ModularDatabaseActions

The **Modular Database Actions** provide similar funcionality as OriginalDatabaseActions. Were mainly created to make avaliable the support for autoincrement columns, handle input and output flexibily and have a consistent interface.

A sucessful action will return the number of affected rows into a sitemap parameter called `row-count`

You can also use FormValidationUsingCocoon before send the data to the database

See the List of DatatypesOfModularDatabaseActions

Main improvements over the OriginalDatabaseActions

1. Allow to store all the database structure in a single file
2. Flexible input and output using modules.
3. Support for autoincrement column type that cover a wide range of database systems.
4. Transactional operations over more than one table

Usage

Describing the Structure of your DB - descriptor.xml

All the metadata are stored in a single XML file, that can contain the description of any number of tables. Into this file we can include description rules for others Actions like FormValidatorAction. In this way we can have all the database metadata in the same file.

Example of 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 `<root>` tag The `root` element of the document is ignored, and can have any name. This allows a single document to contain both database description and form validation configuration.

The `<connection>` tag This tag identifies a database connection pool, see ConnectionPooling.

The `<table>` tag Describes a complete table. The file can contain several table elements. It has two attributes:

- `name` – is the name of the table into the database.
- `alias` – is and optional attribute. Is used to create an alternative name of a table, wich can be used inside a `table-set` element. The `table-set` tag can be used if a complex join expresions is used as table name. Another usage is when different number of columns will be affected by different operations or if a table contains several candidate keys that are used alternatively. This way we can create "different views" of the same table.

To find a table, the descriptor file is searched top-down for tables whose `name` or `alias` match. The first that matches is used.

The `<key>` Tag - Defining Key Columns The descriptor file resembles the one for the OriginalDatabaseActions. Note the absence of `dbcol` and `param` attributes. See the DatatypesOfModularDatabaseActions. It has 2 new attributes:

- `name` - Specifies the database column name. Corresponds to the old `dbcol` attribute.
- `autoincrement` - Indicate if a column is of this type. Autoincrement columns will be handled differently on insert operations.

Instead of specifying a parameter name, the modular actions support the use of different InputModules for each operation through the nested `mode` elements.

No every column needs a **mode** element. The actions default module is defined as **request_param** in the default installation to obtain the values from request parameters. The implicit name of the parameter is **table_name.column_name**.

The <value> Tag - Defining No key or Other Columns Everything said above about the Key Columns applies to value columns as well. Except for the autoincrement attribute. See the DatatypesOfModularDatabaseActions.

Operation Mode Types

Describes the modes of operation against the database. Basically, two different mode types exist:

- **autoincrement** which is used only on INSERT when a key has set this attribute 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 InputModules, it is easy to use different input modules for different tasks and forms.

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

How to obtain Values

As said above, the modular actions default to reading from **request parameters** with a default parameter name. This can be overridden using `mode` elements. Any component that implements the InputModule interface can be used to obtain values. How to make such modules known to Apache Cocoon is described in http://xml.apache.org/cocoon/userdocs/concepts/modules.html.

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

The above example shows that: the `parameter` attribute 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 the `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 into 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**, wich 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.

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.

Note

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.

Inserting Multiple Rows - Sets

A common task need 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.

Select Your Tables - Table-Sets

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

Also, a table-set can override the `mode` attribute 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="request-attr"/>
   </table-set>

   <table-set name="user_groups">
      <table name="user_groups" others-mode="request-param"/>
   </table-set>

</root>
```