Restart Interpreted/Restart Compiled* from the *Run* menu. Avoid doing this if possible, *if the Web Area runs any timer-driven JavaScript code.* Once you perform this operation, holding a menu open for a couple of seconds will crash the application. You might want to simply quit and restart the application just to be on the safe side.

Don't release a process that has ran JavaScript code

If a form that contains a web Area is unloaded, it may lead to some memory problems, *if the Web Area runs any timer-driven JavaScript code.* Once you perform this operation, holding a menu open for a couple of seconds will crash the application. You might want to reuse the process, by calling `HIDE PROCESS` when the `On Close Box` form event, `CANCEL` and `ACCEPT` shortcuts are triggered.

Consider how you update the page URL

The standard way to update the page URL of a displayed Web Area is to run the following code:

```
WA OPEN URL(*;"Chart";$chartDataPath)
```

However, certain JavaScript animations (the swipe animation used by `Highcharts`, in particular) may not run if the page was loaded this way. You need to update the URL variable, as shown below:

```
Chart_url:=$chartDataURL
```

Note that the former is compatible with paths in HFS file system format or `file://` format, but the latter only recognizes the path if passed as an URL.

Keep an eye on memory leaks

If you run the *Console* application alongside 4D, you cannot but notice how many *"autoreleased with no pool in place"* warnings are thrown to the system just by having a Web Area running complicated JavaScript code. They may not cause any immediate harm, but should be taken into account if your application depends heavily on such features. In some cases it might be better to separate the JavaScript application from the main 4D application.

CUSTOMIZING THE JAVASCRIPT GRAPH

An HTML graph can also be customized prior to generation by changing the `GDATA` parameters. In addition, almost any aspect of the graph can be changed on the fly, thanks to JavaScript.

JAVASCRIPT INJECTION

For example, to change the title and subtitle of a *Highcharts* graph displayed in a Web Area, run the code below:

```
$result:=WA Execute JavaScript(*;$webAreaName;"chart.setTitle( {text:
"+JSON_Encode ($title)+"}, {text: "+JSON_Encode ($subTitle)+"} )")
```

Note: It is always good practice to escape any textual data that are provided by the user. The code above may seem to work without the encode method in place, but if the title contains double quotes, apostrophes, or other characters that should be escaped, the code will fail. To learn more about the JSON format, visit the web site.

http://www.json.org/

In this example, we are calling directly a method of the `chart` global variable, which is a *Highcharts* `chart` object. Alternatively, we could wrap this method in the JavaScript source, and expose it as a function.

```
function setTitle(title, subtitle){
      chart.setTitle(title, subtitle);
}
```

The advantage of this approach is that regardless of the underlying drawing library (because the component is not limited to *Highcharts*) we can use the same function name in our 4D code to get the same effect.

```
$result:=WA Execute JavaScript(*;$webAreaName;"setTitle( {text: "+JSON_Encode
($title)+"}, {text: "+JSON_Encode ($subTitle)+"} )")
```

EXPORT THE GRAPH AS PICTURE

Highcharts draws an SVG or VML image depending on the running HTML rendering engine (the browser, or in our case the Web Area), as not all browsers are capable of displaying SVG. It is also possible to get the XML source of the SVG image on any platform by calling the method `chart.getSVG()`.

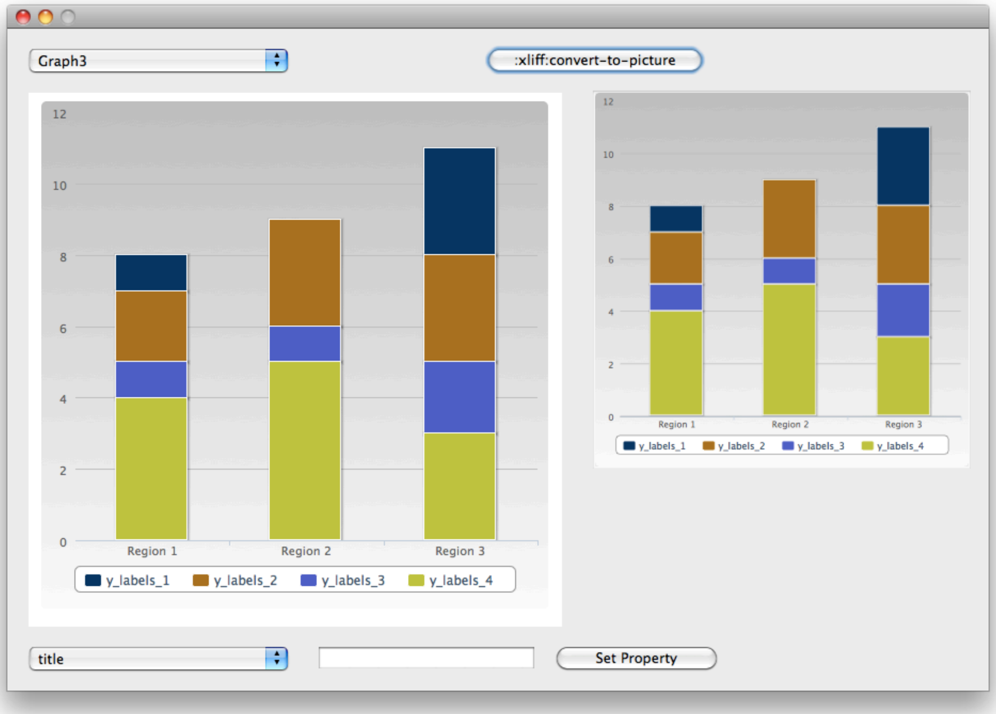
```
$svgText:=WA Execute JavaScript(*;$webAreaName;"chart.getSVG()")
```

Note: You need to import the `exporting.js` *file, which is included in the package.*

Highcharts is primarily designed for Web Applications, not desktop application like 4D. This is especially true in the way the export mechanism works. First, an SVG image is produced on the client side (browser). Then, the data is sent over the Internet to the *Highcharts* Web Services, which performs the image conversion to PNG, PDF or JPG format. It is not really necessary to connect to this server from 4D, which natively supports such conversions. We can simply retrieve the SVG and convert it off line using the native picture commands.

Note: SVG to PDF conversion is only possible with the Mac version of 4D.

In general, SVG is the best choice both in terms of file size and image quality. If the graph needs to be viewed by applications that do not support SVG, then you might have to convert the image to other formats such as PNG. Keep in mind that the 4D command `CONVERT PICTURE` rasterizes the image according to the current picture size. A picture that is sharply displayed on a computer display may not look so fine when printed. If that is the case, then enlarge the SVG with the `TRANSFORM PICTURE` command, before converting it to PNG. In general, a picture magnified by a factor of 5 should look as finely defined in print as it did on screen.



SUMMARY

This section introduced a v11/v12 compatible component that extends the `GRAPH` command of 4D, capable of producing SVG or HTML+JavaScript (*Highcharts*) graphs that are highly customizable. The component separated the style element from the data and uses XSLT to combine the two. The component follows the conventional `GRAPH` syntax to simplify the process of migration. It supports direct graph creation from a listbox data source. It is possible to add more graph types by adding XSLT style sheets.

INTRODUCTION - 2

4D Write has been for many years the preferred option for creating reports and letters that are data driven. The plugin is capable of managing styled text and supports inline/background images as well as embedded expressions.

One common use of 4D Write is to create HTML e-mail templates with field references of the database, in order to generate and send personalized messages systematically.

WHERE THE PLUGIN COMMAND MIGHT FAIL YOU

There are, however, certain known limitations to the plugin as of 4D v12.

No support of Multi Style Text references

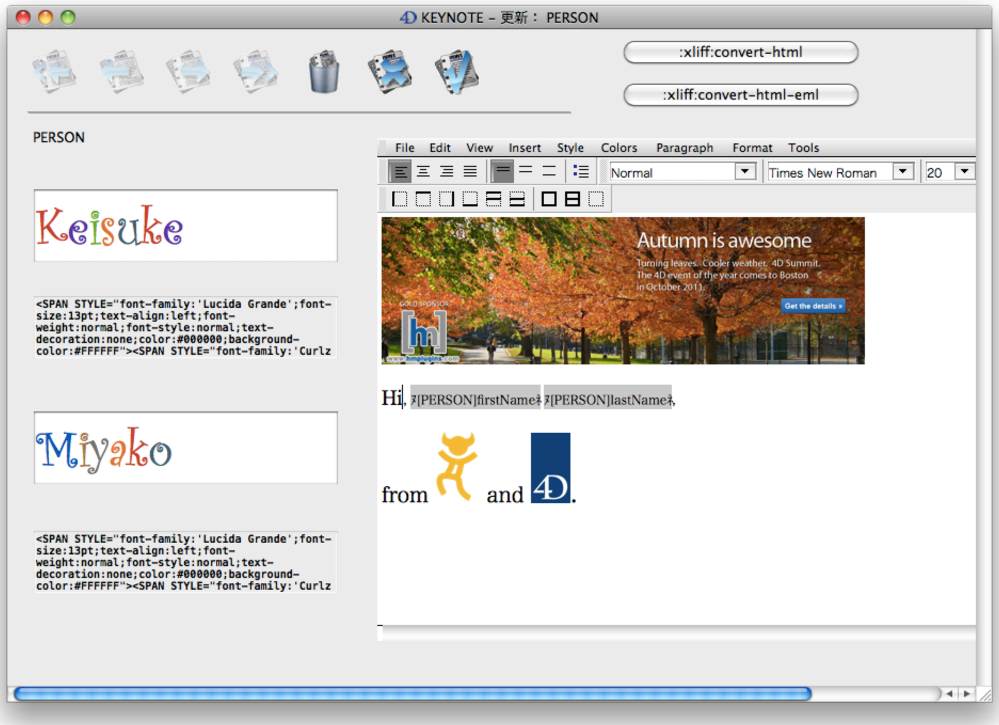
Although it is now possible to have multi style text as 4D fields and variables, when referenced from a 4D Write document, both appear as raw HTML snippets, not rendered in the document as one might expect.

Limited HTML conversion capability

4D Write supports export in HTML 3.2, HTML 4, RTF and *Word* formats, but the integrity of such converted documents are somewhat limited. In particular, complicated documents tend to look very different to the original when converted to HTML.

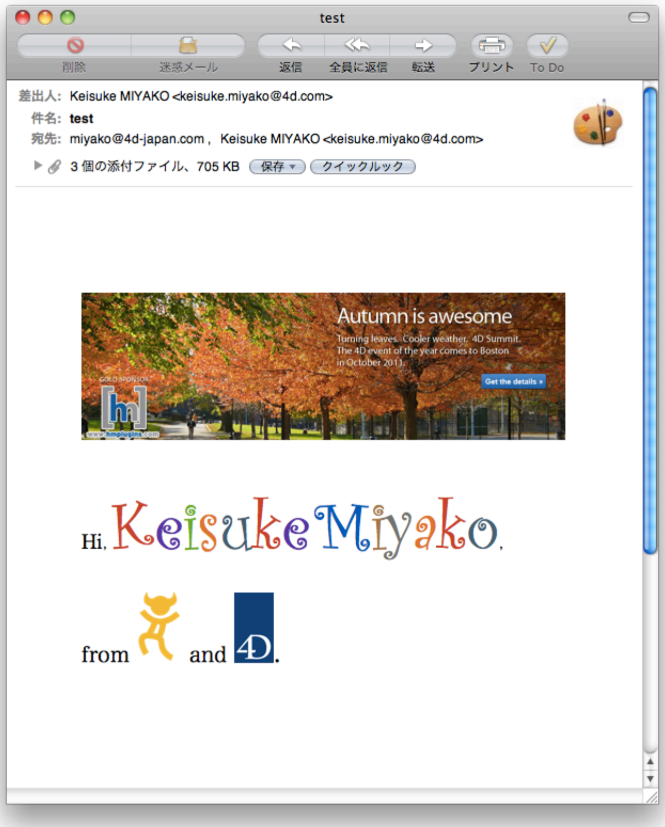
INTRODUCING THE WRITE COMPONENT

This component was developed with the points mentioned above specifically in mind. It offers an easy to use API that can convert 4D Write documents to HTML, which looks exactly identical to the original document. Also available is an option to embed database references and expressions in that converted document, in either multi style or plain text format. In addition, it has a convenience function to create MIME formatted e-mails that are HTML based and includes images.


4D KEYNOTE – 更新： PERSON
:xliff:convert-html
:xliff:convert-html-eml
PERSON
File Edit View Insert Style Colors Paragraph Format Tools
Normal
Times New Roman
20
Keisuke
Miyako
Autumn is awesome
Hi, [PERSON]firstName [PERSON]lastName.
from and .

```
<SPAN STYLE="font-family:'Lucida Grande';font-
size:13pt;text-align:left;font-
weight:normal;font-style:normal;text-
decoration:none;color:#000000;background-
color:#FFFFFF"><SPAN STYLE="font-family:'Curlz
```

```
<SPAN STYLE="font-family:'Lucida Grande';font-
size:13pt;text-align:left;font-
weight:normal;font-style:normal;text-
decoration:none;color:#000000;background-
color:#FFFFFF"><SPAN STYLE="font-family:'Curlz
```



差出人: Keisuke MIYAKO <keisuke.miyako@4d.com>
件名: **test**
宛先: miyako@4d-japan.com， Keisuke MIYAKO <keisuke.miyako@4d.com>
3 個の添付ファイル、705 KB

USING THE WRITE COMPONENT

As with the other component, the idea is to first create and XML sources with all the information needed to create the end product (in this case an HTML or MIME replication of the 4D Write document) and call `XSLT APPLY TRANFORMATION` on the XML. Field references and embedded expressions are converted to 4D HTML tags, so that they can later be replaced by multi style or plain text values using `PROCESS HTML TAGS`. To convert a 4D Write document to an XML data source, use the following component method:

```
$documentData:=WDATA_Create_from_document ($filePath)
```

To convert this XML to HTML, use either of the following methods:

```
$writeData:=WRITE_Create_HTM ($documentData;"HTML")
```

```
$writeData:=WRITE_Create_HTM ($documentData;"XHTML")
```

Note: The first syntax generates an XHTML document with the DOCTYPE included. This format is preferable if the objective is to view the document in a Web Browser or the 4D Web Area form object, since without the DOCTYPE, Internet Explorer will default to the Quirks mode, which is the incorrect HTML/CSS rendering of IE7. The DOCTYPE tells IE9 to render the page according to HTML5 standards. On the other hand, if you intend to post process the XHTML using XSLT APPLY TRANSFORMATION again, use the second syntax since the library may not work with the XHTML DOCTYPE.

The result is a text object of the HTML source code, which can be saved to disk or opened in a web browser.

To apply a specific directive option to the conversion, use `XSLT SET PARAMETER` before running the conversion. Here is the list of the parameters that are applicable.

```
XSLT SET PARAMETER("text_reference";"'4d_html'")
```

Possible values: '4d_html', '4d_text', '4d_htmlvar', '4d_var'

Defines which type of 4D HTML TAGS to use for field references and embedded expressions. `4DHTML` and `4DTEXT` requires version 12.2 or later. If the text to be inserted is preprocessed HTML, for example v12 multi style text, use `4DHTML` or `4DHTMLVAR`. If the text to be inserted is plain text, use `4DTEXT` or `4DVAR`.

```
XSLT SET PARAMETER("text_page";"0")
```

Possible values: 0 or a positive whole number.

Defines which page to export. Pass 0 to export all pages.

```
XSLT SET PARAMETER("content_editable";"'true'")
```

Possible values: 'true' or 'false' (default).

Defines whether the HTML body should be editable on the browser.

```
XSLT SET PARAMETER("include_jquery ";"'true'")
```

Possible values: 'true' or 'false' (default).

Defines whether the *jQuery* source should be included in the HTML. When *jQuery* is included, you can take advantage of the library to perform various tasks in the document context. For example, you can display the HTML in a 4D Web Area and interact with its content by means of `WA EXECUTE JAVASCRIPT FUNCTION`.

Note: The component makes best effort to replicate the original 4D Write document in HTML format, however, one notable exception is that it ignores all tab stops in line, as they do not have a strict equivalent in HTML. An enhanced style sheet that converts margin-based divisions to HTML table may be provided in the future.

To convert the 4D Write document to MIME, that is, HTML format ready to be sent as an e-mail, use the same method to create the XML source but call the following component instead to prepare for e-mail:

```
$template:=WRITE_MIME_Create_template ($documentData;"XHTML")
```

The returned XML is slightly different to the one we used for regular HTML in that it uses `cid` references for images instead of data URI's, and that the included references are already processed in the current record's context. This is because images need to be handled differently in MIME and the HTML and the content needs to be finalized for BASE64 encoding.

The component allows you to put the final touch on the e-mail, by setting the sender, recipient(s), attachment(s) and any other SMTP header/value pairs you might want to define. To this, follow the example given below:

```
ARRAY TEXT($headerNames;4)

$headerNames{1}:="From"
$headerNames{2}:="Subject"
$headerNames{3}:="To"
$headerNames{4}:="To"

ARRAY TEXT($headerValues;4)

$from:="Keisuke MIYAKO <keisuke.miyako@4d.com>"
$subject:="test"
$To1:="miyako@4d-japan.com"
$To2:="Keisuke MIYAKO <keisuke.miyako@4d.com>"

$headerValues{1}:=$from
$headerValues{2}:=$subject
$headerValues{3}:=$To1
$headerValues{4}:=$To2

ARRAY TEXT($attachmentPaths;0)

WRITE_MIME_SET_PARAMETER (->$template;->$headerNames;\
->$headerValues;->$attachmentPaths)
```

Once all the parameters are set, you can call the following component method to convert it into MIME:

```
$mimeData:=WRITE_MIME_Get_data ($template)
```

This is the raw data you send with the `DATA` command during the SMTP protocol.

Note: This MIME data format is suitable for generic low-level SMTP implementations like Gmail-OAuth, but not high-level implementations like 4D Internet Commands. If you wish to prepare HTML body for 4DIC, simply use the previous method used to create HTML for the Web Browser.

SUMMARY

This section introduced a v11/v12 compatible component, which can be used in conjunction with the 4D Write Plugin to better support HTML output. In particular, it allows you to convert a 4D Write document to standard HTML document that looks pretty much identical to the original. The HTML retains its capacity as a report template, as field references and embedded expressions are converted to 4DHTML tags. 4D v12 multi style text is directly inserted in the HTML content. It also supports the creation of MIME HTML data with images embedded.