Report-engine tutorial

Contents

[What is report-engine? 3](#_Toc336018512)

[How to build report-engine? 4](#_Toc336018513)

[Flat Reports 5](#_Toc336018514)

[Your first report 5](#_Toc336018515)

[Report Input 5](#_Toc336018516)

[Report Output 5](#_Toc336018517)

[Report Configuration 5](#_Toc336018518)

[Totals and aggregations 5](#_Toc336018519)

[Pivot Tables /Crosstab reports 6](#_Toc336018520)

[Your first Pivot table report 6](#_Toc336018521)

[The Input 6](#_Toc336018522)

[The Output 6](#_Toc336018523)

[Configuration 6](#_Toc336018524)

[Totals and aggregations 6](#_Toc336018525)

[Advanced features 7](#_Toc336018526)

[Spring integration 7](#_Toc336018527)

[Writing a custom input for your reports 7](#_Toc336018528)

[Writing a custom output for your reports 7](#_Toc336018529)

[FAQ 8](#_Toc336018530)

[Useful links 9](#_Toc336018531)

# What is report-engine?

Report Engine is a set of JAVA classes for reports and pivot tables with support for groupings, totals/subtotals, aggregation. It accepts input from text files, databases, excel or you can write your custom input and exports the report in a multitude of formats (HTML, RTF, PDF, TXT, SVG etc.)

# How to build report-engine?

## Using Maven

* First of all you need Apache Maven installed on your computer. If you don’t have it installed follow the instructions [here](http://maven.apache.org/download.html).
* Add reportengine dependency to your project

<dependency>

<groupId>net.sf.reportengine</groupId>

<artifactId>reportengine</artifactId>

<version>0.4.1</version>

</dependency>

# Flat Reports

## Your first report

Each report needs three elements: input, column configuration and output. Let’s have a look at the report below:

FlatReport flatReport = **new** FlatReport();

//input configuration

IReportInput input = **new** StreamReportInput(**new** FileInputStream("input.txt"));

flatReport.setIn(input);

//output configuration

IReportOutput output = **new** ExcelOutput(**new** FileOutputStream("output.xls"));

flatReport.setOut(output);

//columns configuration

flatReport.addDataColumn(**new** DefaultDataColumn("Country", 0));

flatReport.addDataColumn(**new** DefaultDataColumn("City", 1));

flatReport.addDataColumn(**new** DefaultDataColumn("Population", 2));

//start execution

flatReport.execute();

## Report Input

The main input classes are:

* *StreamReportInput* - handles input from streams (any kind) and reads data columns separated by a specific user-defined separator (comma, tab, etc.)

IReportInput reportInput = new StreamReportInput(

new FileInputStream("c:\commaSeparated.csv"),",");

flatReport.setIn(input);

…

* *DbQueryReportInput* - executes a query and takes the result as input for your reports

1. *For an existing database connection*

java.sql.Connection dbConnection = ...

DbQueryReportInput dbReportInput = new DbQueryReportInput();

dbReportInput.setConnection(connection);

dbReportInput.setSqlStatement("select id, country, region, city, population from DB\_POPULATION\_TABLE ");

1. Or if you don’t have the connection, report-engine can create one for you given the right parameters:

DbQueryReportInput dbReportInput = new DbQueryReportInput();

dbReportInput.setDbConnString("jdbc:hsqldb:mem:countriesDB");

dbReportInput.setDbDriverClass("org.hsqldb.jdbcDriver");

dbReportInput.setDbUser("sa");

dbReportInput.setDbPassword("secret");

dbReportInput.setSqlStatement("select id, country, region, city, population from DB\_POPULATION\_TABLE");

* *MemoryReportInput* - takes an array of objects as input

Object[][] REPORT\_DATA = new Object[][]{

new String[]{"a","b","c","d"},

new String[]{"1","2","3","4"},

new String[]{"x","y","z","t"}

}

IReportInput reportInput = new MemoryReportInput(REPORT\_DATA);

* If these classes don’t cover your needs you can always write your own input by implementing the *IReportInput* interface

## Report Output

The predefined output formats for your reports are:

* *HtmlReportOutput* - fast html output

HtmlOuputhtmlOut **= new** HtmlOutput(**new** FileOutputStream("employees.html")

* *ExcelReportOutput* – creates an excel output

ExcelOuput output = **new** ExcelOutput(**new** FileOuputStream("employees.xls"));

* *StaxReportOutput* - xml output based on STax (streaming xml) technology
* *XsltReportOutput -* output based on an XSLT template - can result in HTML, TXT, SVG, etc.
* *XslFoReportOutput* - output based on XSL-FO framework - can result in PDF, PNG, TXT, ghostscript. Actually, everything supported by [apache fop project](http://xmlgraphics.apache.org/fop/trunk/output.html).

XslFoOutput pdfOutput = **new** XslFoOutput(

**new** FileOutputStream("employees.pdf"));

XslFoOutput pngOutput = **new** XslFoOutput(

**new** FileOutputStream("employees.png"), MimeConstants.*MIME\_PNG*);

* *XmlDOMReportOutput* - xml output based on DOM
* Of course you can always write your own report output by implementing the *IReportOutput* interface

## Report Columns Configuration

There are two kinds of columns: data columns and group columns.

### Data columns

Data columns are normal report columns displaying formatted data and totals. For each column there are a few parameters to set: header, values displayed, calculator-if totals are needed … and that’s about it.

What is the header? It’s the string that will appear in the column header section of the report. In the example below the **Year, Month, Amount** are column headers.

|  |  |  |
| --- | --- | --- |
| **Year** | **Month** | **Amount** |
| 2011 | August | 500 |
| 2011 | Sept. | 300 |

#### What is the input column index?

This is the index of the Input column. Using report-engine you can use a different position for your column in the final report. For instance, in the sample above, I may decide to show the Year column as the third in the final report and show the second (Month) as the first in the report. To do that I will add my columns in the following order:

//show month first with data from column 1 (second input column)

report.addDataColumn( column month having input column index = 1)

//show amount second with data from column 2 (third input column)

report.addDataColumn(column amount having input column index = 2)

//show the year last with data from the column 0 (first)

report.addDataColumn(column year having input column index = 0)

As you can see above, report-engine uses column indexes starting from zero, so zero is our first input column

Attention: For report-engine API the order in which you define your columns is very important because **it defines the output order of your columns**.

What is the calculator?

The calculator compiles to the values of the column in order to get a SUM or an Average or whatever computation comes to your mind.

#### How to set these properties to a column?

Pretty easy:

* by using the setter methods

DefaultDataColumn column = **new** DefaultDataColumn();

column.setHeader("Month");

column.setInputColumnIndex(0);

column.setCalculator(Calculator.*SUM*);

* or by using the constructor

DefaultDataColumn column = **new** DefaultDataColumn("Month",0,Calculator.*SUM*);

DefaultDataColumn column = **new** DefaultDataColumn("Month",0);

DefaultDataColumn column = **new** DefaultDataColumn(0);

### Your first report

It’s now time to build our first report: Expenses report.

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**import** java.io.InputStream;

**import** net.sf.reportengine.FlatReport;

**import** net.sf.reportengine.config.DefaultDataColumn;

**import** net.sf.reportengine.config.IDataColumn;

**import** net.sf.reportengine.in.StreamReportInput;

**import** net.sf.reportengine.out.HtmlOutput;

/\*\*

\* this is my first report

\*

\*/

**public** **class** FirstReport {

/\*\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args) {

**try** {

FlatReport flatReport = **new** FlatReport();

flatReport.setReportTitle("Mothly Expenses report");

//the input

InputStream fileStream = **new** FileInputStream("expenses.csv");

StreamReportInput reportInput = **new** StreamReportInput(fileStream,",");

flatReport.setIn(reportInput);

//the output

HtmlOutput output = **new** HtmlOutput(**new** FileOutputStream("expenses.html"));

flatReport.setOut(output);

//data columns configuration

flatReport.addDataColumn(**new** DefaultDataColumn("Month", 0));

flatReport.addDataColumn(**new** DefaultDataColumn("Spent on", 1));

flatReport.addDataColumn(**new** DefaultDataColumn("Amount", 2));

flatReport.execute();

} **catch** (FileNotFoundException e) {

e.printStackTrace();

}

}

}

### The result should be:

|  |  |  |
| --- | --- | --- |
| My first expenses report | | |
| **Month** | **Spent on** | **Amount** |
| August | food | 500 |
| August | transportation | 300 |
| September | food | 567 |
| September | transportation | 154 |
| September | dinner | 200 |

### Group columns

Group columns are helpful when displaying totals on data columns. At each change in the values of a group column the totals are displayed. One of the differences between group columns and data columns is the possibility to add totals. On group column you cannot add totals, they are only helpful at displaying them.

Let’s check the following example: My list of monthly expenses

|  |  |  |  |
| --- | --- | --- | --- |
| August |  | food | 500$ |
| August |  | transportation | 300$ |
| September |  | food | 567$ |
| September |  | transportation | 154$ |
| September |  | entertainment | 200$ |

If we declare the first column as a group column then report-engine will make sure to display totals, averages (or whatever you’ve set ) at each change in the values of the first column… something like

|  |  |  |  |
| --- | --- | --- | --- |
| August |  | food | 500$ |
| August |  | transportation | 300$ |
| **Total August** |  |  | **800$** |
| September |  | food | 567$ |
| September |  | transportation | 154$ |
| September |  | entertainment | 200$ |
| **Total September** |  |  | **921$** |

### Your first report containing a group Column

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**import** java.io.InputStream;

**import** net.sf.reportengine.FlatReport;

**import** net.sf.reportengine.config.DefaultDataColumn;

**import** net.sf.reportengine.config.DefaultGroupColumn;

**import** net.sf.reportengine.config.IDataColumn;

**import** net.sf.reportengine.config.IGroupColumn;

**import** net.sf.reportengine.core.calc.Calculator;

**import** net.sf.reportengine.in.StreamReportInput;

**import** net.sf.reportengine.out.HtmlOutput;

/\*\*

\* The first report containing a group column.

\*

\* The month column is declared as a group column so

\* after each change in this column the totals will

\* be displayed.

\*/

**public** **class** FirstGroupReport {

**public** **static** **void** main(String[] args) {

**try** {

FlatReport flatReport = **new** FlatReport();

flatReport.setShowTotals(**true**);

flatReport.setShowGrandTotal(**true**);

flatReport.setReportTitle("Mothly Expenses");

//define the input

InputStream fileInput = **new** FileInputStream("expenses.csv");

StreamReportInput reportInput = **new** StreamReportInput(fileInput,",");

flatReport.setIn(reportInput);

//define the output

HtmlOutput output = **new** HtmlOutput(**new** FileOutputStream("xpenses.html"));

flatReport.setOut(output);

//group column configuration

flatReport.addGroupColumn(**new** DefaultGroupColumn("Month", 0, 0));

//data columns configuration

flatReport.addDataColumn(**new** DefaultDataColumn("Spent on",1));

flatReport.addDataColumn(**new** DefaultDataColumn("Amount",2,Calculator.*SUM*));

//start executing the report

flatReport.execute();

} **catch** (FileNotFoundException e) {

e.printStackTrace();

}

}

}

In the example above, let me draw your attention on the following lines of code:

//group column configuration

flatReport.addGroupColumn(**new** DefaultGroupColumn("Month", 0, 0));

This actually declares the Month column as a group column, but now, let’s look how the totals are calculated:

//data columns configuration

flatReport.addDataColumn(**new** DefaultDataColumn("Spent on",1));

**flatReport.addDataColumn(new DefaultDataColumn("Amount",2,Calculator.*SUM*));**

As you can see the third column, besides the header and the index of the input column, has a Calculator.SUM attached. This is to tell report-engine that all values on this column will be summed.

It’s time to see the attributes of a group column: header, input column index and group level. The first two are exactly as the data column’s header and input column index. The third (group level) tells report-engine that among group columns this has the highest priority (priority 0). More on this later…

### More on totals and groupings

In the previous example we’ve used SUM calculator for the third column, but report-engine has more: AVG, MIN, MAX, COUNT, FIRST, LAST. If you need anything else you can also write your own calculator.

Now, let’s see a more complex example: A report having 2 group columns and 2 columns having totals.

Let’s assume I have the list of expenses over the years:

2011 August food 500

2011 August transportation 300

2011 September food 567

2011 September transportation 154

2011 September entertainment 200

2012 January food 205

2012 January transportation 100

2012 February food 301

2012 March entertainment 302

…and I want to build a report that will show the totals on the last column at each change in the year and month. For this I will declare in my report the first and the second columns (year and month) as group columns and I will add a SUM calculator on the last column (last is actually 3 as the count starts from 0).

**package** net.sf.reportengine.samples;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**import** java.io.InputStream;

**import** net.sf.reportengine.FlatReport;

**import** net.sf.reportengine.config.DefaultDataColumn;

**import** net.sf.reportengine.config.DefaultGroupColumn;

**import** net.sf.reportengine.core.calc.Calculator;

**import** net.sf.reportengine.in.StreamReportInput;

**import** net.sf.reportengine.out.HtmlOutput;

/\*\*

\* **yearly expenses report**

\*/

**public** **class** YearlyExpenses {

**public** **static** **void** main(String[] args) {

**try** {

FlatReport flatReport = **new** FlatReport();

flatReport.setReportTitle("Yearly expenses report");

//the input

InputStream fileStream = **new** FileInputStream("yearlyExpenses.txt");

StreamReportInput reportInput = **new** StreamReportInput(fileStream,"\t");

flatReport.setIn(reportInput);

//the output

HtmlOutput reportOutput = **new** HtmlOutput(**new** FileOutputStream("out.html"));

flatReport.setOut(reportOutput);

//groups configuration

flatReport.addGroupColumn(**new** DefaultGroupColumn("Year",

0, //input column index

0 //group priority level

));

flatReport.addGroupColumn(**new** DefaultGroupColumn("Month",

1, //input column index

1 //group priority level

));

//data columns

flatReport.addDataColumn(**new** DefaultDataColumn("Spent on", //header

2, //input col idx Calculator.*COUNT*));

flatReport.addDataColumn(**new** DefaultDataColumn("Amount",//header

3, //input index

Calculator.*SUM*));

//the one and only execute

flatReport.execute();

} **catch** (FileNotFoundException e) {

e.printStackTrace();

}

}

}

If you check carefully the first data column (“Spent on” column) you’ll notice that a Calculator.COUNT has been added. This will count the number of items on which I spent money on during the month/year.

The output should be:

|  |  |  |  |
| --- | --- | --- | --- |
| Yearly expenses report | | | |
| **Year** | **Month** | **Spent on** | **Amount** |
| 2011 | August | food | 500 |
| 2011 | August | transportation | 300 |
| **Total August** |  | **2** | **800** |
| 2011 | September | food | 567 |
| 2011 | September | transportation | 154 |
| 2011 | September | entertainment | 200 |
| **Total September** |  | **3** | **921** |
| **Total 2011** |  | **5** | **1721** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| **Grand Total** |  | **9** | **2629** |

Let’s explain the groupings made:

flatReport.addGroupColumn(**new** DefaultGroupColumn("Year",

0, //input column index

0 //group priority level

));

flatReport.addGroupColumn(**new** DefaultGroupColumn("Month",

1, //input column index

1 //group priority level

));

I will not insist on the header (Year and Month), but I have to draw your attentions on the second and third parameter. The second parameter (input column index) has been discussed before (see the data columns section) – represents the index of the input columns starting with zero. The third parameter – group level or priority – helps report-engine establish a hierarchy among group columns. So, in the example above the year takes precedence over the month groups. How is this translated into the report ? When a change in the year happens then not only the totals for the year are shown but also the totals for the month as in the extract from the final result shown previously:

|  |  |  |  |
| --- | --- | --- | --- |
| **Total September** |  | **3** | **921** |
| **Total 2011** |  | **5** | **1721** |

# Pivot Tables /Crosstab reports

## What is different?

Let’s go back to our first example: the monthly expenses report. As a flat report this used to look like this:

|  |  |  |
| --- | --- | --- |
|  | | |
| **Month** | **Spent on** | **Amount** |
| August | food | 500 |
| August | transportation | 300 |
| September | food | 567 |
| September | transportation | 154 |
| September | dinner | 200 |

Well I just realized that all values in the second column repeat themselves. Wouldn’t be easier to follow and compare data if those values were arranged in the header like below?

|  |  |  |  |
| --- | --- | --- | --- |
| **Month** | **food** | **transportation** | **entertainment** |
| August | 500 | 300 | 0 |
| September | 567 | 154 | 200 |

The answer is: Absolutely yes! Think about a situation where you’d have much more data. Arranged in a pivot table all your data is much easier to follow and compare but this is only possible when the values in the column you want as header repeat themselves. Otherwise the column header would be much longer and it wouldn’t allow you to compare the values.

## What else I have to set up for a Pivot table?

The pivot table accepts all settings of a flat report and it introduces two more settings: the header row and the crosstab data.

### The header rows

As previously seen, the header row is the list of distinct values that should be displayed in the header of the report:



Here’s how you configure a header row based on the values in the second column ( column index = 1) to the report:

report.addHeaderRow(**new** DefaultCrosstabHeaderRow(1));

There is no limit on the rows that can be displayed in the header, still, if you add too many; your report will be hard to follow. Here’s a report with two header rows (in blue):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Males | | | Females | | |
| **Country** | under 20 | under 50 | under 80 | under 20 | under 50 under | 80 |
| Sweden | 1000 | 10 | 4 | 1 | 0 | 0 |
| Norway | 0 | 100 | 0 | 0 | 0 | 0 |
| Italy | 2000 | 0 | 0 | 0 | 0 | 0 |
| Romania | 0 | 0 | 0 | 0 | 200 | 0 |
| France | 300 | 0 | 3000 | 30 | 0 | 0 |

The order in which you add the header rows is very important. For instance for the report above the first header row is the one containing the Males, Females values and the second one would be the one containing “under 20”, “under 50” …

### The crosstab data

The crosstab data is the data shown in the report. It usually comes from an input column.

Here’s the initial data (the input)

|  |  |  |  |
| --- | --- | --- | --- |
| August |  | food | **500** |
| August |  | transportation | **300** |
| September |  | food | **567** |
| September |  | transportation | **154** |
| September |  | entertainment | **200** |

And now, the pivot table result



In order to configure the crosstab data to a report you just have to add an instance of DefaultCrosstabData to the report:

report.setCrosstabData(**new** DefaultCrosstabData(2));

The example above constructs a DefaultCrosstabData based on the third input column (column index 2)

## Your first Pivot table report

It’s time to create our first pivot table report:

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**import** net.sf.reportengine.CrossTabReport;

**import** net.sf.reportengine.config.DefaultCrosstabData;

**import** net.sf.reportengine.config.DefaultCrosstabHeaderRow;

**import** net.sf.reportengine.config.DefaultDataColumn;

**import** net.sf.reportengine.in.IReportInput;

**import** net.sf.reportengine.in.StreamReportInput;

**import** net.sf.reportengine.out.HtmlOutput;

**import** net.sf.reportengine.out.IReportOutput;

/\*\*

\* this is my first pivot table report

\*/

**public** **class** FirstPivotTableReport {

**public** **static** **void** main(String[] args) {

**try**{

CrossTabReport report = **new** CrossTabReport();

//set up the input/output

IReportInput in = **new** StreamReportInput(**new** FileInputStream("expenses.csv"));

report.setIn(input);

IReportOutput output = **new** HtmlOutput(**new** FileOutputStream("Expenses.html"));

report.setOut(output);

//set up data column

report.addDataColumn(**new** DefaultDataColumn("Month", 0));

//set up the header rows (from the second column)

report.addHeaderRow(**new** DefaultCrosstabHeaderRow(1));

//set up the crosstab data

report.setCrosstabData(**new** DefaultCrosstabData(2));

//report execution

report.execute();

}**catch**(FileNotFoundException fnfExc){

fnfExc.printStackTrace();

}

}

}

## The Input

## The Output

## Configuration

## Totals and aggregations

# Advanced features

## Spring integration

## Writing a custom input for your reports

## Writing a custom output for your reports

# FAQ

# Limitations

# Useful links