Database Actions (2.1 legacy document)

Table of contents

Table of contents

Warning:
This document was copied as is from the Cocoon 2.1 documentation, but has not yet been fully reviewed or moved to its new home.

1. Introduction

Two different sets of actions exist, that deal with (object) relational database access through JDBC. The original database actions provide a relatively simple interface to store, modify, delete and retrieve data. They are oriented towards usage of request parameters for input and request attributes together with sitemap variables for output and do not support auto increment column types. In addition, the description of the database structure is split over several files since these actions attempt to use all tables in a provided description.

The modular database actions provide similar functionality. In contrast to the original actions they allow to store the database meta data in a single file and to switch input and output flexible through the use of modules. Even for auto increment columns specific modules exist that cover a wide range of database management systems.

For an overview of column types supported by the modular database actions, see javadocs for JDBCTypeConversions. The types supported by the original actions are documented in AbstractDatabaseAction.

2. Original Database Actions

The original database actions have evolved quite a bit and at different speeds. The add action is certainly the most complete one, providing support for multiple tables and rows. However, the interface has become a bit inconsistent.

If an error occurs, the original database actions will throw an exception.

2.1. Describing the Structure of your DB - descriptor.xml

The key to database actions is a file that describes database meta data in XML. The original actions will ignore all but the first table and act only on one row. Only the add action will try to access all tables that are contained in this description. As a consequence, each HTML form needs to have a corresponding descriptor file if different tables are affected.

The file name has no meaning and does not need to be descriptor.xml - it can even be a Cocoon pipeline. The name of the root element in a descriptor file is ignored. Only table elements nested on first level inside the root element are parsed by the actions. All unknown elements or attributes are ignored.

For each table a table element needs to be present.

```
<?xml version="1.0"?> <employee> <connection>personnel</connection> <table name="employee"> <keys> <key param="employee" dbcol="id" type="int" mode="manual"/> </keys> <values> <value param="name" dbcol="name" type="string"/> <value param="department" dbcol="department_id" type="int"/> </values> </table> </employee>
```

Describes a single table named "employee". In addition a database connection is specified. See here for more information on database connections.

2.1.1. Key Columns

Tables may or may not have key columns. A key column is a column that is part of the primary key. Actually, candidate keys should do as well.

All key columns are contained in a keys child element of the table element. Each column has a key element to define its properties. The dbcol attribute holds the column name, type is the JDBC type name for this column (have a look at AbstactDatabaseAction source for valid type names), param specifies the name of the request parameter to use, and mode sets how the value for this column is obtained on adding a row.

Through the mode attribute the behaviour of the add action can be changed.

Default mode is "automatic" and to let the database create the key value by setting this value to null. The created value can not be read back from the database and will not be available as request attribute or sitemap variable.

A mode of "manual" will query the database for the maximum current value, add 1 to it and use that for a value.

A mode of "form" will use the corresponding request parameter.

A mode of "request-attribute" will use the corresponding request attribute. The name specified in the param attribute will be automatically prefixed with the class name.

Key values will be propagated to sitemap variables and - prefixed with the class name - request attributes.

2.1.2. Other Columns

All other columns are contained in a values child element of the table element. Each column has a value element to define its properties. Properties are similar to those for key columns. A mode attribute does not exist for value columns. Instead, request parameters and request attributes are tried in this order for the specified parameter.

Request attribute names are *not* prefixed with the class name. Thus, to insert the value of a key column of the previous row or previous table into a value column, it needs to be named org.apache.cocoon.acting.AbstractDatabaseAction:key:table:dbcol.

Value columns are propagated to request attributes with class name prefix. They are not available for the sitemap.

3. Modular Database Actions

The modular database actions were mainly created to make auto increment columns available, handle input and output flexibly, and have a consistent interface. A successful action will return the number of rows affected in a sitemap parameter named row-count. The added features required to change the descriptor file format in incompatible ways.

It can be configured if an exception will be thrown when an error occurs.

3.1. Describing the Structure of your DB - descriptor.xml

Like the original actions, the modular actions need meta data in an XML file. However, that file may contain any number of tables, not just the ones needed for a single request. The tables actually used are referenced through a table-set. Unknown elements and attributes are ignored. This way a descriptor file can be shared with other actions like the form validator.

For the flexible input and output handling, the modular database actions rely on modules. Have a look at those before proceeding.

The following is a snippet from a descriptor file.

<root> <connection>personnel</connection> <table name="user" alias="user"> <keys> <key name="uid" type="int" autoincrement="true"> <mode name="auto" type="autoincr"/> </key> </keys> <values> <value name="name" type="string"></value> <value name="firstname" type="string"></value> <value name="uname" type="string"></value> </values> </table>

The table element has an additional attribute alias which is an alternative name to reference the table from a table set. The descriptor file is searched top down for tables whose name or alias match. The aliasn attribute is useful if a complex join expression is used as table name. In such a case modifications like update, insert, delete will likely fail.

Another application of aliases if different numbers of columns should be affected by an operation. or if a table contains several candidate keys that are used alternatively. This way, different views to a table can be created.

3.1.1. Key Columns

The descriptor file resembles the one for the original actions. One major difference is the absence of dbcol and param attributes. Instead there is a name attribute which corresponds to the dbcol attribute and specifies the database column name.

If a column is an auto increment column, the similar named attribute indicates this. Auto increment columns will be handled differently on insert operations.

Instead of specifying a parameter name, the actions support to use different input modules for each operation through the nested mode elements. This is described in more detail below.

Note here though, that not every column needs a mode element: The actions default to the module defined as request which is in a default installation to obtain the values from request parameters. The name of the parameter defaults to table name dot column name.

3.1.2. Other Columns

Apart from the fact that the auto increment columns are only supported for key columns, everything said above applies to value columns as well.

3.1.3. Operation Mode Types

Basically, two different mode types exist: autoincrement which is used whenever data shall be inserted into a table and this particular key column has the auto increment attribute set and others for all other requirements.

In addition, a table-set can specify different mode types to use instead of the predefined type names. Through this, and the fact that every mode can specify a different input module, it is easy to use different input modules for different tasks and forms.

One special mode type name exists that matches all requested ones: all This makes it easier to configure only some columns differently for each table-set.

3.1.4. How to obtain Values

As said above, these actions default to reading from request parameters with a default parameter name. By specifying mode elements, this can be overridden. Any component that implements the InputModule interface can be used to obtain values. How to make such modules known to Apache Cocoon is described elsewhere.

Beside using different input modules, their parameters can be set in place, for example to override parameter names, configure a random generator or a message digest algorithm.

```
<table name="user_groups"> <keys> <key name="uid" type="int"> <mode name="request"
type="request"> <parameter>user_groups.uid</parameter> </mode> <mode
name="attribute" type="attrib">
<parameter>org.apache.cocoon.components.modules.output.OutputModule:user.uid[0]</parameter>
</mode> </key> <key name="gid" type="int" set="master"> <mode name="request"
type="all"> <parameter>user_groups.gid</parameter> </mode> </key> </keys> </table>
```

The above example shows just that: the parameter element is not read by the database action itself but the complete mode configuration object is passed to the input module. Both the request attribute and the request parameter input modules understand this parameter attribute which takes precedence over the default one.

Another feature when obtaining values is tied to the type attribute: Different modes can be used in different situations. The basic setup uses two different mode types: autoincrement when inserting in key columns that have an indicator that they are indeed auto increment columns and others for insert operations on all other columns and all other operations on all columns.

Table-sets can override the default names for these two mode type name categories with arbitrary names except the special name all. A mode that carries the type name "all" is used with all requested type names. Lookup obeys first match principle so that all modes are tested from top to bottom and the first that matches is used.

3.1.5. How to store Values e.g. in your Session

All modular database action can be configured to use any component that implements the OutputModule interface to store values. The output module is chosen on declaring the action in the sitemap or dynamically with a sitemap parameter. If no output module is specified, the default it to use the request attribute module.

The interface does not allow to pass configuration information to the output module. This has to be done when the module is declared e.g. in cocoon.xconf.

3.1.6. Inserting Multiple Rows - Sets

Once common task is to work on more than one row. If the rows are in different tables, this is catered for by table-sets. Operating on multiple rows of one table requires to mark columns that should vary and among those one, that determines the number of rows to work on.

This is done with sets. All columns that cary a set attribute can vary, those, that don't, are kept fixed during the operation. The column that is used to determine the number of rows is required to have a value of master while all others need to have a value of slave for the set attribute. There may be only one master in a set.

Sets can be tagged either on column or on mode level but not both for a single column.

3.1.7. Select Your Tables - Table-Sets

Tables that should be used during an operation can be grouped together with a table-set. A table-set references tables by their name or their alias.

In addition, a table-set can override the mode type names for the two categories "autoincrement" and "others".

Operations spanning multiple tables in a table-set are done in a single transaction. Thus, if one fails, the other is rolled back.

<table name="groups"> <keys> <key name="gid" type="int" autoincrement="true"> <mode name="auto" type="autoincr"/> </key> </keys> <values> <value name="gname" type="string"/> </values> </table> <table-set name="user"> <table name="user"/> </table-set> <table-set name="groups"> <table name="groups"/> </table-set> <table-set name="user+groups"> <table name="user"/> <table name="user_groups" others-mode="attrib"/> </table-set> <table-set name="user_groups"> <table name="user_groups" others-mode="request"/> </table-set> </root>

1. Comments

add your comments