QtReports

USER GUIDE

v0.3.0

HTTPS://GITHUB.COM/PO-31/QTREPORTS

Table of Contents

Chapter 1 Getting Started with QtReports

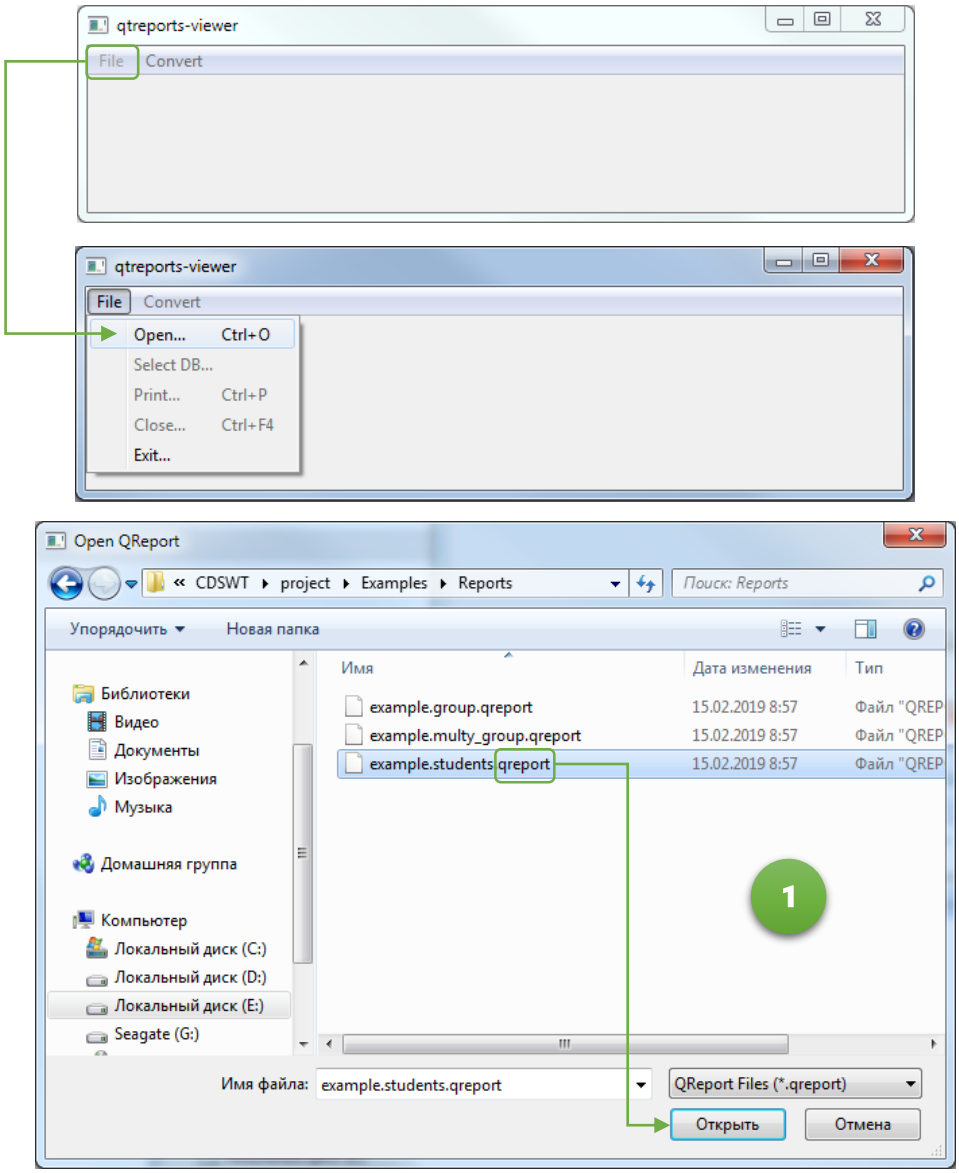
1.1 Introduction to QtReports

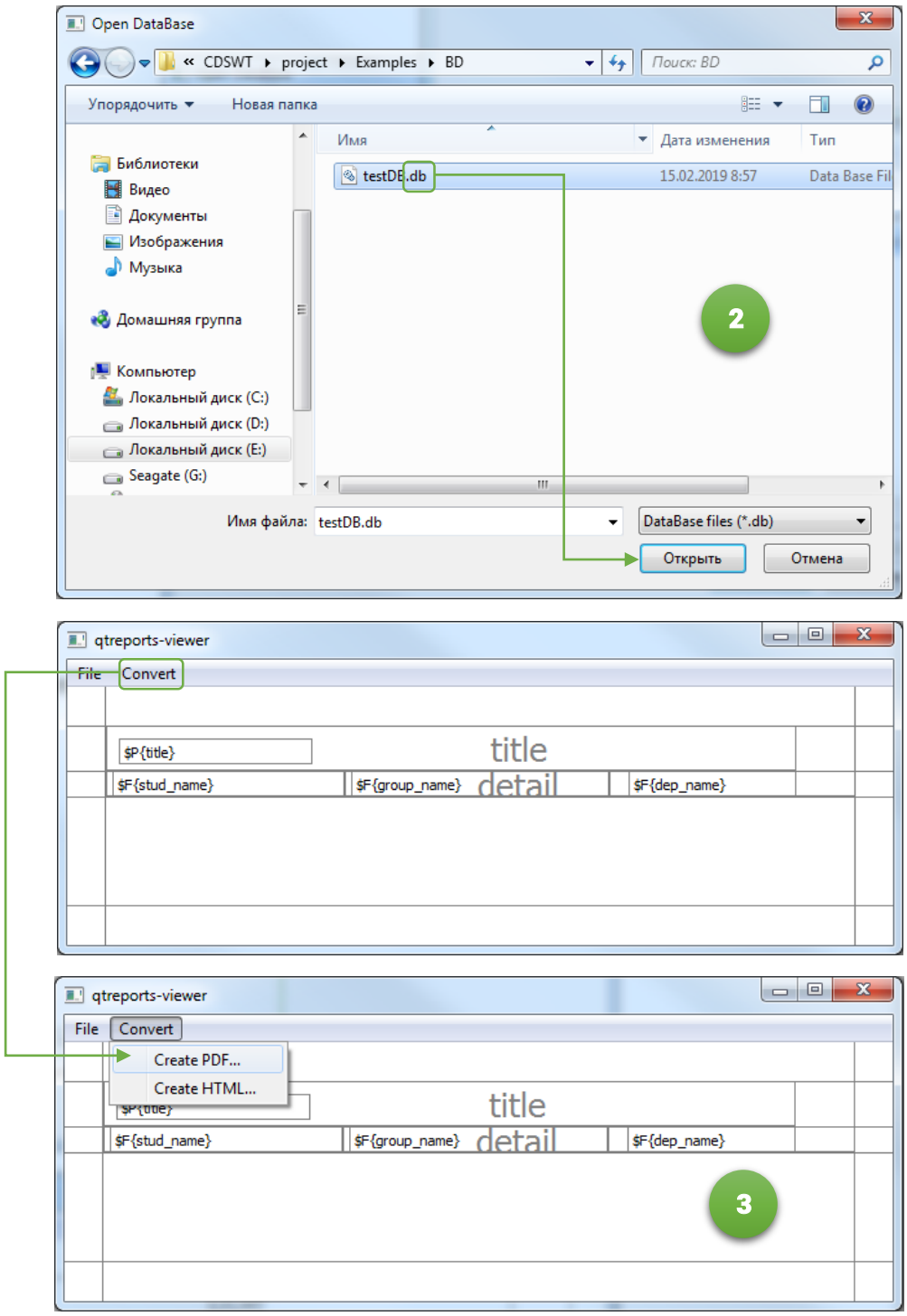
QtReports is a new report generator for Qt-applications, written using the Qt framework. QtReports allows to create complex layouts, containing diagrams, images, subreports, crosstables and many other things.

Data access occurs through the standard Qt class - QSqlDatabase, which allows you to work through drivers with different DBMS; also, data access can be received from XML-files. QtReports provides the ability to export reports in PDF and HTML formats.

1.2 User Interface

In the current version of QtReports (0.3.0), the window interface has only one module - QtReports Viewer, which has the following form:





1. After launching the QtReports Viewer – report template viewer, – you must select a file with the extension «.qreport».

2. Next, you need to select a database - a file with the extension «.db ».

3. Now you have the opportunity to export data from the selected database in accordance with the selected template in one of the provided formats: PDF or HTML.

The report designer in this version isn`t implemented yet.

1.3 Hardware Requirements

QtReports requires a 64-bit or 32-bit processor and at least 50MB of free hard disk space. The recommended amount of RAM is 2GB.

1.4 Software Requirements

Like any other Qt project, a Qt framework is required to build QtReports. To build QtReports, you need Qt Versions at least 5.8 with the MinGW 32bit compiler or MSVC 32 / 64bit compiler.

The current version of QtReports supports the most common operating systems:

- Windows 7 with 32 or 64 bits
- Linux with 32 or 64 bits

1.5 Accessing the Source Code

The latest version of the QtReports source code is available at https://github.com/PO31/QtReports. You can download and compile this source code, but it may contain new unpublished functions and errors, because it is under development. All information necessary for successful project assembly is in the next paragraph of this manual.

1.6 Building and Installing QtReports

Working with QtReports begins with downloading sources from the official repository:

https://github.com/PO-31/QtReports

1.6.1 Building under Qt

Requirements:

- Qt 5+
- gcc-5+/clang-3.4+/msvc14+

Step 1: Unpacking.

- Clone the project or download the latest release from repository: https://github.com/PO-31/QtReports/.
- Unzip the archive into the QtReports folder. The full path to the QtReports folder recommended to be with Latin characters.

Step 2: Building.

You can build a project in several ways, but if you need to specify the version of the library being assembled, you must set the QTREPORTS_VERSION environment variable.

Method 1. By means of QtCreator:

- Using program in the root folder “QtReports" it needs to open the project file "QtReports.pro".
- For building with statistics for “Coverage", set the value of the medium variable “BUILD_COVERAGE" on “TRUE"
- Next, click "Collect" (Hammer button) to build the entire project.
- If it needs to build only specific component, you need to click the RMB on the component project in the project browser and select "Build [Module_Name]".

Method 2. By means of Qmake:

- Start the console / terminal and go to the directory of the module we need
- Type qmake [-spec "Used_Compiler"] "CONFIG += Required_ Assembly_Types (release, debug, coverage, etc.) "Module_Project_Module.pro
- After generating the Makefile, call make [-jNumberOfThreads]

Step 3: Using.

- After building the library file can be found in the folder "build / lib", and the header files - "qtreports / include"
- To connect a compiled library to a Qmake project, specify “LIB + = -lqtreports" in the project file and, if necessary, “LIB + = Path_To_Library", “INCLUDEPATH + = Path_To_Header_Files".

1.6.2 Building under MS Visual Studio

Requirements:

- Microsoft Visial Studio 2015 (further MVS 2015)
- Qt 5.8.0 with comiler MVSC 2015 64-bit
- QtPackage or Qt Visual Studio tools for MVS 2015

Step 1: Installing Qt Package or Qt Visual Studio tools in MVS 2015.

- Launch the MVS 2015 development medium and test your Internet connection.
- Go "Service - Extensions and Updates ..."
- In the left part of the window, select the “Online" tab, and in the search bar (in the upper right corner) select “QtPackage" or “Qt Visual Studio tools".
- In the list of found components, select the one you need and download the installation will be performed automatically.
- Restart MVS 2015 to apply the changes.

Step 2: Configure the QtReports in MVS 2015.

- Go to “File - Open solution or project ...", specify the file QtReports.sln.
- Go "QT5 - Qt Options". Add the path to the compiler MSVC 2015 64-bit for Qt.
- Right-click on the QtReports solution in the “Solution Explorer window", select “Change Solution's Qt Version" and specify the compiler MSVC 2015 64-bit. For each project in the Qt Project Settings window, the Version variable should be set to MSVC 2015 64-bit, if not, then apply manually.
- Set the configuration of the solution “Release" and the platform “x64".

Step 3: Build the solution.

- Building projects should be in the sequence: qtreports, qtreports-tests, qtreportsviewer.

1.7 Report Structure in QtReports

The structure of the report is determined by means of a page divided into various horizontal sections, called «stripes» (tag «band»). When a report is associated with the data from which the output is generated, these sections (bars) are printed a certain number of times, in accordance with the rules established by the author of the report. For example, the “page header" tag is repeated at the beginning of each page, while the «detail» tag is repeated for each entry from the query results.

1.7.1 The Report Life Cycle

The life cycle of a report begins with the creation of its design. Designing a report means creating a kind of template, which is a page with empty space that can be filled with data. Some parts of the page are repeated cyclically, others are stretched across the width of the page, and so on.

The template is saved as a subtype of an XML file called QRXML (“QR" - QtReports). It contains all the basic information about the report layout, including complex formulas for performing calculations, an optional query for retrieving data from a data source, and other functions described in the following chapters.

The life cycle can be divided into two steps:

- Tasks performed at the design stage (developing and planning a report, as well as compiling the source .qreport file, .qrxml).
- Tasks that must be completed at run time (loading the qreport file, populating the report and

The main role of QtReports in the life cycle of a report is to develop this report and create a linked .qreport file, although it is possible to preview the result and export it to all supported formats. QtReports provides support for a wide range of data sources and allows users to test their own data sources. Thus, QtReports is a complete medium for developing and testing reports.

When designing a report using QtReports, you create a .qrxml file, which is an XML document containing the definition of the report layout. Before running the report generation, .qrxml must be converted to a .qreport file. Qreport files are what you need to load into the application to run a report.

Report execution is performed by transferring the .qreport file and data source to QtReports. There are several types of data sources: You can fill a .qreport file with data from a SQL query or from an XML file.

Thanks to the qreport file and data source, you can generate the final document in the format you need.

1.7.2 Tag Band

The Band tag is a paired block that is responsible for the general description of the elements (sections) of the report.

Optional Attributes:

- height - block height (default is 0).
- isSplitAllow - obsolete, replaced by splitType. This is a flag that indicates whether a block is allowed to burst upon expansion (values: true, false).
- splitType - sets the block breaking behavior (values: stretch, prevent, immediate).

Example of Use:

```
<detail>
    <band height="400">
      <staticText>
          <reportElement x="380" y="0" width="200" height="20" />
          <text><![CDATA[Test !!]]></text>
      </staticText>
      <textField>
          <reportElement x="220" y="20" width="100" height="200" />
          <textFieldExpression class="QString">
             <![CDATA[$P{title}]]>
          </textFieldExpression>
      </textField>
    </band>
</detail>
```

Child Elements:

- <staticText>
- <textField>
- <graficElement>

1.7.3 Tag CDATA

CDATA - this is the part of the element content that is marked for the parser as containing only character data or only parameters to be passed, not markup.

Begins with a sequence of <! [CDATA [and ends with characters]]>

Example of Use:

```
 `<sender>Jonh Smith</sender>`
```

The opening and closing tags "sender" will be interpreted as markup. However, if you write it like this:

```
`<![CDATA[<sender>John Smith</sender>]]>`
```

then this code will be interpreted as if it were written:

```
  &lt;sender&gt;John Smith&lt;/sender&gt;
```

Thus, sender tags will be perceived in the same way as “John Smith", that is, as text.

1.7.4 Tag Crosstab

Tag Crosstab – this is a special type of report item that combines data into a two-dimensional grid.

Optional Attributes:

- isRepeatColumnHeaders (true / false) - display column headings when moving to the next line.
- isRepeatRowHeaders (true / false) - display row headers when moving to the next row.
- columnBreakOffset - indicates the vertical space between sections of the crosstab when the crosstab exceeds the table width and two sections are displayed on the same page.
- runDirection - indicates the direction of filling the crosstab.

- ignoreWigth (true / false) - resolution of the table to infinitely expand / or to the left. Crosstabs will be displayed on top of the page border. Used in files where dynamic width can be used.

Example of Use:

```
<crosstab>
     <reportElement x="200" y="0" width="360" height="60"/>
     <rowGroup name="dep_name" width="130">
           <crosstabRowHeader>
                 <cellContents>
                       <textField>
                             <reportElement x="0" y="0" width="130" height="20"/>

     <textFieldExpression><![CDATA[$F{dep_name}]]></textFieldExpression>
                       </textField>
                 </cellContents>
           </crosstabRowHeader>
     </rowGroup>
     <columnGroup name="group_name" height="20">
           <crosstabColumnHeader>
                 <cellContents>
                       <textField>
                             <reportElement x="0" y="0" width="30" height="20"/>

     <textFieldExpression><![CDATA[$F{group_name}]]></textFieldExpression>
                       </textField>
                 </cellContents>
           </crosstabColumnHeader>
     </columnGroup>
     <crosstabCell width="30" height="20">
           <cellContents>
                 <textField>
                       <reportElement x="0" y="0" width="30" height="20"/>

     <textFieldExpression><![CDATA[$F{stud_name}]]></textFieldExpression>
                 </textField>
           </cellContents>
     </crosstabCell>
</crosstab>
```

Child Elements:

- textField

1.7.5 Tag Detail

The detail tag is a paired block that is the “body” of the report. This block contains basic information for each record from the data source. May contain several <band> blocks.

Examples of Use:

```
<detail>
    <band height="400">
        <staticText>
            <reportElement x="380" y="0" width="200" height="20" />
            <text><![CDATA[Тест !!]]></text>
        </staticText>
        <textField>
```

```
            <reportElement x="220" y="20" width="100" height="200" />
            <textFieldExpression class="QString">
                <![CDATA[$P{title}]]>
            </textFieldExpression>
        </textField>
    </band>
</detail>
```

Parent Elements: <report>

Child Elements: <band>

1.7.6 Tag Ellipse *(TBD)*

Tag Ellipse - definition of an ellipse object.

1.7.7 Tag Field

Tag <Field> – it is a data field that will store the values retrieved from the report data source. A report field is the only way to display data from a source in a template report, and to use this data in report expressions to get the desired result.

When using a SQL query in a report, you must ensure that the column for each field is obtained after the query is executed. The corresponding column must carry the same name and have the same data type as the field that displays it.

Required Attributes:

- name - field name (type - QString).
- class - class of field values (type - QString).
- name - element name attribute is required. This allows you to refer to the field in the report by name.
- class - the second attribute; defines the class name for field assignments. The default is QString, but can be changed to any class available at runtime.

Optional Attributes:

- fieldDescription – this additional text fragment can be very useful when implementing user data. For example, you can store a key or any information that may be needed in order to restore a field value from a user data source at runtime.

Example of Use:

EmployeeID	LastName	FirstName	HireDate
int 4	varchar 50	varchar 50	datetime 8

Report fields should be specified as follows:

```
<field name="EmployeeID" class="Integer"/>
<field name="LastName" class="String"/>
<field name="FirstName" class="String"/>
<field name="HireDate" class="Data"/>
<field name="PersonName" class="String" isForPromting="true">
<fieldDescription>PERSON NAME</fieldDescription></field>
```

1.7.8 Tag GraphicElement *(TBD)*

Tag <graphicElement> – implements the graphic components. It includes lines, triangles and ellipses.

Example of Use:

```
<rect>
    <reportElement mode="Opaque" x="5" y="60" width="782" height="80"
forecolor="#000000"/>
    <graphicElement pen="None" fill="Solid"/>
</rect>
```

1.7.9 Tag Group

Tag <group> – this is a flexible way to organize data in a report. They are a sequence of records that have something in common, for example, the value of a field. The report can be arbitrarily groups. The order of the groups declared in the report template is important because the groups are nested inside one another.

Attributes:

- name - required. The name uniquely identifies the group and can be used for other attributes when it is necessary to refer to a specific group in the report. The group name is mandatory and subject to the same naming scheme that is used for report parameters and variables.
- IsStartNewColumn - page break (true | false) default: "false".
- IsStartNewPage - column (column) break (true | false) default: "false". Sometimes you need to insert a page break or column break when a new group begins. In order for the engine to insert a page or column break every time a new group starts, you must specify Attributes IsStartNewPage or IsStartNewColumn, respectively.
- IsReprintHeaderOnEachPage (true | false) default: "false".

Notes:

Data grouping works as intended, only when the records in the data source are already ordered according to the group expression used in the report.

For example, if you want to group some products by country or city of manufacturer, the engine expects to find records in the data source already ordered by country and city.

If this is not the case, it may happen that the records related to a specific country or city will be in different parts of the received document, because the engine does not sort the data.

Example of Use:

```
<group name="group_name">
    <groupExpression class="QString">
        <![CDATA[$F{group_name}]]>
    </groupExpression>
    <groupHeader>
        <band height="40">
            <textField>
                <reportElement x="0" y="0" width="200" height="40" />
                <textFieldExpression class="QString">
                    <![CDATA[$F{group_name}]]>
                </textFieldExpression>
            </textField>
        </band>
```

```
    </groupHeader>
    <groupFooter>
        <band height="40">
            <textField>
                <reportElement x="0" y="0" width="200" height="40" />
                <textFieldExpression class="QString">
                    <![CDATA[$F{group_name}]]>
                </textFieldExpression>
            </textField>
        </band>
    </groupFooter>
</group>
```

Child Elements:

- <groupExpression> is the expression by which the grouping will be performed.
- <groupHeader> - group header - what will be printed before the first element.
- <groupFooter> - is what will be printed after the last element of the group.

1.7.10 Tag GroupExpression

Tag <groupExpression> – child tag of the <group> tag, which sets the field by which the grouping will occur. Mandatory tag when using grouping. Inside using CDATA, the field by which the grouping will occur is described.

Example of Use:

```
    <groupExpression>
        <![CDATA[$F{group_name}]]>
    </groupExpression>
```

1.7.11 Tag GroupFooter

Tag <groupFooter> – child tag of the <group> tag, which contains elements displayed at the end of the group. Content is described using the band tag.

Example of Use:

```
 <groupFooter>
   <band height="40">
     <textField>
       <reportElement x="0" y="0" width="100" height="30" />
       <textFieldExpression class="QString">
         <![CDATA[$F{group_name}]]>
       </textFieldExpression>
     </textField>
   </band>
 </groupFooter>
```

1.7.12 Tag GroupHeader

Tag <groupHeader> – child tag group tag, which contains the elements displayed at the beginning of the group. Content is described using the band tag.

Example of Use:

```
 <groupHeader>
```

```
     <band height="37">
       <textField>
         <reportElement x="33" y="0" width="100" height="30" />
         <textFieldExpression
class="QString"><![CDATA[$F{group_name}]]></textFieldExpression>
       </textField>
     </band>
 </groupHeader>
```

1.7.13 Tag Image

Tag <image> – can be used to insert bitmap images (such as GIF, PNG and JPEG images) in a report.

Attributes:

- scaleImage – indicates how the image should be rendered if its actual size does not fit the size of the image report item. This is because many images are loaded at run time, and there is no way to know their exact size when creating the report template. Possible values for this attribute:
  - Image clipping: if the image size is larger than the selected area, the image will not change its size, but will only be partially displayed (scaleImage = "Clip").
  - Image Forced Size: If the actual image does not fit the size specified for the image element that displays it, the image will stretch so that it fits into the designated output area. The image will be distorted if necessary (scaleImage = "FillFrame").
  - Preserving image proportions: if the actual image does not fit into the image element, it can be adapted to these dimensions, while maintaining its original undeformed proportions (scaleImage = "RetainShape").
  - Stretching the image while maintaining the width: The image can be stretched vertically to match the actual height of the image, while adjusting the width of the image element to match the actual width of the image (scaleImage = "RealSize").

  Notes:

  If the scaleImage type is "Clip" or "RetainShape", and the actual image is smaller than its specified size in the report template or does not have the same proportions, the image cannot occupy the entire space allocated to it in the report template. In such cases, you can set the position of the image within a predefined report space using the "Align" attribute and the VALIGN attribute, which determine the alignment of the image along the horizontal axis (Left, Center, Right) and vertical (Top, Middle, Bottom). By default, the image is aligned to the upper left inner border.

  All image elements have dynamic content, but the images in the report are static and do not necessarily come from a data source or from parameters. As a rule, they are loaded from files on disk and represent logos and other static resources. To display one image several times in a report (for example, a logo appears in the page header), you can cache the image for better performance. When the isUsingCache attribute is set to "TRUE", the reporting engine will attempt to recognize previously uploaded images using their specified source. This is a caching function for image elements whose expressions return objects of any type as an image source. The isUsingCache flag is set to "TRUE" by default for images that have String expressions, and as "FALSE" for all other types. The key used for the cache is the value of the source image expression; key comparisons are performed using the standard EQUALS method. As a result, for images that have an InputStream source with caching turned off, the input stream is read only once, and then the image will be taken from the cache. The

isUsingCache flag should not be set in cases where the image has a dynamic source (for example, the image is loaded from the binary field of the database for each row), because the images will accumulate in the cache and the fill will stop due to an out of memory error. Obviously, the flag should also not be set when one source is used to receive different images (for example, a URL that will return a different image every time it is available).

- isLazy – a flag that determines whether the image should be uploaded, and processed when the report is filled in or during export, if the image is not available at the time of filling. By default, this flag is set to "false". If set to "true", the image is saved during the filling instead of the image itself, and during the export process, the image will be loaded from the reading space from the path. For various reasons, the image may not be available when the handler tries to load it either when filling out a report or during export, especially if the image is loaded from some public URL. For this reason, you can set up a handler that handles missing images during report creation. The OnErrorType attribute for images allows this. It can take the following values:
  - Error: An exception is thrown if the handler cannot load the image (OnErrorType = "Error").
  - Blank: Any image-loading exception is ignored and nothing will be displayed in the created document (OnError Type = "Blank").
  - Icon: If the image does not load successfully, the handler will put a small icon in the document to indicate that the actual image is missing (OnErrorType = "Icon").
- evaluationTime – as is the case with text fields, you can postpone the calculation of an image expression, which by default is slow. This allows you to display in the document images that will be built or selected later in the process of filling the report. The evaluationTime attribute can have the following values:
  - Immediate evaluation: the image expression is calculated when the current range is filled (evaluationTime = "Now").
  - End of evaluation report: image expression is evaluated when the end of the report is reached (`evaluationTime =" Report ")
  - End of report page: image expression is calculated when the end of the current page is reached (evaluationTime = "Page")
  - End of evaluation column: upon reaching the end of the current column, an image expression is evaluated (`evaluationTime =" Column ")
  - End of assessment group: an image expression is calculated for a group by changing the evaluationGroup attribute (evaluatonTime = "Group")
  - Automatic evaluation: each variable involved in the expression of the image is evaluated at the time corresponding to the type of its reset. Fields are currently being evaluated (evaluationTime = "Auto").
- evaluationGroup – the group participates in the image evaluation process when the time evaluation attribute is set in the group.

1.7.14 Tag Line

Tag <line> - definition of a linear object (line).

Attributes:

- direction – lines are drawn as diagonals of a rectangle defined by the properties of the report element. This attribute indicates which of the two diagonals should be drawn.

Example of Use:

```
<line>
```

```
    <reportElement x="100" y="0" width="1" height="100" style="Arial_Normal" />
    <textElement textVAlignment="Bottom" textAlignment="Left">
        <font isBold="true"/>
    </textElement>
</line>
```

1.7.15 Tag Parameter *(TBD)*

Parameter is a link to objects that are transmitted during the report filling process to the report generator engine. Not required elements of the report.

Attributes:

- name – parameter name;
- class – values type of paremeter;

Example of Use:

Parameter declaration:

```
<parameter name="name_parameter" class="name_class" />
```

To use a parameter, the construction is used:

$ P {name_parameter}, where name_parameter is the name of the previously declared parameter.

An example of accessing a specific parameter:

```
<![CDATA[$P{name_parameter}]]>
```

1.7.16 Tag QueryString

Tag <queryString> – the template needed to define the SQL query for the report data, if this data is located in relational databases.

Example of Use:

There are three possible ways to use query parameters:

1. The first method: $P{paramName} - these parameters are used as normal parameters using the following syntax::

```
<queryString>
    <!CDATA[
    SELECT *FROM Orders WHERE orderID <= $P{MaxOrderID} ORDER BY ShipCountry
    ]]>
</queryString>
```

2. The second method *(TBD)*: $P!{paramName} – sometimes it is useful to use parameters to dynamically change a part of an SQL query or to transfer the entire SQL query as a parameter to the report filling procedures. In such cases, the syntax is slightly different, as shown in the example.:

```
<queryString>
  <![CDATA[
  SELECT *FROM $P!{MyTable} ORDER BY $P!{OrderByClause}
  ]]>
</queryString>
```

3. Third method (Not implemented): $ X {functionName, param1, param2, ...} - there are cases when a part of the query must be dynamically built, based on the value of the report parameter, with a part of the query containing the query text and parameter bindings. Such complex elements of a query are entered into a query using the syntax $ X {}. For example, if a report is received as a parameter for a list of countries and must filter orders based on this list, you should write a request of the following form:

```
<queryString>
  <![CDATA[
  SELECT *FROM Order WHERE $X{IN, ShipCountry, CountryList}
  ]]>
</queryString>
```

1.7.17 Tag Rect

Tag <rect> – definition of a rectangle object.

Attributes:

- radius – border radius.

1.7.18 Tag Report

Tag <report> – root of the report. Every report starts and ends with it.

Required Attributes:

- Name – report name.

Optional Attributes:

- PageWidth - report width, default is 595
- PageHeight - report height, default is 842
- Orientation - report sheet orientation, (values: Portrait, Landscape)

Example of Use:

```
<report>
    Report body
</report>
```

Child Elements:

- <Style>
- <Title>
- <Detail>
- <QueryString>
- <Field>

1.7.19 Tag ReportElement

Tag <reportElement> – the first element of each of the child elements of the Tag <band>. Determines how data is placed for this particular item (specifies the position and size of the item before which it is specified).

Required Attributes:

- x - x coordinate
- y is the y coordinate
- width - the width of the element
- height - the height of the element
- style – list of styles

Optional Attributes:

- stretchType - indicates how the current element is stretched when contained in the stretched band element. Values:
  - NoStretch (default): The current element is not stretched.
  - RelativeToTallestObject: The current element will be stretched, adapting to the tall objects in its group.
  - RelativeToBand: The current element will be stretched to match the height of the specific band element.
- positionType - indicates the position of the current element when the specific band element is stretched. Values:
  - Float: the current element will move depending on the size of the surrounding elements.
  - FixeRelativeToTop (default): the current element will maintain a fixed position relative to the top of the band element.
  - FixRelativeToBottom: The current element will maintain a fixed position relative to the bottom of the band element.
- mode - report elements can be transparent or opaque depending on the mode (transparent | opaque) value. The default values for this attribute depend on the type of report element. Graphic elements such as rectangles (<rectangle>) and lines (<line>) are not transparent by default, while images (<image>) are transparent. <staticTexts> and <textFields> are transparent by default, and therefore also report sub elements.

Example of Use:

```
<reportElement x="380" y="0" width="200" height="20" />
```

1.7.20 Tag StaticText

Tag <staticText> – permanent text that does not depend on any data sources.

Example of Use:

```
<staticText>
    <reportElement x="20" y="20" width="250" height="20" />
        <textElement textAlignment="Right" textVAlignment="Top" />
    <text><![CDATA[$P{i am staticText}]]></text>
</staticText>
```

Parent Elements: <band>

Child Elements:

- <reportElement/>
- <textElement/>
- <text></text>

1.7.21 Tag Style

Tag <style> – allows you to define a certain set of properties of elements once, and then use this set in any report block. The style is applied to the <reportElement> element by specifying the name of the style as the attribute style = "", otherwise the default style is applied.

Required Attributes:

- Name – unique style name

Optional Attributes:

- IsDefault - this style will be used by default (values: true, false; by default - false)
- FontSize - font size
- Forecolor - font color (values: black, blue, gray, green, red, yellow, while)
- FontName - font name
- IsBold - determines whether the text is bold (values: true, false)
- IsItalic - determines whether the text is "italic" (values: true, false)

Example of Use:

Styles declaration:

```
<style name="Regular" isDefault="true" fontSize="12" />
<style name="Emphasis" fontSize="12" isBold="tru2 />
```

Using:

```
<reportElement x="180" y="0" width="200" height="20" />
<text><![CDDATA[стиль по умолчанию]]></text>
<reportElement x="180" y="20" width="200" height="20" style="Emphasis" />
<text><![CDATA[стиль "Emphasis"]]></text>
```

Parent Elements: <report>

1.7.22 Tag Summary

Tag Summary – it is one of the section types. It is located once at the end of the report after the Detail section.

Example of Use:

```
     <summary>
    <band height="35">
          <textElement textAlignment="Center" textVAlignment="Middle">
                <staticText>
                      <reportElement x="10" y="10" width="150" height="20" />
                      <text><![CDATA[$P{summary}]]></text>
                </staticText>
          </textElement>
          <textField>
                <textElement textAlignment="Center" textVAlignment="Bottom">
                <reportElement x="310" y="0" width="140" height="40" />
                <textFieldExpression
class="QString"><![CDATA[$F{NManual}]]></textFieldExpression>
                </textElement>
          </textField>
          <rect>
                <reportElement x="0" y="0" width="480" height="30" />
```

```
                <graphicElement pen="Dotted" />
            </rect>
            <line>
                <reportElement x="170" y="10" width="20" height="40" />
            </line>
            <line>
                <reportElement x="200" y="30" width="20" height="-40" />
            </line>
    </band>
</summary>
```

Child Elements:

- textField

1.7.23 Tag Text

Tag <text> – determines the actual static text displayed in the report.

In the template below, we decided to add an XML CDATA section between the tags and </ text>. Although it is not strictly necessary in this case, it allows us to easily modify the text between these tags to include text that would prevent XML from being parsed.

Example of Use:

```
<text><![CDATA[$P{title}]]></text>
```

Parent Elements: <staticText>

1.7.24 Tag TextElement

Tag <textElement>

Optional Attributes:

- textAlignment - text alignment (values: Left, Center, Right, Justified)
- textVAlignment - vertical text alignment (values: Top, Middle, Bottom)

Example of Use:

```
<textElement textAlignment="Left" textVAlignment="Middle">
```

1.7.25 Tag TextField *(TBD)*

Tag <textField> – unlike static text elements that do not change the content of their text, text fields have an associated expression that is evaluated with each iteration in the data source to get the text content to be displayed.

Optional Attributes:

- pattern
- isBlankWhenNull - (values: true, false)
- isStretchWithOverflow (values: true, false; default is false)
- evaluationTime (values: Now, Report, Page, Column, Group, Band, Auto; default is Auto)

Example of Use:

```
<textField>
    <reportElement x="220" y="20" width="100" height="200" />
    <textFieldExpression class="QString">
        <![CDATA[$P{title}]]>
    </textFieldExpression>
</textField>
```

Parent Elements: <band>

Child Elements:

- <reportElement>
- <textElement>
- <textFieldExpression>

1.7.26 Tag TextFieldExpression

Tag <TextFieldExpression> – a text field can only return values for a limited range of the classes listed as follows:

- Boolean
- Byte
- Date
- Timestamp
- Time
- Double
- Float
- Integer
- Long
- Short
- BigDecimal
- Number
- QString (По умолчанию)

Attributes: class - class for text field values.

Example of Use:

```
<textFieldExpression class="QString">
    <![CDATA[$P{title}]]>
</textFieldExpression>
```

1.7.27 Tag Title

Tag <title> – report name is displayed once at the beginning of the report.

Example of Use:

```
<title>
    <band>
        <staticText>
            <reportElement x="100" y="16" width="100" height="20"/>
            <textElement verticalAlignment="Top" textAlignment="Right"/>
            <text><![CDATA[Title]]></text>
        </staticText>
    </band>
```

```
</title>
```

Parent Elements: <report>

Child Elements: <band>

1.7.28 Tag Variable

Tag <variable> – variables simplify a report template by highlighting an expression in one part, which is widely used throughout the report template. They can perform calculations based on the appropriate formula (expression). In its expression (or formula), a variable may use other variables, fields, or parameters. Variables are calculated / incremented with each record from the data source in the order in which they are declared.

Attributes:

- name – it is the name of the variable. It is a required attribute and allows you to refer to a variable by this name.
- class – it is the data type to which the value of the variable belongs (string - by default).
- resetType - the frequency of setting the initial value (None | Report | Page | Column | Group). Value options:
  - None - never initialized by initial value. Contains the value obtained by evaluating the variable expression.
  - Report - initialized with the initial value (<initialValueExpression>) once at the beginning of the report filling (by Default).
  - Page - initialized at the beginning of each page.
  - Column - initialized at the beginning of each new column.
  - Group - is initialized again each time a new group is defined.
- resetGroup - if resetGroup is used, it contains the name of the group and works only in combination with the resetType attribute, the value of which will be resetType = Group.
- incrementType - variable frequency increment (None | Report | Page | Column | Group). By default, the variable is incremented with each record from the data source. Value options:
  - None - the variable is incremented with each entry (by Default).
  - Report - the variable is never incremented.
  - Page - variable is incremented with each new page.
  - Column - the variable is incremented with each new column.
  - Group - incremented each time a new group is defined.
- incrementGroup - if incrementGroup is used, it contains the name of the group and works only in combination with the incrementType attribute, the value of which will be incrementType = Group.
- calculation is an aggregate function (Count | Sum | Average | Lowest | Highest).

Notes:

An expression defining the value of a variable:

```
<variableExpression><![CDATA[$F{absence}]]></variableExpression>
```

Tag which giving the initial value. (may not be used):

```
<initialValuesExpression><![CDATA[0]]></initialValuesExpression>
```

Built-in variables:

- PAGE_NUMBER - contains the current page number. At the end of the report, it contains the total number of pages in the document. Can be used to display the current page and page count at the same time.
- COLUMN_NUMBER - contains the number of the current column. The variable counts the number of columns for each page.
- REPORT_COUNT - contains the total number of records processed in the report.
- PAGE_COUNT - contains the number of records processed in the process of filling the current page.
- COLUMN_COUNT - the number of records processed during the generation of the current column.

Example of Use:

```
<variable name="sumAbsenceByDiscipline" class="Integer" resetType="Group"
resetGroup="Discipline" calculation="Sum">
    <variableExpression><![CDATA[$F{absence}]]></variableExpression>
    <initialValueExpression><![CDATA[0]]></initialValueExpression>
</variable>
<textField>
    <reportElement x="396" y="0" width="159" height="34"/>
    <textElement verticalAlignment="Middle"/>
<textFieldExpression><![CDATA[$V{sumAbsenceByDiscipline}]]></textFieldExpression>
</textField>`
```

1.7.29 Specifying Report Properties *(TBD)*

1.7.30 Columns properties *(TBD)*

1.7.31 Advanced Options *(TBD)*

Chapter 2 Basic Concepts of QtReports

This chapter illustrates the basic concepts of QtReports for a better understanding of how this library works for generating reports.

The QtReports API, XML syntax for defining reports, and all the details of using the library in your own programs are described in this guide. Other information and examples are available at: https://github.com/PO-31/QtReports.

QtReports is published under the MIT license, which allows individuals who have received a copy of this software and accompanying software to use it for free without restriction, including the unlimited right to use, copy, modify, merge, publish, distribute, sublicense and / or sell copies, provided.

The above copyright notice and these terms and conditions must be included in all copies or relevant parts of this Software.

This software is provided "as is" without warranty of any kind, either expressed or implied, including warranties of merchantability, fitness for a particular purpose and non-infringement, but not limited to. In no event shall the authors or rights holders be liable for any claims, damages or other requirements, including, under a contract, delict or other situation arising from the use of software or other actions with the software.

2.1 Files .QREPORT и .QRXML

QtReports defines a report by means of an XML file. The .qrxml file and the .qreport file consist of a set of sections, some of which relate to physical characteristics, such as page size, field layout, and line height; and some of them relate to the definition of logical characteristics, such as the declaration of parameters and variables, as well as the creation of queries for selecting data.

Example of .qrxml file structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<report name="sample_report" orientation="Landscape">
  <style name="Arial_Normal" isDefault="true" fontName="Arial"
    fontSize="12" pdfFontName="c:\tahoma.ttf" pdfEncoding="Cp1251"
    isPdfEmbedded="false" />
  <queryString>
    <![CDATA[ select idImg, nameImg, image from images; ]]>
  </queryString>
  <field name="idImg" class="QString" />
  <field name="nameImg" class="QString" />
  <field name="image" class="QString" />
  <group name="nameImg">
    <groupExpression>
      <![CDATA[$F{nameImg}]]>
    </groupExpression>
    <groupHeader>
      <band height="40">
        <textField>
          <reportElement x="0" y="10" width="50" height="30"/>
          <textFieldExpression
class="QImage"><![CDATA[$F{nameImg}]]></textFieldExpression>
        </textField>
```

```
      </band>
    </groupHeader>
    <groupFooter>
      <band height="10">
      </band>
    </groupFooter>
  </group>
  <title>
    <band height="35">
      <staticText>
        <reportElement x="10" y="10" width="150" height="20" />
        <text><![CDATA[$P{title}]]></text>
      </staticText>
      <ellipse>
        <reportElement x="0" y="0" width="535" height="35" />
      </ellipse>
    </band>
  </title>
  <detail>
    <band height="200">
      <rect>
        <reportElement x="0" y="0" width="535" height="200" />
      </rect>
      <textField>
        <reportElement x="5" y="0" width="20" height="200" />
        <textFieldExpression
class="QString"><![CDATA[$F{idImg}]]></textFieldExpression>
      </textField>
      <line>
        <reportElement x="25" y="0" width="1" height="200" />
      </line>
      <textField>
        <reportElement x="30" y="0" width="140" height="200" />
        <textFieldExpression
class="QString"><![CDATA[$F{nameImg}]]></textFieldExpression>
      </textField>
      <line>
        <reportElement x="170" y="0" width="1" height="200" />
      </line>
      <image>
        <reportElement x="170" y="0" width="425" height="200" />
        <imageExpression class="QString"><![CDATA[$F{image}]]></imageExpression>
      </image>
    </band>
  </detail>
</report>
```

Example of .qrxml file structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<report name="sample_report">
  <style name="Arial_Normal" isDefault="true" fontName="Arial"
    fontSize="12" pdfFontName="c:\tahoma.ttf" pdfEncoding="Cp1251"
    isPdfEmbedded="false" />
  <queryString>
    <![CDATA[ select group_name, stud_name from groups_t NATURAL JOIN students_t
ORDER BY group_name]]>
  </queryString>
```

```
  <field name="group_name" class="QString" />
  <field name="stud_name" class="QString" />
<group name="group_name">
        <groupExpression class="QString">
            <![CDATA[$F{group_name}]]>
        </groupExpression>
  <groupHeader>
      <band height="37">
        <textField>
          <reportElement x="33" y="0" width="100" height="30" />
          <textFieldExpression
class="QString"><![CDATA[$F{group_name}]]></textFieldExpression>
        </textField>
      </band>
  </groupHeader>
  <groupFooter>
    <band height="40">
      <textField>
        <reportElement x="0" y="0" width="100" height="30" />
        <textFieldExpression class="QString">
          <![CDATA[$F{group_name}]]>
        </textFieldExpression>
      </textField>
    </band>
  </groupFooter>
</group>
  <detail>
    <band height="40">
      <staticText>
        <reportElement x="380" y="0" width="200" height="20" />
        <text><![CDATA[Тест !!]]></text>
      </staticText>
      <textField>
        <reportElement x="220" y="20" width="100" height="20" />
        <textFieldExpression class="QString">
          <![CDATA[$P{title}]]>
        </textFieldExpression>
      </textField>
      <textField>
        <reportElement x="51" y="0" width="200" height="20" />
        <textFieldExpression class="QString">
          <![CDATA[$F{group_name}]]>
        </textFieldExpression>
      </textField>
      <textField>
        <reportElement x="252" y="0" width="200" height="20" />
        <textFieldExpression class="QString">
          <![CDATA[$F{stud_name}]]>
        </textFieldExpression>
      </textField>
    </band>
  </detail>
</report>
```

2.2 Data Sources and Print Formats

QtReports works with all DBMS supported by the Qt framework. PDF and HTML formats are available for export and print.

2.3 Compatibility Between Versions

2.4 Expressions

2.4.1 The Type of an Expression

2.4.3 Using an If-Else Construct in an Expression

2.5 A simple program

The following example code illustrates how to use QtReports in Qt programgs.

.pro file:

```
QT += core gui sql widgets printsupport

TARGET = qtreports-viewer
TEMPLATE = app
CONFIG += c++14
INCLUDEPATH += ../qtreports/include/
LIBS   += -L"$$PWD"/../build/lib/
LIBS   += -lqtreports
DESTDIR = "$$PWD"/../build/viewer/

SOURCES += src/main.cpp

QMAKE_CXXFLAGS += -std=c++14

message("Using LIB: $$LIBS")
message("Using spec: $$QMAKESPEC")
message("Compiler: $$QMAKE_CXX")
```

.cpp file:

```
#include <QApplication>
#include <QMessageBox>
#include <QMainWindow>
#include <QAction>
#include <QMenu>
#include <QMenuBar>
#include <QFileDialog>
#include <QDir>
#include <QSqlDatabase>
#include <QSqlError>
#include <qtreports/engine.hpp>

#ifdef WIN32
#include <Windows.h>
#endif

static QMainWindow* mainWindow = nullptr;
static QMenuBar* menuBar = nullptr;
```

```
static QMenu* fileMenu = nullptr;
static QMenu* convertMenu = nullptr;

static QAction* printAction = nullptr;
static QAction* createPdfAction = nullptr;
static QAction* createHtmlAction = nullptr;
static QAction* selectDatabaseAction = nullptr;
static QAction* closeAction = nullptr;
static QAction* openAction = nullptr;
static QAction* exitAction = nullptr;

static QSharedPointer<QWidget> layout;

static qtreports::Engine* engine = nullptr;

void showError(const QString& text);
void closeReport();
void openReport(const QString& reportPath);
void openReport();
void openDatabase(const QString& dbPath);
void openDatabase();

void showError(const QString& text)
{
    QMessageBox::warning(nullptr, "Error: ", text);
}

void closeReport()
{
    engine->close();
    layout.clear();
    if (mainWindow->centralWidget() != nullptr) {
        delete mainWindow->centralWidget();
    }
    convertMenu->setEnabled(false);
    printAction->setEnabled(false);
    closeAction->setEnabled(false);
    selectDatabaseAction->setEnabled(false);
    mainWindow->setWindowTitle("QtReports viewer");
}

void openReport(const QString& reportPath)
{
    closeReport();

    bool result = engine->open(reportPath);
    closeAction->setEnabled(result);
    selectDatabaseAction->setEnabled(result);
    if (!result) {
        showError(engine->getLastError());
        return;
    }
    QFileInfo fileInfo(reportPath);
    mainWindow->setWindowTitle("QtReports viewer - " + fileInfo.fileName());
    layout = engine->createLayout();
    if (layout.isNull()) {
        showError("Widget is empty");
        engine->close();
        return;
```

```
    }
    mainWindow->setCentralWidget(layout.data());
    mainWindow->resize(layout->size());
}

void openReport()
{
    auto reportPath = QFileDialog::getOpenFileName(
        mainWindow,
        QObject::tr("Open QReport"),
        QString(),
        QObject::tr("QReport Files (*.qreport *.qrxml);;All Files (*.*)")
    );
    if (reportPath.isEmpty()) {
        return;
    }
    openReport(reportPath);
}

void openDatabase(const QString& dbPath)
{
    auto db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName(dbPath);
    if (!db.open()) {
        showError("Can not open database. Database error: " +
db.lastError().text());
        return;
    }
    bool result = engine->setConnection(db);
    convertMenu->setEnabled(result);
    printAction->setEnabled(result);
    if (!result) {
        showError(engine->getLastError());
        return;
    }

    //setting report parameters
    QMap <QString, QVariant> map;
    map["title"] = "TITLE";
    map["idPlan"] = "2";
    map["summary"] = "SUMMARY";

    if (!engine->setParameters(map)) {
        showError(engine->getLastError());
    }
}

void openDatabase()
{
    auto dbPath = QFileDialog::getOpenFileName(
        mainWindow,
        QObject::tr("Open DataBase"),
        QString(),
        QObject::tr("DataBase files (*.db);;All Files (*.*)")
    );
    if (dbPath.isEmpty()) {
        return;
    }
    openDatabase(dbPath);
```

```
}

int main(int argc, char *argv[]) {
    QApplication a(argc,argv);

    mainWindow = new QMainWindow();
    mainWindow->setWindowTitle("QtReports viewer");
    mainWindow->resize(800, 600);
    menuBar = new QMenuBar(mainWindow);
    fileMenu = new QMenu("File", menuBar);
    convertMenu = new QMenu("Convert", menuBar);

//For Debug in Windows
#ifdef WIN32
    AllocConsole();
    freopen("CONOUT$", "w", stdout);
    freopen("CONOUT$", "w", stderr);
#endif

    printAction = new QAction(QObject::tr("&Print..."), mainWindow);
    printAction->setShortcuts(QKeySequence::Print);
    printAction->setStatusTip(QObject::tr("Print current report"));
    QObject::connect(printAction, &QAction::triggered, [&]() {
        if (!engine->print()) {
            showError(engine->getLastError());
        }
    });

    createPdfAction = new QAction(QObject::tr("&Create PDF..."), mainWindow);
    createPdfAction->setStatusTip(QObject::tr("Create PDF from current report"));
    QObject::connect(createPdfAction, &QAction::triggered, [&]() {
        auto file = QFileDialog::getSaveFileName(
            mainWindow,
            QObject::tr("Save as PDF"),
            QString(),
            QObject::tr("PDF Files (*.pdf)")
        );
        if (file.isEmpty()) {
            return;
        }
        if (!engine->createPDF(file)) {
            showError(engine->getLastError());
            return;
        }
    });

    createHtmlAction = new QAction(QObject::tr("&Create HTML..."), mainWindow);
    createHtmlAction->setStatusTip( QObject::tr("Create HTML from current
report"));
    QObject::connect(createHtmlAction, &QAction::triggered, [&]() {
        auto file = QFileDialog::getSaveFileName(
            mainWindow,
            QObject::tr("Save as HTML"),
            QString(),
            QObject::tr("HTML Files (*.html *.htm)")
        );
        if (file.isEmpty()) {
            return;
        }
```

```
        bool result = engine->createHTML(file);
        if (!result) {
            showError(engine->getLastError());
            return;
        }
    });

    selectDatabaseAction = new QAction(QObject::tr("&Select database..."),
mainWindow);
    selectDatabaseAction->setStatusTip(QObject::tr("Select database file"));
    QObject::connect(selectDatabaseAction, &QAction::triggered, [&]() {
openDatabase(); });

    closeAction = new QAction(QObject::tr("&Close report"), mainWindow);
    closeAction->setShortcuts(QKeySequence::Close);
    closeAction->setStatusTip(QObject::tr("Close current report"));
    QObject::connect(closeAction, &QAction::triggered, [&]() { closeReport(); });

    openAction = new QAction(QObject::tr("&Open report..."), mainWindow);
    openAction->setShortcuts(QKeySequence::Open);
    openAction->setStatusTip(QObject::tr( "Open an existing report file"));
    QObject::connect(openAction, &QAction::triggered, [ & ]() { openReport(); });

    exitAction = new QAction(QObject::tr("&Exit"), mainWindow);
    exitAction->setShortcuts( QKeySequence::Quit );
    QObject::connect(exitAction, &QAction::triggered, mainWindow,
&QMainWindow::close);

    fileMenu->addActions({ openAction, selectDatabaseAction, printAction,
closeAction, exitAction });
    convertMenu->addActions({ createPdfAction, createHtmlAction });
    menuBar->addMenu(fileMenu);
    menuBar->addMenu(convertMenu);
    mainWindow->setMenuBar(menuBar);

    engine = new qtreports::Engine();
    closeReport();

    if (argc > 1) {
        openReport(argv[1]);
    }
    if (argc > 2) {
        openDatabase(argv[2]);
    }
    mainWindow->show();

    return a.exec();
}
```

Chapter 3 Creating a Simple Report

3.1 Creating a New Report

3.2 Adding and Deleting Report Elements

3.2.1 Adding Fields to a Report

3.2.2 Deleting Fields

3.2.3 Adding other Elements

3.3 Previewing a Report

3.4 Creating a Project Folder

Chapter 4 Working with Fields

4.1 Understanding Fields

4.2 Registration of Fields from a SQL Query

4.3 Fields and Textfields

Chapter 5 Reports Templates

5.1 Template Structure

5.2 Creating and Customizing Templates

5.2.1 Creating a New Template

5.2.2 Customizing a Template

5.3 Saving Template

5.3.1 Creating a Template Directory

5.3.2 Exporting a Template

5.4 Adding Template to QtReports

Chapter 6 Using Parameters

6.1 Managing Parameters

6.2 Default Parameters

6.3 Using Parameters in Queries

6.3.1 Using Parameters in a SQL Query

In QtReports, it is possible to use query parameters ($P{paramName}). The example for use is as below:

```
<queryString>
    <!CDATA[
    SELECT *FROM Orders WHERE orderID <= $P{MaxOrderID} ORDER BY ShipCountry
    ]]>
</queryString>
```

6.3.2 Using Parameters with Null Values

6.3.3 Relative Dates

6.3.4 Passing Parameters from a Program

In QtReports it is possible to pass parameter from program by means of method setParameters(QMap map). The example for use is below:

```
qtreports::Engine engine;
QVERIFY2( engine.open( reportPath ), engine.getLastError().toStdString().c_str()
);
QMap < QString, QVariant > map;
map[ "title" ] = "Best Title in World";
map[ "param1" ] = "Best param1 in World";
qDebug() << endl << "Used map: " << map;
QVERIFY2( engine.setParameters( map ), engine.getLastError().toStdString().c_str()
);
```

Chapter 7 Variables

7.1 Defining a New Variable or Editing an Existing One

7.2 Base Properties of a Variable

7.3 Other Properties of a Variable

7.3.1 Evaluation Time

7.3.2 Calculation Function

7.3.3 Increment Type

7.3.4 Reset Type

7.4 Built-In Variables