

Publish dynamic XML content with Cocoon 2, Part 3: Cocoon 2: Build database-driven sites

Generate and publish XML and HTML content based on database data

Skill Level: Introductory

[Leigh Dodds \(leigh@xmlhack.com\)](mailto:leigh@xmlhack.com)

Developer and editor
Ingenta, Ltd.

30 May 2002

The first two tutorials in this series introduced the basic concepts of the Cocoon 2 architecture and XML Server Pages (XSP) as a means for creating and publishing dynamic XML content. This tutorial goes a natural step further and discusses how to use Cocoon 2 to generate and publish XML and HTML content based on database data. Any Web site that has reached a certain size and complexity will require a database to manage at least part, if not all, of its data.

Section 1. About this tutorial

Should I take this tutorial?

This tutorial assumes that you are familiar with the Cocoon 2 application and with XML Server Pages (XSP). The previous two tutorials in this series, "Introducing Cocoon 2" and "Working with XML Server Pages", cover the relevant background material (see [Resources](#)).

The tutorial is for Java developers who have previous experience working with databases in a Java environment and wish to learn how to apply that knowledge to

building Cocoon 2 applications. You should at least be familiar with JDBC or SQL although no detailed knowledge is required. If you need a refresher, check [Resources](#) for links to other tutorials that cover these technologies.

In this tutorial, you learn to:

- Configure and manage JDBC connection pools for use in Cocoon 2 applications
- Select data from databases and publish it as XML using XML Server Pages
- Insert, update, and delete data
- Configure Cocoon 2 to carry out form validation on user-submitted data

Databases and Cocoon 2

The first two tutorials in this series introduced the basic concepts of the Cocoon 2 architecture and XML Server Pages (XSP) as a means for creating and publishing dynamic XML content. This tutorial goes a natural step further and discusses how to use Cocoon 2 to generate and publish XML and HTML content based on database data. Any Web site that has reached a certain size and complexity will require a database to manage at least part, if not all, of its data.

Java frameworks that provide database access generally include the following basic features:

- Connection pooling for managing connections to the database
- A database abstraction layer constructed around the JDBC API
- Form handling functionality to provide validation and feedback for user-submitted data

Each of these features is supported by Cocoon 2 and will be discussed in this tutorial. Because Cocoon 2 is an XML framework, this functionality can also be used to expose legacy data for application integration purposes as well as direct Web site publishing.

While Cocoon 2 introduces these database integration features, it was not designed as a complete Java application server framework. Cocoon 2 does not compete with servers based around the J2EE architecture as it doesn't provide the same level of resource and transaction management, load-balancing, and deployment options. Cocoon 2 is most suitable for applications that involve little sophisticated business logic and in which processing is mostly aimed at content generation.

However, you can still use Cocoon 2 in conjunction with an EJB server. Cocoon 2 can easily act as a replacement presentation layer rather than the more commonly used Java Servlets and Java Server Pages. In short, Cocoon 2 aims to meet the 80/20 rule by delivering the most flexibility from a small number of features.

Tools

The examples in this tutorial have been created and tested using Cocoon 2.0.2. It is assumed that the user has already successfully installed Cocoon as detailed in the "Introducing Cocoon 2" tutorial. (See [Resources](#)). [Download](#) the example code for this tutorial in `dbc2.tar`, the sample archive file.

To get the most out of this tutorial, you should have access to a relational database, either an open source database such as MySQL, or a commercial product such as DB2. See <http://mysql.com> for download information and technical documentation on MySQL. See <http://www.ibm.com/software/data/db2/> for product information related to IBM's DB2 database. This tutorial does not cover the set-up and installation of a database environment.

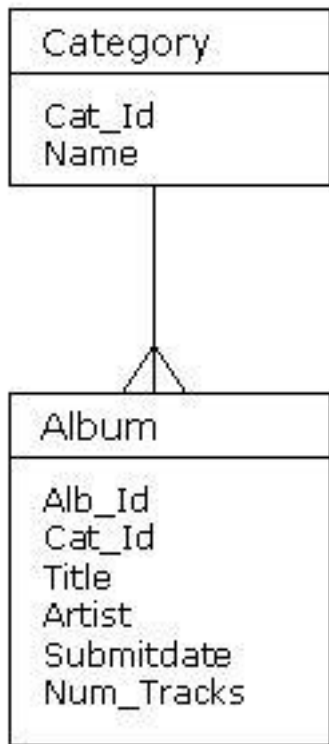
You should have a suitable Java JDBC driver for your database. See [Resources](#) for links to Web sites from which you can obtain JDBC drivers. Drivers are generally packaged as part of a database distribution.

Throughout the tutorial, two environment variables are frequently referenced. These are defined as follows:

- `$TOMCAT_HOME` -- the directory in which the Tomcat application server has been installed (for example, `/usr/local/jakarta-tomcat-4.0.1`)
- `$COCOON_HOME` -- the Cocoon Web application directory; this is typically `$TOMCAT_HOME/webapps/cocoon`

The sample application

Figure 1. Diagram of sample database schema



The examples in this tutorial are all based on a very basic sample application. The sample application has been kept intentionally simple to not obscure the technical details. The tutorial's emphasis is on the technical aspects of the database integration; therefore basic steps such as using XSP pages or configuring pipelines is not covered.

The demonstration application is a music CD catalog capable of storing lists of CDs categorized by musical genre. This basic functionality could be used as the core of an e-commerce system that sells music across the Web, or as a community Web site that allows fans to submit music reviews.

The schema for this simple database contains only two tables whose relationship is shown in the diagram on the left. A *category* has a unique identifier (`cat_id`), and a human-readable name.

An *album* has the following properties:

- A numeric identifier (`alb_id`)
- An associated category (`cat_id`)
- A title

- An artist/group name (`artist`)
- A date recording when it was added to the catalog (`submitdate`)
- A count of the number of tracks on the album (`num_tracks`)

Many other kinds of information could be recorded, but these simple properties are sufficient to demonstrate much of the Cocoon 2 functionality.

Installing the samples

The sample archive, which can be accessed from [Download](#), includes an SQL script to generate the database schema and populate it with some sample data. To install the samples, do the following steps:

1. Shutdown Tomcat.
2. Unzip the sample archive into `$TOMCAT_HOME/webapps`, preserving the directory structure. This creates a `dbc2` directory that contains a Cocoon 2 sitemap and the samples.
3. Copy over the `$COCOON_HOME/WEB-INF` into the newly created `$TOMCAT_HOME/webapps/dbc2` directory. This creates a second instance of the Cocoon 2 Web application for running the samples.
4. Run the `$TOMCAT_HOME/webapps/dbc2/dbc2.sql` script in your database environment to create the sample data.
5. Follow the instructions in the next section, [Configuring database connections](#), to configure a database connection pool.
6. Once this is configured correctly, you can access the samples from `http://localhost:8080/cocoon/dbc2/index.html`.

Section 2. Configuring database connections

Database connections overview

Creating database connections when needed in an application is generally inefficient. This is due to the high cost associated with setting up and then closing

individual connections. Like other expensive processing resources, connections should be managed using a *pool*. Connections can be taken from and placed back into the pool by individual components, leaving the application to manage the pool. Pools are typically created when the application is started, and may grow or shrink in size depending on their usage.

This section introduces how to create database connection pools in Cocoon 2. This simple process involves first installing and loading the appropriate JDBC driver for the database being used, and then configuring Cocoon 2 to use this driver to create one or more database connections. The next sections demonstrate how you must carry out these steps.

Installing a JDBC driver

Installing a JDBC driver is simply a matter of configuring the Java CLASSPATH to ensure that the required classes are available. Consult the documentation for the database that you are using to find out where to obtain the JDBC driver, or consult the list of available JDBC drivers (see [Resources](#)). Drivers are typically packaged as either zip or jar archives.

You can configure the CLASSPATH in several ways, depending on the servlet container that's being used. In general, you can simply edit the startup script for the application server so its initial CLASSPATH environment includes the requisite archive. As an alternative, you could copy the archive into the `$COCOON_HOME/WEB-INF/lib` directory. Cocoon 2 automatically adds any zip or jar archives in this directory to its own CLASSPATH.

If the JDBC drivers are already installed elsewhere (for example, as part of the default database installation), then you can configure Cocoon 2 to load them from an alternate location. Edit the `$COCOON_HOME/WEB-INF/web.xml` configuration file and alter the `extra-classpath` configuration parameter as follows:

```
<init-param>
  <param-name>extra-classpath</param-name>
  <param-value>WEB-INF/extra-classes1:/ABSOLUTE-PATH-TO-ARCHIVE</param-value>
</init-param>
```

Substitute the absolute path to the driver archive where indicated.

Installing a JDBC driver under Tomcat

When you run Cocoon 2 under Tomcat, you can also copy the archive to the `$TOMCAT_HOME/common/lib` directory. Tomcat automatically adds all jar files in this directory to its CLASSPATH. Then, you simply rename zip files with a `.jar`

extension to get Tomcat to add the archive using this method.

While several options are available, it's recommended that you simply place the archive in the `$COCOON_HOME/WEB-INF/lib` directory. This simple copy procedure avoids the need to edit a configuration file. However, if the driver will be used by other Web applications running under Tomcat, you should place the archive in the `$TOMCAT_HOME/common/lib` directory where it will be globally available.

Loading the JDBC driver

Once you have installed the driver, you must load it into the virtual machine and register it with the JDBC `java.sql.DriverManager` (see [Resources](#)) before you can create connections with it. In a command-line application, you can achieve this with the following statement:

```
Class.forName("fully.specified.classname.of.Driver");
```

JDBC drivers must register themselves as soon as the class is loaded. To achieve the same effect in a Web application, you must edit the `load-class` configuration parameter in the `$COCOON_HOME/WEB-INF/web.xml` file. This parameter is already present in the default Cocoon 2 configuration, so instructing Cocoon 2 to load additional drivers is simply a matter of adding a new driver to the existing list.

The following example demonstrates how this is done for the IBM WebSphere and Sun JDBC-ODBC drivers:

```
<init-param>
  <param-name>load-class</param-name>
  <param-value>
    <!-- For IBM WebSphere: -->
    com.ibm.servlet.classloader.Handler

    <!-- For JDBC-ODBC Bridge: -->
    sun.jdbc.odbc.JdbcOdbcDriver
  </param-value>
</init-param>
```

Notice that several drivers can be added within a single `param-value` element. You can simply place new drivers on a separate line within this element. Each driver should be identified by its fully specified class name. Consult the JDBC driver documentation (see [Resources](#)) to determine the name of this class.

Checking the installation

At this point, you can check that the driver has been properly installed and loaded.

Restart Tomcat to ensure that all classes are properly reloaded. Once Tomcat and Cocoon 2 have restarted, check the `$(COCOON_HOME)/WEB-INF/logs/access.log` file for an entry similar to the following:

```
(Unknown-URI) Unknown-thread/CocoonServlet:  
Trying to load class: org.hsqldb.jdbcDriver
```

This indicates that the server is attempting to load the driver. If a `java.lang.ClassNotFoundException` has been logged after this message, then the driver is not correctly installed. Return to [Installing a JDBC driver](#) and ensure that the driver is available from the CLASSPATH. Try an alternate installation method if necessary.

Once the JDBC driver has been correctly installed and loaded, your final configuration step is to define a connection pool.

Configuring a connection pool

Connection pools are configured in the main Cocoon 2 configuration file, `$(COCOON_HOME)/WEB-INF/cocoon.xconf`. Pool configuration options are contained in the `datasources` element. The syntax for this element is shown in the following pseudo-code:

```
<datasources>  
  ...  
  <jdbc name="pool-name">  
    <pool-controller min="1" max="5"/>  
    <auto-commit>true|false</auto-commit>  
    <dburl>JDBC-connection-string</dburl>  
    <user>database-username</user>  
    <password>database-password</password>  
  </jdbc>  
  ...  
</datasources>
```

These elements have the following meanings:

- `jdbc` -- Defines a JDBC connection pool. Cocoon 2 can manage a number of pools simultaneously. Each pool must have a unique `name`. This name is used to reference the pool from application code.
- `pool-controller` -- Controls pool management options. When Cocoon 2 initializes the pool, it immediately creates a number of connections according to the value of the `min` attribute. The `max` attribute governs the maximum possible size of the pool. When creating pools for Oracle databases, add an optional attribute, `oradb`, with a value of `true` to

ensure that connections terminated by the database are properly identified.

- `auto-commit` -- Is optional and indicates whether connections should automatically commit. The default value is `true`. If set to `false`, the application is required to perform its own commits. The recommended value for this option is default unless complete transaction control is desired.
- `dburl` -- Is the JDBC connect string to be supplied to the `DriverManager.getDriver` method (see [Resources](#)).
- `user` and `password` -- Are the username and password, respectively, entered to access the database.

Once the connection pool has been configured, you must restart Cocoon 2 to re-read the configuration file. Unlike the sitemap, Cocoon 2 only reads the `cocoon.xconf` file when it starts. Simply shut down and restart Tomcat to have the changes take effect. At this point Cocoon 2 should have successfully created a pool of connections to the application database.

The next section introduces the ESQL logicsheet, which provides the database access layer in Cocoon 2. Later sections contain detailed examples of working with the logicsheet.

Section 3. The ESQL logicsheet

ESQL logicsheet overview

As described in the previous tutorial in this series, "[Working with XML Server Pages](#)," logicsheets are a mechanism for creating custom tags for use in XSP pages. You implement these logicsheets using XSLT transforms. A code generator uses the XSLT transforms to replace the tags with Java code that implements the desired functionality.

The ESQL logicsheet is a thin layer above the standard JDBC API, defining a number of tags that map to particular JDBC functionality. The logicsheet is therefore a simple way to generate JDBC code for an application.

This section reviews the basic structural elements that make up the ESQL logicsheet. The sections that follow describe how these elements can be used to retrieve and manipulate database data.

The core ESQL elements, part 1

The example XML document below illustrates the basic structure of an XSP page that uses the ESQL logicsheet. Notice that the ESQL namespace, `http://apache.org/cocoon/SQL/v2`, has been declared on the `xsp:page` element:

```
<xsp:page
  language="java"
  xmlns:xsp="http://apache.org/xsp"
  xmlns:esql="http://apache.org/cocoon/SQL/v2">

<root>
  <esql:connection>

    <esql:execute-query>

      <!-- connection
      information -->
      <esql:pool/>

      <!-- SQL query -->
      <esql:query/>

      <!-- result processing
      elements -->
      <esql:results/>
      <esql:update-results/>
      <esql:no-results/>
      <esql:error-results/>

    </esql:execute-query>

  </esql:connection>

</root>

</xsp:page>
```

You must use a number of common structural elements when working with the ESQL logicsheet. Each of these elements derives its functionality from an equivalent object in the JDBC API. The presence of these elements is used to drive the generation of specific sections of JDBC code in the compiled form of the XSP page.

The `esql:connection` is equivalent to a JDBC `Connection` (see [Resources](#)). Just as all JDBC operations ultimately derive from a particular database connection, each of the ESQL elements must be properly nested within an `esql:connection` element. It is possible to have multiple connection elements within a single XSP page, allowing a single page to interact with multiple data sources. The connection element must also contain other elements that define how a database connection is to be created (for example, `esql:pool`). This is reviewed in [Defining the connection](#).

The `esql:execute-query` element is equivalent to the JDBC

PreparedStatement object (see [Resources](#)). It defines how to execute individual queries within a given database connection as well as how the results of those queries should be processed. It is acceptable to nest individual `esql:execute-query` elements to create nested database queries. See [Nesting and grouping query results](#) for more information.

The core ESQL elements, part 2

The `esql:query` element contains the SQL query (SELECT, INSERT, UPDATE, or DELETE) to be carried out using this connection:

```
<xsp:page
  language="java"
  xmlns:xsp="http://apache.org/xsp"
  xmlns:esql="http://apache.org/cocoon/SQL/v2">

<root>
  <esql:connection>

    <!-- connection
    information -->
    <esql:pool/>

    <esql:execute-query>

      <!-- SQL query -->
      <esql:query/>

      <!-- result processing
      elements -->
      <esql:results/>
      <esql:update-results/>
      <esql:no-results/>
      <esql:error-results/>

    </esql:execute-query>

  </esql:connection>

</root>

</xsp:page>
```

The processing of query results is dependent on the type of query being carried out. The processing of SELECT statements is defined within the `esql:results` element, while the results of INSERTs, UPDATEs, and DELETEs are processed within the `esql:update-results` element. Be careful to use the correct element, as Cocoon 2 will not generate an error if the wrong element is used.

Both the `esql:results` and `esql:update-results` elements usually contain other ESQL tags that provide fine-grained access to the query results. These tags, which have their equivalents in the `ResultSet` and `ResultSetMetaData` objects (see [Resources](#)), are reviewed in the sections that follow.

The `esql:no-results` and `esql:error-results` elements contain tags that describe the processing that should be carried out if a query generates no results, or an `SQLException` (see [Resources](#)), respectively.

In all cases, you can mix the ESQL tags with user-defined tags that are directly output by the XSP page. You can also use XSP tags, and those from other logicsheets, to define additional processing as required. The only real restriction is to ensure that the XSP page remains well-formed.

Defining the connection

You can define how the connection corresponding to an `esql:connection` element is to be created in two ways.

The first is to simply refer to a previously defined connection pool that has been configured with the instructions provided in [Configuring database connections](#) . This is achieved using the `esql:pool` element:

```
<esql:connection>
  <esql:pool>myPoolName</esql:pool>
  <esql:execute-query>
    ...
  </esql:execute-query>
</esql:connection>
```

The content of the element should correspond to the name of a JDBC datasource defined in `cocoon.xconf`.

The second is to define the connection parameters directly within the `esql:connection` element. This is done using the following elements:

- `esql:dburl` -- the JDBC connection string
- `esql:username` -- the username to connect to the database
- `esql:password` -- the password for the above username
- `esql:driver` -- the JDBC driver
- `esql:autocommit` -- indicates that the JDBC connection should automatically commit

The benefit to the latter approach is that the connection details can be dynamically generated -- for example, retrieving the username and password from the current session. While this offers a great degree of flexibility, it removes the performance benefits of letting Cocoon manage the connections using a database pool.

Therefore, unless this dynamic behavior is required, it is recommended that connections be defined using the `esql:pool` element.

Section 4. Performing queries

Queries overview

This section demonstrates how to use the basic ESQL elements introduced in the previous section to generate simple XML documents from database queries. A number of additional elements are also introduced that allow access to results at the row and column level.

The functionality is illustrated using a number of simple examples that demonstrate the flexibility of the ESQL logic sheet.

Simple query processing

Taking the basic XSP structure described in the previous section, it's possible to create a simple XSP page that will query the category table in the sample database and generate a simple, sorted list of music categories. You can access the full source code for this example from [Download](#).

The SQL query is defined as the content of the `esql:query` element, as follows:

```
<esql:query>
  select cat_id, name from category
  order by cat_id
</esql:query>
```

The definition of how to process each row in the results of this query is achieved using the `esql:row-results` element, which is a child of `esql:results`. The content of this element is translated into code that is executed for every row in the `JDBC ResultSet` that's generated by the query. For example, this element describes *row level* processing:

```
<esql:results>
  <esql:row-results>
    <category>
      <esql:get-columns/>
    </category>
  </esql:row-results>
```

```
</esql:results>
```

The above example declares that a `category` element is generated for every row in the results, and indicates that the logicsheet should automatically map the row data into XML elements. Each column becomes a separate XML element whose content is the value of that column for the current row. Therefore, executing this XSP page (for example, by visiting the URL `http://localhost:8080/dbc2/categories.xml`) produces the following output:

```
<categories>
  <category>
    <CAT_ID>1</CAT_ID><NAME>Blues</NAME>
  </category>
  ...more results...
</categories>
```

Notice that the element names exactly match an uppercase version of the column names defined in the query. While this provides a very quick means for producing XML from a query, the resulting names may not be desirable, especially if a database naming convention means that the generated XML will be difficult to read. It's possible to use simple SQL aliasing to alter the names of individual elements (e.g. using `select cat_id as id, name from ...`). If you prefer a lower-case version, add a `tag-case` attribute to `esql:get-columns`. This attribute can have the value `lower` or `upper`, indicating the preferred case for the automatically generated tag names.

A more flexible alternative takes advantage of some additional ESQL elements that allow direct access to the values for individual columns in the results.

Processing individual rows, part 1

Where greater control over the results processing is required, the ESQL logicsheet provides a number of *getter* elements that provide type-specific access to individual columns in the results from a query. These getter methods are directly equivalent to the numerous `get` methods on the `java.sql.ResultSet` object (see [Resources](#)).

Continuing the example from [Simple query processing](#), the following code demonstrates how to insert data from the result into specific named elements (see the complete example file, `categories2.xml`, available from [Download](#)).

```
<esql:results>
  <esql:row-results>
    <category>
      <id><esql:get-int column="cat_id"/></id>
      <name><esql:get-string column="name"/></name>
```

```

    </category>
  </esql:row-results>
</esql:results>

```

Notice that the numeric `cat_id` column is referenced using the `esql:get-int` element, while the `name` column is accessed using `esql:get-string`. There are additional variations of this element for gaining access to boolean, date, long and other columns.

You can mix these ESQL tags with other XSP tags, allowing the freedom to map the results to XML in many different ways. For example, the following demonstrates using the `xsp:attribute` element to create attributes rather than elements in the results of the query (see `categories3.xml`):

```

<esql:results>
  <esql:row-results>
    <category>
      <xsp:attribute name="id">
        <esql:get-string column="cat_id"/>
      </xsp:attribute>
      <xsp:attribute name="name">
        <esql:get-string column="name"/>
      </xsp:attribute>
    </category>
  </esql:row-results>
</esql:results>

```

results in data in the form:

```

<categories>
  <category id="1" name="Blues"/>
  <category id="2" name="Classical"/>
  ...more results...
</categories>

```

Processing individual rows, part 2

While it is generally safe to use the `get-string` element regardless of the underlying database type, type-specific variants ensure that the data is already cast to the appropriate Java type or object. This is useful when the results of a query might be processed in other ways, such as using custom code. A trivial example of this uses Java code to assign an additional attribute to the results, depending on whether the category identifier is even or odd (see `categories4.xml`, available from [Download](#)):

```

<esql:results>
  <esql:row-results>
    <category>
      ...xsp:attribute elements as before...

```

```
<xsp:logic>
  if (<esql:get-int column="cat_id"/> % 2 == 0)
  {
    <xsp:attribute name="rowtype">even</xsp:attribute>
  }
  else
  {
    <xsp:attribute name="rowtype">odd</xsp:attribute>
  }
</xsp:logic>
</category>
</esql:row-results>
</esql:results>
```

This helps illustrate that the ESQL elements simply provide a shorthand for calling methods on the `ResultSet` object (see [Resources](#)) and can be freely mixed with Java code within `xsp:logic` elements.

Other data manipulation elements

The ESQL logicsheet includes a number of other result- and row-level elements that provide further support for custom processing of results. Again, the majority of these elements have an equivalent method in the `ResultSet` or `ResultSetMetadata` objects.

These elements include:

- `esql:get-column-count` -- Returns the number of columns in the results. Use this element to set up loops where columns need to be processed in sequence.
- `esql:get-metadata` -- Returns a reference to the `ResultSetMetadata`.
- `esql:get-resultset` -- Returns a reference to the JDBC `ResultSet` for direct manipulation.
- `esql:get-row-position` -- Gets the position of the current row in the results. Use this element to provide row numbers for tables and so forth.
- `esql:get-column-name` -- Gets the name of the current column, which must be referenced by its position. Use this element when sequentially processing the columns in a result set.
- `esql:is-null` -- Tests whether a given column has a NULL value.
- `esql:get-xml` -- Adds to the functionality provided by a JDBC `ResultSet`. The value of the named column is retrieved and parsed as XML. The value of the column can be optionally wrapped in another element (specified by a `root` attribute) before parsing. Use this element

for occasions where raw XML data is stored in a database.

Refer to the ESQL logicsheet documentation (see [Resources](#)) for information on the complete set of available elements.

Handling empty results and errors

The previous examples assumed that a query always successfully completes and returns results. Obviously, this is not always the case: Some queries may return no data and database errors can occur (for example, due to connection failures). The ESQL logicsheet provides tags that allow you to handle these conditions.

If a query returns no results, you can take action by using the `esql:no-rows` element. `no-categories.xml` (part of the example code that can be accessed from [Download](#)) contains a query that will not generate results because it attempts to select a nonexistent identifier in the `category` table. The relevant portions of this example are shown below:

```
...
<esql:results><results/></esql:results>
<esql:no-results><no-results/></esql:no-results>
...
```

The output of this example should show that the `no-results` element is generated.

The `esql:error-results` element provides a hook for processing `SQLException` generated during the processing of query results. Accessing the stack trace and message associated with the exception is demonstrated in `error.xsp` (see [Download](#)), which simulates a database error by attempting to extract data from a nonexistent column. The relevant fragment of this example is shown here:

```
...
<esql:error-results>
  <error>
    <message><esql:get-message/></message>
    <trace><esql:get-stacktrace/></trace>
    <string><esql:to-string/></string>
  </error>
</esql:error-results>
...
```

It's important to differentiate exception conditions that occur when results are being processed from those that occur when a database connection is being created or the query contains syntax errors. These more serious errors are not passed to the XSP page and are handled by Cocoon 2 itself.

Passing parameters to queries

Passing parameters from an HTTP request through to a database query is a desirable feature. Implementing it is straightforward in XSP using a combination of the ESQL and XSP-Request logic sheets.

For example, using the sample database, it would be useful to be able to extract a list of all albums in a given category. Here, a simple select of all columns from the album table needs to be qualified by a particular `cat_id`. The following is extracted from `albums-in-category.xsp` (see [Download](#)), which contains the complete example:

```
<esql:query>
select * from album
where cat_id = <xsp-request:get-parameter name="id"/>
</esql:query>
```

Invoke the URL

`http://localhost:8080/dbc2/albums-in-category.xml?id=1` to return a list of all albums in the "Blues" category. In this example, if there are no albums in a category, then an empty list is returned; however, a non-existent category causes a `no-such-category` element to be generated.

Section 5. Nesting and grouping query results

Overview

```
<categories>
  <category id="1">
    <name>...</name>
    <albums>
      <album>
        <id>1</id>
        <title>...</title>
        <artist>...</artist>
        <tracks>...</tracks>
      </album>
      <album>
        <id>2</id>
        <title>...</title>
        <artist>...</artist>
        <tracks>...</tracks>
      </album>
    </albums>
  </category>
</categories>
```

```
</category>
<category id="2">
  <name>...</name>
  <albums/>
</category>
</categories>
```

Mapping relational data to the hierarchical model of an XML document often requires the ability to perform nested queries on the relational data. This allows records from one table (the *master*) to be mapped into XML elements containing other child elements that have been generated from related data in other tables (the *detail*).

One possible mapping of the sample database into an XML document is shown on the left. Note that each `category` element may contain a number of nested `album` elements. There are two ways to achieve this in ESQL. The first of these involves using a nested query and the second an ESQL feature that allows automated grouping of data generated from a query.

Both methods are demonstrated in this section.

Nested queries

```
<esql:connection>
  <esql:execute-query>
    <esql:query/>
    <esql:results>
      <!-- results from query
      one -->
      <esql:execute-query>
        <esql:query/>
        <esql:results>
          <!-- results from query
          two -->
        </esql:results>
      </esql:execute-query>
    </esql:results>
  </esql:execute-query>
</esql:connection>
```

An `esql:connection` element may contain multiple, nested `esql:execute-query` elements, allowing query results to be nested inside one another, as shown in the code fragment on the left.

These nested queries do not have to interact with one another, or they can take their parameters from the HTTP request, so no additional syntax is required. However, in some circumstances, the inner query is dependent on the outer query. In the preceding mapping example, the outer query simply has to generate a list of categories while the inner query must only return the albums for a particular category.

See the `categories_and_albums.xsp` file, available from [Download](#), for details

on how these kinds of nested queries work in Cocoon 2. The first query is straightforward: It just selects both columns from the category table. However, the inner query introduces a new piece of ESQL syntax:

```
<esql:query>
  select alb_id as id, title, artist,
  num_tracks as tracks
  from album
  where cat_id = <esql:get-int column="cat_id" ancestor="1"/>
</esql:query>
```

Notice that in the WHERE clause for the query, the `esql:get-int` element is used to retrieve the value of the current category in the outer query. The new `ancestor` element indicates from which outer query the value is being retrieved -- such as the results associated with the first `esql:execute-query` ancestor.

This is a powerful technique, as you can nest any number of queries inside one another, and extract data from as far up the nested query structure as necessary. The technique is similar to that used in conventional report writing tools, where a single master query provides context for a number of other queries that are used to build up the report results.

Grouping data, part 1

In an alternative method for achieving the same results, you can retrieve all the required data in a single query and then use the ESQL grouping elements to identify how you want the data broken down into nested sections.

A simple join of the category and album tables might generate the following results (not all columns are shown):

cat_id	name	alb_id	title
1	Blues	1	The Healer
1	Blues	2	Mr Lucky
2	Classical	3	The Four Seasons

Notice that the first two rows have albums that are in the same category and so have the same values for the `cat_id` column. These patterns in the results of table joins can be exploited by the `esql:group` and `esql:member` elements to collate together related records. In this example, albums that share a common `cat_id` are listed together.

`Categories_and_albums2.xsp` (available from [Download](#)) demonstrates the required syntax, an extract of which is included here:

```
<esql:row-results>
  <esql:group group-on="cat_id">
    <category>
      <!-- process category data -->
      <albums>
        <esql:member>
          <album>
            <!-- process album data -->
            <id><esql:get-string column="alb_id"/></id>
          </album>
        </esql:member>
      </albums>
    </category>
  </esql:group>
</esql:row-results>
```

Grouping data, part 2

The `esql:group` element, from the [Grouping data, part 1](#) example, has a `group-on` attribute. This attribute identifies the column in the results that can be used to distinguish the outer elements (in other words, the categories). The contents of the element define processing that will be applied to each *distinct* category, as opposed to each row of the results.

The `esql:member` element defines the processing for each row that shares the common column defined by the `group-on` attribute. Here the element contains the processing required to structure the XML for each album, with the final XML structure being identical to that generated by the nested query version. You can see the two sets of results at

http://localhost:8080/dbc2/categories_and_albums.xml and
http://localhost:8080/dbc2/categories_and_albums2.xml.

Understanding how to generate XML data from a database query is only one aspect of the ESQL functionality. It is also possible to use the logicsheet to allow data to be inserted, updated, and deleted. This capability is briefly reviewed in the next section.

Section 6. Modifying data

Inserts, updates, deletes

Insert, *updated*, and *delete* operations on database data all share a common feature: They update the current state of the database by adding, removing, or altering records. This is distinct from a *select*, which simply returns the current

database state. ESQL supports the full range of these operations, all of which can be entered in the `esql:query` element. However, as noted earlier (see [The ESQL logicsheet](#)), the results of these operations are reported differently.

Rather than using the `esql:results` element described in the previous sections and examples, you must use the `esql:update-results` element to process the results of the three update operations. Also, unlike the wide range of elements available for processing query results, updates only return a count of the number of elements affected by the operation. This can be accessed using the `esql:get-update-count` element.

The sample application (available from [Download](#)) includes an example for each of the update operations:

- `insert-album.xsp` -- Inserts a new album record
- `update-album.xsp` -- Updates an existing album record
- `delete-album.xsp` -- Deletes an existing album record

In each case the parameters for the update operation are taken from the HTTP request. The number of affected records is reported in either an `inserted`, `updated`, or `deleted` record as illustrated in the example in the next panel.

Update operation example

In the sample application, you can request the URL

`http://localhost:8080/dbc2/delete-album?id=1` to delete the album whose `cat_id` is 1. The results are:

```
<albums>
  <deleted>1</deleted>
</albums>
```

If you request the URL a second time, then the deleted count is zero. This allows the results of an operation to be identified by the Cocoon 2 pipeline. You cannot use the `esql:no-results` nor `esql:error-results` elements in conjunction with these types of queries.

As an aside, you should use care when designing these kinds of interfaces. For demonstration purposes, it's easier to demonstrate modifying database data using simple GET requests, but real applications should always use HTTP POST requests. The HTTP specification clearly describes that GET requests should not have these kind of side effects. You can easily identify the request method using the `xsp-request:get-method` element, as follows:

```
<xsp:logic>
if (<xsp-request:get-method/>.equals("POST"))
{
//perform insert or update
}
if (<xsp-request:get-method/>.equals("GET"))
{
//perform query
}
...etc
</xsp:logic>
```

The other important consideration is to check the data submitted by the user to ensure that only valid data is inserted. The next section introduces the form validation capabilities of Cocoon 2.

Section 7. Form validation

Form validation overview

Form validation is simply the process of taking user-submitted data, usually entered into an HTML form, and validating it against the business rules of an application. At its simplest, this validation involves testing the types and lengths of fields to ensure that they conform to the database constraints. To generate immediate feedback for the user, perform these tests as soon as forms are submitted.

Cocoon 2 provides a component, the Form Validator Action, which is capable of carrying out a number of different validation tests. The action is configured to carry out specific tests on request parameters by creating a separate XML description of the form fields. This description is used as a parameter by the action, which can apply the appropriate validation rules and then signal to the sitemap whether the processing was successful.

This section describes how to:

- Create a form description
- Invoke the validation
- Access the results of the validation using the Form Validator logicsheet

Describing a form

The example below demonstrates the basic syntax of the XML format supported by the Form Validator Action for describing form fields (see also `insert-album.xml` in the example code available from [Download](#)). It describes the validation rules for the data submitted by a user who enters a new album into the sample database.

```
<root>
  <!-- field definitions -->
  <parameter name="id" type="long"
    min="1" max="99999" nullable="no"/>
  <parameter name="cat" type="long"
    min="1" max="999" nullable="no"/>
  <parameter name="title" type="string"
    max-len="100" nullable="no"/>
  <parameter name="artist" type="string"
    max-len="100" nullable="no"/>
  <parameter name="tracks" type="long"
    max="99" nullable="no"/>

  <constraint-set
    name="insert-album">
    <validate name="id"/>
    <validate name="cat"/>
    <validate name="title"/>
    <validate name="artist"/>
    <validate name="tracks"/>
  </constraint-set>
</root>
```

Note these things about the example. First, the root element of the document is ignored and so can have any name. Second, the document is divided up into two sections: a number of field definitions followed by a *constraint set*.

Each of the field definitions describe a user-submitted parameter. The parameters must all have unique `name` attributes. Fields must also have a `type`, which can be one of the following: `long`, `double`, or `string`. Each field can then define a number of validation rules that are specified as additional attributes:

- `nullable` -- Identifies whether the field can be null
- `default` -- Identifies a default value for the field, if it is not supplied
- `min` and `max` -- Indicates minimum and maximum values
- `min-len` and `max-len` -- Indicates minimum and maximum lengths
- `matches-regex` -- Defines a POSIX regular expression that must correctly match the parameter value; this allows for much richer content validation, such as e-mail address formats

The constraint sets describe a combination of the defined fields that will be validated in one pass, as described next. The `validate` elements in a constraint set support two additional attributes:

- `equals-to` -- Defines a fixed string that the parameter must match
- `equals-to-param` -- Defines the name of another parameter that should have the same value, such as checking whether a user has entered the same password twice on a form that verifies a new password

Validating fields

Now that you've defined the validation rules, the next step is to apply them to a given form submission. Like any other Cocoon 2 component, you must first declare the Form Validator Action in the sitemap before it can be used:

```
<map:components>
...
<map:actions>
<map:action name="form-validator"
src="org.apache.cocoon.acting.FormValidatorAction"/>
</map:actions>
...
</map:components>
```

The action is then added to a Cocoon 2 pipeline as follows:

```
<map:match pattern="form/insert-album">
<map:act type="form-validator">
<map:parameter name="descriptor"
value="context://insert-album.xml"/>
<map:parameter name="validate-set"
value="insert-album"/>
<!-- if success -->
...
</map:act>
<!-- if fail -->
...
</map:match>
```

The above pipeline is invoked when posting the "insert-album" form. Its first step is to invoke the Form Validator Action and pass two parameters to it. The first parameter, `descriptor`, defines the XML file describing the validation rules. The `context://` URL scheme refers to the file system below the `$COCOON_HOME` directory structure. The second parameter, `validate-set`, defines a constraint set defined within that file. All HTTP request parameters that match a parameter name in that constraint set are validated using the configured rules.

If validation is successful, then the other pipeline components within the `map:act` element are processed. This is where the `insert-album.xsp` (see [Download](#)) XSP page would be invoked to insert the data that is known to be valid.

If validation is unsuccessful, only the components following the `map:act` element

are executed. In this way, the behavior of Actions is similar to "if" statements: If they successfully carry out their processing, then the block of pipeline components nested inside them is executed; otherwise, that block is skipped.

Using the Form Validator logicsheet, you can provide the user with detailed feedback on why a particular form failed.

Using the Form Validator logicsheet

You can only use the Form Validator logicsheet in conjunction with the Form Validator Action. It is typically used within XSP pages that are executed if a form fails to validate. The action stores a detailed breakdown of its results as an HTTP request parameter. This logicsheet provides a simple tag-based API for interpreting those results.

The namespace for this logicsheet is

`http://apache.org/xsp/form-validator/2.0.`

Each of the following elements from this logicsheet accepts a `name` attribute, which indicates the name of the field whose validation status is being tested:

- `formval:is-ok` -- Returns a boolean indicating whether field validated successfully
- `formval:is-null` -- Indicates the field was null when it shouldn't have been
- `formval:is-toosmall` -- Indicates the field was less than `min-len` in length, or `min` value
- `formval:istoolarge` -- Indicates the field was greater than `max-len` in length, or `max` value
- `formval:is-nomatch` -- Indicates the field failed to match the configured regular expression

You can also use a simple `if` test that corresponds to each of these elements: The `formval:on-ok` tests whether the named field validated correctly, and if so, its contents are evaluated. These two variants allow a great deal of flexibility, as the `on-*` versions can be used for simple tests, while the `if-*` versions can be used within custom code (in other words, in `xsp:logic` elements).

In circumstances where you need to apply several of these tests to the same field, rather than using the `name` attribute repeatedly, you can nest the tests within a `validate` element. This element must have a `name` attribute that provides the context for any nested form validation tags.

Section 8. Summary

Where next?

Dynamic content publishing is an everyday reality on the Web. To meet this demand, any mature server framework must provide functionality for working with information held in relational databases. This tutorial has reviewed the main components in Cocoon 2 that support this functionality.

The initial section of the tutorial contained detailed instructions for configuring data sources in Cocoon 2, allowing the server to manage connection pools on behalf of individual applications. This configuring involves installing and loading a JDBC driver and configuring various resource management options for the connection pool.

The tutorial then progressed to describing the basic elements in the ESQL logicsheet. This logicsheet provides the expected range of functionality required for working with database data, including selecting, inserting, updating, and deleting records. A number of examples based around a sample database illustrated this functionality. In particular, various options for mapping relational data into XML documents were discussed. These included options for element naming, processing query results with custom logic, and creating hierarchical documents using nested queries and data grouping.

Lastly, to ensure data integrity in user-submitted data, the tutorial reviewed the functionality provided by the Form Validator Action and its supporting logicsheet. These components provide a means to declaratively associate validation rules with HTTP request parameters.

Downloads

Description	Name	Size	Download method
Sample code	x-dbc2-tutorial-code.tar	40KB	HTTP

[Information about download methods](#)

Resources

Learn

- Check out the first two Cocoon 2 tutorials in this series: "[Introducing Cocoon 2](#)" (*developerWorks*, March 2002) and "[Working with XML Server Pages](#)" (*developerWorks*, April 2002).
- Take the other tutorials in this [developerWorks series on Cocoon](#).
- Looking for a JDBC driver? Search the Sun [JDBC Driver database](#).
- Get background on building database-backed Web sites using JSP, Java Beans, and JDBC with Robert Brunner's *developerWorks* tutorial, "[Building Web-based applications with JDBC](#)" (*developerWorks*, December 2001).
- Try Brunner's tutorial [Advanced database operations with JDBC](#) for additional training that expands your JDBC knowledge (*developerWorks*, November 2001).
- Read [Enterprise JavaBeans fundamentals](#) for basics on Enterprise JavaBean (EJB) programming and the Java 2 Enterprise Edition environment (*developerWorks*, April 2003).
- ESQL isn't the only way to generate XML from relational data. Nick Chase's [Using JDBC to extract data into XML](#) provides a DOM-based alternative. (*developerWorks*, October 2001) Chase has also written a tutorial on how to [insert data from XML into a database](#) (*developerWorks*, October 2001).
- Investigate the real capabilities of a Web-based CD catalog over at [MusicBrainz](#).
- To register a JDBC driver, as described in [Loading the JDBC driver](#), see [java.sql.DriverManager](#).
- Refer to the [ESQL logicsheet documentation](#) for information on the complete set of available elements.
- For a reference on the `DriverManager.getDriver` method, as described in [Configuring a connection pool](#), see [http://java.sun.com/j2se/1.3/docs/api/java/sql/DriverManager.html#getDriver\(java.lang.String\)](http://java.sun.com/j2se/1.3/docs/api/java/sql/DriverManager.html#getDriver(java.lang.String)).
- Below is a list of JDBC objects and their ESQL logicsheet element equivalents:
 - The JDBC [Connection](#) object = the `esql:equivalent` element (see [The core ESQL elements, part 1](#)).
 - The JDBC [PreparedStatement](#) object = the `esql:execute-query` element (see [The core ESQL elements, part 1](#)).
 - The JDBC [ResultSet](#) object = the `esql:results` element (see [The](#)

[core ESQL elements, part 2](#)).

- The JDBC `ResultSetMetaData` object = the `esql:update-results` element (see [The core ESQL elements, part 2](#))
 - The JDBC `SQLException` object = the `esql:error-results` element (see [The core ESQL elements, part 2](#)).
 - The JDBC `java.sql.ResultSet` object = the `esql:get-int` and the `esql:get-string` elements (see [Processing individual rows, part 1](#)).
- Find out how you can become an [IBM Certified Developer in XML and related technologies](#).
 - Explore many more XML resources on the [developerWorks XML zone](#).

Get products and technologies

- Need an open source database for building Web applications? Look into [MySQL](#), a favorite among developers.
- [IBM WebSphere Studio Application Developer](#) is an easy-to-use, integrated development environment for building, testing, and deploying J2EE™ applications, including generating XML documents from DTDs and schemas.

About the author

Leigh Dodds



Leigh Dodds is team leader of the Research and Technology Group at Ingenta, Ltd. He has five years of experience in developing on the Java platform, and he has spent the last three years working with XML and related technologies. Leigh is also a contributing editor to [xmlhack.com](#), and has been writing the regular "XML-Deviant" column on [XML.com](#) since February 2000. He holds a bachelor's degree in biological science, and a master's in computing. When he's not wrestling with pointy brackets, Leigh can be found making silly noises with his son, Ethan.