ReportEngine tutorial

Version

Contents

[What is report-engine? 4](#_Toc428975972)

[What report-engine can do for you? 4](#_Toc428975973)

[What java version is ReportEngine designed for? 4](#_Toc428975974)

[Limitations 4](#_Toc428975975)

[How to add report-engine to your project? 5](#_Toc428975976)

[Using Maven 5](#_Toc428975977)

[Using Groovy Grape 5](#_Toc428975978)

[Using Grails 5](#_Toc428975979)

[Using Scala SBT 5](#_Toc428975980)

[Using Apache Buildr 5](#_Toc428975981)

[ReportEngine logging 6](#_Toc428975982)

[What is a Report? 7](#_Toc428975983)

[Report Output 8](#_Toc428975984)

[Report Title 9](#_Toc428975985)

[Paragraph 9](#_Toc428975986)

[Hello World report 9](#_Toc428975987)

[Flat Tables 10](#_Toc428975988)

[What is a flat table? 10](#_Toc428975989)

[How to build a Flat Table? 11](#_Toc428975990)

[The Flat Table input 11](#_Toc428975991)

[Flat Table Columns Configuration 12](#_Toc428975992)

[Data columns 12](#_Toc428975993)

[Your first table report 13](#_Toc428975994)

[Sorting your column data 16](#_Toc428975995)

[Programmatically formatting the data 16](#_Toc428975996)

[More on totals and groupings 19](#_Toc428975997)

[What if my input data doesn’t have the group columns sorted 21](#_Toc428975998)

[Pivot (or Crosstab) tables 23](#_Toc428975999)

[What is a pivot table? 23](#_Toc428976000)

[What I have to set up for a Pivot table? 23](#_Toc428976001)

[The pivot table header rows 23](#_Toc428976002)

[The pivot table data 24](#_Toc428976003)

[Your first Pivot table report 24](#_Toc428976004)

[Totals and groupings for pivot reports 25](#_Toc428976005)

[Useful links 26](#_Toc428976006)

# 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 and databases and exports the data in a multitude of formats: HTML, RTF, PDF, TXT, SVG etc.

# What report-engine can do for you?

For any kind of data (files, databases, etc) report engine can help you re-arranging, computing sums, averages, combining columns and exporting your data.

# What java version is ReportEngine designed for?

ReportEngine is developed and designed for JAVA 6 or higher

# Limitations

* ReportEngine cannot create charts (but I encourage you to create report components for charts)
* No programmatic paging. This doesn’t mean there’s no paging at all but the paging is supported by the viewer-application.

# 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:'

# ReportEngine logging

ReportEngine is using [SLF4J](http://slf4j.org/) as logging framework. The Simple Logging Facade for Java (SLF4J) serves as a simple facade or abstraction for various logging frameworks (e.g. java.util.logging, logback, log4j) allowing the end user to plug in the desired logging framework at deployment time.

What does that mean? If you use log4j in your current project and you want to see the logs of ReportEngine you just need to add the slf4j-log4j bridge (jar file) in your classpath. For maven this can be done like:

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.7.5</version>

</dependency>

The same for other logging frameworks: just add the bridge in your classpath.

[Here’s a nice tutorial](http://saltnlight5.blogspot.ro/2013/08/how-to-configure-slf4j-with-different.html) on how to configure SLF4J with different logging frameworks. More details on SLF4J can be found in [SLF4J’s user manual](http://slf4j.org/manual.html)

# What is a Report?

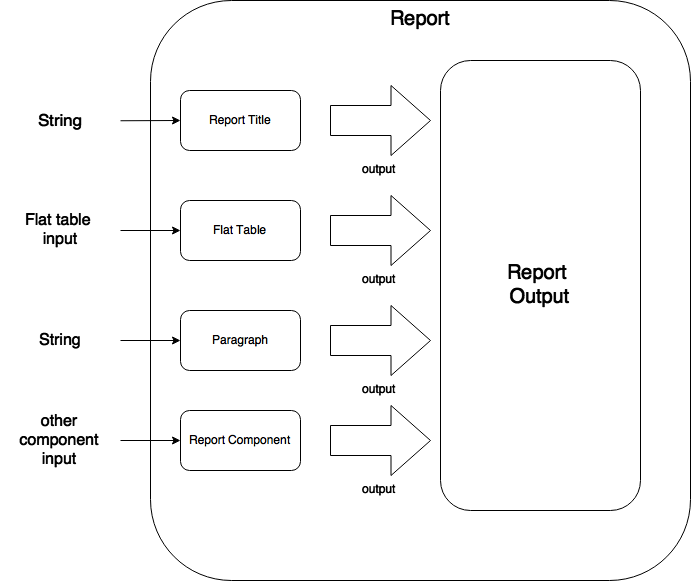
A report is a collection of components displayed one by one by the ReportEngine API. An example of a report component could be: the title of the report, a paragraph, a table inside the report, a pivot table or any other custom report component build by the users of this API. Let’s take an example: below you’ll find a simple report with multiple components:

|  |  |  |
| --- | --- | --- |
| The average salary per country | | |
|  | | |
| **Country** | **Population** | **Salary** |
| Burma | 300 | 234 |
| Uganda | 100 | 134 |
| Papua | 34 | 56 |
| Virgin Islands | 5 | 1004 |
|  |  |  |
|  |  |  |
| The above table shows fictional data. Please don’t use this  data in production systems ☺ | | |

|  |
| --- |
| <--- this is the report title |
|  |
|  |
| <--- this is a flat table (component) |
|  |
|  |
|  |
|  |
| <---- empty line |
|  |
| <---- paragraph |
|  |

The figure above shows a report with 4 report components: a report title, a table, an empty line and a paragraph.

Each report component has an input (with the exception of the empty line) and uses the output of the report to display itself:



Let’s start the discussion of components with the output since the output is shared by all components via the report configuration.

# Report Output

Some of the most important output formats for your reports are:

* *Html (via HtmlReportOutput class): html output that creates a html page with styles into any java.io.Writer*

ReportOuputoutput **= new** HtmlReportOutput(**new** FileWriter("employees.html"))

* *Excel (via ExcelXmlReportOutput* ): creates an excel-xml output into any *java.io.Writer*

ReportOuput output = **new** ExcelXmlReportOutput(**new** FileWriter("empl.xml"));

* *Pdf (via PdfReportOutput ): outputs into any java.io.OutputStream*

ReportOutput output = **new** PdfReportOutput(**new** FileOutputStrem("empl.pdf"));

All output classes can be found in the net.sf.reportengine.out package. Other formats accepted are: svg, fo and many other types of output are under construction.

One can always write his own report output by implementing the *ReportOutput* interface as described [here](#_Writing_a_custom).

Now, let’s discuss one by one the report components:

# Report Title

This is a simple string that will be displayed centered at the beginning of the report. As you probably guessed, the input for this report component is a String and this is the only configuration needed by this component:

ReportTitle title= new ReportTitle("Census data")

Please note that you can add as many titles as you like in a report.

# Paragraph

This component is under construction

# Hello World report

Now that we’ve seen all aspects of a report (the report, its components and the output) it’s time to build a very simple report:

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

**import** java.io.FileWriter;

**import** java.io.IOException;

**import** net.sf.reportengine.Report;

**import** net.sf.reportengine.ReportBuilder;

**import** net.sf.reportengine.components.ReportTitle;

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

**public** **class** HelloWorldReport {

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

//preparation of output and components

HtmlReportOutput output = **new** HtmlReportOutput(**new** FileWriter("Hello.html"));

ReportTitle title = **new** ReportTitle("Hello World report");

//report set-up

Report report = **new** ReportBuilder(output)

.add(title)

.build();

//report execution

report.execute();

}

}

**The only piece of code which needs to be discussed is the creation and set-up of the report. ReportEngine provides a fluent builder for the construction of its most important interface, the net.sf.reportengine.Report. The usage of this fluent builder should be something like:**

Report report = **new** ReportBuilder(output)

.add(title)

.add(table)

.add(paragraph)

.add(pivotTable)

…

.build();

**Please note that nothing is done until you call the report.execute() method.**

**It’s time to see two heavy report components the: Flat Table and the Pivot Table**

# Flat Tables

## What is a flat table?

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

|  |  |  |
| --- | --- | --- |
| **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 most important features of the Flat Table are: grouping and aggregation, sorting, programmatic values formatting, non-programmatic table and values formatting and more.

## How to build a Flat Table?

Each flat table needs several elements configured: input and column definitions. Let’s have a look at the flat table below:

FlatTable table = **new** FlatTableBuilder(**new** TextTableInput("employees.txt"))

//columns configuration

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

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

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

.build();

## The Flat Table input

The main input classes are:

* ***TextTableInput***- 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 an example below:

TableInput tableInput = **new** TextTableInput("employees.txt", "\t");

The example above builds the input based on a file having as data-separator the TAB character. Now let’s see a more sophisticated use of the TextTableInput class

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

TableInput tableInput = **new** TextTableInput(

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

* ***SqlInput*** - executes a query and sends the result as input for any flat table

TableInput sqlInput = **new** SqlTableInput(

"jdbc:hsqldb:mem:countriesDB",

"org.hsqldb.jdbcDriver",

"user",

"password",

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

This class creates a connection to the database and executes the provided query. In case you want to re-use a connection you should have a look into the SqlConnectionBasedTableInput and JdbcResultsetTableInput:

java.sql.Connection dbConnection = ...

TableInput connBasedInput = **new** SqlConnectionBasedTableInput(

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

false);

* ***InMemoryTableInput – this is an array holding the java objects you want to serve as input for your flat table***
* ***Custom Input***

If these classes don’t cover your needs you can always write your own input by implementing the *TableInput* interface or, even better, by extending the AbstractTableInput abstract class as shown [here](#_Writing_a_custom)

## Flat Table 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 configurations to set:

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

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 | |

#### What values to display?

ReportEngine can be instructed to get data from a specific **input** column or the user can define its own way of getting data. For the moment we will discuss only the default implementation. To customize your own data columns please consult the ***Writing a custom data column*** 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 ReportEngine to display your data on the desired position:

**new** DefaultDataColumn

.Builder(0) // builds a column based on the first input column

.header("Month")

.build();

ReportEngine uses column indexes starting from zero, so zero is our first input column. Another important note: for report-engine, the order in which you define your columns is very important because **it defines the output order of your columns**.

For instance, assuming an input with three columns: year, month and amount spent (like the one presented above in the [“What is the column header ?“](#_What_is_the) section, 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

table.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

table.addDataColumn(column amount having inputColumnIndex = 2)

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

table.addDataColumn(column year having inputColumnIndex = 0)

#### What is the group calculator?

The group 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 [net.sf.reportengine.core.calc.GroupCalculator](http://reportengine.sourceforge.net/html/apidocs/net/sf/reportengine/core/calc/GroupCalculator.html) interface.

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

* by using the builder (recommended)

DefaultDataColumn column = **new** DefaultDataColumn.Builder(0)

.header("Amount")

.useCalculator(GroupCalculators.*SUM*)

.horizAlign(HorizontalAlign.*LEFT*)

.build();

* or by using one of the available constructors (strongly discouraged but useful for demonstration purposes)

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

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

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

### Your first table report

It’s now time to build our first report containing a table: a report showing my expenses during several months. We will use as input [a file](http://svn.code.sf.net/p/reportengine/code/trunk/reportengine-samples/inputData/expenses.csv) containing this list of expenses. Our initial target is to create just a simple html report containing all input column and nothing more:

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

**import** java.io.FileWriter;

**import** java.io.IOException;

**import** net.sf.reportengine.Report;

**import** net.sf.reportengine.ReportBuilder;

**import** net.sf.reportengine.components.FlatTable;

**import** net.sf.reportengine.components.FlatTableBuilder;

**import** net.sf.reportengine.components.ReportTitle;

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

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

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

/\*\*

\* this is your first report having the following steps

\*

\* 1. construct the report output (html in this case)

\* 2. construct the flat table having the expenses.csv file as input

\* 3. build the report for the output defined in step 1 by adding a title, the previous table

\* 5. report execution

\*/

**public** **class** FirstReportWithATable {

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

//step 1:constructing the report output

HtmlReportOutput reportOutput = **new** HtmlReportOutput(

**new** FileWriter("c:/temp/FirstReportWithATable.html"));

//step 2: constructing a table with 3 columns

FlatTable flatTable =

**new** FlatTableBuilder(**new** TextTableInput("./inputData/expenses.csv",","))

.addDataColumn(**new** DefaultDataColumn.Builder(0).header("Month").build())

.addDataColumn(**new** DefaultDataColumn.Builder(1).header("Spent on").build())

.addDataColumn(**new** DefaultDataColumn.Builder(2).header("Amount").build())

.build();

//step 3: building a report with two components (a title and a flat table)

//FileWriter is used just for demo purposes

Report report = **new** ReportBuilder(reportOutput)

.add(**new** ReportTitle("My first report"))

.add(flatTable)

.build();

//report execution

report.execute();

}

}

After executing the code, the result should be an html file like below:

|  |  |  |
| --- | --- | --- |
| My first report | | |
| Month | **Spent on** | **Amount** |
| August | food | 500 |
| August | transport | 300 |
| September | food | 567 |
| September | transport | 154 |
| September | entertainment | 200 |
| October | food | 345 |
| October | transport | 123 |

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** java.io.FileWriter;

**import** java.io.IOException;

**import** net.sf.reportengine.Report;

**import** net.sf.reportengine.ReportBuilder;

**import** net.sf.reportengine.components.FlatTable;

**import** net.sf.reportengine.components.FlatTableBuilder;

**import** net.sf.reportengine.components.ReportTitle;

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

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

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

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

/\*\*

\* the first and the second columns are aligned horizontally

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

\*/

**public** **class** ColumnsWithAlignmentReport {

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

FlatTable table =

**new** FlatTableBuilder(**new** TextTableInput("./input/expenses.csv",","))

.addDataColumn(**new** DefaultDataColumn.Builder(0)

.header("Month") .horizAlign(HorizAlign.***LEFT***)

.build())

.addDataColumn(**new** DefaultDataColumn.Builder(1)

.header("Spent on ?")

.horizAlign(HorizAlign.***LEFT***)

.build())

.addDataColumn(**new** DefaultDataColumn.Builder(2)

.header("Amount") .horizAlign(HorizAlign.***RIGHT***)

.build())

.build();

**new** ReportBuilder(

**new** HtmlReportOutput(**new** FileWriter("./ColumnsWithAlign.html")))

.add(**new** ReportTitle("Report with columns aligned programmatically"))

.add(table)

.build()

.execute();

}

}

### Sorting your column data

Sorting data on a specific column can be easily done by calling one of the sorting methods: sortAsc() , sortDesc() available in the builder of [DefaultDataColumn](http://reportengine.sourceforge.net/html/apidocs/net/sf/reportengine/config/DefaultDataColumn.Builder.html).

flatTable.addDataColumn(**new** DefaultDataColumn.Builder(2)

.header("Amount")

.**sortAsc()**

.build());

### Programmatically formatting the data

Another useful feature of every type of column is value-formatting. Currently data and group columns can be formatted using String.format() which is called by the framework. You just need to specify the string format as in [the specs](http://docs.oracle.com/javase/1.5.0/docs/api/java/util/Formatter.html).

flatTable.addDataColumn(

**new** DefaultDataColumn.Builder(2)

.header("Amount")

.**valuesFormatter(**"%.2f"**)**

.build());

flatTable.addDataColumn(

**new** DefaultDataColumn.Builder(2)

.header("Birth Date")

.**valuesFormatter(**"%tD"**))**

.build());

Please note that totals follow another formatting because they usually have another data type. To format the total values (i.e. those returned by calculators) you need to use the long version of the useCalculator() method as in the example below:

DefaultDataColumn column = **new** DefaultDataColumn.Builder(0)

.header("Amount")

.useCalculator(GroupCalculators.*SUM,* "%.2f")

.horizAlign(HorizontalAlign.*LEFT*)

.build();

**Group columns**

Group columns are helpful when one needs to group rows in order to show subtotals for each group or just a better display of the data. Let’s see an example with my 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 ReportEngine will make sure to display totals, averages (or whatever you’ve set) at each change in the values of the first column:

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

In ReportEngine in order to group data you need two settings:

1. Declare the group column ( by using the DefaultGroupColumn class)
2. Use a GroupCalculator (like SUM, AVG, etc.) on the data column which values should be summed /averaged.

**import** java.io.FileWriter;

**import** java.io.IOException;

**import** net.sf.reportengine.Report;

**import** net.sf.reportengine.ReportBuilder;

**import** net.sf.reportengine.components.FlatTable;

**import** net.sf.reportengine.components.FlatTableBuilder;

**import** net.sf.reportengine.components.ReportTitle;

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

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

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

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

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

/\*\*

\* The first report containing a group column.

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

\* after each change of a month a total will be displayed

\* on the Amount column where the calculator has been added

\*/

**public** **class** FirstReportWithGroups {

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

//constructing a flat table with 3 columns: first is declared as a group column

//the third contains the group calculator (in this case an SUM)

FlatTable flatTable =

**new** FlatTableBuilder(**new** TextTableInput("./input/expenses.csv",","))

.addGroupColumn(**new** DefaultGroupColumn.Builder(0).header("Month").build())

.addDataColumn(**new** DefaultDataColumn.Builder(1).header("On What?").build())

.addDataColumn(**new** DefaultDataColumn.Builder(2)

.header("Amount")

.useCalculator(GroupCalculators.***SUM***)

.build())

.build();

//building and executing the report

**new** ReportBuilder(**new** HtmlReportOutput(

**new** FileWriter("./target/MonthlyExpensesUsingGroups.html")))

.add(**new** ReportTitle("Monthly Expenses"))

.add(flatTable)

.build()

.execute();

}

}

Let me draw your attention on the way we defined the Month column as a group column:

.addGroupColumn(**new** Default**Group**Column.Builder(0) //input col. idx

.header("Month")

.build());

Don’t forget, calculators are specific to data columns not to group columns, that’s why, in the previous example, we’ve added the SUM to a data column:

flatTable.addDataColumn(**new** DefaultDataColumn.Builder(2)

.header("Amount")

.useCalculator(GroupCalculators.*SUM*)

.build());

The result of your first report containing a group should be something like:

|  |  |  |
| --- | --- | --- |
| **Monthly Expenses** | | |
|  | | |
| **Month** | **On What?** | **Amount** |
| August | food | 500 |
|  | transport | 300 |
| Total August |  | 800 |
| September | food | 567 |
|  | transport | 154 |
|  | entertainment | 200 |
| Total September |  | 921 |
| October | food | 345 |
|  | transport | 123 |
| Total October |  | 468 |
| **Grand Total** |  | **2189** |

### 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 with totals.

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

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |

Let’s build a report that will show the totals spent for each year and month. For this we have to declare the first and the second columns (year and month) as group columns and add a SUM calculator on the last column (last is actually 3 as the count starts from 0).

It’s time to introduce another attribute of the group columns: the group level which helps ReportEngine prioritize between multiple group columns. If the table has only one group column this parameter is not important but if you have more than one groups then the group level becomes important. The next section will further clarify this:

**new** DefaultGroupColumn.Builder(1) //1 is the input column index

.header("Month")

.level(2) //2 - is the grouping level

.build());

The source code for such a report should look like:

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** net.sf.reportengine.Report;

**import** net.sf.reportengine.ReportBuilder;

**import** net.sf.reportengine.components.FlatTable;

**import** net.sf.reportengine.components.FlatTableBuilder;

**import** net.sf.reportengine.components.ReportTitle;

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

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

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

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

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

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

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

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

FlatTable table =

**new** FlatTableBuilder(**new** TextTableInput("./input/yearlyExpenses.txt","\t"))

//groups configuration

.addGroupColumn(**new** DefaultGroupColumn.Builder(0)

.header("Year")

.horizAlign(HorizAlign.***LEFT***)

.level(0)

.build())

.addGroupColumn(**new** DefaultGroupColumn.Builder(1)

.header("Month")

.horizAlign(HorizAlign.***LEFT***)

.level(1)

.build())

//data columns

.addDataColumn(**new** DefaultDataColumn.Builder(2)

.header("Spent on")

.horizAlign(HorizAlign.***LEFT***)

.build())

.addDataColumn(**new** DefaultDataColumn.Builder(3)

.header("Amount")

.horizAlign(HorizAlign.***RIGHT***)

.useCalculator(GroupCalculators.***SUM***, "%.2f")

.build())

.build();

//build and execute the report

**new** ReportBuilder(

**new** PdfReportOutput(**new** FileOutputStream("./YearlyExpensesReport.pdf")))

.add(**new** ReportTitle("Yearly expenses report"))

.add(table)

.build()

.execute();

}

}

Notice that the second data column contains a SUM calculator which will compute the amount spent on that specific month/year. The output should be a pdf file like:

|  |  |  |  |
| --- | --- | --- | --- |
| **Yearly expenses report** | | | |
| **Year** | **Month** | **Spent on** | **Amount** |
| 2011 | August | food | 500 |
|  |  | transportation | 300 |
| Total August |  |  | 800.00 |
| 2011 | September | food | 567 |
|  |  | transportation | 154 |
|  |  | entertainment | 200 |
| Total September |  |  | 921.00 |
| Total 2011 |  |  | 1721.00 |
| 2012 | January | food | 205 |
|  |  | transportation | 100 |
| Total January |  |  | 305.00 |
| 2012 | February | food | 301 |
| Total February |  |  | 301.00 |
| 2012 | March | entertainment | 302 |
| Total March |  |  | 302.00 |
| Total 2012 |  |  | 908.00 |
| **Grand Total** |  |  | **2629.00** |

Let’s see one more time the groupings:

.addGroupColumn(**new** DefaultGroupColumn.Builder(0) //input column index

.header("Year")

.level(0) //group priority

.build())

.addGroupColumn(**new** DefaultGroupColumn.Builder(1) //input column index

.header("Month")

.level(1) //group priority

.build())

Now let’s discuss about the group level or group priority: in the previous configuration the year group takes precedence over the month group. 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** |  |  | **921** |
| **Total 2011** |  |  | **1721** |

### What if my input data doesn’t have the group columns sorted

In order to perform correct groupings the ReportEngine needs all data on group columns to be sorted. Usually the data comes sorted already (especially from sql queries where a simple “order by” statement can solve the ordering). If your group columns data is not sorted already you should inform ReportEngine about this by calling the sortValues() method from the builder of the FlatTable. This way, the reporting mechanism will programmatically sort your values:

**import** java.io.FileWriter;

**import** java.io.IOException;

**import** net.sf.reportengine.Report;

**import** net.sf.reportengine.ReportBuilder;

**import** net.sf.reportengine.components.FlatTable;

**import** net.sf.reportengine.components.FlatTableBuilder;

**import** net.sf.reportengine.components.ReportTitle;

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

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

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

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

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

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

/\*\*

\* When using group columns, the data in those columns needs to be sorted

\* otherwise the report engine will see a change of group in every row.

\* In this example, the input data for the flat table is not sorted and

\* the report engine is informed about this by using the sortValues() method

\*/

**public** **class** UnsortedGroupValues {

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

FlatTable table =

**new** FlatTableBuilder(**new** TextTableInput("./input/unsortedExpenses.csv",","))

**.sortValues()** //inform reportengine that it has to sort the values

.addGroupColumn(**new** DefaultGroupColumn.Builder(0)

.header("Month")

.horizAlign(HorizAlign.***LEFT***)

.build())

.addDataColumn(**new** DefaultDataColumn.Builder(1)

.header("On What?")

.horizAlign(HorizAlign.***LEFT***)

.build())

.addDataColumn(**new** DefaultDataColumn.Builder(2)

.header("Amount")

.useCalculator(GroupCalculators.***SUM***)

.horizAlign(HorizAlign.***RIGHT***)

.build())

.build();

//build and execute the report

**new** ReportBuilder(**new** HtmlReportOutput(

**new** FileWriter("./target/MonthlyExpensesFromUnsortedInput.html")))

.add(**new** ReportTitle("Monthly Expenses"))

.add(table)

.build()

.execute();

}

}

# Pivot (or Crosstab) tables

## What is a pivot table?

Pivot tables are particular types of tables 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 it 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 |

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 I have to set up for a Pivot table?

The pivot table accepts all settings of a flat table (input, output, data columns, group columns) and it introduces two more settings: the header rows and the pivot data.

### The pivot table header rows

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

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

|  |
| --- |
| <---- header row |
|  |
| **<---- crosstab data** |

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

pivotTable.addHeaderRow(**new** DefaultPivotHeaderRow(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 pivot table 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:

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

|  |
| --- |
| <---- header row |
|  |
| **<---- crosstab data** |

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

**new** PivotTable.Builder().pivotData(**new** DefaultPivotData(2));

The example above constructs a DefaultPivotData 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.FileWriter;

**import** java.io.IOException;

**import** net.sf.reportengine.Report;

**import** net.sf.reportengine.ReportBuilder;

**import** net.sf.reportengine.components.PivotTable;

**import** net.sf.reportengine.components.PivotTableBuilder;

**import** net.sf.reportengine.components.ReportTitle;

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

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

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

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

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

/\*\*

\* this is your first pivot table report

\*/

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

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

PivotTable table =

**new** PivotTableBuilder(**new** TextTableInput("./input/expenses.csv", ","))

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

.addHeaderRow(**new** DefaultPivotHeaderRow(1))

.pivotData(**new** DefaultPivotData(2))

.build();

**new** ReportBuilder(**new** HtmlReportOutput(**new** FileWriter("./ExpensesPivot.html")))

.add(**new** ReportTitle("This is my first report with a pivot table"))

.add(table)

.build()

.execute();

}

}

## Totals and groupings for pivot reports

Everything a flat can do is also available for Pivot tables: groupings and aggregations, totals, subtotals, grand totals, etc. 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 |

* the programmatic configuration:

**import** java.io.FileWriter;

**import** java.io.IOException;

**import** net.sf.reportengine.Report;

**import** net.sf.reportengine.components.PivotTable;

**import** net.sf.reportengine.ReportBuilder;

**import** net.sf.reportengine.components.PivotTableBuilder;

**import** net.sf.reportengine.components.ReportTitle;

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

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

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

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

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

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

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

/\*\*

\* Sample Pivot table with groupings and sub-totals by year

\*/

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

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

PivotTable pivotTable =

**new** PivotTableBuilder(**new** TextTableInput("./input/yearlyExpenses.txt", "\t"))

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

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

.addHeaderRow(**new** DefaultPivotHeaderRow(2))

.pivotData(**new** DefaultPivotData.Builder(3)

.useCalculator(GroupCalculators.***SUM***, "%.2f")

.build())

.showGrandTotal()

.showTotals()

.build();

**new** ReportBuilder(**new** HtmlReportOutput(**new** FileWriter("PivotGroupByYear.html")))

.add(**new** ReportTitle("Yearly expenses arranged as a pivot table"))

.add(pivotTable)

.build()

.execute();

}

}

* and the result:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Year** | **Month** | **food** | **transportation** | **entertainment** | **Grand Total** |
| 2011 | August | 500 | 300 | 0 | 800 |
|  | September | 567 | 154 | 200 | 921 |
| Total 2011 |  | 1067.00 | 454.00 | 200.00 | 1721.00 |
| 2012 | January | 205 | 100 | 0 | 305 |
|  | February | 301 | 0 | 0 | 301 |
|  | March | 0 | 0 | 302 | 302 |
| Total 2012 |  | 506.00 | 100.00 | 302.00 | 908.00 |
| Grand Total |  | 1573.00 | 554.00 | 502.00 | 2629.00 |

# 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