Oracle Business Intelligence Publisher User's Guide
Release 10.1.3.2
Part Number B40017-01

Building a Data Template

Introduction

The BI Publisher data engine enables you to rapidly generate any kind of XML data structure against any database in a scalable, efficient manner. The data template is the method by which you communicate your request for data to the data engine. It is an XML document whose elements collectively define how the data engine will process the template to generate the XML.

The data engine supports the following functionality:

- Single and multiple data queries
- Query links
- Parameters
- Aggregate functions (SUM, AVG, MIN, MAX, COUNT)
- Event triggers
- Multiple data groups

The XML output generated by the data engine supports the following:

- Unicode for XML Output

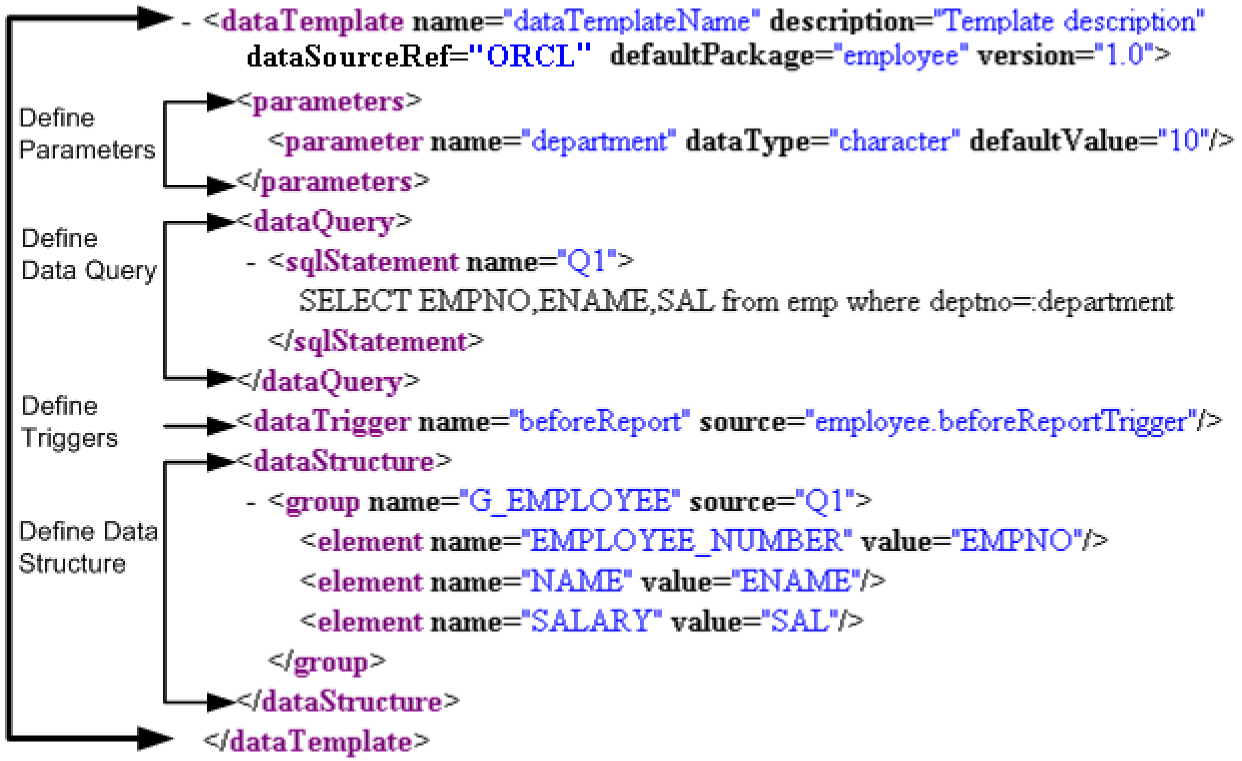
Unicode is a global character set that allows multilingual text to be displayed in a single application. This enables you to develop a single multilingual application and deploy it worldwide.

- Canonical format

  The data engine generates date elements using the canonical ISO date format: YYYY-MM-DDTHH24:MI:SS.FF3TZH:TZM for a mapped date element, and ######.## for number elements in the data template XML output.

The Data Template Definition

The data template is an XML document that consists of four basic sections: define parameters, define triggers, define data query, define data structure. This structure is shown in the following graphic:



```
- <dataTemplate name="dataTemplateName" description="Template description"
    dataSourceRef="ORCL"  defaultPackage="employee" version="1.0">
  <parameters>
    <parameter name="department" dataType="character" defaultValue="10"/>
  </parameters>
  <dataQuery>
  - <sqlStatement name="Q1">
      SELECT EMPNO,ENAME,SAL from emp where deptno=:department
    </sqlStatement>
  </dataQuery>
  <dataTrigger name="beforeReport" source="employee.beforeReportTrigger"/>
  <dataStructure>
  - <group name="G_EMPLOYEE" source="Q1">
      <element name="EMPLOYEE_NUMBER" value="EMPNO"/>
      <element name="NAME" value="ENAME"/>
      <element name="SALARY" value="SAL"/>
    </group>
  </dataStructure>
</dataTemplate>
```

As shown in the sample figure, the data template consists of a <parameters> section in which parameters are declared in child <parameter> elements; a <dataQuery> section in which the SQL queries are defined in child <sqlStatement> elements; and a <dataStructure> section in which the output XML structure is defined.

The table below lists the elements that make up the XML data template. Each element is described in detail in the following sections. Required elements are noted.

Element	Attributes/Description
dataTemplate (Required)	Attributes: • name (Required) • description • version (Required) • defaultPackage - the PL/SQL package name to resolve any lexical references, group filters, or data triggers defined in the template. • dataSourceRef - (Required) the default data source reference for the entire data template.
properties	Consists of one or more <property> elements to support the XML output and Data Engine specific properties.
property	Attributes: • name (Required) - the property name. • value - valid values for this property.
parameters	Consists of one or more <parameter> elements.
parameter	Attributes: • name (Required) - the parameter name that will be referenced in the template. • dataType - valid values are: "character", "date", "number" • defaultValue - value to use for the parameter if none supplied from the data • include_in_output - whether this parameter should appear in the XML output or not. The valid values are “true” and “false”.
lexicals	(Supported for queries against the Oracle E-Business Suite only). Consists of one or more lexical elements to support flexfields.
lexical	There are four types of key flexfield-related lexicals as follows: • oracle.apps.fnd.flex.kff.segments_metadata • oracle.apps.fnd.flex.kff.select • oracle.apps.fnd.flex.kff.where • oracle.apps.fnd.flex.kff.order_by
dataQuery (Required)	Consists of one or more <sqlstatement> or <xml> elements.

sqlstatement (Required)	Attributes: • name (Required) - the unique query identifier. Note that this name identifier will be the same across the data template. Enter the query inside the CDATA section.
xml	Attributes: • name (Required) - the unique query identifier. • expressionPath – Xpath expression
url	Attributes: • method – either GET or POST • realm - authentication name • username- valid username • password - valid password
link	Attributes: • parentQuery - specify the parent query name. • parentColumn - specify the parent column name. • childQuery - specify the child query name. • childColumn - specify the child column name. • condition - the SQL operator that defines the relationship between the parent column and the child column. The following values for condition are supported: =, <, <=, >, >=
dataTrigger	Attributes: • name (Required) - the event name to fire this trigger • source (Required) - the PL/SQL <package name>.<function name>
dataStructure	(Required for multiple queries) Defines the structure of the output XML. Consists of <group> and <element> elements to specify the structure. This section is optional for single queries; if not specified, the data engine will generate flat XML.
group	Consists of one or more <element> elements and sub <group> elements. Attributes: • name (Required) - the XML tag name to be assigned to the group.

	• source (Required) - the unique query identifier for the corresponding sqlstatement from which the group's elements will be derived. • groupFilter - the filter to apply to the output data group set. Define the filter as: <package name>.<function name>. **Note:** Applying a filter has performance impact. Do not use this functionality unless necessary. When possible, filter data using a WHERE clause in your query.
element (Required)	Attributes: • name - the tag name to assign to the element in the XML data output. • value (Required) - the column name for the SQL statement. Note that for aggregations in which the column name is in another group, the value must be defined as <group name>.<column/alias name>. • function - supported functions are: SUM(), COUNT(), AVG(), MIN(), MAX()

Constructing the Data Template

You can use any text or XML editor to write a data template.

Data Template Declaration

The <dataTemplate> element is the root element. It has a set of related attributes expressed within the <dataTemplate> tag.

Attribute Name	Description
name	(Required) Enter the data template name.
description	(Optional) Enter a description of this data template.
version	(Required) Enter a version number for this data template.
defaultPackage	This attribute is required if your data template contains lexical references or any other calls to PL/SQL.
dataSourceRef	(Required) The default data source reference for the entire data template.

Properties Section

Use the <properties> section to set properties to affect the XML output and data engine execution.

Example:

```
<properties>
   <property name="include_parameters" value="false" />
   <property name="include_null_Element" value="false" />
   <property name="include_rowsettag" value="false" />
   <property name="scalable_mode" value="on" />
</properties>
```

The following table shows the supported properties:

Property Name	**Description**
include_parameters	Indicates whether to include parameters in the output. Valid values are: • True (default) • False
include_null_Element	Indicates whether to remove or keep the null elements in the output. Valid values are: • True (default) • False
xml_tag_case	Allows you to set the case for the output XML element names. Valid values are: • upper (default) • lower • as_are (The case will follow the definition in the dataStructure section.)
db_fetch_size	Sets the number of rows fetched at a time through the jdbc connection. The default value is 500.
scalable_mode	Sets the data engine to execute in scalable mode. This is required when processing a large volume of data. Valid values: • on • off (default)
include_rowsettag	Allows you to include or exclude the Rowset Tag from the output. Valid values: • true (default) • false

debug_mode	Turns debug mode on or off. Valid values: • on • off (default)

Parameters Section

A parameter is a variable whose value can be set at runtime. Parameters are especially useful for modifying SELECT statements and setting PL/SQL variables at runtime. The Parameters section of the data template is optional.

How to Define Parameters

The <parameter> element is placed between the open and close <parameters> tags. The <parameter> element has a set of related attributes. These are expressed within the <parameter> tag. For example, the name, dataType, and defaultValue attributes are expressed as follows:

```
<parameters>
   <parameter name="dept" dataType="number" defaultValue="10"/>
</parameters>
```

Attribute Name	Description
name	**Required**. A keyword, unique within a given Data Template, that identifies the parameter.
dataType	Optional. Specify the parameter data type as "character", "date", or "number". Default value is "character". For the "date" dataType, the following three formats (based on the canonical ISO date format) are supported: • YYYY-MM-DD (example: 1997-10-24) • YYYY-MM-DD HH24:MI:SS (example: 1997-10-24 12:00:00) • YYYY-MM-DDTHH24:MI:SS.FF3TZH:TZM
defaultValue	Optional. This value will be used for the parameter if no other value is supplied from the data at runtime.
include_in_output	Optional. Whether this parameter should appear in XML output or not. The valid values are “true” and “false”.

How to Pass Parameters

To pass parameters, (for example, to restrict the query), use bind variables in your query. For example:

Query:

```
SELECT * FROM EMP
WHERE deptno=:department
```

At runtime, the value of department is passed to the query:

```
SELECT * FROM EMP
WHERE deptno=10
```

Data Query Section

The <dataQuery> section of the data template is required.

Supported Column Types

The following column types are selectable:

- VARCHAR2, CHAR
- NUMBER
- DATE, TIMESTAMP
- BLOB/BFILE (conditionally supported)

  BLOB image retrieval is supported in the following two cases:

  - Using the SetSQL API (see SQL to XML Processor)
  - In the data template when no Structure section is defined. The returned data must be flat XML.

  The BLOB/BFILE must be an image. Images are retrieved into your results XML as base64 encoding. You can retrieve any image type that is supported in the RTF template (jpg, gif, or png). You must use specific syntax to render the retrieved image in your template. See Rendering an Image Retrieved from BLOB Data.

- CLOB (conditionally supported)

  The CLOB must contain text or XML. Data cannot be escaped inside the CLOB column.

- XMLType (conditionally supported)

  XMLType can be supported if it is converted to a CLOB using the getClobVal() method.

- REF CURSOR (conditionally supported)

  A REF CURSOR is supported inside the SQL statement when only one results set is returned.

How to Define SQL Queries

The <sqlStatement> element is placed between the open and close dataQuery tags. The <sqlStatement> element has a related attribute, name. It is expressed within the <sqlStatment> tag. The query is entered in the CDATA section. For example:

```
<dataQuery>
  <sqlStatement name="Q1">
  <![CDATA[SELECT DEPTNO,DNAME,LOC from dept]]>
  </sqlStatement>
</dataQuery>
```

Attribute Name	Description
name	A unique identifying name for the query. Note that this name will be referred to throughout the data template.

If your column names are not unique, you must use aliases in your SELECT statements to ensure the uniqueness of your column names. If you do not use an alias, then the default column name is used. This becomes important when you specify the XML output in the dataStructure section. To specify an output XML element from your query you declare a value attribute for the element tag that corresponds to the source column.

Tip: Performing operations in SQL is faster than performing them in the data template or PL/SQL. It is recommended that you use SQL for the following operations:

- Use a WHERE clause instead of a group filter to exclude records.
- Perform calculations directly in your query rather than in the template.

How to Define an XML Data Source

Place the <xml> element between the open and close dataQuery tags. The <xml> element has the related attributes: name, which is a unique identifier; and expressionPath, which can be used to link the SQL query and the XML data. Linking the SQL query and XML data enables you to leverage capabilities such as aggregation and summarization.

Example:

```
<xml name="empxml" expressionPath=".//ROW[DEPTNO=$DEPTNO]">
<url method="GET" realm="" username="" password="">file:///d:/dttest/employee.xml</url>
</xml>
```

Lexical References

You can use lexical references to replace the clauses appearing after SELECT, FROM, WHERE, GROUP BY, ORDER BY, or HAVING. Use a lexical reference when you want the parameter to replace multiple values at runtime.

Create a lexical reference using the following syntax:

```
&parametername
```

Define the lexical parameters as follows:

- Before creating your query, define a parameter in the PL/SQL default package for each lexical reference in the query. The data engine uses these values to replace the lexical parameters.
- Create your query containing lexical references.

For example:

```
Package employee
AS
 where_clause varchar2(1000);
  .....

Package body employee
 AS
 .....
where_clause := 'where deptno=10';
.....
```

Data template definition:

```
<dataQuery>
 <sqlstatement name="Q1">
 <![CDATA[SELECT ENAME, SAL FROM EMP &where_clause]]>
</sqlstatement>
</dataQuery>
```

How to Define a Data Link Between Queries

If you have multiple queries, you must link them to create the appropriate data output. In the data template, there are two methods for linking queries: using bind variables or using the <link> element to define the link between queries.

Tip: To maximize performance when building data queries in the data template:

BI Publisher tests have shown that using bind variables is more efficient than using the link tag.

The following example shows a query link using a bind variable:

```
<dataQuery>
 <sqlstatement name="Q1">
 <![CDATA[SELECT EMPNO, ENAME, JOB from EMP
   WHERE DEPTNO = :DEPTNO]]>
 </sqlstatement>
</dataQuery>
```

The <link> element has a set of attributes. Use these attributes to specify the required link information. You can specify any number of links. For example:

```
<link name="DEPTEMP_LINK" parentQuery="Q1" parentColumn="DEPTNO" childQuery="Q_2" childColumn="DEPARTMENTNO"/>
```

Attribute Name	Description
name	Required. Enter a unique name for the link.
parentQuery	Specify the parent query name. This must be the name that you assigned to the corresponding <sqlstatement> element. See How to Define Queries.
parentColumn	Specify the parent column name.
childQuery	Specify the child query name. This must be the name that you assigned to the corresponding <sqlstatement> element. See How to Define Queries.
childColumn	Specify the child column name.

Using Data Triggers

Data triggers execute PL/SQL functions at specific times during the execution and generation of XML output. Using the conditional processing capabilities of PL/SQL for these triggers, you can do things such as perform initialization tasks and access the database.

Data triggers are optional, and you can have as many <dataTrigger> elements as necessary.

The <dataTrigger> element has a set of related attributes. These are expressed within the <dataTrigger> tag. For example, the name and source attributes are expressed as follows:

```
<dataTrigger name="beforeReport" source="employee.beforeReport()"/>
<dataTrigger name="beforeReport" source="employee.beforeReport(:Parameter)"/>
```

Attribute Name	Description
name	The event name to fire this trigger.
source	The PL/SQL <package name>.<function name> where the executable code resides.

The location of the trigger indicate at what point the trigger fires:

- Place a beforeReport trigger anywhere in your data template before the <dataStructure> section.. A beforeRepot trigger fires before the dataQuery is executed.
- Place an afterReport trigger after the <dataStructure> section. An afterReport trigger fires after you exit and after XML output has been generated.

Data Structure Section

In the data structure section you define what the XML output will be and how it will be structured. The complete group hierarchy is available for output. You can specify all the columns within each group and break the order of those columns; you can use summaries, and placeholders to further customize within the groups. The dataStructure section is required for multiple queries and optional for single queries. If omitted for a single query, the data engine will generate flat XML.

Defining a Group Hierarchy

In the data template, the <group> element is placed between open and close <dataStructure> tags. Each <group> has a set of related elements. You can define a group hierarchy and name the element tags for the XML output.

Creating Break Groups

Use a break group to produce subtotals or add placeholder columns. A break group suppresses duplicate values in sequential records. You should set an Order By clause in the SQL query to suppress duplicate values.

Assign a name to the group, and declare the source query, then specify the elements you want included in that group. When you specify the element, you assign it a name that will be used as the XML output tag name, and you declare the source column as the value. If you do not assign a name, the value (or source column name) will be used as the tag name.

For example:

```
<dataStructure>
    <group name="G_DEPT" source="Q1" ">
        <element name="DEPT_NUMBER" value="DEPTNO"  />
        <element name="DEPT_NAME"   value="DNAME"/>
         <group name="G_EMP" source="Q2">
                <element name="EMPLOYEE_NUMBER" value="EMPNO" />
                <element name="NAME" value="ENAME"/>
                <element name="JOB" value="JOB" />
         </group>
     </group>
   </dataStructure>
```

The following table lists the attributes for the <group> element tag:

Attribute Name	Description

name	Specify any unique name for the group. This name will be used as the output XML tag name for the group.
source	The name of the query that provides the source data for the group. The source must come from the name attribute of the <sqlStatement> element.

The following table lists the attributes for the <element> element tag:

Attribute Name	Description
name	Specify any name for the element. This name will be used as the output XML tag name for the element. The name is optional. If you do not specify a name, the source column name will be used as the XML tag name.
value	The name of the column that provides the source data for the element (from your query).

Applying Group Filters

It is strongly recommended that you use a WHERE clause instead of a group filter to exclude records from your extract. Filters enable you to conditionally remove records selected by your queries, however, this approach impacts performance. Groups can have user-created filters, using PL/SQL.

The PL/SQL function must return a boolean value (TRUE or FALSE). Depending on whether the function returns TRUE or FALSE, the current record is included or excluded from the XML data output.

For example, a sample PL/SQL function might be:

```
function G_EMPFilter return boolean is
begin
  if sal < 1000 then
    return (FALSE);
  else
    return (TRUE);
end if;
end;
```

An example of the group filter in your data template definition would be:

```
<group name="G_DEPT" source="Q1" groupFilter="empdata.G_EMPFilter(:DEPTSAL)">
   <element name="DEPT_NUMBER" value="DEPTNO"  />
   <element name="DEPT_NAME"   value="DNAME"/>
   <element name="DEPTSAL"     value="G_EMP.SALARY" function="SUM()"/>
```

Creating a Summary Column

A summary column performs a computation on another column's data. Using the function attribute of the <element> tag, you can create the following summaries: sum, average, count, minimum, and maximum.

To create a summary column, you must define the following three attributes in the element tag:

Attribute	Description
name	The XML tag name to be used in the XML data output.
source	The name of the column that contains the data on which the summary calculation is to be performed. The source column remains unchanged.
function	The aggregation function to be performed. The type tells the XDO data engine how to compute the summary column values. Valid values are: SUM(), AVG(), COUNT(), MAX(), and MIN().

The break group determines when to reset the value of the summary column. For example:

```
<group name="G_DEPT" source="Q1">
   <element name="DEPT_NUMBER" value="DEPTNO"  />
   <element name="DEPTSAL" value="G_EMP.SALARY" function="SUM()"/>
       <group name="G_EMP" source="Q2">
          <element name="EMPLOYEE_NUMBER" value="EMPNO" />
          <element name="NAME" value="ENAME"/>
          <element name="JOB" value="JOB" />
          <element name="SALARY" value="SAL"/>
        </group>
</group>
```

Flexfield Support

Note: This section applies to data templates written to query the Oracle Applications database.

Flexfields are defined in the data template using lexical parameters.

How to define a flexfield

1. Define the SELECT statement to use for the report data.
2. Within the SELECT statement, define each flexfield as a lexical. Use the &LEXICAL_TAG to embed flexfield related lexicals into the SELECT statement.
3. Define the flexfield-related lexicals using XML tags in the data template.

```
<dataTemplate ...
     <parameters ...
     </parameters>
```

```
      <lexicals ...
         <lexical type="oracle.apps.fnd.flex.kff..."
                  name="<Name of the lexical>"
                  comment="<comment>"
                   />
         <lexical type="oracle.apps.fnd.flex.kff..."
                   name="<Name of the lexical>"
                   comment="<comment>"
                   />
        </lexicals>

        <dataQuery>
          <sqlStatement ...

            SELECT &FLEX_SELECT flex_select_alias
            FROM some_table st, code_combination_table cct
            WHERE st.some_column = 'some_condition'
                  AND &FLEX_WHERE
            ORDER BY st.some_column, &FLEX_ORDER_BY
           </sqlStatement>
         </dataQuery>
         <dataStructure .../>

</dataTemplate>
```

Flexfield Lexicals

There are four types of KFF-related lexicals. These are:

- oracle.apps.fnd.flex.kff.segments_metadata
- oracle.apps.fnd.flex.select
- oracle.apps.fnd.flex.kff.where
- oracle.apps.fnd.flex.kff.order_by

Following are descriptions of each type of KFF lexical:

oracle.apps.fnd.flex.kff.segments_metadata

Use this type of lexical to retrieve flexfield-related metadata. Using this lexical, you are not required to write PL/SQL code to retrieve this metadata. Instead, define a dummy SELECT statement, then use this lexical to get the metadata.

The XML syntax for this lexical is as follows:

```
<lexicals>
  <lexical
    type="oracle.apps.fnd.flex.kff.segments_metadata"
    name="Name of the lexical"
    comment="Comment"
    application_short_name="Application Short Name of the KFF"
    id_flex_code="Internal code of the KFF"
    id_flex_num="Internal number of the KFF structure"
    segments="For which segment(s) is this metadata requested?"
    show_parent_segments="Should the parent segments be listed?"
    metadata_type="Type of metadata requested"/>
</lexicals>
```

The following table lists the attributes for the segements_metadata lexical:

Attribute	Description
application_short_name	(Required) The application short name of the key flexfield. For example: SQLGL.
id_flex_code	(Required) the internal code of the key flexfield. For example: GL#
id_flex_num	(Required) Internal number of the key flexfield structure. For example: 101
segments	(Optional) Identifies for which segments this data is requested. Default value is "ALL". See the *Oracle Applications Developer's Guide* for syntax.
show_parent_segments	(Optional) Valid values are "Y" and "N". Default value is "Y". If a dependent segment is displayed, the parent segment is automatically displayed, even if it is not specified as displayed in the segments attribute.
metadata_type	(Required) Identifies what type of metadata is requested. Valid values are: above_prompt - above prompt of segment(s). left_prompt - left prompt of segment(s)

This example shows how to request the above_prompt of the GL Balancing Segment, and the left_prompt of the GL Account Segment.

```
SELECT &FLEX_GL_BALANCING_APROMPT alias_gl_balancing_aprompt, &FLEX_GL_ACCOUNT_LPROMPT alias_gl_account_lprompt
FROM dual

<lexicals>
  <lexical type="oracle.apps.fnd.flex.kff.segments_metadata"
   name="FLEX_GL_BALANCING_APROMPT"
```

```
    comment="Comment"
    application_short_name="SQLGL"
    id_flex_code="GL#"
    id_flex_num=":P_ID_FLEX_NM"
    segments="GL_BALANCING"
    metadata_type="ABOVE_PROMPT"/>
   <lexical type="oracle.apps.fnd.flex.kff.segments_metadata"
    name="FLEX_GL_ACCOUNT+LPROMPT"
    comment="Comment"
    application_short_name="SQLGL"
    id_flex_code="GL#"
    id_flex_num=":P_ID_FLEX_NUM"
    segments="GL_ACCOUNT"
    metadata_type="LEFT_PROMPT"/>
</lexicals>
```

oracle.apps.fnd.flex.kff.select

This type of lexical is used in the SELECT section of the statement. It is used to retrieve and process key flexfield (kff) code combination related data based on the lexical definition.

The syntax is as follows:

```
<lexicals>
   <lexical
      type="oracle.apps.fnd.flex.kff.select"
      name="Name of the lexical"
      comment="Comment"
      application_short_name="Application Short Name of the KFF"
      id_flex_code="Internal code of the KFF"
      id_flex_num="Internal number of the KFF structure"
      multiple_id_flex_num="Are multiple structures allowed?"
      code_combination_table_alias="Code Combination Table Alias"
      segments="Segments for which this data is requested"
      show_parent_segments="Should the parent segments be listed?"
      output_type="output type"/>
</lexicals>
```

The following table lists the attributes for this lexical:

Attribute	Description

application_short_name	(Required) The application short name of the key flexfield. For example: SQLGL.
id_flex_code	(Required) the internal code of the key flexfield. For example: GL#
id_flex_num	(Conditionally required) Internal number of the key flexfield structure. For example: 101. Required if MULTIPLE_ID_FLEX_NUM is "N".
multiple_id_flex_num	(Optional) Indicates whether this lexical supports multiple structures or not. Valid values are "Y" and "N". Default is "N". If set to "Y", then flex will assume all structures are potentially used for data reporting and it will use <code_combination_table_alias>.<set_defining_column_name> to retrieve the structure number.
code_combination_table_alias	(Optional) Segment column names will be prepended with this alias.
segments	(Optional) Identifies for which segments this data is requested. Default value is "ALL". See the *Oracle Applications Developer's Guide* for syntax.
show_parent_segments	(Optional) Valid values are "Y" and "N". Default value is "Y". If a dependent segment is displayed, the parent segment is automatically displayed, even if it is not specified as displayed in the segments attribute.
output_type	(Required) Indicates what kind of output should be used as the reported value. Valid values are: value - segment value as it is displayed to user. padded_value - padded segment value as it is displayed to user. Number type values are padded from the left. String type values are padded on the right.
description	Segment value's description up to the description size defined in the segment definition.
full_description	Segment value's description (full size).
security	Returns Y if the current combination is secured against the current user, N otherwise.

This example shows how to report concatenated values, concatenated descriptions, the value of the GL Balancing Segment, and the full description of the GL Balancing Segment for a single structure:

```
SELECT &FLEX_VALUE_ALL alias_value_all,
      &FLEX_DESCR_ALL alias_descr_all,
      &FLEX_GL_BALANCING alias_gl_balancing,
      &FLEX_GL_BALANCING_FULL_DESCR alias_gl_balancing_full_descr,
      ...
       FROM gl_code_combinations gcc,
            some_other_gl_table sogt
      WHERE gcc.chart_of_accounts_id = :p_id_flex_num
        and sogt.code_combination_id = gcc.code_combination_id
       and <more conditions on sogt>


 <lexicals>
    <lexical
       type="oracle.apps.fnd.flex.kff.select"
       name="FLEX_VALUE_ALL"
       comment="Comment"
```

```
        application_short_name="SQLGL"
        id_flex_code="GL#"
        id_flex_num=":P_ID_FLEX_NUM"
        multiple_id_flex_num="N"
        code_combination_table_alias="gcc"
        segments="ALL"
        show_parent_segments="Y"
        output_type="VALUE"/>
     <lexical
        type="oracle.apps.fnd.flex.kff.select"
        name="FLEX_DESCR_ALL"
        comment="Comment"
        application_short_name="SQLGL"
        id_flex_code="GL#"
        id_flex_num=":P_ID_FLEX_NUM"
        multiple_id_flex_num="N"
        code_combination_table_alias="gcc"
        segments="ALL"
        show_parent_segments="Y"
        output_type="DESCRIPTION"/>
     <lexical
        type="oracle.apps.fnd.flex.kff.select"
        name="FLEX_GL_BALANCING"
        comment="Comment"
        application_short_name="SQLGL"
        id_flex_code="GL#"
        id_flex_num=":P_ID_FLEX_NUM"
        multiple_id_flex_num="N"
        code_combination_table_alias="gcc"
        segments="GL_BALANCING"
        show_parent_segments="N"
        output_type="VALUE"/>
     <lexical
        type="oracle.apps.fnd.flex.kff.select"
        name="FLEX_GL_BALANCING_FULL_DESCR"
        comment="Comment"
        application_short_name="SQLGL"
        id_flex_code="GL#"
        id_flex_num=":P_ID_FLEX_NUM"
        multiple_id_flex_num="N"
        code_combination_table_alias="gcc"
        segments="GL_BALANCING"
```

```
      show_parent_segments="N"
      output_type="FULL_DESCRIPTION"/>
 </lexicals>
```

oracle.apps.fnd.flex.kff.where

This type of lexical is used in the WHERE section of the statement. It is used to modify the WHERE clause such that the SELECT statement can filter based on key flexfield segment data.

The syntax for this lexical is as follows:

```
<lexicals>
   <lexical
      type="oracle.apps.fnd.flex.kff.where"
      name="Name of the lexical"
      comment="Comment"
      application_short_name="Application Short Name of the KFF"
      id_flex_code="Internal code of the KFF"
      id_flex_num="Internal number of the KFF structure"
      code_combination_table_alias="Code Combination Table Alias"
      segments="Segments for which this data is requested"
      operator="The boolean operator to be used in the condition"
      operand1="Values to be used on the right side of the operator"
      operand2="High value for the BETWEEN operator"/>
</lexicals>
```

The attributes for this lexical are listed in the following table:

Attribute	**Description**
application_short_name	(Required) The application short name of the key flexfield. For example: SQLGL.
id_flex_code	(Required) the internal code of the key flexfield. For example: GL#
id_flex_num	(Conditionally required) Internal number of the key flexfield structure. For example: 101. Required if MULTIPLE_ID_FLEX_NUM is "N".
code_combination_table_alias	(Optional) Segment column names will be prepended with this alias.
segments	(Optional) Identifies for which segments this data is requested. Default value is "ALL". See the *Oracle Applications Developer's Guide* for syntax.
operator	(Required) Valid values are: =, <, >, <=, >=, !=, <>, ||, BETWEEN, LIKE
operand1	(Required) Values to be used on the right side of the conditional operator.
operand2	(Optional) High value for the BETWEEN operator.

full_description	Segment value's description (full size).
security	Returns Y if the current combination is secured against the current user, N otherwise.

This example shows a filter based on the GL Account segment and the GL Balancing Segment:

```
SELECT <some columns>
     FROM gl_code_combinations gcc,
          some_other_gl_table sogt
    WHERE gcc.chart_of_accounts_id = :p_id_flex_num
      and sogt.code_combination_id = gcc.code_combination_id
      and &FLEX_WHERE_GL_ACCOUNT
      and &FLEX_WHERE_GL_BALANCING
      and <more conditions on sogt>


<lexicals>
   <lexical
      type="oracle.apps.fnd.flex.kff.where"
      name="FLEX_WHERE_GL_ACCOUNT"
      comment="Comment"
      application_short_name="SQLGL"
      id_flex_code="GL#"
      id_flex_num=":P_ID_FLEX_NUM"
      code_combination_table_alias="gcc"
      segments="GL_ACCOUNT"
      operator="="
      operand1=":P_GL_ACCOUNT"/>
   <lexical
      type="oracle.apps.fnd.flex.kff.where"
      name="FLEX_WHERE_GL_BALANCING"
      comment="Comment"
      application_short_name="SQLGL"
      id_flex_code="GL#"
      id_flex_num=":P_ID_FLEX_NUM"
      code_combination_table_alias="gcc"
      segments="GL_BALANCING"
      operator="BETWEEN"
      operand1=":P_GL_BALANCING_LOW"
      operand2=":P_GL_BALANCING_HIGH"/>
</lexicals>
```

oracle.apps.fnd.flex.kff.order_by

This type of lexical is used in the ORDER BY section of the statement. It returns a list of column expressions so that the resulting output can be sorted by the flex segment values.

The syntax for this lexical is as follows:

```
<lexicals>
  <lexical
   type="oracle.apps.fnd.flex.kff.order_by"
   name="Name of the lexical"
   comment="Comment"
   application_short_name="Application Short Name of the KFF"
   id_flex_code="Internal code of the KFF"
   id_flex_num="Internal number of the KFF structure"
   multiple_id_flex_num="Are multiple structures allowed?"
   code_combination_table_alias="Code Combination Table Alias"
   segments="Segment(s)for which data is requested"
   show_parent_segments="List parent segments?"/>
</lexicals>
```

The attributes for this lexical are listed in the following table:

Attribute	Description
application_short_name	(Required) The application short name of the key flexfield. For example: SQLGL.
id_flex_code	(Required) the internal code of the key flexfield. For example: GL#
id_flex_num	(Conditionally required) Internal number of the key flexfield structure. For example: 101. Required if MULTIPLE_ID_FLEX_NUM is "N".
multiple_id_flex_num	(Optional) Indicates whether this lexical supports multiple structures or not. Valid values are "Y" and "N". Default is "N". If set to "Y", then flex will assume all structures are potentially used for data reporting and it will use <code_combination_table_alias>.<set_defining_column_name> to retrieve the structure number.
code_combination_table_alias	(Optional) Segment column names will be prepended with this alias.
segments	(Optional) Identifies for which segments this data is requested. Default value is "ALL". See the *Oracle Applications Developer's Guide* for syntax.
show_parent_segments	(Optional) Valid values are "Y" and "N". Default value is "Y". If a dependent segment is displayed, the parent segment is automatically displayed, even if it is not specified as displayed in the segments attribute.

The following example shows results sorted based on GL Account segment and GL Balancing segment for a single structure KFF.

```
SELECT <some columns>
       FROM gl_code_combinations gcc,
            some_other_gl_table sogt
      WHERE gcc.chart_of_accounts_id = :p_id_flex_num
        and sogt.code_combination_id = gcc.code_combination_id
        and <more conditions on sogt>
      ORDER BY <some order by columns>,
               &FLEX_ORDER_BY_GL_ACCOUNT,
               &FLEX_ORDER_BY_GL_BALANCING

     <lexicals>
        <lexical
           type="oracle.apps.fnd.flex.kff.order_by"
           name="FLEX_ORDER_BY_GL_ACCOUNT"
           comment="Comment"
           application_short_name="SQLGL"
           id_flex_code="GL#"
           id_flex_num=":P_ID_FLEX_NUM"
           code_combination_table_alias="gcc"
           segments="GL_ACCOUNT"
           show_parent_segments="Y"/>
        <lexical
           type="oracle.apps.fnd.flex.kff.order_by"
           name="FLEX_ORDER_BY_GL_BALANCING"
           comment="Comment"
           application_short_name="SQLGL"
           id_flex_code="GL#"
           id_flex_num=":P_ID_FLEX_NUM"
           code_combination_table_alias="gcc"
           segments="GL_BALANCING"
           show_parent_segments="N"/>
     </lexicals>
```

Using the Data Engine Java API

This section describes how to utilize BI Publisher's data engine outside of the BI Publisher Enterprise user interface through the Java APIs. Use the descriptions in this section in conjunction with the Javadocs included with your installation files.

Calling a Data Template from the Java API

The following classes comprise the data engine utility Java API:

- oracle.apps.xdo.oa.util.DataTemplate (OA wrapper API)

- oracle.apps.xdo.dataengine.DataProcessor (Core wrapper API)

The DataProcessor class is the main class to use to execute a data template with the BI Publisher Data Engine. To use this API, you will need to instantiate this class and set parameter values for the data template, connection and output destination. Once the parameters are set, you can start processing by calling processData() method.

This example provides a sample data template file, then shows an annotated Java code sample of how to call it.

The sample data template is called EmpDataTemplate.xml and is stored as /home/EmpDataTemplate.xml:

```
<?xml version="1.0" encoding="WINDOWS-1252" ?>
 <dataTemplate name="EmpData" description="Employee Details" Version="1.0">
 <parameters>
  <parameter name="p_DeptNo" dataType="character" />
 </parameters>
 <dataQuery>
  <sqlStatement name="Q1">
  <![CDATA[
   SELECT d.DEPTNO,d.DNAME,d.LOC,EMPNO,ENAME,JOB,MGR,HIREDATE,
    SAL,nvl(COMM,0)
   FROM dept d, emp e
   WHERE d.deptno=e.deptno
   AND d.deptno = nvl(:p_DeptNo,d.deptno)
   ]]>
   </sqlStatement>
  </dataQuery>
  <dataStructure>
  <group name="G_DEPT" source="Q1">
   <element name="DEPT_NUMBER" value="DEPTNO" />
   <element name="DEPT_NAME" value="DNAME" />
   <element name="DEPTSAL" value="G_EMP.SALARY"
    function="SUM()" />
   <element name="LOCATION" value="LOC" />
   <group name="G_EMP" source="Q1">
    <element name="EMPLOYEE_NUMBER" value="EMPNO" />
    <element name="NAME" value="ENAME" />
    <element name="JOB" value="JOB" />
    <element name="MANAGER" value="MGR" />
    <element name="HIREDATE" value="HIREDATE" />
    <element name="SALARY" value="SAL" />
   </group>
  </group>
```

```
   </dataStructure>
  </dataTemplate>
```

The following code sample is an annotated snippet of the Java code used to process the data template by the data engine:

```
 {
  try {

   //Initialization - instantiate the DataProcessor class//
   DataProcessor dataProcessor = new DataProcessor();

    //Set Data Template to be executed
   dataProcessor.setDataTemplate("/home/EmpDataTemplate.xml");

   //Get Parameters - this method will return an array of the
 //parameters in the data template
   ArrayList parameters = dataProcessor.getParameters();
 // Now we have the arraylist we need to iterate over
 // the parameters and assign values to them
   Iterator it = parameters.iterator();

    while (it.hasNext())
   {
      Parameter p = (Parameter) it.next();
      if (p.getName().equals("p_DeptNo"))
 // Here we assign the value ‘10' to the p_DeptNo parameter.
 // This could have been entered from a report submission
 // screen or passed in from another process.
        p.setValue(new "10");
    }
 // The parameter values now need to be assigned
 // to the data template; there are two methods
 // available to do this: 1. Use the setParameters
 // method to assign the 'parameters' object to the template:
    dataProcessor.setParameters(parameters);

 // 2. or you can assign parameter values using a hashtable.

    Hashtable parameters = new Hashtable();
    parameters.put("p_DeptNo","10");
    dataProcessor.setParameters(parameters);
```

```
// Now set the jdbc connection to the database that you
// wish to execute the template against.
// This sample assumes you have already created
// the connection object ‘jdbcConnection'
   dataProcessor.setConnection(jdbcConnection);
// Specify the output directory and file for the data file
   dataProcessor.setOutput("/home/EmpDetails.xml")
// Process the data template
   dataProcessor.processData();
 } catch (Exception e)
  {
  }
 }
```

SQL to XML Processor

The data engine not only supports data generation from data templates, but it can also return data by simply passing it a SQL statement. This functionality is similar to the native database support for generating XML with the added advantage that you can retrieve huge amounts of data in a hierarchical format without sacrificing performance and memory consumption. You SQL statement can also contain parameters that can be given values prior to final processing.

The processor will generate XML in a ROWSET/ROW format. The tag names can be overridden using the setRowsetTag and setRowsTag methods.

The following annotated code sample shows how to use the setSQL method to pass a SQL statement to the data engine and set the element names for the generated data:

```
//Initialization - instantiate the DataProcessor class
DataProcessor dataProcessor = new DataProcessor();
 // Set the SQL to be executed
 dataProcessor.setSQL(  "select invoicenum, invoiceval
                        from invoice_table where
                        supplierid = :SupplID");
//Setup the SuppID value to be used
Hashtable parameters = new Hashtable();
parameters.put("SupplID ","2000");
//Set the parameters
dataProcessor.setParameters(parameters);
//Set the db connection
dataProcessor.setConnection(jdbcConnection);
//Specify the output file name and location
dataProcessor.setOutput("/home/InvoiceDetails.xml")
//Specify the root element tag name for the generated output
dataProcessor.setRowsetTag(“INVOICES”);
//Specify the row elemen tag name for the generated outputt
```

```
dataProcessor.setRowsetTag(“INVOICE”);
//Execute the SQL
dataProcessor.processData();
```

Other Useful Methods

The data engine has several very useful functions that can be used to generate objects or files that can be used with the other BI Publisher APIs:

writeDefaultLayout – once the DataTemplate has been instantiated you can call this method to generate a default RTF template that can be used with the RTFProcessor to create an XSL template to be used with the FOProcessor. Alternatively, the default RTF can be loaded into Microsoft Word for further formatting. This method can generate either a String or Stream output.

writeXMLSchema - once the DataTemplate has been instantiated you can call this method to generate an XML schema representation of your data template. This is very useful if you are working with PDF templates and need to create mapping from the PDF document to your XML data.

setScalableModeOn – if you know you are going to return a large dataset or have a long running query you can specify that the data engine enter scalable mode. This will cause it to use the disk rather than use memory to generate the output.

setMaxRows – this allows you to specify a fixed number of rows to be returned by the engine. This is especially useful when you want to generate some sample data to build a layout template against.

Sample Data Templates

This section contains two sample data templates:

- Employee Listing
- General Ledger Journals Listing

The sample files are annotated to provide a better description of the components of the data template. To see more data template samples, see the BI Publisher page on Oracle Technology Network (OTN). From here you can copy and paste the samples to get you started on your own data templates.

Employee Listing Data Template

This template extracts employee data and department details. It has a single parameter, Department Number, that has to be populated at runtime. The data is extracted using two joined queries that use the bind variable method to join the parent (Q1) query with the child (Q2) query. It also uses the event trigger functionality using a PL/SQL package "employee" to set the where clause on the Q1 query and to provide a group filter on the G_DEPT group.

The sample data template will generate the following XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<dataTemplateName>
```

```
  <LIST_G_DEPT>
   <G_DEPT>
    <DEPT_NUMBER>10</DEPT_NUMBER>
    <DEPT_NAME>ACCOUNTING</DEPT_NAME>
    <LOCATION>NEW YORK</LOCATION>
    <LIST_G_EMP>
     <G_EMP>
      <EMPLOYEE_NUMBER>7782</EMPLOYEE_NUMBER>
      <NAME>CLARK</NAME>
      <JOB>MANAGER</JOB>
      <MANAGER>7839</MANAGER>
      <HIREDATE>1981-06-09T00:00:00.000-07:00</HIREDATE>
      <SALARY>2450</SALARY>
     </G_EMP>
     <G_EMP>
      <EMPLOYEE_NUMBER>7839</EMPLOYEE_NUMBER>
      <NAME>KING</NAME>
      <JOB>PRESIDENT</JOB>
      <MANAGER/>
      <HIREDATE>1981-11-17T00:00:00.000-08:00</HIREDATE>
      <SALARY>5000</SALARY>
     </G_EMP>
     ...
    </LIST_G_EMP>
    <DEPTSAL>12750</DEPTSAL>
   </G_DEPT>
   <G_DEPT>
    <DEPT_NUMBER>20</DEPT_NUMBER>
    <DEPT_NAME>RESEARCH</DEPT_NAME>
    <LOCATION>DALLAS</LOCATION>
    <LIST_G_EMP>
     <G_EMP>
      <EMPLOYEE_NUMBER>7369</EMPLOYEE_NUMBER>
      <NAME>SMITH</NAME>
      <JOB>CLERK</JOB>
      ...
     </G_EMP>
    </LIST_G_EMP>
    <DEPTSAL>10875</DEPTSAL>
   </G_DEPT>
 </LIST_G_DEPT>
 </dataTemplateName>
```

Following is the data template used to extract this data.

```
<?xml version="1.0" encoding="WINDOWS-1252" ?>
- The template is named, an optional description
- can be provided and the default package, if any, is identified:
<dataTemplate name="Employee Listing" description="List of
Employees" dataSourceRef="ORCL_DB1" defaultPackage="employee"
 version="1.0">
    <parameters>
- Defines a single parameter for the Department Number
- with default of 20:
       <parameter name="p_DEPTNO" dataType="character"
        defaultValue="20"/>
    </parameters>
    <dataQuery>
       <sqlStatement name="Q1">
- This extracts the department information based on a
- where clause from a pl/sql package:
          <![CDATA[SELECT DEPTNO,DNAME,LOC from dept
            where &pwhereclause
            order by deptno]]>
       </sqlStatement>
       <sqlStatement name="Q2">
- This second query extracts the employee data and joins to
- the parent query using a bind variable, :DEPTNO
           <![CDATA[SELECT  EMPNO,ENAME,JOB,MGR,HIREDATE,SAL,nvl
             (COMM,0) COMM
                 from EMP
                 WHERE DEPTNO = :DEPTNO ]]>
       </sqlStatement>
       </dataQuery>
- A call is made to a before fetch trigger to set the
- where clause variable in the department query, &pwhereclause:
       <dataTrigger name="beforeReport"
        source="employee.beforeReportTrigger"/>
       <dataStructure>
- The following section specifies the XML hierarchy
- for the returning data:
         <group name="G_DEPT" source="Q1"
```

```
          groupFilter="employee.G_DEPTFilter(:DEPT_NUMBER)">
- There is a group filter placed on the DEPT group.
- This is returned from the employee.G_DEPTFilter plsql package.
- It passes the DEPT_NUMBER value ("name" attribute) rather
- than the DEPTNO value ("value" attribute)

         <element name="DEPT_NUMBER" value="DEPTNO"  />
         <element name="DEPT_NAME"   value="DNAME"/>
- This creates a summary total at the department level based
- on the salaries at the employee level for each department:
         <element name="DEPTSAL" value="G_EMP.SALARY"
          function="SUM()"/>
                   <element name="LOCATION" value="LOC"  />
           <group name="G_EMP" source="Q2">
              <element name="EMPLOYEE_NUMBER" value="EMPNO" />
              <element name="NAME" value="ENAME"/>
              <element name="JOB" value="JOB" />
              <element name="MANAGER" value="MGR"/>
              <element name= "HIREDATE" value="HIREDATE"/>
              <element name="SALARY" value="SAL"/>
            </group>
       </group>
   </dataStructure>
</dataTemplate>
```

The PL/SQL Package:

```
- This is the package specification, it declares the global
- variables and functions contained therein
function BeforeReportTrigger return boolean;
p_DEPTNO NUMBER;
pwhereclause varchar2(3200);
function G_DEPTFilter(deptno number) return boolean;
END;

/
- This is the package body, it contains the code for the
- functions/procedures

create or replace package body employee as

- this is the event trigger called from the data template
- prior to the data fetch. It sets the where clause
- for the department query (Q1) based on the incoming
- data template parameter
 FUNCTION BeforeReportTrigger return boolean is
 begin
  IF (p_DEPTNO=10) THEN
    pwhereclause :='DEPTNO =10';
  elsif (p_DEPTNO=20) THEN
    pwhereclause:='DEPTNO =20';
  elsif (p_DEPTNO=30) THEN
    pwhereclause:='DEPTNO =30';
  elsif (p_DEPTNO=40) THEN
    pwhereclause:='DEPTNO =20';
  else
    pwhereclause:='1=1';
  end if;
 end;
  RETURN(TRUE);
- This function specifies a group filter on the Q1 group.
- If the department number is 30 then the data is not returned.
 FUNCTION G_DEPTFilter(deptno number) return boolean is
 BEGIN
   if deptno = 30 then
    return (FALSE);
  end if;

  RETURN (TRUE);
 end;
END;
/
```

General Ledger Journals Data Template Example

This data template extracts GL journals data from the E-Business Suite General Ledger schema. It is based on an existing Oracle Report that has been converted to a data template format. It follows the same format as the Employee data template but has some added functionality.

```
<?xml version="1.0" encoding="UTF-8" ?>
<dataTemplate name="GLRGNJ" dataSourceRef="ORA_EBS"
 defaultPackage="GLRGNJ" version="1.0">
 <parameters>
- Parameter declaration, these will be populated at runtime.
  <parameter name="P_CONC_REQUEST_ID" dataType = "number"
```

```
     defaultValue="0"></parameter>
    <parameter name="P_JE_SOURCE_NAME" dataType="character">
    </parameter>
    <parameter name="P_SET_OF_BOOKS_ID" dataType="character"
     defaultValue="1"></parameter>
    <parameter name="P_PERIOD_NAME" dataType="character">Dec-97
    </parameter>
    <parameter name="P_BATCH_NAME" dataType="character"></parameter>
    <parameter name="P_POSTING_STATUS" dataType="character"
     defaultValue="P"></parameter>
    <parameter name="P_CURRENCY_CODE" dataType="character"
     defaultValue="USD"></parameter>
    <parameter name="P_START_DATE" dataType = "date"></parameter>
    <parameter name="P_END_DATE" dataType = "date"></parameter>
    <parameter name="P_PAGESIZE" dataType = "number"
     defaultValue="180"></parameter>
    <parameter name="P_KIND" dataType = "character"
     defaultValue="L"></parameter>
   </parameters>
   <lexicals>
  - Flexfield lexical declaration, this specifies the setup required
  - for these flexfield functions.
  - The first will return the full accounting flexfield with
  - the appropriate delimiter e.g. 01-110-6140-0000-000
  <lexical type ="oracle.apps.fnd.flex.kff.select"
     name ="FLEXDATA_DSP"
     application_short_name="SQLGL"
     id_flex_code="GL#"
     id_flex_num=":STRUCT_NUM"
     multiple_id_flex_num="N"
     code_combination_table_alias="CC"
     segments="ALL"
     show_parent_segments="Y"
     output_type="VALUE" />
  - The second will return ‘Y’ if the current combination is
  - secured against the current user, ‘N’ otherwise
  <lexical type ="oracle.apps.fnd.flex.kff.select"
           name ="FLEXDATA_SECURE"
           application_short_name="SQLGL"
           id_flex_code="GL#"
           id_flex_num=":STRUCT_NUM"
           multiple_id_flex_num="N"
```

```
          code_combination_table_alias="CC"
          segments="ALL"
          show_parent_segments="Y"
          output_type="SECURITY" />
  </lexicals>
 <dataQuery>
 <sqlStatement name="Q_MAIN">
 <![CDATA[
  SELECT
  S.user_je_source_name              Source,
  B.name                             Batch_Name,
  B.default_effective_date           Batch_Eff_date,
  B.posted_date                      Batch_Posted_Date,
  B.je_batch_id                      Batch_Id,
  B.running_total_accounted_dr       B_TOT_DR,
  B.running_total_accounted_cr       B_TOT_CR,
  D.je_header_id                     Header_id,
  D.name                             Header_Name,
  C.user_je_category_name            Category,
  D.running_total_accounted_dr       H_TOT_DR,
  D.running_total_accounted_cr       H_TOT_CR,
  J.je_line_num                      Je_Line_Num,
  decode(nvl(CC.code_combination_id, -1), -1, 'A',null)
  FLEXDATA_H,
  J.effective_date                   Line_Effective_Date,
  J.description                      Line_Description,
  J.accounted_dr                     Line_Acc_Dr,
  J.accounted_cr                     Line_Acc_Cr,
  D.currency_code                    Currency_Code,
  D.external_reference               Header_Reference,
  &POSTING_STATUS_SELECT             Recerence1_4,
  nvl(J.stat_amount,0)               Line_Stat_Amount,
  GLL.description                    Batch_Type,
  B.actual_flag                      Actual_Flag,
  GLL2.meaning                       Journal_Type,
  SOB.consolidation_sob_flag         Cons_Sob_Flag,
         &FLEXDATA_DSP FLEXDATA_DSP,
         &FLEXDATA_SECURE FLEXDATA_SECURE
  FROM gl_lookups GLL, gl_je_sources S, gl_je_categories C,
   gl_je_lines J, gl_code_combinations CC, gl_je_headers D,
   gl_je_batches B, gl_lookups GLL2, gl_sets_of_books SOB
  WHERE GLL.lookup_code = B.actual_flag
  AND GLL.lookup_type = 'BATCH_TYPE'
  AND GLL2.lookup_type = 'AB_JOURNAL_TYPE'
  AND GLL2.lookup_code = B.average_journal_flag
  AND SOB.set_of_books_id = :P_SET_OF_BOOKS_ID
  AND S.je_source_name = D.je_source
  AND C.je_category_name = D.je_category
 AND J.code_combination_id = CC.code_combination_id(+)
 AND J.je_header_id = D.je_header_id
 AND &CURRENCY_WHERE
 AND D.je_source = NVL(:P_JE_SOURCE_NAME, D.je_source)
 AND D.je_batch_id = B.je_batch_id
 AND &POSTING_STATUS_WHERE
 AND B.name = NVL(:P_BATCH_NAME, B.name)
 AND &PERIOD_WHERE
 AND B.set_of_books_id = :P_SET_OF_BOOKS_ID
 ORDER BY S.user_je_source_name,
 B.actual_flag,
 B.name,
 B.default_effective_date,
 D.name,
 J.je_line_num
 ]]>
  </sqlStatement>
 </dataQuery>
 - The original report had an AfterParameter
 - and Before report triggers
 <dataTrigger name="afterParameterFormTrigger"
  source="GLRGNJ.afterpform"/>
 <dataTrigger name="beforeReportTrigger"
  source="GLRGNJ.beforereport"/>
 <dataStructure>
 - A very complex XML hierarchy can be built with summary
 - columns referring to lower level elements
 <group name="G_SOURCE" dataType="varchar2" source="Q_MAIN">
   <element name="Source" dataType="varchar2" value="Source"/>
   <element name="SOU_SUM_ACC_DR" function="sum" dataType="number"
    value="G_BATCHES.B_TOTAL_DR"/>
   <element name="SOU_SUM_ACC_CR" function="sum" dataType="number"
    value="G_BATCHES.B_TOTAL_CR"/>
   <element name="SOU_SUM_STAT_AMT" function="sum"
    dataType="number" value="G_BATCHES.B_TOT_STAT_AMT"/>
   <group name="G_BATCHES" dataType="varchar2" source="Q_MAIN">
```

```
<element name="Actual_Flag" dataType="varchar2"
 value="Actual_Flag"/>
<element name="Batch_Id" dataType="number" value="Batch_Id"/>
<element name="Batch_Name" dataType="varchar2"
 value="Batch_Name"/>
<element name="Batch_Eff_date" dataType="date"
 value="Batch_Eff_date"/>
<element name="Journal_Type" dataType="varchar2"
 value="Journal_Type"/>
<element name="Cons_Sob_Flag" dataType="varchar2"
 value="Cons_Sob_Flag"/>
<element name="Batch_Type" dataType="varchar2"
 value="Batch_Type"/>
<element name="Batch_Posted_Date" dataType="date"
 value="Batch_Posted_Date"/>
<element name="B_TOT_DR" dataType="number" value="B_TOT_DR"/>
<element name="B_TOTAL_DR" function="sum" dataType="number"
 value="G_HEADERS.H_Total_Dr"/>
<element name="B_TOT_CR" dataType="number" value="B_TOT_CR"/>
<element name="B_TOTAL_CR" function="sum" dataType="number"
 value="G_HEADERS.H_Total_Cr"/>
<element name="B_TOT_STAT_AMT" function="sum" dataType="number"
 value="G_HEADERS.H_TOT_STAT_AMT"/>
<element name="B_TOTAL_STAT" function="sum" dataType="number"
 value="G_HEADERS.H_Total_Stat"/>
<group name="G_HEADERS" dataType="varchar2" source="Q_MAIN">
 <element name="Header_id" dataType="number"
  value="Header_id"/>
 <element name="Header_Name" dataType="varchar2"
  value="Header_Name"/>
 <element name="Category" dataType="varchar2"
  value="Category"/>
 <element name="Header_Reference" dataType="varchar2"
  value="Header_Reference"/>
 <element name="Currency_Code" dataType="varchar2"
  value="Currency_Code"/>
 <element name="H_TOT_DR" dataType="number" value="H_TOT_DR"/>
 <element name="H_Total_Dr" function="sum" dataType="number"
  value="G_LINES.Line_Acc_Dr"/>
 <element name="H_TOT_CR" dataType="number" value="H_TOT_CR"/>
 <element name="H_Total_Cr" function="sum" dataType="number"
  value="G_LINES.Line_Acc_Cr"/>
```

```
     <element name="H_TOT_STAT_AMT" function="sum"
      dataType="number"
      value="G_LINES.Line_Stat_Amount"/>
     <element name="H_Total_Stat" function="sum" dataType="number"
      value="G_LINES.Line_Stat_Amount"/>
     <group name="G_LINES" dataType="varchar2" source="Q_MAIN"
 groupFilter="GLRGNJ.g_linesgroupfilter(:G_LINES.FLEXDATA_SECURE)">
      <element name="Je_Line_Num" dataType="number"
       value="Je_Line_Num"/>
      <element name="FLEXDATA_H" dataType="varchar2"
       value="FLEXDATA_H"/>
      <element name="FLEXDATA_DSP"  dataType="varchar2"
       value="FLEXDATA_DSP"/>
      <element name="Line_Description" dataType="varchar2"
       value="Line_Description"/>
      <element name="Recerence1_4" dataType="varchar2"
       value="Recerence1_4"/>
      <element name="Line_Acc_Dr" dataType="number"
       value="Line_Acc_Dr"/>
      <element name="Line_Acc_Cr" dataType="number"
       value="Line_Acc_Cr"/>
      <element name="Line_Stat_Amount" dataType="number"
       value="Line_Stat_Amount"/>
      <element name="Line_Effective_Date" dataType="date"
       value="Line_Effective_Date"/>
      <element name="FLEXDATA_SECURE"  dataType="varchar2"
       value="FLEXDATA_SECURE"/>
     </group>
    </group>
   </group>
  </group>
 <element name="R_TOT_DR" function="sum" dataType="number"
  value="G_SOURCE.SOU_SUM_ACC_DR"/>
 <element name="R_TOT_CR" function="sum" dataType="number"
  value="G_SOURCE.SOU_SUM_ACC_CR"/>
 <element name="R_TOT_STAT_AMT" function="sum" dataType="number"
  value="G_SOURCE.SOU_SUM_STAT_AMT"/>
 <element name="JE_SOURCE_DSP" function="first" dataType="number"
  value="G_SOURCE.Source"/>
 <element name="REP_BATCH_ID" function="first" dataType="number"
  value="G_BATCHES.Batch_Id"/>
 <element name="C_DATEFORMAT"  dataType="varchar2"
```

```
  value="C_DATEFORMAT"/>
</dataStructure>
- There is an after fetch trigger, this can be used to clean up
- data or update records to report that they have been reported
<dataTrigger name="afterReportTrigger"
 source="GLRGNJ.afterreport"/>
</dataTemplate>
```

Employee XML Datasource Data Template

This data template combines data that exists in a table called "dept" with data from an xml file called "employee.xml". It follows the same format as the Employee data template but the employee data comes from an xml file instead of from the emp table.

```
<?xml version="1.0" encoding="WINDOWS-1252" ?>
<dataTemplate name="Employee Listing" description="List of Employees" v
 ersion="1.0">
    <parameters>- Defines a single parameter for the Department Number
                - with default of 20:
       <parameter name="p_DEPTNO" dataType="character"
        defaultValue="20"/>
    </parameters>
    <dataQuery>
       <sqlStatement name="Q1">
          <![CDATA[SELECT DEPTNO,DNAME,LOC from dept
                  order by deptno]]>
       </sqlStatement>
 <xml name="empxml" expressionPath=".//ROW[DEPTNO=$DEPTNO]"> - Defines name
    -  and link to DEPTNO in Q1
 <url method="GET" realm="" username="" password="">
  file:///d:/dttest/employee.xml</url> - Defines url for xml data
 </xml>
</dataQuery>-
     <dataStructure>- The following section specifies the XML hierarchy
                   - for the returning data:
         <group name="G_DEPT" source="Q1"
          <element name="DEPT_NUMBER" value="DEPTNO"  />
         <element name="DEPT_NAME"   value="DNAME"/>
- This creates a summary total at the department level based
- on the salaries at the employee level for each department:
         <element name="DEPTSAL" value="G_EMP.SALARY"
          function="SUM()"/>
                 <element name="LOCATION" value="LOC"  />
           <group name="G_EMP" source="empxml">
              <element name="EMPLOYEE_NUMBER" value="EMPNO" />
              <element name="NAME" value="ENAME"/>
              <element name="JOB" value="JOB" />
              <element name="MANAGER" value="MGR"/>
              <element name= "HIREDATE" value="HIREDATE"/>
              <element name="SALARY" value="SAL"/>
            </group>
       </group>
   </dataStructure>
</dataTemplate>
```

Contents | Previous | Top of Page | Next

Previous Next

Home Book List Contents Index Contact Us