



JasperReports for Java Developers

Create, Design, Format, and Export
Reports with the World's Most Popular
Java Reporting Library

David R. Heffelfinger



Chapter 4

"Creating Dynamic Database Reports"

In this package, you will find:

A Biography of the author of the book

A preview of Chapter 4 -Creating Dynamic Database Reports

A synopsis of the book's content

Information on where to buy this book

About the Author

David R. Heffelfinger has been developing software professionally since 1995; he has been using Java as his primary programming language since 1996. He has worked on many large-scale projects for several clients including Freddie Mac, Fannie Mae, and the US Department of Defense. He has a Masters degree in Software Engineering from Southern Methodist University. David is editor in chief of Ensode.net (<http://www.ensode.net>), a website about Java, Linux, and other technology topics.

I would like to thank everyone at Packt Publishing, particularly Douglas Paterson, Patricia Weir, Nikhil Bangera, and Priyanka Baruah, and the technical reviewer, Thomas Ose. This book wouldn't have been a reality without your help. I would especially like to thank my family for their support.

A special dedication goes to my wife and daughter.

For More Information: http://www.packtpub.com/JasperReports/book
--

4

Creating Dynamic Database Reports

In the previous chapter, we learned how to create our first report. The simple report in the previous chapter contained no dynamic data. In this chapter, we will explore how to create a report from data obtained from a database.

In this chapter, we will cover the following topics:

- How to embed SQL queries into a report definition
- How to pass rows returned by an SQL query to a report via a datasource
- How to use report fields to display data obtained from a database in a report
- How to display data from a database in a report by using the `<textField>` element of the JRXML template

Datasources: Definition



A datasource is what JasperReports uses to obtain data to generate a report. Data can be obtained from databases, XML files, arrays of objects, collections of objects, and XML files.

This chapter focuses on using databases as a datasource. The next chapter discusses other types of datasources.

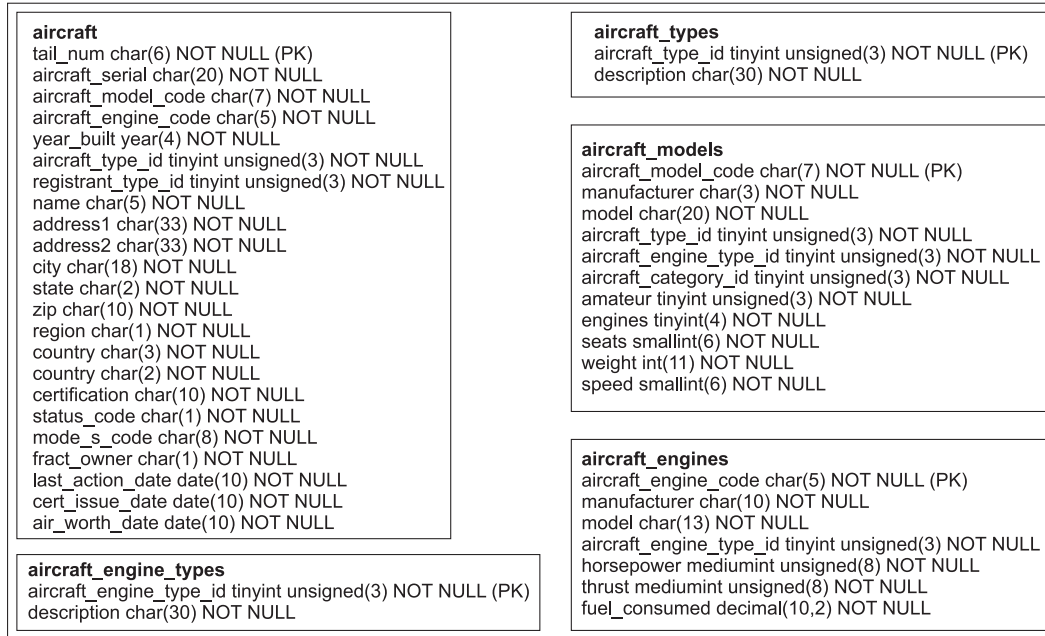
Database for Our Reports


In this chapter, we will use a MySQL database to obtain data for our reports. The database is a subset of public domain data that can be downloaded from <http://dl.flightstats.us>. The original download is 1.3GB. However, we deleted most of the tables and a lot of the data to trim the download size considerably.

For More Information: <http://www.packtpub.com/JasperReports/book>

A MySQL dump of the modified database can be found at this book's website:
<http://www.packtpub.com/support>.

The database contains the following tables: **aircraft**, **aircraft_models**, **aircraft_types**, **aircraft_engines**, and **aircraft_engine_types**. The database structure can be seen in the following diagram:



 The FlightStats database uses the default MyISAM MySQL engine, which does not support referential integrity (foreign keys). That is the reason for which we don't see any arrows in the diagram indicating dependencies between the tables.

Let us create a report that will show the most powerful aircraft in the database, say, those with a horsepower of 1000 or above. The report will show the aircraft tail number, aircraft serial number, the aircraft model, and the aircraft's engine model. The following query will give us these results:

```
select a.tail_num, a.aircraft_serial, am.model as aircraft_model,
ae.model as engine_model
from aircraft a, aircraft_models am, aircraft_engines ae
where a.aircraft_engine_code in ( select aircraft_engine_code
                                from aircraft_engines
```

```

                                where horsepower >= 1000)
and am.aircraft_model_code = a.aircraft_model_code
and ae.aircraft_engine_code = a.aircraft_engine_code

```

Generating Database Reports

There are two ways to generate database reports. It can either be done by embedding SQL queries into the JRXML report template, or by passing data from the database to the compiled report via a datasource. We will discuss both of these techniques.

We will first create the report by embedding the query into the JRXML template. We will then generate the same report by passing it a datasource containing the database data.

Embedding SQL Queries into a Report Template

JasperReports allows us to embed database queries into a report template. This can be achieved by using the `<queryString>` element of the JRXML file. The following example demonstrates this technique:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jasperReport PUBLIC "-//JasperReports//DTD Report Design//EN"
"http://jasperreports.sourceforge.net/dtds/jasperreport.dtd">
<jasperReport name="DbReport">
  <queryString>
    <![CDATA[select a.tail_num,
                  a.aircraft_serial,
                  am.model as aircraft_model,
                  ae.model as engine_model
from aircraft a,
   aircraft_models am,
   aircraft_engines ae
where a.aircraft_engine_code in ( select
                                aircraft_engine_code
                                from aircraft_engines
                                where horsepower >= 1000)
and am.aircraft_model_code = a.aircraft_model_code
and ae.aircraft_engine_code = a.aircraft_engine_code]]>
  </queryString>
  <field name="tail_num" class="java.lang.String"/>
  <field name="aircraft_serial" class="java.lang.String"/>
  <field name="aircraft_model" class="java.lang.String"/>
  <field name="engine_model" class="java.lang.String"/>

```

```
<pageHeader>
  <band height="30">
    <staticText>
      <reportElement x="0" y="0" width="69" height="24"/>
      <textElement verticalAlignment="Bottom"/>
      <text>
        <![CDATA[Tail Number: ]]>
      </text>
    </staticText>
    <staticText>
      <reportElement x="140" y="0" width="79" height="24"/>
      <text>
        <![CDATA[Serial Number: ]]>
      </text>
    </staticText>
    <staticText>
      <reportElement x="280" y="0" width="69" height="24"/>
      <text>
        <![CDATA[Model: ]]>
      </text>
    </staticText>
    <staticText>
      <reportElement x="420" y="0" width="69" height="24"/>
      <text>
        <![CDATA[Engine: ]]>
      </text>
    </staticText>
  </band>
</pageHeader>
<detail>
  <band height="30">
    <textField>
      <reportElement x="0" y="0" width="69" height="24"/>
      <textFieldExpression class="java.lang.String">
        <![CDATA[${F{tail_num}}]>
      </textFieldExpression>
    </textField>
    <textField>
      <reportElement x="140" y="0" width="69" height="24"/>
      <textFieldExpression class="java.lang.String">
        <![CDATA[${F{aircraft_serial}}]>
      </textFieldExpression>
    </textField>
  </band>
</detail>
```

```

</textField>
<textField>
  <reportElement x="280" y="0" width="69" height="24"/>
  <textFieldExpression class="java.lang.String">
    <![CDATA[${F{aircraft_model}}]>
  </textFieldExpression>
</textField>
<textField>
  <reportElement x="420" y="0" width="69" height="24"/>
  <textFieldExpression class="java.lang.String">
    <![CDATA[${F{engine_model}}]>
  </textFieldExpression>
</textField>
</band>
</detail>
</jasperReport>

```

There are a few JXML elements in this example we haven't seen before.

As we mentioned, the `<queryString>` element is used to embed a database query into the report template. As we can see in the example, the `<queryString>` element contains the query to be executed, wrapped in a CDATA block. The `<queryString>` element has no attributes or sub-elements other than the CDATA block containing the query.



Text wrapped inside an XML CDATA block is ignored by the XML parser. As we can see in the example, our query contains the `>` character, which would invalidate the XML if it wasn't inside a CDATA block. A CDATA block is optional, if the data inside it does not break the XML structure. However, for consistency and maintainability, we have used it wherever it is allowed, in the given example.

The `<field>` element defines fields that are populated at run time when the report is filled. Field names must match the column name or alias of the corresponding column in the SQL query. The `class` attribute of the `<field>` element is optional. Its default value is `java.lang.String`. Even though all of our fields are Strings, we still added the `class` attribute for clarity. As can be seen in the example above, the syntax to obtain the value of a report field is `${F{field_name}}`, where `field_name` is the name of the field as defined in its definition.

The next attribute that we haven't discussed before is the `<textField>` attribute. Text fields are used to display dynamic textual data in reports. In this case, we are using them to display the value of the fields. Like all sub-elements of `<band>`, `<textField>` must contain a `<reportElement>` sub-element indicating the text field's height, width, and x, y coordinates within the band. The data that is displayed in text fields is defined by the `<textFieldExpression>` sub-element of `<textField>`. The

<textFieldExpression> has a single sub-element, which is the report expression that will be displayed by the text field, wrapped in an XML CDATA block. In this example, each text field is displaying the value of a field. Therefore, the expression inside <textFieldExpression> uses the <field> syntax explained previously.

Compiling a report containing a query is no different from compiling a report without a query. It can be done either programmatically or through the custom JasperReports JRC ANT task. We covered compiling reports in Chapter 3.

Generating the Report

As mentioned previously, in JasperReports terminology, the action of generating a report from a binary report template is called *filling* the report. To fill a report containing an embedded database query, we must pass a database connection object to the report. The following example illustrates this process:

```
package net.ensode.jasperbook;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.HashMap;

import net.sf.jasperreports.engine.JRException;
import net.sf.jasperreports.engine.JasperFillManager;

public class DbReportFill
{
    Connection connection;

    public void generateReport()
    {
        try
        {
            Class.forName("com.mysql.jdbc.Driver");

            connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/flightstats?user=user&password=secret");

            System.out.println("Filling report...");
            JasperFillManager.fillReportToFile("reports/DbReport.jasper",
                                                new HashMap(), connection);
            System.out.println("Done!");
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```



```

        connection.close();
    }
    catch (JRException e)
    {
        e.printStackTrace();
    }
    catch (ClassNotFoundException e)
    {
        e.printStackTrace();
    }
    catch (SQLException e)
    {
        e.printStackTrace();
    }
}

public static void main(String[] args)
{
    new DbReportFill().generateReport();
}
}

```

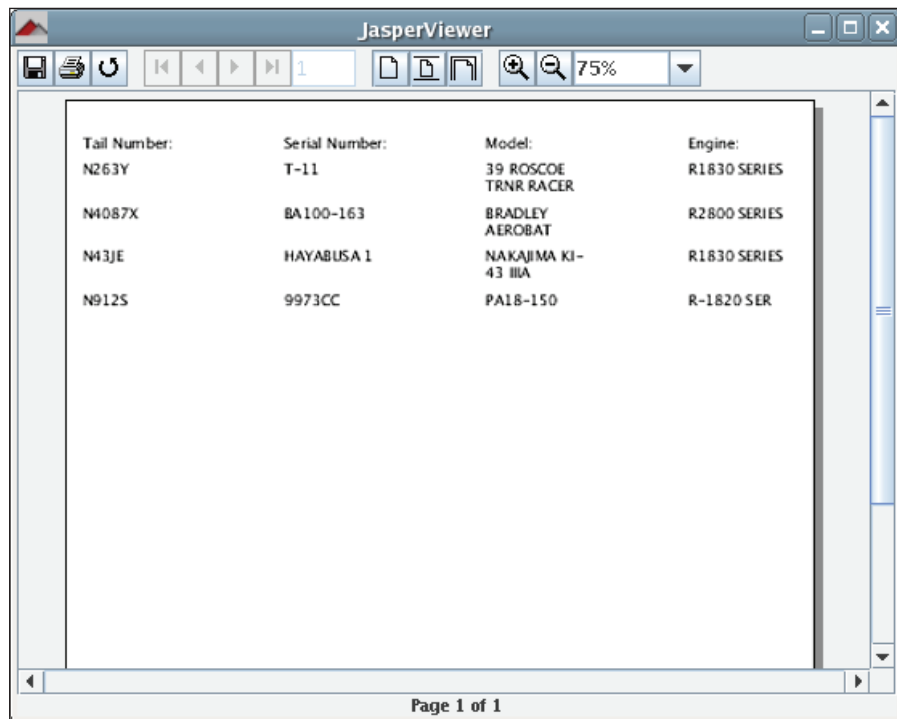
As can be seen in this example, a database connection is passed to the report in the form of a `java.sql.Connection` object as the last parameter of the static `JasperFillManager.fillReportToFile()` method. The first two parameters are of type `String`, used to indicate the location of the binary report template, or Jasper file, and an instance of a class implementing the `java.util.Map` interface, used to pass additional parameters to the report. Since for this report we don't need to pass any additional parameters, we used an empty `HashMap`.

There are six overloaded versions of the `JasperFillManager.fillReportToFile()` method. Three of them take a `Connection` object as a parameter. Refer to Chapter 3 for a description of the other versions of this method that take a `Connection` object as a parameter.



For simplicity, our examples open and close database connections every time they are executed. It is usually a better idea to use a connection pool, since connection pools increase performance considerably. Most Java EE application servers come with connection pooling functionality. The Commons-dbcp component of Jakarta Commons includes utility classes for adding connection pooling capabilities to applications that do not make use of an application server.

After executing the preceding example, a new report, or jrprint file, is saved to disk. We can view it by using the **JasperViewer** utility included with JasperReports.



The **JasperViewer** utility is discussed in detail in Chapter 3.

In this example, we created the report and immediately saved it to disk. The `JasperFillManager` class also contains methods to send a report to an output stream, or to store it in memory in the form of a `JasperPrint` object. Storing the compiled report in a `JasperPrint` object allows us to further manipulate the report. We could, for example, export it to PDF or another format.

The method used to store a report into a `JasperPrint` object is `JasperFillManager.fillReport()`. The method used to send the report to an output stream is `JasperFillManager.fillReportToStream()`. These two methods accept the same parameters as `JasperFillManager.fillReportToFile()`, and are trivial to use once we are familiar with this method. Refer to the JasperReports API for details.

In the next example, we will fill our report and immediately export it to PDF by taking advantage of the `net.sf.jasperreports.engine.JasperRunManager.runReportToPDFStream()` method.

```
package net.ensode.jasperbook;

import java.io.IOException;
import java.io.InputStream;
import java.io.PrintWriter;
import java.io.StringWriter;
import java.sql.Connection;
import java.sql.DriverManager;
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.JasperRunManager;

public class DbReportServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException
    {
        Connection connection;
        ServletOutputStream servletOutputStream =
            response.getOutputStream();
        InputStream reportStream =
            getServletConfig().getServletContext().getResourceAsStream
                ("/reports/DbReport.jasper");

        try
        {
            Class.forName("com.mysql.jdbc.Driver");

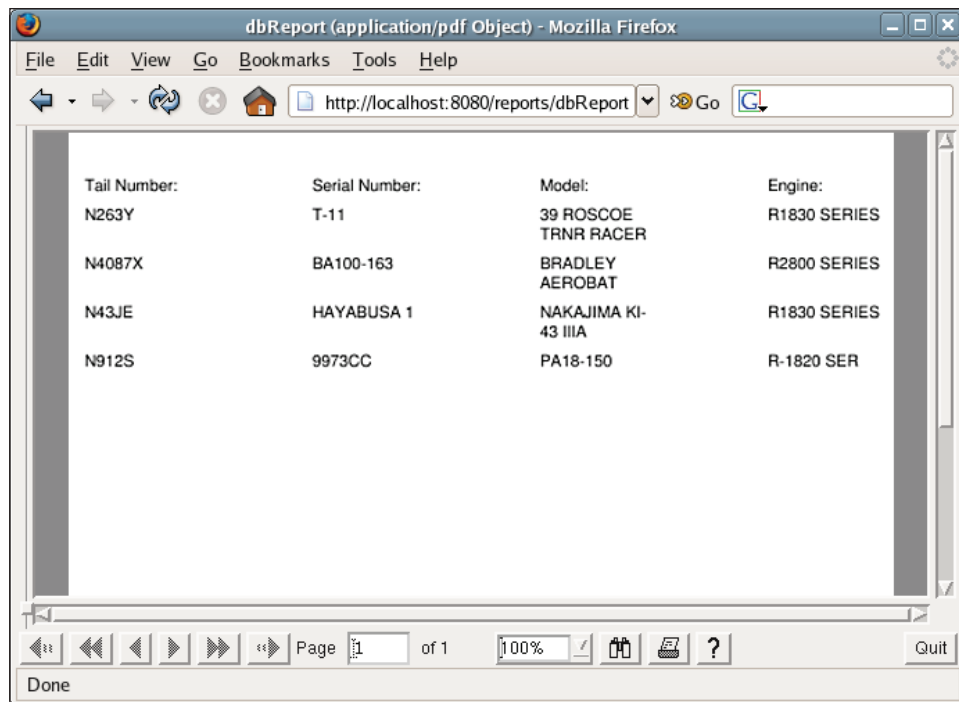
            connection = DriverManager.getConnection ("jdbc:mysql://
                localhost:3306/flightstats?user=dbuser&password=secret");

            JasperRunManager.runReportToPdfStream(reportStream,
                servletOutputStream, new HashMap(), connection);

            connection.close();
        }
    }
}
```

```
        response.setContentType("application/pdf");
        servletOutputStream.flush();
        servletOutputStream.close();
    }
    catch (Exception 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());
    }
}
```

We had already discussed this technique in the previous chapter. The only difference, here, is that we are passing a connection to the report to generate a database report. After deploying this servlet and pointing the browser to its URL, we should see a screen like the following:



Although not directly related to database reporting, one more thing worth mentioning is that we used the `<pageHeader>` element of the JRXML template for laying out the report labels. If our report had more than one page, these labels would have appeared at the top of every page.

Modifying a Report Query via Report Parameters

Embedding a database query into a report template is the simplest way to generate a database report. However, it is not very flexible. If we need to modify the report query, it is also necessary to modify the report's JRXML template.

The sample JRXML template, discussed in the previous section, generates a report that displays all aircraft in the database with a horsepower equal to or greater than 1000. If we wanted to generate a report to display all aircraft with a horsepower greater than or equal to 750, we would have to modify the JRXML and recompile it. That's a lot of work for such a small change. Fortunately, JasperReports allows us to modify an embedded database query easily, by using report parameters. The following JRXML template is a new version of the one we saw in the previous section, modified to take advantage of report parameters.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jasperReport PUBLIC "-//JasperReports//DTD Report Design//EN"
"http://jasperreports.sourceforge.net/dtds/jasperreport.dtd">
<jasperReport name="DbReportParam">
  <parameter name="hp" class="java.lang.Integer"/>
  <queryString>
    <![CDATA[select a.tail_num,
                  a.aircraft_serial,
                  am.model as aircraft_model,
                  ae.model as engine_model
from aircraft a,
   aircraft_models am,
   aircraft_engines ae
where a.aircraft_engine_code in (
                                select aircraft_engine_code
                                from aircraft_engines
                                where horsepower >= $P{hp})
and am.aircraft_model_code = a.aircraft_model_code
and ae.aircraft_engine_code = a.aircraft_engine_code]]>
  </queryString>
  <field name="tail_num" class="java.lang.String"/>
  <field name="aircraft_serial" class="java.lang.String"/>
  <field name="aircraft_model" class="java.lang.String"/>
  <field name="engine_model" class="java.lang.String"/>
```

```
<pageHeader>
  <band height="30">
    <staticText>
      <reportElement x="0" y="0" width="69" height="24"/>
      <textElement verticalAlignment="Bottom"/>
      <text>
        <![CDATA[Tail Number: ]]>
      </text>
    </staticText>
    <staticText>
      <reportElement x="140" y="0" width="79" height="24"/>
      <text>
        <![CDATA[Serial Number: ]]>
      </text>
    </staticText>
    <staticText>
      <reportElement x="280" y="0" width="69" height="24"/>
      <text>
        <![CDATA[Model: ]]>
      </text>
    </staticText>
    <staticText>
      <reportElement x="420" y="0" width="69" height="24"/>
      <text>
        <![CDATA[Engine: ]]>
      </text>
    </staticText>
  </band>
</pageHeader>
<detail>
  <band height="30">
    <textField>
      <reportElement x="0" y="0" width="69" height="24"/>
      <textFieldExpression class="java.lang.String">
        <![CDATA[${tail_num}]]>
      </textFieldExpression>
    </textField>
    <textField>
      <reportElement x="140" y="0" width="69" height="24"/>
```

```

        <textFieldExpression class="java.lang.String">
            <![CDATA[${F{aircraft_serial}}]>
        </textFieldExpression>
    </textField>
</textField>
<reportElement x="280" y="0" width="69" height="24"/>
<textFieldExpression class="java.lang.String">
    <![CDATA[${F{aircraft_model}}]>
</textFieldExpression>
</textField>
<reportElement x="420" y="0" width="69" height="24"/>
<textFieldExpression class="java.lang.String">
    <![CDATA[${F{engine_model}}]>
</textFieldExpression>
</textField>
</band>
</detail>
</jasperReport>

```

The only difference between this JRXML template and the previous one is that we declared a report parameter in the following line:

```
<parameter name="hp" class="java.lang.Integer"/>
```

We then used the declared parameter to dynamically retrieve the horsepower in the where clause of the report query. As can be seen in the example above, the value of a report parameter can be retrieved by using the syntax `${F{paramName}}`, where `paramName` is the parameter name as defined in its declaration (`hp` in the example above).

Passing a parameter to a report from Java code is very simple. In most of the examples we have seen so far, we have been passing an empty `HashMap` to report templates when we fill them. The purpose of that `HashMap` is to pass parameters to the report template. The following servlet is a new version of the one we saw in the previous section, modified to send a report parameter to the report template:

```

package net.ensode.jasperbook;

import java.io.IOException;
import java.io.InputStream;
import java.io.PrintWriter;
import java.io.StringWriter;
import java.sql.Connection;

```

```
import java.sql.DriverManager;
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.JasperRunManager;

public class DbReportParamServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response)
        throws ServletException, IOException
    {
        Connection connection;
        ServletOutputStream servletOutputStream =
            response.getOutputStream();
        InputStream reportStream =
            getServletConfig().getServletContext().getResourceAsStream(
                "/reports/DbReportParam.jasper");
        HashMap parameterMap = new HashMap();
        parameterMap.put("hp", new Integer(750));
        try
        {
            Class.forName("com.mysql.jdbc.Driver");
            connection = DriverManager.getConnection("jdbc:mysql://
                localhost:3306/flightstats?user=dbuser&password=secret");

            JasperRunManager.runReportToPdfStream(reportStream,
                servletOutputStream, parameterMap, connection);

            connection.close();

            response.setContentType("application/pdf");
            servletOutputStream.flush();
            servletOutputStream.close();
        }
    }
}
```



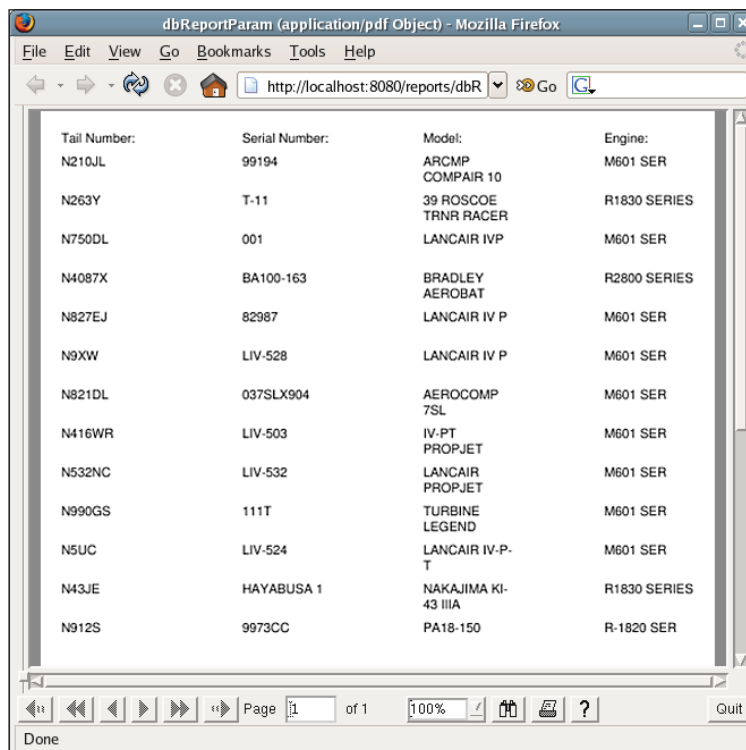
```

    }
    catch (Exception 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());
    }
}
}

```

The only difference between this servlet and the one in the previous section is that we declare a `HashMap` and populate it with the report parameters. Notice how the `HashMap` key must match the report parameter name.

After deploying the servlet and directing the browser to its URL, we should see a report like the following:



Tail Number:	Serial Number:	Model:	Engine:
N210JL	99194	ARCMP COMPAIR 10	M601 SER
N263Y	T-11	39 ROSCOE TRNR RACER	R1830 SERIES
N750DL	001	LANCAIR IVP	M601 SER
N4087X	BA100-163	BRADLEY AEROBAT	R2800 SERIES
N827EJ	82987	LANCAIR IV P	M601 SER
N9XW	LIV-528	LANCAIR IV P	M601 SER
N821DL	037SLX904	AEROCOMP 7SL	M601 SER
N416WR	LIV-503	IV-PT PROPJET	M601 SER
N532NC	LIV-532	LANCAIR PROPJET	M601 SER
N990GS	111T	TURBINE LEGEND	M601 SER
N5UC	LIV-524	LANCAIR IV-P- T	M601 SER
N43JE	HAYABUSA 1	NAKAJIMA KI- 43 IIA	R1830 SERIES
N912S	9973CC	PA18-150	R-1820 SER

Dynamically modifying report queries is only one of the many possible uses of report parameters. Report parameters are discussed in more detail in the next chapter.

Database Reporting via a Datasource

Another way to generate reports based on database data is by using a datasource. In JasperReports terminology, a datasource is a class implementing the `net.sf.jasperreports.engine.JRDataSource` interface.

To use a database as a datasource, the JasperReports API provides the `net.sf.jasperreports.engine.JRResultSetDataSource` class. This class implements `JRDataSource` and has a single public constructor that takes `java.sql.ResultSet` as its only parameter. `JRResultSetDataSource` provides no public methods or variables. To use it, all we need to do is provide a result set to its constructor and pass it to the report via the `JasperFillManager` class.

Let us modify the above JRXML template so that it uses a `JRResultSetDataSource` to obtain database data.

The only change we need to make on the JRXML template is to remove the `<queryString>` element.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jasperReport PUBLIC "-//JasperReports//DTD Report Design//EN"
"http://jasperreports.sourceforge.net/dtds/jasperreport.dtd">
<jasperReport name="DbReportDS">
  <field name="tail_num" class="java.lang.String"/>
  <field name="aircraft_serial" class="java.lang.String"/>
  <field name="aircraft_model" class="java.lang.String"/>
  <field name="engine_model" class="java.lang.String"/>
  <pageHeader>
    <band height="30">
      <staticText>
        <reportElement x="0" y="0" width="69" height="24"/>
        <textElement verticalAlignment="Bottom"/>
        <text>
          <![CDATA[Tail Number: ]]>
        </text>
      </staticText>
      <staticText>
        <reportElement x="140" y="0" width="69" height="24"/>
        <text>
```

```

        <![CDATA[Serial Number: ]]>
    </text>
</staticText>
<staticText>
    <reportElement x="280" y="0" width="69" height="24"/>
    <text>
        <![CDATA[Model: ]]>
    </text>
</staticText>
<staticText>
    <reportElement x="420" y="0" width="69" height="24"/>
    <text>
        <![CDATA[Engine: ]]>
    </text>
</staticText>
</band>
</pageHeader>
<detail>
    <band height="30">
        <textField>
            <reportElement x="0" y="0" width="69" height="24"/>
            <textFieldExpression class="java.lang.String">
                <![CDATA[${F{tail_num}}]>
            </textFieldExpression>
        </textField>
        <textField>
            <reportElement x="140" y="0" width="69" height="24"/>
            <textFieldExpression class="java.lang.String">
                <![CDATA[${F{aircraft_serial}}]>
            </textFieldExpression>
        </textField>
        <textField>
            <reportElement x="280" y="0" width="69" height="24"/>
            <textFieldExpression class="java.lang.String">
                <![CDATA[${F{aircraft_model}}]>
            </textFieldExpression>
        </textField>
        <textField>
            <reportElement x="420" y="0" width="69" height="24"/>
            <textFieldExpression class="java.lang.String">

```

```
        <![CDATA[${F{engine_model}}]>
    </textFieldExpression>
</textField>
</band>
</detail>
</jasperReport>
```

The procedure for compiling a database report, by using `JRResultSetDataSource`, is no different from what we have already seen. To fill the report, we need to execute a database query in our Java code, and pass the query results to the report in a `datasource`, as can be seen in the following example:

```
package net.ensode.jasperbook;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.HashMap;

import net.sf.jasperreports.engine.JRException;
import net.sf.jasperreports.engine.JRResultSetDataSource;
import net.sf.jasperreports.engine.JasperFillManager;

public class DbReportDSFill
{
    Connection connection;
    Statement statement;
    ResultSet resultSet;

    public void generateReport()
    {
        try
        {
            String query = "select a.tail_num, a.aircraft_serial, "
                + "am.model as aircraft_model, ae.model as engine_model from "
                + "aircraft a, "
                + "aircraft_models am, aircraft_engines ae where a.aircraft_ "
                + "engine_code in ("
                + "select aircraft_engine_code from aircraft_engines "
                + "where horsepower >= 1000) and am.aircraft_model_code = "
                + "a.aircraft_model_code "
        }
    }
}
```

```
+ "and ae.aircraft_engine_code = a.aircraft_engine_code";

Class.forName("com.mysql.jdbc.Driver");

connection = DriverManager.getConnection ("jdbc:mysql://
    localhost:3306/flightstats?user=user&password=secret");
statement = connection.createStatement();
resultSet = statement.executeQuery(query);

JRResultSetDataSource resultSetDataSource = new
    JRResultSetDataSource(resultSet);

System.out.println("Filling report...");
JasperFillManager.fillReportToFile("reports/DbReportDS.jasper",
    new HashMap(), resultSetDataSource);
System.out.println("Done!");

resultSet.close();
statement.close();
connection.close();
}
catch (JRException e)
{
    e.printStackTrace();
}
catch (ClassNotFoundException e)
{
    e.printStackTrace();
}
catch (SQLException e)
{
    e.printStackTrace();
}
}

public static void main(String[] args)
{
    new DbReportDSFill().generateReport();
}
}
```

As can be seen in this example, to provide a report with database data, by using `JRResultSetDataSource`, we must execute the database query from the Java code, and wrap the resulting `resultSet` object into an instance of `JRResultSetDataSource`, by passing it to its constructor. The instance of `JRResultSetDataSource` must then be passed to the `JasperFillManager.fillReportToFile()` method. Strictly speaking, any method that takes an instance of a class implementing `JRDataSource` can be called. In this example, we wish to save the report to a file. Therefore, we chose to use the `fillReportToFile()` method. This method fills the report with data from the datasource and saves it to a file in the file system. It has the potential of throwing a `JRException`, if there is something wrong. Therefore, this exception must either be caught, or declared in the `throws` clause.

After executing this code, a report, identical to the first one we saw in the previous section, is generated. The following example demonstrates how a web-based report can be created by using a database datasource:

```
package net.ensode.jasperbook;

import java.io.IOException;
import java.io.InputStream;
import java.io.PrintWriter;
import java.io.StringWriter;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
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.JRResultSetDataSource;
import net.sf.jasperreports.engine.JasperRunManager;

public class DbDSReportServlet extends HttpServlet
{

    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException
```

```
{
    Connection connection;
    Statement statement;
    ResultSet resultSet;

    ServletOutputStream servletOutputStream =
        response.getOutputStream();
    InputStream reportStream = getServletConfig().getServletContext().
        getResourceAsStream("/reports/DbReportDS.jasper");

    try
    {
        String query = "select a.tail_num, a.aircraft_serial, "
            + "am.model as aircraft_model, ae.model as engine_model from "
            + "aircraft a, "
            + "aircraft_models am, aircraft_engines ae where a.aircraft_ "
            + "engine_code in ("
            + "select aircraft_engine_code from aircraft_engines "
            + "where horsepower >= 1000) and am.aircraft_model_code = "
            + "a.aircraft_model_code "
            + "and ae.aircraft_engine_code = a.aircraft_engine_code";

        Class.forName("com.mysql.jdbc.Driver");

        connection = DriverManager.getConnection ("jdbc:mysql:// "
            + "localhost:3306/flightstats?user=dbuser&password=secret");
        statement = connection.createStatement();
        resultSet = statement.executeQuery(query);

        JRResultSetDataSource resultSetDataSource = new
            JRResultSetDataSource(resultSet);

        JasperRunManager.runReportToPdfStream(reportStream,
            servletOutputStream, new HashMap(), resultSetDataSource);

        resultSet.close();
        statement.close();
        connection.close();

        response.setContentType("application/pdf");
        servletOutputStream.flush();
        servletOutputStream.close();
    }
}
```

```
    }  
    catch (Exception 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());  
    }  
}
```

This code is very similar to the previous examples. It executes an SQL query via JDBC and wraps the resulting result set in an instance of `JRResultSetDataSource`. This instance of `JRResultSetDataSource` is then passed to the `JasperRunManager.runReportToPdfStream()` method to export the report to PDF format and stream it to the browser window.

All the examples in this chapter use simple SQL `select` queries to obtain report data. It is also possible to obtain report data from the database by calling stored procedures or functions (if supported by the RDBMS and JDBC driver we are using).

Database Report Methods Compared

Although embedding a database query into a report template is a simple way to create database reports with JasperReports, it is also the least flexible. Using a `JRResultSetDataSource` involves writing some more code, but results in more flexible reports, since the same report template can be used for different datasources.

Depending on our needs, we choose the appropriate method. If we are sure we will always be using a database as a datasource for our report, and that the database query is unlikely to change much, then embedding the database query into the JRXML template at design time is the most straightforward solution. If the query is likely to change, or if we need to use datasources other than a database for our reports, then using a `JRResultSetDataSource` provides the required flexibility.



Some report design tools will only generate database reports by embedding a database query into the report template. If we are using one of these tools, then we have little choice but to use this method. We are free to remove the `<queryString>` element from the JRXML after we are done designing the report and pass the `JRResultSetDataSource` at run time. However, if we do this, we lose the ability to modify the report template from the report designer.

Summary

In this chapter, we covered the different ways through which we can create database reports. We learned to embed SQL queries in a report template by using the `<queryString>` JRXML element as well as to populate an instance of `JRResultSetDataSource` with data from a result set and use it to fill a report.

The chapter also dealt with the procedure to declare report fields to access data from individual columns in the result set of the query used to fill the report. Towards the end of the chapter, we generated reports that are displayed in the user's web browser in PDF format.

JasperReports for Java Developers

JasperReports was started by Teodor Danciu, in 2001, when he was faced with the task of evaluating reporting tools for a project he was working on. The existing solutions that he found were too expensive for his project's budget. Therefore, he decided to write his own reporting tool, JasperReports, which has now become immensely popular, and is currently one of the most popular (if not the most popular) Java reporting tool available.

JasperReports is an open-source Java class library designed to aid developers with the task of adding reporting capabilities to Java applications by providing an API to facilitate the ability to generate reports from any kind of Java application. Though primarily used to add reporting capabilities to web-based applications, it can also be used to create standalone desktop or command-line Java applications for report generation.

This book steers you through each point of report setup, to creating, designing, formatting, and exporting reports with data from a wide range of datasources, and integrating JasperReports with other Java frameworks

What This Book Covers

Chapter 1 covers JasperReports' history, and its features and gives us an overview of the steps involved in generating reports using JasperReports.

Chapter 2 shows us how to embed JasperReports into client and server-side Java applications. We will install JasperReports and learn how to identify and install required libraries. We will also see how to set up our development and execution environment to add reporting capabilities to Java applications.

In *Chapter 3* we create our first static JasperReports both programmatically and by using the ANT tool. We will see how to work with JRXML and binary report templates to generate reports in JasperReports' native format. We will then learn how to view these reports.

In *Chapter 4* we learn how to create dynamic reports. We will do this by embedding SQL queries in the JRXML report template, or by passing the database data to the compiled report via a datasource.

In *Chapter 5* we cover how to use datasources other than databases to create reports. Specifically, we will learn to create reports from empty datasources, Java objects, TableModels, XML data, and also from our custom-created datasources.

In *Chapter 6* we cover how to create elaborate layouts for our reports by adding background images or text to a report, logically grouping report data, conditionally printing report data, and creating subreports.

In *Chapter 7* we cover how to take advantage of JasperReports' graphical features and create reports with graphical data like geometric shapes, images, and 2-D and 3-D charts.

Chapter 8 discusses advanced JasperReports' features like creating crosstab (cross-tabulation) reports and adding anchors, hyperlinks, and bookmarks. We then see how to work with

For More Information: <http://www.packtpub.com/JasperReports/book>

subdatasets and how to execute snippets of Java code by using scriptlets. This chapter also shows how to display report text in different languages.

In *Chapter 9* we cover how to export our reports to all formats supported by JasperReports; these include PDF, RTF, Excel, HTML, CSV, XML, and plain text. We also see how to direct exported reports to a browser.

Chapter 10 covers iReport, which is a report designer that can help us visually generate JRXML templates. This chapter shows how to install and get started with iReport. iReport can be used to do everything that we have covered so far in this book and this chapter shows us how.

Chapter 11 covers the integration of JasperReports with three of the most popular Java web application frameworks around—Spring Web MVC, JavaServer Faces, and Struts. We shall also see how to generate reports with data obtained using Hibernate, which is a popular Java Object Relational Mapping tool.

What You Need for This Book

To use this book, you will of course need JasperReports. This is freely downloadable from <http://www.sourceforge.net/projects/jasperreports>.

JasperReports has its own requirements for proper and successful functioning: Java Development Kit (JDK) 1.4 or newer (<http://java.sun.com/javase/downloads/index.jsp>), ANT 1.6 or newer (<http://ant.apache.org/>), iReport 1.2 or newer (<http://ireport.sourceforge.net/>). Any operating system supporting Java can be used (any modern version of Microsoft Windows, Mac OS X, Linux, or Solaris).

Where to buy this book

You can buy JasperReports for Java Developers from the Packt Publishing website: <http://www.packtpub.com/JasperReports/book>

Free shipping to the US, UK, Europe, Australia, New Zealand and India.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.



www.PacktPub.com

For More Information: <http://www.packtpub.com/JasperReports/book>