Getting Started With JasperReports

Author: David R. Heffelfinger

Introduction

I've recently been researching reporting tools for a project I will be soon be working on. One of the tools I've been looking at is JasperReports. JasperReports is a very popular open source (LGPL) reporting library written in Java. Unfortunately it is not very well documented and I had a hard time coming up with a simple report. After some research, I was able to generate a simple report, this article summarizes what needs to be done to get started with JasperReports. See Resources for information on how to find additional information and documentation about JasperReports.

Getting Started

JasperReports' reports are defined in XML files, which by convention have an extension of jrxml. A typical jrxml file contains the following elements:

- `<jasperReport>` - the root element.
- `<title>` - its contents are printed only once at the beginning of the report
- `<pageHeader>` - its contents are printed at the beginning of every page in the report.
- `<detail>` - contains the body of the report.
- `<pageFooter>` - its contents are printed at the bottom of every page in the report.
- `<band>` - defines a report section, all of the above elements contain a `band` element as its only child element.

All of the elements are optional, except for the root `jasperReport` element. Here is an example jrxml file that will generate a simple report displaying the string "Hello World!"

```
<?xml version="1.0"?>
<!DOCTYPE jasperReport
  PUBLIC "-//JasperReports//DTD Report Design//EN"
  "http://jasperreports.sourceforge.net/dtds/jasperreport.dtd">

<jasperReport name="Simple_Report">
 <detail>
    <band height="20">
      <staticText>
        <reportElement x="180" y="0" width="200" height="20"/>
        <text><![CDATA[Hello World!]]></text>
      </staticText>
    </band>
  </detail>
</jasperReport>
```

For this simple example, I omitted the optional `<title>`, `<pageHeader>` and `<pageFooter>` elements. The `<staticText>` element, unsurprisingly, displays static text on the report, as can be seen, it contains a single `<text>` element defining the text that will be displayed.

jrxml files need to be "compiled" into a binary format that is specific to JasperReports, this can be achieved by calling the `compileReport()` method on the `net.sf.jasperreports.engine.JasperCompileManager` class. There are several overloaded versions of this method, in our example, we will use the one that takes a single `String` parameter, consult the JasperReport documentation for details on the other versions of the method.

```
public class JasperReportsIntro
{
  public static void main(String[] args)
  {
    JasperReport jasperReport;
    JasperPrint jasperPrint;
    try
    {
      jasperReport = JasperCompileManager.compileReport(
          "reports/jasperreports_demo.jrxml");
      jasperPrint = JasperFillManager.fillReport(
          jasperReport, new HashMap(), new JREmptyDataSource());
      JasperExportManager.exportReportToPdfFile(
          jasperPrint, "reports/simple_report.pdf");
    }
    catch (JRException e)
    {
      e.printStackTrace();
    }
  }
}
```

A jrxml file needs to be compiled only once, but for this simple example it is compiled every time the application is executed. Before a report can be generated, it needs to be "filled" with data, this is achieved by calling the `fillReport()` method on the `net.sf.jasperreports.engine.JasperFillManager` class. Again, there are several overloaded versions of the `fillReport()` method, here we will use one that takes three parameters, an instance of `net.sf.jasperreports.engine.JasperReport`, a `java.util.HashMap` containing any parameters passed to the report, and an instance of a class implementing the `net.sf.jasperreports.engine.JRDataSource` interface. The line that accomplishes this in our example above is

```
jasperPrint = JasperFillManager.fillReport(
          jasperReport, new HashMap(), new JREmptyDataSource());
```

Since our simple report takes no parameters, we pass an empty `HashMap` as the second parameter, and an instance of `net.sf.jasperreports.engine.JREmptyDataSource` as the third parameter. `JREmptyDataSource` is a convenience class included with JasperReports, it is basically a DataSource with no data.

Finally, in our example, we export the report to a PDF file, this way it can be read by Adobe Acrobat, XPDF, Evince, or any other PDF reader, the line that accomplishes this in our example is

```
JasperExportManager.exportReportToPdfFile(
          jasperPrint, "reports/simple_report.pdf");
```

The parameters are self-explanatory.

Displaying JasperReports PDF Reports on the Browser

This article provides a brief howto on how to display PDF reports generated with JasperReports from a web application to the user's browser. Basic knowledge of JasperReports and Java Servlet programming is assumed. See the Resources section for links to JasperReports and Servlet programming tutorials and documentation.

The trick to sending a PDF report generated by JasperReports to the user's browser is to call the `net.sf.jasperreports.engine.JasperRunManager.runReportToPdf()` method. That method has several overloaded versions, the one we will use here has three parameters, a `String` representing the absolute path of the compiled report (jasper file), an instance of a class implementing the `java.util.Map` interface, and an instance of a class implementing the `net.sf.jasperreports.engine.JRDataSource` interface. The `JasperRunManager.runReportToPdf()` method returns an array of bytes that can be passed as a parameter to the `write()` method of the `javax.servlet.ServletOutputStream` class. An instance of `ServletOutputStream` can be obtained from the `getOutputStream()` method of the `javax.servlet.http.HttpServletResponse` class. The best way to explain and visualize all of this is by example, the following code segment demonstrates this technique:

```
package net.ensode.jasperreportsbrowserdemo;

import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.StringWriter;
import java.util.HashMap;

import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import net.sf.jasperreports.engine.JREmptyDataSource;
import net.sf.jasperreports.engine.JRException;
import net.sf.jasperreports.engine.JasperRunManager;

public class JasperReportsBrowserDemoServlet extends HttpServlet
{
```

```
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException
  {
    ServletOutputStream servletOutputStream = response.getOutputStream();
    File reportFile = new File(getServletConfig().getServletContext()
        .getRealPath("/reports/Simple_Report.jasper"));
    byte[] bytes = null;

    try
    {
      bytes = JasperRunManager.runReportToPdf(reportFile.getPath(),
          new HashMap(), new JREmptyDataSource());

      response.setContentType("application/pdf");
      response.setContentLength(bytes.length);

      servletOutputStream.write(bytes, 0, bytes.length);
      servletOutputStream.flush();
      servletOutputStream.close();
    }
    catch (JRException e)
    {
      // display stack trace in the browser
      StringWriter stringWriter = new StringWriter();
      PrintWriter printWriter = new PrintWriter(stringWriter);
      e.printStackTrace(printWriter);
      response.setContentType("text/plain");
      response.getOutputStream().print(stringWriter.toString());
    }
  }
}
```

As can be seen in the example, the easiest way to obtain the absolute path of the jasper file is to call the `getRealPath()` method of an instance of a class implementing the `javax.servlet.ServletContext` interface. For our simple example, we pass an empty instance `java.util.HashMap` and an instance of `net.sf.jasperreports.engine.JREmptyDataSource` as the other two parameters to the `JasperRunManager.runReportToPdf()` method, more complex applications would pass some data inside these two parameters.

Creating Database Reports With JasperReports

Introduction

Our above two articles on JasperReports, have had reports displaying static text only. Although useful to demonstrate how to generate a report using JasperReports, those examples have little practical value. In practice, the vast majority of reports are created to display some kind of database data. In this article we

will demonstrate how to generate a report from database data using JasperReports. Basic familiarity with JasperReports is assumed, see Resources for other introductory articles on JasperReports.

The Data

For this article, we will develop a sample report for a fictitious technology training institute. The report will contain the course number, course name and instructor for each course. The code will read from two database tables, the following SQL scripts were used to create the tables:

```
create table instructors(
  instructor_id int8 primary key,
  first_nm varchar,
  last_nm  varchar);
```

and

```
create table courses (
  course_id int8 primary key,
  course_cd varchar,
  course_nm varchar,
  instructor_id int8
  REFERENCES instructors(instructor_id));
```

The Relational Database Management System used was PostgreSQL 8.0, but the scripts can be easily modified to be used by another RDBMS. The database will initially be populated with the following data:

```
mydb=# select * from instructors;
 instructor_id | first_nm |   last_nm
---------------+----------+--------------
             1 | David    | Heffelfinger
             2 | John     | Doe
             3 | Alice    | Jones
             4 | Mary     | Wang
             5 | Pedro    | Gonzalez
(5 rows)
```

and

```
mydb=# select * from courses;
 course_id | course_cd |                course_nm                | instructor_id
-----------+-----------+-----------------------------------------+-------------- -
         1 | JAVAEEI   | Introduction To Java EE                 |             1
         2 | JAVAEEA   | Advanced Java EE                        |             1
         3 | LINUXI    | Introduction To Linux                   |             2
         4 | LINUXA    | Advanced Linux                          |             3
         5 | JASPI     | Introduction to JasperReports           |             4
         6 | SWINGI    | Client Side Java Programming with Swing |             5
         7 | JAVAI     | Introduction to Java                    |             5
(7 rows)
```

Creating The JRXML File

As explained in our previous article, the XML definition file for a JasperReports report is called a JRXML file. The JRXML file for our database report is defined as follows:

```
<?xml version="1.0"?>
<!DOCTYPE jasperReport
  PUBLIC "-//JasperReports//DTD Report Design//EN"
  "http://jasperreports.sourceforge.net/dtds/jasperreport.dtd">

<jasperReport name="Database_Report">
   <field name="course_cd" class="java.lang.String" />
   <field name="course_nm" class="java.lang.String" />
   <field name="first_nm" class="java.lang.String" />
   <field name="last_nm" class="java.lang.String" />
   <detail>
      <band height="20">
         <textField>
            <reportElement x="10" y="0" width="600" height="20" />
            <textFieldExpression class="java.lang.String">
               <![CDATA[$F{course_cd}]]>
            </textFieldExpression>
         </textField>
         <textField>
            <reportElement x="80" y="0" width="200" height="20" />
            <textFieldExpression class="java.lang.String">
               <![CDATA[$F{course_nm}]]>
            </textFieldExpression>
         </textField>
         <textField>
            <reportElement x="280" y="0" width="200" height="20" />
            <textFieldExpression class="java.lang.String">
               <![CDATA[$F{first_nm}]]> + " " + <![CDATA[$F{last_nm}]]>
            </textFieldExpression>
         </textField>
      </band>
   </detail>
</jasperReport>
```

Relevant sections of the JRXML file for database reporting are shown in **bold**. The name of the `<field>` elements must map database columns in the query used to gather data for the report. The `class` attribute of the `<field>` elements must correspond to the appropriate class of the data in the result set for the query.

These fields are then accessed using the `$F{fieldName}` syntax inside the `<textFieldExpression>` elements, as shown in the example above. The generated report will have a line for each row returned in the query.