Embedding the Pentaho Reporting Engine

Help and Support Resources

If you have questions that are not covered in this guide, or if you would like to report errors in the documentation, please contact your Pentaho technical support representative.

Support-related questions should be submitted through the Pentaho Customer Support Portal at http://support.pentaho.com.

For information about how to purchase support or enable an additional named support contact, please contact your sales representative, or send an email to sales@pentaho.com.

For information about instructor-led training on the topics covered in this guide, visit http://www.pentaho.com/training.

Limits of Liability and Disclaimer of Warranty

The author(s) of this document have used their best efforts in preparing the content and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, express or implied, with regard to these programs or the documentation contained in this book.

The author(s) and Pentaho shall not be liable in the event of incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of the programs, associated instructions, and/or claims.

Trademarks

Pentaho (TM) and the Pentaho logo are registered trademarks of Pentaho Corporation. All other trademarks are the property of their respective owners. Trademarked names may appear throughout this document. Rather than list the names and entities that own the trademarks or insert a trademark symbol with each mention of the trademarked name, Pentaho states that it is using the names for editorial purposes only and to the benefit of the trademark owner, with no intention of infringing upon that trademark.

Company Information

Pentaho Corporation
Citadel International, Suite 340
5950 Hazeltine National Drive
Orlando, FL 32822
Phone: +1 407 812-OPEN (6736)
Fax: +1 407 517-4575
http://www.pentaho.com

E-mail: communityconnection@pentaho.com

Sales Inquiries: sales@pentaho.com

Documentation Suggestions: documentation@pentaho.com

Sign-up for our newsletter: http://community.pentaho.com/newsletter/

Contents

Introduction

The Pentaho Reporting engine is a small set of open source Java classes that enables programmers to retrieve information from a data source, format and process it according to specified parameters, then generate user-readable output. This document provides guidance and instructions for using the Pentaho Reporting SDK to embed the Pentaho Reporting engine into a new or existing Java application.

There are four sample applications in this document, all of which are included in the SDK as **.java** files. Each adds one level of complexity or shows a different kind of output.

You should read this guide in order, from this point all the way to the end of the second example. The remaining portion contains extra information about the Pentaho Reporting engine's capabilities, licensing details, further examples, and information on where to get help and support.

Required Knowledge and Expertise

This document is strictly for Java software developers. You must be familiar with importing JARs into a project, and be comfortable reading inline comments in code to figure out advanced functionality on your own. Proficiency in connecting to data sources is a helpful skill for developing your own application around the Pentaho Reporting engine, but is not required to follow the examples.

Obtaining the Pentaho Reporting SDK

You can download the latest Pentaho Reporting software development kit (SDK) from http://reporting.pentaho.com.

The SDK is available as both a .tar.gz and a .zip archive; both contain the same files, but the .zip file format is more Windows-friendly, and .tar.gz is more Mac-, Linux-, and Unix-friendly.

Once downloaded, unpack the Pentaho Reporting SDK archive to a convenient and accessible location. If you use the Eclipse or IntelliJ IDEA development environments, this directory will also serve as your workspace.

In an effort to reduce the size of the SDK, the source code of its constituent libraries is not included. If you need to see the source to any of the software distributed with the Pentaho Reporting SDK, see *Source Code Links* on page 26 for instructions.

Using the Included Eclipse Project

If you use the Eclipse or IntelliJ IDEA development environments, you can use the Eclipse project included with the Pentaho Reporting SDK to work directly with the example source code. Simply select the unpacked Pentaho Reporting SDK directory as your workspace.

You can also launch the **Sample1.java** and **Sample2.java** example applications directly from the file browser in Eclipse.

Embedding the Reporting Engine Into a Java Application

This section shows in detail how to build a simple reporting application around the Pentaho Reporting engine. There are three classes for the two examples shown in this section:

1. AbstractReportGenerator.java
2. Sample1.java
3. Sample2.java

You can find the full example source code, plus the .prpt report file they use, in the `/source/org/pentaho/reporting/engine/classic/samples/` directory in the Pentaho Reporting SDK.

Overview

In the following samples, the interaction with the Reporting engine follows these basic steps:

1. Boot (initialize)
2. Get the report definition
3. Get the data for the report (if it is created outside of the report definition)
4. Get any report generation parameters (optional)
5. Generate the report output in the requested format

With the samples, this allows us to create an abstract base class for all the samples (AbstractReportGenerator). This class defines the abstract methods:

- **getReportDefinition()**: this loads/creates/returns the report definition
- **getDataFactory()**: this returns the data to be used by the reporting engine (if the report definition does not tell the engine how to retrieve the data).
- **getReportParameters()**: this returns the set of parameters the reporting engine will use while generating the report

The **generateReport()** method tells the reporting engine to generate the report using the above method, and creates the output in one of the following methods (using the OutputType parameter): HTML, PDF, or XLS (Excel). A full list of output types is listed later in this guide, but to keep these examples simple, we'll concentrate on these three.

Sample1.java

In this sample, the **getReportDefinition()** method loads the report definition from a PRPT file created using the Pentaho Report Designer. This report definition defines the following:

- Data Query (retrieving a list of customers based on a set of customer names)
- Report Title
- Report Header – set of 4 columns (Customer Number, Customer Name, Postal Code, Country)
- Report Data – set of 4 columns (Customer Number, Customer Name, Postal Code, Country)

The **getDataFactory()** method returns null to indicate that no data factory is required to be provided. In this example, the source of data is defined in the report definition.

The **getReportParameters()** method defines three parameters in a HashMap:

Parameter Name	Parameter Value	Description
Report Title	Simple Embedded Report Example with Parameters	The value of this parameter will be placed in the Report Title that is centered on the top of each page in the report. In the report definition, the Report Title field is a Text Field whose value is “Report Title”. This indicates that the field will use the value of the parameter “Report Title” when the report is generated.
Col Headers BG Color	yellow	The value of this parameter will be used as the background color of the column header fields. In the report

Parameter Name	Parameter Value	Description
		definition, all four of the column header fields are defined with a bg-color style of “=[Col Headers BG Color]”. This indicates that the value of the “Col Header BG Color” parameter will be used as that value.
Customer Names	"American Souvenirs Inc", "Toys4GrownUps.com", "giftsbymail.co.uk", "BG&E Collectables", "Classic Gift Ideas, Inc"	The value of this parameter defines a set of Customer Names that will be used in the data query. This allows the sample to define which customers will be used in the report at the time the report is generated. SELECT "CUSTOMERS"."CUSTOMERNAME", "CUSTOMERS"."POSTALCODE", "CUSTOMERS"."COUNTRY", "CUSTOMERS"."CUSTOMERNUMBER" FROM "CUSTOMERS" WHERE "CUSTOMERS"."CUSTOMERNAME" IN (${Customer Names})

The **main()** method creates an output filename in which the report will be generated and then starts the report generation process.

Sample2.java

In this sample, the **getReportDefinition()** method creates a blank report and sets the query name to “ReportQuery”. It then adds a report pre-processor called **RelationalAutoGeneratorPreProcessor**.

Report pre-processors execute during the report generation process after the data query has been executed but before the report definition is used to determine the actual layout of the report. The benefit of this is that the **RelationalAutoGeneratorPreProcessor** will use the column information retrieved from the data query to add header fields in the Page Header and data fields in the Item Band of the report definition for each column of data in the result set.

The **getDataFactory()** method first defines the **“DriverConnectionProvider”** which contains all the information required to connect to the database. It then defines the “DataFactory” which will use the connection provider to connect to the database. The Data Factory then has the query set which will be used in report generation. The query name **“ReportQuery”** must match the query name defined when the report definition was created or else the report will contain no data.

The **getReportParameters()** method is not used in this example, so it returns null.

The **main()** method creates an output filename in which the report will be generated and then starts the report generation process.

Sample 0: The Base Class

The **AbstractReportGenerator** class shown below is extended by the two primary example applications. It contains the basic logic that creates a report, leaving the details of input and output to the classes that extend it:

```
/*
 * This program is free software; you can redistribute it and/or modify it
 under the
 * terms of the GNU Lesser General Public License, version 2.1 as
 published by the Free Software
 * Foundation.
 *
```

```
 * You should have received a copy of the GNU Lesser General Public
 License along with this
 * program; if not, you can obtain a copy at http://www.gnu.org/licenses/
old-licenses/lgpl-2.1.html
 * or from the Free Software Foundation, Inc.,
 * 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * This program is distributed in the hope that it will be useful, but
 WITHOUT ANY WARRANTY;
 * without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
 PARTICULAR PURPOSE.
 * See the GNU Lesser General Public License for more details.
 *
 * Copyright 2009 Pentaho Corporation.  All rights reserved.
 *
 * Created July 22, 2009
 * @author dkincade
 */
package org.pentaho.reporting.engine.classic.samples;

import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.util.Map;

import org.pentaho.reporting.engine.classic.core.ClassicEngineBoot;
import org.pentaho.reporting.engine.classic.core.DataFactory;
import org.pentaho.reporting.engine.classic.core.MasterReport;
import
 org.pentaho.reporting.engine.classic.core.ReportProcessingException;
import org.pentaho.reporting.engine.classic.core.layout.output.
AbstractReportProcessor;
import org.pentaho.reporting.engine.classic.core.modules.output.pageable.
base.PageableReportProcessor;
import org.pentaho.reporting.engine.classic.core.modules.output.pageable.
pdf.PdfOutputProcessor;
import org.pentaho.reporting.engine.classic.core.modules.output.table.base
.FlowReportProcessor;
import org.pentaho.reporting.engine.classic.core.modules.output.table.base
.StreamReportProcessor;
import org.pentaho.reporting.engine.classic.core.modules.output.table.html
.AllItemsHtmlPrinter;
import org.pentaho.reporting.engine.classic.core.modules.output.table.html
.FileSystemURLRewriter;
import org.pentaho.reporting.engine.classic.core.modules.output.table.html
.HtmlOutputProcessor;
import org.pentaho.reporting.engine.classic.core.modules.output.table.html
.HtmlPrinter;
import org.pentaho.reporting.engine.classic.core.modules.output.table.html
.StreamHtmlOutputProcessor;
import org.pentaho.reporting.engine.classic.core.modules.output.table.xls.
FlowExcelOutputProcessor;
import org.pentaho.reporting.libraries.repository.ContentLocation;
import org.pentaho.reporting.libraries.repository.DefaultNameGenerator;
import org.pentaho.reporting.libraries.repository.stream.StreamRepository;

/**
 * This is the base class used with the report generation examples. It
 contains the actual <code>embedding</code>
 * of the reporting engine and report generation. All example embedded
 implementations will need to extend this class
 * and perform the following:
 * <ol>
 * <li>Implement the <code>getReportDefinition()</code> method and return
 the report definition (how the report
 * definition is generated is up to the implementing class).
```

```
 * <li>Implement the <code>getTableDataFactory()</code> method and return
 the data factory to be used (how
 * this is created is up to the implementing class).
 * <li>Implement the <code>getReportParameters()</code> method and return
 the set of report parameters to be used.
 * If no report parameters are required, then this method can simply
 return <code>null</code>
 * </ol>
 */
public abstract class AbstractReportGenerator
{
  /**
   * The supported output types for this sample
   */
  public static enum OutputType
  {
    PDF, EXCEL, HTML
  }

  /**
   * Performs the basic initialization required to generate a report
   */
  public AbstractReportGenerator()
  {
    // Initialize the reporting engine
    ClassicEngineBoot.getInstance().start();
  }

  /**
   * Returns the report definition used by this report generator. If this
 method returns <code>null</code>,
   * the report generation process will throw a
 <code>NullPointerException</code>.
   *
   * @return the report definition used by thus report generator
   */
  public abstract MasterReport getReportDefinition();

  /**
   * Returns the data factory used by this report generator. If this
 method returns <code>null</code>,
   * the report generation process will use the data factory used in the
 report definition.
   *
   * @return the data factory used by this report generator
   */
  public abstract DataFactory getDataFactory();

  /**
   * Returns the set of parameters that will be passed to the report
 generation process. If there are no parameters
   * required for report generation, this method may return either an
 empty or a <code>null</code> <code>Map</code>
   *
   * @return the set of report parameters to be used by the report
 generation process, or <code>null</code> if no
   *         parameters are required.
   */
  public abstract Map<String, Object> getReportParameters();

  /**
   * Generates the report in the specified <code>outputType</code> and
 writes it into the specified
   * <code>outputFile</code>.
   *
   * @param outputType the output type of the report (HTML, PDF, HTML)
   * @param outputFile the file into which the report will be written
   * @throws IllegalArgumentException  indicates the required parameters
 were not provided
```

```
  * @throws IOException               indicates an error opening the file
for writing
  * @throws ReportProcessingException indicates an error generating the
report
  */
 public void generateReport(final OutputType outputType, File outputFile)
     throws IllegalArgumentException, IOException,
ReportProcessingException
 {
   if (outputFile == null)
   {
     throw new IllegalArgumentException("The output file was not
specified");
   }

   OutputStream outputStream = null;
   try
   {
     // Open the output stream
     outputStream = new BufferedOutputStream(new
FileOutputStream(outputFile));

     // Generate the report to this output stream
     generateReport(outputType, outputStream);
   }
   finally
   {
     if (outputStream != null)
     {
       outputStream.close();
     }
   }
 }

 /**
  * Generates the report in the specified <code>outputType</code> and
writes it into the specified
  * <code>outputStream</code>.
  * <p/>
  * It is the responsibility of the caller to close the
<code>outputStream</code>
  * after this method is executed.
  *
  * @param outputType   the output type of the report (HTML, PDF, HTML)
  * @param outputStream the stream into which the report will be written
  * @throws IllegalArgumentException  indicates the required parameters
were not provided
  * @throws ReportProcessingException indicates an error generating the
report
  */
 public void generateReport(final OutputType outputType, OutputStream
outputStream)
     throws IllegalArgumentException, ReportProcessingException
 {
   if (outputStream == null)
   {
     throw new IllegalArgumentException("The output stream was not
specified");
   }

   // Get the report and data factory
   final MasterReport report = getReportDefinition();
   final DataFactory dataFactory = getDataFactory();

   // Set the data factory for the report
   if (dataFactory != null)
   {
     report.setDataFactory(dataFactory);
   }
```

```
   // Add any parameters to the report
   final Map<String, Object> reportParameters = getReportParameters();
   if (null != reportParameters)
   {
     for (String key : reportParameters.keySet())
     {
       report.getParameterValues().put(key, reportParameters.get(key));
     }
   }

   // Prepare to generate the report
   AbstractReportProcessor reportProcessor = null;
   try
   {
     // Greate the report processor for the specified output type
     switch (outputType)
     {
       case PDF:
       {
         final PdfOutputProcessor outputProcessor =
             new PdfOutputProcessor(report.getConfiguration(),
outputStream, report.getResourceManager());
         reportProcessor = new PageableReportProcessor(report,
outputProcessor);
         break;
       }

       case EXCEL:
       {
         final FlowExcelOutputProcessor target =
             new FlowExcelOutputProcessor(report.getConfiguration(),
outputStream, report.getResourceManager());
         reportProcessor = new FlowReportProcessor(report, target);
         break;
       }

       case HTML:
       {
         final StreamRepository targetRepository = new
StreamRepository(outputStream);
         final ContentLocation targetRoot = targetRepository.getRoot();
         final HtmlOutputProcessor outputProcessor = new
StreamHtmlOutputProcessor(report.getConfiguration());
         final HtmlPrinter printer = new
AllItemsHtmlPrinter(report.getResourceManager());
         printer.setContentWriter(targetRoot, new
DefaultNameGenerator(targetRoot, "index", "html"));
         printer.setDataWriter(null, null);
         printer.setUrlRewriter(new FileSystemURLRewriter());
         outputProcessor.setPrinter(printer);
         reportProcessor = new StreamReportProcessor(report,
outputProcessor);
         break;
       }
     }

     // Generate the report
     reportProcessor.processReport();
   }
   finally
   {
     if (reportProcessor != null)
     {
       reportProcessor.close();
     }
   }
 }
```

```
}
```

Sample 1: Static Report Definition, JDBC Input, PDF Output

The simplest embedding scenario produces a static report (no user input regarding a data source or query), with JDBC input from the Pentaho-supplied SampleData HSQLDB database, and produces a PDF on the local filesystem.

```
/*
 * This program is free software; you can redistribute it and/or modify it
 under the
 * terms of the GNU Lesser General Public License, version 2.1 as
 published by the Free Software
 * Foundation.
 *
 * You should have received a copy of the GNU Lesser General Public
 License along with this
 * program; if not, you can obtain a copy at http://www.gnu.org/licenses/
old-licenses/lgpl-2.1.html
 * or from the Free Software Foundation, Inc.,
 * 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * This program is distributed in the hope that it will be useful, but
 WITHOUT ANY WARRANTY;
 * without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
 PARTICULAR PURPOSE.
 * See the GNU Lesser General Public License for more details.
 *
 * Copyright 2009 Pentaho Corporation.  All rights reserved.
 *
 * Created July 22, 2009
 * @author dkincade
 */
package org.pentaho.reporting.engine.classic.samples;

import java.io.File;
import java.io.IOException;
import java.net.URL;
import java.util.Map;
import java.util.HashMap;

import org.pentaho.reporting.engine.classic.core.DataFactory;
import org.pentaho.reporting.engine.classic.core.MasterReport;
import
 org.pentaho.reporting.engine.classic.core.ReportProcessingException;
import org.pentaho.reporting.libraries.resourceloader.Resource;
import org.pentaho.reporting.libraries.resourceloader.ResourceException;
import org.pentaho.reporting.libraries.resourceloader.ResourceManager;

/**
 * Generates a report in the following scenario:
 * <ol>
 * <li>The report definition file is a .prpt file which will be loaded and
 parsed
 * <li>The data factory is a simple JDBC data factory using HSQLDB
 * <li>There are no runtime report parameters used
 * </ol>
 */
public class Sample1 extends AbstractReportGenerator
{
  /**
   * Default constructor for this sample report generator
   */
  public Sample1()
  {
  }
```

```
  /**
   * Returns the report definition which will be used to generate the
 report. In this case, the report will be
   * loaded and parsed from a file contained in this package.
   *
   * @return the loaded and parsed report definition to be used in report
 generation.
   */
  public MasterReport getReportDefinition()
  {
    try
    {
      // Using the classloader, get the URL to the reportDefinition file
      final ClassLoader classloader = this.getClass().getClassLoader();
      final URL reportDefinitionURL = classloader.getResource("org/
pentaho/reporting/engine/classic/samples/Sample1.prpt");

      // Parse the report file
      final ResourceManager resourceManager = new ResourceManager();
      resourceManager.registerDefaults();
      final Resource directly =
 resourceManager.createDirectly(reportDefinitionURL, MasterReport.class);
      return (MasterReport) directly.getResource();
    }
    catch (ResourceException e)
    {
      e.printStackTrace();
    }
    return null;
  }

  /**
   * Returns the data factory which will be used to generate the data used
 during report generation. In this example,
   * we will return null since the data factory has been defined in the
 report definition.
   *
   * @return the data factory used with the report generator
   */
  public DataFactory getDataFactory()
  {
    return null;
  }

  /**
   * Returns the set of runtime report parameters. This sample report uses
 the following three parameters:
   * <ul>
   * <li><b>Report Title</b> - The title text on the top of the report</
li>
   * <li><b>Customer Names</b> - an array of customer names to show in the
 report</li>
   * <li><b>Col Headers BG Color</b> - the background color for the column
 headers</li>
   * </ul>
   *
   * @return <code>null</code> indicating the report generator does not
 use any report parameters
   */
  public Map<String, Object> getReportParameters()
  {
    final Map parameters = new HashMap<String, Object>();
    parameters.put("Report Title", "Simple Embedded Report Example with
 Parameters");
    parameters.put("Col Headers BG Color", "yellow");
    parameters.put("Customer Names",
        new String [] {
            "American Souvenirs Inc",
```

```
            "Toys4GrownUps.com",
            "giftsbymail.co.uk",
            "BG&E Collectables",
            "Classic Gift Ideas, Inc",
        });
    return parameters;
  }

  /**
   * Simple command line application that will generate a PDF version of
 the report. In this report,
   * the report definition has already been created with the Pentaho
 Report Designer application and
   * it located in the same package as this class. The data query is
 located in that report definition
   * as well, and there are a few report-modifying parameters that will be
 passed to the engine at runtime.
   * <p/>
   * The output of this report will be a PDF file located in the current
 directory and will be named
   * <code>SimpleReportGeneratorExample.pdf</code>.
   *
   * @param args none
   * @throws IOException indicates an error writing to the filesystem
   * @throws ReportProcessingException indicates an error generating the
 report
   */
  public static void main(String[] args) throws IOException,
 ReportProcessingException
  {
    // Create an output filename
    final File outputFilename = new File(Sample1.class.getSimpleName() +
 ".pdf");

    // Generate the report
    new Sample1().generateReport(AbstractReportGenerator.OutputType.PDF,
 outputFilename);

    // Output the location of the file
    System.err.println("Generated the report [" +
 outputFilename.getAbsolutePath() + "]");
  }
}
```

Sample 2: Static Report Definition, JDBC Input, HTML Output

This example produces a static report (no user input regarding a data source or query), with JDBC input from the Pentaho-supplied SampleData HSQLDB database, and produces an HTML file on the local filesystem.

```
/*
 * This program is free software; you can redistribute it and/or modify it
 under the
 * terms of the GNU Lesser General Public License, version 2.1 as
 published by the Free Software
 * Foundation.
 *
 * You should have received a copy of the GNU Lesser General Public
 License along with this
 * program; if not, you can obtain a copy at http://www.gnu.org/licenses/
old-licenses/lgpl-2.1.html
 * or from the Free Software Foundation, Inc.,
 * 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * This program is distributed in the hope that it will be useful, but
 WITHOUT ANY WARRANTY;
```

```
 * without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
 PARTICULAR PURPOSE.
 * See the GNU Lesser General Public License for more details.
 *
 * Copyright 2009 Pentaho Corporation.  All rights reserved.
 *
 * Created July 22, 2009
 * @author dkincade
 */
package org.pentaho.reporting.engine.classic.samples;

import java.io.File;
import java.io.IOException;
import java.util.Map;

import org.pentaho.reporting.engine.classic.core.DataFactory;
import org.pentaho.reporting.engine.classic.core.MasterReport;
import
 org.pentaho.reporting.engine.classic.core.ReportProcessingException;
import org.pentaho.reporting.engine.classic.core.PageDefinition;
import org.pentaho.reporting.engine.classic.core.wizard.
RelationalAutoGeneratorPreProcessor;
import org.pentaho.reporting.engine.classic.core.modules.
misc.datafactory.sql.SQLReportDataFactory;
import org.pentaho.reporting.engine.classic.core.modules.
misc.datafactory.sql.DriverConnectionProvider;

/**
 * Generates a report in the following scenario:
 * <ol>
 * <li>The report definition file is a .prpt file which will be loaded and
 parsed
 * <li>The data factory is a simple JDBC data factory using HSQLDB
 * <li>There are no runtime report parameters used
 * </ol>
 */
public class Sample2 extends AbstractReportGenerator
{
  private static final String QUERY_NAME = "ReportQuery";

  /**
   * Default constructor for this sample report generator
   */
  public Sample2()
  {
  }

  /**
   * Returns the report definition which will be used to generate the
 report. In this case, the report will be
   * loaded and parsed from a file contained in this package.
   *
   * @return the loaded and parsed report definition to be used in report
 generation.
   */
  public MasterReport getReportDefinition()
  {
    final MasterReport report = new MasterReport();
    report.setQuery(QUERY_NAME);
    report.addPreProcessor(new RelationalAutoGeneratorPreProcessor());
    return report;
  }

  /**
   * Returns the data factory which will be used to generate the data used
 during report generation. In this example,
   * we will return null since the data factory has been defined in the
 report definition.
   *
```

```
   * @return the data factory used with the report generator
   */
  public DataFactory getDataFactory()
  {
    final DriverConnectionProvider sampleDriverConnectionProvider = new
 DriverConnectionProvider();
    sampleDriverConnectionProvider.setDriver("org.hsqldb.jdbcDriver");
    sampleDriverConnectionProvider.setUrl("jdbc:hsqldb:./sql/sampledata");
    sampleDriverConnectionProvider.setProperty("user", "sa");
    sampleDriverConnectionProvider.setProperty("password", "");

    final SQLReportDataFactory dataFactory = new
 SQLReportDataFactory(sampleDriverConnectionProvider);
    dataFactory.setQuery(QUERY_NAME,
        "select CUSTOMERNAME, CITY, STATE, POSTALCODE, COUNTRY from
 CUSTOMERS order by UPPER(CUSTOMERNAME)");

    return dataFactory;
  }

  /**
   * Returns the set of runtime report parameters. This sample report does
 not use report parameters, so the
   * method will return <code>null</code>
   *
   * @return <code>null</code> indicating the report generator does not
 use any report parameters
   */
  public Map<String, Object> getReportParameters()
  {
    return null;
  }

  public static void main(String[] args) throws IOException,
 ReportProcessingException
  {
    // Create an output filename
    final File outputFilename = new File(Sample2.class.getSimpleName() +
 ".html");

    // Generate the report
    new Sample2().generateReport(AbstractReportGenerator.OutputType.HTML,
 outputFilename);

    // Output the location of the file
    System.err.println("Generated the report [" +
 outputFilename.getAbsolutePath() + "]");
  }
}
```

Pentaho Reporting's Capabilities

Now that you are familiar with the basic functions of the Pentaho Reporting engine, you're prepared to learn more about its advanced features, explained in the subsections below.

Technological Advantages

The Pentaho Reporting engine offers unique functionality not found in competing embeddable solutions:

- **Does not require a JDK at runtime.** While you do need a Java Development Kit installed on your development machine, you do not need a JDK to run a program that embeds the Pentaho Reporting engine -- just a standard Sun Java Runtime Environment.
- **All processing is done in memory.** No temporary files are created by the Reporting engine. A program that relies on the Pentaho Reporting engine for report generation can run on a diskless system.
- **Potentially backwards-compatible to JDK 1.2.** The Pentaho Reporting architect has given special consideration to users and developers on legacy systems. While Pentaho focuses its in-house development and QA efforts on JRE 1.6.0, it is possible to use the Reporting engine in older JREs by adding JDBC and JNDI libraries.
- **Dynamically and automatically adjustable components.** The Pentaho Reporting engine detects JARs that add functionality at runtime, so you can add new JARs to expand the engine's capabilities, or remove unnecessary JARs to reduce your application's memory and disk space footprint.
- **Low memory footprint.** A Pentaho Reporting-based application can run with as little as 64MB of memory (though 128MB would dramatically increase report processing speed).
- **Totally configurable through runtime parameterization.** Every style, function, query, and report element is fully customizable by passing parameters to the Reporting engine when you render a report.
- **OpenFormula integration.** OpenFormula is an open standard for mathematical formulas. You can easily create your own custom formulas, or you can customize the ones built into the Pentaho Reporting engine with this clearly and freely documented standard.
- **Simple resource management.** Using the OpenDocument Format (ODF), the Pentaho Reporting engine bundles all report resources, including the data source connection information, query, and even binary resources like images into one canonical file. This simplifies physical resource management and eliminates relative path problems.

Input Types

The Pentaho Reporting engine can connect to virtually any data source:

- JDBC
- JNDI
- Kettle (Pentaho Data Integration)
- Simple SQL (JDBC Custom)
- Pentaho Metadata
- Mondrian MDX
- OLAP4J
- XML
- Simple table
- Scripting data sources (JavaScript, Python, TCL, Groovy, BeanShell)
- Java method invocation
- Hibernate

If your data source is not directly supported, you can use Pentaho Data Integration to transform it into a more report-friendly format, or you can design your own custom data source interface.

Output Types

The Pentaho Reporting engine can create reports in a variety of relevant file formats:

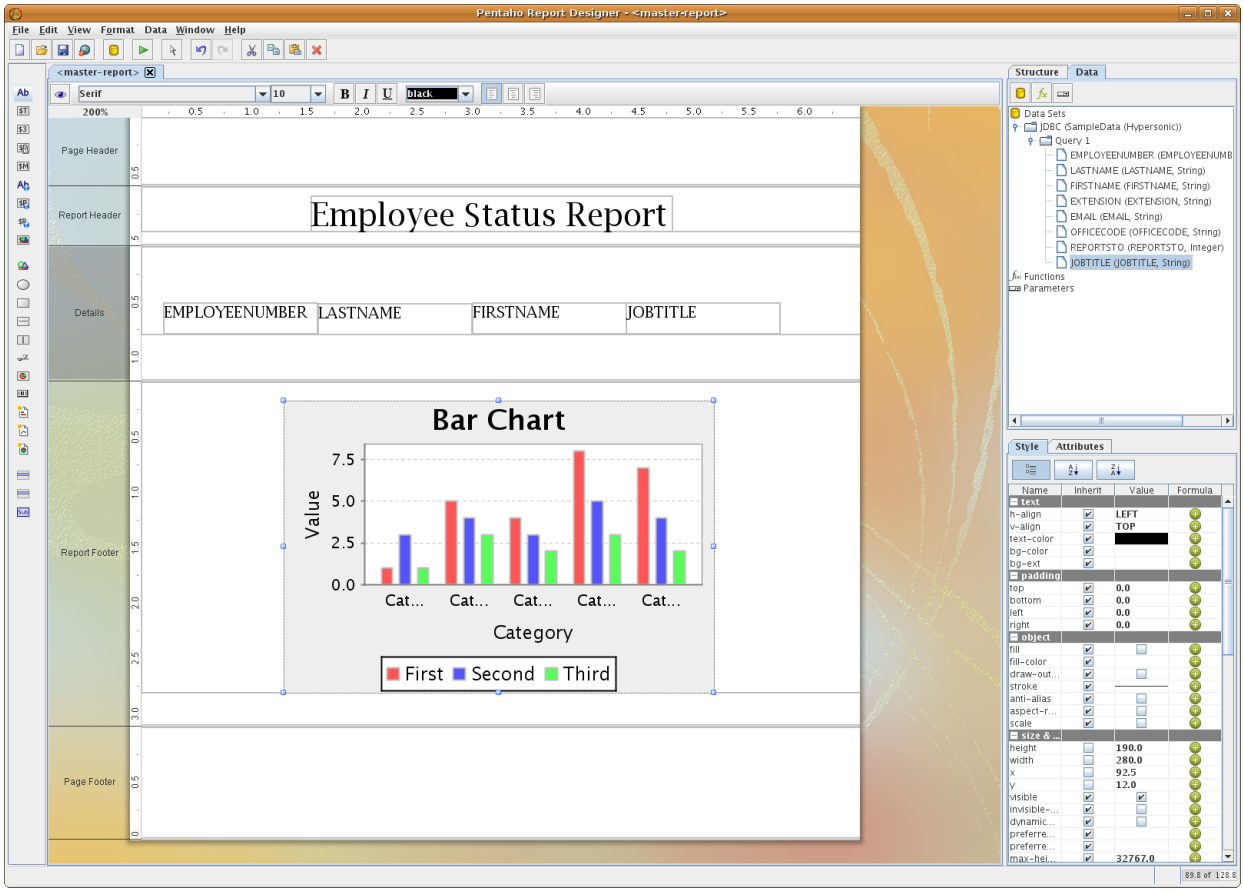
- PDF

- HTML
- Excel
- CSV
- RTF
- XML
- Plain text

All of the output types listed above are highly customizable in terms of style, formatting, and pagination. You can also specify your own output type if none of the standard choices are sufficient.

Pentaho Report Designer

The examples in this guide accept data source input and create user-readable output, which is essentially what the Pentaho Report Designer does with its graphical user interface. In addition to being a powerful report creation and design tool, Report Designer is also an extraordinary example of a Java application that embeds the Pentaho Reporting engine.



You can also create report definition files with Report Designer, then use your custom Reporting engine-based application to render them at a later time.

Other Embedding Scenarios

Pentaho offers many embeddable structures -- not just the Reporting engine. You can also embed or extend the Pentaho Analysis engine (Mondrian), the Pentaho BI Platform, part or all of Pentaho Data Integration (Kettle), and the Weka data mining engine. This guide is focused on reporting, however, so the below scenarios only involve the reporting components of the Pentaho BI Suite.

Building a Custom Reporting Tool

The examples in this guide have covered simple scenarios that don't involve a high degree of user interactivity. It's easy to imagine how far you can expand the example code, even to the point of building your own client tools. On a slightly smaller scale, you could build a report-generation program that merely takes some parameters from a user, then silently emails the report to designated recipients via the Java mail component. You could also design a reporting daemon or service that listens for incoming requests and outputs reports to a Web server.

Pentaho Report Designer is built on the Pentaho Reporting engine, as is the ad hoc reporting functionality built into the Pentaho User Console in the BI Platform. If you need a graphical report creation tool, it would be easier to modify Report Designer than it would be to rewrite it from scratch. For Web-based ad hoc reporting, you will have an easier time embedding the entire BI Platform than trying to isolate and embed just the ad hoc component.

Hacking Pentaho Report Designer

Perhaps you do not need to create a whole new content creation program around the Pentaho Reporting engine; instead, you can enhance or reduce the functionality of Pentaho Report Designer to match your needs.

Report Designer is both modular and extensible, so you can remove or disable large portions of it, or create your own custom data sources, output formats, formulas, and functions. You can also customize Report Designer with your own background images, icons, language, and splash screen.

Embedding the Pentaho BI Platform

If your Web-based reporting application needs scripting, scheduling, and security functionality, it makes more sense to embed the slightly larger Pentaho BI Platform instead of writing a large amount of your own code to add to the Reporting engine. The BI Platform contains powerful scripting and automation capabilities, an email component, report bursting functionality, user authorization and authentication features, and a cron-compatible scheduling framework.

The BI Platform is the heart of the larger Pentaho BI Server, which is a complete J2EE Web application that provides engines for Pentaho Reporting, Data Integration, and Analysis, as well as a fully customizable Web-based user interface that offers ad hoc reporting, real-time analysis views, and interactive dashboard creation.

The BI Server is fully customizable, so your options range from simple rebranding to removing entire components or developing your own plugins to add major user-facing functionality.

License Information

The entire Pentaho Reporting SDK is freely redistributable. Most of it is open source software, but its constituent JARs are under a few different licenses. If you intend to embed and distribute any part of this SDK, you must be familiar with the licensing requirements of the pieces you use.

You can read all of the relevant licenses in text files in the **licenses** subdirectory in the Pentaho Reporting SDK.

Developer Support

The examples in this guide are simple and easy to follow, but with more complex requirements come more advanced programs. While reading the source code comments can help quite a bit, you may still need help to develop an application within a reasonable timeframe. Should you need personal assistance, you can have direct access to the most knowledgeable support resources through a Pentaho Enterprise Edition software vendor annual subscription:

ISV/OEM support options

If phone and email support are not enough, Pentaho can also arrange for an on-site consulting engagement:

Consultative support options

Anatomy of the Pentaho Reporting SDK

SDK Directory Structure

```
/
/documentation
/licenses
/samples
/WebContent
/../META-INF
/../WEB-INF
/../../lib
/lib
/source
/../org
/../../pentaho
/../../../reporting
/../../../../engine
/../../../../../classic
/../../../../../../samples
/sql
```

Directory	Content Description
Documentation	Where the Embedding the Pentaho Reporting Engine PDF is located
Licenses	Contains text files with licensing information
Samples	The eclipse project directory, which contains the samples shown in this guide
Samples/WebContent	WebContent information used with Sample 4 (mainly the WEB-INF/ web.xml)
Samples/lib	The lib directory which makes up the Reporting Engine SDK
Samples/source	The source files used to make up the four reporting samples
Samples/sql	The file-based HSQLDB instance used with the samples

Content of the Samples Directory

File	Purpose
build.properties	Ant properties used with the build script
build.xml	Ant build script
common_build.xml	Ant Build Script
ivysettings.xml	Settings for Ivy (used with build)
ivy.xml	Dependencies for project (used with Ivy – used with build)
.project	Eclipse project file
.classpath	Eclipse classpath file
samples.iml	IntelliJ project file
Sample*.bat	Runs the sample (1/2/3) program on Windows
Sample *.launch	Runs the sample (1/2/3) program from within Eclipse
Sample*.sh	Runs the sample (1/2/3) project on linux
Sample4.war	The WAR that can be dropped in a Servlet Container (Tomcat) and executed

JAR Reference

The Pentaho Reporting SDK consists of the following Pentaho-authored JARs:

JAR File Name	Purpose
libbase	The root project for all reporting projects. Provides base services like controlled boot-up, modularization, configuration. Also contains some commonly used helper classes.
libdocbundle	Support for ODF-document-bundle handling. Provides the engine with the report-bundle capabilities and manages the bundle-metadata, parsing and writing.
libfonts	Font-handling library. Performs the mapping between physical font files and logical font names. Also provides performance optimized font-metadata and font-metrics.
libformat	A performance optimized replacement for JDK TextFormat classes. Accepts the same patterns as the JDK classes, but skips the parsing. Therefore they are less expensive to use in terms of CPU and memory.
libformula	Our OpenFormula implementation. Provides a implementation of the OpenFormula specification. Basically a way to have Excel-style formulas without the nonsense Excel does.
libloader	Resourceloading and caching framework. Used heavily in the engine to load reports and other resources in the most efficient way.
libpixie	Support for rendering WMF (windows-meta-files).
librepository	Abstraction-layer for content-repositories. Heavily used by LibDocbundle and our HTML export.
libserializer	Helper classes for serialization of Java-objects. A factory based approach to locate serializers based on the class of the object we want to serialize. needed as major parts of the JDK are not serializable on their own.
libxml	Advanced SAX-parsing framework and namespace aware XML writing framework used in the engine and libdocbundle.
pentaho-reporting-engine-classic-core	The Pentaho Reporting engine core, which itself consists of modular sub-projects.

Included third-party JARs

JAR File Name	Purpose
activation	The JavaBeans Activation Framework, which determines the type of the

JAR File Name	Purpose
	given data, encapsulates it, discovers the operations available on it, and to instantiates the appropriate bean to execute those operations.
backport-util-concurrent	A library which implements concurrency capabilities found in Java 5.0 and 6.0, which allows building fully-portable concurrent applications for older JREs.
batik-awt-util, batik-bridge, batik-css, batik-dom, batik-ext, batik-gui-util, batik-gvt, batik-parser, batik-script, batik-svg-dom, batik-util, batik-xml	The core Batik SVG toolkit, which adds scalable vector graphics support to a Java application.
bsf	The Apache Jakarta Bean Scripting Framework, which provides scripting language support within Java applications, and access to Java objects and methods from scripting languages.
bsh	The Bean Shell, which dynamically executes standard Java syntax and extends it with common scripting conveniences such as loose types, commands, and method closures like those in Perl and JavaScript.
commons-logging-api	The Apache Commons Logging library, which allows writing to a variety of different logging services in a common format.
itext	Enables dynamic PDF generation.
jsr107cache	A Java cache API specification.
ehcache	A distributed cache library that uses the jsr107cache API.
mail	The Java Mail API, which allows you to send email from a Java application without requiring a separate mail server.
poi	A Java API that allows you to read from and write to Microsoft file formats.
xml-apis	The Apache Commons XML DOM library, which allows you to read from, write to, and validate XML files.

JARs exclusive to the embedding samples

JAR File Name	Purpose
hsqldb	HSQLDB database engine and JDBC driver.
pentaho-reporting-engine-classic-samples	The sample applications explained in this guide.

Source Code Links

Pentaho maintains a Subversion repository for Pentaho Reporting. It consists of many individual, modular projects, all of which are listed below. You can also traverse the repository with a Web browser by replacing the svn:// with an http:// . As is customary with Subversion repositories, the **trunk** is where active development happens; **tags** represent snapshots of official releases; and **branches** are separate codelines generally established for new releases.

JAR File Name	Source Code Repository
libbase	svn://source.pentaho.org/pentaho-reporting/libraries/libbase
libdocbundle	svn://source.pentaho.org/pentaho-reporting/libraries/libdocbundle
libfonts	svn://source.pentaho.org/pentaho-reporting/libraries/libfonts
libformat	svn://source.pentaho.org/pentaho-reporting/libraries/libformat
libformula	svn://source.pentaho.org/pentaho-reporting/libraries/libformula
libloader	svn://source.pentaho.org/pentaho-reporting/libraries/libloader
libpixie	svn://source.pentaho.org/pentaho-reporting/libraries/pixie
librepository	svn://source.pentaho.org/pentaho-reporting/libraries/librepository
libserializer	svn://source.pentaho.org/pentaho-reporting/libraries/libserializer
libxml	svn://source.pentaho.org/pentaho-reporting/libraries/libxml
pentaho-reporting-engine-classic-core	svn://source.pentaho.org/pentaho-reporting/engines/classic/trunk/core

Included third-party JARs

Below are URLs for the source code for the third-party JARs included in the SDK:

JAR File Name	Source Code Repository
batik-awt-util-1.7.jar, batik-bridge-1.7.jar, batik-css-1.7.jar, batik-dom-1.7.jar, batik-ext-1.7.jar, batik-gui-util-1.7.jar, batik-gvt-1.7.jar, batik-parser-1.7.jar, batik-script-1.7.jar, batik-svg-dom-1.7.jar, batik-util-1.7.jar, batik-xml-1.7.jar	http://archive.apache.org/dist/xmlgraphics/batik/batik-src-1.7.zip
bcmail-jdk14-1.38.jar, bcmail-jdk14-138.jar, bcprov-jdk14-1.38.jar, bcprov-jdk14-138.jar, bctsp-jdk14-1.38.jar	http://www.bouncycastle.org/latest_releases.html
bsf-2.4.5.jar	http://mirror.its.uidaho.edu/pub/apache/jakarta/bsf/source/bsf-src-2.4.5.tar.gz
bsh-1.3.0.jar	svn://ikayzo.org/svn/beanshell
commons-logging-api-1.1.jar	http://www.gossipcheck.com/mirrors/apache/commons/logging/source/commons-logging-1.1.1-src.tar.gz
ehcache-core-2.0.1.jar	svn://ehcache.svn.sourceforge.net/viewvc/ehcache/branches/ehcache-2.0.1/
itext-2.1.7.jar, itext-rtf-2.1.7.jar	svn://itext.svn.sourceforge.net/svnroot/itext/tags/iText_2_1_7/

JAR File Name	Source Code Repository
js-1.7R1.jar	http://www.mozilla.org/rhino/download.html
mail-1.4.5.jar	http://kenai.com/projects/javamail/downloads/download//javamail-1.4.2-src.zip
poi-3.0.1-jdk122-final-20071014.jar	http://www.uniontransit.com/apache/poi/release/src/poi-src-3.0.1-FINAL-20070705.tar.gz
xml-apis-1.0.b2.jar	http://svn.apache.org/repos/asf/xml/commons/tags/xml-commons-1_0_b2/

JARs exclusive to the embedding samples

JAR File Name	Source Code Repository
hsqldb	svn://hsqldb.svn.sourceforge.net/svnroot/hsqldb
pentaho-reporting-engine-classic-samples	svn://source.pentaho.org/pentaho-reporting/engines/classic/trunk/samples
SDK assembly project	svn://source.pentaho.org/pentaho-reporting/engines/classic/trunk/sdk

More Examples

If you have successfully worked with the first two sample applications and want to see a Pentaho report render in a more realistic user-facing application scenario, then continue on to samples 3 and 4 below. They use the same basic report logic as before, but render interactive reports in a Swing window and a Java servlet that you can deploy into a Web application server like Tomcat or JBoss.

Sample 3: Dynamically Generated, JDBC Input, Swing Output

Sample3.java generates the same report as created in Sample1.java (using the PRPT file generated with Report Designer, connecting to the file-based HSQLDB database, and using a few parameters), but it uses a Swing helper class defined in the Reporting engine to render the report in a Swing preview window. This basic functionality allows for:

- Runtime dynamic changing of report input parameters (in the Swing window, changes to the parameters can be submitted by clicking on the Update button)
- Pagination of the report (showing one page at a time)
- Exporting the report in different formats (PDF, HTML, XLS, etc.)

The details of how to use Swing to preview the report are contained in the following engine classes (see the source files included with the SDK for more information):

- **org.pentaho.reporting.engine.classic.core.modules.gui.base.PreviewDialog**: The dialog window that contains the preview pane and handles basic menu functionality
- **org.pentaho.reporting.engine.classic.core.modules.gui.base.PreviewPane**: The pane that handles the report generation, page switching, printing, and report export functionality

```
/*
 * This program is free software; you can redistribute it and/or modify it
 under the
 * terms of the GNU Lesser General Public License, version 2.1 as
 published by the Free Software
 * Foundation.
 *
 * You should have received a copy of the GNU Lesser General Public
 License along with this
 * program; if not, you can obtain a copy at http://www.gnu.org/licenses/
old-licenses/lgpl-2.1.html
 * or from the Free Software Foundation, Inc.,
 * 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * This program is distributed in the hope that it will be useful, but
 WITHOUT ANY WARRANTY;
 * without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
 PARTICULAR PURPOSE.
 * See the GNU Lesser General Public License for more details.
 *
 * Copyright 2009 Pentaho Corporation.  All rights reserved.
 *
 * Created July 22, 2009
 * @author dkincade
 */
package org.pentaho.reporting.engine.classic.samples;

import java.net.URL;
import java.util.HashMap;
import java.util.Map;

import org.pentaho.reporting.engine.classic.core.ClassicEngineBoot;
import org.pentaho.reporting.engine.classic.core.DataFactory;
import org.pentaho.reporting.engine.classic.core.MasterReport;
import
 org.pentaho.reporting.engine.classic.core.modules.gui.base.PreviewDialog;
import org.pentaho.reporting.libraries.resourceloader.Resource;
import org.pentaho.reporting.libraries.resourceloader.ResourceException;
```

```
import org.pentaho.reporting.libraries.resourceloader.ResourceManager;

/**
 * Generates a report using a paginated Swing Preview Dialog. The
 parameters for this report
 * can be modified while previewing the dialog and the changes can be seen
 instantly.
 * <p/>
 * The report generated in this scenario will be the same as created in
 Sample1:
 * <ol>
 * <li>The report definition file is a .prpt file which will be loaded and
 parsed
 * <li>The data factory is a simple JDBC data factory using HSQLDB
 * <li>There are no runtime report parameters used
 * </ol>
 */
public class Sample3 {

  /**
   * @param args
   */
  public static void main(String[] args) {
    // initialize the Reporting Engine
    ClassicEngineBoot.getInstance().start();

    // Get the complete report definition (the report definition with the
 data factory and
    // parameters already applied)
    Sample3 sample = new Sample3();
    final MasterReport report = sample.getCompleteReportDefinition();

    // Generate the swing preview dialog
    final PreviewDialog dialog = new PreviewDialog();
    dialog.setReportJob(report);
    dialog.setSize(500, 500);
    dialog.setModal(true);
    dialog.setVisible(true);
    System.exit(0);
  }

  /**
   * Generates the report definition that has the data factory and
   * parameters already applied.
   * @return the completed report definition
   */
  public MasterReport getCompleteReportDefinition() {
    final MasterReport report = getReportDefinition();

    // Add any parameters to the report
    final Map<String, Object> reportParameters = getReportParameters();
    if (null != reportParameters) {
      for (String key : reportParameters.keySet()) {
        report.getParameterValues().put(key, reportParameters.get(key));
      }
    }

    // Set the data factory for the report
    final DataFactory dataFactory = getDataFactory();
    if (dataFactory != null) {
      report.setDataFactory(dataFactory);
    }

    // Return the completed report
    return report;
  }

  /**
```

```
   * Returns the report definition which will be used to generate the
 report. In this case, the report will be
   * loaded and parsed from a file contained in this package.
   *
   * @return the loaded and parsed report definition to be used in report
 generation.
   */
  private MasterReport getReportDefinition() {
    try {
      // Using the classloader, get the URL to the reportDefinition file
      // NOTE: We will re-use the report definition from SAMPLE1
      final ClassLoader classloader = this.getClass().getClassLoader();
      final URL reportDefinitionURL = classloader
          .getResource("org/pentaho/reporting/engine/classic/samples/
Sample1.prpt");

      // Parse the report file
      final ResourceManager resourceManager = new ResourceManager();
      resourceManager.registerDefaults();
      final Resource directly =
 resourceManager.createDirectly(reportDefinitionURL, MasterReport.class);
      return (MasterReport) directly.getResource();
    } catch (ResourceException e) {
      e.printStackTrace();
    }
    return null;
  }

  /**
   * Returns the set of runtime report parameters. This sample report uses
 the following three parameters:
   * <ul>
   * <li><b>Report Title</b> - The title text on the top of the report</
li>
   * <li><b>Customer Names</b> - an array of customer names to show in the
 report</li>
   * <li><b>Col Headers BG Color</b> - the background color for the column
 headers</li>
   * </ul>
   *
   * @return <code>null</code> indicating the report generator does not
 use any report parameters
   */
  private Map<String, Object> getReportParameters() {
    final Map parameters = new HashMap<String, Object>();
    parameters.put("Report Title", "Simple Embedded Report Example with
 Parameters");
    parameters.put("Col Headers BG Color", "yellow");
    parameters.put("Customer Names", new String[] { "American Souvenirs
 Inc", "Toys4GrownUps.com", "giftsbymail.co.uk",
        "BG&E Collectables", "Classic Gift Ideas, Inc", });
    return parameters;
  }

  /**
   * Returns the data factory which will be used to generate the data used
 during report generation. In this example,
   * we will return null since the data factory has been defined in the
 report definition.
   *
   * @return the data factory used with the report generator
   */
  private DataFactory getDataFactory() {
    return null;
  }
}
```

Sample 4: Dynamically Generated, JDBC Input, Java Servlet Output

Note: This example assumes you have a Java application server, such as Tomcat or JBoss, installed, configured, running, and accessible to you.

Sample4.java is an HttpServlet which generates an HTML report similar to Sample2 (dynamically created report definition based on the data set, a static data set, and no parameters). In the **generateReport(...)** method, the report is generates as HTML into an output stream which is routed directly to the browser. As noted in the comments of this method, a small simple change can be made to generate PDF output instead of HTML output.

Directions for Running Sample4

To execute Sample4, the following steps will deploy and run it using Tomcat 5.5:

1. Copy Sample4.war into the webapps directory of a working Tomcat instance
2. Start the Tomcat server (bin/startup.sh or bin\startup.bat)
3. In a browser, navigate to the following URL: `http://localhost:8080/Sample4/`

```
/*
 * This program is free software; you can redistribute it and/or modify it
 under the
 * terms of the GNU Lesser General Public License, version 2.1 as
 published by the Free Software
 * Foundation.
 *
 * You should have received a copy of the GNU Lesser General Public
 License along with this
 * program; if not, you can obtain a copy at http://www.gnu.org/licenses/
old-licenses/lgpl-2.1.html
 * or from the Free Software Foundation, Inc.,
 * 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * This program is distributed in the hope that it will be useful, but
 WITHOUT ANY WARRANTY;
 * without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
 PARTICULAR PURPOSE.
 * See the GNU Lesser General Public License for more details.
 *
 * Copyright 2009 Pentaho Corporation.  All rights reserved.
 *
 * Created July 22, 2009
 * @author dkincade
 */
package org.pentaho.reporting.engine.classic.samples;

import java.io.IOException;
import java.io.OutputStream;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.swing.table.AbstractTableModel;

import org.pentaho.reporting.engine.classic.core.ClassicEngineBoot;
import org.pentaho.reporting.engine.classic.core.MasterReport;
import org.pentaho.reporting.engine.classic.core.PageDefinition;
import
 org.pentaho.reporting.engine.classic.core.ReportProcessingException;
import org.pentaho.reporting.engine.classic.core.TableDataFactory;
import org.pentaho.reporting.engine.classic.core.modules.
output.table.html.HtmlReportUtil;
import org.pentaho.reporting.engine.classic.core.wizard.
RelationalAutoGeneratorPreProcessor;
```

```
/**
 * Servlet implementation which generates a report and returns the report
 as an HTML
 * stream back to the browser.
 */
public class Sample4 extends HttpServlet
{
  /**
   * Default constructor for this sample servlet
   */
  public Sample4()
  {
  }

  /**
   * Initializes the servlet - we need to make sure the reporting engine
 has been initialized
   */
  public void init()
  {
    // Initialize the reporting engine
    ClassicEngineBoot.getInstance().start();
  }

  /**
   * Handles the GET request. We will handle both the GET request and POST
 request the same way.
   */
  protected void doGet(final HttpServletRequest req, final
HttpServletResponse resp) throws ServletException, IOException
  {
    generateReport(req, resp);
  }

  /**
   * Handles the POST request. We will handle both the GET request and
 POST request the same way.
   */
  protected void doPost(final HttpServletRequest req, final
HttpServletResponse resp) throws ServletException, IOException
  {
    generateReport(req, resp);
  }

  /**
   * Generates a simple HTML report and returns the HTML output back to
 the browser
   */
  private void generateReport(final HttpServletRequest req, final
HttpServletResponse resp) throws ServletException, IOException
  {
    // Generate the report definition
    final MasterReport report = createReportDefinition();

    // Run the report and save the HTML output to a byte stream
    resp.setContentType("text/html"); // Change to "application/pdf" for
PDF output
    OutputStream out = resp.getOutputStream();
    try
    {
      // Use the HtmlReportUtil to generate the report to a Stream HTML
      HtmlReportUtil.createStreamHTML(report, out);

      //NOTE: Changing this to use PDF is simple:
      // 1. Change the above setContent call to use "application/pdf"
      // 2. Instead of HtmlReportUtil, use the following line:
      // PdfReportUtil.createPDF(report, out)
    }
```

```
    catch (ReportProcessingException rpe)
    {
      rpe.printStackTrace();
    }
    finally
    {
      out.close();
    }
  }

  private MasterReport createReportDefinition()
  {
    // Create a report using the autogenerated fields
    final MasterReport report = new MasterReport();
    report.addPreProcessor(new RelationalAutoGeneratorPreProcessor());

    // Add the data factory to the report
    report.setDataFactory(new TableDataFactory("Sample4Query", new
Sample4TableModel()));
    report.setQuery("Sample4Query");

    // return
    return report;
  }

  /**
   * The table model used for this sample.
   * <br/>
   * In a real situation, this would never happen (a JNDI datasource
connected up to
   * customer data would be more normal). But for a sample, some hard
coded
   * data is to be expected.
   */
  private static class Sample4TableModel extends AbstractTableModel
  {
    /**
     * The sample data
     */
    private static final Object[][] data = new Object[][]
    {
      new Object[] { "Acme Industries", 2500, 18.75 },
      new Object[] { "Brookstone Warehouses", 5000, 36.1245 },
      new Object[] { "Cartwell Restaurants", 18460, 12.9 },
      new Object[] { "Domino Builders", 20625, 45.52 },
      new Object[] { "Elephant Zoo Enclosures", 750, 19.222 },
    };

    /**
     * Returns the number of columns of data in the sample dataset
     */
    public int getColumnCount()
    {
      return data[0].length;
    }

    /**
     * Returns the number of rows in the sample data
     */
    public int getRowCount()
    {
      return data.length;
    }

    /**
     * Returns the data value at the specific row and column index
     */
    public Object getValueAt(int rowIndex, int columnIndex)
    {
```

```
      if (rowIndex >= 0 && rowIndex < data.length && columnIndex >= 0 &&
 columnIndex < data[rowIndex].length)
      {
        return data[rowIndex][columnIndex];
      }
      return null;
    }

  }
}
```