HowToCreateHiearchicalXmlUsingESQL

How to create hiearchical xml using ESQL

This How-To describes the steps to write xsp that produces XML hierarchical structure using ESQL nested queries.

Author: **Pavel Vrecion**

Purpose

There is quite often need to create master-detail web pages based on database queries.

Typical example is company that has more departments with employees.

In this case we would like to create XML structure starting with company, with more child elements describing department and on next level also child elements for each employee. Note that for such tasks Cocoon has already partial support in ESQL called group queries, but this is limited to two level queries. We will examine the other way, how to achieve this. It will enable us to create richer XML structures.

During the process we will also learn how to interweave Java and SQL.

Prerequisites

We will need Cocoon (of course) and database with at least two tables linked by “one to many” reference.

In the example we will use Firebird database engine and we will utilize example databases employee.fdb, which is included in Firebird installation. But other databases with JDBC support will work, too.

Consult Cocoon and/or JDBC and your database documentations for details regarding database installation, running, parameters and JDBC connections.

Steps briefly

We will:

1. create at least two tables (or you can skip this step and use already existing database)

2. define database connection pool in cocoon.xconf (Cocoon main configuration file)

3. create application with:

- simple sitemap
- and simple xsp (XML Server Page)

4. create simple stylesheet to transform XML into HTML

Steps in detail

Database tables

We will need two tables, one for department and second for employee.

Two tables will be linked through “one to many” relationship.

Employee table will have department foreign key department id, which will link employee to department.

You can create tables with following SQL DDL statements:

```
CREATE TABLE DEPARTMENT (
    DEPT_NO     CHAR(3) NOT NULL,
    DEPARTMENT  VARCHAR(25) NOT NULL,
    HEAD_DEPT   CHAR(3),
    LOCATION    VARCHAR(15),
    PHONE_NO    VARCHAR(30)
```

```
CREATE TABLE EMPLOYEE (
    EMP_NO       INTEGER,
    FIRST_NAME   VARCHAR(30),
    LAST_NAME    VARCHAR(40),
    PHONE_EXT    VARCHAR(4),
    DEPT_NO      CHAR(3),
    JOB_CODE     VARCHAR(4),
    JOB_COUNTRY  VARCHAR(15)
```

Well, if you like, you can also create some integrity constraints and other fancy database stuff like sequences, triggers, but for our purpose SQL, we have mentioned above, is fine.

Note that column `DEPT_NO` is primary key in `DEPARTMENT` table (and so should have mandatory and unique value). Same column and its value is used in `EMPLOYEE` table to link specific employee to specific department. HEAD_DEPT is “self reference” and can be used to create rich company structure that consists from divisions, departments, sub-departments etc.

To verify that everything works you should test database using some SQL tool. It is good idea to add some test, but meaningful, data during the process, too.

When we have tables in the database we must tell Cocoon, where to find the data and how to connect to database. That is why we will add new datasource to Cocoon configuration file.

Adding new datasource to cocoon.xconf

Find `cocoon.xconf` (usually located in Cocoon directory `build/webapp/WEB-INF`), open it with some text editor, find datasources section and add new connection parameters. After editing datasource part of {`cocoon.xconf` it will look somehow like this:
{{{<datasources>
....
<!-- Test company database -->
<jdbc name="dbCompany">
<pool-controller max="5" min="1"/>
<auto-commit>true</auto-commit>
<dburl>jdbc:firebirdsql:localhost/3050:c:\PathToDB\EMPLOYEE.FDB</dburl>
<user>MyName</user>
<password>MyPassword</password>
</jdbc>
.....
}}}
We suppose that you have already downloaded and installed database engine and libraries, and you have added JDBC driver name to Cocoon `web.xml` configuration file (located in the same directory as `cocoon.xconf`).

Your `web.xml` should have param-value element under servlet/init-param elements similar to this:
{{{<init-param>
<param-name>load-class</param-name>
<param-value>

<!-- For parent ComponentManager sample:
org.apache.cocoon.samples.parentcm.Configurator -->

<!-- For IBM WebSphere:
com.ibm.servlet.classloader.Handler -->

<!-- For Database Driver: -->
org.hsqldb.jdbcDriver

<!-- For Firebird: -->
org.firebirdsql.jdbc.FBDriver

<!-- For MySQL Driver: -->
com.mysql.jdbc.Driver

</param-value>
</init-param>
}}}
Note that in this example there are specified drivers for several database engines. If you use some other database, please consult corresponding documentation.

After these steps we should have database and Cocoon up and running, Cocoon also should already “know” how to connect to database.

We are ready to start writing xsp and ultimately to show some data to crowd of cheering users.

Writing XSP

As you remember, main purpose of our xsp is to present data from database in rich xml structure. We will do it by executing SQL SELECT command that will return rowset of departments.

We will save department id (identification, or primary key) into Java variable and use it in nested query.

Nested query will then produce list of employees for each department.

Clear? No? Don't worry, and let's go to examples.

Create new directory called test (for example) in Cocoon `build/webapp` directory (other applications are located there, too) and create new text file named empxml.xsp (you can name it differently, of course). Add following code to it and save it.

```
{{{<?xml version="1.0"?>
<xsp:page language="java"
xmlns:xsp="http://apache.org/xsp"
xmlns:esql="http://apache.org/cocoon/SQL/v2"
xmlns:xsp-request="http://apache.org/xsp/request/2.0"
xmlns:xsp-session="http://apache.org/xsp/session/2.0"
xmlns:cinclude="http://apache.org/cocoon/include/1.0"
xmlns:source="http://apache.org/cocoon/source/1.0"
xmlns:util="http://apache.org/xsp/util/2.0"
>

<!-- declare integer variable called deptno -->
<xsp:logic>int deptno=-1;</xsp:logic>

<!-- create root xml element -->
<Company>

<!-- start esql section using database connection declared in cocoon.xconf -->
<esql:connection>
<esql:pool>dbCompany</esql:pool>

<!-- query departments -->
<esql:execute-query>
<esql:query>select DEPT_NO, DEPARTMENT, LOCATION, PHONE_NO from DEPARTMENT</esql:query>
<esql:results>
<esql:row-results>
<!-- get department id from database and store it in deptno variable -->
<xsp:logic>deptno=<esql:get-int column="DEPT_NO"/>;</xsp:logic>
<!-- create XML element for department and add some interesting values from database -->
<Department>
<DepartmentName><esql:get-string column="DEPARTMENT"/></DepartmentName>
<DepartmentLocation><esql:get-string column="LOCATION"/></DepartmentLocation>
<DepartmentPhone><esql:get-string column="PHONE_NO"/></DepartmentPhone>
<!-- create XML element for emloyees and add some data -->
<Employees>

<!-- nested query for employees
For each record in DATABASE table do the query for all employees in given department.
In select statement saved value of department number (primary key) is used
-->
<esql:execute-query>
<esql:query>
select FIRST_NAME, LAST_NAME, PHONE_EXT, JOB_CODE, JOB_COUNTRY
from EMPLOYEE where DEPT_NO = <xsp:expr>deptno</xsp:expr>
order by LAST_NAME, FIRST_NAME
</esql:query>
<esql:results>
<esql:row-results>
<!-- create XML element for employee and add some child elements with values -->
<Employee>
<Lastname><esql:get-string column="LAST_NAME"/></Lastname>
<Firstname><esql:get-string column="FIRST_NAME"/></Firstname>
<JobCode><esql:get-string column="JOB_CODE"/></JobCode>
<JobCountry><esql:get-string column="JOB_COUNTRY"/></JobCountry>
<PhoneExt><esql:get-string column="PHONE_EXT"/></PhoneExt>
</Employee>
</esql:row-results>
</esql:results>
</esql:execute-query>

</Employees>
</Department>

</esql:row-results>
</esql:results>
</esql:execute-query>
</esql:connection>
</Company>

</xsp:page>
}}}
```

OK we have xsp page. Note that we have filled Java variable **with** ESQL command

```
{{{<xsp:logic>deptno=<esql:get-int column="DEPT_NO"/>;</xsp:logic>
}}}
```

And then used Java variable **in** ESQL command

```
{{{<esql:query>
select FIRST_NAME, LAST_NAME, PHONE_EXT, JOB_CODE, JOB_COUNTRY
from EMPLOYEE where DEPT_NO = <xsp:expr>deptno</xsp:expr>
order by LAST_NAME, FIRST_NAME
</esql:query>
}}}
```

That is the main trick of nested queries and hierarchical structures.

As a next step we will create simple sitemap in our application directory.

Simple sitemap

Sitemap will be really simple. Its only purpose is to show one page, we have just created. Create new text file called sitemap.xmap in application directory and fill it with following text:

```
{{{<?xml version="1.0"?>
<map:sitemap xmlns:map="http://apache.org/cocoon/sitemap/1.0">

<map:pipelines>
<map:pipeline>

<map:match pattern="employees.xml">
<map:generate type="serverpages" src="empxml.xsp" />
<map:serialize type="xml"/>
<map:serialize/>
</map:match>

</map:pipeline>
</map:pipelines>

</map:sitemap>
}}}
```

At this stage we should have 2 files in our example application directory:

- `empxml.xsp`
- `sitemap.xmap`

We have created xsp that produces xml structure. We have also sitemap with pipeline that calls this xsp when Cocoon gets employee.xml request.

So let’s try it. In your favorite browser type url (when you run Cocoon locally, your url will probably be `http://localhost:8888/test/employees.xml`).

You should see xsp result in the structured XML form, something like this:

```
{{{<Company>
<Department>
<DepartmentName>Corporate Headquarters</DepartmentName>
<DepartmentLocation>Monterey</DepartmentLocation>
<DepartmentPhone>(408) 555-1234</DepartmentPhone>
<Employees>
</Employees>
</Department>
<Department>
<DepartmentName>Sales and Marketing</DepartmentName>
<DepartmentLocation>San Francisco</DepartmentLocation>
<DepartmentPhone>(415) 555-1234</DepartmentPhone>
<Employees>
<Employee>
<Lastname>MacDonald</Lastname>
<Firstname>Mary S.</Firstname>
<JobCode>VP</JobCode>
<JobCountry>USA</JobCountry>
<PhoneExt>477</PhoneExt>
</Employee>
<Employee>
<Lastname>Yanowski</Lastname>
<Firstname>Michael</Firstname>
<JobCode>SRep</JobCode>
<JobCountry>USA</JobCountry>
<PhoneExt>492</PhoneExt>
</Employee>
</Employees>
</Department>
<Department>
<DepartmentName>Engineering</DepartmentName>
......
}}}
```

Note that department can be linked to parent department through HEAD_DEPT column. So, as homework, you can enhance example application by modifying xsp so that it will contain nested queries:

```
select * from DEPARTMENT where HEAD_DEPT is null -- (get company divisions)
select * from EMPLOYEE where DEPT_NO = <xsp:expr>deptno</xsp:expr> -- (get division employees)
select * from DEPARTMENT where HEAD_DEPT = <xsp:expr>deptno</xsp:expr> -- (get divisions departments i.e.
second level departments)
select * from EMPLOYEE where DEPT_NO = <xsp:expr>deptno</xsp:expr> -- (get department employees)
```

and so on for other levels. In this case resulting XML will have more levels because DEPARTMENT elements will have child DEPARTMENT elements (you can also use DIVISION - DEPARTMENT, or other tag names according to your needs or taste).

You should see xml, maybe even in the structure, you have expected. You are happy, but your users expect some fancy graphics with company logo, and they strictly demand yellow text on pink background.

Well, make them happy, too. We will write xsl and postpone our celebration, but just a little.

XSL has one hidden virtue, it divides data and presentation. And we just know, that our users will strictly demand something else next week, and quite often they will act in separated groups divided by languages, browsers etc. This approach opens the way how to serve them all, and how to make later changes fast.

Simple XSL

To keep things simple and clear, we will start with several HTML lines for each department and one table for department employees.

Create new file `employees.xsl` in application directory and fill it with following content:

```
{{{<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
>

<xsl:template match="Company">
<html>
<head><title>My company departments</title></head>
</html>
<body>
```

```
<xsl:for-each select = "Department">
<h1><xsl:value-of select="DepartmentName"/></h1>
Location: <xsl:value-of select="DepartmentLocation"/><br/>
Phone: <xsl:value-of select="DepartmentPhone"/><br/>
<xsl:for-each select = "Employees"><br/>
<table border="1">
<caption>Department employees</caption>
<thead>
<th>Employee</th>
<th>Job code</th>
<th>Job country</th>
<th>Phone extension</th>
</thead>
<tbody>
<xsl:for-each select = "Employee">
<tr>
<td>
<xsl:value-of select="Lastname"/>
<xsl:text> </xsl:text>
<xsl:value-of select="Firstname"/>
</td>
<td><xsl:value-of select="JobCode"/></td>
<td><xsl:value-of select="JobCountry"/></td>
<td><xsl:value-of select="PhoneExt"/></td>
</tr>
</xsl:for-each>
</tbody>
</table>
</xsl:for-each>
<hr/>
</xsl:for-each>

</body>
</xsl:template>

</xsl:stylesheet>
```
}}}
Now we will use this xslt code to transform xml data into html. To do it, we must enhance Cocoon application sitemap a little. Add several lines to `sitemap.xmap`, so the new sitemap will look like this:

```
<?xml version="1.0"?>
<map:sitemap xmlns:map="http://apache.org/cocoon/sitemap/1.0">

        <map:pipelines>
                <map:pipeline>

                        <map:match pattern="employees.xml">
                                <map:generate type="serverpages" src="empxml.xsp" />
                                <map:serialize type="xml"/>
                                <map:serialize/>
                        </map:match>

                        <map:match pattern="employees.html">
                                <map:generate type="serverpages" src="empxml.xsp" />
                                <map:transform src="employees.xsl"/>
                                <map:serialize type="html"/>
                                <map:serialize/>
                        </map:match>

                </map:pipeline>
        </map:pipelines>

</map:sitemap>
```

At this stage we should have 3 files in our example application directory:

- `empxml.xsp` (producing xml from database)
- `sitemap.xmap` (describing our application components and basic flow of pages)
- `employees.xsl` (transforming xml into html)

After calling `employees.html` (for example `http://localhost:8888/test/employees.html` we should see html formatted text:

Corporate Headquarters

Location: Monterey

Phone: (408) 555-1234

	Department employees		
Employee	Job code	Job country	Phone extension

Sales and Marketing

Location: San Francisco

Phone: (415) 555-1234

	Department employees		
Employee	Job code	Job country	Phone extension
Warvick Mary S.	VP	USA	477
Yanowski Michael	SRep	USA	492

Engineering

Location: Monterey

Phone: (408) 555-1234

	Department employees		
Employee	Job code	Job country	Phone extension
Brown Kelly	Admin	USA	202
Nelson Robert	VP	USA	250

....

In praxis you will probably spend some more time playing with HTML layout, fonts, colors and graphics, but believe me, better is to find somebody who really has the taste, and who will do HTML coding in xsl much better then we, programmers, usually do.

Achievements

We have learned how to use xsp and how to mix Java and SQL codes to create hierarchical structures.

Such approach has several advantages. Code is short and more readable.

Hierarchical XML structure simplifies transformations like using XSLT for HTML pages, as well as any other automated processing. And, as a bonus, data, logic and presentation layers are kept apart.

Comments

Care to comment on this How-To?

Got another tip?

Help keep this How-To relevant by passing along any useful feedback to the author, Pavel Vrecion.