Report-engine tutorial

Version

Contents

[What is report-engine? 4](#_Toc353104888)

[What report-engine can do for you? 4](#_Toc353104889)

[Limitations 4](#_Toc353104890)

[How to add report-engine to your project? 5](#_Toc353104891)

[Using Maven 5](#_Toc353104892)

[Using Groovy Grape 5](#_Toc353104893)

[Using Grails 5](#_Toc353104894)

[Using Scala SBT 5](#_Toc353104895)

[Using Apache Buildr 5](#_Toc353104896)

[What is a flat report? 6](#_Toc353104897)

[The code structure of a flat report 6](#_Toc353104898)

[The input 7](#_Toc353104899)

[Report Output 8](#_Toc353104900)

[Report Columns Configuration 9](#_Toc353104901)

[Data columns 9](#_Toc353104902)

[Your first report 10](#_Toc353104903)

[After executing the code, the result should be an html file 11](#_Toc353104904)

[Group columns 13](#_Toc353104905)

[Your first report containing a group Column 14](#_Toc353104906)

[More on totals and groupings 15](#_Toc353104907)

[Auto configured flat reports 18](#_Toc353104908)

[Pivot Tables /Crosstab reports 20](#_Toc353104909)

[What is a pivot table? 20](#_Toc353104910)

[What else I have to set up for a Pivot table? 20](#_Toc353104911)

[The header rows 20](#_Toc353104912)

[The crosstab data 21](#_Toc353104913)

[Your first Pivot table report 21](#_Toc353104914)

[Totals and groupings for pivot reports 22](#_Toc353104915)

[Advanced features 25](#_Toc353104916)

[Spring integration 25](#_Toc353104917)

[Writing a custom data column 27](#_Toc353104918)

[Writing a custom input 29](#_Toc353104919)

[Useful links 32](#_Toc353104920)

# What is report-engine?

Report Engine is a set of JAVA classes for reports and pivot tables with support for columns, groupings, totals/subtotals. It accepts input from memory, files, databases and exports the data in a multitude of formats : HTML, RTF, PDF, TXT, SVG etc.

# What report-engine can do for you?

Whenever you have data arranged into columns, report engine can come in handy by helping you not only export your original data but also by computing sums, averages etc. at any level.

# Limitations

* Report-engine cannot create charts.
* No programmatic paging. This doesn’t mean there’s no paging at all but the paging is supported by the viewer-application.
* Printing is fully handled by the viewer-application (e.g Adobe Reader, Firefox, Internet Explorer, etc.)
* The final report is always a table with columns

# How to add report-engine to your project?

## Using Maven

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

<dependency>

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

<artifactId>reportengine</artifactId>

<version></version>

</dependency>

## Using Groovy Grape

@Grapes(

@Grab(group='net.sf.reportengine', module='reportengine', version='')

)

## Using Grails

compile 'net.sf.reportengine:reportengine:'

## Using Scala SBT

libraryDependencies += "net.sf.reportengine" % "reportengine" % ""

## Using Apache Buildr

'net.sf.reportengine:reportengine:jar:'

## What is a flat report?

This is a normal tabular report (don't get confused by its name) whose layout will look like:

|  |  |  |
| --- | --- | --- |
| Report title | | |
| **Header 1** | **Header 2** | **Header 3** |
| data 11 | data 12 | data 13 |
| data 21 | data 22 | data 23 |
| data 31 | data 32 | data 33 |
| data 41 | data 42 | data 43 |

## The code structure of a flat report

Each report needs several elements configured: title, input, output and column definitions. Let’s have a look at the report below:

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

flatReport.setReportTitle("Statistics");

//input configuration

flatReport.setIn(**new** TextInput("employees.txt"));

//output configuration

flatReport.setOut(**new** ExcelOutput("output.xls"));

//columns configuration

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

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

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

//start execution of the report

flatReport.execute();

## The input

The main input classes are:

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

Let’s see the example below:

ReportInput input = **new** TextInput("employees.txt", "\t");

//constructs the input based on a file having as data-separator between data //columns the TAB character

…but remember, TextInput can get data from any java.io.Reader (not only files).

URL url = **new** URL("http://www.mysite.com/inputData/expenses.csv");

TextInput reportInput = **new** TextInput(

**new** InputStreamReader(url.openStream()),",");

flatReport.setIn(reportInput);

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

1. If you already have a database connection you can use it like:

java.sql.Connection dbConnection = ...

SqlInput sqlInput = **new** SqlInput();

sqlInput.setConnection(connection);

sqlInput.setSqlStatement(

"select id, country, region, city, population from DB\_TABLE");

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

SqlInput sqlInput = **new** SqlInput();

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

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

sqlInput.setDbUser("sa");

sqlInput.setDbPassword("secret");

sqlInput.setSqlStatement(

"select id, country, region, city, population from DB\_TABLE");

* ***Custom Input***

If these classes don’t cover your needs you can always write your own input by implementing the *ReportInput* interface or, even better, by extending the [AbstractReportInput](http://reportengine.sourceforge.net/html/apidocs/net/sf/reportengine/in/AbstractReportInput.html) class as instructed [here](#_Writing_a_custom)

## Report Output

The predefined output formats for your reports are:

* *HtmlReportOutput - fast html output that creates a html page with styles into any java.io.Writer (including files)*

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

* *ExcelReportOutput* – creates an excel output into a java.io.OutputStream (files included)

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

* *StaxReportOutput - xml output based on STax (streaming xml) technology*
* *XslFoReportOutput - output based on XSL-FO framework - can result in PDF, PNG*, Postscript or everything supported by [apache fop project](http://xmlgraphics.apache.org/fop/trunk/output.html). The default implementation outputs to PDF :

XslFoOutput pdfOutput = **new** XslFoOutput("employees.pdf");

But if you want another format, you can use the MimeConstants provided by apache fop project

XslFoOutput pngOutput = **new** XslFoOutput( "employees.png",

MimeConstants.*MIME\_PNG*);

* Of course you can always write your own report output by implementing the *ReportOutput* interface as [here](#_Writing_a_custom)

## Report Columns Configuration

There are two kinds of columns accepted by a flat report: data columns and group columns.

### Data columns

Data columns are normal report columns displaying data and total results. For each column there are a few parameters to set:

* + header
  + values to be displayed
  + calculator (if totals of any kind are needed )
  + data formatter
  + horizontal alignment

Let’s discuss each of these parameters in detail.

What is the column 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 | Aug. | 500 | |
| 2011 | Sept. | 300 | |
| 2011 | Oct. | 134 | |

#### How can I tell report-engine what values to display on a particular column?

Report-engine can be instructed to get data from a specific **input** column or the user can define its own way of getting data for each input data row. For the moment we will discuss only the default implementation. To customize your own data columns please consult the section

The default implementation for a data column ([net.sf.reportengine.config.DefaultDataColumn](http://reportengine.sourceforge.net/html/apidocs/net/sf/reportengine/config/DefaultDataColumn.html)) has an inputColumnIndex attribute which tells report-engine which column from your input should be displayed in that column. Using inputColumnIndex you instruct report-engine to display your data on the desired position. For instance, assuming an input like the one above, I may decide to show the Year column as the third in the final report and show the Month column 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)

//the column will be displayed first because it is added first in the report

report.addDataColumn( column month having inputColumnIndex = 1)

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

//the values in the Amount will be displayed second because this

//column is added second

report.addDataColumn(column amount having inputColumnIndex = 2)

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

report.addDataColumn(column year having inputColumnIndex = 0)

As you can see, report-engine uses column indexes starting from zero, so zero is our first input column. Also please note that, for report-engine, 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 all values of a column in order to get a SUM or an Average or whatever computation comes to your mind. You can use an existing calculator (SUM, AVG, MIN, MAX, COUNT, FIRST, LAST) or you can create your own by implementing the ICalculator interface.

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

* by using the setter methods (recommended)

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

column.setHeader("Month");

column.setInputColumnIndex(0);

column.setCalculator(Calculator.*SUM*);

column.setHorizAlign(HorizontalAlign.*LEFT*);

* or by using one of the available constructors

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: my expenses report. We will use as input [a file](http://svn.code.sf.net/p/reportengine/code/samples/expenses.csv) containing the list of my expenses during several months. Our initial target is to create just a simple html report containing all input column and nothing more.

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

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

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

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

/\*\*

\* this is your first report having the following steps

\*

\* 1. construct a flat report

\* 2. adds an input to my report

\* 3. adds an output

\* 4. configures the columns of my report

\* 5. executes the report

\*/

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

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

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

flatReport.setReportTitle("My first expenses report");

//the input

TextInput reportInput = **new** TextInput("./inputData/expenses.csv",",");

flatReport.setIn(reportInput);

//the output

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

flatReport.setOut(reportOutput);

//columns configuration (using column constructors – not recommended)

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

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

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

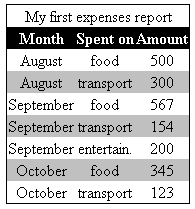
flatReport.execute();

}

}

The always up to date source code can be found [here](http://svn.code.sf.net/p/reportengine/code/trunk/reportengine-samples/src/main/java/net/sf/reportengine/samples/flat/FirstReport.java)

### After executing the code, the result should be an html file



Now, there are some things we can improve, for instance, the “Amount” column should have its values right-aligned while the other string columns should be left aligned. Let’s see how we can do this:

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

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

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

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

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

/\*\*

\* this report is the same as {@link FirstReport} only that

\* the first and the second columns are horizontally-aligned to left

\* and the third (Amount column) is right aligned

\*/

**public** **class** SecondReport {

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

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

flatReport.setReportTitle("My first expenses report");

//the input

TextInput reportInput = **new** TextInput("./inputData/expenses.csv",",");

flatReport.setIn(reportInput);

//the output

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

flatReport.setOut(reportOutput);

//columns configuration

DefaultDataColumn monthColumn = **new** DefaultDataColumn();

monthColumn.setHeader("Month");

monthColumn.setInputColumnIndex(0);

monthColumn.setHorizAlign(HorizontalAlign.*LEFT*);

flatReport.addDataColumn(monthColumn);

DefaultDataColumn destinationColumn = **new** DefaultDataColumn();

destinationColumn.setHeader("Money spent on ..");

destinationColumn.setInputColumnIndex(1);

destinationColumn.setHorizAlign(HorizontalAlign.*LEFT*);

flatReport.addDataColumn(destinationColumn);

DefaultDataColumn amountColumn = **new** DefaultDataColumn();

amountColumn.setHeader("Amount");

amountColumn.setInputColumnIndex(2);

amountColumn.setHorizAlign(HorizontalAlign.*RIGHT*);

flatReport.addDataColumn(amountColumn);

//report execution

flatReport.execute();

}

}

The always up to date source code can be found [here](http://svn.code.sf.net/p/reportengine/code/trunk/reportengine-samples/src/main/java/net/sf/reportengine/samples/flat/SecondReport.java)

### Group columns

As the name suggests, group columns are helpful when one needs groups of data in order to have a better totals display. At each change in the values of a declared group column the totals are displayed. Please keep in mind that totals can be only added to the Data Columns by setting a Calculator, a group column helps displaying and organizing the report. 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$** |

Please note that calculators are added to data columns not to group columns.

### Your first report containing a group Column

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

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

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

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

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

**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) **throws** Exception{

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

flatReport.setShowTotals(**true**);

flatReport.setShowGrandTotal(**false**);

flatReport.setReportTitle("Mothly Expenses");

//define the input

TextInput reportInput = **new** TextInput("expenses.csv”);

flatReport.setIn(reportInput);

//define the output

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

flatReport.setOut(output);

//group column configuration

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

0, //input col. idx

0 //group priority

));

//data columns configuration

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

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

2,

Calculators.*SUM*));

//start executing the report

flatReport.execute();

}

}

The full-code of the above report can be found [here](http://svn.code.sf.net/p/reportengine/code/trunk/reportengine-samples/src/main/java/net/sf/reportengine/samples/flat/FirstGroupReport.java).

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 with header, input column index and group level. The first two have been discussed in the Data Columns section, but the third it’s something new: the group level. The group level tells report-engine that among group columns this has the highest priority (priority 0). If your report has only one group column this parameters is not so important but if you have more than one group column then the group level becomes much more important. The next section will further clarify this.

Let’s look how the totals are calculated:

//data columns configuration

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

**flatReport.addDataColumn(**

**new DefaultDataColumn("Amount", 2, Calculators.*SUM*));**

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

### More on totals and groupings

Now, let’s see a more complex example: my yearly expenses report, a report having 2 group columns and 2 data columns having totals.

Here’s a list of my expenses over two years (simplified for clarity sake):

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).

**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.TextInput;

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

/\*\*

\* **yearly expenses report**

\*/

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

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

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

flatReport.setReportTitle("Yearly expenses report");

//the input

flatReport.setIn(**new** TextInput("yearlyExpenses.txt","\t"));

flatReport.setShowGrandTotal(**false**);

flatReport.setShowTotals(**true**);

//the output

HtmlOutput reportOutput = **new** HtmlOutput("yearlyExpensesOut.html"));

flatReport.setOut(reportOutput);

//groups configuration

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

0, //input column index

0 ));//group priority

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

1, //input column index

1. )); //group priority

//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();

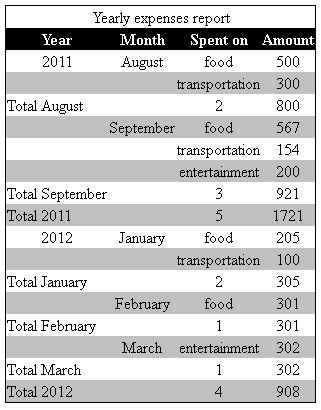
}

}

The full source code can be found [here](http://svn.code.sf.net/p/reportengine/code/trunk/reportengine-samples/src/main/java/net/sf/reportengine/samples/flat/YearlyExpenses.java)

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

The output should be:



Let’s explain the groupings made:

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

0, //input column index

0 ));//group priority

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

1, //input column index

1));//group priority

I will not insist on the header (Year and Month), but I have to draw your attention on the second and third parameters. 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 or precedence 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 December** |  | **3** | **921** |
| **Total 2011** |  | **5** | **1721** |

# Auto configured flat reports

Starting with version 0.8.0 report engine added a new kind of report: the auto configured report. This report gets the most out of the input and tries to configure the report by itself. Of course the default configuration (guessed from the input metadata) can be always overwritten.

A simple auto configured flat report looks like:

SqlInput input = **new** SqlInput();

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

…

AutodetectFlatReport report = **new** AutodetectFlatReport();

report.setIn(input);

report.setOut(**new** ExcelOutput("./output/FirstAutoconfiguredReport.xls"));

report.execute();

The full source code can be found [here](http://svn.code.sf.net/p/reportengine/code/trunk/reportengine-samples/src/main/java/net/sf/reportengine/samples/autodetect/AutoconfigReportFirstSample.java)

As you can see there’s no column configuration, everything is guessed by checking the metadata of the sql input. How is this possible? By checking the metadata of your sql column and providing default values according to the type of column. For instance, if your column contains numbers the alignment of your column will be to the right.

When a special configuration needs to be set to one column, the auto-configured report gives you the opportunity to do it:

report.forColumn("COUNTRY").setGroup(**true**).setHAlign(HorizAlign.*CENTER*);

report.forColumn("REGION").setGroup(**true**).setHeader("East/West");

report.forColumn("VALUE").setCalculator(Calculators.*SUM*);

For the moment this is only supported for reports having as input the sql input: SqlInput.

Here’s an example of an auto configured report where some of the default values are being overwritten:

The full source code can be found [here](http://svn.code.sf.net/p/reportengine/code/trunk/reportengine-samples/src/main/java/net/sf/reportengine/samples/autodetect/AutoconfigFlatSecondSample.java)

**import** net.sf.reportengine.AutoconfigFlatReport;

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

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

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

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

/\*\*

\* simple auto configured report

\* having some default configurations overwritten

\*/

**public** **class** AutoconfigFlatSecondSample {

**public** **static** **void** main(String... args){

SqlInput input = **new** SqlInput();

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

input.setDbConnString("jdbc:hsqldb:file:./inputData/databases/testdb");

input.setDbUser("SA");

input.setDbPassword("");

input.setSqlStatement(

"select cntry, region, city, sex, religion, value "+

"from testreport "+

"order by country, region, city");

AutoconfigFlatReport report = **new** AutoconfigFlatReport();

report.setIn(input);

report.setOut(**new** HtmlOutput("./output/ConfiguredAutodetect.html"));

report.forColumn("cntry").setGroup(**true**).setHAlign(HorizAlign.*CENTER*);

report.forColumn("region").setGroup(**true**).setHeader("East/West");

report.forColumn("value").setCalculator(Calculators.*SUM*);

report.execute();

}

}

# Pivot Tables /Crosstab reports

## What is a pivot table?

Pivot tables are particular types of reports where data is arranged as a 2 dimensional table. 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 | entertainment | 200 |

Note that all values in the second column repeat themselves. Wouldn’t be easier to follow and compare data if those would have been arranged like in the table below?

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

Absolutely! 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 (input, output, data columns, group columns) 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** 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.TextInput;

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

/\*\*

\* this is my first pivot table report

\*/

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

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

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

//set up the input/output

report.setIn(**new** TextInput("expenses.csv"));

report.setOut(**new** HtmlOutput("xpenses.html"));

//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();

}

}

The source code for the above report can be found [here](http://svn.code.sf.net/p/reportengine/code/trunk/reportengine-samples/src/main/java/net/sf/reportengine/samples/pivot/FirstPivotTableReport.java)

## Totals and groupings for pivot reports

Everything a flat report supports is also available for Pivot tables: Grouping columns, totals, grand total. Now let’s see the yearly expenses flat report translated into a pivot table:

* the input

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

* then … the code

**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.config.DefaultGroupColumn;

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

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

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

/\*\*

\* my first pivot table with groupings and totals

\*/

**public** **class** YearlyExpensesPivotTable {

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

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

//setting the input

report.setIn(**new** TextInput("yearlyExpenses.txt", "\t"));

//setting the output

report.setOut(**new** HtmlOutput("yrlyXpensesPivot.html"));

//set up the group and data columns

report.addGroupColumn(**new** DefaultGroupColumn("Year", 0, 0));

report.addDataColumn(**new** DefaultDataColumn("Month", 1));

//set up the header rows, crosstab data

report.addHeaderRow(**new** DefaultCrosstabHeaderRow(2));

report.setCrosstabData(**new** DefaultCrosstabData(3, Calculators.*SUM*));

report.setShowTotals(**true**);

//the report execution

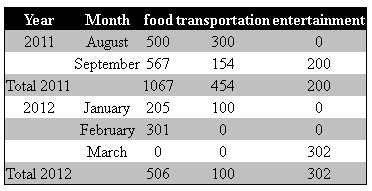
report.execute();

}

}

The full source code for the report above can be found [here](http://svn.code.sf.net/p/reportengine/code/trunk/reportengine-samples/src/main/java/net/sf/reportengine/samples/pivot/YearlyExpensesPivotTable.java)

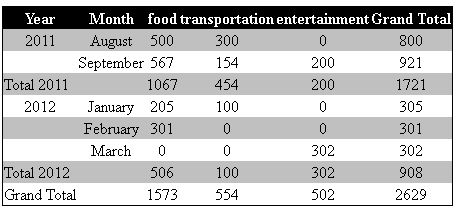
* and the result:



If you need grand totals you just have to add:

report.setShowGrandTotal(**true**);

And you’ll get:



# Advanced features

## Spring integration

Any report can be easily configured in spring, just remember what any report needs: input, columns configuration and output. A simple text file input can be configured in spring like this :

<bean id=*"input"* class=*"net.sf.reportengine.in.TextInput"*>

<property name=*"filePath"* value=*"./inputData/expenses.csv"*/>

<property name=*"separator"* value=*","*/>

</bean>

…and an html output can be configured like:

<bean id=*"output"* class=*"net.sf.reportengine.out.HtmlOutput"*>

<property name=*"filePath"* value=*"./output/springConfiguredReport.html"*/>

</bean>

…finally a data column can be defined in spring like:

<bean class=*"net.sf.reportengine.config.DefaultDataColumn"*>

<property name=*"header"* value=*"Amount"*/>

<property name=*"inputColumnIndex"* value=*"2"*/>

<property name=*"calculator"*>

<bean class=*"net.sf.reportengine.core.calc.SumCalculator"* />

</property>

</bean>

Now, using the easily configurable lists of beans, here’s the final spring configuration for my report:

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans>

<bean id=*"expensesReport"* class=*"net.sf.reportengine.FlatReport"*>

<property name=*"reportTitle"* value=*"Spring Configured Report"* />

<property name=*"showTotals"* value=*"true"* />

<property name=*"showGrandTotal"* value=*"true"*/>

<property name=*"showDataRows"* value=*"true"* />

<property name=*"in"* ref=*"input"*/>

<property name=*"out"* ref=*"output"*/>

<property name=*"groupColumns"*>

<list>

<bean class=*"net.sf.reportengine.config.DefaultGroupColumn"*>

<property name=*"header"* value=*"Month"*/>

<property name=*"inputColumnIndex"* value=*"0"*/>

<property name=*"groupingLevel"* value=*"0"*/>

</bean>

</list>

</property>

<property name=*"dataColumns"*>

<list>

<bean class=*"net.sf.reportengine.config.DefaultDataColumn"*>

<property name=*"header"* value=*"Spent On"*/>

<property name=*"inputColumnIndex"* value=*"1"*/>

</bean>

<bean class=*"net.sf.reportengine.config.DefaultDataColumn"*>

<property name=*"header"* value=*"Amount"*/>

<property name=*"inputColumnIndex"* value=*"2"*/>

<property name=*"calculator"*>

<bean class=*"net.sf.reportengine.core.calc.SumCalculator"* />

</property>

</bean>

</list>

</property>

</bean> <!—end report bean -->

<!-- THE INPUT -->

<bean id=*"input"* class=*"net.sf.reportengine.in.TextInput"*>

<property name=*"filePath"* value=*"./inputData/expenses.csv"*/>

<property name=*"separator"* value=*","*/>

</bean>

<!-- THE OUTPUT -->

<bean id=*"output"* class=*"net.sf.reportengine.out.HtmlOutput"*>

<property name=*"filePath"* value=*"./output/springConfiguredReport.html"*/>

</bean>

</beans>

The source code for the above spring configuration can be found [here](http://svn.code.sf.net/p/reportengine/code/trunk/reportengine-samples/src/main/resources/application-context.xml).

The java code is very simple:

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

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

/\*\*

\* this is my first spring configured flat report

\*/

**public** **class** SpringConfiguredFlatReport {

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

ApplicationContext context =

**new** ClassPathXmlApplicationContext("application-context.xml");

FlatReport report = (FlatReport)context.getBean("expensesReport");

report.execute();

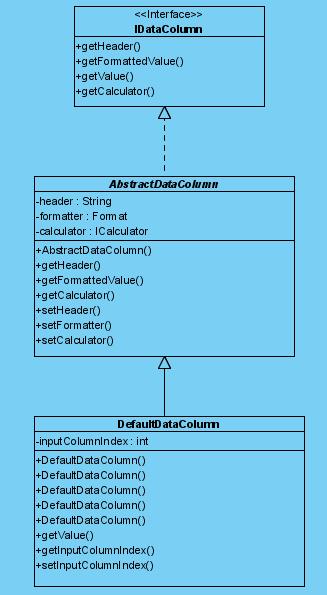
}

}

The source code for the above report can be found [here](http://svn.code.sf.net/p/reportengine/code/trunk/reportengine-samples/src/main/java/net/sf/reportengine/samples/flat/SpringConfiguredFlatReport.java).

## Writing a custom data column

As you probably remember, report-engine doesn’t support only DefaultDataColumns. There’s a class hierarchy supporting the columns feature.



If you don’t want a special column you can always implement the [IDataColumn](http://reportengine.sourceforge.net/html/apidocs/net/sf/reportengine/config/IDataColumn.html) interface or, even better, extend the [AbstractDataColumn](http://reportengine.sourceforge.net/html/apidocs/net/sf/reportengine/config/AbstractDataColumn.html) abstract class. Let’s assume you want to append the values of two other columns into one. There no default implementation for this behavior but this is where you can extend the framework and add your custom implementation. Let’s assume you have the following 4 columns input

**Bonus FirstName Salary LastName**

100 john 2000 doe

170 jack 1250 the ripper

220 tom 1340 jones

34 bill 254 clinton

… but you need a list of only two columns where the first contains the full name (i.e. 2nd + 4th column) and the second column contains the sum of the first and the third

First let’s see the code for the full name (2nd + 4th column values)

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

**import** net.sf.reportengine.core.algorithm.NewRowEvent;

/\*\*

\* This custom column contains the appended values

\* of the second and the fourth columns

\*/

**public** **class** FullNameCustomDataColumn **extends** AbstractDataColumn {

/\*\*

\* Constructor for full name data column

\* **@param** header

\*/

**public** FullNameCustomDataColumn(){

**super**("Full Name");

}

@Override

**public** Object getValue(NewRowEvent newRowEvent) {

Object[] inputRow = newRowEvent.getInputDataRow();

String secondColumnValue = (String)inputRow[1];

String fourthColumnValue = (String)inputRow[3];

//append the second and fourth column values

**return** secondColumnValue + " " + fourthColumnValue;

//this is just for instructional purposes.

//One should use a String Builder/Buffer for such operations

}

}

You can find the code [here](http://svn.code.sf.net/p/reportengine/code/trunk/reportengine-samples/src/main/java/net/sf/reportengine/samples/customColumns/FullNameCustomDataColumn.java).

The [SumCustomColumn](http://svn.code.sf.net/p/reportengine/code/samples/src/main/java/net/sf/reportengine/samples/customColumns/SumCustomColumn.java) is almost the same only that returns an integer. More important is the code for of the report itself:

**import** java.io.FileOutputStream;

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

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

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

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

/\*\*

\* this report uses a custom column (full name column)

\*/

**public** **class** CustomColumnFlatReport {

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

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

flatReport.setReportTitle("My custom columns report");

//the input

flatReport.setIn(**new** TextInput("names.txt","\t"));

//the output

HtmlOutput reportOutput = **new** HtmlOutput("customColsResult.html");

flatReport.setOut(reportOutput);

//columns configuration

flatReport.addDataColumn(**new** FullNameCustomDataColumn());

flatReport.addDataColumn(**new** SumCustomColumn());

flatReport.execute();

}

}

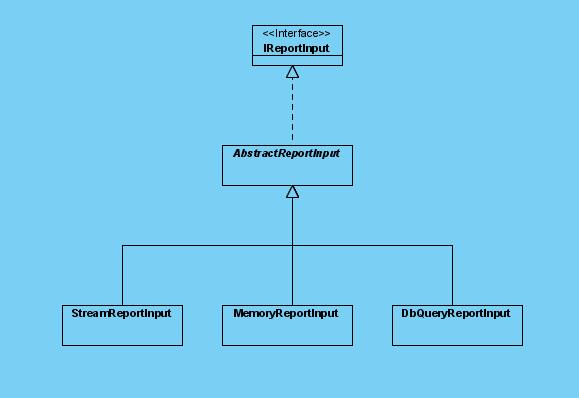
The code above can be found [here](http://svn.code.sf.net/p/reportengine/code/trunk/reportengine-samples/src/main/java/net/sf/reportengine/samples/customColumns/CustomColumnFlatReport.java)

The final output will look like:

|  |  |
| --- | --- |
| My custom columns report | |
| **Full-name** | **Full Salary** |
| john doe | 2100 |
| jack the ripper | 1420 |
| tom jones | 1560 |
| bill clinton | 2880 |

## Writing a custom input

The hierarchy for input is straightforward: an interface defining the basic behavior is the parent then an abstract class with multiple children.



Basically your input class should be able to provide to its listeners data line by line. The methods of the ReportOutput interface are:

/\*\*

\* opens the input for reading

\*/

**public** **void** open();

/\*\*

\* closes the input and releases all resources used

\*/

**public** **void** close();

/\*\*

\* retrieves the next row of data

\* **@return** an array of data objects

\*/

**public** Object[] nextRow();

/\*\*

\* true if there are any rows left otherwise false

\* **@return** true if the input has more rows to return

\*/

**public** **boolean** hasMoreRows();

# Useful links

* Reportengine website: <http://reportengine.sourceforge.net>
* The report engine source code can be found at: http://svn.code.sf.net/p/reportengine/code
* The samples presented in this tutorial can be found at : <http://svn.code.sf.net/p/reportengine/code/trunk/reportengine-samples/>
* Developer’s email: dragos dot balan at gmail dot com