XML - Using XML with Databases

http://www.brainbell.com/tutorials/XML/TOC_Using_XML_With_Databases.htm 5/11/10

In this tutorial you'll learn

- The basic theory behind the relational database model
- How to use SQL, the query language for relational databases
- How to export data from a database as XML
- How to write a program that queries a database and formats the results as XML

Using XML with Databases

Although XML is certainly a great storage format for shuttling data around the Web, most web applications store their data in relational databases. Knowing this, it stands to reason that many web applications will end up accessing data from a database yet interacting with it and sharing it with other applications as XML data. This has a lot do with the fact that XML can be used not only as a document format, but also as a way to represent data in a highly structured manner. In this tutorial, you learn how relational databases work and how you can integrate your XML applications with relational databases. More specifically, you find out how to export database data as XML code, as well as how to make SQL queries on a database and format the results in XML. In this tutorial you'll learn

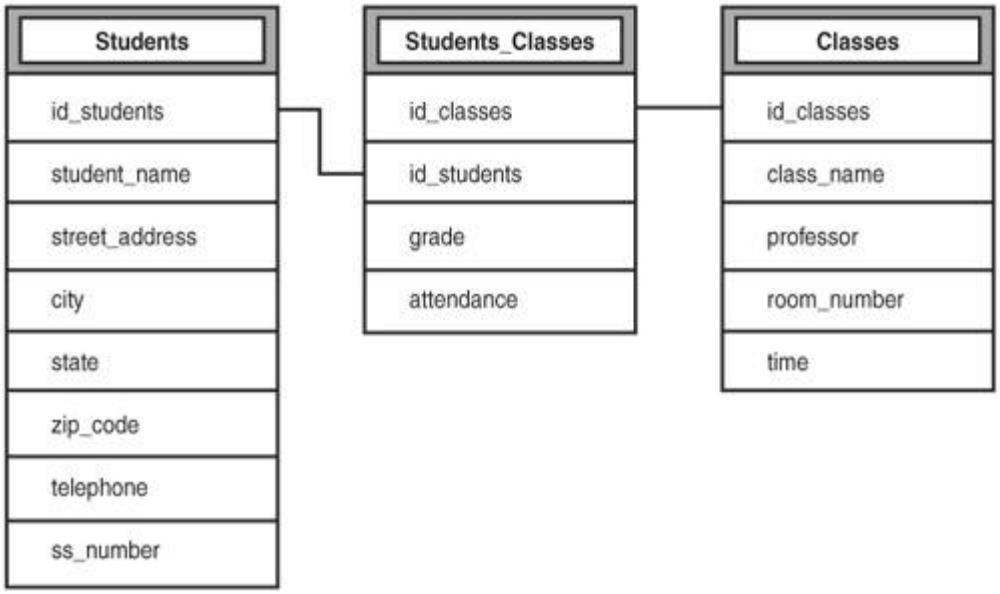
- The basic theory behind the relational database model
- How to use SQL, the query language for relational databases
- How to export data from a database as XML
- How to write a program that queries a database and formats the results as XML

A Quick Relational Database Primer

- Before you can learn about relating XML to databases, you need to learn about databases themselves. When most people think of databases, they're thinking specifically about relational databases. All of the popular database productsMicrosoft SQL Server, Oracle, IBM DB2, MySQLuse the relational model. In turn, most web and business applications use one relational database or another for data storage.
- The relational database model is all about tables. All of the data is stored in a tabular format, and relationships between tables are expressed through data shared among those tables. Tables in relational databases are just like tables in HTML or tables in this book. They consist of rows and columns. Each row represents a record in the database, and each column represents one field in each of the records.

- A group of tables is generally referred to as a schema, which conceptually isn't all that different from an XML schema. In a schema, some or all of the tables are generally related to one another. Let's look at how those relationships work. Ordinarily, every table contains a column (or group of columns) that contains data that uniquely identifies that row in the table. In most cases, this is an ID field that simply contains a number that sets that row apart from the others in the table. This value is referred to as the primary key. In relational database design, the primary key is extremely important because it is the root of relationships between tables.
- Here's a simple example. Let's say you have a table called `students`. The `students` table contains, among other bits of data, a column called `id_students`. The table might also include the student's name, address, and phone number. You might also have a second table, called `majors`. This table contains the major and minor for all of the students, under the assumption that no student has more than one major or minor.
- This is what is referred to as a one-to-one relationship. Each record in the `students` table can have one corresponding row in the `majors` table. There are two other types of relationships between tablesone-to-many and many-to-many. The students table contains a column called `id_students`, which serves as its primary key. The `majors` table should contain a column that contains student IDs. This is referred to as a foreign key, because it's a reference to a primary key in another table. The foreign key is used to implement the one-to-one relationship between the records in the two tables.
- In a one-to-many relationship, a record in one table can have a reference to many records in a second table, but each record in the second table can have a reference to only one record in the first table. Here's an example: Let's say I create a table called `grades`, which contains a column for student IDs as well as columns for class names and the grades themselves. Because a student can take multiple classes, but each grade applies to only one student, the relationship between `students` and `grades` is a one-to-many relationship. In this case, `id_students` in the `grades` table is a foreign key relating to the `students` table.
- An example of a many-to-many relationship is the relationship between students and classes. Each student is usually enrolled in several classes, and each class usually contains multiple students. In a relational database, such a relationship is expressed using what is sometimes referred to as a joining tablea table that exists solely to express the relationship between two pieces of data. The schema contains two tables, `students` and `classes`. You already know about the `students` table; the `classes` table contains information about the classes offeredthe name of the professor, the room where the class is held, and the time at which the class is scheduled.
- By the Way
- Before you can deal with integrating databases and XML, you need to understand both databases and XML. You've been learning about XML for a while now, so consider this a crash course in database theory.
-
- To relate `students` to `classes`, you need a third table, called `classes_students` (or a similarly descriptive name). At a bare minimum, this table must include two columns, `id_students` and `id_classes`, both of which are foreign keys pointing to the `students` and `classes` tables, respectively. These two columns are used to express the many-to-many relationship. In other words, both of the other two tables have one-to-many

relationships with this table. Using this table, each student can be associated with several classes, and each class can be associated with any number of students. It may also contain properties that are specific to the relationship, rather than to either a student or a class specifically. For example, a student's grade or her attendance record for the class might be stored in this table. This table structure is illustrated in Figure 19.1.

- **Figure 19.1. The tables in a many-to-many relationship.**
- [View full size image]
- 

The World's Shortest Guide to SQL

One term you can't go far into databases without encountering is SQL, which stands for Structured Query Language. SQL is the language used to retrieve, add, modify, and delete records in databases. Let's look at each of these features in turn.

By the Way

Incidentally, the pronunciation of SQL is somewhat of a contentious issue. The official party line is that SQL should be pronounced "es queue el." However, many people opt for the more casual and also more efficient pronunciation, "sequel." Count me in the latter camp!

Retrieving Records Using SELECT

Just about everything in SQL is carried out via a query, which is simply the act of communicating with the database according to an established set of SQL commands. The query used to retrieve data from a database is called the `SELECT` statement. It has several parts, not all of which are mandatory. The most basic `SELECT` statement is composed of two partsthe select list and the `FROM` clause. A very simple `SELECT` statement looks like this:

```
SELECT *
FROM students
```

Following are the database records returned as the results of the query:

```
+-------------+-----------------+--------------------+-------+--------------
-+---------+
| id_students | student_name    | city               | state | classification
| tuition |
+-------------+-----------------+--------------------+-------+--------------
-+---------+
|           1 | Franklin Pierce | Hillsborough       | NH    | senior
|    5000 |
|           2 | James Polk      | Mecklenburg County | NC    | freshman
|   11000 |
|           2 | Warren Harding  | Marion             | OH    | junior
|    3500 |
+-------------+-----------------+--------------------+-------+--------------
-+---------+
```

In this case, the `*` is the select list. The select list indicates which database columns should be included in the query results. When a `*` is supplied, it indicates that all of the columns in the table or tables listed in the `FROM` clause should be included in the query results.

The `FROM` clause contains the list of tables from which the data will be retrieved. In this case, the data is retrieved from just one table, `students`. I'll explain how to retrieve data from multiple tables in a bit.

Let's go back to the select list. If you use a select list that isn't simply `*`, you include a list of column names separated by commas. You can also rename columns in the query results (useful in certain situations), using the `AS` keyword, as follows:

```
SELECT id_students AS id, student_name, state
FROM students
```

As the results show, only the student name and state columns are returned for the records:

```
+------+-----------------+-------+
| id   | student_name    | state |
+------+-----------------+-------+
|    1 | Franklin Pierce | NH    |
|    2 | James Polk      | NC    |
|    2 | Warren Harding  | OH    |
+------+-----------------+-------+
```

The `id_students` column is renamed id in the query results using the reserved word 'AS'. The other keyword you'll often use in a select statement is `DISTINCT`. When you include `DISTINCT` at the beginning of a select statement, it indicates that no duplicates should be included in the query results. Here's a sample query:

```
SELECT DISTINCT city
FROM students
```

And here are the results:

```
+--------------------+
| city               |
+--------------------+
| Hillsborough       |
| Mecklenburg County |
| Marion             |
+--------------------+
```

Without `DISTINCT`, this query would return the `city` of every student in the `students` table. In this case, it returns only the distinct values in the table, regardless of how many of each of them there are. In this case, there are only three records in the table and each of them has a unique city, so the result set is the same as it would be if `DISTINCT` were left off.

The `WHERE` Clause

Both of the previous queries simply return all of the records in the `students` table. Often, you'll want to constrain the resultset so that it returns only those records you're actually interested in. The `WHERE` clause is used to specify which records in a table should be included in the results of a query. Here's an example:

```
SELECT student_name
FROM students
WHERE id_students = 1
```

Only the record with the matching ID is returned in the results:

```
+-----------------+
| student_name    |
+-----------------+
| Franklin Pierce |
+-----------------+
```

When you use the `WHERE` clause, you must include an expression that filters the query results. In this case, the expression is very simple. Given that `id_students` is the primary key for this table, this query is sure to return only one row. You can use other comparison operators as well,

like the > or != operators. It's also possible to use Boolean operators to create compound expressions. For example, you can retrieve all of the students who pay more than $10,000 per year in tuition and who are classified as freshmen using the following query:

```
SELECT student_name
FROM students
WHERE tuition > 10000
AND classification = 'freshman'
```

Following are the results of this query:

```
+--------------+
| student_name |
+--------------+
| James Polk   |
+--------------+
```

There are also several other functions you can use in the `WHERE` clause that enable you to write more powerful queries. The `LIKE` function allows you to search for fields containing a particular string using a regular expression like syntax. The `BETWEEN` function allows you to search for values between the two you specify, and `IN` allows you to test whether a value is a member of a set you specify.

By the Way

Because the goal in this tutorial is ultimately to learn how to use XML with databases, I won't go into any more detail on these query functions, but feel free to do some additional SQL learning online at http://www.w3schools.com/sql/default.asp, or pick up a book on SQL. Fortunately, you don't have to be a SQL guru to get the benefits of this lesson.

Inserting Records

The `INSERT` statement is used to insert records into a table. The syntax is simple, especially if you plan on populating every column in a table. To insert a record into `majors`, use the following statement:

```
INSERT INTO majors
VALUES (115, 50, 'Math', 'English')
```

The values in the list correspond to the `id_majors`, `id_students`, `major`, and `minor` columns respectively. If you only want to specify values for a subset of the columns in the table, you must specify the names of the columns as well, as in the following:

```
INSERT INTO students
(id_students, student_name)
VALUES (50, 'Milton James')
```

When you create tables, you can specify whether values are required in certain fields, and you can also specify default values for fields. For example, the `classification` column might default to `freshman` because most new student records being inserted will be for newly enrolled students, who are classified as freshmen.

Updating Records

When you want to modify one or more records in a table, the `UPDATE` statement is used. Here's an example:

```
UPDATE students
SET classification = 'senior'
```

The previous SQL statement will work, but I bet you can figure out what's wrong with it. Nowhere is it specified which records to update. If you don't tell it which records to update, it just assumes that you want to update all of the records in the table, thus the previous query would turn all of the students into seniors. That's probably not what you have in mind. Fortunately, the `UPDATE` statement supports the `WHERE` clause, just like the `SELECT` statement.

```
UPDATE students
SET classification = 'senior'
WHERE id_students = 1
```

That's more like it. This statement updates the classification of only one student. You can also update multiple columns with one query, as in the following:

```
UPDATE students
SET classification = 'freshman', tuition = 7500
WHERE id_students = 5
```

As you can see from the example, you can supply a list of fields to update with your `UPDATE` statement, and they will all be updated by the same query.

Deleting Records

The last SQL statement I'll discuss is the `DELETE` statement, which is similar to the `UPDATE` statement. It accepts a `FROM` clause, and optionally a `WHERE` clause. If you leave out the `WHERE` clause, it deletes all the records in the table. Here's an example:

```
DELETE FROM students
```

```
WHERE id_students = 1
```

You now know just enough about SQL to get into trouble! Actually, your newfound SQL knowledge will come in handy a bit later in the lesson when you develop an application that carefully extracts data from a database and encodes it in XML. But first, you find out how to export an entire database table as XML.

Databases and XML

When you integrate XML with databases, the first question that you must look at is how you're using XML in your application. There are two broad categories of XML applicationsthose that use XML for data storage, and those that use XML as a document format. The approach for database integration depends on which category your application falls into.

Although XML is commonly thought of as a document format, it's also very popular as a format for data storage. Many applications use XML files to store their configuration, as well as relying on remote procedure calling services like XML-RPC and SOAP to format the messages that they exchange using XML.

The fact that XML is highly structured and can be tested to ensure that it's both well-formed and valid in a standardized, programatic fashion takes a lot of the burden of reading and modifying the data file off of the application developer when he or she is writing a program.

Let's look at a couple of real world examples where XML might need to be integrated with a relational database. The structured nature of XML makes it a good choice to use as a data interchange format. Let's say that a company periodically receives inventory information from a supplier. That information might be stored in an Oracle database on a server in the supplier's system but might need to be imported into an Access database when the company receives it. XML would make a good intermediate format for the data because it's easy to write programs that import and export the data and because, by using XML, the data can be used in future applications that require it as well.

Another example might be a service that syndicates news articles. The news articles could be distributed via XML files so that they could easily be transformed for presentation on the Web, or they could be imported into a relational database and published from there.

By the Way

Incidentally, there already exists an XML language for storing news articles in XML documents. You learn a great deal more about XML and how it can be used to code news articles in Tutorial 24, "Syndicating the Web with RSS News Feeds."

Resolving XML Data into Database Tables

The question you face when you integrate applications that use XML for data storage with relational databases is the degree to which you want to take advantage of the features of the relational database. If you simply insert entire XML documents into the database, you can't use advanced SQL features to retrieve specific bits of information from the XML documents.

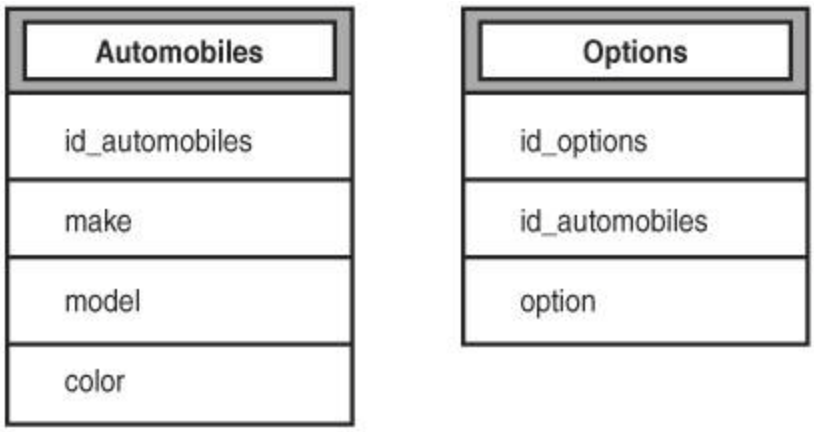
Here's an XML document that is used to store information related to automobiles:

```
<dealership>
  <automobile make="Buick" model="Century" color="blue">
    <options>
      <option>cruise control</option>
      <option>CD player</option>
    </options>
  </automobile>
  <automobile make="Ford" model="Thunderbird" color="red">
    <options>
      <option>convertible</option>
      <option>leather interior</option>
      <option>heated seats</option>
    </options>
  </automobile>
</dealership>
```

Now, let's look at how you might design a database to store this information. As I mentioned earlier, the path of least resistance is just to stick the whole XML document in a field. However, that probably isn't a good idea for this file because it contains more than one automobile "record."

Instead, let's look at what a database schema for the information in the XML file would look like. A diagram of the schema appears in Figure 19.2.

Figure 19.2. The schema that corresponds to the automobiles example XML document.



As you can see, I turned the XML document into two tables, `automobiles` and `options`. The `automobiles` table contains all the information stored in the attributes of the `automobile` tag in the XML document. Because automobiles have a one-to-many relationship to options, I created a separate table for them. In the options table, `id_automobiles` is a foreign key that relates back to a specific automobile in the automobiles table.

To make sure you understand why the automobile options were broken out into a separate database table, consider that the number of options for a single automobile can vary from one automobile to the next. This is a scenario where a single database field in the `automobiles` table can't account for a varying amount of data; hence the one-to-many relationship. Therefore, the solution is to break out the options into a separate table where each row is tied back to a specific automobile. Then you can add as many options as you want for one automobile as long as each option includes the appropriate automobile ID.

Storing XML Documents in a Database

If you're storing entire XML documents in a database, you don't need to worry about translating the XML document format into a tabular database structure. Instead, you just need to extract the information from the document that you need to use in the relational database world and create columns for that. As an example, if you store newspaper articles as XML documents, the section, headline, author, body, and perhaps more information will all be included in the XML document within their own tags. It is then possible to process the XML code to access each portion of the document.

If you store those documents in a database and plan on publishing them on the Web from that database, you may want to consider breaking up the XML data so that it can be retrieved more easily. For example, you might want separate columns for the section and writer so that you can write simple SQL statements that retrieve the documents based on those values. Either way, you would be retrieving XML code from the database, which is far different than the earlier automobile example where the database data has been translated from XML into pure database content.

Exporting an XML Document from a Database

If you need to pull data from a database for processing as XML on a one-time basis, or maybe periodically but not necessarily in real-time, you might consider just exporting the data manually. Most databases offer an "export as XML" option that converts a database table into a structured XML document with the database columns turned into XML tags. This is a very simple approach to quickly generating an XML document from a database that you might now otherwise be able to access without database tools.

I regularly use the MySQL database for online projects. MySQL is a very popular open source database that does a great job for small- to medium-scale applications. A nice front-end is available for MySQL called phpMyAdmin, which provides a web-based user interface for interacting with a MySQL database. phpMyAdmin provides a very easy-to-use export feature that will export any MySQL database as an XML document.

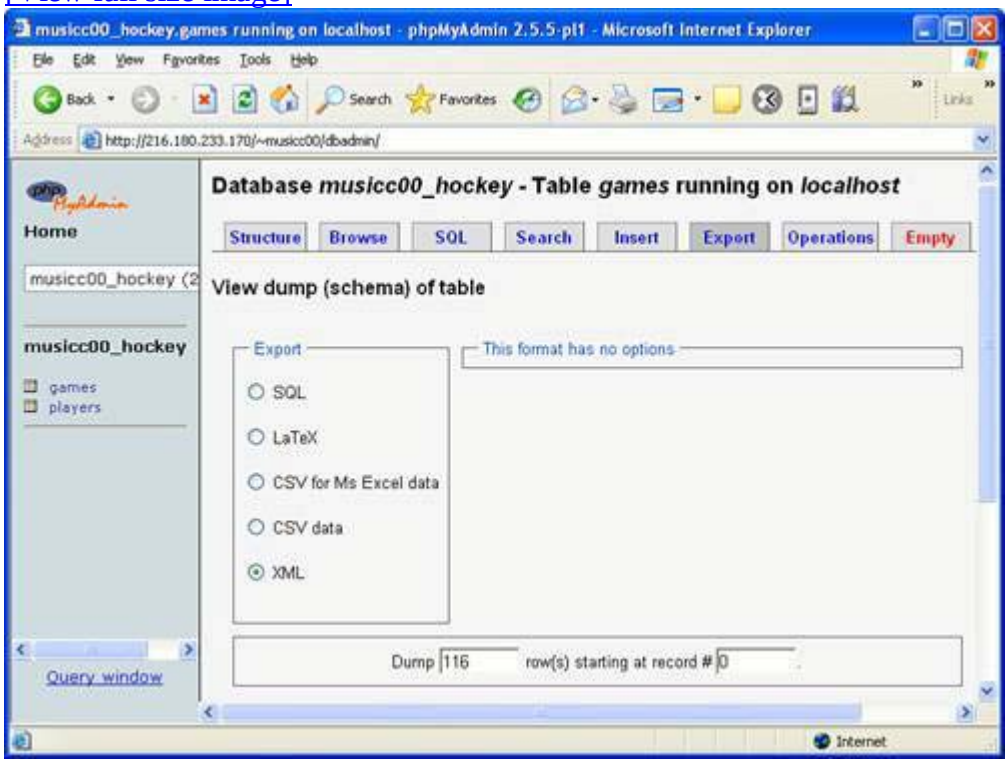
By the Way

If you're interested in using MySQL and phpMyAdmin, please visit http://www.mysql.com/ and http://www.phpmyadmin.net/. The details of installing and configuring a MySQL database are unfortunately beyond the scope of this lesson but you'll find plenty of information at the previously mentioned web sites.

To get started exporting an XML document from a MySQL database, open the database in phpMyAdmin, and select the table you want to export. Then click the Export tab. Within the Export options, click XML to indicate that XML is the output data format. If you want to generate an XML file that is stored on the web server, click the Save As File option. Otherwise, just click the Go button to generate the XML code and view it directly in the browser. Figure 19.3 shows how the exported XML document is shown in a web browser.

Figure 19.3. The newly exported XML document is immediately opened in the web browser.

[View full size image]



Now you can choose to save the XML file locally or otherwise use the XML code for further processing and manipulation. The key point to realize is that with one button click you've converted an entire tabular database into a well-formed XML document.

Accessing Data from a Database as XML

Although manually exporting an XML document from a database can be useful, it isn't quite the same as drilling into a database via a SQL query and extracting exactly the data you need. A more realistic example would involve generating XML code on the fly based upon a SQL query. Fortunately, I have exactly such an example for you to check out.

The example you're about to see extracts data from a real database that I created to manage the statistics for my recreational hockey team, Music City Mafia. The database is a MySQL database that stores statistics for both games and players. In this example, you're only concerned with game data, which is stored in a database table called games. To access the data and initiate a SQL query, I'm using PHP, which is an open source scripting language used to create dynamic web pages. PHP has very good integration with MySQL, and is a great option for dynamic web page development that involves MySQL databases and XML.

By the Way

PHP is a recursive acronym that stands for PHP Hypertext Processer. To learn more about PHP, visit the official PHP web site at http://www.php.net/.

Although the code you're about to see is written in PHP, you don't have to understand the PHP language in order to get the gist of what's going on. The key things to pay attention to are the SQL query being made on the database and the generation of the XML code. PHP is used to carry out these tasks but the code isn't too terribly difficult to decipher.

Listing 19.1 contains the code for the `mcm_schedule.php` sample web page that uses PHP to dynamically generate an XML file based upon a MySQL database query.

Listing 19.1. The Hockey Game Schedule PHP Example Document

```
 1: <?php
 2: // Connect to the database
 3: $mcm_db = mysql_connect("localhost", "admin", "password");
 4: mysql_select_db("mcm_hockey", $mcm_db);
 5:
 6: // Issue the query
 7: $mcm_query = sprintf("SELECT date, time, opponent, location, type,
outcome,
 8:   gf, ga, overtime FROM games WHERE season=\"%s\" ORDER BY
 9:   date", $season);
10: $mcm_result = mysql_query($mcm_query, $mcm_db);
11:
```

```
12: // Format the query results as XML
13: if (mysql_num_rows($mcm_result) > 0) {
14:   // Assemble the XML code
15:   $xml ="<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\r\n";
16:   $xml.="<games>\r\n";
17:   while (list($date, $time, $opponent, $location, $type, $outcome,
18:     $gf, $ga, $overtime) = mysql_fetch_array($mcm_result)) {
19:     $formatted_date = date("F j, Y", strtotime($date));
20:     $formatted_time = date("g:ia", strtotime($time));
21:     $xml.= sprintf(" <game date=\"%s\" time=\"%s\">\r\n",
22:       $formatted_date, $formatted_time);
23:     $xml.= sprintf(" <opponent>%s</opponent>\r\n", $opponent);
24:     $xml.= sprintf(" <location>%s</location>\r\n", $location);
25:     $xml.= sprintf(" <score outcome=\"%s\" overtime=\"%s\">
26:       %s - %s</score>\r\n", $outcome, $overtime, $gf, $ga);
27:     $xml.= " </game>\r\n";
28:   }
29:   $xml.="</games>";
30:
31:   // Write the XML code to the file mcm_results.xml
32:   $file= fopen("mcm_results.xml", "w");
33:   fwrite($file, $xml);
34:   fclose($file);
35:
36:   echo "The XML document has been written - <a href=\"mcm_results.xml\">
37:     view the XML code.</a>";
38: } else {
39:   echo "Sorry, no matching records found.";
40: }
41: // Close the database
42: mysql_close($mcm_db);
43:?>
```

The first few lines of the page establish a database connection and open the Music City Mafia hockey database. A SQL query is then constructed based upon a parameter (`$season`) that is passed into the page via the URL. The point of this parameter is to allow you to limit the XML file to a particular season of data. For example, to generate an XML file with only the game data for the 2005 Summer hockey season, the following URL is used: http://www.musiccitymafia.com/mcm_schedule.php?season=Summer%202005.

The `%20` near the end of URL is just a separator to provide a space between the word `Summer` and the word `2005`. The result of this URL is that the `mcm_schedule.php` web page assigns the value `Summer 2005` to the variable `$season`, which can then be used throughout the PHP code. And, in fact, it is when the SQL query is issued in lines 7 through 9 of the listing. More specifically, the date, time, opponent, location, type, outcome, goals for, goals against, and overtime database fields are selected from the games table but only for the Summer 2005 season. The result of this query is stored in the `$mcm_result` variable (line 10).

By the Way

In PHP programming, all variable names are preceded by a dollar sign (`$`).

The next big chunk of code goes through the results of the SQL query one record at a time, formatting the data into XML code. Notice that the XML processor directive is first generated (line 15), followed by a root tag, `<games>` (line 16). Each piece of pertinent game data is then further formatted into XML code in lines 17 through 28. The document is wrapped up with a closing `</games>` tag in line 29.

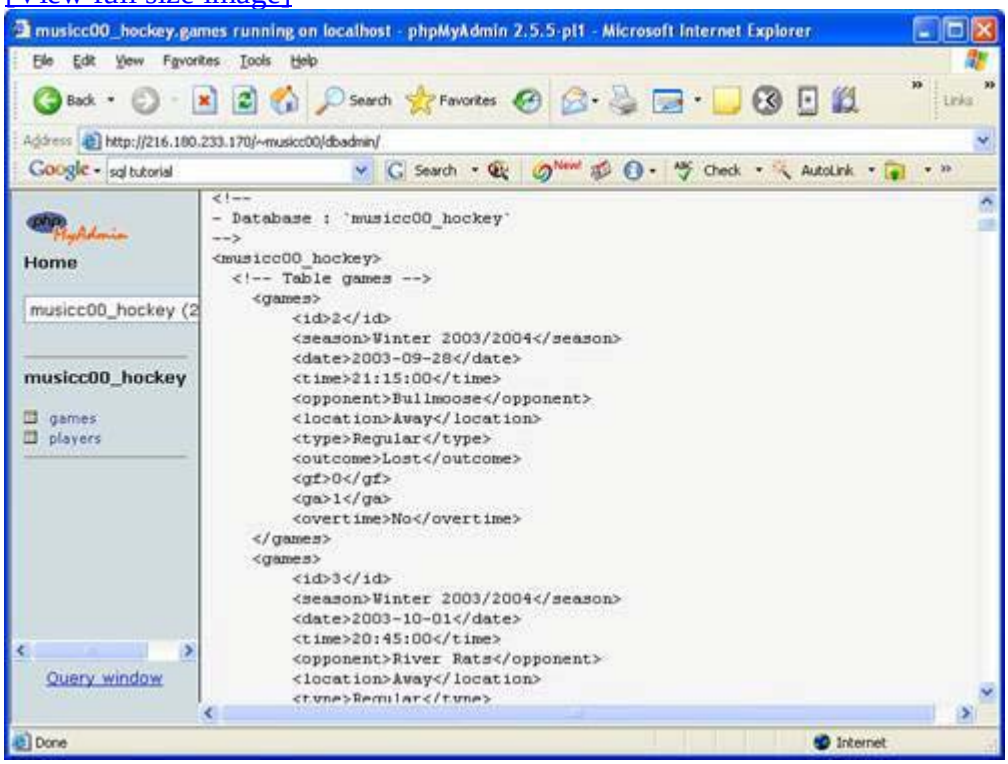
The last important step in the PHP code is writing the XML data to a file. The file is named `mcm_results.xml`, and the XML data is written to it with just a few lines of code (lines 32 to 34). A simple line of HTML code is then written to the browser so that you can access the XML file. More specifically, a link is generated that allows you to click and view the XML document (lines 36 and 37).

The remaining code in the PHP web page prints an error message if no records were found for the database query (line 39), and then closes the database (line 42).

Figure 19.4 shows the finished PHP document as viewed in Internet Explorer.

Figure 19.4. The hockey game schedule PHP document generates an XML file from a SQL database query, and then provides a link to the file.

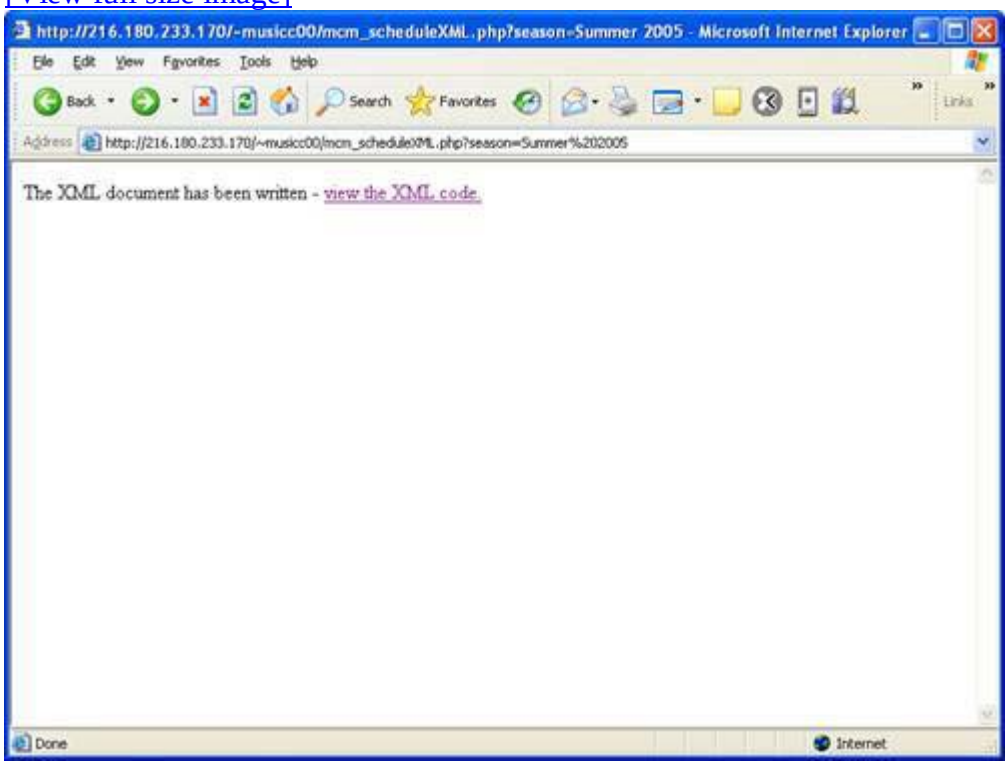
[View full size image]



Notice in the URL in the figure that the Summer 2005 season was specified, which limits the database query results to only those games in the Summer 2005 season. If you click the link on the page, the XML file is opened in the browser, as shown in Figure 19.5.

Figure 19.5. The resulting XML document contains cleanly formatted XML code that was dynamically generated from a database query.

[View full size image]



This figure reveals how the dynamically generated XML document contains structured data that originated from a purely tabular database. You can now run with this XML code and manipulate it just as you would any other XML document. You can transform it using XSLT, style it using CSS or XSL-FO, or automatically process it using some kind of specialized XML tool.

By the Way

Although I've focused on the open source MySQL database throughout this lesson, many commercial databases also include support for XML. Additionally, there are native XML databases, also called NXDs, that allow you to work entirely in XML code with database queries always resulting in pure XML code. To learn more about XML database products, visit http://www.rpbourret.com/xml/XMLDatabaseProds.htm.

Summary

The purpose of this tutorial was to introduce the concepts behind relational databases and explain how to integrate them with your XML applications. First, I provided a brief introduction to the theory behind relational databases, which is important to understand when you're mixing them with a different sort of data structure, like the one provided by XML. I then provided a very brief overview of SQL, which is the language used to query all relational databases. I then described the issues that arise when you're integrating relational databases with XML applications, and explained how using XML for data storage and XML as a document format differ. Finally, I demonstrated how to export XML data from an existing database, as well as how to perform a SQL query on a database and format the results as an XML document.

Q&A

Q. Don't some databases provide features specifically for handling XML?

A. Most major relational databases (like Oracle, Microsoft SQL Server, and IBM's DB2) support XML natively. The problem is that none of them support XML in the same way. If you use a database that provides XML support, you should look at the vendor documentation and decide whether you want to use it. The specific XML support in each of those databases is unfortunately beyond the scope of this lesson.

Q. What about object-oriented databases? Aren't they better suited to use with XML?

A. Object-oriented databases are more appropriate for storing XML data than relational databases, generally speaking. They're designed explicitly to handle the treelike data structures associated with object-oriented design. Treelike is also the best way to describe most XML data structures. However, object-oriented databases have not displaced relational databases on the market and are not standardized in the same manner as relational databases.