HTML Report Library

Table of Contents

1 Delphi HTML Report Library

1.1 Introduction

HRL is a template-based reporting library for Delphi, designed to generate reports using databases and XML. Templates are simple HTML files containing special tags, and output is pure HTML suitable for displaying in any browser and sending by mail (all of the graphics are directly integrated into the text, so when sending mail you don't have to attach pictures separately) . HRL can be used in client applications to generate and view report inside the program, and in server applications or Web-servers.

As a template language HRL uses widely known **Mustache** with some extensions.

Build-in powerful **scripting** and **expression** evaluation engine allow to implement complex data processing algorithms and encapsulate all logic inside report.

Library supports all versions of Delphi from **Delphi 5** to **Delphi 11, Lazarus**, and also supports Unicode for old non-unicode versions of Delphi using widestring.

HRL supports all FMX platforms – **Win32/64, OSX, iOS, Android, Linux** (printing is available only in Win/ OSX).

HRL fully supports **Right-to-Left languages**, including order of table columns and location of chart data.

HRL supports major data-access components (**FireDac, IBX, UIB, DOA, Unidac**) and formats (JSON, XML). Adapters for other libraries could be implemented very simply and takes not more than 10 minutes.

1.2 Overview

Report template is a standard HTML file and could contain any of HTML 4.01 tags and most CSS3 properties. To generate content HRL uses special tags such as:

DATAPACKET - to generate table data

CHART - to generate charts

BARCODE - to generate barcodes.

QRCODE - to generate QR codes

report-objects - objects generated inside report

It is possible to register additional custom tags.

After first processing stage, special tags content is replaces with generated HTML, so the output is always pure HTML and can be displayed in any browser.

1.3 Hello World!

Simplest report looks like

```
Hello world!
```

Enclosing tags (<html>) are not necessary.

1.4 Templates

HRL uses **Mustache** template syntax. Context variables are set as XML objects, where attributes are treated as variables and nodes as sections

For example, passing a object

```
<animals>
 <animal name="Dog" color="yellow"/>
 <animal name="Bird" color="green"/>
</animals>
```

To the template

```
<html>
{{#animals}}
 <p style="color: {{color}}">{{name}}</p>
{{/animals}}
</html>
```

results

dog

bird

Document-level templates are processed after special tags, so data generated during tags processing (for example, from the database using DATAPACKET tag) could be used anywhere in the report.

1.4.1 Template processing exclusion

Some parts of the document (for example internal DATAPACKET templates) should not be processed at the document parsing stage. For such cases, there is a special exception of **mustache** syntax**:** tags beginning with **<template-,** and **<script** are not processed.

1.4.2 Pseudo-sections and special variables

-first section is processed only at first iteration of loop

-last section is processed only at last iteration of loop

-index loop counter starting from 1

. - reference to the current context (used in cycles)

.. reference to the context of the upper level (used in cycles)

@ var - nested variable - {{ {{var}} }}

context - main report context variable

1.4.3 Example of logical sections

For example, we have a list of employees, some of whom have e-mails. Those names should be displayed as a link, and the rest – as a plain text. For this, data must contain a flag field with value of true or false. For example:

```
<list>
 <employee name="Willis" email="bruce@hollywood.com" hasemail="true"/>
 <employee name="Stallone" />
</list>
```

template:

```
{{#employee}}

 {{#hasemail}}
   <a href="{{email}}">{{name}}</a><br/>
 {{/hasemail}}

 {{^hasemail}}
  {{name}}<br/>
 {{/hasemail}}

{{/employee}}
```

Result:

Willis
Stallone

Other type of logical section is named section. For example we want to process following XML:

```
<main>
<section1>
…
</section1>
<section2>
…
</section2>
<section2>
…
</section2>
</main>
```

To apply template only to tags named section2 (and exclude section1) add ? before the section name:

```
{{#main}}
{{#?section2}}
…
{{/?section2}}
{{/main}}
```

1.4.4 XPATH sections

Sections can contain XPATH expressions. Section name is treated as XPATH expression if it contais slash (/) symbol.

F.E.

{{#MyVar/Customers[state='active']}}

...

{{/MyVar/Customers[state='active']}}

1.4.5 Template functions

Sections can be either functions. If a name of the section coincides with the registered function, the function applies to the entire contents of the section. For example:

```
{{#upper}} {{name}} {{/upper}}
```

will output a variable name in uppercase.

Functions are registered globally for the entire project:

```
type
 THtTemplateFunction =function(s: hstring): hstring;
procedure RegisterHtTemplateFunction(const Name: string; TF: THtTemplateFunction);
```

Template function example

Consider we need to display each letter of some text in a separate box.

First we register the function:

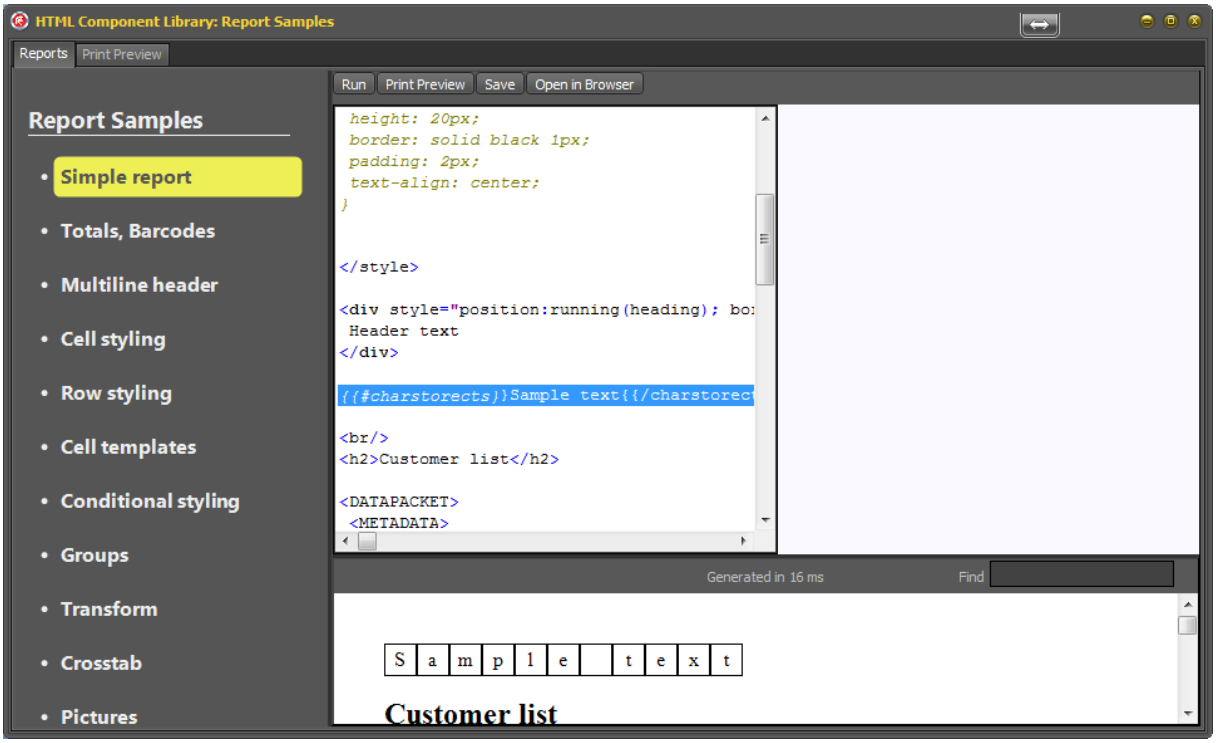
```
function CharstoRects(const s: hstring): hstring;
var i: integer;
begin
 Result:='';
 for i:=1 to length(s) do
  Result:=Result+'<div class="charrect">'+s[i]+'</div>';
end;
```

And set styles in report

```
.charrect {
 display: inline-block;
 width: 20px;
 height: 20px;
 border: solid black 1px;
 padding: 2px;
 text-align: center;
}

{{#charstorects}}Sample text{{/charstorects}}
```

Result:



Built-in functions:

uper – upper case

Lower – lower case

Trim – trim string

Include – include file

Charstorects -see above

Padleft - has two parameters – count of chars and text, add count spaces to left. Example: {{#padleft}} 4 text{/padleft}

Date – Print current date using provided format

formatmoney - format value using # ### ### ##0.## format

formatdate - parameters: value|date format

eq - parameters A=B:C - returns C if A=B

1.4.6 Value formats

Template variables can be formatted using number or date formats. Format is passed in brackets after variable name. For example:

{{PRICE(# ##0.00)}}

{{STARTDATE(dd.mm.yyyy}}}}

Format can be applied only to values containing numbers, dates and timestamps.

1.4.7 Escaping

To display variable content without escaping HTML use three brackets or & prefix:

{{{HTMLFIELD}}}

{{&HTMLFIELD}}

1.5 Pagination

To control pagination use CSS **page-break-before, page-break-after** and **page-break-inside** properties.

These properties are valid only for block elements.

For example, if we want to start new page before each **h2** element:

```
h2 {page-break-before: always}
```

To insert a page break at a random location there is special element defined in standard report styles: pagebreak.

To start a new page just put it anywhere.

```
<pagebreak/>
```

pagebreak tag can also contain page orientation attribute:

```
<pagebreak orientation="landscape"/>
```

1.5.1 Header and Footer

To display headers and footers use CSS property position = running (heading / footer)

Standard report style declares two classes: .pageheader and . pagefooter allowing display element in a header or footer. For example

```
<div class="pageheader">
 Fish world
</div>
```

To display a page number use CSS property **content: counter (page),** for displaying page count use **content:counter(pagecount).**

Standard report style declare **.page** and .pagecount classes for displaying page numbers and count. For example

```
<div class="pageheader">
 Fish world. Page: <span class=”page”/>
</div>
```

1.5.2 Page size

To set page size inside report use **@page** CSS rule in <style> section.

Examples:

@page {size: A4 landscape}

@page {size: 20cm 10cm}

@page {size: 5.2in 3in; margin: 10mm 10mm}

1.6 Tables

Tables are displayed using special tag DATAPACKET.

Datapacket has following format:

```
<DATAPACKET>
 <METADATA>
  <FIELDS>
   <FIELD>
    …
   </FIELDS>
 <SQL>
```

```
  …
 </SQL>
 <SCRIPT>
 ...
 <SCRIPT>
</METADATA>
 <ROWDATA>
 …
 </ROWDATA>
</DATAPACKET>
```

All **DATAPACKET** nodes are optional. **SQL** and **ROWDATA** are mutually exclusive, only one should exist.

When processing **SQL** tag, query is executed and result converted to XML is substituted to a **ROWDATA** tag of **DATAPACKET.**

Example of simple DATAPACKET:

```
<DATAPACKET>
 <METADATA>
  <FIELDS>
    <FIELD name="NAME"/>
    <FIELD name="COLOR"/>
  </FIELDS>
 </METADATA>
<ROWDATA>
  <R NAME="Dog" COLOR="brown"/>
  <R NAME="Bird" COLOR=”green"/>
 </ROWDATA>
</DATAPACKET>
```

Result:

NAME	COLOR
Dog	Brown
Bird	Green

Example of DATAPACKET with SQL

```
<DATAPACKET>
 <METADATA>
  <FIELDS>
   <FIELD name="CUSTNO"/>
   <FIELD name="ADDR1" />
   <FIELD name="COMPANY" />
   <FIELD name="CITY" />
   <FIELD name="STATE" />
   <FIELD name="PHONE" />
```

```
   <FIELD name="CONTACT" />
  </FIELDS>
  <SQL>
   select * from customer
  </SQL>
 </METADATA>
</DATAPACKET>
```

Result:

CUSTNO	ADDR1	COMPANY	CITY	STATE	PHONE	CONTACT
1221	4-976 Sugarloaf Hwy	Dive ShoppeKauai	KauaiKapaa	HI	808-555-0269	Erica Norman
1231	PO Box Z-547	Unisco	Freeport		809-555-3915	George Weathers
1351	1Neptune Lane	Sight Diver	Kato Paphos		357-6-876708	Phyllis Spooner

For SQL sources, field names must be in upper case.

1.6.1 Table templates: General information .

All parts of a table, such as table itself, column header, row and cell can be specified by templates. All templates looks like:

```
<template-…>..<template-...>
```

All templates have **custom** (= true / false) attribute determines whether the template set only internal content of element or replaces full element.
For example, a template for a cell with value “5”

```
<template-cell>
 <i>{{value}}</i>
</template-cell>
```

Produces:

```
<td (attributes…)> <i>5</i></td>
```

But template

```
<template-cell custom="true">
 <i>{{value}}</i>
</template-cell>
```

Produces:

```
<i>5<i>
```

1.6.2 Table Header

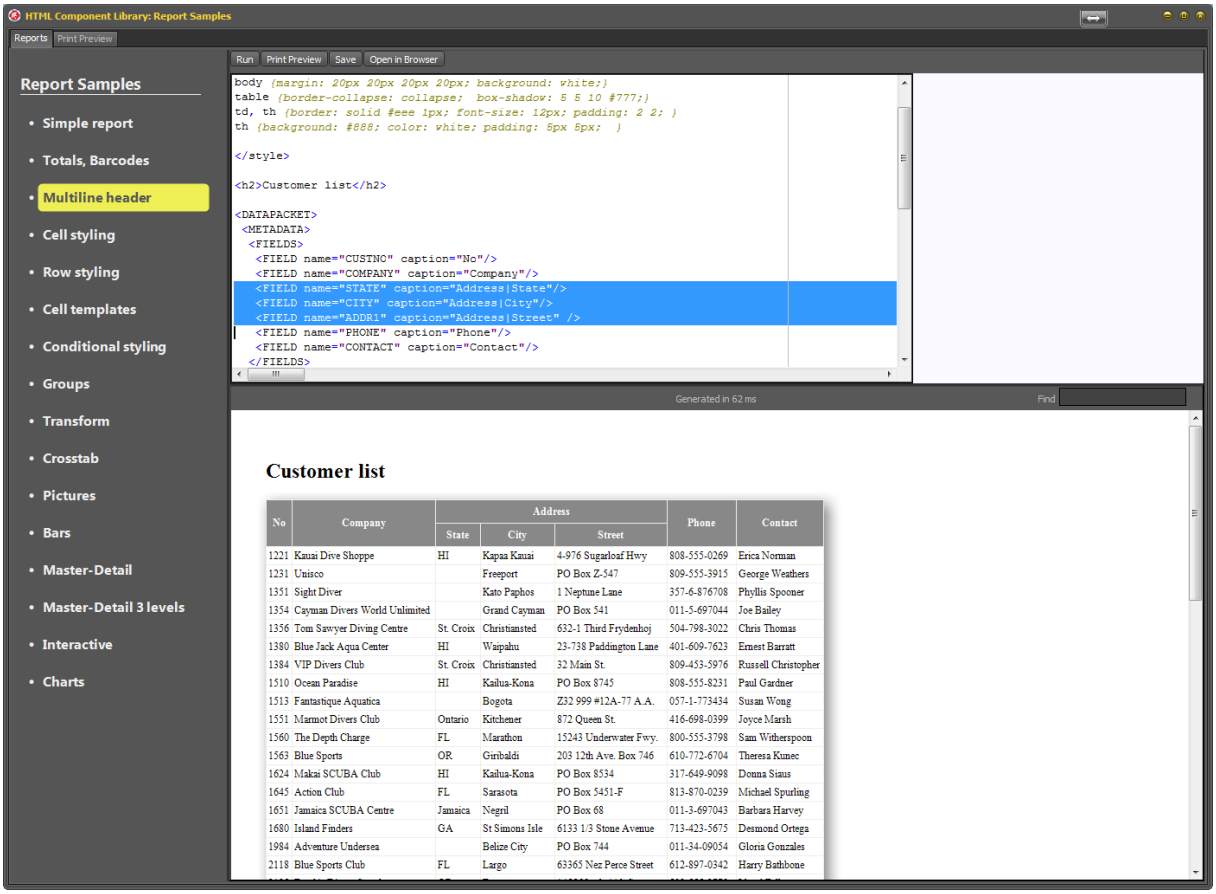
Table header is generated from **caption** attribute of fields. If **caption** is not specified, field name is used.
To hide header, set METADATA.**header attribute to** "false".
Complex header could be specified using a **template-header** template
Default template for header is
`<th {{{attributes}}}>{{value}}</th>`

1.6.3 Multiline headers

Multiline headers are specified using | as caption parts separator.

If several fields starts with one text before the separator, this text is used as a common header for those fields. For example

```
<FIELD name="STATE" caption="Address|State"/>
<FIELD name="CITY" caption="Address|City"/>
<FIELD name="ADDR1" caption="Address|Street />
```



No	Company	Address			Phone	Contact
		State	City	Street		
1221	Kauai Dive Shoppe	HI	Kapaa Kauai	4-976 Sugarloaf Hwy	808-555-0269	Erica Norman
1231	Unisco		Freeport	PO Box Z-547	809-555-3915	George Weathers
1351	Sight Diver		Kato Paphos	1 Neptune Lane	357-6-876708	Phyllis Spooner
1354	Cayman Divers World Unlimited		Grand Cayman	PO Box 541	011-5-697044	Joe Bailey
1356	Tom Sawyer Diving Centre	St. Croix	Christiansted	632-1 Third Frydenhoj	504-798-3022	Chris Thomas
1380	Blue Jack Aqua Center	HI	Waipahu	23-738 Paddington Lane	401-609-7623	Ernest Barratt
1384	VIP Divers Club	St. Croix	Christiansted	32 Main St.	809-453-5976	Russell Christopher
1510	Ocean Paradise	HI	Kailua-Kona	PO Box 8745	808-555-8231	Paul Gardner
1513	Fantastique Aquatica		Bogota	Z32 999 #12A-77 A.A.	057-1-773434	Susan Wong
1551	Marmot Divers Club	Ontario	Kitchener	872 Queen St.	416-698-0399	Joyce Marsh
1560	The Depth Charge	FL	Marathon	15243 Underwater Fwy.	800-555-3798	Sam Witherspoon
1563	Blue Sports	OR	Giribaldi	203 12th Ave. Box 746	610-772-6704	Theresa Kunec
1624	Makai SCUBA Club	HI	Kailua-Kona	PO Box 8534	317-649-9098	Donna Siaus
1645	Action Club	FL	Sarasota	PO Box 5451-F	813-870-0239	Michael Spurling
1651	Jamaica SCUBA Centre	Jamaica	Negril	PO Box 68	011-3-697043	Barbara Harvey
1680	Island Finders	GA	St Simons Isle	6133 1/3 Stone Avenue	713-423-5675	Desmond Ortega
1984	Adventure Undersea		Belize City	PO Box 744	011-34-09054	Gloria Gonzales
2118	Blue Sports Club	FL	Largo	63365 Nez Perce Street	612-897-0342	Harry Bathbone

1.6.4 Formatting fields

To specify format of a fields use “format” attribute of a **FIELD tag.**
For numeric fields, format are same as for Delphi **FormatFloat function,** for date fields – as for **FormatDateTime function.**
By default, numeric fields uses **# ### ### ##0.##** format.

Alignment of fields could be specified by **align** (left, right, center) attribute. Right is default alignment for numeric fields and left for all others.

“empty” attribute could be used to set default value for empty cells.

Complex formatting of a field could be perfomed by a **template-cell template.**
Default cell template looks like

```
<td {{{attributes}}}>{{value}}</ td>
```

In the template you can use variables and cycle pseudo-sections, for example, you can make text bold only in first row:

```
{{#-first}}<b>{{/-first}}
 {{value}}
{{#-first}}</b>{{/-first}}
```

Also field can be formatted using **template** attribute. In this attribute **#** is substituted by current cell value and **@FIELD@** by field value.

To use HTML formatting in a cell value, set **FIELD.html** attribute to true. Field value will be placed to the output HTML without encoding of special HTML symbols.

1.6.5 Totals

To generate totals use column attribute **total.** Possible values:

- **sum** - sum
- **max** - maximum value
- **min** - minimum value
- **avg** - average value
- **count (*)** - number of rows
- **count (distinct)** - number of unique values

Total cell could be formatted separately using **totalvalueformat attribute of a FIELD tag.**
When using formatted values sum in totals could differ from sum of visible values. To prevent it use **precision** attribute (=decimal places) in FIELD.

1.6.6 Styling

There are many ways of styling table cells. In addition to a **template-cell** template we can use CSS as following:

1. attribute **FIELD.class** specifies the class that will be applied to all cells in the column. For example

```
<styles>
.green_cell {background: #eeffee}
</styles>
…
   <FIELD name="CITY" caption="City" class="green_cell"/>
```

   Fills all **CITY** column cells with green

2. It is possible to use an attribute field with the field name . For example

```
td [field="COMPANY"] {background: yellow}
```

   Make all cells in COMPANY column yellow.
   Row and cell attributes could be combined. So,
   `tr[state="HI"] td[field="COMPANY"] {background: yellow}`
   Fills **COMPANY** cells only for rows where State = HI

3. Cell class could be directly specified in SQL by alias. For example

```
   <style>.
   FL {background: yellow}c.
</ style>
…
   Select c.*, C.STATE as ROW_CLASS from customer c
```

Makes yellow all rows with STATE = FL

Class could be specified for a single cell. In this case, an alias <FIELD> _CLASS is used.
Example:

```
   select c.*, iif(position('6' in Phone)>0, 'red_cell', '') as
PHONE_CLASS from customer c
```

Will set **red_cell** class for all cells in the **PHONE** column containing '6'.

4. All rows in the table contains row attribute with a line number which also could be used for styling. In addition, the even rows contain class **odd.** Following style example grayed out all even rows:

```
tr.odd {background: #eee}
```

5. All cells contain a type of the field in a class attribute. The combination of these attributes could be used for styling.

   Set center alignment for all integer cells:

   ```
   td.integer {text-align: center}
   ```

1.6.7 Groups

To make a grouped report use METADATA.group attrribute. It specifies a comma-separated list of fields that are grouped.
By default, totals are displayed for each group. If you want to add a common totals below table use alltotals = "true" attribute of METADATA tag.
Group row (tr) are marked with group class, which could be used for styling.

Group rows are generated using **template-group-row** template**.**
Default template-group template contains space-separated grouping fields.

By default, row will be sorted by group fields, to prevent this add meta attribute groupsort="false"

1.6.8 Cross reports

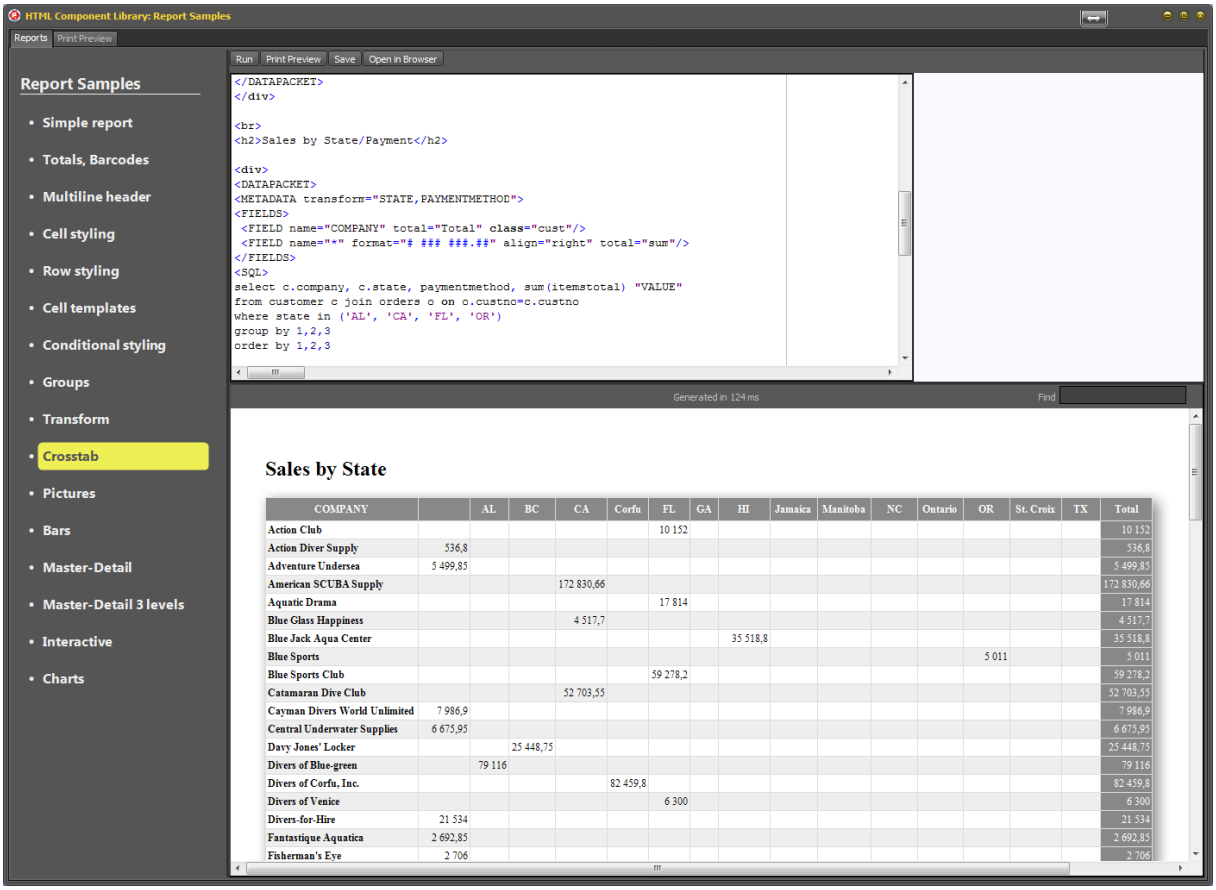
To make a cross-report use META.transform attribute, which specifies a list of comma-separated fields.
Calculated field in the query must have an alias VALUE.
Parameters for calculated columns are set in a field named *. For example

```
<FIELD name="*" format="# ### ###.##" align="right" total="sum"
totalcolumn="false"/>
```

Cross-report example:

```
<DATAPACKET>
 <METADATA transform="STATE">
  <FIELDS>
   <FIELD name="COMPANY" total="Total" class="cust"/>
   <FIELD name="*" format="# ### ###.##" align="right" total="sum" totalcolumn="max"/>
  </FIELDS>
  <SQL>
   select c.company, c.state, paymentmethod, sum(itemstotal) "VALUE"
   from customer c join orders o on o.custno=c.custno
   group by 1,2,3
   order by 1,2,3
  </SQL>
 </METADATA>
</DATAPACKET>
```

And the result of its execution:



To avoid gaps in columns, specify a range of possible values in “transform” attribute. For example, transform = "sellyear (2005-2014)"

You can specify an expression for caption of range columns using caption attribute for "*" column. Example:

```
<FIELD name="*" format="# ### ###.##" align="right" total="sum" caption="?copy(value, 1, 2)"/>
```

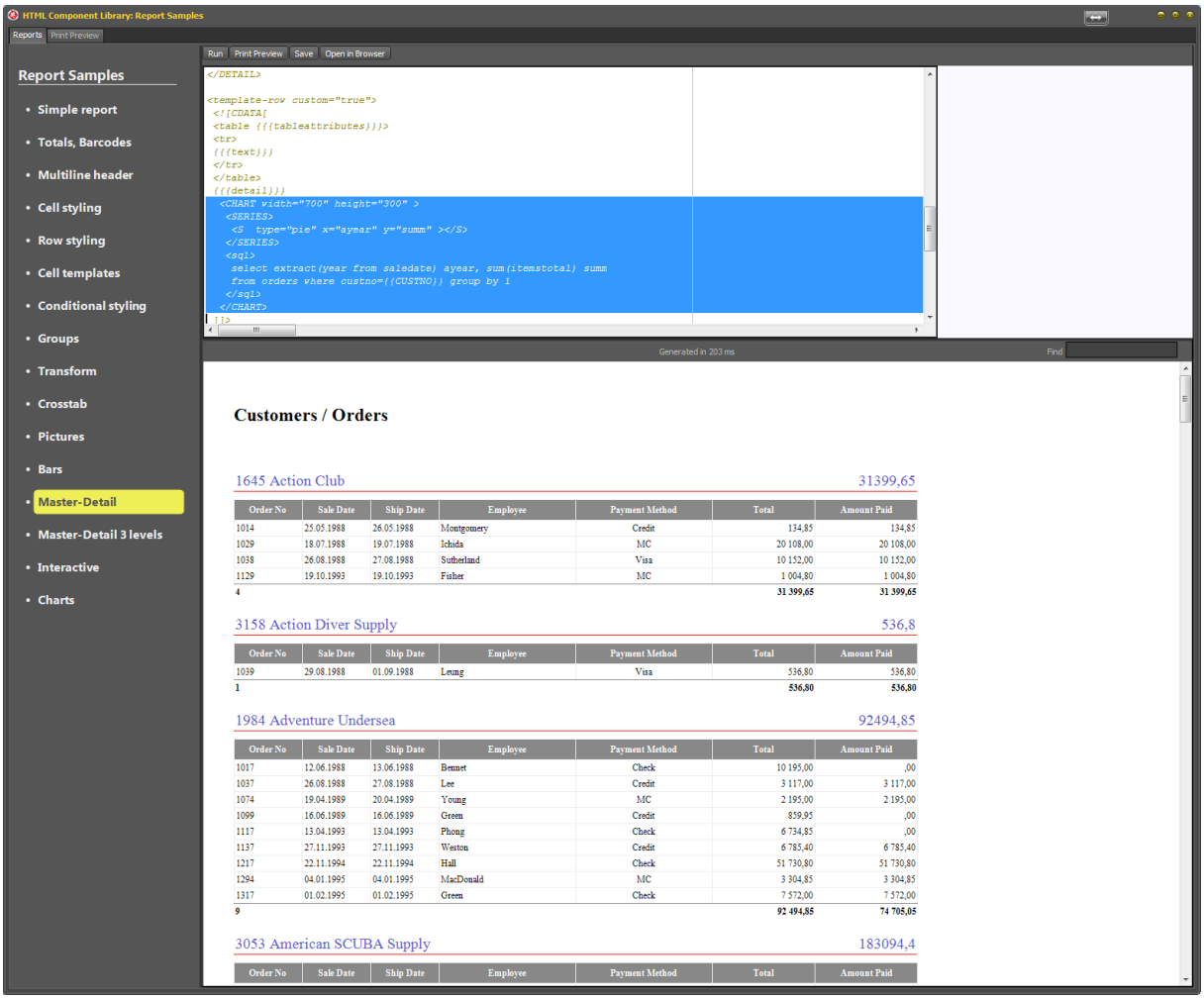
To disable column with row totals set totalcolumn="false" for * field.

1.6.9 Master-Detail

Detail datasets nested to a DATAPACKET section has same format as top-level DATAPACKET but with tag name DETAIL. In detail packet SQL, values of master fields (current row) could be used as SQL parameters like :CUSTNO. For example:

```
select o.orderno, cast(saledate as date) saledate, cast(shipdate as date) shipdate,
 paymentmethod, itemstotal, amountpaid, e.lastname
from orders o
join employee e on e.empno=o.empno
where custno=:custno
order by 1
```

Result:



You can use nested DATAPACKET in cell or row templates. In this case SQL parameters should be set via template variables, like {{master.CUSTNO}}

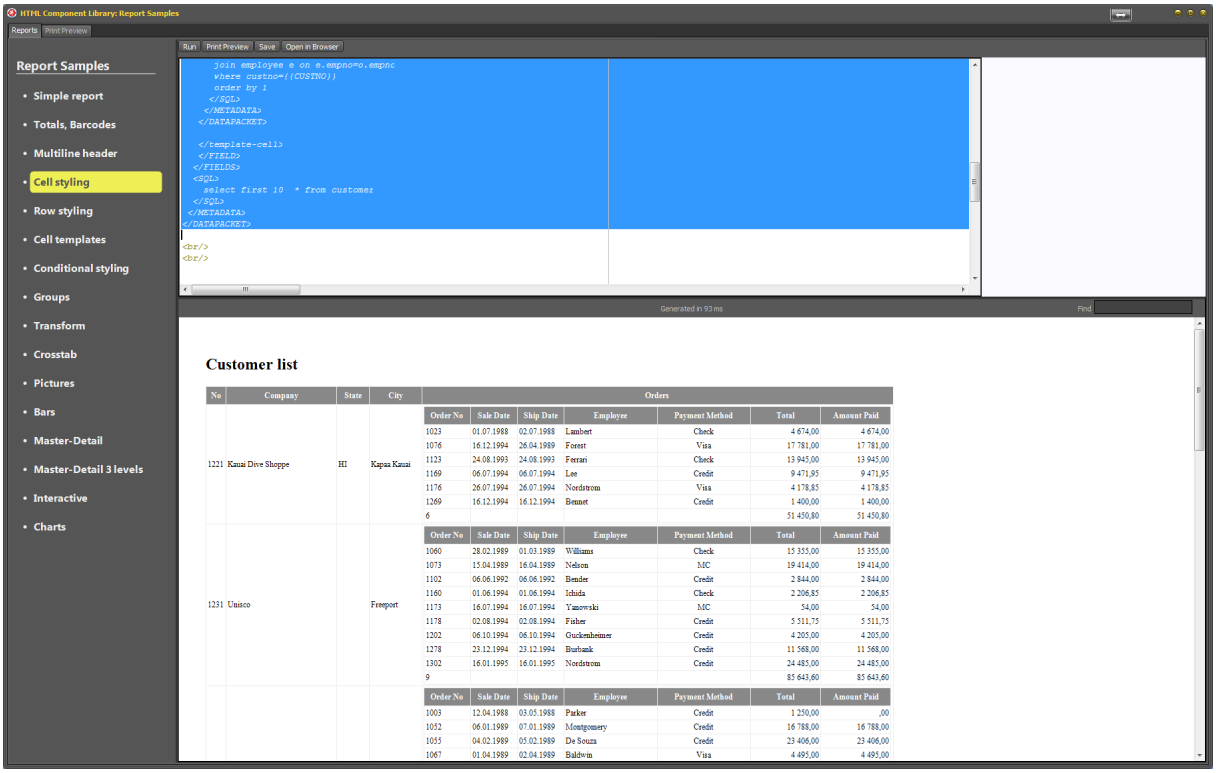
Example of a DATAPACKET embedded in cell

```
<DATAPACKET>
 <METADATA>
  <FIELDS>
   <FIELD name="CUSTNO" caption="No"/>
   <FIELD name="COMPANY" caption="Company"/>
   <FIELD name="STATE" caption="State"/>
   <FIELD name="CITY" caption="City" />
   <FIELD name="Orders">
   <template-cell>

    <DATAPACKET width="700">
     <METADATA>
      <FIELDS>
       <FIELD name="ORDERNO" caption="Order No" width="10%"/>
       <FIELD name="SALEDATE" caption="Sale Date" width="10%"/>
       <FIELD name="SHIPDATE" caption="Ship Date" width="10%"/>
       <FIELD name="LASTNAME" caption="Employee" width="20%"/>
       <FIELD name="PAYMENTMETHOD" caption="Payment Method" align="center" width="20%"/>
       <FIELD name="ITEMSTOTAL" caption="Total" total="sum" width="15%"/>
       <FIELD name="AMOUNTPAID" caption="Amount Paid" width="15%"/>
      </FIELDS>
     <SQL>
      select o.orderno, cast(saledate as date) saledate, cast(shipdate as date) shipdate,
      paymentmethod, itemstotal, amountpaid, e.lastname
      from orders o
      join employee e on e.empno=o.empno
      where custno={{CUSTNO}}
      order by 1
     </SQL>
    </METADATA>
   </DATAPACKET>

   </template-cell>
   </FIELD>
  </FIELDS>
  <SQL>
    select * from customer
  </SQL>
 </METADATA>
</DATAPACKET>
```

Result:



1.6.10 Drop-down groups

To create a drop-down groups with DETAIL packet data, use the following template for the master table:

```
<template-row custom="true">
 <![CDATA[
  <table {{{tableattributes}}}>
   <tr>
    {{{text}}}
   </tr>
  </table>
  <input type="checkbox" class="tree" id="{{CUSTNO}}"/>{{{detail}}}
 ]]>
</template-row>
```

We use input element as a storage for group state.
In the cell template of the main table specify:

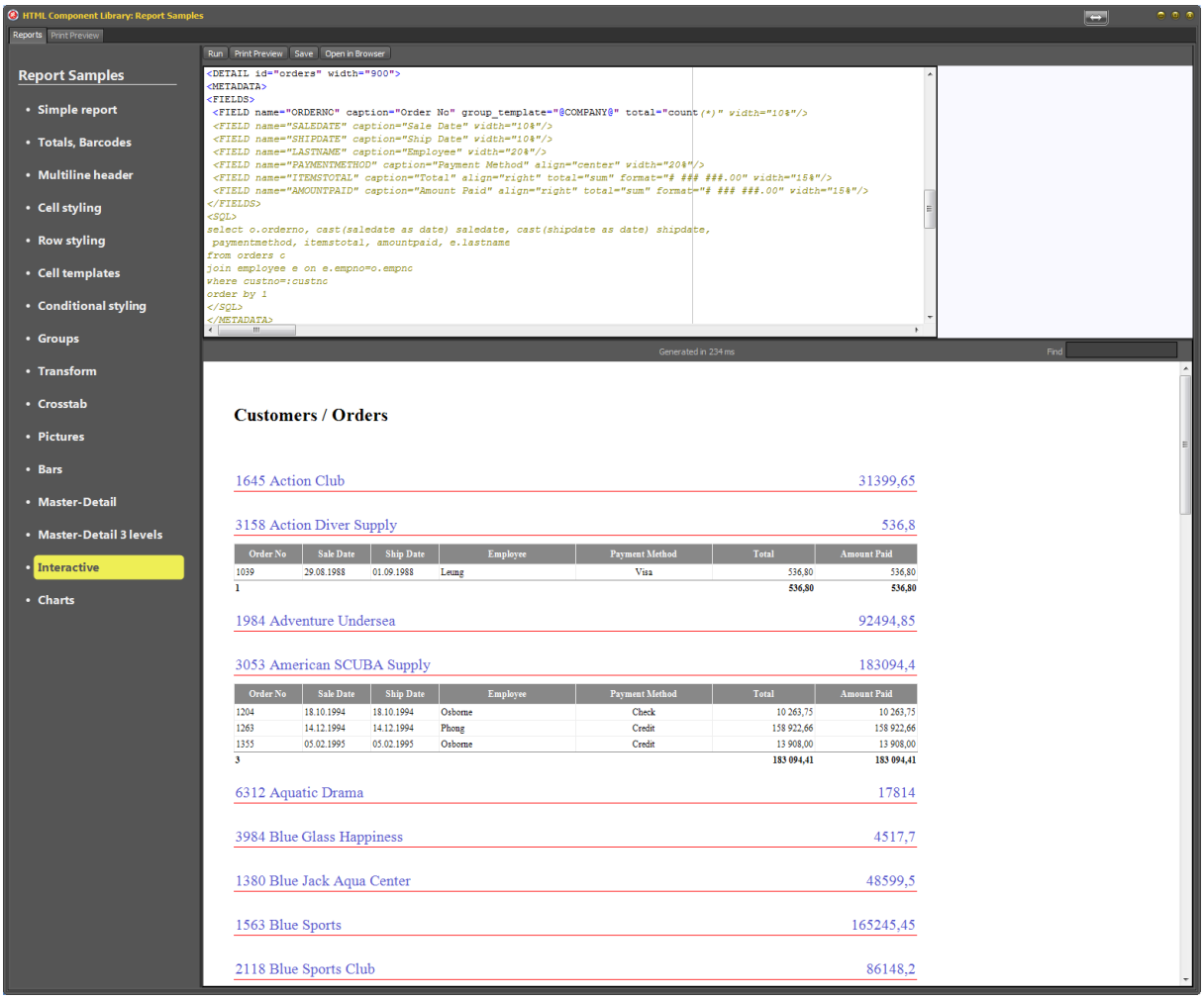
```
  <template-cell>
   <label for="{{CUSTNO}}"><a>{{CUSTNO}} {{COMPANY}}</a></label>
  </template-cell>
```

And add styles:

```
table {
 overflow: hidden;
```

```
 display: block;
 transition: height 0.3s;
 transition: opacity 0.3s;
 }
.tree {display:block; position: absolute; left:-500; top:-500}
input + table {height: 0; opacity: 0.2;}
input:checked + table {height: auto; opacity: 1;}
```

Result:



1.6.11 Images

To display an images from the database use standard **img** tag with src="data..." parameter .

```
<template-cell>
 <img src="{{{GRAPHIC}}}">
</template-cell>
```

1.6.12 Output non-tabular data

DATAPACKET could be used to display non-tabular data like lists or labels. In this case templates should be used with custom =true parameter, which removes the framing tags <table> and <tr> and allowing to define full structure of a data.

Consider output a simple list of customers:

```
<DATAPACKET>
 <METADATA>
   <SQL>
   select * from customer
  </SQL>
 </METADATA>
  <template-table custom="true">
    <ul>
    {{#rows}}
     <li>{{COMPANY}}</li>
    {{/rows}}
    </ul>
  </template-table>
</DATAPACKET>
```

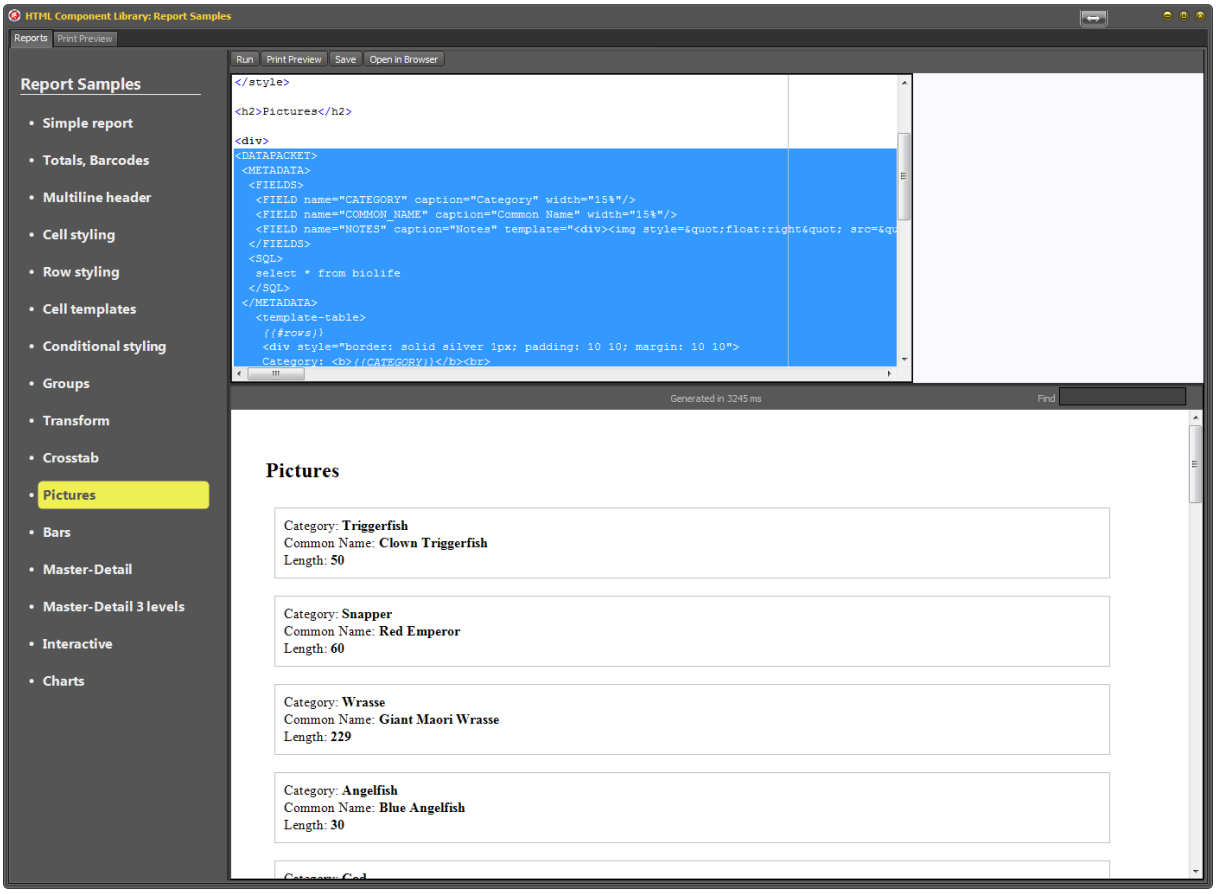
Result:

- Kauai Dive Shoppe
- Unisco
- Sight Diver
- Cayman Divers World Unlimited
- Tom Sawyer Diving Centre
- Blue Jack Aqua Center

More complex template:

```
<DATAPACKET>
 <METADATA>
  <SQL>
   select * from biolife
  </SQL>
 </METADATA>
   <template-table custom="true">
    {{#rows}}
    <div style="border: solid silver 1px; padding: 10 10; margin: 10 10">
    Category: <b>{{CATEGORY}}</b><br>
    Common Name: <b>{{COMMON_NAME}}</b><br>
    Length: <b>{{LENGTH__CM_}}</b><br>
    </div>
    {{/rows}}
  </template-table>
</DATAPACKET>
```

Result:



By default, template-table contains

```
{{{header}}} {{{body}}}
```

And for custom templates

```
<table {{{attributes}}}>
 {{{header}}}
 {{{body}}}
</table>
```

Also XML object created by datapacket (like any other XML report object) can be used to produce non-tabular HTML. Example:
Consider we have datapacket with name="mypacket".
After this datapacket following section can be placed:

```
<ul>
{{#mypacket.ROWDATA}}
  <li>{{NAME}}</li>
{{/mypacket.ROWDATA}}
</ul>
```

1.6.13 Examples of using table template

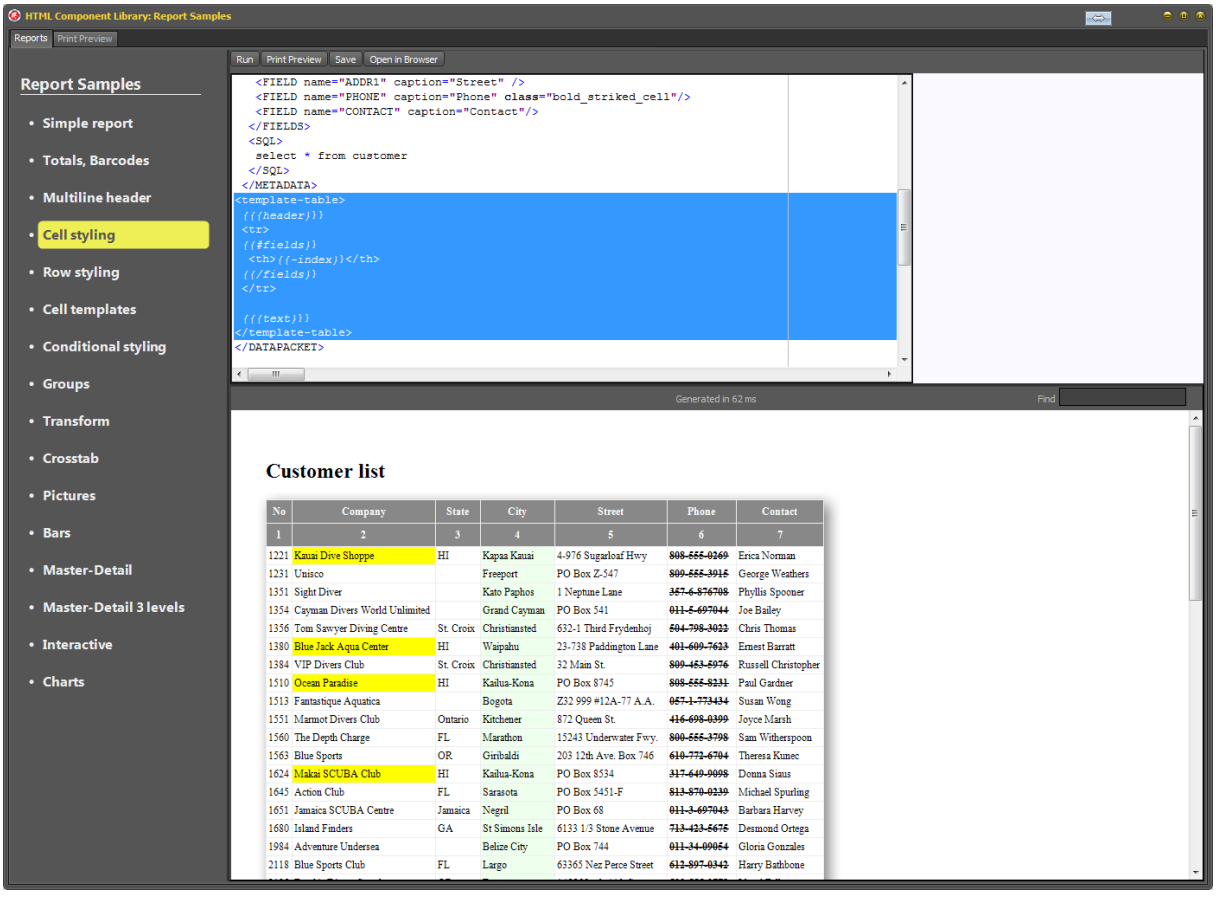
1.6.13.1 Column numbering.

Here we use **fields** variable to add a line between table header body

```
<template-table>
  {{{header}}}
   <tr>
    {{#fields}}
     <th>{{-index}}</th>
    {{/fields}}
  </tr>

  {{{body}}}
</template-table>
```

Result:



More complex example: what if first column should be left blank and numbering should start in second column.

```
<template-table>
```

```
 {{{header}}}
 <tr>
  <th></th>
  {{#fields}}
   {{^-last}}
    <th>{{-index}}</th>
   {{/-last}}
  {{/fields}}
  </tr>

 {{{body}}}
</template-table>
```

Inverted pseudo- section ^-last process all iterations except last one.

1.6.14 Column moving and sizing

You could allow user to adjust column width and move columns.

For column sizing add

```
   resize: horizontal;
```

to CSS th style

For column moving add

```
   draggable: true;
```

to CSS th style

1.6.15 User Grouping

To allow user to group tables add

```
draggable: true;
```

to th style, and add drop target for grouping columns – any element with table-group-box class.

Also you could set style for group rows.

For example:

```
<style>

.table-group-box {background: #888; border: solid #white 1px;
padding: 3 5; margin: 5 0; color: white}

 .group {background: #8a8; font-weight: bold; font-size: 16px;
color: white}
```

```
</style>

<div class="table-group-box">Drag a column header here to group by
that column</div>
```

Group fields on `.table-group-box` element has CSS class `.group-tag`, which could be used to change its look.

Grouping should not be used on tables having “rowspan” cells.

1.6.16 Expressions

Field value and some of field parameters can be calculated using expressions.
For detailed description of expressions syntax and available functions please refer to the HTML Scripter documentation.

For field value use "value" attrubute. Example:
<FIELD value="copy(company, 1, pos(',', company) - 1)" .../>

Also expressions can be used in following FIELD attributes:

- empty
- class
- align
- format
- template
- style
- background

To use expression in these attributes, start it with =.
Example:

```
align="=iif(rownum>1, 'left', 'right')"
```

Expression has the following predefined variables:

- <**field**> - all row fields.
- **CurrentRow**: THtXMLNode - current row
- **field:** string - current field name
- **rownum**: integer - current row number
- **context**: THtXMLNode - report context
- **masterdata**: THtXMLNode - current master row for detail packet.

1.6.17 XPath

Field value can be defined using XPath expression. It is useful when datapacket is generated from complex XML .

Set FIELD.path attribute to any valid XPath expression. For example:

```
<FIELD path="Notes/Note[Kind='general']" .../>
```

1.6.18 Script

Script section can be used for complex data processing and generating.
For detailed description of script syntax and available functions please refer to the HTML Scripter documentation.

Script is executed right after SQL section (if present) and before any other formatting is applied.

Script has the following predefined variables:

- <all variables defined by Report.AddVariable and generated by datapackets>
- **datapacket:** THtXMlNode - current datapacket
- **rowdata** : THtXMLNode - current rowdata node
- **fields** - current fields node
- **report** : THtReport - report object
- **context**: THtXMLNode - report context
- **adapter** : THtSQLAdapter - current SQL adapter
- **masterdata**: THtXMLNode - current master row for detail packet.

Example of generating data using script:

```
<DATAPACKET>
 <METADATA>
  <FIELDS>
   <FIELD name="CUSTNO"/>
   <FIELD name="ADDR1" />
   <FIELD name="COMPANY" />
   <FIELD name="CITY" />
   <FIELD name="STATE" />
   <FIELD name="PHONE" />
   <FIELD name="CONTACT" />
  </FIELDS>
  <SCRIPT>
   for i:= 1 to 100 do begin
     R := RowData.Add('R');
     R.Attr['COMPANY'] := 'Company';
   end;
  </SCRIPT>
 </METADATA>
 <ROWDATA/>
</DATAPACKET>
```

Example of processing data using script:

```
<DATAPACKET>
 <METADATA>
  <FIELDS>
   <FIELD name="CUSTNO"/>
   <FIELD name="ADDR1" />
   <FIELD name="COMPANY" />
   <FIELD name="CITY" />
   <FIELD name="STATE" />
   <FIELD name="PHONE" />
   <FIELD name="CONTACT" />
  </FIELDS>
  <SQL>
   select * from customer
  </SQL>
  <SCRIPT>
  for i:= 0 to Rowdata.Count - 1 do
    RowData.Nodes[i].Attr['CITY'] := 'test';
  </SCRIPT>
 </METADATA>
</DATAPACKET>
```

1.6.19 Filtering

Filter expression can be applied to table data. To set datapacket filter use SQL.filter attribute.

Example:

<SQL filter="AnsiStartsWith(Company, 'A')">

1.6.20 Limit record count

To limit number of records in rowset use sql maxcount attribute. F.e:

<sql maxcount="10">...</sql>

Rest records will be grouped into one record with first field set to "...(Number of records)", other fields with total="sum" will contain summary value.

1.6.21 Sorting

Table can be sorted by any column using sql **order** attribute. Sorting is applied after any transforms (for cross reports) and filtering.

For example to sort cross report by total column in descending order use (^ means descending):

<sql order="^FTOTAL">...</sql>

1.6.22 User sorting

To allow user to sort tables by clicking on columns header set METADATA.autosort attribute to true. You can adjust sort indicators using CSS classes **.sort-up** and **.sort-down**

By default they are:

```
.sort-up:after {content: " \25BC"; }
.sort-down:after {content: " \25B2";
```

When **autosort** is set to true, <a onclick="this.parent.parent.parent.SortColumn:=this.parent.Col;"> is added to each column header (th) by setting header cell template variables sort-before and sort-after. So default header cell template now is:

```
<th {{{attributes}}}>{{{sort-before}}}{{{value}}}{{{sort-after}}}</th>
```

Sorting should not be used on tables having “rowspan” cells.

1.7 Charts

Charts are created using CHART tag. It has the following format:

```
     <CHART width="…" height="…" valueformat="//" active="true/
false">
      <SERIES>
       <S type="type" x="x-field" y="y-field" id="id" marks="true/
false"></S>
      </SERIES>
      <SQL>
       …
      </SQL>
     </CHART>
```

Where “type” attribute defines type of the chart, and “x-field” and “y-field” attributes specify labels and values.
Valueformat attribute is used to specify legend and marks values format (via FormatFloat function).
When chart contains several series, z order for series can be set via **zindex** attribute.
id attribute is used to assign unique ID to each chart element.

1.7.1 Chart Colors

Colors are specified by COLORS tag. It contains initial color and steps for Hue and Saturation coordinates in HSV color space.
After passing over full circle in Hue, next step in Saturation space is performed.

```
<COLORS start="#aa8888" hue_step="30" saturation_step="10"/>
```

For multiseries chart color can be specified in a "color" attribute:

```
<SERIES>
 <S color="green" ... >
```

Also colors can be specified explicitly as child nodes for COLORS tag:

```
<COLORS start="#aa8888" hue_step="30" saturation_step="10">
 <COLOR>#203030</COLOR>
 <COLOR>#20aaB0</COLOR>
</COLORS>
```

To define gradient (bar and area series) add COLORS2 node with the same parameters containing second color for gradient. F.e.

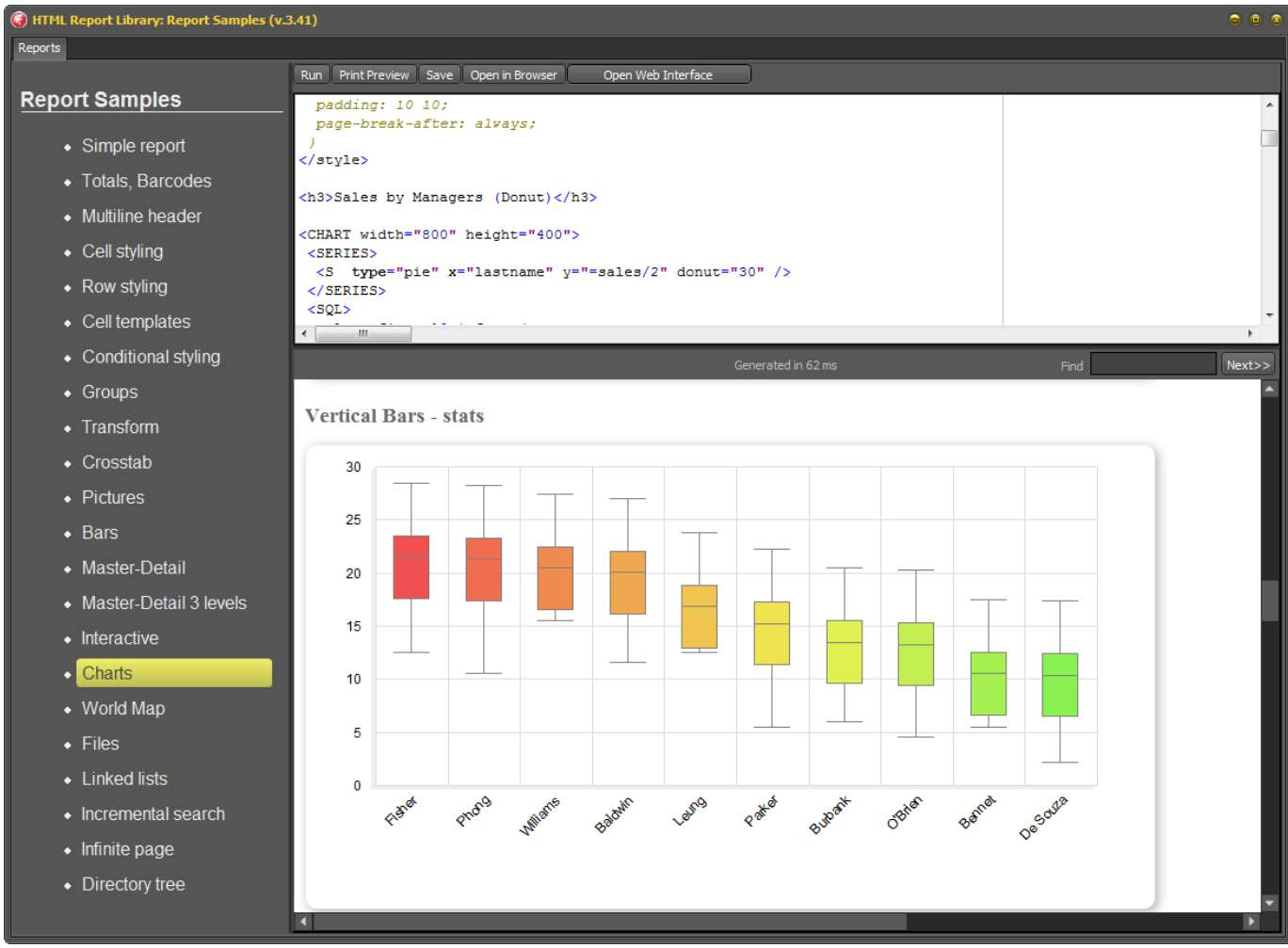
```
<COLORS start="#aa8888" hue_step="30" saturation_step="10"/>
<COLORS2 start="#008888" hue_step="30" saturation_step="10"/>
```

1.7.2 Statistical Errors

Error attribute is used to specify field containing statistical errors. For example:
 <S type="bar" x="lastname" y="sales" stack="syear" error="e" percent3d="0" />

1.7.3 Box-and-whiskers diagram

Use boxplot="true" attribute to display box-and-whiskers diagram. Example:
 <S type="vbar" x="lastname" y="sales" q1="q1" q2="q2" q3="q3" qmin="qmin" qmax="qmax" boxplot="true"/>



1.7.4 Legend

To show or hide legnd set series attribute legend=”true/false”
Position and axes titles:

<LEGEND position="top/bottom/left/right" x-name=”x title” y-name=”y title” x-color="color for x name and labels" y-color="color for y name and labels"/>

To hide legend set legend="false" series attribute.

Following CSS classes can be used for styling legend and its items:

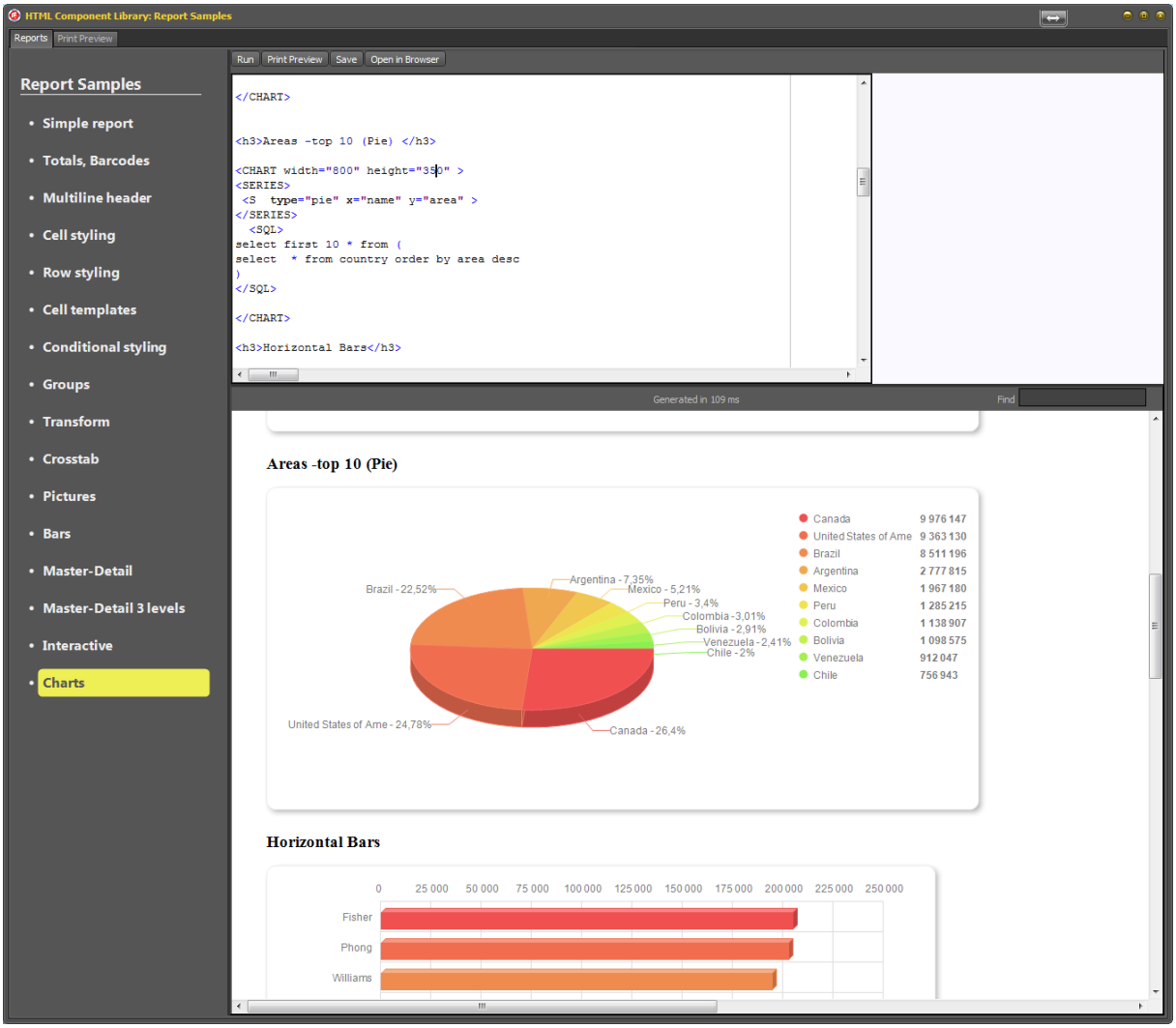
.chart-legend
.chart-legend-item
#xmarks
#ymarks

1.7.5 Pie chart

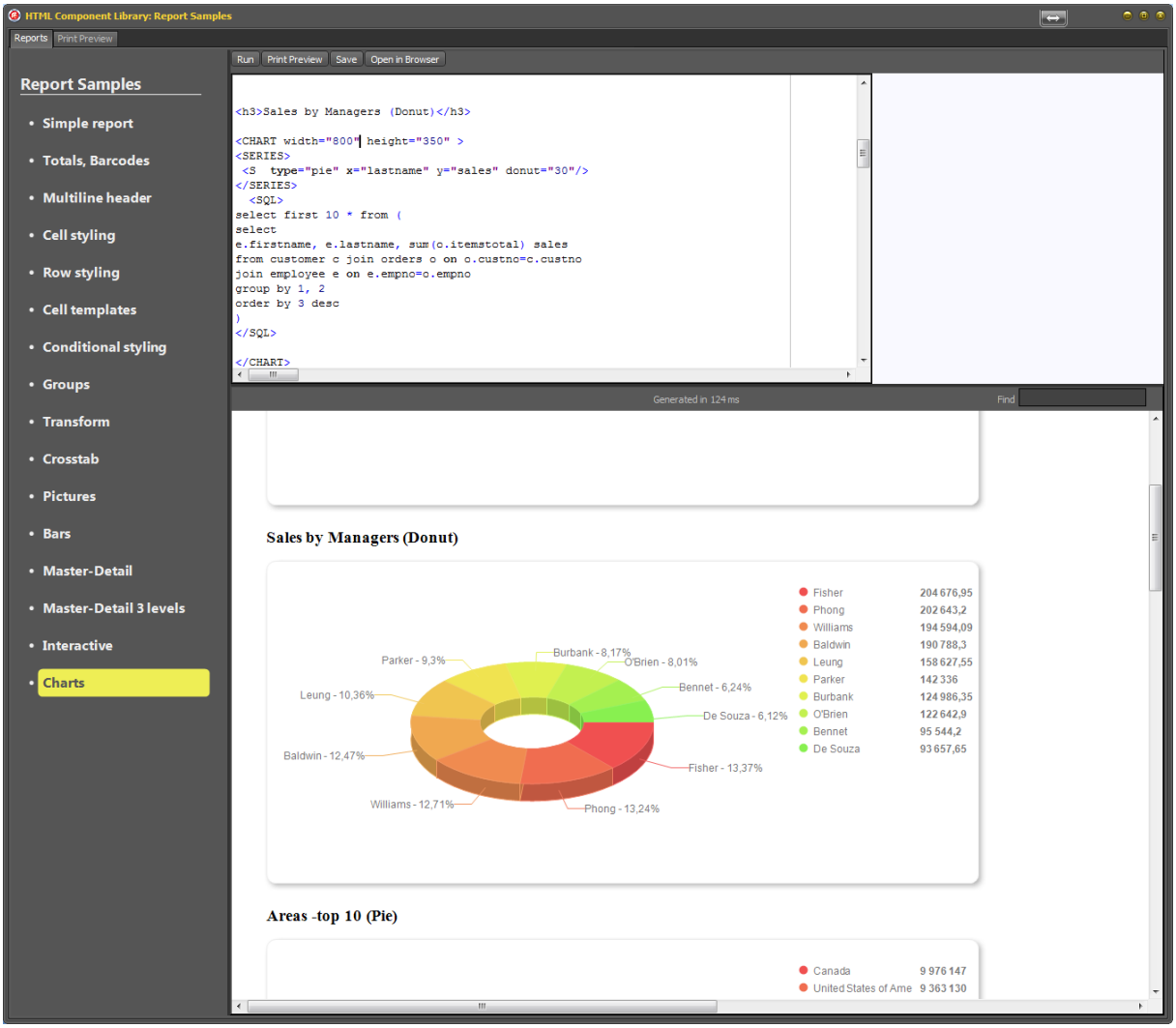
To generate pie chart set series type to “pie”.
Pie chart example:

```
<CHART width="800" height="400">
 <SERIES>
  <S type="pie" x="name" y="area" startangle="30" direction="ccw">
 </SERIES>
 <SQL>
  select first 10 * from (
   select  * from country order by area desc
  )
 </SQL>
</CHART>
```

Result:



Pie series may have an additional parameter donut - radius of the inner circle.

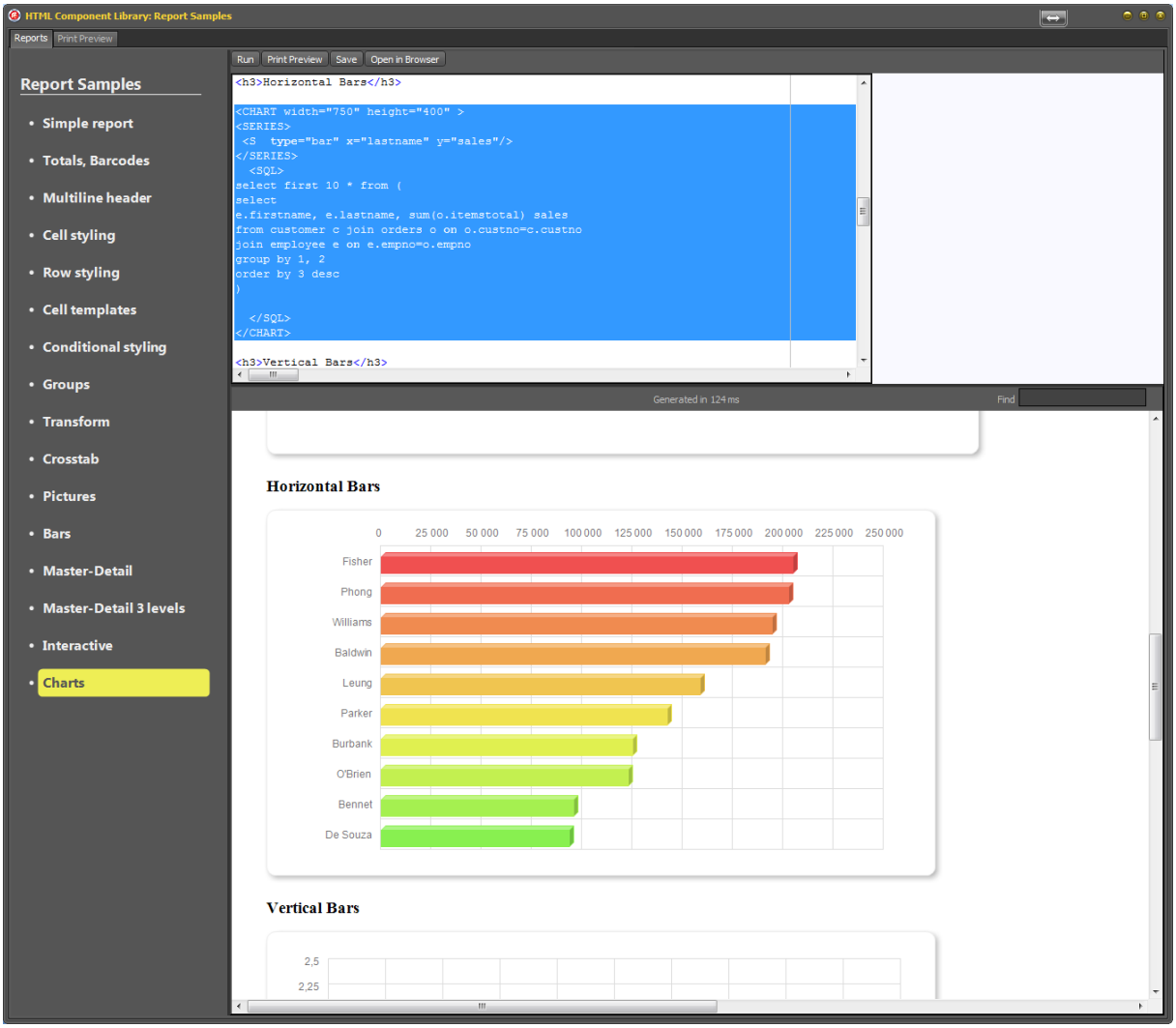


1.7.6 Horizontal bars

For horizontal-bar chart set type to **bar.**
Example:

```
<CHART width="750" height="400" >
 <SERIES>
  <S type="bar" x="lastname" y="sales"/>
 </SERIES>
 <SQL>
  select first 10 * from (
   select e.firstname, e.lastname, sum(o.itemstotal) sales
   from customer c join orders o on o.custno=c.custno
   join employee e on e.empno=o.empno
   group by 1, 2
   order by 3 desc
  )
 </SQL>
</CHART>
```

Result:



Following CSS classes can be used for styling bar sides:
.bar-face
.bar-side
.bar-end

1.7.7 Vertical columns

For vertical columns chart set type to v**bar.**
Example:

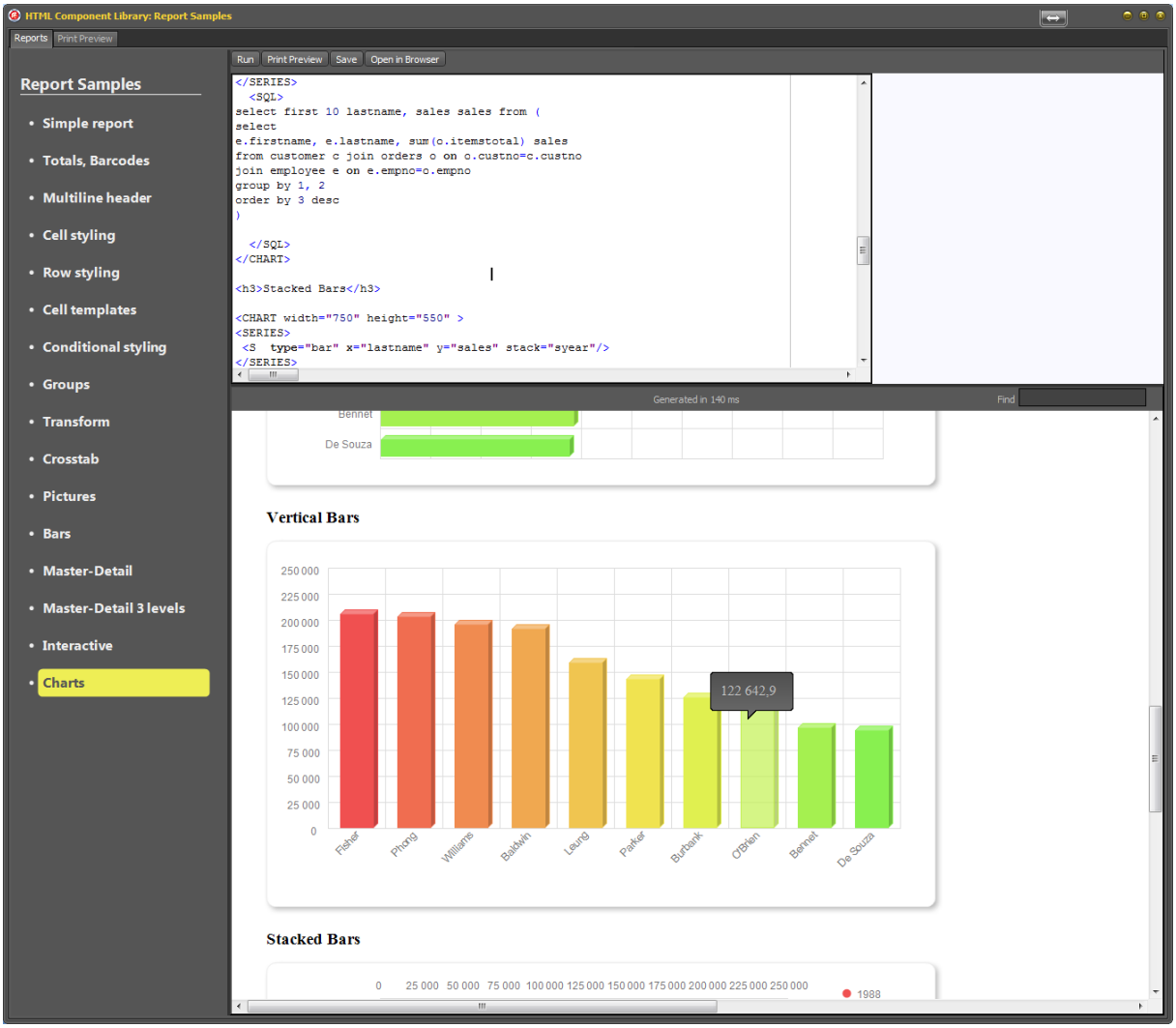


Chart columns have hints with column values.

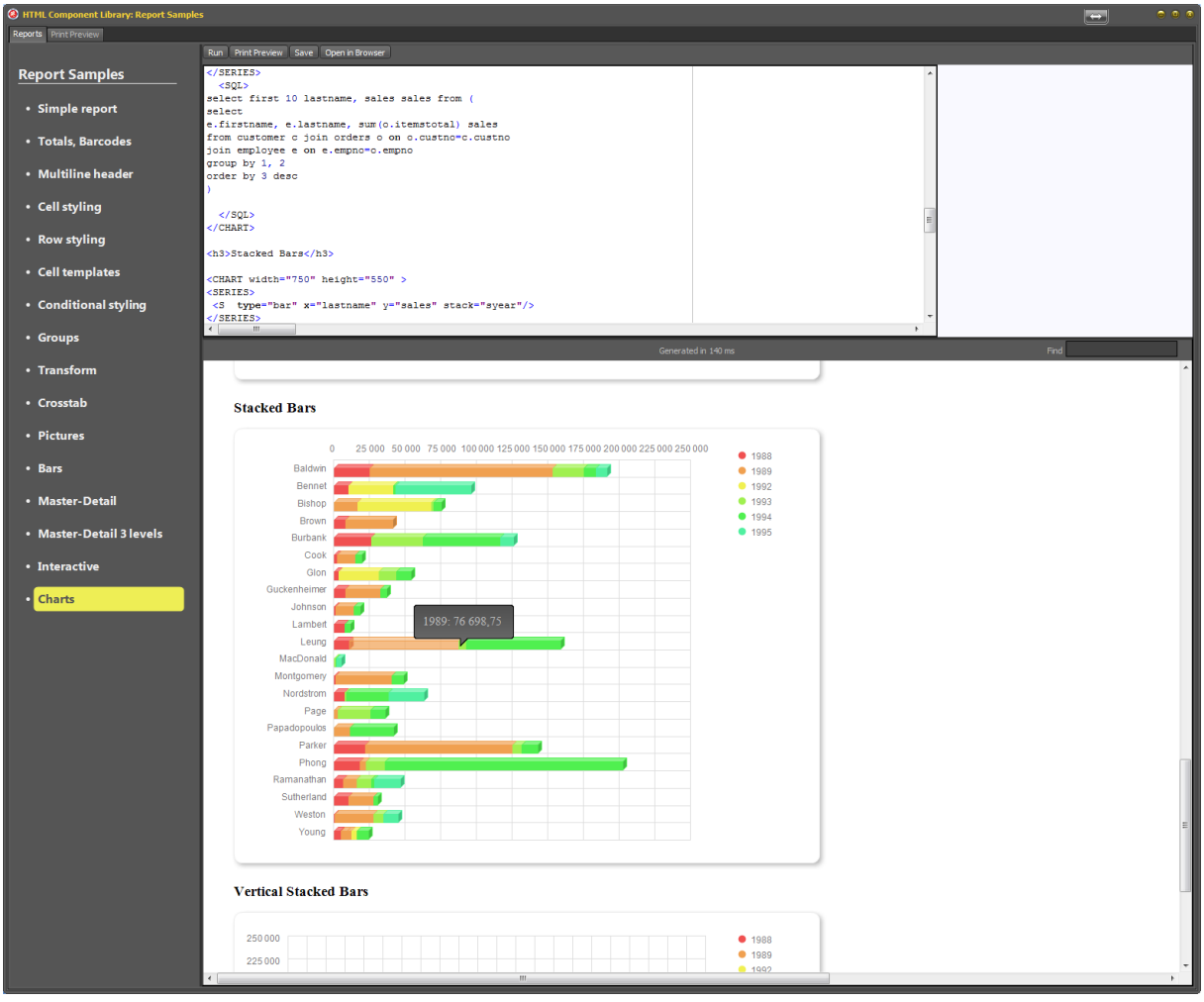
Following CSS classes can be used for styling bar sides:
.bar-face
.bar-side
.bar-end

1.7.8 Composite (stacked) charts

Vertical and horizontal charts could be stacked. Example:

```
<CHART width="750" height="400">
 <SERIES>
  <S type="vbar" x="lastname" y="sales" stack="syear"/>
 </SERIES>
 <SQL>
  select first 80 * from (
   select e.firstname, e.lastname, extract(year from saledate) syear,
    sum(o.itemstotal) sales
   from customer c
    join orders o on o.custno=c.custno
    join employee e on e.empno=o.empno
   group by 1, 2,3
   order by 1,2,3
  )
 </SQL>
</CHART>
```

Result:



Also stacked charts can be created from list of series. Set stack field for each series to "*", f.e.:

```
<S type="bar" x="DEPTH" y="BEFORE" name="Pressure before" stack="*"></S>
<S type="bar" x="DEPTH" y="AFTER" name="Pressure after" stack="*"></S>
```

1.7.9 Side-by-side charts

Use stacked chart with sidebyside attribute:

```
<S  type="vbar" x="lastname" y="sales" stack="syear" percent3d="0" sidebyside="true"/>
```

1.7.10 Arranging stack (legend) values

To arrange legend values for stacked and side-by-side chart use stackorder attribute.

Stackorder attribute contains expression taking stack field value and returning integer for ordering stack values.

For example if stack field contains years and we want to display it in reverse order stackorder attribute should be set to:

stackorder="-strtoint(value)"

1.7.11 Arranging chart values

To arrange chart values use series **order** attribute.

Order attribute contains expression taking field value and returning integer, string or float for ordering chart values.

For example if field contains volume and we want to display it from largest to smallest, **order** attribute should be set to:

order="-strtofloat(volume)"

For stacked chart use **x-order** attribute to arrange x values, f.e.:

x-order="StrtoInt(value)"

1.7.12 Areas

Use series type=”area”

1.7.13 Lines

Use series type=”line”. Line series has following additional attributes

linedots="true" : show points on a line.

x-type="number" : x-axe values are numbers.
hint="fieldname" : hint field.

1.7.14 Scatter

Use series type=”scatter”

1.7.15 Charts inside table

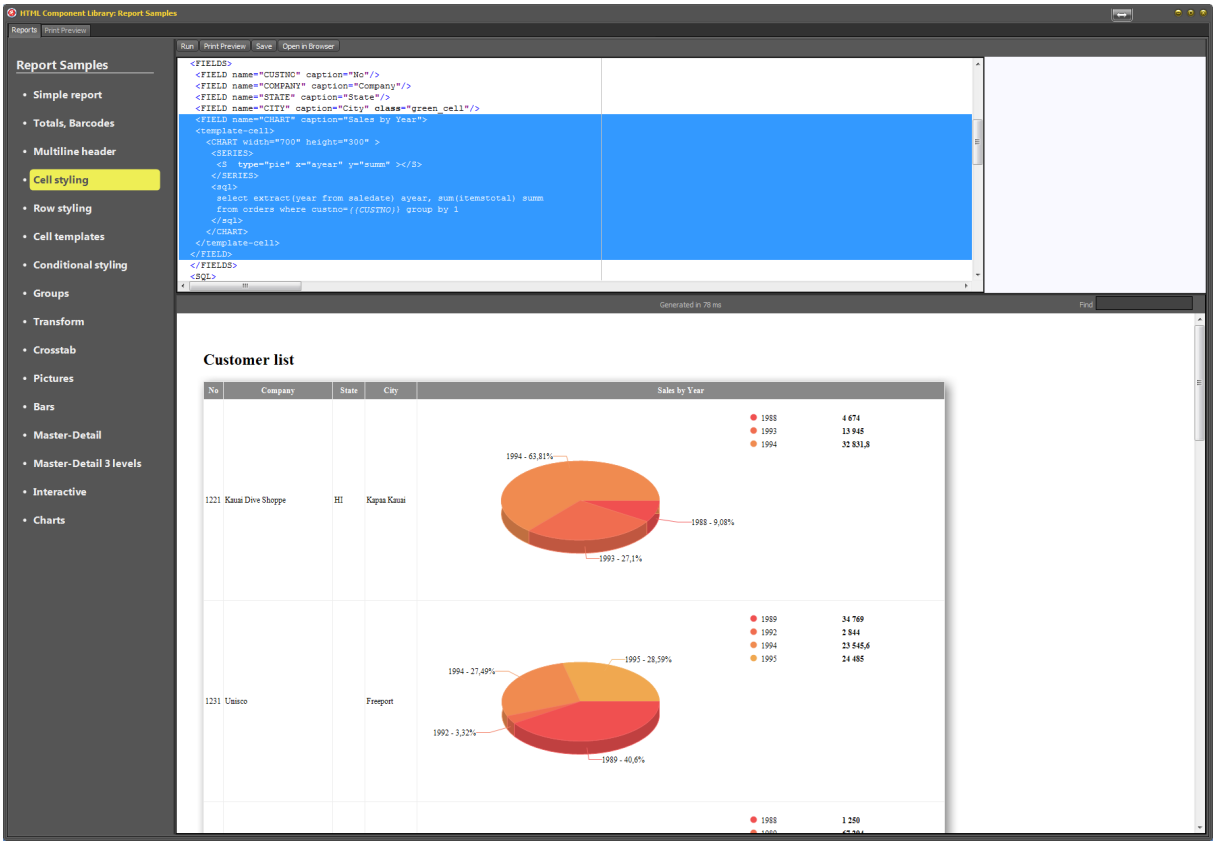
Charts could be generated in any template of a table (DATAPACKET) – in table, row or cell.

Example of inserting chart into table cell.

Cell template:

```
<FIELD name="CHART" caption="Sales by Year">
 <template-cell>
   <CHART width="700" height="300" >
    <SERIES>
     <S type="pie" x="ayear" y="summ"></S>
    </SERIES>
    <sql>
     select extract(year from saledate) ayear, sum(itemstotal) summ
     from orders where custno={{CUSTNO}} group by 1
    </sql>
   </CHART>
 </template-cell>
</FIELD>
```

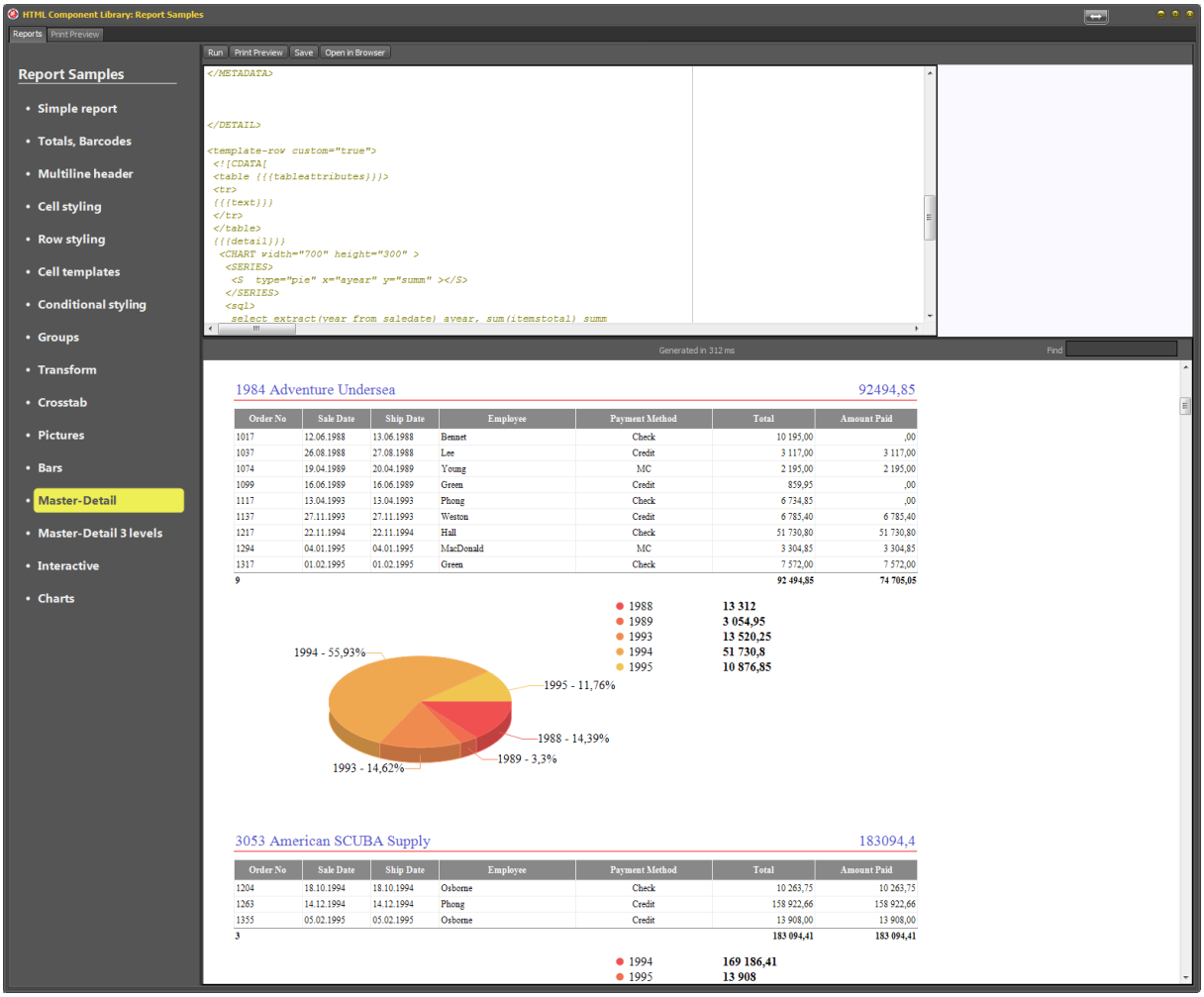
Result



Example of generating chart after detail table:

```
<template-row custom="true">
 <![CDATA[
  <table {{{tableattributes}}}>
   <tr>
    {{{text}}}
   </tr>
  </table>
  {{{detail}}}
  <CHART width="700" height="300" >
   <SERIES>
    <S  type="pie" x="ayear" y="summ" ></S>
   </SERIES>
   <sql>
    select extract(year from saledate) ayear, sum(itemstotal) summ
    from orders where custno={{CUSTNO}} group by 1
   </sql>
  </CHART>
 ]]>
</template-row>
```

Result:



1.7.16 Expressions

Some of series parameters can be calculated using expressions. To use expression in these attributes, start it with =.

For detailed description of expressions syntax and available functions please refer to the HTML Scripter documentation.

Following attributes can contain expressions:

- yfield
- q1
- q2
- q3
- qmax
- qmin
- error
- hint

All field values of current row are available in expression as predefined variables.

1.7.17 Filtering

Filter expression can be applied to chart data. To set chart filter use SQL.filter attribute.

Example:

```
<SQL filter="AnsiStartsWith(Company, 'A')">
```

1.7.18 Limit values count

To make chart for top N values use series **maxcount** attribute, f.e.

```
<S type="bar" x="lastname" y="sales" maxcount="10"/>
```

Values will be sorted from largest to smallest and only first **maxcount** values will remain in chart. For stacked charts sorting is performed by sum of stacked values.

1.7.19 Interactive charts

To make chart interactive set CHART attribute active to 'true'.

Report document should contain <script type="passcript">..</script> section, otherwise some interactive features will not work.

1.8 Barcode

To generate a barcode use special barcode element. Example:

```
<barcode width="100" height="30" code="12345"/>
```

Width and height should be specified using attributes or CSS.

Barcodes are generated in vector (SVG) format.

1.9 QR Codes

To generate a QR code use special qrcode element. Example:

```
<qrcode width="100" code="12345"/>
```

Optional ecc attribute defines ECC level (Medium, Low, High, Quartile).

Real width and height could be specified using CSS style.

QR codes are generated in vector (SVG) format.

1.10 Database adapters

Base class for SQL adapter contains two functions:

```
THtSQLAdapter=class
public
 function CreateDataset(const SQL: string; const Params: array of variant; D: TDataset=nil): TDataset; virt
 procedure DestroyDataset(D: TDataset); virtual; abstract;
end;
```

First should execute SQL with provided parameters and return result as dataset. If dataset is passed in last parameter it should be used instead of creating new dataset.

Second function is intended to destroy dataset after use (or store it in a pool).

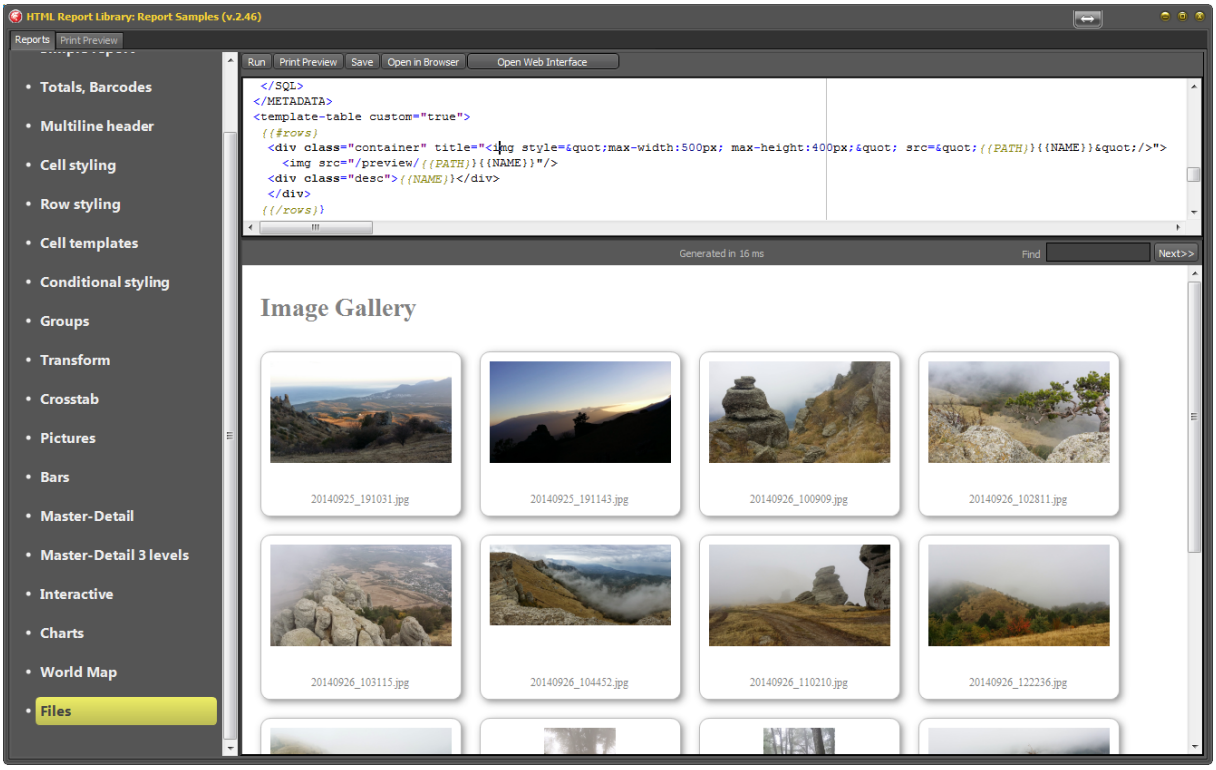
1.10.1 Non-SQL adapters

Adapters could be also used for non-SQL (non-dataset) data sources.

These adapters should override function

```
function GetXML(const SQL: hstring; const Params: array of variant): THtXMLNode; virtual;
```

For example please see THtDirectoryAdapter in htsql.pas unit, providing access to files in a specified directory.



1.10.2 Adapters registration

Adapters should be registered via RegisterHTSQLAdapter (Name, AdapterClass) function. To register default adapter pass blank string to registration function.

Adapter name could be used in any SQL tag in type attribute. For example

```
<SQL type="directory">
 pics\*.jpg
</SQL>
```

1.10.3 Build-in adapters

THtXMLAdapter - htreports unit. Allow to use XML as dapasource. Set Datapacket.Metadata.SQL value to Xpath expression.

Example:

```
Report.AddVariable('mynode', MyNode);
...
<SQL type="xml">
mynode/users
</SQL>
```

Since THtXMLNode can also parse JSON via CreatefromJSON method, this adapter can also be used to work with JSON data.

JSON is translated into XML using the following rules:

- JSON array is translated into 'array' node with children represented array items, item can be 'object', 'array' or 'value' depending on its type. Value contains JSON value in node text (<value>This is JSON</value>).
- JSON object is translated into 'object' node with children nodes for array or object and attributes for simple values.

THtDirectoryAdapter - htsql unit. Use directory as datasource.

Example:

```
<SQL type="directory">
c:\delphi\*.pas
</SQL>
```

THtFiredacAdapter - htfiredac unit. Adapter for FireDAC library.

THtUniDACAdapter - htfiredac unit. Adapter for UniDAC library.

THtIBXAdapter - htibx unit. Adapter for IBX library.

THtUIBAdapter - htuib unit. Adapter for UIB library.

1.11 Report objects

In addition to passing objects to report via **AddVariable**, objects can be defined inside report.

```
<report-objects>
  <object name="cust" sql="select * from customer" type=""/>
</report-objects>
```

Object attributes:

- **name** - variable name
- caption - variable caption

- **type** - adapter type
- **sql** - query
- **value** - expression
- **requires** - required template variable. If this variable is empty, exception will be raised.
- **requires-message** - custom message for "requires" exception.

Script

Object also has optional **script** node. Script is executed after SQL.
For detailed description of script syntax and available functions please refer to the HTML Scripter documentation.

Script has the following predefined variables:

- <all variables defined by Report.AddVariable and generated by datapackets>
- **report** : THtReport - report object
- **context**: THtXMLNode - report context
- **adapter** : THtSQLAdapter - current SQL adapter

XML object

When SQL returns single record with single field named XML, content of this field is treated as XML object and registered in report instead of whole result set.

1.12 Using reports in Delphi

1.12.1 Creating report

```
R:= THtReportDocument.Create;
try
 R.Parse(TemplateText);
 s := R.Render;
finally
  R.Free
end;
```

1.12.2 Providing database connection

THtReport document has SQLAdapter property which could be used to set SQL adapter directly.

If SQLAdapter is not set report uses global variable **HtSQLAdapter**.

1.12.3 Passing parameters to report

Passing XML data:

Example:

```
Animals:= THtXMLNode.Create('<animals><animal name="Dog" color="yellow"/>'+
  '<animal name="Bird" color="green"/></animals>');
R := THtReportDocument.Create;
try
 R.RegisterVariable('animals', Animals, true);
 R.Parse(TemplateText);
 s := R.Render;
finally
  R.Free
end;
```

Passing Delphi objects

For converting Delphi object to XML using RTTI (2010+) there are two functions in htxml unit:

ObjecttoXML(const O: TObject; ClassProp: boolean = false; Node: THtXMLNode = nil; UpperCaseName: boolean = true): THtXMLNode

RecordtoXML(const R; RecTypeInfo: pointer; ClassProp: boolean = false): THtXMLNode

Passing JSON

To pass **JSON** data to report use **THtXMLNode.CreatefromJSON** and pass created object via RegisterVariable function.

1.12.4 Displaying report result

Simply copy generated report text to HtPanel:

```
HtPanel1.HTML.Text:=s;
```

1.12.5 Generating Print Preview

Sample code for generating print preview (VCL):
For report already displayed in HtPanel:

```
uses HtPreviewFrame;

...
ShowHtPrintPreview(HtPanel1.Doc);
```

if report is not displayed in HtPanel:

```
uses HtPreviewFrame;

...
ShowHtPrintPreview(ReportText);
```

1.12.6 Printing report

Sample code to print document:

```
PrintDoc.Print(PrintDialog1);
```

Pass nil to print immediately without dialog.

1.12.7 Export to PDF

Note: For PDF export on Windows you should have SynPDF library installed

Generate print preview document and use **Doc.Surface.G.SavetoPDF** function.