

Reports

This document describes the format of the report definition files. Report definitions are described in XML files, which should be placed in a specific directory. The directory location is configurable in the config.xml. The name of a report definition file should always end in ".xml".

All example reports shown here make use of the testdata that ships with xReporter (the tables containing president data).

NOTE

To help in creating a report definition, an XML Schema grammar is available (see xreporter/src/resources/schema/xmlschema/xreporter.xsd). To validate all your reports against this schema, simply enter "xreporter validate " in your xReporter configuration directory (subnote: this command will always use the XML Schema included with xReporter, regardless of any xsi:schemaLocation attributes or DOCTYPE declarations, which will be ignored)

Simple reports

We start with a simple report that executes a select statement on a table. Below the report definition is shown.

```
<report id="sample1">
  <name>sample1.name</name>
  <description>sample1.description</description>
  <required-datasource-type>president</required-datasource-type>
  <catalogs>
    <catalog>samples</catalog>
  </catalogs>
  <output>
    <database>
      <sql>
        <dialect types="default">
          <literal>select Pres_Name, Birth_Yr, Death_Age
            from president</literal>
        </dialect>
      </sql>
      <columns>
        <column id="presname" field="Pres_Name">
          <title>title.presname</title>
          <type base="string" />
        </column>
        <column id="birthyr" field="Birth_Yr">
          <title>title.birthyr</title>
          <type base="long" />
        </column>
        <column id="deathage" field="Death_Age">
          <title>title.deathage</title>
          <type base="long" />
        </column>
      </columns>
    </database>
  </output>
</report>
```

The document element of a report definition is report. This element must have an id attribute, whose value must be a unique identification for that report. id's should always start with a letter, and can then contain letters, numbers, and the characters . (dot), - (dash) and _ (underscore).

Using the name and description elements you can respectively provide a short and a long description which will be shown to the user. These elements do not really contain these texts, instead they contain a resource bundle key. Such a key is used, together with the user's language, to lookup the text from a resource bundle. A resource bundle key can be chosen freely, but may not contain spaces, or the characters '=' and '!'.

The `required-datasource-type` element indicates which type of datasource this report requires. For more information see the description of datasource types in the [datasource documentation](#)¹.

Reports can be grouped in a hierarchical structure of catalogs. Each report can be part of one or more catalogs. These catalogs are specified using one or more catalog elements inside the catalogs element. In the example report only one catalog is specified. The contents of the catalog element should be a slash-separated (/) list of resource bundle keys. The resource bundle keys should of course refer to the names of the catalogs.

The output element describes everything concerned with the output that the report generates. The output element has one child element: `database` or `http`. The `database` element in its turn should contain at least two items: the SQL statement to be executed, and the columns to be displayed. The `http` element will be described later on.

The `sql` element can contain one or more dialect elements. The dialect element allows to specify other SQL statements according to the SQL-dialect supported by the data source (see also the [information in the datasource documentation](#)²). The types attribute of the dialect element should contain a comma-separated list of dialect names. It can also contain the special value "default", to indicate the SQL statement to be executed if none is available for the specific dialect of the selected data source. In the example above only a default SQL statement is defined.

The dialect element contains a list of elements which build the actual SQL statement. Each element will add a piece of SQL to the SQL statement. In this simple example there's only one element, `literal`. The content of a `literal` element is inserted literally in the SQL statement.

Finally, we reach the `columns` element. This element describes what columns should be fetched from the result of the SQL statement, and what the datatype of each of these columns is. The `columns` element contains a number of column elements. The field attribute of the column element determines the field to be fetched from the result of the query. The `id` attribute is optional (xReporter will assigns id's itself of the form `col1`, `col2`, ... if not specified). Additionally the datatype must be specified. Using the attribute `datatype-id` on the column element you can refer to a datatype that is declared in the datatype catalog. In the example report however another technique is used: the datatype is declared inline. The syntax is the same as in the datatype catalog. See the [datatype documentation](#)³ for more information on this.

The access to a report can be limited based on the roles of the user. The roles that have permission to use a report are not defined in the report definition itself, but these are managed in a database table. The reason for this is that assigning permissions is a separate responsibility from creating report definitions. A database table is also easier to manage (easy update/insert/delete commands). The table contains two columns: `report-id` and `rolename`. For each role that has the right to access a certain report, a record should be inserted in that table (see also the [administration documentation](#)⁴). The exact table and column names, and the datasource containing the table are configurable in the `config.xml`.

Summary: to make a report usable, the following conditions must be fulfilled:

- The report must be stored in a file with extension ".xml" in the directory configured in the `config.xml` (usually the directory called "reports" in you xReporter configuration directory)

1. daisy:84-cd (Data Sources)
2. daisy:84-cd (Data Sources)
3. daisy:85-cd (Datatypes)
4. daisy:88-cd (Administration)

- The report should not contain errors. If a report contains errors (for example: it's not a well-formed XML file), then these will be logged. In the default xReporter log configuration, this is logged to the file called xreporter-{currentdate}.log. You can also check your reports using the "xreporter validate" command from within your xReporter configuration directory.
- The report must be present in at least one catalog, otherwise the user cannot select it.
- The user must select a datasource that supports the type that is required by the report, as specified in the required-datasource-type element.
- At least one of the roles of the user should provide access to the report.

xReporter will scan the reports directory periodically to see if there were any changed, added or removed report files. By default this is done every 10 seconds. During development, you may want to lower that value to e.g. 5 seconds. This can be done in the config.xml file (after which you will need to redeploy to Phoenix using the "xreporter deploy" command, as described in the [deployment documentation](#)⁵).

Sorting

The output of the sample1 report was not sorted, as a consequence the rows appeared in an arbitrary order. To sort the output we could add an order by clause to the select statement, such as:

```
<sql>
  <literal>select Pres_Name, Birth_Yr, Death_Age from president order by
    Pres_Name</literal>
</sql>
```

xReporter however offers an alternative, whereby the user can select the sort order himself (or herself). There are two possibilities for this: the user can either choose from a list of predefined sort orders, or can define the sort order himself by selecting the columns on which he wants to sort.

Firstly, the definition of the SQL statement has to be extended using the orderby element. The orderby element determines where in the SQL statement that the order by clause should be inserted. Here's the relevant fragment from the report definition:

```
<sql>
  <literal>select Pres_Name, Birth_Yr, Death_Age from president</literal>
  <orderby/>
</sql>
```

By doing just this the user will already be able to assemble a sort order himself, but the report will not be sorted by default.

Below is the sample2 report, which is an extension of the sample1 report but with predefined and default sort orders.

```
<report id="sample2">
  <name>sample2.name</name>
  <description>sample2.description</description>
  <required-datasource-type>president</required-datasource-type>
  <catalogs>
    <catalog>samples</catalog>
  </catalogs>
  <output>
    <database>
      <sql>
        <dialect types="default">
          <literal>select Pres_Name, Birth_Yr, Death_Age from president</literal>
```

5. daisy:81-cd (config/build/deploy process)

```

    <orderby/>
  </dialect>
</sql>
<columns>
  <column id="presname" field="Pres_Name">
    <title>title.presname</title>
    <type base="string"/>
  </column>
  <column id="birthyr" field="Birth_Yr">
    <title>title.birthyr</title>
    <type base="long"/>
  </column>
  <column id="deathage" field="Death_Age">
    <title>title.deathage</title>
    <type base="long"/>
  </column>
</columns>
<orderby-choices default="samplesort">
  <orderby-choice id="samplesort">
    <description>sample2.sort.deathage-presname</description>
    <sql>Death_Age, Pres_Name</sql>
  </orderby-choice>
</orderby-choices>
</database>
</output>
</report>

```

Note the new `orderby-choices` element. This element can contain one or more `orderby-choice` elements, that each have an `id` attribute. The value of the `id`'s are subject to the same rules as the `id`'s for report definitions. Each `orderby-choice` element contains a `description` element (which contains a resource bundle key) and a `sql` element. The `sql` element should contain a SQL-fragment that is inserted literally in the SQL statement on the location of the `orderby` element, without the text "order by". This way, it is also possible to sort on columns not visible to the user.

Using the default attribute of the `orderby-choices` element you can specify which sort order should be used by default. The default attribute is not required.

Enabling/Disabling `orderby-choices` depending on what columns are currently selected

It is possible to disable `orderby` choices based on what columns are currently selected (= visible):

```
<orderby-choice id="foo" if-columns-selected="col1 col4">
```

Forcing to sort on specific columns

If you want to make sure that the report is sorted on certain columns, next to the user-selected columns, you can add an `append` attribute on the `<orderby>` SQL part:

```
<orderby append="col5, col6"/>
```

The value of the `append` attribute will be added after the normal `orderby` selection (if any), the leading comma will be inserted as necessary.

Query By Example

Query By Example (QBE) is the functionality whereby the user refines the result of the report by specifying conditions in a grid. Conditions that are on the same line are AND'ed together, conditions on multiple lines

are OR'ed together. xReporter supports QBE. To activate this two things need to be done in the report definition: indicate where the conditions should be inserted in the SQL statement, and indicate on what columns QBE is allowed.

Below is the sample3 report, which allows QBE on all columns:

```
<report id="sample3">
  <name>sample3.name</name>
  <description>sample3.description</description>
  <required-datasource-type>president</required-datasource-type>
  <catalogs>
    <catalog>samples</catalog>
  </catalogs>
  <output>
    <database>
      <sql>
        <dialect types="default">
          <literal>select Pres_Name, Birth_Yr, Death_Age from president
            where 1=1</literal>
          <qbe/>
          <orderby/>
        </dialect>
      </sql>
      <columns>
        <column id="presname" field="Pres_Name" allow-qbe="true">
          <title>title.presname</title>
          <type base="string"/>
        </column>
        <column id="birthyr" field="Birth_Yr" allow-qbe="true">
          <title>title.birthyr</title>
          <type base="long"/>
        </column>
        <column id="deathage" field="Death_Age" allow-qbe="true">
          <title>title.deathage</title>
          <type base="long"/>
        </column>
      </columns>
      <orderby-choices default="samplesort">
        <orderby-choice id="samplesort">
          <description>sample3.sort.deathage-presname</description>
          <sql>Death_Age, Pres_Name</sql>
        </orderby-choice>
      </orderby-choices>
    </database>
  </output>
</report>
```

Note that each column element now has an attribute `allow-qbe` with the value `true`. If this attribute is not mentioned the default is `false`. Next to that the element `qbe` is now added inside the `dialect` element. At the location of this element the conditions will be inserted. If the QBE is not used and thus no conditions are specified, nothing will be inserted. Otherwise something starting with "AND" will be inserted. Therefore, the literal element before the `qbe` element is changed with the addition "where 1=1".

When the resultset contains columns with the same name (thus from different tables), `qbe` will not work on these columns. In that case you have to use the attribute `canonical`. In this attribute you can put the tablename and the columnname. For example: `president.Pres_Name`. If the attribute `canonical` exists, `qbe` uses the value of `canonical` instead of the value of `field`.

Extending datatypes

In the examples up to now the datatypes of the columns are always put inline within the element. It was also mentioned already that you can use the `datatype-id` attribute to refer to a datatype that is declared in the

datatype catalog. In that case the content of the column element should be empty. To extend from an existing datatype, simply combine both: use the datatype-id attribute to refer to an existing datatype, and add extra elements in the column element to redefine or add properties of the datatype. In this way, a new anonymous datatype will be created that extends from the datatype referred to in the datatype-id attribute.

Steps

Interaction-steps (for parameterized reports)

In the previous section we discussed how using the QBE the user can filter the report output. It is also possible to ask parameters and conditions before the output of the report is shown. These parameters can be asked in one or more interaction steps. In the next section we'll also see how in between these interaction steps SQL statements can be executed to create temporary tables.

The difference between a parameter and a condition is that a parameter is just a value, while a condition is the combination of an operator (less than, equals, ...) and a value.

Below is the report sample4, which is an extension of the sample3 report. An interaction step is added which adds a required condition on the field birth year.

```
<report id="sample4">
  <name>sample4.name</name>
  <description>sample4.description</description>
  <required-datasource-type>president</required-datasource-type>
  <catalogs>
    <catalog>samples</catalog>
  </catalogs>
  <flow>
    <step id="step1">
      <interaction>
        <condition id="birthyr" required="true">
          <prompt>prompt.birthyr</prompt>
          <title>prompt.birthyr</title>
          <type base="long"/>
        </condition>
      </interaction>
    </step>
  </flow>
  <output>
    <database>
      <sql>
        <dialect types="default">
          <literal>select Pres_Name, Birth_Yr, Death_Age from president
            where 1=1</literal>
          <condition idref="birthyr" field="Birth_Yr"/>
          <qbe/>
          <orderby/>
        </dialect>
      </sql>
      <columns>
        <column id="presname" field="Pres_Name" allow-qbe="true">
          <title>title.presname</title>
          <type base="string"/>
        </column>
        <column id="birthyr" field="Birth_Yr" allow-qbe="true">
          <title>title.birthyr</title>
          <type base="long"/>
        </column>
        <column id="deathage" field="Death_Age" allow-qbe="true">
          <title>title.deathage</title>
          <type base="long"/>
        </column>
      </columns>
    </database>
  </output>
</report>
```

```

</columns>
<orderby-choices default="samplesort">
  <orderby-choice id="samplesort">
    <description>sample4.sort.deathage-presname</description>
    <sql>Death_Age, Pres_Name</sql>
  </orderby-choice>
</orderby-choices>
</database>
</output>
</report>

```

New in this example is the flow element. It contains one or more step elements. Each step element should have an id attribute. That id is subject to same rules as id's for report definitions (see earlier). A step element contains either an interaction element, or an execute element, or a combination of both. Currently we'll only look at the interaction element. An interaction element can contain one or more condition and/or parameter elements. Parameters are not used in this example, but are used in the following example. The condition element should have an id attribute. It can optionally have an attribute "required", indicating whether the condition is required. By default this is true. Inside the condition element there's again a description of the datatype. It's again possible to use a datatype-id attribute to refer to an existing datatype, or a combination of a datatype-id attribute and child elements to extend from an existing datatype.

The consequence of the addition of the flow element in the example above is that after selection of the report, the user will now be presented with a form where (s)he must fill in the condition.

To effectively use the condition in the SQL statement a condition element is used inside the dialect element. The condition element then needs to have an idref attribute, whose value must correspond with the id attribute of a condition asked during an interaction step. Next to that the field attribute must indicate on what field the condition should be applied. Suppose that in the above example the user selected the operator 'less than' and entered the value 1700. In this case the following will be inserted at the location of the condition element: "AND Birth_YR < 1700".

Suppose now that we don't want the user to select the operator himself. It should always be "less than". In that case a parameter can be used. Let's also suppose that the user is not required to fill in the parameter. The flow-part of the report now becomes:

```

<flow>
  <step id="step1">
    <interaction>
      <parameter id="birthyr" required="false">
        <prompt>prompt.birthyr</prompt>
        <title>prompt.birthyr</title>
        <type base="long"/>
      </parameter>
    </interaction>
  </step>
</flow>

```

And the sql-part could become the following:

```

<sql>
  <dialect types="default">
    <literal>select Pres_Name, Birth_Yr, Death_Age from president
      where l=1 and Birth_Yr &lt; </literal>
    <parameter idref="birthyr"/>
    <qbe/>
    <orderby/>
  </dialect>
</sql>

```

The parameter tag only adds a value in the SQL statement. Since the parameter is optional this could leave a gap in the SQL statement in case the user has not filled in a value. Therefore, two attributes are available on the parameter element: "sql-before" and "sql-after", which can contain pieces of SQL that will only be inserted if the parameter has a value. The sql-part now becomes as follows:

```
<sql>
  <dialect types="default">
    <literal>select Pres_Name, Birth_Yr, Death_Age from president
      where 1=1</literal>
    <parameter idref="birthyr" sql-before="and Birth_Yr &lt;"/>
    <qbe/>
    <orderby/>
  </dialect>
</sql>
```

In the examples above only one condition or parameter was used, but that could as well have been more than one.

Interaction step validation

The values of parameters and conditions are validated according to their datatype configuration. However, it can also be useful to do cross-field validation, such as checking that the value of one parameter is less than the value of another parameter, or that the time between two dates does not exceed a certain number of days. This is possible with step-level validation.

Step-level validation is configured as follows:

```
<flow>
  <step id="step1">
    <interaction>

      [... one or more parameters and conditions ... ]

    <validation>
      <java class="..."/>
      [... more validators ...]
    </validation>

  </interaction>
</step>
</flow>
```

The only currently supported validation mechanism is by means of a Java class. This Java class should implement the following interface:

```
org.outerj.xreporter.report.definition.StepInteractionValidator
```

See the javadoc of that interface for more details.

An example of all this can be found in the sample report `pres_step_validation`.

Execute-steps (for creation of temporary tables)

Sometimes a query is too complex to execute in one SQL statement. In those cases it's possible to first create one or more temporary tables. xReporter will make sure that these tables have unique names, and that they are removed automatically when the report instance expires.

For queries that cannot be solved by executing a few preparatory SQL statements, it is also possible to plug in a custom Java class to create a temporary table(s).

An example of executing a preparatory SQL statement is shown in the following report. The aim of the report is to find out which hobby is practiced most by the married presidents.

```
<report id="favhobby">
  <name>favhobby.name</name>
  <description>favhobby.description</description>
  <required-datasource-type>president</required-datasource-type>
  <catalogs>
    <catalog>catalog.presidents/catalog.lists</catalog>
    <catalog>catalog.presidents/catalog.USA/catalog.allreports</catalog>
  </catalogs>
  <flow>
    <step id="step1">
      <execute>
        <sql>
          <dialect types="default">
            <literal>create table</literal>
            <temptable id="married_pres"/>
            <literal>select distinct(Pres_Name) from pres_marriage</literal>
          </dialect>
        </sql>
      </execute>
    </step>
  </flow>
  <output>
    <database>
      <sql>
        <dialect types="default">
          <literal>select hobby, count(*) hobbycount from
            pres_hobby,</literal>
          <temptable idref="married_pres"/>
          <literal>marriedpres where pres_hobby.Pres_Name =
            marriedpres.Pres_Name group by hobby order by hobbycount
            desc</literal>
        </dialect>
      </sql>
      <columns>
        <column field="hobby" datatype-id="basic-string">
          <title>favhobby.title.hobby</title>
        </column>
        <column field="hobbycount" datatype-id="basic-long">
          <title>favhobby.title.hobbycount</title>
        </column>
      </columns>
    </database>
  </output>
</report>
```

The step element now contains an execute element. The execute element can contain multiple actions. Either it are sql elements that describe a SQL statement to be executed, or it are java elements that describe a java-class to be executed, or a combination of both. In the above example, there's just one sql element, but it could be multiple ones.

New inside the sql element is the temptable element. The id attribute assigns a logical name to the temporary table. The final name of the temporary table that will be created in the database has the form tmp_{report-instance-id}_{temptable-id} (the parts between brackets are variable). The temptable-id can be freely chosen (but make sure it only contains characters allowed in table names, and keep it as short as possible). The report-instance-id is a unique id assigned by xReporter to each report instance. It has the form {server-id}_{sequence-number}. Herein the server-id a user-configurable string that identifies the xReporter server (configurable in config.xml, it's purpose is to avoid naming conflicts if multiple xReporter servers use the same database). The sequence-number is an automatically generated sequence number.

By using the temptable-tag with the id attribute, the name of that table will also be remembered by xReporter. xReporter will clean the table when the report instance expires. At the location of the temptable-tag, the name of the temporary table will of course be inserted in the SQL statement.

In the output-part of the report the temptable element is used again, but this time with an idref attribute. This simply inserts the name of the temporary table at that location, so that can be referred to the earlier created temporary table.

Custom Java classes

In cases where more complex logic is needed, a custom Java class can be implemented. This Java class should implement the Executable interface:

```
package org.outerj.xreporter.report.definition;

public interface Executable
{
    public void execute(ExecutionContext executionContext) throws Exception;
}
```

The execute method will be called by xReporter. The parameter executionContext provides, among others, access to the current datasource of the report and to the parameters and conditions entered by the user. There are also methods to retrieve a name for a temporary table, and to register that temporary table for automatic cleanup by xReporter.

As an example, you'll find the sample5 report below. This report first asks a number from the user. In the execute-part a Java class is then called.

```
<report id="sample5">
  <name>sample5.name</name>
  <description>sample5.description</description>
  <required-datasource-type>president</required-datasource-type>
  <catalogs>
    <catalog>samples</catalog>
  </catalogs>
  <flow>
    <step id="step1">
      <interaction>
        <parameter id="count" required="true">
          <prompt>prompt.count</prompt>
          <type base="long">
            <validation>
              <minInclusive value="3"/>
              <maxExclusive value="20"/>
            </validation>
          </type>
        </parameter>
      </interaction>
      <execute>
        <java class="sample.SampleCustomExecute"/>
      </execute>
    </step>
  </flow>
  <output>
    <database>
      <sql>
        <dialect types="default">
          <literal>select data from</literal>
          <temptable idref="sample"/>
          <qbe/>
          <orderby/>
        </dialect>
      </sql>
```

```

<columns>
  <column id="data" field="data" allow-qbe="true">
    <title>title.data</title>
    <type base="long"/>
  </column>
</columns>
</database>
</output>
</report>

```

Source code of the sample.SampleCustomExecute class:

```

package sample;

import org.outerj.xreporter.report.definition.Executable;
import org.outerj.xreporter.report.definition.ExecutionContext;
import org.outerj.xreporter.report.instance.Parameter;

import java.sql.PreparedStatement;
import java.sql.Connection;

public class SampleCustomExecute implements Executable
{
    public void execute(ExecutionContext executionContext) throws Exception
    {
        String tempTableName = executionContext.getTemporaryTableName("sample");
        executionContext.registerTemporaryTableName(tempTableName);
        int count = ((Long)((Parameter)executionContext
            .getInputField("count")).getValue()).intValue();

        Connection conn = null;
        PreparedStatement stmt = null;

        try
        {
            conn = executionContext.getDataSource().getConnection();
            conn.setAutoCommit(true);

            stmt = conn.prepareStatement("create table " + tempTableName
                + " (data integer(10))");
            stmt.execute();
            stmt.close();

            stmt = conn.prepareStatement("insert into " + tempTableName
                + " values (?)");
            for (int i = 1; i <= count; i++)
            {
                stmt.setString(1, String.valueOf(i));
                stmt.execute();
            }
        }
        finally
        {
            if (stmt != null) try { stmt.close(); } catch (Exception e) {}
            if (conn != null) try { conn.close(); } catch (Exception e) {}
        }
    }
}

```

Consult the xReporter javadoc for more information on the various xReporter interfaces such as Executable.

The Java class is considered single threaded. This means that never more than one thread will access a certain instance of the class. The instances are not reused.

If desired, the Java class can implement a number of Avalon interfaces. Avalon is a component framework on which xReporter is based internally. The interfaces that can be implemented by the Java class are:

LogEnabled, Composable, Configurable (the java element from the report definition will be passed as configuration) and Initializable. See the Avalon documentation for more information on these interfaces.

To compile the custom Java class the xreporter-all.jar file can be added to the CLASSPATH. This xreporter-all.jar file will be created during building xReporter and can be found in the build directory. However, the default xReporter build system will automatically build your Java classes if you put them below conf.home/src/java. When running "ant phoenix-deploy", it will then compile that code and put it in the jar lib/user-extensions.jar. This jar is by default referenced from the config.xml, so you don't need to do anything else.

Linking between reports

Linking between reports allows to have a kind of "drill down" functionality. The links are defined in the report definition. Links can either be attached to columns (in which case the values in the column will be links), or not (in which case the links will be shown in additional columns).

The link is defined by giving the report id of the report to link to, and (optionally) specifying a number of parameters for the report. Optionally a title can be defined for the link (which will usually be shown in a tooltip when moving over the link), as well as the datasource if it is different from the current one.

Here's an example of how to define a link on a column:

```
[...]
<column id="party" field="Party">
  <title>title.party</title>
  <type base="string"/>

  <link>
    <title>linktitle.show-details</title>
    <report-id>pres_party_detail</report-id>
    <parameter id="party" column-id="party"/>
  </link>
</column>
[...]
```

Most of the configuration speaks for itself. The content of the title element should be a resource bundle key. The parameter element can occur multiple times. Its id attribute specifies the id of a parameter in the destination report. The step in which this parameter occurs does not matter. The column-id attribute refers to one of the columns in the current report (in this example the current column) whose value will be used for the parameter.

As said before, links can also be defined separately from columns. Here's an example:

```
[...]
</columns>
<links>
  <link>
    <title>linktitle.show-details</title>
    <report-id>pres_party_detail</report-id>
    <parameter id="party" column-id="party"/>
  </link>
</links>
[...]
```

If you need to specify a different datasource, use an element <datasource-id> within the <link> element.

Reports with grouping

NOTE

Grouping is only available in xReporter versions > 1.0 Note Reports with grouping don't support chunking. Their output is always shown on one page. They do support the other common features though, like sorting and filtering.

The output of the reports we discussed until now was always a simple table structure. In this section, we're going to look at how we can use "grouping" to group similar values together and add summaries in between them. If you don't know what I'm talking about, have a look at the example report titled " Presidents with grouping" (available in the online demo at <http://xreporter.cocoondev.org>⁶).

To create a group, you need to specify the column whose values need to be grouped. xReporter will then automatically sort the report on that column, so that values which are the same are placed together. There is always one "top level group", this is a group that contains the whole table and is thus not associated with any column. The basic way in which groups are declared in a report definition is as follows:

```
[...]
<output>
  <database>
    [...]
    <group>
      <group column="aValidColumnId">
        <group column="anotherValidColumnId" />
      </group>
    </group>
  </database>
</output>
[...]
```

The outermost <group>-tag represents the top level group. It can contain nested <group>-tags which should have a "column" attribute. The value of this attribute should be the ID of one of the columns defined earlier in the report. So the example above means: first group on the values of the column with id "aValidColumnId", then group on the values of the column with id "anotherValidColumnId".

NOTE

When defining groups, xReporter will automatically sort on the columns corresponding with the groups. This requires that the <dialect>-element contains a <orderby/> tag, otherwise this will not work as expected.

Of course, just grouping the values is not so useful in itself, so lets add some summaries to them. Here's an example of how to declare them:

```
[...]
<group>
  <group column="aValidColumnId">
    <summaryfield id="someSum" expression='SSum(Column("aValidColumnId"))'
      label="summaryfield.sum" align-on-column="aValidColumnId">
      <type base="bigdecimal" />
    </summaryfield>
  <group column="anotherValidColumnId">
    <summaryfield id="someAverage" expression='SAvg(Column("anotherValidColumnId"))'
      label="summaryfield.avg" align-on-column="anotherValidColumnId">
      <type base="bigdecimal" />
    </summaryfield>
  </group>
</group>
</group>
```

6. <http://xreporter.cocoondev.org/>

A summary field is declared with the `<summaryfield>` tag, which takes the following attributes:

- **id** an id for this summary field. This should be unique within a `<group>` tag, but the same id can be reused in different `<group>` tags.
- **expression** an expression defining what should be calculated. More information on these expression follows below. Note that we use single quotes for this attribute, because double quotes are needed in expressions to define strings.
- **label** a label that will be displayed before the calculated value. As always, this is a resource bundle key. This attribute is optional.
- **align-on-column** an optional attribute that specifies the ID of the column on which this summary value should be aligned. If not specified, the summaryfield will be placed on a separate line spanning all columns (at least, that's what the default xReporter stylesheets do).

Although not shown in the above example, multiple summaryfields can be defined for each group. Summary fields can also be defined for the top-level group, providing the ability to add "grand totals" to the report. It is also legal to have only a top level group and no sub-groups.

As usual, each summaryfield should also indicate its type, either using child-elements within the summaryfield, or using a `datatype-id` attribute on the summaryfield element.

For reasons of convenience in writing reports and custom stylesheets there have been some (backwards) compatible additions to the `<group>` section of report-definition files. These are only available on versions of xreporter > 1.2.1. The additions allow to:

- specify a group that groups on the values of multiple columns.
- specify some implicit summaryfield on the level of the group itself.

Grouping on multiple columns

Letting your group match more then one column can be done by replacing the `group/@column` attribute with nested `<column>` elements in the group's structure:

```
[...]
<group><!-- never forget the grouping-all-records group -->
  <group>
    <columns>
      <column idref="aValidColumnId" />
      <column idref="anotherValidColumnId" />
    </columns>
    <!-- with more room for nested declared <group> or <summaryfield>'s -->
  </group>
</group>
```

Implicit group-associated summaryfield

As mentioned: the logical use of the grouping functionality is to add summaryfields to them. Because of this xreporter now carries an implicit summaryfield to every `<group>` you define. To use it, you can just add the `@label` and `@expression` attributes known from the `<summaryfield>` directly to the `<group>` element. (`@id` and `@align-on-column` have no meaning here). If the `@expression` is used, then the nested `<type>` specifying the return-type of the expression should be placed as the first child element of the `<group>`.

```
[...]
<group>
  <group label="label.group.party_and_stateborn"
    expression='Concat( "[" , Column("party"), " , ",
      Column("stateborn") , "]. Names from ", SFirst(Column("presname")),
```

```
    " to " , SLast(Column("presname")) )' >
<type base="string" />
<columns>
  <column idref="party" />
  <column idref="stateborn" />
</columns>
<!-- possibly more nested summaryFields and/or groups -->
</group>
</group>
```

NOTE

The @label will be ignored if no @expression is present, since it is assumed to be the description of the expression's return-value.

NOTE

These new attributes to the <group> cannot be used in combination with the @column; they enforce the usage of the nested <columns >.

Summary expressions

The expressions are the same as the ones used in other places in xReporter, and their general functionality is described in a [separate document](#)⁷. There are however some special functions only available in summaries, described here.

SSum(number) : number

Calculates a "Summary Sum". Its argument should be an expression returning a number. This expression will be evaluated for each row in the current group, and the results of each evaluation are added together. To do anything useful it should contain the Column function.

Column(string)

This function should only be used as an argument to one of the summary functions such as SSum or SAvg. It takes as parameter the ID of a column. The Column function will return the value of that column for the current row in the group. Remember from the description of the SSum function that a summary function will evaluate its argument for each row in the current group, and the Column function thus knows what the current row is, and is able to retrieve a value from it. The type of the return value depends on the column.

SAvg(number) : number

Summary function calculating the average of the values in a group. See also the description of SSum.

SMin(number) : number

Summary function returning the minimum of the values in a group. See also the description of SSum.

SMax(number) : number

Summary function calculating the maximum of the values in a group. See also the description of SSum.

7. daisy:91-cd (Expression Language)

SCount() : number

Summary function returning the number of rows in the current group.

SFirst(anytype) : anytype

Summary function returning the enclosed expression executed on the context of the first row of the group.

SLast(anytype) : anytype

Summary function returning the enclosed expression executed on the context of the last row of the group.

NOTE

The functions SFirst() and SLast() can be used as a more performant choice over SMax() or SMin(). However they assume that you properly control the ordering of the resultset. If not use the SMax and SMin alternatives.

Some expression tricks

By allowing any subexpression as argument to summary functions, some powerful effects can be achieved. For example, in the "President Grouping" example report we count the number of presidents older then 80 using the following expression:

```
SSum(If(Column("deathage") > 80, 1, 0))
```

Reports with HTTP-output

Until now, all the sample report definitions did queries on databases. To allow integration with existing applications, xReporter can also execute a query on an HTTP-accessible service.

In this case the database element in the report definition is replaced by an http element. All other functionality remains the same.

The query on the HTTP server consists of a GET operation of a certain path, with additionally a dynamically generated query string. The query string can contain information on the data source, the user and the current report. The result of the GET operation should be a well-formed XML document.

An example is the "orbitarium" report shown below. It uses a public webservice that offers information on the position of the planets. A parameter is used to ask the user the name of a planet.

```
<report id="orbitarium">
  <name>orbitarium.name</name>
  <description>orbitarium.description</description>
  <required-datasource-type>orbitarium</required-datasource-type>
  <catalogs>
    <catalog>catalog.misc</catalog>
  </catalogs>
  <flow>
    <step id="askplanet">
      <interaction>
        <parameter id="planet">
          <prompt>orbitarium.prompt.planet</prompt>
          <type base="string"/>
        </parameter>
      </interaction>
    </step>
  </flow>
```

```

<output>
  <http host="www.orbitarium.com" port="80" path="/orbitarium/servlet/ows/">
    <parameter name="method" expression='DataSourceParam("geocentric-position")' />
    <parameter name="planet" expression='planet' />
  </http>
</output>
</report>

```

The attributes host, port and path on the http element are required. The http element can contain a number of parameter elements. The values of the parameters must be given as expressions.

Reports with Custom-output

Finally, to provide in even more flexibility the output can also (starting release 1.2.2) be produced by arbitrary Java classes plugged into the report-definition. This is achieved by simply introducing the <java> tag inside the <output> section of your report:

```

<report id="president_sheet">

  <name>president.sheet.name</name>

  <description>president.sheet.description</description>

  <sortcode>1</sortcode>

  <required-datasource-type>president</required-datasource-type>

  <catalogs>
    <catalog>catalog.presidents/catalog.lists</catalog>
    <catalog>catalog.presidents/catalog.USA/catalog.allreports</catalog>
  </catalogs>

  <flow>
    <step id="askname">
      <interaction>
        <parameter id="presname" required="true">
          <prompt>prompt.presname</prompt>
          <title>prompt.presname</title>
          <type base="string">
            </type>
          <searchlist>
            <sql>
              <dialect types="default">
                <literal>select Pres_Name from president order by Pres_Name</literal>
              </dialect>
            </sql>
            <output-fields value="Pres_Name" />
          </searchlist>
        </parameter>
      </interaction>
    </step>
  </flow>

  <output>
    <java class="sample.PresidentSheetOutput">
      <parameter idref="presname" />
    </java>
  </output>
</report>

```

The provided Java-class needs to implement the OutputGenerator interface to actually produce the output by emitting SAX events.

```

public interface OutputGenerator
{
    public void generateOutputSaxFragment(ContentHandler contentHandler, int chunkOffset,
        int chunkLength,
        ExecutionContext executionContext, ResourceHandle resourceHandle) throws
        Exception;
}

```

After instantiation of your component it will be taken through the Avalon Component Lifecycle which allows it to interact naturally with the Phoenix component container. (e.g. implementing Configurable gives access to any nested XML in the report-file). A sketch-sample is to be found here: (see samples for more detail)

```

package sample;

import java.util.*;
import org.apache.avalon.framework.configuration.*;
import org.outerj.xreporter.report.definition.*;
import org.outerj.xreporter.resource.ResourceHandle;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class PresidentSheetOutput implements OutputGenerator, Configurable
{
    private static final String OUT_PRESIDENT_ELM = "president";
    private static final String OUT_ROOT_ELM = "sheet";
    private static final String XMLNS_PFX = "xmlns";
    private static final String XMLNS_URI = "http://www.w3.org/XML/1998/namespace";
    private static final String OUT_NS_URI = "http://outerx.org/xreporter/demo/presidents/
info-sheet";
    private static final String OUT_NS_PFX = "p";

    private static final String CONFIG_PARAM_ELM = "parameter";
    private static final String CONFIG_PARAM_IDREF_ATT = "idref";
    private String paramId;

    public void configure(Configuration configuration) throws ConfigurationException
    {
        paramId =
configuration.getChild(CONFIG_PARAM_ELM).getAttribute(CONFIG_PARAM_IDREF_ATT);
        if (paramId == null || paramId.trim().equals(""))
            throw new ConfigurationException("Configuration requires nested <" +
CONFIG_PARAM_ELM + " "
                + CONFIG_PARAM_IDREF_ATT + "=\"" + paramId + "\" /> element.");
    }

    public void generateOutputSaxFragment(ContentHandler ch, int chunkOffset, int
chunkLength,
        ExecutionContext context, ResourceHandle resourceHandler) throws Exception
    {
        Object value = context.getExpressionContext().resolveVariable(paramId);
        String presidentName = value.toString();

        ch.startPrefixMapping(OUT_NS_PFX, OUT_NS_URI);
        final AttributesImpl nsAtts = new AttributesImpl();
        nsAtts.addAttribute(XMLNS_URI, OUT_NS_PFX, makeQName(XMLNS_PFX, OUT_NS_PFX),
"PCDATA", OUT_NS_URI);
        ch.startElement(OUT_NS_URI, OUT_ROOT_ELM, makeQName(OUT_NS_PFX, OUT_ROOT_ELM),
nsAtts);

        ch.startElement(OUT_NS_URI, OUT_PRESIDENT_ELM, makeQName(OUT_NS_PFX,
OUT_PRESIDENT_ELM),
            new AttributesImpl());
        ch.characters(presidentName.toCharArray(), 0, presidentName.length());
    }
}

```

```

        ch.endElement(OUT_NS_URI, OUT_PRESIDENT_ELM, makeQName(OUT_NS_PFX,
OUT_PRESIDENT_ELM));

        ch.endElement(OUT_NS_URI, OUT_ROOT_ELM, makeQName(OUT_NS_PFX, OUT_ROOT_ELM));
        ch.endPrefixMapping(OUT_NS_PFX);

    }

    private static String makeQName(final String pfx, final String lName)
    {
        if (pfx == null || pfx.length() == 0)
            return lName;
        // else
        return pfx + ":" + lName;
    }
}

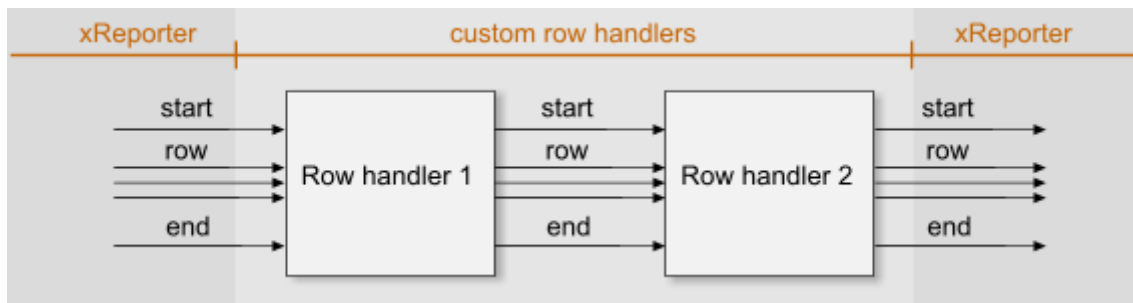
```

Manipulating the result rows using row handlers

When using normal database output, it is possible to plug-in so-called *row handlers* which can manipulate the rows between the moment they are read from the database and the final report output is produced.

Row handlers are connected in a chain, the first row handler has a pointer to the next, and so on. xReporter generates events (= calls methods) on the first row handler in the chain, which is then responsible for forwarding these events to the next row handler, and so on. At the end of the chain comes a component which includes the resulting rows in the report output (= serializes the rows as XML).

This system should be familiar to people who know SAX filters.



Row handlers can modify rows (drop rows, generate additional rows, generate modified rows), but cannot change the column structure.

A row handler should implement the following Java interface:

```
org.outerj.xreporter.report.output.RowHandler
```

A row handler can also implement a number of Avalon framework interfaces (such as Configurable and LogEnabled).

Row handlers are configured in the report definition as follows:

```

<output>
  <database>
    [ ... the sql and the columns ... ]
    <row-handlers>

      <row-handler class="com.mycomp.rowhandler.FooRowHandler">
        [... optional config data for the row handler ..]

```

```

</row-handler>

<row-handler class="com.mycomp.rowhandler.BarRowHandler">
  [... optional config data for the row handler ..]
</row-handler>

</row-handlers>
</database>
</output>

```

For examples, see:

- The 'residency' report in the testconf
- The 'PeriodMerger' class in the source tree.

Various

Disallowing sorting on specific columns

On the <column> element you can add an attribute allow-orderby with as values:

- true
- false
- if-selected: only allow sorting on this column if it is selected as a visible column in the report output

Working with the selected columns

It can sometimes be useful to reference the current column selection (= the visible columns) in the report output SQL. This can be done with the selected-column-list element. These are the possible syntaxes:

```

<selected-column-list colrefs="*" />

<selected-column-list colrefs="col1 col2 col3" />

<selected-column-list colrefs="col1 col2 col3" fallback="col1" />

```

The first variant will insert (comma-separated) the list of all selected columns.

The second variant will insert the list of all the specified columns, but only if they are currently selected.

The last variant uses the fallback attribute to specify what should be inserted if none of the specified columns is currently selected.

To see a useful example of <selected-column-list>, see the presidentcount sample report.

Storing parameter or condition values on a per-user basis

Although somewhat hidden here, this is a powerful feature. It allows you to store the value of a parameter, or the operator and values of a condition, for a certain user. If the user later reruns the same report, the value will be automatically prefilled. This is achieved simply by adding an attribute named "store" to the parameter or condition element. The value of this attribute is an id you choose yourself. Example:

```

<condition id="birthy" required="true" store="birthy">
  <prompt>prompt.birthy</prompt>
  <type base="long" />
</condition>

```

Here we gave the store attribute the same value as the id attribute, but that does not have to be case.

You can also give another condition on another report the same value for the store attribute. When the user than runs that report, the value will be automatically pre-filled over there too. Thus this feature works across reports. The only requirement is that the type (string, long, date, ...) of the parameters or conditions is the same.

"calculate" SQL part

If a calculated value needs to be inserted in a SQL element, this can be done using the "calculate" tag. This can be useful if a value needs to be derived from a parameter entered by the user. The report fragment below shows how the value of a parameter called "userparam" can be multiplied with 5 before inserting it in the SQL statement.

```
<sql>
  <dialect types="default">
    <literal>select * from mydb where factor &lt;</literal>
    <calculate expression="userparam * 5" type="long"/>
  </dialect>
</sql>
```

The whitespace pitfall

When building up the actual SQL statements out of the various <literal >, <temptable>, <calculate>, <parameter> parts there are regular spaces added around the various parts. One needs to be aware of this in order to build up the correct resulting SQL syntax. Two distinct examples show what we mean here:

1. Placing a <parameter> value inside a SQL LIKE '%..%' part will add unwanted spaces and additional quotes around the parameter-value when using a mix of <literal> and <parameter>. To force the correct concatenation however you can use the <calculate>:

```
<sql>
  <literal>
    SELECT groupname
    FROM groups
    WHERE groupname LIKE
  </literal>
  <calculate expression='Concat("%", groupname, "%")' />
</sql>
```

2. Adding a literal '.fieldname' to a referenced <temptable> will put an awkward space between the generated name of the temporary table and the .fieldname that needs to follow. To accomodate this, the <temptable> tag has a dedicated attribute @field:

```
<sql>
  <literal>
    UPDATE</literal><temptable idref="pp_01"/><literal>
    SET ccn = ((
      SELECT pp_02.ccn
      FROM
    </literal>
    <temptable idref="pp_02" fieldname="pp_02" />
    <literal>
      WHERE pp_02.case_id =
    </literal>
    <temptable idref="pp_01" fieldname="case_id" />
  </sql>
```

3. In a similar way there are times one wants to create indices on the temptables that require a global unique name that is however made unique accross report instances in a same way as is done with the temptable-names. To allow for this kind of arbitrary suffixing of the table-name a dedicated attribute @sub-name is available on the <temptable> element. @field and @sub-name are both optional, but only one of them can be used (and is meaningful) at any given time.

```
<sql>
  <dialect types="default">
    <literal>create table</literal>
    <temptable id="married_pres"/>
    <literal>select distinct(Pres_Name) from pres_marriage</literal>
  </dialect>
</sql>
<sql>
  <dialect types="default" >
    <literal>create unique index </literal>
    <temptable idref="married_pres" sub-name="_Pres_Name_IDX" />
    <literal>on</literal>
    <temptable idref="married_pres" />
    <literal>(Pres_Name)</literal>
  </dialect>
</sql>
```

WARNING

Make sure the creation of the index is happening in the same step as the creation of the table. In that case revisiting the step will drop the table (and the dependent index) automatically. If you don't however the end-user can revisit the intermediate step that only creates the index which will yield an error when re-creating an index that already exists.

Ordering of reports in the report catalog

By default, the reports in the report catalog are shown in whatever order the operating system happened to give the list of report files to xReporter. On Windows, they are usually sorted by filename, but don't count on this.

To alter the ordering, you can specify a sortcode in the report definition:

```
<report id="sample1">
  <name>sample1.name</name>
  <description>sample1.description</description>
  <sortcode>abc</sortcode>
  ...
```

The sortcode tag can contain whatever string you like. The ordering is not locale-dependent.

About report reloading

As mentioned before, xReporter will periodically scan the report directory to see if there were any changes. In case a report was modified, its report definition will be reloaded. If this reloading fails (due to errors in the report), the report will no longer be available in the report catalog until you correct these errors.

Existing report instances living in memory will keep working against the report definitions they were created against. So if you change a report definition, you will need to start a new report instance to notice the changes, existing report instances will remain unchanged.

Saved report instances

xReporter end-users can store their parameterized reports for later re-execution. The stored state includes parameters and conditions, the sort order, the QBE settings, the column configuration, ...

When the user later re-executes the report, a new report instance is created and its state is adjusted in pretty much the same way as it was originally applied: the interaction steps will be one by one read their state and (if applicable) run any <execute> parts, possibly creating temporary tables etc.

If the report definition changed since the last execution, it might be possible that the report cannot be restored. For example, if you added a new required parameter to an interaction step, restoring the report state will fail at that step. Adding non-required parameters will not cause failure, nor will removing parameters.

Some technical details about report execution

Each time the user changes some output options (sort order, column layout, previous/next chunk, ...) the database query will be re-executed on the database. xReporter will never cache data.

xReporter will always run through the entire result set, even if only a certain chunk of the result is required. On the one hand, this is required because it is not possible to define the chunk in (standard) SQL. On the other hand, this is needed to determine the size of the result set (which is required to show to the user). Concerning chunking, xReporter assumes that the database will, for the same query, always return the records in the same order.

For most types of reports, except maybe those that would return tens of thousands of rows, the above remarks should cause no big problems. If however your query is complex, it may be a problem that it is re-executed each time the user switches a chunk or changes the column layout. This can be avoided though by first creating a temporary table with the results, and then doing the final query from that temporary table (thus the temporary table then serves as a cache).